```
#ifdef RCSID
static const char rcsid[] =
  "$Id: fftconv.c,v 1.1 2005/07/12 07:42:10 cperciva Exp $";
#endif

#include <assert.h>
#include <math.h>
#include <stdint.h>

#include "fftconv.h"
```

1 FFT normalization

The FFT and inverse FFT given in fft.c are unnormalized, i.e., the length-N FFT followed by a length-N inverse FFT leaves the output equal to N times the input. To remedy this, the function *tricl_fftconv_scale*(DAT, n) should be called at some point.

As in fft.c, n must satisfy $0 \leq n \leq 29$, and DAT must be an array of 2^n complex values (2^{n+1} doubles).

```
void tricl_fftconv_scale(double * DAT, int n)
{
        double s;
        int32_t i;

        assert(0 <= n && n <= 29);

        s = ldexp(1.0, -n);
        for (i = 0; i < 2 << n; i++)
                DAT[i] = DAT[i] * s;
}
```

2 Pointwise complex products

The function *tricl_fftconv_mulpw*($DAT1$, $DAT2$, n) computes the product of 2^n pairs of complex values from $DAT1$ and $DAT2$ and writes the resulting values into $DAT1$.

As usual, n must satisfy $0 \leq n \leq 29$, and $DAT1$ and $DAT2$ must be non-overlapping arrays of 2^n complex values (2^{n+1} doubles).

```
void tricl_fftconv_mulpw(double * __restrict DAT1,
    double * __restrict DAT2, int n)
{
        double xr, xi;
        int i;
```

```
        assert(0 <= n && n <= 29);

        for (i = 0; i < 1 << n; i++) {
                xr = DAT1[i * 2];
                xi = DAT1[i * 2 + 1];

                DAT1[i * 2] = xr * DAT2[i * 2] - xi * DAT2[i * 2 + 1];
                DAT1[i * 2 + 1] = xr * DAT2[i * 2 + 1] + xi * DAT2[i * 2];
        }
}
```

The function *tricl_fftconv_sqrpw*(DAT, n) squares 2^n complex values from DAT and writes the resulting values into DAT.

As usual, n must satisfy $0 \leq n \leq 29$, and DAT must be an array of 2^n complex values (2^{n+1} doubles).

```
void tricl_fftconv_sqrpw(double * DAT, int n)
{
        double xr, xi;
        int i;

        assert(0 <= n && n <= 29);

        for (i = 0; i < 1 << n; i++) {
                xr = DAT[i * 2];
                xi = DAT[i * 2 + 1];

                DAT[i * 2] = xr * xr - xi * xi;
                DAT[i * 2 + 1] = 2 * xr * xi;
        }
}
```

3 FFT convolution

To compute a length-2^n convolution of two vectors X and Y:

```
tricl_fft_makelut(LUT, n);
tricl_fft_fft(X, n, LUT);
tricl_fft_fft(Y, n, LUT);
tricl_fftconv_mulpw(X, Y, n);
tricl_fft_ifft(X, n, LUT);
tricl_fftconv_scale(X, n);
```

although the call to tricl_fftconv_scale can be performed at any point in the process, and on either X or Y.

Theorem 1. *When computed in this manner, the convolution z of two length-2^n complex vectors x and y will satisfy*

$$|z' - z|_\infty < |x| \cdot |y| \cdot \left((1+\epsilon)^{3n} (1+\epsilon\sqrt{5})^{3n+1} (1+1.5\epsilon)^{3n} - 1 \right) < |x| \cdot |y| \cdot (14.3n + 2.3)\epsilon$$

where $\epsilon = 2^{-53}$ is the maximum relative error in double-precision floating-point arithmetic.

Proof. The FFT used is a split-radix FFT, not a radix-2 FFT, but the argument from Theorem 5.1 of [1] still applies (the only difference as far as error bounds are concerned is that the split-radix FFT has fewer complex multiplications; but this reduction does not affect the worst case). From [2] we note that we can take $\beta = 1.5\epsilon$ to complete the proof. □

References

[1] C. Percival, *Rapid multiplication modulo the sum and difference of highly composite numbers*, Math. Comp. **72** (2003), 387–395.

[2] C. Percival, *roots.c*, in *TRICL*, `http://www.daemonology.net/tricl/`, (2005).

A Copyright