1. Introduction.

FOURIER

Fourier Transform Utilities

John Walker

This program provides a general purpose fast Fourier transform, its inverse, and computation of a power spectrum from a Fourier transform.

⟨ fourier_test.c 1 ⟩ ≡
```
#define REVDATE "5th February 2002"
```
See also section 17.

2. Program global context.

```
#include "config.h"      /* System-dependent configuration */
  ⟨ Preprocessor definitions ⟩
  ⟨ Application include files 4 ⟩
  ⟨ Class implementations 12 ⟩
```

3. We export the class definitions for this package in the external file `fourier.h` that programs which use this library may include.

```
⟨ fourier.h 3 ⟩ ≡
#ifndef FOURIER_HEADER_DEFINES
#define FOURIER_HEADER_DEFINES
#include <math.h>      /* Make sure math.h is available */
#include <iostream>
#include <exception>
#include <stdexcept>
#include <string>
#include <vector>
#include <algorithm>
  using namespace std;
#include <assert.h>
  ⟨ Class definitions 6 ⟩
#endif
```

4. The following include files provide access to external components of the program not defined herein.

```
⟨ Application include files 4 ⟩ ≡
#include "fourier.h"      /* Class definitions for this package */
```

This code is used in section 2.

5. Fourier transform.

The *fourierTransform* class provides a general-purpose fast Fourier transform, both forward and inverse. It operates on arrays of complex numbers (type *fourierTransform*::*complex*) which must be a power of two in length. Once the transform is initialised for a given data vector size, any number of vectors of that size may be transformed.

6. Begin by declaring a table of powers of two which will come handy in the transformation code.

```
⟨ Class definitions 6 ⟩ ≡
  static const int pwr_two[] = {1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768,
      65536, 131072, 262144, 524288, 1048576, 2097152, 4194304, 8388608, 16777216, 33554432, 67108864,
      134217728, 268435456, 536870912, 1073741824};
```

See also sections 7, 8, 9, 10, and 11.

This code is used in section 3.

7. We begin the declaration of the *fourierTransform* by declaring its *complex* type, which users of the class refer to as *fourierTransform*::*complex*.

```
⟨ Class definitions 6 ⟩ +≡
  class fourierTransform {
public:
  typedef struct {
    double r, i;
  } complex;
```

8. Private instance variables hold the sin and cos tables we use during the transforms and the size of the data vector given as $\ln_2 n$ where n is the number of **complex** numbers in the vector.

```
⟨ Class definitions 6 ⟩ +≡
private:
  complex *c;
  int ln;
  void buildtable(void);
```

9. We provide the following query functions to return properties of the transform.

```
⟨ Class definitions 6 ⟩ +≡
  unsigned int getTransformSize(void)
  {
    return pwr_two[ln];
  }
  unsigned int getTransformSizePowerOfTwo(void)
  {
    return static_cast⟨unsigned int⟩(ln);
  }
```

10. If you know the size of the data table in advance, you may simply construct the **fourierTransform** with its size as the argument. Otherwise, you can construct the transform and then set the size later with *setTransformSize*. You can change the transform size at any time by calling *setTransformSize*. The sine and cosine table is automatically released by the destructor.

```
⟨ Class definitions 6 ⟩ +≡
public:
  fourierTransform(unsigned int nsamples)
  : c(Λ) {
    setTransformSize(nsamples);
  }
  fourierTransform( )
  : c(Λ) { }
  void setTransformSize(unsigned int nsamples)
  {
    int p = 1;
    for (ln = 0; p < nsamples; ln++) {
      p ≪= 1;
    }
    if (pwr_two[ln] ≠ nsamples) {
      throw (out_of_range("fourierTransform::setTransformSize: data vector size is not\
          a power of two"));
    }
    buildtable( );
  }
  ∼fourierTransform( )
  {
    if (c ≠ Λ) {
      delete c;
    }
  }
```

11. We wind up the class definitions by declaring the functions which actually perform the transformation. The low-level *bitreverse*, *fft*, and *ifft* methods leave the bit reversing up to the caller, which can save time if you're twiddling things at low level. The high-level *transform* and *inverse_transform* methods do the bit reversing internally.

```
⟨ Class definitions 6 ⟩ +≡
  void bitreverse(complex *x);
  void fft(complex *x);
  void ifft(complex *x);
  void transform(complex *x)
  {
    bitreverse(x);
    fft(x);
  }
  void inverse_transform(complex *x)
  {
    bitreverse(x);
    ifft(x);
  }
  void power_spectrum(const complex *x, double *p); } ;
```

