MATERIALS AND METHODS

This chapter elaborates on the various hardware / software tools used, experimental procedures followed, data beds used and a thorough explanation of the newly outlined algorithms for hiding sensitive data within images.

3.1 Equipment/ Tools Used

The equipment used to carry out this investigation were Nikon Coolpix P600 digital camera, HP laptop and a Pentium IV, 1.7 GHz desktop machine.

3.1.1 Camera Specifications

3.1.1.1 Overview:

The Nikon Coolpix P600 is a 16 mega pixel advanced point and shoot camera from Nikon India private limited. It has a lens which supports good zoom abilities. This camera also has vibration reduction features for better quality photography. It comes with a practical display screen which makes it easy to find the best angle for photos. The camera has the picture control options to help set preferences according to the subject and the situation.

3.1.1.2 Lens

The camera has a 60x optical zoom NIKKOR lens that delivers sharp telephoto shots, and a 120x Dynamic Fine Zoom which aids to capture minute details. The lens offers 60x optical zoom from 24 mm wide-angle to 1440 mm telephoto, and using Dynamic Fine Zoom provides amazing 120x zoom capability without compromising the high resolution.
3.1.1.3 Vibration Reduction

An enhanced lens-shift Vibration Reduction delivers unparalleled sharpness even at full zoom. This gives the flexibility of a slower shutter speed even at 60x zoom. This camera is equipped with ACTIVE mode to further minimize camera shake, when shooting from a moving vehicle or while walking and thus ensuring blur-free images.

3.1.1.4 Sensor Technology

The Nikon Coolpix P600 camera employs a CMOS image sensor. The CMOS image sensor used, has backside illumination which delivers high-quality images even when it is dark. The Nikon lens uses Super ED glass for a substantial improvement in chromatic aberration reduction.

3.1.1.5 Focus Technology

The Nikon Coolpix P600 has a fast auto focus and shutter response. This allows clicking photographs immediately when an opportunity arises. Rapid action can be clearly captured using Continuous H (in maximum image size at 7 fps, up to 7 frames) mode.

3.1.1.6 Picture Processing Technology

Nikon's exclusive Active D-Lighting function, offers the ability to preserve details in the highlights and shadowy areas of images, shot in high-contrast scenes such as those that are backlit. The camera enables customization prior to shooting by adjusting vividness of colors, contrast, and image sharpness depending on the type of subject, scene or intent. The technical specifications of the camera are as shown in table 3.1.
3.1.2 Desktop Configurations

The desktop system used was an OEM system made by Samsung. The system had a Pentium IV processor with a 22 inch LED monitor for better color experience. The system details are specified in table 3.2.

<table>
<thead>
<tr>
<th>Table 3.1 Technical Specifications of Nikon P600 digital camera</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Number of effective pixels</strong></td>
</tr>
<tr>
<td><strong>Image sensor</strong></td>
</tr>
<tr>
<td><strong>Lens</strong></td>
</tr>
<tr>
<td><strong>Focal length</strong></td>
</tr>
<tr>
<td><strong>f/-number</strong></td>
</tr>
<tr>
<td><strong>Construction</strong></td>
</tr>
<tr>
<td><strong>Digital zoom magnification</strong></td>
</tr>
<tr>
<td><strong>Vibration reduction</strong></td>
</tr>
<tr>
<td><strong>Motion blur reduction</strong></td>
</tr>
<tr>
<td><strong>Autofocus (AF)</strong></td>
</tr>
<tr>
<td><strong>Focus range</strong></td>
</tr>
<tr>
<td><strong>Focus-area selection</strong></td>
</tr>
<tr>
<td>Shooting Modes</td>
</tr>
<tr>
<td>----------------------------------------</td>
</tr>
<tr>
<td>Continuous Shooting</td>
</tr>
<tr>
<td>ISO sensitivity</td>
</tr>
<tr>
<td>Exposure Metering mode</td>
</tr>
<tr>
<td>Exposure control</td>
</tr>
<tr>
<td>Shutter</td>
</tr>
<tr>
<td>Speed</td>
</tr>
<tr>
<td>Aperture</td>
</tr>
<tr>
<td>Range</td>
</tr>
</tbody>
</table>
Table 3.2 Technical Configurations of Desktop

<table>
<thead>
<tr>
<th>Microprocessor</th>
<th>Intel Pentium IV 1.7 GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>1 GB DDR RAM</td>
</tr>
<tr>
<td>Video</td>
<td>NVDIA GeForce AGP card</td>
</tr>
<tr>
<td>Video Memory</td>
<td>256 MB</td>
</tr>
<tr>
<td>Hard Disk</td>
<td>160 GB IDE</td>
</tr>
<tr>
<td>Multimedia drive</td>
<td>56x multi read CD-ROM drive</td>
</tr>
<tr>
<td>Operating System</td>
<td>Windows XP 32 bit with Service pack 3</td>
</tr>
<tr>
<td>Chipset</td>
<td>VIA IDE chipset</td>
</tr>
<tr>
<td>Keyboard</td>
<td>Samsung PS2 multimedia 104 key keyboard</td>
</tr>
<tr>
<td>Mouse</td>
<td>PS 2 compatible mouse</td>
</tr>
<tr>
<td>LAN</td>
<td>VIA Rhine 10/100 fast Ethernet adaptor</td>
</tr>
<tr>
<td>Other Devices</td>
<td>• Two USB 2.0 ports</td>
</tr>
<tr>
<td></td>
<td>• VIA audio chipset</td>
</tr>
<tr>
<td></td>
<td>• VGA port</td>
</tr>
</tbody>
</table>

3.1.3 Laptop Configuration

The laptop used was HP dv7-6178us, manufactured by Hewlett-Packard. The laptop had an I7 processor with a 17.3 inch LED monitor with ATI Radeon graphics for optimum color production and system performance. The laptop configuration details are listed in table 3.3.

Table 3.3 Specifications of HP laptop

<table>
<thead>
<tr>
<th>Product Name</th>
<th>dv7-6178us</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microprocessor</td>
<td>2nd generation Intel Core i7-2630QM Processor 2.00GHz with Turbo Boost Technology up to 2.90 GHz</td>
</tr>
<tr>
<td>Memory</td>
<td>6GB DDR3 System Memory</td>
</tr>
<tr>
<td>Video Graphics</td>
<td>Radeon HD 6490M s</td>
</tr>
<tr>
<td>Hard Drive</td>
<td>750GB (5400RPM)</td>
</tr>
<tr>
<td>Multimedia Drive</td>
<td>SuperMulti DVD Burner</td>
</tr>
<tr>
<td>Display</td>
<td>17.3” High Definition+ HP BrightView LED Display (1600 x 900)</td>
</tr>
<tr>
<td>Network Card</td>
<td>Integrated 10/100/1000 Gigabit Ethernet LAN</td>
</tr>
<tr>
<td>Wireless Connectivity</td>
<td>• 802.11b/g/n WLAN with WiDi (36)</td>
</tr>
<tr>
<td>Sound</td>
<td>• Beats Audio with the HP Triple Bass Reflex Subwoofer</td>
</tr>
<tr>
<td>Keyboard</td>
<td>Full-Size Island-Style Keyboard with Integrated Numeric Keypad</td>
</tr>
<tr>
<td>Pointing Device</td>
<td>Touchpad supporting Multi-Touch gestures. With LED border accent light and On/Off button.</td>
</tr>
<tr>
<td>External Ports</td>
<td>• Digital Media Card Reader for Secure Digital and Multimedia cards</td>
</tr>
<tr>
<td></td>
<td>• 2 SuperSpeed USB 3.0</td>
</tr>
<tr>
<td></td>
<td>• 2 Universal Serial Bus (USB) 2.0</td>
</tr>
<tr>
<td></td>
<td>• 1 HDMI</td>
</tr>
<tr>
<td></td>
<td>• 1 VGA (15-pin)</td>
</tr>
<tr>
<td></td>
<td>• 1 RJ -45 (LAN)</td>
</tr>
<tr>
<td></td>
<td>• 2 Headphone-out</td>
</tr>
<tr>
<td></td>
<td>• 1 Microphone-in</td>
</tr>
<tr>
<td>Other Devices</td>
<td>• HP TrueVision HD Webcam with integrated digital microphone</td>
</tr>
<tr>
<td></td>
<td>• HP SimplePass with integrated fingerprint reader</td>
</tr>
</tbody>
</table>
3.2 Development Environment

3.2.1 Processor Requirement
Since the application deals with a lot of complex computational requirements, it demands a processor with good computational abilities. Hence a minimum of a Pentium IV with a 1.7 GHz processor would be required.

