Making　Steganography

64 回生　goya

がんばる部長さんシリーズ第２弾[1]。今度は「ステガノグラフィ」という技術を実装していきます。ぜんぜん初心者向けではないのが残念。言語はなぜか Java を使用。あと、丁寧語云々は気にしない方針で。以下、目次。

１．ステガノグラフィーとは

２．画像ファイルの基本

３．画像ファイルのさまざまな分析

４．データの読み込みとビットプレーンへの分解

５．埋め込むプログラム

６．参考文献とプログラム配布

１．ステガノグラフィーとは

「ステガノグラフィー(steganography)」とは画像や音声など、関係のないデータ(vessel, dummy とよばれる)に別のデータを埋め込む技術のこと。暗号(cryptgraphy)とは別の方向性を持った秘密通信の手段の１つなのだが、暗号化と比べるとマイナーな分野になっている。だが、マイナーだからつまらないわけではない[2]。

ちなみに、これと似たようなものに「電子すかし」という画像の著作権などを示す語句を埋め込む技術や QR コード(２次元バーコード)などもある。

この文章はそれを実際にできるプログラムを組んでいく過程を早送りで辿っていく。ちなみにアルゴリズムは、もっとも実装が簡単と思われる、LSB（最下位ビット）埋め込み方式を採用してみた。

２．画像ファイルの基本

まずは画像の基本を少し解説することに。何年か前の部誌にもそれっぽいことが書いてあったのだが一応コピペではない。

コンピュータで扱う画像を、大きく２つに分けると、

・ラスタ（ビットマップ）

点を集めて画像を作ったもの。その結果、拡大すると点の境界線がギザギザになって見えてしまい(ジャギーと呼ぶ)、縮小すると、点のデータが失われてしまう。

1）後輩の方が書く量多くね？という事実には目を瞑って欲しい。

2）そうでも思わないとやってられない。

・ベクタ

あるデータを元に画像を描いたもの。(関数のグラフのような) ビットマップの欠点はなくなっているが、逆に複雑な画像を書くのは難しい。また、対応しているツールが少ないのも問題。

の二種類に分けられる。

さらに、ビットマップも圧縮方法によって数々な種類があるが、今回は、その中でよく目にする４つを紹介する。

１．Windows ビットマップ (.bmp)

Windows の標準の形式。全てのデータを記述してあるので（つまり圧縮しないので）ファイルサイズが他の形式と比較してかなり大きくなる。色の種類で 24bit（フルカラー）, 256 色, 白黒などに分類される。

２．GIF (.gif)

外国の企業がライセンスを失ったので無料で使えるようになった…らしい形式。

１色を選んで透明色に指定できる。（選んだ色は画面に表示されない）

また、同じ色が広い範囲に広がっている画像を特に小さく圧縮できるので アニメーションのような用途によく用いられる。ゲームのエフェクトなども時々見かける。ちなみに、使える色は 256 色限定という欠点がある。可逆圧縮なのでサイズを小さくしてからまた元に戻しても一緒の画像になる。全体的に４の png の方が優れているような気がする。

３．JPEG (.jpg)