12. The private *buildtable* method generates the sine and cosine tables used by the transformation. These are kept in the real and imaginary parts of an array of **complex** half the size of the data table.

```
⟨ Class implementations 12 ⟩ ≡
  void fourierTransform::buildtable(void)
  {
    int n, halfn, i;
    double theta;
    static const double PI2 = 6.28318530717958647692528 7;
    n = pwr_two[ln];
    halfn = pwr_two[ln − 1];
    if (c ≠ Λ) {
      delete c;
    }
    c = new complex[halfn];
    for (i = 0; i < halfn; i++) {
      theta = (PI2 ∗ i)/n;
      c[i].r = cos(theta);
      c[i].i = −sin(theta);
    }
  }
```

See also sections 13, 14, 15, and 16.

This code is used in section 2.

13. If you're using the low level *fft* and *ifft* methods, you'll need to call *bitreverse* to re-order the input vector before calling them.

```
⟨ Class implementations 12 ⟩ +≡
  void fourierTransform::bitreverse(complex *x)
  {
    int i, j, k, n;
    complex hold;
    n = pwr_two[ln];
    k = 0;
    j = 0;
    while (true) {
      if (k > j) {
        hold.r = x[j].r;
        hold.i = x[j].i;
        x[j].r = x[k].r;
        x[j].i = x[k].i;
        x[k].r = hold.r;
        x[k].i = hold.i;
      }
      j++;
      if (j ≡ n) {
        break;
      }
            /* k = k + 1, (k is n bits long and bit reversed). */
      i = ln - 1;
      while (i ≥ 0) {
        if (k & pwr_two[i]) {
          k = k - pwr_two[i];
        }
        else {
          break;
        }
        i--;
      }
      k = k + pwr_two[i];
    }
  }
```

14. Fourier transform. You must call *bitreverse* first, or use the high level *transform* method which does this for you.

```
⟨ Class implementations 12 ⟩ +≡
  void fourierTransform::fft(complex *x)
  {
    unsigned int stuckbit, stuckbitcomplement, halfn, t, b;
    unsigned int shft, msk, wt;
    unsigned int c0, c1;
    unsigned int n;
    double qr, qi, xtr, xti, xbr, xbi, cwtr, cwti;
    assert(c ≠ Λ);
    n = pwr_two[ln];
    msk = n − 1;
    halfn = pwr_two[ln − 1];
    shft = ln − 1;
    stuckbit = 1;
    c0 = ln;
    while (c0−−) {
      t = 0;
      stuckbitcomplement = ∼stuckbit;
      c1 = halfn;
      while (c1−−) {
        wt = (t ≪ shft) & msk;
        b = t + stuckbit;
        xbr = x[b].r;
        xbi = x[b].i;
        cwtr = c[wt].r;
        cwti = c[wt].i;
        qr = xbr ∗ cwtr − xbi ∗ cwti;
        qi = xbr ∗ cwti + xbi ∗ cwtr;
        xtr = x[t].r;
        xti = x[t].i;
        x[b].r = xtr − qr;
        x[b].i = xti − qi;
        x[t].r = xtr + qr;
        x[t].i = xti + qi;
        t = (b + 1) & stuckbitcomplement;
      }
      stuckbit ≪= 1;
      shft −= 1;
    }
  }
```

15. Inverse Fourier transform. You must call *bitreverse* first, or use the high level *inverse_transform* method which does this for you. Note that values in the inverse transform will be scaled by the size of the data set and should be divided by its size to recover the values of the original transform.

```
⟨ Class implementations 12 ⟩ +≡
  void fourierTransform::ifft(complex *x)
  {
    unsigned int stuckbit, stuckbitcomplement, halfn, t, b;
    unsigned int shft, msk, wt;
    unsigned int c0, c1;
    unsigned int n;
    double qr, qi;
    assert(c ≠ Λ);
    n = pwr_two[ln];
    msk = n − 1;
    halfn = pwr_two[ln − 1];
    shft = ln − 1;
    stuckbit = 1;
    c0 = ln;
    while (c0−−) {
      t = 0;
      stuckbitcomplement = ∼stuckbit;
      c1 = halfn;
      while (c1−−) {
        b = t + stuckbit;
        wt = (t ≪ shft) & msk;
        qr = x[b].r ∗ c[wt].r + x[b].i ∗ c[wt].i;
        qi = −(x[b].r ∗ c[wt].i) + x[b].i ∗ c[wt].r;
        x[b].r = x[t].r − qr;
        x[b].i = x[t].i − qi;
        x[t].r += qr;
        x[t].i += qi;
        t = (b + 1) & stuckbitcomplement;
      }
      stuckbit ≪= 1;
      shft −= 1;
    }
  }
```