3.2.2 Disk space Requirement
This application deals with a lot of high resolution images which in the uncompressed form would require a lot of space. A minimum requirement of 500 MB would be a very reasonable amount of disk space required.

3.2.3 Memory Requirement
This Steganography application involves a lot of computations while processing the images as well as processing the data. A requirement of 1 GB would be a good minimum requirement.

3.3 Softwares/ Platform Used

3.3.1. Microsoft Visual Studio
Visual Studio is a complete set of development tools for building ASP.NET Web applications, XML Web Services, desktop applications, and mobile applications. Visual Basic, Visual C#, and Visual C++ , all use the same integrated development environment (IDE) from Microsoft, which enables, tool sharing and eases the creation of mixed-language solutions. In addition, these languages use the functionality of the .NET Framework, which provides access to key technologies that simplify the development of ASP Web applications and XML Web Services. Support for other languages such as
Python and Ruby, among others is available via language services installed separately. It also supports XML, HTML/XHTML, JavaScript and CSS.

3.3.2 Matlab 7.0

Matlab (Matrix Laboratory) is a high-performance technical computing language and interactive environment, used for algorithm development, data visualization, data analysis and numeric computation. It provides tools to acquire, analyse, and visualize data, in a very easy to use friendly manner. MATLAB features a family of application-specific solutions called toolboxes. Toolboxes are comprehensive collections of MATLAB functions (M-files) that extend the MATLAB environment to solve particular classes of problems. Areas in which toolboxes are available include signal processing, control systems, neural networks, fuzzy logic, wavelets, simulation, and many others.

Key Features

- High-level language for numerical computation, visualization, and application development.
- Interactive environment for modelling, design, simulation and prototyping.
- Easy to use mathematical functions for linear algebra, statistics, Fourier analysis, filtering, optimization, numerical integration, and solving differential equations.
- Built-in graphics and functions for data analysis, exploration, and visualization
- Tools for building applications with graphical user interfaces
- MATLAB based algorithms can be easily integrated with external applications and languages such as C, Java, .NET, and Microsoft Excel.
Matlab’s functionality can be greatly expanded by the addition of toolboxes.

There are also some limitations:

- It uses an enormous amount of memory and on slow computers it is very hard to use.
- It sits “on top” of Windows, getting as much CPU time as Windows allows it to have. This makes real-time applications very complicated.

3.3.3 OpenCV

**OpenCV** (*Open Source Computer Vision*) is a library of programming functions aimed at real-time computer vision, developed by Intel Russia research centre. It is free for use under the open source BSD license. It is a cross-platform library that focuses mainly on real-time image processing. OpenCV 2 includes major changes to the C++ interface, aiming at easier, more type-safe patterns, new functions, and better implementations for existing ones in terms of performance (especially on multi-core systems). OpenCV is written in C++ and its primary interface is in C++, but it still retains a less comprehensive, though extensive older C interface. It also supports full interfaces in Python, Java and MATLAB.

**Key features:**

- Open source computer vision library in C/C++.
- Optimized and intended for real-time applications.
- OS/hardware/window-manager independent.
- Generic image/video loading, saving, and acquisition.
- Has both, low and high level Application Programming Interfaces (API).
- Basic image processing (filtering, edge detection, corner detection, sampling and interpolation, color conversion, morphological operations, histograms, image pyramids).
- Various dynamic data structures (lists, queues, sets, trees, graphs).

### 3.3.4 CxImage library

CxImage is a C++ class, to manage almost any kind of images. It was written by David e Pizzolato. It can load, save, display and transform images in a very simple and fast way. CxImage is open source and licensed under the zlib license. With more than 200 functions, and with comprehensive working demos, CxImage offers all the tools to build simple image processing applications on a fast learning curve. Supported file formats are: BMP, GIF, JPG, JP2, PNG, TIF, RAW and many more.

CxImage is highly portable and has been tested with Visual C++ 6 / 2008, C++ Builder 3 / 6, MinGW on Windows, and with gcc 3.3.2 on Linux. The library can be linked statically, or through a DLL or an activeX component.

There are so many outstanding graphics libraries, such as OpenIL, FreeImage, PaintLib, however they are all license agreement bound. The glue to connect all the modules and the C libraries is CxFile; a virtual class that provides the standard methods to access the data from a file on the disk or in memory.
A CxImage object is basically a bitmap, with the addition of some member variables to store useful information:

```cpp
class CxImage
{
...
protected:
void* pDib;  //contains the header, the palette, the pixels
BITMAPINFOHEADER head;  //standard header
CXIMAGEINFO info;  //extended information
BYTE* pSelection;  //selected region
BYTE* pAlpha;  //alpha channel
CxImage** pLayers;  //generic layers
}
```

A CxImage object is also a set of layers. The buffers in each layer are allocated only when necessary. CxImage::pDib is the background image. CxImage::pAlpha is the transparency layer. CxImage::pSelection is the selection layer, used to create regions of interest for image processing.

Over these 3 specific planes, other generic layers can be added, stored in CxImage::pLayers. The generic layers are full CxImage objects, so complex structures of
nested layers can be built. The whole library is quite big, in the configuration header file \texttt{ximacfg.h}; switches to enable or disable a specific graphic format or feature present. Each JPG, PNG and TIFF library adds about 100KB to the final application, while the CxImage impact is about 50KB. So it is important to support and link only the formats that the application really needs.

The \texttt{CxImgLib.dsw} workspace shows the libraries required to build an application (\texttt{demo.exe}), including almost all the features and the formats available in CxImage. All the libraries must be compiled before you can link the final application. CxImage has a good support for memory files, new methods and file formats, and it is more portable.

**Key features**

- Supports various image formats
- Supports various types of filters and thresholding operations.
- Supports various Colorspaces: RGB, HSL, CMYK, YUV, YIQ.

The various file format dependencies are tabulated in table 3.4.
### Table 3.4 File format dependencies in CxImage

<table>
<thead>
<tr>
<th>Formats</th>
<th>#define</th>
<th>required libraries</th>
<th>size [Kbyte]</th>
</tr>
</thead>
<tbody>
<tr>
<td>BMP</td>
<td>CXIMAGE_SUPPORT_BMP</td>
<td>built in</td>
<td>24</td>
</tr>
<tr>
<td>GIF</td>
<td>CXIMAGE_SUPPORT_GIF</td>
<td>jpeg</td>
<td>88</td>
</tr>
<tr>
<td>ICO</td>
<td>CXIMAGE_SUPPORT_ICO</td>
<td>png, zlib</td>
<td>104</td>
</tr>
<tr>
<td>TGA</td>
<td>CXIMAGE_SUPPORT_TGA</td>
<td>mng, zlib, jpeg</td>
<td>148</td>
</tr>
<tr>
<td>PCX</td>
<td>CXIMAGE_SUPPORT_PCX</td>
<td>tiff, zlib, jpeg</td>
<td>124</td>
</tr>
<tr>
<td>WBMP</td>
<td>CXIMAGE_SUPPORT_WBMP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WMF</td>
<td>CXIMAGE_SUPPORT_WMF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JPEG</td>
<td>CXIMAGE_SUPPORT_JPEG</td>
<td>jpeg</td>
<td>88</td>
</tr>
<tr>
<td>PNG</td>
<td>CXIMAGE_SUPPORT_PNG</td>
<td>png, zlib</td>
<td>104</td>
</tr>
<tr>
<td>MNG</td>
<td>CXIMAGE_SUPPORT_MNG</td>
<td>mng, zlib, jpeg</td>
<td>148</td>
</tr>
<tr>
<td>TIFF</td>
<td>CXIMAGE_SUPPORT_TIF</td>
<td>tiff, zlib, jpeg</td>
<td>124</td>
</tr>
<tr>
<td>JBIG</td>
<td>CXIMAGE_SUPPORT_JBG</td>
<td>jbig</td>
<td>28</td>
</tr>
<tr>
<td>PNM,PPM,PGM RAS</td>
<td>CXIMAGE_SUPPORT_PNM CXIMAGE_SUPPORT_RAS</td>
<td>jasper</td>
<td>176</td>
</tr>
<tr>
<td>JPEG-2000</td>
<td>CXIMAGE_SUPPORT_JP2 CXIMAGE_SUPPORT_JPC CXIMAGE_SUPPORT_PGX</td>
<td>jasper</td>
<td>176</td>
</tr>
</tbody>
</table>