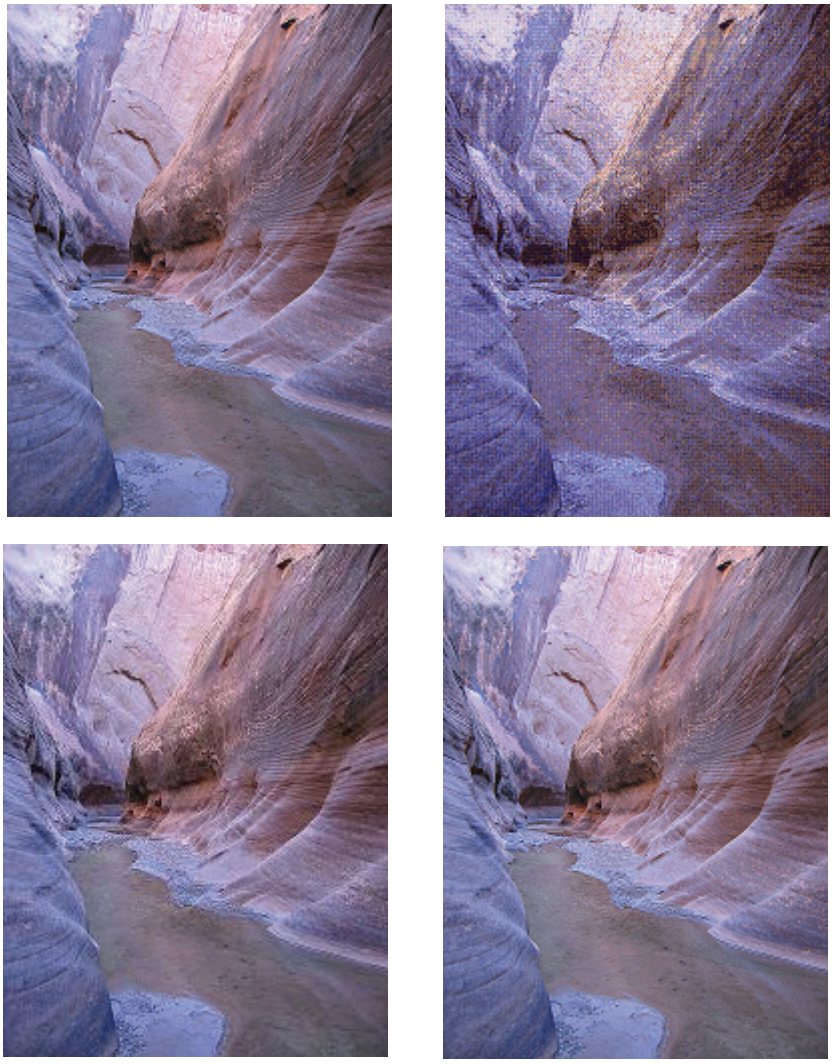
GIF と共にインターネット上ではよく用いられる形式。基本は不可逆圧縮なので、GIF とは違い、小さくすると画像が汚くなり戻せない。また、周波数云々の変換を行っているため、ノイズが発生することもある。しかし、その代わり使える色はフルカラーなので写真などを表現するのには向いている。ちなみに、一応可逆圧縮の方式もあるらしいが、ライセンス等々の問題で使われない事が多いそうだ。

４．PNG (.png)

最近できた圧縮方法。可逆圧縮なのは GIF と同じだが、圧縮率が更に高く、フルカラーを使うことができて、半透明な色の表示もできるようになっている。

写真データでは、JPEG と比べて容量は大きくなってしまうので、写真には JPEG を使うべきだろう。また、PNG では情報が失われないので、PNG で一時的な情報を保存、JPEG を完成形式とするのが良いかと。

今回扱うプログラムで対応しているのはビットマップのうちの「Windows ビットマップ」だけになる。他の形式ではデータが圧縮されているため元のデータに変換しないといけないからだ。以下画像の比較（bmp, gif, jpg, png）



ファイルのサイズはそれぞれ 145, 28, 14, 132 KB[3]。

3．画像ファイルのさまざまな分析

BMP においては(他の形式もそうだが)、　ヘッダ　と呼ばれるものがファイルの最初についている。ヘッダというのは画像の大きさや解像度などの概要が書かれている部分で、直接の中身のデータはヘッダの部分の後に続くデータに書かれている。

