1 About the entry

This is the readme file for Linus Åkesson's contribution to the 2015 installment of the Underhanded C contest.

The source code is organized into three files, with a total of 66 lines of code: `match.h`, `match.c` and `spectral_contrast.c`.

The complete source code is included at the end of this document (and, of course, as regular files in the archive).

Also included in the archive is a driver program, `test.c`, that generates raw data for the plots used in this file *and in the spoilers document*. Reading `test.c` too closely may thus give away the secret.

What follows is intended as official documentation for the fissile material detector. It may or may not be misleading.

❧

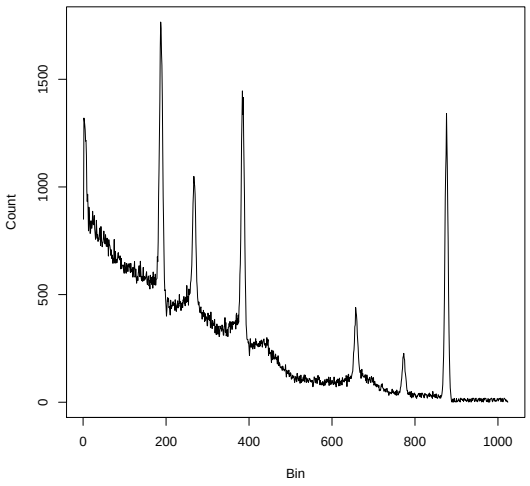
2 Theory of operation

The `match` function compares two gamma-ray spectra: A reference pattern and a test pattern. Figure 1 shows a typical gamma-ray spectrum.

When comparing two materials, we are mainly concerned with the position and relative height of peaks. Each input spectrum is therefore *pre-processed* as follows: A smoothing operation is applied to remove most of the noise (Figure 2). To get rid of the slanted noise floor, the first-order differential is computed (Figure 3). The smoothing operation is applied a second time to obtain the final *spectral fingerprint* (Figure 4).

The similarity of the two fingerprints is then determined by computing their *spectral contrast angle* [1]. Each pattern is regarded as a multi-dimensional vector. The dot product of two

Figure 1: Typical gamma-ray spectrum.



normalized vectors is equal to the cosine of the angle between them:

$$\frac{A}{||A||} \cdot \frac{B}{||B||} = \cos(\theta)$$

The `match` function computes the cosine as described above, and compares it directly to the supplied threshold parameter. The cosine will reach its maximum value (1.0) when the two vectors are fully congruent, i.e. all peaks are precisely aligned in the two samples, and their relative sizes match perfectly.

On its own, however, this criterion is not sufficient to identify a nuclear warhead as eligible for dismantling. The reason is that the algorithm compares the shape while ignoring the overall size of the two spectra. Thus, a country may present a warhead containing only a small fraction of the expected fissile material, and the test vector would still align perfectly with the reference vector.

Therefore, a second criterion is introduced: The total amount of gamma-ray activity in the test pattern—defined as the sum of all bins—must be at least as great as some given fraction of the total amount of gamma-ray activity in the reference pattern. The same threshold parameter is used to indicate the desired fraction.

Figure 2: After the first smoothing operation.

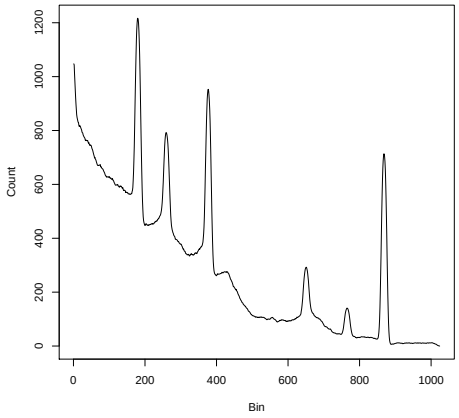


Figure 3: After taking the differential.

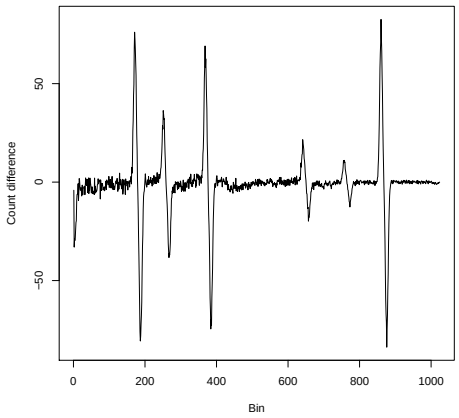
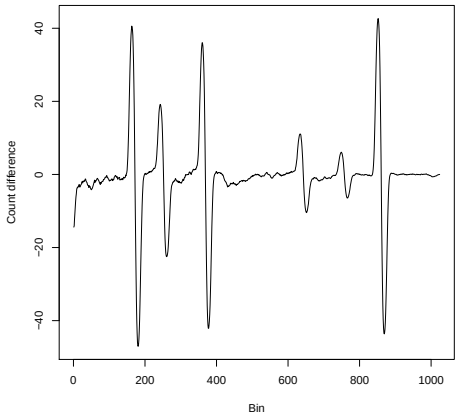


Figure 4: Final spectral fingerprint.



3 Evaluation

A set of 1000 artificial materials were generated, with each material represented as a set of peaks with randomized relative sizes. For each material, a pair of spectra were generated, with small random variations. The `match` routine was probed using a binary search to determine the *critical threshold*, i.e. the highest threshold at which the routine considers the two generated spectra to be matching. The result is shown as dark bars in the histogram in Figure 5.

