Two-part Coding Images under Euclidian Distortion

Steven de Rooij and Paul Vitányi

August 22, 2005

This document is a summary and reminder of the work I have been doing on the two-part coding of the image of the mouse, before my trip to Australia, so that Paul knows what I have been doing and so that I know where I left off when I return. This text is not written with publication in mind.

The image is coded using a two part code. The first part contains as much as possible of the structure of the image, in the form of an approximation of the image that compresses well, and the second part contains the difference between the approximation and the original image. The image of the mouse probably does not contain a lot of noise, but still we would expect the "most noisy" part of the data to be encoded in the second part of the two-part code. We have:

$$L(\text{mouse}) = L_1(\text{approximation}) + L_2(\text{residue})$$

L_1 should be the length function of a code that is able to detect as much of structure of the data as possible, and use that structure to achieve as good compression of the data as is possible. In the ideal case we would plug in Kolmogorov complexity here. In practise, L_1 denotes the code length of a simple general-purpose block-sorting compressor which is very similar to other widely available block sorting compressors such as bzip2 or zzip.

This document is really about possible implementations of L_2. Since the data that has to be coded by L_2 is modeled as noise, it should be a "flat" code, in other words:

Axiom 1 *L_2 should assign the same codelength to the residue for all approximating images with the same distortion.*

We use Euclidian distortion, which is computed as the sum of the squares of the differences in the grayshade values of the pixels. Probably, it would be easier to design an efficient code for L_2 if the distortion were computed as the sum of the absolute differences, but I consider the squared distortion more natural and it is an interesting problem in its own right to consider good codes for L_2. Strictly speaking, L_2 should also encode the actual distortion, to enable the decoder to reconstruct the original image, but we have ignored this for the moment, since the contribution to the code length is presumably small. Another issue is that one could try to construct the code such that it can only represent residues within some small distortion range; we simplify things by allowing our code to represent every residue up to some maximum distortion, so we only fix a certain maximum. It may be worthwhile to find an efficient resolution to such details in the future.

The following sections describe a number of different possible codes for L_2. After that we plot the lengths of the various alternative codes as a function of the distortion, and the structure function for the mouse using one of the more reasonable codes for L_2.

1 Codes

1.1 Notation

We use the following notation:

d	The number of pixels (dimensions) in the image
e	The Euclidian distortion, $\sum_{i=1}^{d} \Delta_i^2$
O_i	The colour value of pixel number i in the original image
A_i	The colour value of pixel number i in the approximated image
Δ_i	The colour value of pixel number i in the residue

1.2 Exact counting

The best code L_2 would be achieved by giving an index in the list of all valid residue images. A residue image is valid if:

1. It corresponds to a distortion in the correct range (under our simplification: the distortion is lower than a particular maximum)
2. The sum of the approximation and the residue *could* be the original image. This is only possible if all pixel values lie between 0 and 255 inclusive. (I think that this constraint is of minor importance: I believe that the fraction of residues that is not valid for this reason is small under most circumstances.)

To count the number of residues that is valid under the first constraint $R_1(d, e)$, we can use the following recursive definition:

$$\begin{aligned} R_1(d,e) &= \sum_{i=0}^{\lfloor\sqrt{e}\rfloor} R_1(d-1, e-i^2) \\ R_1(d,0) &= 1 \\ R_1(0,e) &= I(e=0) \end{aligned}$$

This quantity is very hard to compute in practise since one has to add extremely large numbers together (about 256^d). To obtain useful codes, we have to make sure that we can compute the *logarithm* of the number of possibilities directly, which is of manageable size.

If we also want to take the second constraint into account, we additionally need A to compute the number of possibilities:

$$R_{1,2}(A,d,e) = \sum_{\substack{0\leq i<\sqrt{e} \\ i:\ 0\leq A_d - i \\ A_d + i \leq 255}} R_{1,2}(A,d-1,e-i^2),$$

with similar base cases as before.

Given that this approach is not practical, we give a number of different codes.

1.3 Trivial code

An easy upper bound on the size of valid residues is the total number of possible images. Here we make use only of the second constraint: every pixel in the original image has a value between 0 and 255. Thus, the number of possible images is 256^d, and the number of valid residues cannot be any larger, so an upper bound on the code length is $8d$ bits.

1.4 Ball code

This code approximates R_1 with the volume of an n-dimensional ball. The following image illustrates the idea in two dimensions (so this would be a two-pixel image).