16. Compute the power spectrum from a Fourier transform of a data set. The power spectrum is just the the absolute values of the complex Fourier transform vector.

```
⟨ Class implementations 12 ⟩ +≡
  void fourierTransform::power_spectrum(const complex *x, double *p)
  {
    unsigned int j;
    for (j = 0; j < pwr_two[ln]; j++) {
#define sqr(x) ((x) ∗ (x))
      p[j] = sqrt((sqr(x[j].r) + sqr(x[j].i))/pwr_two[ln]);
#undef sqr
    }
  }
```

17. Test program.

```
⟨ fourier_test.c 1 ⟩ +≡
  ⟨ Test program include files 20 ⟩;
  ⟨ Show how to call test program 19 ⟩;
#define FFTPOW 16
#define FFTSIZE (1 ≪ FFTPOW)
  int main(int argc, char *argv[])
  {
    extern char *optarg;      /* Imported from getopt */
    extern int optind;
    int opt;
    try {
      ⟨ Process command-line options 18 ⟩;
      fourierTransform ft(FFTSIZE);
      fourierTransform::complex a[FFTSIZE], b[FFTSIZE], c[FFTSIZE];
        /* Synthesise a test function from superposed sine curves. */
#define P1 773.0
#define A1 1
#define P2 593.0
#define A2 0.5
#define P3 191.0
#define A3 1
      for (int i = 0; i < FFTSIZE; i++) {
        a[i].r = sin(static_cast⟨double⟩(i)/P1) * A1;
#ifdef P2
        a[i].r += sin(static_cast⟨double⟩(i)/P2) * A2;
#endif
#ifdef P3
        a[i].r += sin(static_cast⟨double⟩(i)/P3) * A3;
#endif
        a[i].i = 0;
      }
#ifdef CRAPTASTROPHE
      b = a;
#else
      memcpy(b, a, sizeof b);
#endif
      ft.transform(b);
#ifdef CRAPTASTROPHE
      c = b;
#else
      memcpy(c, b, sizeof c);
#endif
      ft.inverse_transform(c);
      string fileName = "fourier";
      ofstream gp((fileName + ".gp").c_str( )), dat((fileName + ".dat").c_str( ));
        /* Data columns: 1: Index 2: Input function 3: Power spectrum 4: Inverse FFT results */
      double pspec[FFTSIZE];
      ft.power_spectrum(b, pspec);
#define PLOTSIZE 1000
```

```
    for (int j = 0; j < PLOTSIZE; j++) {
      c[j].r /= PLOTSIZE;
      c[j].i /= PLOTSIZE;
      dat << j << " " << a[j].r << " " << pspec[j] << " " << c[j].r << endl;
    }
    gp << "set term pbm small color" << endl;
#define PLOT_BOTH
#ifdef PLOT_INPUT
    gp << "plot \"" << fileName << ".dat\" using 1:2 with lines" << endl;
#endif
#ifdef PLOT_INVERSE
    gp << "plot \"" << fileName << ".dat\" using 1:4 with lines" << endl;
#endif
#ifdef PLOT_BOTH
    gp << "plot \"" << fileName << ".dat\" using 1:2 with lines, ";
    gp << "\"" << fileName << ".dat\" using 1:4 with lines" << endl;
#endif
#ifdef PLOT_POWER
    gp << "plot \"" << fileName << ".dat\" using 1:3 with lines" << endl;
#endif
    string command("gnuplot ");
    command += fileName + ".gp | ppmtogif > " + fileName + ".gif";
#ifdef DEV_PLOT_DEBUG
    cout << command << endl;
#else
    command += " 2>/dev/null";
#endif
    gp.close();
    dat.close();
    system(command.c_str());
#ifndef DEV_PLOT_DEBUG      /* Delete the temporary files used to create the plot */
    remove((fileName + ".gp").c_str());
    remove((fileName + ".dat").c_str());
#endif
  }
  catch(exception &e)
  {
    cout << "Blooie!!!  Exception popped: " << e.what() << endl;
#ifndef CORE_DUMP
#ifdef STACK_TRACE
    char s[160];
    sprintf(s,
      "/bin/echo 'where\nq' >/tmp/gdbcmd ; gdb  -batch --command ""/tmp/gdbcmd %s %d",
      argv[0], getpid());
    system(s);
    sleep(5);
#endif
    throw;      /* Re-throw exception to dump core */
#endif
  }
  return 0;
```

```
}
```

