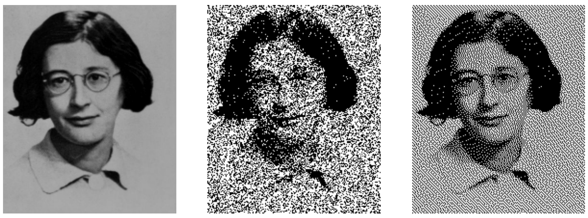
How to hide data in dithering patterns

In this note we describe a simple method for encoding arbitrary data in dithered binary images. The density is about 0.25 bits per pixel in non-saturated regions, and zero bits in saturated regions. Unless the encoded data has some pattern, the encoding is not visible.

1 Description of the method

Sometimes you want to represent gray-scale data by black and white dots. The naive technique is to throw a random binary pixel with the probability of being white given by the gray level. This is called random dithering, and is trivial to implement, but it loses a lot of resolution. A better technique is *error diffusion*, where you traverse the pixels in order and select the black or white value that minimizes the ongoing average error. Notice that this depends on the order of traversal. For uniform regions it tends to produce visible patterns, and this can be avoided by traversing the pixels in a more or less random way (for example, a hilbert curve is often used).



gray-scale random error diffusion

Since there is a lot of choice when dithering an image, we can encode a lot of information in these choices. Assuming that we will be able to recover the binary image exactly, the simplest way to encode the data is to have a **table of patterns** such as this:

pattern																
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
intensity	0	1	1	2	1	2	2	3	1	2	2	3	2	3	3	4
group	–	a_0	a_1	–	b_0	c_0	c_1	d_0	b_1	e_0	e_1	d_1	–	f_0	f_1	–

A binary image is divided in 2×2 cells, and each cell is identified with one of the patterns of the table (cells marked with – are not used). Then the pairs of patterns x_0 and x_1, which have always the same intensity, are considered equivalent and each of them is used to encode a bit of information, losing the original pattern.

The *potential bit content* of a binary image is defined as the number of 2×2 cells that match a valid pattern in this table. Notice that saturated regions (either black or white) can not encode any information, so that it is better to avoid them as much as possible. They can be avoided, for example, by applying a retinex-like transform in the input image, before dithering.

	original	$\gamma = 0.5$	$\gamma = 2$	retinex
gray				
binary				
bytes	987.25	232	343.5	881.875

The following figure shows the effect of the actual encoding. We encode a stream of random bits, and a stream of zero bits. Notice that the stream of zeros introduces a visible pattern in the image. To avoid these patterns, the data to be encoded must have a uniform distribution (for example, by compressing it).

			(click on each image to see it bigger)
input	random	zeros	

2 Implementation

A C implementation of this technique is available in imscript, as the program `mdither`. All the experiments described in this page have been created automatically by extracting the comments in the source (see the HTML source to view them).

2.1 Floyd-Sternberg dithering

To binarize a gray-scale image by Floyd-Sternberg dithering you can use the program "`dither`"

```
dither i/weil.png weil-dit.png
```

weil.png

weil-dit.png

2.2 Counting the carrying capacity of an image

The program "`mdither count`" prints the number of bits, bytes, kilobites and megabytes that can be potentially encoded on a given image

```
mdither count weil-dit.png > weil-capacity.txt
```

```
15218 bits 1902.25 bytes 1.85767 k 0.00181413 M
```

2.3 Encoding bits into a carrier image

The program "`mdither encode`" encodes a stream of bytes into a carrier image. In the following example we encode a random stream of bits and a stream of zeros in the same carrier image.

```
mdither encode weil-dit.png weil-random.png < /dev/urandom
mdither encode weil-dit.png weil-zeros.png  < /dev/zero
```

weil-random.png

weil-zeros.png

2.4 Decoding bits from an image

And this information can be extracted by the program "`mdither decode`":

```
mdither decode weil-random.png | hexdump -vn 128 > weil-random.txt
mdither decode weil-zeros.png  | hexdump -vn 128 > weil-zeros.txt
```

Contents of file `weil-random.txt`:

```
0000000 50af e554 6422 abe3 36e2 fe6d 975b 08f4
0000010 8bbf 3a8d 0942 169f 9f8c eeaa 95e6 ed52
0000020 8380 f072 da5f be96 0148 6507 b5e0 f4b1
0000030 b800 c305 e2f0 4246 c31e cd6b 83ed 7376
0000040 1ef4 0a11 8153 cce2 9f23 7103 5a32 f1a6
0000050 0a26 9e9b 309e 53ec fc37 c0f0 bb9b e6b1
0000060 b64f 5687 efce 4ff5 db2f 7b0b 9e3e db44
0000070 05bd f78e 4160 a69c 095d c39f 6902 8c0c
0000080
```

Contents of file `weil-zeros.txt`:

