

admindojo*tutorials and apps*

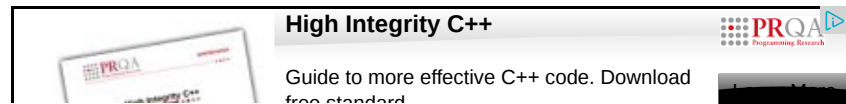
Discrete Fourier Transform in C++ with fftw

Posted on [March 12, 2011](#) by [fmauro](#)

Like

11

Tweet

[+](#) Share / Save [f](#) [t](#) [i](#)

This tutorial will give you a short introduction to using libfftw in C++ under Linux.

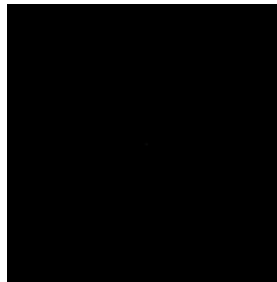
The discrete Fourier-Transform (DFT) applied on an Image will translate the Image-Pixels into frequency and magnitude components.

Some knowledge of complex numbers is required to understand the transform itself but we'll keep the math to the bare minimum.

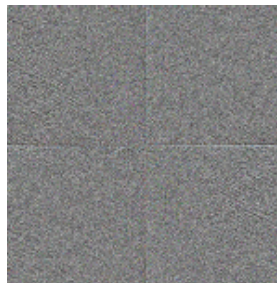
The final result should yield a set of images that describe the original image and should look like this:



Input image



Magnitude image



Phase image

The magnitude appears to be black but isn't. To make the information visible we scale the image logarithmically.



Magnitude scaled to
visibility

We can recombine the magnitude and phase information with the inverse DFT and get:



Output image

As you can see in the image above, the inverse DFT of the magnitude and phase images produces the original image with barely no visible changes.

If you play around with the phase/magnitude you can alter the output image. Keep in mind that we will be saving

16bit images and most editing software under linux only supports 8bit color-spaces.

Re-saving the magnitude or phase image in 8bit color-space and performing the iDFT will return a very bad result:



Output created from 8bit
sources

Step 1: Setup

To open images of different filetypes we will use libmagick and of course libfftw for the transform itself.

To be able to compile your program you'll need (apart from your common g++ compiler) to download libfftw, libmagick++ and their respective development files.

If you're running Debian/Ubuntu you can get those from your repos by typing:

```
1 | sudo apt-get install libmagick++3 imagemagick libmagick++-dev libfftw3-3 libfftw3-dev
```

or get it on the web:

[libfftw](#)

[Imagemagick](#)

Create a folder and source structure:

```
1 | mkdir fourier
2 | touch fourier/Makefile
3 | touch fourier/main.cpp
```

The makefile to our little project will enable us to use `make` instead of having to type the whole `g++` compile-command.

Makefile

```
1 | CXXFLAGS=`Magick++-config --cppflags --cxxflags --ldflags --libs` `pkg-config
2 | CXX=g++
3 |
4 | all: fourier
5 |
6 | fourier: main.o
7 |     $(CXX) -o $@ $(CXXFLAGS) $<
8 |
9 | main.o: main.cpp
10 |     $(CXX) -c -o $@ $(CXXFLAGS) $<
11 |
12 | clean:
13 |     rm main.o
14 |
15 | install:
16 |     cp fourier /usr/bin/
17 |
18 | uninstall:
19 |     rm /usr/bin/fourier
```

Step 2: Programming

We will keep the code very basic and simple, when implementing the code in your own applications you might want to add some sort of exception handling and comments.

We start off with our includes:

main.cpp

```
1  #include <fftw3.h>
2  #include <Magick++.h>
3  #include <math.h>
4  #include <complex>
5  #include <iostream>
6  #include <string>
7  using namespace Magick;
8  using namespace std;
```

Most of these should be known to you, we add the libfftw and magick++ headers to get the functionality we need and also the `std::complex` class to perform some basic operations on complex numbers.

We will want our application to act in the following way:

```
fourier in outMag outPhase
```

for the forward and

```
fourier -ift inMag inPhase out
```

for the inverse transform.