### 3.3.5 VSL

Virtual Steganographic laboratory (VSL) is a free image steganography and steganalysis software, in the form of a graphical block diagramming tool. It allows testing and adjusting different steganographic techniques and provides simple GUI along with modular, plug-in architecture, available under the GNU General Public License. It is written in Java, so it is cross-platform software and it can be executed on any operating system and is written and maintained by Michal Węgrzyn. Goal of the application, is hiding data in digital images, detecting its presence and testing its robustness using any number of different adjustable techniques. The pre-requisite to run VSL is Java 1.5 (5.0)
or above. To run an experiment, modules have to be arranged in processing flows by connecting them. All flows must begin with an Input module. Loop functionality can be obtained by connecting modules. There can be more than one flow at a time. VSL can write and read images that are covered by Java. JAI (Java Advanced Imaging API) can be installed in order to be able to work with more image readers and writers. To create new modules one of the interfaces from vsl-commons library must be implemented. For example; if new steganographic method is to be created, it must implement Steganographic Technique interface. When new module jar is created, two files must be edited: etc\modules.xml and MANIFEST.MF, inside vsl-app jar. VSL is available at http://vsl.sourceforge.net.

Data can be hidden with basic Least Significant Bit (LSB) method, with more advanced Karhunen-Loeve Transform (KLT) technique or by F5 algorithm, which uses DCT transformation in JPEG files. For steganalysis two advanced techniques can be used. First, RS-Analysis: efficient steganalysis for LSB methods - and the second one - Binary Similarity Measures (BSM) method with Support Vector Machines (SVMs) classifier: blind steganalysis (universal) technique, which can be used to detect any kind of steganography. VSL contains also many other modules - several distortion techniques, which can be used to test resistance of steganographic technique. Program has built-in modules, which helps with research, reports, file handling, image analysis etc.

3.3.6 Simple Steganalysis Suite

Simple Steganalysis suite (SSS) is a simple java based tool, to perform image steganalysis. It has an implementation of the most famous visual attacks, described by Westfeld and Pfitzmann in their paper titled “Attacks on steganographic systems” (Westfeld and Pfitzmann, 1999).
SSS current features include:

- LSB Enhancement
- Chi-Square test
- Neighborhood histogram
- Pixel Difference histogram
- Difference Histogram Attack
- Primary Sets

3.4 Theoretical Concepts

3.4.1 Visual Perception

Visual perception, is the ability to interpret the surrounding environment by processing information that is contained in visible light using the sense organs. The resulting perception is also known as eyesight, sight, or vision. Each sense organ is part of a sensory system, which receives sensory inputs and transmits sensory information to the brain.

A sketch of the anatomical components of the human eye is shown in Figure 3.2. The main components are the iris, lens, pupil, cornea, retina, vitreous humor, optic disk and optic nerve.
The retina is made up of two types of cells namely rod cells and cone cells. These cells were named because of their shape as viewed in the microscope and are highly responsive to light. Max Schultze (1825-1874) discovered that the retinal cones are the color receptors of the eye, and the retinal rod cells, while not sensitive to color, are very sensitive to light at low levels. It is the cone cells that allow us to see in color. It is because cone cells remain un-stimulated in low light environments that we do not see color in dimly lit places. In the human eye, there are many more rod cells in the retina, than there are cone cells. A schematic drawing of rod and cone cells is shown in Figure 3.3. The cells are divided into two sections. The bottom portion is called the inner segment. It contains the nucleus and the synaptic ending. The synaptic ending attaches to the neurons which produce signals that go to the brain. The top portion is called the outer segment. The outer segment is comprised of a membrane which is folded into several layers of disks. The disks are comprised of cells that contain the molecules that absorb...
the light. Pigments are also found in cone cells. There are three types of cone cells, each of which contains a visual pigment. These pigments are called the red, blue or green visual pigment. The cone cells detect the primary colors, and the brain mixes these colors in seemingly infinitely variable proportions, so that we can perceive a wide range of colors. The choice of primary colors is related to the physiology of the human eye. Good primary colors are stimuli that maximize the difference between the responses of the cone cells of the human retina, to light of different wavelengths, and that thereby makes a large color triangle.

The normal three kinds of light-sensitive photoreceptor cells in the human eye (cone cells) respond most to yellow (long wavelength or L), green (medium or M), and violet (short or S) light. The difference in the signals received from the three kinds allows the brain to differentiate a wide gamut of different colors, while being most sensitive (overall) to yellowish-green light and to differences between hues in the green-to-orange region.

As an example, suppose that light in the orange range of wavelengths enters the eye and strikes the retina. Light of these wavelengths would activate both the medium and long wavelength cones of the retina, but not equally; the long wavelength cells will respond more. The difference in the response can be detected by the brain, and this difference is the basis of our perception of orange. Thus, the orange appearance of an object results from light from the object entering our eye and stimulating the different cones simultaneously, but to different degrees.
3.4.2 RGB Colour model

The RGB color model is a convenient color model for computer graphics, because the human visual system works in a way that is very similar to an RGB color space. The RGB color model is an additive color model in which red, green, and blue light are added together in various ways to reproduce a broad array of colors.

To form a color with RGB, three colored light beams (one red, one green, and one blue) are superimposed (for example by emission from a black screen, or by reflection from a white screen). Each of the three beams is called a component of that color, and each of them can have an arbitrary intensity, from fully off to fully on, in the mixture. The RGB
color model is additive, in the sense that the three light beams are added together, and their light spectra, add wavelength for wavelength, to make the final color's spectrum.

Zero intensity for each component gives the darkest color (no light, considered the black), and full intensity of each gives a white; the quality of this white depends on the nature of the primary light sources, but if they are properly balanced, the result is a neutral white, matching the system's white point. When the intensities for all the components are the same, the result is a shade of gray, darker or lighter depending on the intensity. When the intensities are different, the result is a colorized hue, more or less saturated, depending on the difference of the strongest and weakest of the intensities of the primary colors employed.

When one of the components has the strongest intensity, the color has a hue value near this primary color (reddish, greenish, or bluish), and when two components have the same strongest intensity, then the color is a hue of a secondary color (a shade of cyan, magenta or yellow). A secondary color is formed by the sum of two primary colors, of equal intensity: cyan is green+blue, magenta is red+blue, and yellow is red+green. Every secondary color is the complement of one primary color; when a primary and its complementary secondary color are added together, the result is white: cyan complements red, magenta complements green, and yellow complements blue.

The RGB color model itself does not define what is meant by red, green, and blue colorimetrically, and so the results of mixing them are not specified as absolute, but relative to the primary colors. When the exact chromaticities of the red, green, and blue primaries are defined, the color model then becomes an absolute color space. The RGB model uses a Cartesian system of coordinates to map the available colors to the volume of a cube.
The Cartesian system representation of the RGB model has the following appearance:

![Cartesian system representation of the RGB model]

**Figure 3.4 Cartesian system representation of the RGB model**

The RGB color model has the following characteristics:

- The major axes of the cube are assigned to the three primary colors (red, blue, green) and their complements (magenta, yellow, and cyan).
- The origin of the coordinate system (0,0,0) is black. It corresponds to a total absence of color.
- The major diagonal of the cube extends from black (0,0,0) to white (255,255,255). This diagonal represents values that mix equivalent portions of the primary colors to produce gray scales.

### 3.5 Data beds Used

The proposed algorithms have been tested on 200 real time camera clicked digital images and on the UCID10K (Schaefer, 2010) and nevercompressed images databases having about 15150, 24 bit RGB images and yet another database having about 200, 24 bit PNG
images. The resolution of cover images varied from 256*256, 512*512 to 1024*1024 pixels. For the secret text data we used text files of varying sizes from 6 kb to 225 kb. For the secret images various grayscale and colored images were tested.