3）なぜか png のサイズが大きめ。理由はよく分からない。

中身だけでいいじゃないかと思う人もいるかもしれないが、画像の幅が分からなければ１列ズラッと色が並んだものしかできないと思えば納得できるはず。

画像内容において、ステガノグラフィー技術に関連する情報としては、

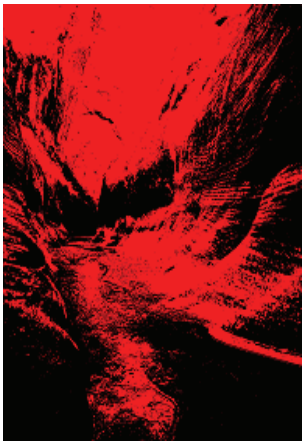
- ビットプレーン (Bit-Plane)
- アルファ値

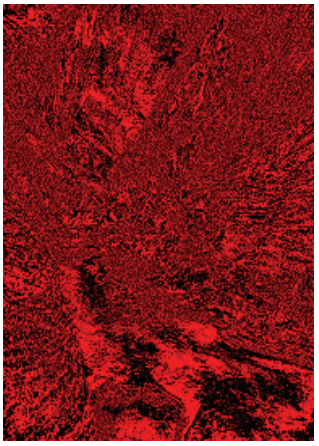
というものが挙げられる。

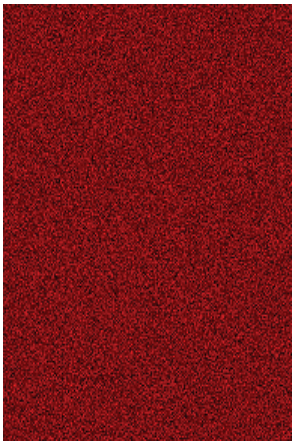
ビットプレーンとは画像をビット単位まで分解したもの。例えば、

１．RGB(Red, Green, Blue)の３色を分解して３枚のプレーンを作る。

２．３枚のそれぞれのプレーンを各ビットに分解して計 24 枚のプレーンを作る、という風にして順に分解する。この結果、それぞれの色のプレーンのうち、１枚目は重要度が高く、偏りが大きいプレーンになり、８枚目は色の調整程度の重要度で、ランダム状になる。今回試した方法ではこれの８枚目のプレーンにデータを埋め込んだ。


プレーン１


プレーン４


プレーン８

プレーン１から徐々にノイズ状になっていくのがわかる。今回は、このノイズ状のプレーン８に情報を書き込んでいった。

アルファ値とは、画像の透明度を表す数値。これが 255 だったら完全に不透明、0 だったら透明になる。この情報も埋め込み領域に使えるらしい。今回の方法では関係ないのでこれ以上詳しい説明は無し。

４．データの読み込みとビットプレーンへの分解

長い長い前置きはここまで。ようやくここからはプログラムを書いていくことになります。

まず、データを読みこんで、ビットプレーンへと分解する作業をするプログラムを書くことにします。そのためにはビットマップファイルのヘッダーに関する

知識が必要になってきますが、面倒なので説明は省きます。

まずはファイル読み込みのソースコードをご覧あれ[4]。（抜粋）

```
public byte[][][] loadImage(){
	byte[] fh = new byte[14];		//FileHeader
	int offset = 0;		//Headerのオフセット
	byte[] ih;	//InfoHeader

	byte[][][] imageFile = null;//返すファイル
	try{
	input.read(fh);
	if(fh[0]!='B' || fh[1]!='M')		;ビットマップでない
		return null;
	offset = fh[10] + fh[11] <<8;
	ih = new byte[offset-fh.length];
	input.read(ih);
	if(ih[14] != 24)
		return null;
	//以上の操作でフルカラー(24bit)ビットマップのみを取得
	int width = (ih[4] & 0xFF) + (ih[5] & 0xFF)<<8+ (ih[6] & 0xFF)
<<16+ (ih[7] & 0xFF) <<24;
	int height = (ih[8] & 0xFF) + (ih[9] & 0xFF) <<8 + (ih[10] & 0xFF)
<<16 +(ih[11] & 0xFF) <<24;
		//ヘッダを保存ファイルにも書き込むので保存しておく
		header = new byte[offset];
		for(int i=0;i<fh.length;i++)
			header[i] = fh[i];
		for(int i=0;i<ih.length;i++)
			header[i + fh.length] = ih[i];
//ここまでヘッダ読み込み
	imageFile = new byte[3][width][height];
	byte[] buf = new byte[3];  //RGBの３色を１単位で読み込み
	for(int i=0;i<height;i++){
		for(int j=0;j<width;j++){
			input.read(buf);
```