The black dots represent the allowed residues. With every dot we associate a little region of space with unit volume: a d-dimensional hypercube centered on the solution. To get an upper bound on the number of dots, we can upper bound the volume of space that is used by all the little hypercubes. We do this with a sphere of which the radius has been chosen such that all hypercubes fall completely within the sphere. All points can be at most $\sqrt{e}$ from the center of the sphere, but the hypercubes in which they lie can extend a little bit further. The largest possible distance from the centre within a hypercube is $\frac{1}{2}\sqrt{d}$, so all hypercubes must lie within a sphere of radius $r = \sqrt{e} + \frac{1}{2}\sqrt{d}$. The volume of a sphere with radius r in d dimensions is:

$$V(d,r) = \frac{\pi^{d/2}}{\Gamma(1+d/2)} r^d$$

This allows us to compute an upper bound on the number of solutions. Similarly, to get a lower bound, we can compute the volume of a sphere

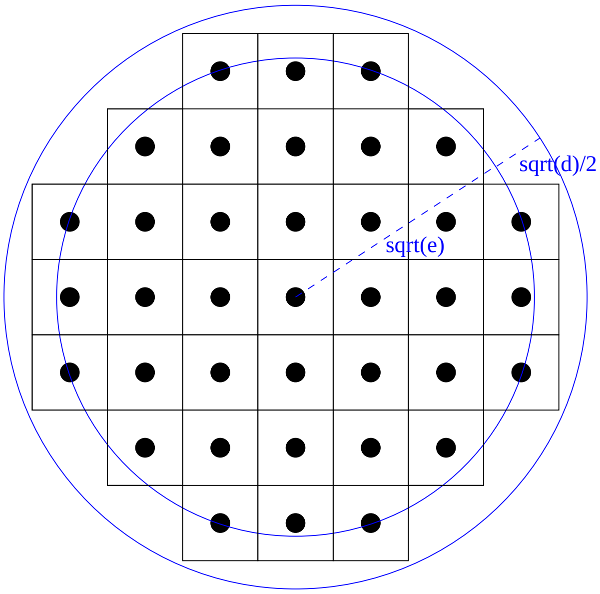


Figure 1: Construction of a sphere with a volume as least as large as the number of possible residues.

that is completely covered with hypercubes; we use a sphere with radius $\sqrt{e} - \frac{1}{2}\sqrt{d}$ for this.

It is to be expected that for low distortion, the volume used by the hypercubes is overestimated by a lot, while for high distortion the lower and the upper bound should become close. This is confirmed by the plot of this function in Figure 2. In practise, we will typically want to compute the number of solutions for a very high dimensionality ($d = 2560$ for the mouse), but with the distortion varying between 0 and some very high number. It may be inefficient to use this code when the distortion is low compared to the dimensionality.

1.5 Gaussian code

A completely different approach is to encode all residue pixels independently. Let $L(\Delta_i)$ denote the number of bits needed to encode residue pixel number i. Because of the code axiom, we must have:

$$L(x) = b + c \cdot x^2.$$

If we now look at the probability distribution that corresponds to this code we get:

$$P(x) = 2^{-L(x)} = 2^{-b} 2^{-c \cdot x^2} = C \cdot e^{-c \cdot x^2} = C \cdot e^{-x^2/2\sigma^2}$$

This is a discretized version of the normal distribution. The value of σ^2 that minimizes the code length is the value that maximizes the likelihood. If we assume that the discrete case is reasonably similar to the continuous case, the parameter σ^2 can be estimated as the average of the squared deviations, which is exactly e/d. Plugging this in the equation we get for the whole image:

$$P(\Delta) = \prod_{i=1^d} C \cdot e^{-\Delta_i^2/2\sigma^2} = C^d e^{-\sum_{i=1}^d \Delta_i^2/2\sigma^2} = C^d e^{-\frac{1}{2}d},$$

and the corresponding codelength is $-d \lg C + \frac{1}{2} d \lg e$.