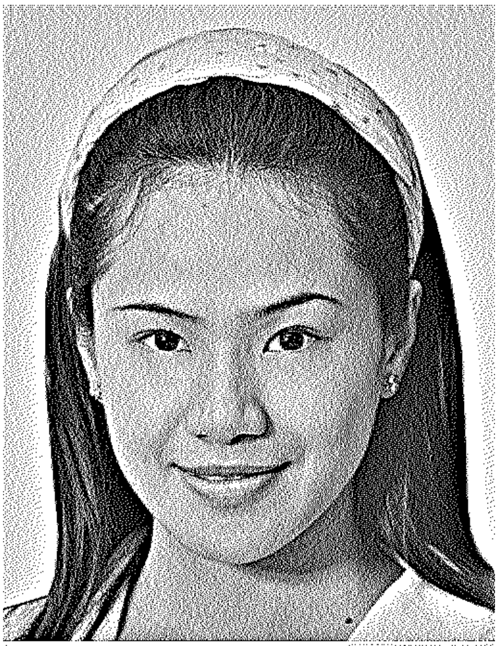
```
0000000 0000 0000 0000 0000 0000 0000 0000 0000
0000010 0000 0000 0000 0000 0000 0000 0000 0000
0000020 0000 0000 0000 0000 0000 0000 0000 0000
0000030 0000 0000 0000 0000 0000 0000 0000 0000
0000040 0000 0000 0000 0000 0000 0000 0000 0000
0000050 0000 0000 0000 0000 0000 0000 0000 0000
0000060 0000 0000 0000 0000 0000 0000 0000 0000
0000070 0000 0000 0000 0000 0000 0000 0000 0000
0000080
```

3 Examples

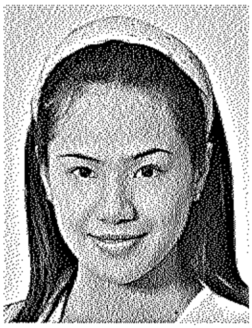
Here we show examples of random bits encoded into the example images of this project, using different resolutions. En each case, we show the binary image along the number of bytes of encoded information it contains.

In all cases, the images were pre-processed by a linear retinex filter and a contrast change that forces the background to be a light-gray (in order to maximize the available space for encoding the information).

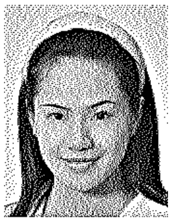
3.1 Test image "photo 1"



7214.5



1694.38



742.25

411.625

252.5

3.2 Test image "photo 2"

8466.88

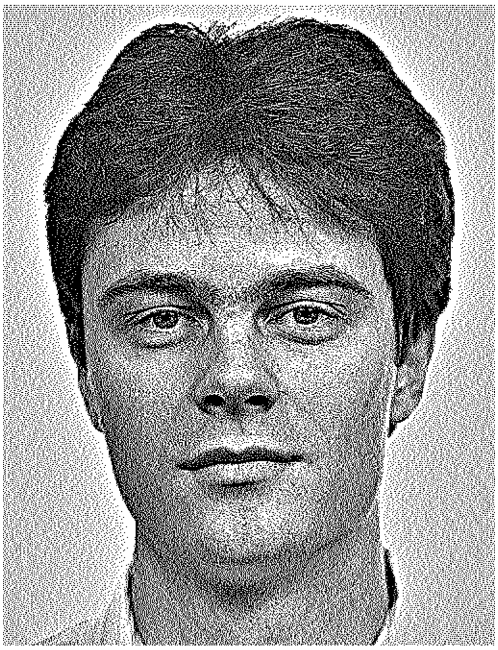
1903.38

815.75

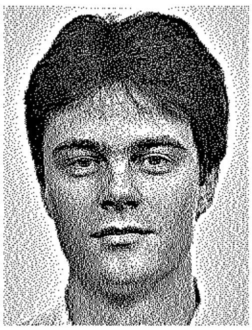
452.875

282

3.3 Test image "photo 3"



7955.5



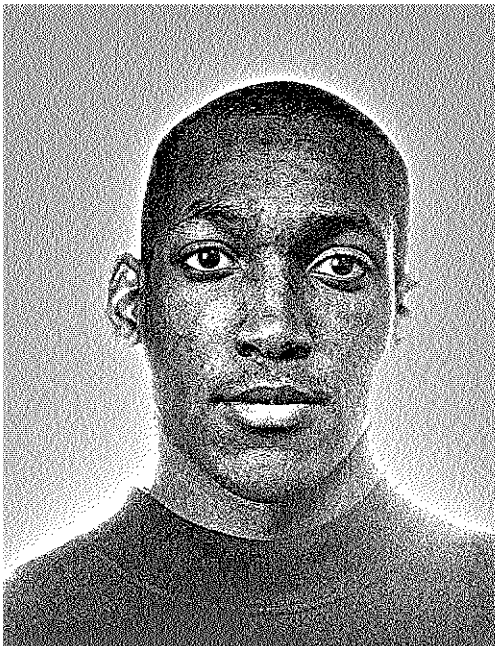
1794.75

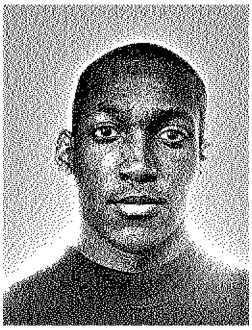
749.125

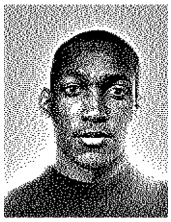
413

257.25

3.4 Test image "photo 4"


8424.38


2004.38

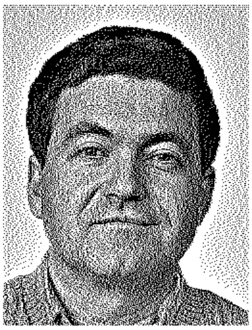

854.875


474.75


298.25

3.5 Test image "photo 5"


8019.5


1849.62


803.375


453.875


284.75