3.6 Proposed Methodology

3.6.1 Proposed Steganography Schemes

The proposed objective of this research was to develop a steganographic technique for covert communication over overt channels. A total of two techniques, in the spatial domain have been developed for concealing secret text as well as secret images in innocuous images. They have been named as Arithmetic division based data embedding (ADDE) and Bit Stream based data embedding (BSDE).

In a 24-bit bitmap, each pixel is represented by three bytes, representing the red, green and blue colour values for that pixel. The higher the number, the more intense is the colour for that pixel. In 24-bit bitmaps, the number of bytes per row is always end-padded with zeros to be a multiple of four. However using these extra bytes to hide data would be unwise, as these bytes are supposed to contain zeros and any alteration would be easily detectable. Thus, to ensure the image is inconspicuous, only the LSBs of the actual pixel data should be altered. Both of the proposed methods work on this principle and modify the LSB’s only. LSB embedding exploits the fact that the level of precision in many image formats is far greater than that perceivable by average human vision. Therefore, a modified image with slight variations in its colors, will be indistinguishable from the original by a human eye (Johnson and Jajodia, 1998). Embedding brings about distortion in the visual and statistical properties of an image. This in turn may aid in the identification of a stego image. The goal of any steganography technique is to safeguard these properties while carrying out data embedding (Islam et al., 2014). Edges in images
are areas with strong intensity contrasts, a jump in intensity from one pixel to the next. Thus the pixels in edges are found to be good candidates for data embedding, in comparison to the other smoother areas of the image. ADDE and BSDE use edges of a coloured image for the data hiding process.

Features of proposed algorithms

- The secret data is hidden inside the edges of the carrier image.
- Edges identified using Canny algorithm with modified thresholds.
- The data is hidden within a color pixel using the least two significant bits of all the three colour channels of a twenty four bit image.
- We use a combination of logical operations for hiding and also for retrieving the data. The main purpose of these operations is to replace the respective carrier image data bits with the secret data bits.
- We use bit masks to erase the respective bits which are not under consideration. This is done for both the carrier image as well as the secret data.
- Wavelets based image compression algorithm is used to reduce size of secret image data.
- Optimization criteria used to minimize bit storage.

3.6.1.1 Arithmetic Division based Data Embedding (ADDE)

Digital images are stored in computer systems as an array of points (pixels). The size of each pixel depends on the format of the image and normally ranges from 1 byte to 3 bytes. Each unique numerical pixel value corresponds to a color; thus, an 8-bit pixel is capable of displaying 256 different colors. A 24-bit color image has three color components corresponding to Red, Green and Blue. The three components are normally
quantized using 8 bits. Each pixel is represented with three bytes to indicate the intensity of these three colors (RGB). Each byte can have a value from 0 to 255, representing the intensity of the color. The darkest color value is 0 and the brightest is 255. Transparency is controlled by the addition of information to each element of the pixel data. A 24-bit pixel value can be stored in 32 bits. The extra 8 bits is for specifying transparency. This is known as the alpha channel. The Least Significant Bit insertion technique is a common, simple approach to embed information in a cover image. The LSB (the 8th bit) of each pixel, for a specific color channel or for all color channels is replaced with a bit of the secret message. For increasing the embedding capacity, two or more LSBS in each pixel can be used to embed messages. However, there is a trade-off between the embedding payload and the quality of the stego image (Cheddad et al., 2010). Embedding in the higher bitplanes, increases the embedding capacity, but adds a larger noise component to the value of each pixel, resulting in the distortion of an image. Embedding in the 4th LSB generates more visual distortion to the cover image as the hidden information is seen as “non-natural” (Cheddad et al., 2008).

The proposed technique works on 24 bit RGB images as cover images and has a data embedding capacity of maximum 6 bits per pixel. It applies a division operation and hides the secret data in the edges of a 24 bit cover image. In order to store the data onto the image, the data is considered to be a sequence of bytes.

In an effort to ensure imperceptibility and good stego image quality, we consider the least two significant bits of all the three color channels, of a 24 bit image to store the data; However since using only the least 2 significant bits could mean a very limited storage capacity, we employ an optimization technique to minimize the size of the data bytes while maximizing the data stored. The optimization technique used is a two stage
process. In the first stage, we reduce the magnitude of the data using an arithmetic division operation. Each byte of data is stored in the form of quotient and remainder. The entire array of data is then represented in the form of quotient and remainder arrays. The quotient can have a maximum size of 5 bits and the remainder can be of 3 bits. This is because we choose the divisor to be 8. The value 8 was chosen as the divisor since we could reduce the storage size of the quotient from an 8 bit value to a maximum 5 bit value. In the second stage, we further reduce the size by implementing an optimization logic. After the arithmetic division operation, sizes of all the bytes of our quotient data are analysed. Based on the sizes of the quotient values, we decide the range of the bits to be used for storage. The range would form the optimization criterion. The number of bits to be used to store the quotient data is specified by a flag array. The combination of the optimization criterion and the flag array is said to optimize and specify the number of bits to be used for data storage. Since the number of bits used to store the data varies, this technique will also be resistant to statistical attacks. The choice of the cover image can be of any file format supported by the CxImage library, such as Windows bitmap format or PNG or JPG file format. However the output stego file would be saved in either the BMP or PNG lossless data formats.

The details of the optimization criterion and the flag value combinations along with the bit sizes signified, are mentioned in table 3.5
Table 3.5 Mapping criterion used in ADDE

<table>
<thead>
<tr>
<th>Optimization Criterion</th>
<th>Flag Value</th>
<th>Bit Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>R G B</td>
<td>0 1 0</td>
<td>0 2</td>
</tr>
<tr>
<td></td>
<td>0 1 0</td>
<td>1 5</td>
</tr>
<tr>
<td></td>
<td>0 1 1</td>
<td>0 2</td>
</tr>
<tr>
<td></td>
<td>0 1 1</td>
<td>1 4</td>
</tr>
<tr>
<td></td>
<td>1 0 0</td>
<td>0 2</td>
</tr>
<tr>
<td></td>
<td>1 0 0</td>
<td>1 3</td>
</tr>
<tr>
<td></td>
<td>1 0 1</td>
<td>0 3</td>
</tr>
<tr>
<td></td>
<td>1 0 1</td>
<td>1 4</td>
</tr>
<tr>
<td></td>
<td>1 1 0</td>
<td>0 4</td>
</tr>
<tr>
<td></td>
<td>1 1 0</td>
<td>1 5</td>
</tr>
<tr>
<td></td>
<td>1 1 1</td>
<td>0 3</td>
</tr>
<tr>
<td></td>
<td>1 1 1</td>
<td>1 5</td>
</tr>
</tbody>
</table>

The ADDE method data storage schematic diagram is as shown in figure 3.5 below. The horizontal wavy line represents the identified 24-bit edge pixels. The first edge pixel always stores the optimization criteria (3 bits) as derived from table 3.5. The second edge pixel stores the length of length field, i.e. – the number of bits used to store the length field. From the third edge pixel onwards begins the length, followed by the Flag array, quotient array and finally the remainder array.
3.6.1.1 Using ADDE With Text

The block diagram of the arithmetic division based data embedding for embedding secret text within an image is as shown in figure 3.6. The proposed ADDE data embedding algorithm as explained in figure 3.7 would take Cover image (C), Secret text data (M) and the length of Secret data (L) as inputs.

