```
#ifdef RCSID
static const char rcsid[] =
  "$Id: fft.c,v 1.2 2005/07/11 11:31:39 cperciva Exp $";
#endif

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

#include "local.h"

#include "fft.h"
#include "roots.h"
```

1 Twiddle factors

The function *tricl_fft_makelut*(LUT, n) generates an FFT lookup table suitable for use in computing FFTs of length up to 2^n. This table is in fact a series of tables of sizes 4, 8, 16, $\ldots 2^{n-2}$ complex values; the table of size 2^{k-2} contains the 2^kth roots of unity in the first quadrant, in order of increasing angle.

For example, *tricl_fft_makelut*(LUT, 32) would generate a table containing the following values:

$$
\begin{gathered}
0+0i\\
0+0i\\
0+0i\\
0+0i\\
cos(0)+sin(0)i\\
cos(\pi/8)+sin(\pi/8)i\\
cos(\pi/4)+sin(\pi/4)i\\
cos(3\pi/8)+sin(3\pi/8)i\\
cos(0)+sin(0)i\\
cos(\pi/16)+sin(\pi/16)i\\
cos(\pi/8)+sin(\pi/8)i\\
cos(3\pi/16)+sin(3\pi/16)i\\
cos(\pi/4)+sin(\pi/4)i\\
cos(5\pi/16)+sin(5\pi/16)i\\
cos(3\pi/8)+sin(3\pi/8)i\\
cos(7\pi/16)+sin(7\pi/16)i
\end{gathered}
$$

(Note that the first four complex values are left zeroed, in order that the table of size N starts at the Nth complex value.)

The input n must satisfy $0 \leq n \leq 29$, and LUT must have space to store 2^n doubles (i.e., 2^{n+3} bytes).

```
void tricl_fft_makelut(double * LUT, int n)
{
        int32_t i;

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

If $n < 4$, we just need to write 2^n zeroes.

```
        if (n < 4) {
                for (i = 0; i < (1 << n); i++)
                        LUT[i] = 0.0;
                return;
        }
```

Otherwise, write 8 zeroes…

```
        for (i = 0; i < 8; i++)
                LUT[i] = 0.0;
```

…generate the table of size 2^{n-2}…

```
        tricl_roots_makelut(LUT + (1 << (n - 1)), n);
```

…and then generate the smaller tables by copying every second value from the next larger table.

```
        for (i = (1 << (n - 1)) - 2; i >= 8; i -= 2) {
                LUT[i] = LUT[i * 2];
                LUT[i + 1] = LUT[i * 2 + 1];
        }
}
```

2 The FFT

We use a recursive "exponent -1" split-radix decimation-in-frequency FFT, based roughly on ideas from Bernstein [1].

2.1 Building blocks

We build the FFT and inverse FFT out of seven primative blocks: *FFT_PM*, *FFT_SRM* and *IFFT_SRM*, *FFT_SRM_PI_4* and *IFFT_SRM_PI_4*, and *FFT_SRM_W* and *IFFT_SRM_W*.

The macro *FFT_PM* transforms (a, b) into $(a + b, a - b)$, i.e., it performs a length-2 FFT (which is also a length-2 inverse FFT).

```
#define FFT_PM(a, b)    do {    \
        double t;               \
                                \
        t = (b)[0];             \
        (b)[0] = (a)[0] - t;    \
        (a)[0] += t;            \
        t = (b)[1];             \
        (b)[1] = (a)[1] - t;    \
        (a)[1] += t;            \
} while (0)
```

The macro *FFT_SRM* performs "split-radix mixing" of four values: It transforms (a, b, c, d) into $(a + c, b + d, (a - c) + (b - d)i, (a - c) - (b - d)i)$ – this equates to one pass of FFT on half the data and two passes of FFT minus twiddling on the other half of the data.

The macro *IFFT_SRM* is the inverse of *FFT_SRM*.