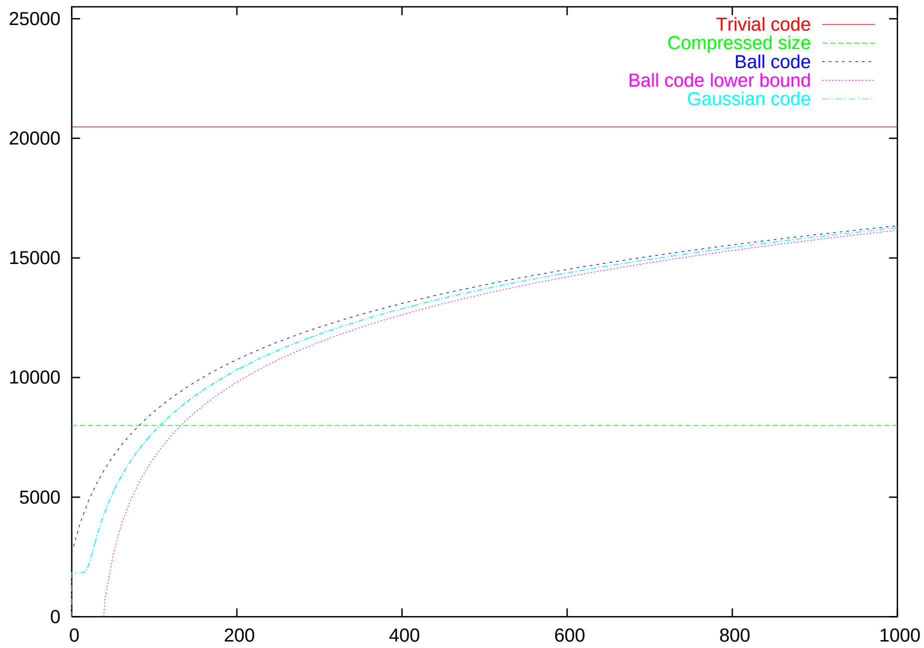
This code seems to be a bit better in practice than the ball code, even though it is also not too efficient when the distortion is low (because then the independence assumption is costly).

2 Results

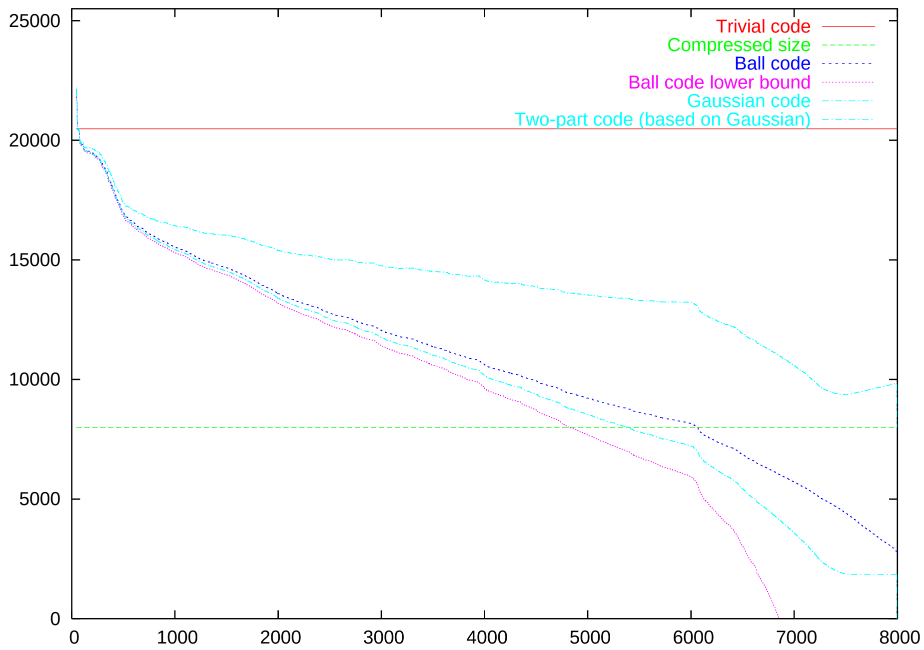
The image of the mouse has the following statistics:

- d $64 \cdot 40 = 2560$ pixels
- e $23,933,378$ when the codelength for the approximation is the smallest necessary to produce an image at all (43.4 bits), and 0 for the original image (of which the codelength is 7995.1 bits.)

For these d and e we have obtained the following graphs. The first graph is made using the dimensionality of the mouse, but it is not based on experimental data, it just relates the distortion to the log of the ball size. The second graph shows the structure function as it is found for the mouse.



The distortion against the codelength of the residue image (log ball size)



The codelength of the approximation against the codelength of the residue image.