4) 「きれいなコード」ではないかも知れないが気にしない方針で。

```
                    //RGBの反転を元に戻す
                    imageFile[0][j][i] = buf[2];
                    imageFile[1][j][i] = buf[1];
                    imageFile[2][j][i] = buf[0];
              }
              for(int j=3*width; j%4 != 0; j++)
                    input.read();
        }
        input.close();
        }catch(Exception e){e.printStackTrace();}
        return imageFile;
}
```

多少解説です。前半はヘッダ読み込みで、ファイルの情報を取得しています。詳しい内容は、Wikipediaで探してください。ちなみに、分かる人にはわかりますが、headerというフィールドが紛れ込んでいます。これは抜粋元のクラスに属する変数です。処理後に、出力するときに使うので保存しておきます[5]。ちなみに、inputというのも、headerと同じくクラス変数で、ファイルの読み込み口の役割を果たしています。

後半部は、前半で取得した情報を利用して、画像の内容を取得します。

特に変わったところは少ないのですが、注意点が２つ。

まず、ビットマップに書き込まれている順番は、BGR（青・緑・赤）です。何となくRGBというのが普通なので、データに格納する順番を入れ替えてあります。

次に、

```
        for(int j=3*width; j%4 != 0; j++)
              input.read();
```

という部分。これは、パディングという仕様が関係しています。画像のデータは、各ピクセルBGRの３バイトが１単位として並んでいるのですが、３×１行のピクセル数　が４の倍数でなくてはならないのです[6]。そして、足りない部分はゼロで勝手に埋めます。

さらに豆知識ですが、この画像データ、実は左下から格納されているのです。普通左上じゃね？と思うのだが、ビットマップは普通じゃないということらしい[7]。

5）ここで挫折した人は、NPCAでプログラムを学べば理解できるようになります。
6）こんな仕様のせいで数日は無駄にした。まったく腹立たしい。
7）ありがたいことに上下が逆で認識していてもソースコードにはまったく関係ない。

次に、ビットプレーンへの分解。これもまたソースコードの抜粋をどうぞ。

```
public byte[][][] splitBitPlane(byte[][][] rgbPlane){
      int width = rgbPlane[0].length;
       int height = rgbPlane[0][0].length;
       byte[][][] planes = new byte[rgbPlane.length * 8][width][height];

       for(int color=0; color<rgbPlane.length; color++){
         for(int x=0; x<width; x++){
           for(int y=0; y < height; y++){
              int dot = 0xFF & rgbPlane[color][x][y];
              for(int bit_count = 7; bit_count >=0; bit_count--){
                     planes[color*8+bit_count][x][y] = (byte)(dot % 2);
                     dot  >>>= 1;
              }
           }
         }
       }
       return planes;
}
```

さて、解説。まず、見ただけでも、やたら繰り返しが多いこと、メモリの使い方がすさまじいこと[8]が分かります。メモリの方は Java の仕様のせいです[9]。

その中で中心的な処理は、

```
       int dot = 0xFF & rgbPlane[color][x][y];
       for(int bit_count = 7; bit_count >=0; bit_count--){
              planes[color*8+bit_count][x][y] = (byte)(dot % 2);
              dot  >>>= 1;
       }
```