The generated stego image is sent over an overt channel to the recipient. The recipient then runs the data extraction algorithm as shown in figure 3.8 to extract the secret data.
Figure 3.6 Flowchart of text embedding using ADDE technique
Algorithm: ADUE Text Embedding Algorithm

Inputs: C - Cover Image, M - Secret Text Data, L - Length Of Secret Data
Outputs: S - Stego Image

// Initialize the Stego image with the Cover image data
S ← C

// Initialize Bit Mask for 2 LSB and 1 LSB
LSBMASK ← 11111100
LSBIMASK ← 11111110

// Erase the 2 LSB of Cover image using Mask and Logical AND operation
C ← bitAND(C, LSBMASK)

// Determine Canny edge detection thresholds depending upon image characteristics
th, tl ← getThreshold(C)

// Use Canny Edge detection to create an edge map E
E ← CannyEdge(C, th, tl)

// Create two arrays Q and R of quotient and remainder
// from the secret data using divisor R
Q ← M/R
R ← mod(M, R)

// Scan array Q and generate a flag array F with a mapping
// for the number of bits required per byte of Q
for every element Q1 of Q and F1 of F do
    B ← GetNoOfBits(Q1)
    if B ≤ 4 then
        F1 ← 0
    else
        F1 ← 1
    end if
end for

// We store the Flag array into the stego image using the edge map
for every 3 elements F1, F2, F3 of F do
    P ← GetEdgePixel(S, E)
    RVALUE ← GetRedChannel(P)
    GVALUE ← GetGreenChannel(P)
    BVALUE ← GetBlueChannel(P)
    RENC ← bitAND(RVALUE, LSBMASK)
    GENC ← bitAND(GVALUE, LSBMASK)
    BENC ← bitAND(BVALUE, LSBMASK)
end for

// Embed the flag array data
RENC ← RENC*F1
GENC ← GENC*F2
BENC ← BENC*F3

Algorithm continued on next page.....
Algorithm ADDE Text Embedding Algorithm

```plaintext
// Replace the color values in the current pixel
ReplacePixelColor(P, RENC, GENC, BENC)

// Set the pixel P in stego image S
SetPixel(S, P)
end for

// We store the quotient array Q into the stego image S using the bit pattern specified in array F
for every element Q of Q do
P ← GetFlagElement(F)
P ← GetEdgePixel(S, P)
RVALUE ← GetRedChannel(P)
GVALUE ← GetGreenChannel(P)
BVALUE ← GetBlueChannel(P)
if F1 == 0 then
RENC ← bitAND(RVALUE, LSB1MASK)
GENC ← bitAND(GVALUE, LSB1MASK)
BENC ← bitAND(BVALUE, LSB1MASK)
RENC ← RENC[(Q1 >> 2) & LSB1MASK]
GENC ← GENC[(Q1 >> 1) & LSB1MASK]
BENC ← BENC[(Q1 & LSB1MASK)]
else
RENC ← bitAND(RVALUE, LSB2MASK)
GENC ← bitAND(GVALUE, LSB2MASK)
BENC ← bitAND(BVALUE, LSB2MASK)
RENC ← RENC[(Q1 >> 3) & LSB2MASK]
GENC ← GENC[(Q1 >> 1) & LSB2MASK]
BENC ← BENC[(Q1 & LSB2MASK)]
end if
// Replace the color values in the current pixel
ReplacePixelColor(P, RENC, GENC, BENC)
// Set the pixel P in stego image S
SetPixel(S, P)
end for

// We store the remainder array R into the stego image S
for every element R of R do
P ← GetFlagElement(F)
P ← GetEdgePixel(S, P)
RVALUE ← GetRedChannel(P)
GVALUE ← GetGreenChannel(P)
BVALUE ← GetBlueChannel(P)
RENC ← bitAND(RVALUE, LSB1MASK)
GENC ← bitAND(GVALUE, LSB1MASK)
BENC ← bitAND(BVALUE, LSB1MASK)
RENC ← RENC[(R1 >> 2) & LSB1MASK]
GENC ← GENC[(R1 >> 1) & LSB1MASK]
BENC ← BENC[(R1 & LSB1MASK)]
// Replace the color values in the current pixel
ReplacePixelColor(P, RENC, GENC, BENC)
// Set the pixel P in stego image S
SetPixel(S, P)
end for
```

Figure 3.7 ADDE text embedding algorithm
3.6.1.1.2 Using ADDE with Images

The arithmetic division based embedding algorithm was also implemented using an image as the secret data that had to be concealed within a cover image. The proposed technique first compresses the secret data (image) using a compression method in the wavelet domain, applies a division operation on the compressed data and then hides it, in the edges of the Image. Since images require large bandwidth, compression is useful to decrease the data storage capacity and thus reduce bandwidth. Here, secret image compression is achieved using the Antonini wavelet and arithmetic coding of the transformed coefficients. The flow chart is as shown in figure 3.9 and algorithm is as explained in figure 3.10. The secret image extraction algorithm is explained in figure 3.11.
Figure 3.9 Flow chart of Image Embedding in ADDE
Algorithm ADDE Image Embedding Algorithm:

Inputs: C - Cover Image, M - Secret Image Data, L - Length Of Secret Image Data
Outputs: S - Stego Image

// Initialize the Stego image with the Cover image data
S ← C

// Initialize Bit Mask for 2 LSB and 1 LSB
LSB2MASK ← 11111100
LSB1MASK ← 11111110

// Erase the 2 LSB of Cover image using Mask and Logical AND operation
C ← bitAND(C, LSB2MASK)

// Determine Canny edge detection thresholds depending upon image characteristics
th, tl ← getThreshold(C)

// Use Canny Edge detection to create an edge map E.
E ← CannyEdge(G, th, tl)

// Compress secret image using wavelet based compression algorithm
M ← Compress(M)

// Create two arrays Q and R of quotient and remainder
// from the secret data using divisor 8
Q ← M/8
R ← mod(M, 8)

// Scan array Q and generate a flag array F with a mapping
// for the number of bits required per byte of Q
for every element Ql of Q and Fl of F do
    B ← GetNoOfBits(Ql)
    if B < 4 then
        Fl ← 0
    else
        Fl ← 1
    end if
end for

// We store the Flag array into the stego image using the edge map
for every 3 elements F1, F2, F3 of F do
    P ← GetEdgePixel(S, E)
    RVALUE ← GetRedChannel(P)
    GVALUE ← GetGreenChannel(P)
    BVALUE ← GetBlueChannel(P)
    BENC ← bitAND(RVALUE, LSB1MASK)
    GENC ← bitAND(GVALUE, LSB1MASK)
    BENC ← bitAND(BVALUE, LSB1MASK)

    // Embed the flag array data
    RENC ← BENC|F1
    GENC ← GENC|F2
    BENC ← BENC|F3
Figure 3.10 ADDE image embedding algorithm
Consider the image skype.bmp to be hidden within LenaRGB.bmp. An edge map of LenaRGB.bmp would first be created using the proposed modified Canny edge detection algorithm. Skype.bmp would then be compressed using the wavelet compression algorithm. The binary data representation would then be divided to view the byte data in the form of quotient and remainder. The quotients are then scanned to determine their minimum and maximum representation bit size. Accordingly, the
optimization criterion is selected from the ADDE mapping table. In our example, the minimum bit size was seen to be 3 and the maximum bit size was found to be 5. Thus the optimization criteria to be stored in the first edge pixel was RGB =111. The second edge pixel holds the number of bits required to store the length field. The third edge pixel onwards stores the actual length. Following the flag array is being filled. The first flag value to be stored was 1 and so on. Every edge pixel stored the flag for 3 elements. Then the quotient array is being filled. Let’s consider the first quotient element to be 30. The encoding and decoding of a value into an edge pixel is as explained below:

Combinations of logical operations are used for hiding and also for retrieving the data. The main purpose of these operations is to replace the respective carrier image data bits with the secret data bits. Bit masks are used to erase the respective bits which are not under consideration. This is done for both the carrier image as well as the secret data. The bit masks used are LSB1MASK= 11111100 and LSB2MASK = 00000011.

The pixel data of a twenty four bit image will be in the form of a triplet value of Red, Green and Blue. Let us denote this as (RVALUE, GVALUE, BVALUE) . The final resultant pixel too will be in the form of a triplet value of Red, Green and Blue. Let us denote this as (RENC, GENC, BENC). Let us denote our data byte value to be stored as DATA.

Consider we have an initial data value of 240 to be hidden. We divide this value and represent it in terms of quotient and remainder. Considering 8 as the divisor, we obtain quotient = 30 and remainder = 0.

Now, consider we want to store the value of 30 (quotient) in a fully white pixel of quotient array.

For a white pixel, Pixel data: RVALUE = 255, GVALUE = 255, BVALUE =255
In binary: RVALUE = 11111111, GVALUE = 11111111, BVALUE = 11111111
DATA = (quotient value) = 30 = 011110.