```
#define FFT_SRM(a, b, c, d)     do {    \
        double t0r, t0i, t1r, t1i;      \
                                        \
        t0r = (a)[0] - (c)[0];          \
        (a)[0] += (c)[0];               \
        t0i = (a)[1] - (c)[1];          \
        (a)[1] += (c)[1];               \
        t1r = (b)[0] - (d)[0];          \
        (b)[0] += (d)[0];               \
        t1i = (b)[1] - (d)[1];          \
        (b)[1] += (d)[1];               \
                                        \
        (c)[0] = t0r - t1i;             \
        (d)[0] = t0r + t1i;             \
        (c)[1] = t0i + t1r;             \
        (d)[1] = t0i - t1r;             \
} while (0)

#define IFFT_SRM(a, b, c, d)    do {            \
        double t0r, t0i, t1r, t1i, t2r, t2i;    \
                                                \
        t2r = (c)[0];                           \
        t2i = (c)[1];                           \
```

```
        t0r = t2r + (d)[0];                     \
        t0i = t2i + (d)[1];                     \
        t1r = t2i - (d)[1];                     \
        t1i = (d)[0] - t2r;                     \
                                                \
        (c)[0] = (a)[0] - t0r;                  \
        (a)[0] += t0r;                          \
        (c)[1] = (a)[1] - t0i;                  \
        (a)[1] += t0i;                          \
        (d)[0] = (b)[0] - t1r;                  \
        (b)[0] += t1r;                          \
        (d)[1] = (b)[1] - t1i;                  \
        (b)[1] += t1i;                          \
} while (0)
```

The macro *FFT_SRM_PI_4* performs split-radix mixing and twiddling of four values: It transforms (a, b, c, d) into $(a+c, b+d, ((a-c)+(b-d)i)w, ((a-c)-(b-d)i)\bar{w})$ where $w = \exp(i\pi/4) = (1+i)/\sqrt{2}$. This is equivalent to *FFT_SRM* followed by rotating c left by $\pi/4$ and rotating d right by $\pi/4$.

The macro *IFFT_SRM_PI_4* is the inverse of *FFT_SRM_PI_4*.

```
#define SQRTHALF TRICL_ROOTS_SQRTHALF

#define FFT_SRM_PI_4(a, b, c, d)        do {    \
        double t0r, t0i, t1r, t1i, t2r, t2i;    \
                                                \
        t0r = (a)[0] - (c)[0];                  \
        (a)[0] += (c)[0];                       \
        t0i = (a)[1] - (c)[1];                  \
        (a)[1] += (c)[1];                       \
        t1r = (b)[0] - (d)[0];                  \
        (b)[0] += (d)[0];                       \
        t1i = (b)[1] - (d)[1];                  \
        (b)[1] += (d)[1];                       \
                                                \
        t2r = t0r - t1i;                        \
        t2i = t0i + t1r;                        \
        t0r += t1i;                             \
        t0i -= t1r;                             \
                                                \
        (c)[0] = (t2r - t2i) * SQRTHALF;        \
        (c)[1] = (t2r + t2i) * SQRTHALF;        \
        (d)[0] = (t0r + t0i) * SQRTHALF;        \
        (d)[1] = (t0i - t0r) * SQRTHALF;        \
} while (0)
```

```
#define IFFT_SRM_PI_4(a, b, c, d)       do {    \
        double t0r, t0i, t1r, t1i, t2r, t2i;    \
                                                \
        t0r = ((c)[0] + (c)[1]) * SQRTHALF;     \
        t0i = ((c)[1] - (c)[0]) * SQRTHALF;     \
        t1r = ((d)[0] - (d)[1]) * SQRTHALF;     \
        t1i = ((d)[0] + (d)[1]) * SQRTHALF;     \
                                                \
        t2r = t0i - t1i;                        \
        t2i = t1r - t0r;                        \
        t0r += t1r;                             \
        t0i += t1i;                             \
                                                \
        (c)[0] = (a)[0] - t0r;                  \
        (a)[0] += t0r;                          \
        (c)[1] = (a)[1] - t0i;                  \
        (a)[1] += t0i;                          \
        (d)[0] = (b)[0] - t2r;                  \
        (b)[0] += t2r;                          \
        (d)[1] = (b)[1] - t2i;                  \
        (b)[1] += t2i;                          \
} while (0)
```

The macro *FFT_SRM_W* performs split-radix mixing and twiddling of four values: It transforms (a, b, c, d) into $(a+c, b+d, ((a-c)+(b-d)i)w, ((a-c)-(b-d)i)\bar{w})$ where w is a root of unity.

The macro *IFFT_SRM_W* is the inverse of *FFT_SRM_W*.