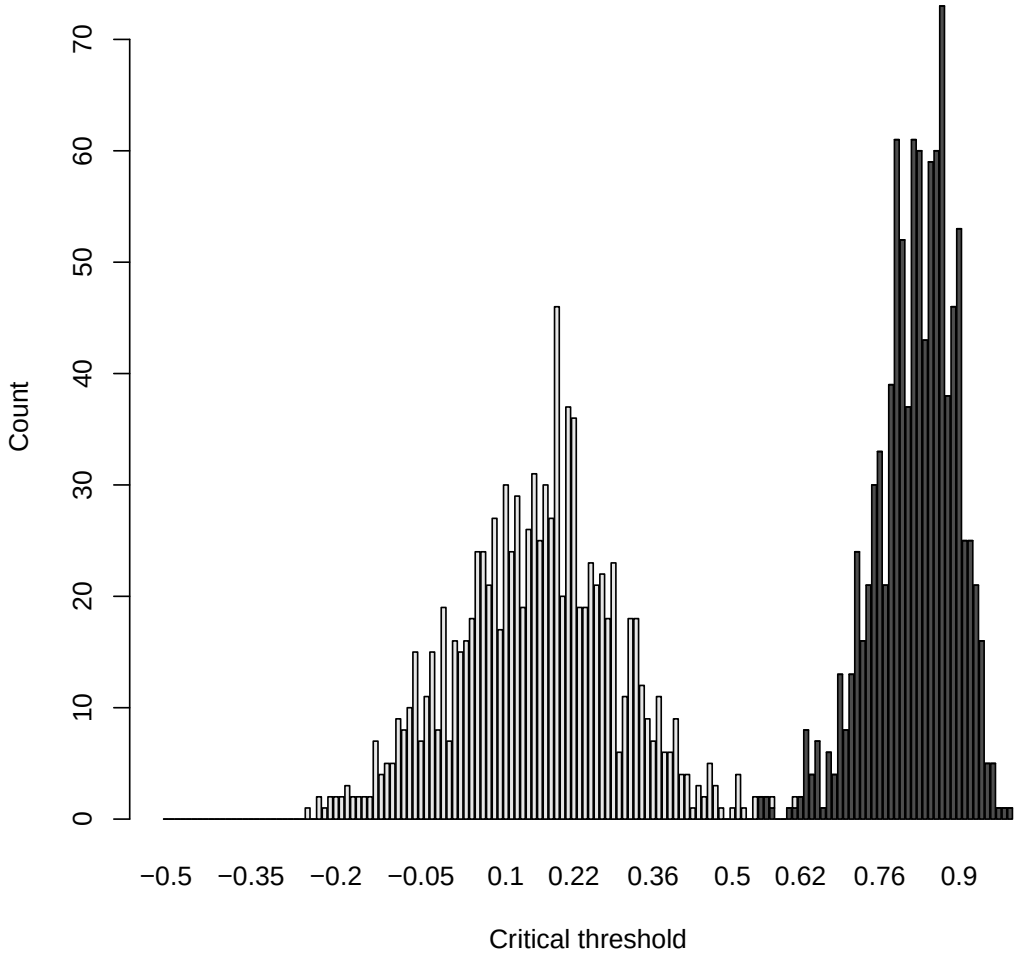
Next, a set of 1000 pairs of artificial materials were generated, and the critical threshold for finding a `match` between spectra of the two different materials was computed in a similar manner. The result is shown as light bars in the same histogram.

It follows from a visual inspection of the histogram that a threshold value of 0.6 should work well in practice.

References

[1] K. X. Wan, I. Vidavsky, and M. L. Gross. Comparing similar spectra: from similarity index to spectral contrast angle. *Journal of the American Society for Mass Spectrometry*, 13(1):85 – 88, 2002.

Figure 5: Histogram over critical thresholds for spectra of the same material (dark) and of different materials (light).



Complete source code

match.h

```
#define N_SMOOTH 16     /* Size of smoothing kernel */
typedef double float_t; /* Desired precision for floating-point vectors */

int match(float_t *test, float_t *reference, int bins, double threshold);
double spectral_contrast(float_t *a, float_t *b, int length);
```

match.c

```
#include <string.h> /* memcpy */
#include "match.h"

static double total(float_t *v, int length) {
        double sum = 0;
        int i;
        for(i = 0; i < length; i++) sum += v[i];
        return sum;
}

static void smoothen(float_t *v, int length) {
        double sum;
        int i, j;
        for(i = 0; i < length; i++) {
                sum = 0;
                for(j = 0; j < N_SMOOTH && i + j < length; j++)
                        sum += v[i + j];
                v[i] = sum / N_SMOOTH;
        }
}

static void differentiate(float_t *v, int length) {
        int i;
        for(i = 0; i < length - 1; i++) v[i] = v[i + 1] - v[i];
        v[length - 1] = 0;
}

static void preprocess(float_t *v, float_t *source, int length) {
        memcpy(v, source, length * sizeof(*v));
        smoothen(v, length);
        differentiate(v, length);
        smoothen(v, length);
}

int match(float_t *test, float_t *reference, int bins, double threshold) {
        float_t t[bins], r[bins];
        if(total(test, bins) < threshold * total(reference, bins)) return 0;
        preprocess(t, test, bins);
        preprocess(r, reference, bins);
        return spectral_contrast(t, r, bins) >= threshold;
}
```

spectral_contrast.c

```
#include <math.h> /* sqrt */

static double dot_product(float_t *a, float_t *b, int length) {
        double sum = 0;
        int i;
        for(i = 0; i < length; i++) sum += a[i] * b[i];
        return sum;
}

static void normalize(float_t *v, int length) {
        double magnitude = sqrt(dot_product(v, v, length));
        int i;
        for(i = 0; i < length; i++) v[i] /= magnitude;
}

double spectral_contrast(float_t *a, float_t *b, int length) {
        normalize(a, length);
        normalize(b, length);
        return dot_product(a, b, length);
}
```