So in our `main.cpp` we add

`main.cpp`

```
1  int main(int argc, char** argv)
2  {
3      if(argc < 4) {
4          cout << "usage:\t" << argv[0] << " in outMag outPhase" << endl
5              << "\t" << argv[0] << " -ift inMag inPhase out" << endl;
6          return 0;
7      }
8
9      string arg = argv[1];
10
11     if(arg != "-ift") {
12         // FORWARD TRANSFORM
13     }
14     else {
15         // BACKWARD TRANSFORM
16     }
17     return 0;
18 }
```

We will open the images with `imagemagick`. You can do so in C++ directly in the `Magick::Image` constructor.

To get access to the underlying pixels we use this method:

Using ImageMagick

```
1  Image img("myfilename");
2
3  // lock image for modification
4  img.modifyImage();
5  Pixels pixelCache(img);
6  PixelPacket* pixels;
7
8  // get desired area in image
9  pixels = pixelCache.get(0, 0, sizeX, sizeY);
10
11 // read/write pixel (12,3)
12 *(pixel + 12 + (3 * sizeX)) = Color("Blue");
13
14 // write changes
15 pixelCache.sync()
```

Now that we know how to access image pixels, let's write the prototypes for the forward and backward Fourier-Transform.

Right above the `main` method add

main.cpp

```
1  void fft(int squareSize, PixelPacket* pixels, PixelPacket* outMag, PixelPacket*
2  {
3  }
4
5  void ift(int squareSize, PixelPacket* inMag, PixelPacket* inPhase, PixelPacket*
6  {
7  }
```


Our prototypes describe methods that will take pointers to a `PixelPacket` array that represent our images.

Also we will only work with square Images (we will adjust them to our needs in `main`)

In the completed `main` method we prepare the images and call the respective method:

`main.cpp`

```
1  int main(int argc, char** argv)
2  {
3      if(argc < 4) {
4          cout << "usage:\t" << argv[0] << " in outMag outPhase" << endl
5              << "\t" << argv[0] << " -ift inMag inPhase out" << endl;
6          return 0;
7      }
8
9      string arg = argv[1];
10
11     if(arg != "-ift") {
12         // FORWARD FOURIER TRANSFORM
13         Image img(argv[1]);
14
15         // get the length of the longer side of the image
16         int squareSize = img.columns() < img.rows() ? img.rows() : img.columns();
17
18         // the geometry of our padded image
19         Geometry padded(squareSize, squareSize);
20         padded.aspect(true);
21
22         // make image square
23         img.extent(padded);
24
25         // create templates for magnitude and phase
26         Image mag(Geometry(squareSize, squareSize), "Black");
```

```
27     Image phase(Geometry(squareSize, squareSize), "Black");
28
29     // get image pixels
30     img.modifyImage();
31     Pixels pixelCache(img);
32     PixelPacket* pixels;
33     pixels = pixelCache.get(0, 0, squareSize, squareSize);
34
35     // get magnitude pixels
36     mag.modifyImage();
37     Pixels pixelCacheMag(mag);
38     PixelPacket* pixelsMag;
39     pixelsMag = pixelCacheMag.get(0, 0, squareSize, squareSize);
40
41     // get phase pixels
42     phase.modifyImage();
43     Pixels pixelCachePhase(phase);
44     PixelPacket* pixelsPhase;
45     pixelsPhase = pixelCachePhase.get(0, 0, squareSize, squareSize);
46
47     // perform fft
48     fft(squareSize, pixels, pixelsMag, pixelsPhase);
49
50     // write changes
51     pixelCache.sync();
52     pixelCacheMag.sync();
53     pixelCachePhase.sync();
54
55     // save files
56     mag.write(argv[2]);
57     phase.write(argv[3]);
58 }
59 else {
```

```
60         // BACKWARD FOURIER TRANSFORM
61     }
62     return 0;
63 }
```

Both our functions still do nothing with the provided pixels, so let's take a quick look at the libfftw api so we know how to use it.

libfftw provides several methods to do FFT on arrays of complex or real values and in multiple dimensions which we will not need to use. Our methods will only do transforms from real (although stored in a complex array) to complex and back. To do a transform libfftw provides a so called `plan` on which the operations are executed on. On creation one must supply the plan with an input and output array which have to be allocated first.