```
#define FFT_SRM_W(a, b, c, d, w)        do {    \
        double t0r, t0i, t1r, t1i, t2r, t2i;    \
                                                \
        t0r = (a)[0] - (c)[0];                  \
        (a)[0] += (c)[0];                       \
        t0i = (a)[1] - (c)[1];                  \
        (a)[1] += (c)[1];                       \
        t1r = (b)[0] - (d)[0];                  \
        (b)[0] += (d)[0];                       \
        t1i = (b)[1] - (d)[1];                  \
        (b)[1] += (d)[1];                       \
                                                \
        t2r = t0r - t1i;                        \
        t2i = t0i + t1r;                        \
        t0r += t1i;                             \
        t0i -= t1r;                             \
```

```
                                                    \
        (c)[0] = t2r * (w)[0] - t2i * (w)[1];   \
        (c)[1] = t2i * (w)[0] + t2r * (w)[1];   \
        (d)[0] = t0r * (w)[0] + t0i * (w)[1];   \
        (d)[1] = t0i * (w)[0] - t0r * (w)[1];   \
} while (0)

#define IFFT_SRM_W(a, b, c, d, w)       do {    \
        double t0r, t0i, t1r, t1i, t2r, t2i;    \
                                                \
        t0r = (c)[0] * (w)[0] + (c)[1] * (w)[1];\
        t0i = (c)[1] * (w)[0] - (c)[0] * (w)[1];\
        t1r = (d)[0] * (w)[0] - (d)[1] * (w)[1];\
        t1i = (d)[1] * (w)[0] + (d)[0] * (w)[1];\
                                                \
        t2r = t0i - t1i;                        \
        t2i = t1r - t0r;                        \
        t0r += t1r;                             \
        t0i += t1i;                             \
                                                \
        (c)[0] = (a)[0] - t0r;                  \
        (a)[0] += t0r;                          \
        (c)[1] = (a)[1] - t0i;                  \
        (a)[1] += t0i;                          \
        (d)[0] = (b)[0] - t2r;                  \
        (b)[0] += t2r;                          \
        (d)[1] = (b)[1] - t2i;                  \
        (b)[1] += t2i;                          \
} while (0)
```

2.2 Small FFTs

Up to length 2^4, we provide hard-coded FFTs:

```
static void fft_0(__unused double * __restrict DAT,
    __unused double * __restrict LUT)
{

        /* Do nothing */
}

static void fft_1(double * __restrict DAT,
    __unused double * __restrict LUT)
{
```

```
        FFT_PM(DAT, DAT + 2);
}

static void fft_2(double * __restrict DAT,
    __unused double * __restrict LUT)
{

        FFT_SRM(DAT, DAT + 2, DAT + 4, DAT + 6);
        FFT_PM(DAT, DAT + 2);
}

static void fft_3(double * __restrict DAT,
    __unused double * __restrict LUT)
{

        FFT_SRM(DAT, DAT + 4, DAT + 8, DAT + 12);
        FFT_SRM_PI_4(DAT + 2, DAT + 6, DAT + 10, DAT + 14);
        FFT_PM(DAT + 8, DAT + 10);
        FFT_PM(DAT + 12, DAT + 14);
        FFT_SRM(DAT, DAT + 2, DAT + 4, DAT + 6);
        FFT_PM(DAT, DAT + 2);
}

static void fft_4(double * __restrict DAT,
    double * __restrict LUT)
{

        FFT_SRM(DAT, DAT + 8, DAT + 16, DAT + 24);
        FFT_SRM_W(DAT + 2, DAT + 10, DAT + 18, DAT + 26, LUT + 10);
        FFT_SRM_PI_4(DAT + 4, DAT + 12, DAT + 20, DAT + 28);
        FFT_SRM_W(DAT + 6, DAT + 14, DAT + 22, DAT + 30, LUT + 14);
        FFT_SRM(DAT + 16, DAT + 18, DAT + 20, DAT + 22);
        FFT_PM(DAT + 16, DAT + 18);
        FFT_SRM(DAT + 24, DAT + 26, DAT + 28, DAT + 30);
        FFT_PM(DAT + 24, DAT + 26);
        FFT_SRM(DAT, DAT + 4, DAT + 8, DAT + 12);
        FFT_SRM_PI_4(DAT + 2, DAT + 6, DAT + 10, DAT + 14);
        FFT_PM(DAT + 8, DAT + 10);
        FFT_PM(DAT + 12, DAT + 14);
        FFT_SRM(DAT, DAT + 2, DAT + 4, DAT + 6);
        FFT_PM(DAT, DAT + 2);
}
```

2.3 Large FFTs

For lengths 2^5 up to 2^{29}, we define a generic split-radix FFT macro which outputs a function for computing that particular FFT length (doing some computations and then recursing into the length 2^{n-1} and length 2^{n-2} FFTs, as the split-radix FFT normally does). We then construct an array of pointers to the FFT functions and tricl_fft_fft looks up and calls the correct function – this is around 5–10% faster than a single recursive FFT function.

This could be made slightly faster by using *FFT_SRM_PI_4*, but the difference is less than 1%, so it isn't worth the cost of the increased code size which would result.