の部分です。各バイトを８ビットごとに分解して行きます。

２進数への変換を思い出してくれれば、ちょうどこんな感じだと分かるでしょう。

>>>　という演算子は、ビットを一つ右にシフトし、空いたところを 0 で埋めるという演算子です[10]。分かりにくければ　÷２　とでも考えれば大体 OK です。

8）planes を宣言したところでメモリの使用量が８倍に･･･。

9）仕様のせいでまた１週間ぐらい無駄にした。コピーの種類が２種類あるせいらしい。

10）例として 1001(2)を">>>=1" すると 0100(2) になります。">>>=2"で 0010(2)。

５．埋め込むプログラム

さて、大詰めです。分解したビットプレーンにデータを埋め込んでいきます。

```
public void emb(){
	if(!sizePassFlag){
		System.out.println("ERROR: TOO LARGE FILE");
		return;
	}
	FileInputStream reader = null;
	try{
		reader = new FileInputStream(embed.getAbsolutePath());
		for(int i=0; i<8; i++){
			ioArray(splitBits((byte)(filesize & 0xFF)), false);
			filesize >>= 8;
		}
		byte b;
		while(( b = (byte)(reader.read()) ) != -1)
			ioArray(splitBits(b), false);

		reader.close();
	}catch(Exception e){e.printStackTrace();}
}
private void ioArray(byte[] carrier, boolean readFlag){
	int width = planes[0][0].length;
	int height = planes[0].length;
	if(x+8 >= width){    //列の終端
		int rest = width - x;
		if(readFlag)
		 System.arraycopy(planes[plane][y], x, carrier, 0, rest);
		else
		 System.arraycopy(carrier, 0, planes[plane][y], x, rest);
		x = 0; y++;
		if(y == height){     //色の終端
			x = y = 0; plane += 8;
		}
```

```
            if(readFlag)
             System.arraycopy(planes[plane][y], x, carrier, rest, 8-
rest);
            else
             System.arraycopy(carrier, rest, planes[plane][y], x, 8-
rest);
            x = 8-rest;
      }else{
            if(readFlag)
             System.arraycopy(planes[plane][y], x, carrier, 0, 8);
            else
             System.arraycopy(carrier, 0, planes[plane][y], x,  8);
            x += 8;
      }
}
private byte[] splitBits(byte b){
      int x = b & 0xFF;
      byte[] array = new byte[8];
      for(int i=0; i<8; i++){
            array[7-i] = (byte)(x%2);
            x /= 2;
      }
      return array;
}
```

解説も苦しくなってきます。sizePassFlag、x、y、plane、embed、planes がすべてクラス変数で、関数 splitBits は、バイトからビットの配列に変換する関数です。

emb の方では、埋め込むファイルのサイズ（何ビットか）を求め、１バイト区切りで書き込んだ後、ファイルをビット配列に変換しつつ上書きしていきます。

書き込み処理を行う ioArray では、carrier がデータのやり取りに使う配列で、readFlag は、読み込み処理かどうかを示します。データを取り出すときにも同じ部分を取り出していかなければならないので、処理を一つの関数にまとめてあります。ちなみに sizePassFlag で、埋め込める容量かどうかを確認済みなので、ioArray では例外処理をする必要がありません。

emb()のあとは、ビットプレーンをまたバイトに戻して、ファイルに書き込んでやれば完成です。取り出すときは、同様にビットプレーンに分解して、取り出すサイズを読み込んで、同じように読み込めば、元のデータが取り出せることにな

ります。

試作品で実際に埋め込んでみた写真を載せて見ます。白黒になってしまいますが、カラーでもほぼ判らないぐらいにきっちり埋め込まれています。


左が元データ、右が埋め込み後のデータです。どこが変わったか判りますか？

６．参考文献とプログラム配布

以上でステガノグラフィ作成は終わりです[11]。改めて見るに初心者向けではありません。しかし、中級者以上の方はおそらくフムフムと思ったことでしょう。

参考にさせていただいたサイトを紹介しておきます。

http://www.datahide.com/

ここではBPCSという名前のアルゴリズムを用いたステガノグラフィについてじっくりと解説してくれています。最初はこれを作ろうとしたのですが、少し難しく、挫折してしまいました。それを流用して作ったのが今回のプログラムです。

現状のLBS方式では、理論的には約12.5%の情報しか埋め込むことができませんが、BPCSでは50%近くまで埋め込むことができるそうです。

まるで完成したかの様に書いていますが、実は現在のプログラムにはバグがあって、比較的大きいサイズのファイルを埋め込むと、正しくデータが取り出せない状況が続いています[12]。バグが取れたらnpcaのサイトで公開すると思いますので、

http://trlocon.ddo.jp/~npca/

をたまに訪れてください。近々リニューアルしてるかも。　この部誌を読んでいただきありがとうございました。それでは皆さん、また来年お会いしましょう。

11）私の部誌の負担もようやく終わりです。
12）4月7日現在。1 KB程度のテキストなら問題なく埋め込み可能。