Using fftw to perform FFT

```
1  fftw_plan plan;
2  fftw_complex *in, *out;
3
4  // allocate memory for input and output arrays
5  in = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * sizeX * sizeY);
6  out = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * sizeX * sizeY);
7
8  // create the plan
9  plan = fftw_plan_dft_2d(sizeX, sizeY, in, out, FFTW_FORWARD, FFTW_ESTIMATE);
10
11 // fill the array
12 for(int i = 0, i < sizeY; i++)
13     for(int j = 0; j < sizeX; j++)
14         in[j + (sizeX * i)][0] = randomNumber; // in[][0] is the real part of
15
16 // execute the specified transform
```

```
17  fftw_execute(plan);
18
19  // use output
20  [...]
21
22  // free allocated memory
23  fftw_destroy_plan(plan);
24  fftw_free(out);
25  fftw_free(in);
```

We have to create a plan for every color-channel in the image (we will only cover RGB) and create input and output-arrays respectively.

Let's add just that to our `fft` method:

`main.cpp`

```
1  void fft(int squareSize, PixelPacket* pixels, PixelPacket* outMag, PixelPacket* outPhase)
2  {
3      fftw_plan planR, planG, planB;
4      fftw_complex *inR, *inG, *inB, *outR, *outG, *outB;
5
6      // allocate input arrays
7      inR = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
8      inG = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
9      inB = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
10
11     // allocate output arrays
12     outR = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
13     outG = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
14     outB = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
15
16     // create plans
```

```
17     planR = fftw_plan_dft_2d(squareSize, squareSize, inR, outR, FFTW_FORWARD,
18     planG = fftw_plan_dft_2d(squareSize, squareSize, inG, outG, FFTW_FORWARD,
19     planB = fftw_plan_dft_2d(squareSize, squareSize, inB, outB, FFTW_FORWARD,
20
21     // TODO: assign color-values to input arrays
22
23     // perform FORWARD fft
24     fftw_execute(planR);
25     fftw_execute(planG);
26     fftw_execute(planB);
27
28     // TODO: calculate output mag/phase
29
30     // free memory
31     fftw_destroy_plan(planR);
32     fftw_destroy_plan(planG);
33     fftw_destroy_plan(planB);
34     fftw_free(inR); fftw_free(outR);
35     fftw_free(inG); fftw_free(outG);
36     fftw_free(inB); fftw_free(outB);
37 }
```

Putting some values to perform the fft on would be nice. We will use the color-values as they are and put them into the input arrays for each channel.

main.cpp

```
1  void fft(int squareSize, PixelPacket* pixels, PixelPacket* outMag, PixelPacke
2  {
3      fftw_plan planR, planG, planB;
4      fftw_complex *inR, *inG, *inB, *outR, *outG, *outB;
5
```

```
6 // allocate input arrays
7 inR = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
8 inG = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
9 inB = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
10
11 // allocate output arrays
12 outR = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
13 outG = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
14 outB = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
15
16 // create plans
17 planR = fftw_plan_dft_2d(squareSize, squareSize, inR, outR, FFTW_FORWARD, FFTW_ESTIMATE);
18 planG = fftw_plan_dft_2d(squareSize, squareSize, inG, outG, FFTW_FORWARD, FFTW_ESTIMATE);
19 planB = fftw_plan_dft_2d(squareSize, squareSize, inB, outB, FFTW_FORWARD, FFTW_ESTIMATE);
20
21 // assign values to real parts (values between 0 and MaxRGB)
22 for(int i = 0; i < squareSize * squareSize; i++) {
23     PixelPacket current = *(pixels + i);
24     double red = current.red;
25     double green = current.green;
26     double blue = current.blue;
27
28     // save as real numbers
29     inR[i][0] = red;
30     inG[i][0] = green;
31     inB[i][0] = blue;
32 }
33
34 // perform FORWARD fft
35 fftw_execute(planR);
36 fftw_execute(planG);
37 fftw_execute(planB);
38
```

```
39     // TODO: calculate output mag/phase
40
41     // free memory
42     fftw_destroy_plan(planR);
43     fftw_destroy_plan(planG);
44     fftw_destroy_plan(planB);
45     fftw_free(inR); fftw_free(outR);
46     fftw_free(inG); fftw_free(outG);
47     fftw_free(inB); fftw_free(outB);
48 }
```

A tiny bit of math

Now that the FFT has been performed we have an output array of complex numbers for each channel (RGB). To get the information we need, namely magnitude and phase, we will perform some simple math on these numbers.