```
#define FFT_FUNC(n, nm1, nm2, len)                                          \
static void fft_ ## n(double * __restrict DAT,                              \
    double * __restrict LUT)                                                \
{                                                                           \
        uint32_t i;                                                         \
                                                                            \
        FFT_SRM(DAT, DAT + len * 2, DAT + len * 4, DAT + len * 6);          \
        for (i = 2; i < 2 * len; i += 2)                                    \
                FFT_SRM_W(DAT + i, DAT + len * 2 + i,                       \
                    DAT + len * 4 + i, DAT + len * 6 + i,                   \
                    LUT + len * 2 + i);                                     \
                                                                            \
        fft_ ## nm2(DAT + len * 4, LUT);                                    \
        fft_ ## nm2(DAT + len * 6, LUT);                                    \
        fft_ ## nm1(DAT, LUT);                                              \
}

FFT_FUNC(5, 4, 3, 8)
FFT_FUNC(6, 5, 4, 16)
FFT_FUNC(7, 6, 5, 32)
FFT_FUNC(8, 7, 6, 64)
FFT_FUNC(9, 8, 7, 128)
FFT_FUNC(10, 9, 8, 256)
FFT_FUNC(11, 10, 9, 512)
FFT_FUNC(12, 11, 10, 1024)
FFT_FUNC(13, 12, 11, 2048)
FFT_FUNC(14, 13, 12, 4096)
FFT_FUNC(15, 14, 13, 8192)
FFT_FUNC(16, 15, 14, 16384)
FFT_FUNC(17, 16, 15, 32768)
FFT_FUNC(18, 17, 16, 65536)
FFT_FUNC(19, 18, 17, 131072)
FFT_FUNC(20, 19, 18, 262144)
FFT_FUNC(21, 20, 19, 524288)
FFT_FUNC(22, 21, 20, 1048576)
FFT_FUNC(23, 22, 21, 2097152)
FFT_FUNC(24, 23, 22, 4194304)
```

```
FFT_FUNC(25, 24, 23, 8388608)
FFT_FUNC(26, 25, 24, 16777216)
FFT_FUNC(27, 26, 25, 33554432)
FFT_FUNC(28, 27, 26, 67108864)
FFT_FUNC(29, 28, 27, 134217728)

static void (* fft_list[])(double * __restrict, double * __restrict) = {
        fft_0, fft_1, fft_2, fft_3, fft_4, fft_5, fft_6, fft_7,
        fft_8, fft_9, fft_10, fft_11, fft_12, fft_13, fft_14,
        fft_15, fft_16, fft_17, fft_18, fft_19, fft_20, fft_21,
        fft_22, fft_23, fft_24, fft_25, fft_26, fft_27, fft_28,
        fft_29
};

void tricl_fft_fft(double * __restrict DAT, int size, double * __restrict LUT)
{

        assert(0 <= size &&
            size <= (int)(sizeof(fft_list) / sizeof(fft_list[0])));

        fft_list[size](DAT, LUT);
}
```

2.4 Small IFFTs

Up to length 2^4, we provide hard-coded inverse FFTs:

```
static void ifft_0(__unused double * __restrict DAT,
    __unused double * __restrict LUT)
{

        /* Do nothing */
}

static void ifft_1(double * __restrict DAT,
    __unused double * __restrict LUT)
{

        FFT_PM(DAT, DAT + 2);
}

static void ifft_2(double * __restrict DAT,
    __unused double * __restrict LUT)
{
```

```
        FFT_PM(DAT, DAT + 2);
        IFFT_SRM(DAT, DAT + 2, DAT + 4, DAT + 6);
}

static void ifft_3(double * __restrict DAT,
    __unused double * __restrict LUT)
{

        FFT_PM(DAT, DAT + 2);
        IFFT_SRM(DAT, DAT + 2, DAT + 4, DAT + 6);
        FFT_PM(DAT + 8, DAT + 10);
        FFT_PM(DAT + 12, DAT + 14);
        IFFT_SRM(DAT, DAT + 4, DAT + 8, DAT + 12);
        IFFT_SRM_PI_4(DAT + 2, DAT + 6, DAT + 10, DAT + 14);
}