Step I – Clear the last 2 LSB of an edge pixel (in quotient array)
In our example we are considering a white pixel in quotient array to be replaced:
RENC = RVALUE & LSB1MASK = 11111111 & 11111100 = 11111100
GENC = GVALUE & LSB1MASK = 11111111 & 11111100 = 11111100
BENC = BVALUE & LSB1MASK = 11111111 & 11111100 = 11111100

Step II – Encode the secret data (quotient value) into cleared value in quotient array.
RENC = RENC | ((DATA >> 4) & LSB2MASK)
    = 11111100 | ((011110 >> 4) & 00000011)
    = 11111101 = 253 (In decimal).

GENC = GENC | ((DATA >> 2) & LSB2MASK)
    = 11111100 | ((011110 >> 2) & 00000011)
    = 11111111 = 255 (In decimal).

BENC = BENC | (DATA & LSB2MASK)
    = 11111100 | (011110 & 00000011)
    = 11111110 = 254 (In decimal)
After encoding, we now have the secret data (quotient value) encoded as R = 253, G = 255, B = 254 in decimal values.

In order to decode the encoded pixel data we first extract the data bits.
Applying the formula:

\[
\begin{align*}
RDEC &= RENC \& \text{LSB2MASK} = 11111101 \& 00000011 = 00000001 \\
GDEC &= GENC \& \text{LSB2MASK} = 11111111 \& 00000011 = 00000011 \\
BDEC &= BENC \& \text{LSB2MASK} = 11111110 \& 00000011 = 00000010
\end{align*}
\]

In order to retrieve the final data (quotient value), we apply the operation:

\[
\text{DATA} = (\text{RDEC} \ll 4) | (\text{GDEC} \ll 2) | (\text{BDEC})
\]

\[
\text{DATA} = (00000001 \ll 4) | (00000011 \ll 2) | (00000010)
\]

\[
= 00011110
\]

\[
= 30 \text{ (In decimal)}.
\]

Thus the decoded data is the same as the encoded.

### 3.6.1.2 Bit Stream based Data Embedding (BSDE)

The first data embedding technique ADDE, utilized three arrays namely the quotient array, remainder array and the flag array in its implementation. After a lot of executions, it was noticed that a lot of storage space is required by these three arrays. The second technique BSDE uses another technique, in order to reduce the number of arrays and increase the secret data embedding capacity for a given cover image. In this bit stream based data embedding technique too, we consider a 24 bit bitmap image as the cover image. We use the 2 LSB’s of each color channel of the edges of the cover image to store the secret data in the form of a bit stream.

In the BSDE technique, we employ the 24 bit image data present within the edge pixels of the image. An edge map of the edges is created using the modified Canny edge detection technique. The last two least significant bits of all the three color channels of
red, green and blue of the edge pixels are used to store the data. Thus we achieve an embedding capacity of 6 bits per pixel.

In this method we use a flag array to give us the size of each byte of data. We use a stream of continuous bits to store the data. The flag array and the bit stream are both stored into the edge pixels of the carrier image using the last two LSB. A size mapping process is carried out for each byte of data. The bit stream is constructed by appending the data bytes one after another, after the size mapping operation is carried out. In order to optimize the data, the amount of bits used by each byte of data is mapped to a value set in the flag array. Table 3.6 mentions the size mapping used to optimize the storage.

<table>
<thead>
<tr>
<th>Flag Value</th>
<th>Size Used</th>
<th>Data Size Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>3 - 1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4 - 5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>6 - 8</td>
</tr>
</tbody>
</table>

With the help of the above table, the storage space is decided for each byte of data. We use this size to store the actual data. As can be seen from the table, if the data value is zero, no data is stored in the bit stream. The bit stream is stored as a continuous series of bits, using the last two LSB of the red, green and blue channels of the edge pixels of the
carrier image. When retrieving the original data, the flag array and also the bit stream are required. The flag array is used to obtain the number of bits of storage per byte of data. Accordingly the original data has to be reconstructed.

The BSDE method data storage schematic diagram is as shown in figure 3.12 below. The horizontal wavy line represents the identified 24-bit edge pixels. The first edge pixel always stores the length of length field, i.e. the number of bits used to store the length field. From the second edge pixel onwards begins the length of secret data, followed by the flag array stream of data. In the BSDE method 2-bit flags are used as derived from table 3.6, and thus each edge pixel can store the flags for 3 elements. For the bit stream array, the last 2 lsb’s of every channel colour of every edge pixel is used to store data. Thus every pixel stores 6 bits of secret data.

![Figure 3.12 BSDE method data embedding schematic diagram](image)

3.6.1.2.1 Using BSDE with Text

The flowchart for BSDE text embedding is as shown in figure 3.13. The algorithm for the same is written in figure 3.14.
Figure 3.13 Flow Chart for BSDE Text Embedding Technique
Algorithm: BSDE Text Embedding Algorithm

Inputs: C – Cover Image, M – Secret Text Data, L – Length Of Secret Data
Outputs: S – Stego Image

// Initialize the Stego image with the Cover image data
S ← C

// Initialize Bit Mask for 2 LSB
LSB2MASK ← 11111100

// Erase the 2 LSB of the Cover image using the Mask and Logical AND operation
C ← bitAND(C, LSB2MASK)

// Determine Canny edge detection thresholds depending upon image characteristics
th, tl ← getThreshold(C)

// Use Canny Edge detection to create an edge map E.
E ← CannyEdge(C, th, tl)

// Scan secret data M and generate a flag array F with the number of bits required per byte of M
for every byte M1 of M and every element F1 of F do
    N ← GetNoOfBits(M1)
    if N == 0 then
        F1 ← 0
    else if 0 ≤ N ≤ 8 then
        F1 ← 3
    else if 4 ≤ N ≤ 5 then
        F1 ← 2
    else if 3 ≤ N ≤ 1 then
        F1 ← 1
    end if
end for

// We store the flag array into the stego image using the edge map
for every 3 elements F1,F2,F3 of F do
    P ← GetEdgePixel(S,E)
    RVALUE ← GetRedChannel(P)
    GVALUE ← GetGreenChannel(P)
    BVALUE ← GetBlueChannel(P)
    RENC ← bitAND(RVALUE, LSB2MASK)
    GENC ← bitAND(GVALUE, LSB2MASK)
    BENC ← bitAND(BVALUE, LSB2MASK)

    // Embed the flag array data
    RENC ← RENC+F1
    GENC ← GENC+F2
    BENC ← BENC+F3

    // replace the color values in the current pixel
    ReplacePixelColor(P,RENC,GENC,BENC)
end for

// Set the pixel P in stego image S
SetPixel(S,P)
end for
Figure 3.14 BSDE Text Embedding Algorithm
The BSDE text extraction algorithm is explained in figure 3.15.
3.6.1.2.2 Using BSDE With Images

The bit stream based embedding algorithm was also implemented using an image as the secret data that had to be concealed within a cover image. This proposed technique first compresses the secret data (image) using a compression method in the wavelet domain. The compressed data is then transformed into a flag array and a stream of bits containing the data. The two arrays are then hidden in the edges of the Image. Since images require large bandwidth, compression is useful to decrease the secret data size and increase the data storage capacity and thus reduce bandwidth. Here, secret image compression is achieved using quantization and arithmetic coding of the transformed coefficients.