18. We use *getopt* to process command line options. This permits aggregation of options without arguments and both `-d`*arg* and `-d` *arg* syntax.

```
⟨ Process command-line options 18 ⟩ ≡
  while ((opt = getopt(argc, argv, "nu-:")) ≠ −1) {
    switch (opt) {
    case 'u':     /* -u Print how-to-call information */
      case '?': usage();
      return 0;
    case '-':     /* -- Extended options */
      switch (optarg[0]) {
      case 'c':     /* --copyright */
        cout ≪ "This␣program␣is␣in␣the␣public␣domain.\n";
        return 0;
      case 'h':     /* --help */
        usage();
        return 0;
      case 'v':     /* --version */
        cout ≪ PRODUCT ≪ "␣" ≪ VERSION ≪ "\n";
        cout ≪ "Last␣revised:␣" ≪ REVDATE ≪ "\n";
        cout ≪ "The␣latest␣version␣is␣always␣available\n";
        cout ≪ "at␣http://www.fourmilab.ch/eggtools/eggshell\n";
        return 0;
      }
    }
  }
```

This code is used in section 17.

19. Procedure *usage* prints how-to-call information.

```
⟨ Show how to call test program 19 ⟩ ≡
  static void usage(void)
  {
    cout ≪ PRODUCT ≪ "␣␣--␣␣Analyse␣eggsummary␣files.␣␣Call:\n";
    cout ≪ "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣" ≪ PRODUCT ≪ "␣[options]␣[infile]␣[outfile]\n";
    cout ≪ "\n";
    cout ≪ "Options:\n";
    cout ≪ "␣␣␣␣␣␣␣␣␣␣␣--copyright␣␣␣␣␣␣␣␣Print␣copyright␣information\n";
    cout ≪ "␣␣␣␣␣␣␣␣␣␣␣-u,␣--help␣␣␣␣␣␣␣␣Print␣this␣message\n";
    cout ≪ "␣␣␣␣␣␣␣␣␣␣␣--version␣␣␣␣␣␣␣␣␣Print␣version␣number\n";
    cout ≪ "\n";
    cout ≪ "by␣John␣Walker\n";
    cout ≪ "http://www.fourmilab.ch/\n";
  }
```

This code is used in section 17.

20. We need the following definitions to compile the test program.

⟨ Test program include files 20 ⟩ ≡

```
#include "config.h"      /* Our configuration */      /* C++ include files */
#include <iostream>
#include <fstream>
#include <exception>
#include <stdexcept>
#include <string>
  using namespace std;
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#ifdef HAVE_GETOPT
#ifdef HAVE_UNISTD_H
#include <unistd.h>
#endif
#else
#include "getopt.h"      /* No system getopt—use our own */
#endif
#include "fourier.h"      /* Class definitions for this package */
```

This code is used in section 17.

21. Index. The following is a cross-reference table for `fourier`. Single-character identifiers are not indexed, nor are reserved words. Underlined entries indicate where an identifier was declared.

⟨ Application include files 4 ⟩ Used in section 2.
⟨ Class definitions 6, 7, 8, 9, 10, 11 ⟩ Used in section 3.
⟨ Class implementations 12, 13, 14, 15, 16 ⟩ Used in section 2.
⟨ Process command-line options 18 ⟩ Used in section 17.
⟨ Show how to call test program 19 ⟩ Used in section 17.
⟨ Test program include files 20 ⟩ Used in section 17.
⟨ `fourier.h` 3 ⟩
⟨ `fourier_test.c` 1, 17 ⟩

FOURIER