static void ifft_4(double * __restrict DAT,
    double * __restrict LUT)
{

        FFT_PM(DAT, DAT + 2);
        IFFT_SRM(DAT, DAT + 2, DAT + 4, DAT + 6);
        FFT_PM(DAT + 8, DAT + 10);
        FFT_PM(DAT + 12, DAT + 14);
        IFFT_SRM(DAT, DAT + 4, DAT + 8, DAT + 12);
        IFFT_SRM_PI_4(DAT + 2, DAT + 6, DAT + 10, DAT + 14);
        FFT_PM(DAT + 16, DAT + 18);
        IFFT_SRM(DAT + 16, DAT + 18, DAT + 20, DAT + 22);
        FFT_PM(DAT + 24, DAT + 26);
        IFFT_SRM(DAT + 24, DAT + 26, DAT + 28, DAT + 30);
        IFFT_SRM(DAT, DAT + 8, DAT + 16, DAT + 24);
        IFFT_SRM_W(DAT + 2, DAT + 10, DAT + 18, DAT + 26, LUT + 10);
        IFFT_SRM_PI_4(DAT + 4, DAT + 12, DAT + 20, DAT + 28);
        IFFT_SRM_W(DAT + 6, DAT + 14, DAT + 22, DAT + 30, LUT + 14);
}
```

2.5 Large IFFTs

As with FFTs, we define a generic split-radix inverse FFT macro for sizes 2^5 up to 2^{29}.

```
#define IFFT_FUNC(n, nm1, nm2, len)                                     \
static void ifft_ ## n(double * __restrict DAT,                         \
    double * __restrict LUT)                                            \
{                                                                       \
        uint32_t i;                                                     \
```

```
                                                                    \
        ifft_ ## nm1(DAT, LUT);                                     \
        ifft_ ## nm2(DAT + len * 4, LUT);                           \
        ifft_ ## nm2(DAT + len * 6, LUT);                           \
                                                                    \
        IFFT_SRM(DAT, DAT + len * 2, DAT + len * 4, DAT + len * 6); \
        for (i = 2; i < 2 * len; i += 2)                            \
                IFFT_SRM_W(DAT + i, DAT + len * 2 + i,              \
                    DAT + len * 4 + i, DAT + len * 6 + i,           \
                    LUT + len * 2 + i);                             \
}

IFFT_FUNC(5, 4, 3, 8)
IFFT_FUNC(6, 5, 4, 16)
IFFT_FUNC(7, 6, 5, 32)
IFFT_FUNC(8, 7, 6, 64)
IFFT_FUNC(9, 8, 7, 128)
IFFT_FUNC(10, 9, 8, 256)
IFFT_FUNC(11, 10, 9, 512)
IFFT_FUNC(12, 11, 10, 1024)
IFFT_FUNC(13, 12, 11, 2048)
IFFT_FUNC(14, 13, 12, 4096)
IFFT_FUNC(15, 14, 13, 8192)
IFFT_FUNC(16, 15, 14, 16384)
IFFT_FUNC(17, 16, 15, 32768)
IFFT_FUNC(18, 17, 16, 65536)
IFFT_FUNC(19, 18, 17, 131072)
IFFT_FUNC(20, 19, 18, 262144)
IFFT_FUNC(21, 20, 19, 524288)
IFFT_FUNC(22, 21, 20, 1048576)
IFFT_FUNC(23, 22, 21, 2097152)
IFFT_FUNC(24, 23, 22, 4194304)
IFFT_FUNC(25, 24, 23, 8388608)
IFFT_FUNC(26, 25, 24, 16777216)
IFFT_FUNC(27, 26, 25, 33554432)
IFFT_FUNC(28, 27, 26, 67108864)
IFFT_FUNC(29, 28, 27, 134217728)

static void (* ifft_list[])(double * __restrict, double * __restrict) = {
        ifft_0, ifft_1, ifft_2, ifft_3, ifft_4, ifft_5, ifft_6, ifft_7,
        ifft_8, ifft_9, ifft_10, ifft_11, ifft_12, ifft_13, ifft_14,
        ifft_15, ifft_16, ifft_17, ifft_18, ifft_19, ifft_20, ifft_21,
        ifft_22, ifft_23, ifft_24, ifft_25, ifft_26, ifft_27, ifft_28,
        ifft_29
};
```

```
void tricl_fft_ifft(double * __restrict DAT, int size,
    double * __restrict LUT)
{

        assert(0 <= size &&
            size <= (int)(sizeof(ifft_list) / sizeof(ifft_list[0])));

        ifft_list[size](DAT, LUT);
}
```

References

[1] D.J. Bernstein, *djbfft*, http://cr.yp.to/djbfft.html

A Copyright

```
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
* LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
* OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
* SUCH DAMAGE.
```