The steps are illustrated in the following flow chart, figure 3.16. Figure 3.17 describes the algorithm for secret image embedding using the BSDE technique. The reverse BSDE based secret image extraction process is explained in figure 3.18.
Figure 3.16 Flow Chart for BSDE Image embedding Technique
Algorithm BSDE Image Embedding Algorithm

Inputs: C – Cover Image, M – Secret Text Data, L – Length Of Secret Data
Outputs: S – Stego Image

// Initialize the Stego image with the Cover image data
S ← C

// Initialize Bit Mask for 2 LSB
LSB2MASK ← 11111100

// Erase the 2 LSB of the Cover image using the Mask and Logical AND operation
C ← bitAND(C, LSB2MASK)

// Determine Canny edge detection thresholds depending upon image characteristics
th, tl ← getThresholds(C)

// Use Canny Edge detection to create an edge map E.
E ← CannyEdge(C, th, tl)

// Compress secret image using wavelet based compression algorithm
M ← Compress(M)

// Scan secret data M and generate a flag array F with the number of bits required per byte of M
for every byte M[i] of M and every element F[i] of F do
  N ← GetNoOfBits(M[i])
  if N = 0 then
    F[i] ← 0
  else if 6 ≤ N ≤ 8 then
    F[i] ← 3
  else if 4 ≤ N ≤ 5 then
    F[i] ← 2
  else if 3 ≤ N ≤ 1 then
    F[i] ← 1
  end if
end for

// We store the Flag array into the stego image using the edge map
for every 3 elements F1,F2,F3 of F do
  P ← GetEdgePixel(S, E)
  RVALUE ← GetRedChannel(P)
  GVALUE ← GetGreenChannel(P)
  BVALUE ← GetBlueChannel(P)
  RENC ← bitAND(RVALUE, LSB2MASK)
  GENC ← bitAND(GVALUE, LSB2MASK)
  BENC ← bitAND(BVALUE, LSB2MASK)

  // Embed the flag array data
  RENC ← RENC|F1
  GENC ← GENC|F2
  BENC ← BENC|F3

  // replace the color values in the current pixel
  ReplacePixelColor(P, RENC, GENC, BENC)
end for

// Set the pixel P in stego image S
SetPixel(S, P)

Algorithm continued on next page...
Algorithm BSDE Image Embedding Algorithm

// Generate a bit stream B by concatenating all the bits M1 to Mn of M
B ← M1M2M3.....Mn

// We store the bit stream B into the stego image S
for every 6 bits B1 of B do
    P ← GetEdgePixie(S,E)
    RVVALUE ← GetRedChannel(P)
    GVVALUE ← GetGreenChannel(P)
    BVVALUE ← GetBlueChannel(P)
    RENC ← bitAND(RVVALUE, LSB2MASK)
    GENC ← bitAND(GVVALUE, LSB2MASK)
    BENC ← bitAND(BVVALUE, LSB2MASK)
    RENC ← RENC|((B1 >> 2)&LSB2MASK)
    GENC ← GENC|((B1 >> 1)&LSB2MASK)
    BENC ← BENC|(B1&LSB2MASK)

// Replace the color values in the current pixel
ReplacePixelColor(P,RENC,GENC,BENC)

// Set the pixel P in stego image S
SetPixel(S,P)
end for

Figure 3.17 BSDE Image Embedding Algorithm
Algorithm BSDE Image Extraction Algorithm

Inputs: S - Stego Image
_outputs: M - Secret Text Data

// Initialize Bit Mask for 2 LSB
_LSB2MASK_ ← 11111100

// Erase the 2 LSB of the Stego image using the Mask and Logical AND operation
_S_ ← _bitAND(S, LSB2MASK)

// Determine Canny edge detection thresholds depending upon image characteristics
_th, tl_ ← _getThreshold(S)_

// Use Canny Edge detection to determine edge map E.
_E_ ← _CannyEdge(S, th, tl)_

// Read the flag array F from edge pixels of stego image
_F_ ← _GetFlagArray(S)_

// set the correct length values in F
_for every element _F1_ of F do
  _if_ _F1_ == 0 then
    _F1_ ← 0
  _else if_ _F1_ == 1 then
    _F1_ ← 3
  _else if_ _F1_ == 2 then
    _F1_ ← 5
  _else if_ _F1_ == 3 then
    _F1_ ← 8
  _end if_
_for end

// Read the bit stream B from edge pixels of stego image.
_B_ ← _GetBitStream(S)_
_Count_ ← 0
_for every data byte _M1_ of M and element _F1_ of F do
  _M1_ ← 0
  _for_ _i_ = 0 to _F1_ do
    _M1_ ← _M1_ << 1
    _M1_ ← _M1_ | _B_(_Count_ + _i_)
  _end for_
  _Count_ ← _Count_ + _F1_
_for end

// Uncompress the bytes of data to get the image
_M_ ← _Uncompress(M)_

Figure 3.18 BSDE Image Extraction Algorithm
3.6.1.2.3 Mathematically understanding BSDE with an example

Consider the image skype.bmp to be hidden within LenaRGB.bmp using BSDE technique.

An edge map of LenaRGB.bmp would first be created using the modified canny edge detection algorithm. Skype.bmp would then be compressed using the wavelet compression algorithm. The binary data representation would then be subject to a mapping process. The size mapping process is carried out for each byte of data using the BSDE mapping table. In order to optimize the data, the amount of bits used by each byte of data is mapped to a value set in the flag array. The bit stream is constructed by appending the data bytes one after another, after the size mapping operation is carried out. The flag array and the bit stream are both stored into the edge pixels of the carrier image using the last two LSB of every edge pixel.

Consider for example the bytes of data obtained from skype.bmp are 0,7,30,100,……. Here the data length =4 and length of length = 3 (since 4 is represented using 3 bits).

In binary, the number of bits used to represent the data would be 0,3,5,7.

On carrying out size optimization using the BSDE optimization table, the data sizes that would be used are 0,3,5,8 and the corresponding flag arrays would be set to 00, 10, 01, 11.

This data now has to be stored on the edge map. The first edge pixel always stores the length of length field. In our example – 3. The second edge pixel always stores data length. Then the flag array is stored as a stream followed by the data bytes.

In our example the flag array would have values - 00, 10, 01, 11 and the data stream would have values 111111001100100…. The actual process to encode and decode a pixel value is same as explained for ADDE method.
3.6.2 Wavelet based Data Compression algorithm

The proposed algorithm uses the Antonini wavelet filter proposed by Antonini et al., (1992) and the Alistair Moffat’s linear time algorithm for adaptive arithmetic coding(Alistair Moffat, 1990; Moffat et al., 1998). It takes as inputs the image, and various other coding parameters like number of subbands, quantization step size, etc.

The basic steps for wavelet based image compression are as shown in figure 3.19 below:

![Figure 3.19 Wavelet based image compression](image)

The basic steps for wavelet based image de-compression are as shown in figure 3.20 below:

![Figure 3.20 Wavelet based image de-compression](image)
A wavelet decomposition of a function $f$ defined on $\mathbb{R}^d$ is an expression (Devore et al., 1991) of the form

$$f = \sum C_{j,k} \phi_{j,k}$$

(3.1)

where the coefficients $C_{j,k}$ depend on $f$ and the functions

$$\phi_{j,k}(x) := \phi(2^k(x-j/2^k))$$

are the dyadic dilates (by $2^k$) and translates (by $j/2^k$) of a single function $\phi$ called a wavelet.

The data compression problem in the wavelet domain is seen as one of approximating $f$ by a compressed function $f_1$. In a lossless compression algorithm, the error between the original and compressed functions will be zero.

The algorithm calculates the quantized coefficients $C_{1,j,k}$ and the compressed function takes the form

$$f_1 = \sum C_{1j,k} \phi_{j,k}$$

(3.2)

It is very much necessary to measure the error between $f$ and $f_1$, which is assumed to be defined on some interval $I$ in $\mathbb{R}^d$. We have used the $L^p(I)$ norms error metric with $0 < p \leq \infty$. 
This norm is defined by

\[ | f - f_1 |_{L^p(I)} = \left( \int | f(x) - f_1(x) |^p \, dx \right)^{1/p} \]  

(3.3)

A Discrete Wavelet transform refers to the multi-resolution decomposition of a signal. Low and high pass filters are used to extract the low frequency coefficients and the high frequency coefficients of a signal. Selected coefficients are then quantized and encoded that helps in the image compression. The proposed image compression algorithm based on the concept of wavelets is explained in figure 3.21.

![Wavelet compression Algorithm](image)

**Figure 3.21 Wavelet compression Algorithm**

3.6.3 Edge Detection Method

Edges are sudden variation in the grey level or colour of image pixels and edge detection is locating areas with strong intensity contrasts. The main principle of edge detection is
to extract the high frequency of image edge using first derivative, second derivative and high pass filtering. An important part in edge detection is the use of an appropriate filter, to handle the problems of noise and preserving the fine details of an image. Edge detection is of great importance in fields such as X-rays, computer vision, image analysis and security applications in image processing. Edge detection is a significant area of image processing and machine vision, due to the fact that edges are considered to be the important features for analysing the most essential information contained in images.

Various researches on the optimal canny edge operator have been carried out and many improvement measures have been proposed which could enhance the precision and accuracy of the edge detection. There are also some deficiencies of the dual thresholds of Canny operator. A too-high-setting threshold may miss important information, while a too-low-setting may put too much importance on the minor details. The main idea is to select thresholds, which minimize the within-class variance or maximize the between-class variance. This threshold can both suppress noise and keep the edge fine.

In the proposed edge detection method, the edgemap was obtained by a modified Canny operator using dynamically calculated high and low threshold parameters, according to the actual characteristics of the image. The entire secret image is scanned at run time and the median calculated. By experimenting on various types of images a conclusion was drawn that the thresholds could be set as 2/3 of median and 4/3 of median in order to obtain a continuous set of real edges. This enabled to get more integrated information and the continuity of the edge was strong, and positioning accurate. The block diagram of the proposed edge detection method is as shown in figure 3.22.
Steps for Edge detection using canny algorithm and dynamically calculated thresholds

1. Smoothing: Blurring of the image to remove noise.
2. Finding gradients: The edges should be marked where the gradients of the image has large magnitudes.
3. Non-maximum suppression: Only local maxima should be marked as edges.
4. Determine thresholds based on image characteristics.
5. Double thresholding: Potential edges are determined by thresholding.
6. Edge tracking by hysteresis: Final edges are determined by suppressing all edges that are not connected to a very certain (strong) edge.
Figure 3.22 Flow Chart for edge detection technique
The Canny edge algorithm with modified thresholds is given in figure 3.23.

![Algorithm - Canny Edge Detection Algorithm](image)

**Figure 3.23 Modified Canny Edge Detection Algorithm**

### 3.6.4 Application Graphical User Interface (GUI)

The application graphical user interface (GUI) is designed in a user friendly manner using Microsoft Visual Studio. It includes five main modules: Determine edges of cover image, Secret image compression, Embedding, Extraction and Controller. The GUI allows the user to choose the extract or embedding functions for the secret data. Encode module is implemented to embed the secret data (text or image) using the proposed
algorithms (ADDE or BSDE). The Decode module allows the receiver to extract the hidden data. The controller module is responsible for converting the secret data to binary for embedding purposes. If the secret data to be embedded is an image, it needs to be compressed first using a wavelet based image compression algorithm. If the secret data does not fit within the selected cover image, the user is prompted to change either the message or the image. The stego image will be saved onto the hard disk in the location specified. To hide a file inside a cover image, click on the button "Choose cover image" and on the file dialog that appears, select the file that you want to use as cover image. The next step is to determine the edges within the selected image. Then select the secret text or secret image that is to be hidden by clicking on the button “Select secret data”. If secret data is an image, click the “Compress image” button. Select the algorithm to use for encoding. Finally click the “Encode data” button.

To retrieve a file from a stego-image, select the image to retrieve the file from. Select the button marked "Choose encoded image" and from the file dialog that opens pick the file and click "Decode". Select the algorithm to use for decoding. If secret data is an image, click the “De-compress image” button. The secret data file will be saved onto the hard disk in the location specified. Sample Screenshots of the GUI are shown in Appendix A1.

3.7 Performance Evaluation Measures

Yalman and Erturk, (2013) identified that a fundamental task in most image processing applications is the visual evaluation of a modified image. They outlined many measures for examining image quality, such as the mean structural similarity, mean absolute error, mean square error (MSE), and peak signal-to-noise ratio (PSNR).
3.7.1 Mean Square Error (MSE)

As a performance measurement for image distortion, mean square error and the peak signal to noise ratio could be used as the key measures. MSE refers to the cumulative squared error between the modified (compressed image or stego image) and the original image (Jayachandran and Manikandan, 2010). MSE is computed by performing byte by byte comparisons of the two images. Assuming \( I(i,j) \) to be the original image and \( I'(i,j) \) to be the modified image, the MSE is computed as shown in equation 3.4.

\[
MSE = \frac{1}{MN} \times \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} [I(i,j) - I'(i,j)]^2
\]

(3.4)

Where \( i \) and \( j \) are the image coordinates and \( M \) and \( N \) are the dimensions of the image.

For coloured images we consider the difference between the individual red, green and blue channel components.

3.7.2 Peak Signal to Noise Ratio (PSNR)

The PSNR metric, measured in decibels (dB), expresses the degree of noise introduced after embedding the secret data, by comparing the original against the modified (compressed image or stego image). The higher the PSNR value, the more the modified image resembles the original one (Mare et al., 2011). Human visual system is unable to distinguish images with PSNR more than 35 dB (Ghasemi and Shanbehzadeh, 2010).

The PSNR is a universal formula, which can be straightforwardly applied when we are dealing with gray-scale images; however, when confronting true RGB colour images, we generate the average MSE of the three colour channels and then calculate the PSNR (Cheddad et al., 2008). PSNR value is inversely proportional to MSE value. If MSE is zero, PSNR becomes infinite, means no distortion occurs after embedding.
The mathematical formula to compute PSNR is given in equation 3.5.

\[
PSNR = 10 \times \log_{10}(\frac{C_{max}^2}{MSE})
\]

Where MSE refers to the mean square error as explained in equation 3.4, \(C_{max}\) is the maximum possible pixel value of the image. It usually takes the value 1 for double precision intensity images and takes a value of 255 for 8 bit intensity images.

### 3.7.3 Structural Similarity Index (SSIM)

The structural similarity (SSIM) index is a method used to determine the similarity between two images. The resultant SSIM index is a decimal value between -1 and 1, and value 1 is achieved only in the case of two identical sets of data. Typically it is calculated on window sizes of 8×8. SSIM considers image degradation as perceived change in structural information. Structural information considers that the pixels have strong interdependencies especially when they are spatially close.

The SSIM metric is calculated on various windows of an image using equation 3.6. The measure between two windows and \(\hat{Y}\) of common size \(N \times N\) is \(\text{(Wang et al., 2004; Yalman and Erturk, 2013)}:\)
3.7.4 Compression Ratio or Compression Power

Compression ratio quantifies the reduction in data representation size, produced by a data compression algorithm. It is the ratio between the uncompressed file size and the compressed size.

\[ C_r = \frac{U}{C} \]

Where:
- \( C \) : Compressed File Size
- \( U \) : Uncompressed File Size
- \( C_r \) : Compression Ratio

3.7.5 Space Saving

Defined as the reduction in size relative to the uncompressed file size. Calculated using equation 3.8.
3.7.6 Embedding Rate

Embedding rate denotes the average number of bits that can be embedded per pixel into a given cover image.

3.7.7 Embedding Capacity

It is the size of the payload data in a cover image that can be modified without deteriorating the integrity of the cover image. The steganographic embedding operation needs to preserve the statistical properties of the cover image in addition to its perceptual quality. Capacity is represented by bits per pixel (bpp) and calculated using Equation 3.9.

Embedding capacity corresponds to the total capacity of a given cover image

$$ Embedding\ Capacity = \frac{S_{ij}}{C_{ij}} $$

Where, \( S_{ij} \) is the size of the payload image,
\( C_{ij} \) is the size of the cover image.
3.7.8 Basic Statistical Measures

Since embedding secret data in a cover image results in modification of cover, steganography inevitably leaves some traces in the statistical properties of the image. The basic statistical analysis of the cover images and the stego images was carried out using the mean and standard deviation measures. Standard deviation is the square root of the Variance. Variance is the average of the squared differences from the Mean. The mean is the average of the pixel values. The standard deviation is a measure of how far the signal fluctuates from the mean. The variance represents the power of this fluctuation. Mean is calculated as shown in equation 3.10. Standard deviation can be represented using equation 3.11.

\[
\mu = \frac{1}{N} \times \sum_{(i,j)\in R} x[i,j]
\]  
(3.10)

\[
\sigma = \sqrt{\frac{1}{N-1} \times \sum_{(i,j)\in R} (x[i,j] - \mu)^2}
\]  
(3.11)

Where N refers to number of pixels, x[i,j] refers to each score and \( \mu \) refers to mean or average