The magnitude M is calculated as the square root of the scalar product of the components of the complex result z :

$$z = a + ib$$

$$M = |z| = \sqrt{a^2 + b^2}$$

The phase φ is calculated with the `arg` operator. Both definitions can be found in the [polar coordinate system wiki](#).

$$\varphi = \arg(z)$$

To calculate the phase we will use the implementation of `std::arg` in the `complex` class.

As we can only save positive values to files (from 0 to MaxRGB) we will have to shift the phase from $[-\pi, \pi]$ to $[0, 2\pi]$ followed by a scaling to $[0, \text{MaxRGB}]$ and scale/shift it back when doing the inverse DFT. Also libfftw for performance reasons scales the output by $(\text{sizeX} * \text{sizeY})$ which we will correct before calculating phase and magnitude by dividing it by just that.

The code to do all this stuff:

main.cpp

```
1  void fft(int squareSize, PixelPacket* pixels, PixelPacket* outMag, PixelPacket* outPhase)
2  {
3      fftw_plan planR, planG, planB;
4      fftw_complex *inR, *inG, *inB, *outR, *outG, *outB;
5
6      // allocate input arrays
7      inR = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
8      inG = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
9      inB = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
10
11     // allocate output arrays
12     outR = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
13     outG = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
14     outB = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
15
16     // create plans
17     planR = fftw_plan_dft_2d(squareSize, squareSize, inR, outR, FFTW_FORWARD, FFTW_ESTIMATE);
18     planG = fftw_plan_dft_2d(squareSize, squareSize, inG, outG, FFTW_FORWARD, FFTW_ESTIMATE);
19     planB = fftw_plan_dft_2d(squareSize, squareSize, inB, outB, FFTW_FORWARD, FFTW_ESTIMATE);
20
21     // assign values to real parts (values between 0 and MaxRGB)
22     for(int i = 0; i < squareSize * squareSize; i++) {
23         PixelPacket current = *(pixels + i);
24         double red = current.red;
25         double green = current.green;
26         double blue = current.blue;
27
28         // save as real numbers
29         inR[i][0] = red;
```



```
30         inG[i][0] = green;
31         inB[i][0] = blue;
32     }
33
34     // perform FORWARD fft
35     fftw_execute(planR);
36     fftw_execute(planG);
37     fftw_execute(planB);
38
39     // transform imaginary number to phase and magnitude and save to output
40     for(int i = 0; i < squareSize * squareSize; i++) {
41         // normalize values
42         double realR = outR[i][0] / (double)(squareSize * squareSize);
43         double imagR = outR[i][1] / (double)(squareSize * squareSize);
44
45         double realG = outG[i][0] / (double)(squareSize * squareSize);
46         double imagG = outG[i][1] / (double)(squareSize * squareSize);
47
48         double realB = outB[i][0] / (double)(squareSize * squareSize);
49         double imagB = outB[i][1] / (double)(squareSize * squareSize);
50
51         // magnitude
52         double magR = sqrt((realR * realR) + (imagR * imagR));
53         double magG = sqrt((realG * realG) + (imagG * imagG));
54         double magB = sqrt((realB * realB) + (imagB * imagB));
55
56         // write to output
57         (*(outMag + i)).red = magR;
58         (*(outMag + i)).green = magG;
59         (*(outMag + i)).blue = magB;
60
61         // std::complex for arg()
62         complex<double> cR(realR, imagR);
```

```
63         complex<double> cG(realG, imagG);
64         complex<double> cB(realB, imagB);
65
66         // phase
67         double phaseR = arg(cR) + M_PI;
68         double phaseG = arg(cG) + M_PI;
69         double phaseB = arg(cB) + M_PI;
70
71         // scale and write to output
72         (*(outPhase + i)).red = (phaseR / (double)(2 * M_PI)) * MaxRGB;
73         (*(outPhase + i)).green = (phaseG / (double)(2 * M_PI)) * MaxRGB;
74         (*(outPhase + i)).blue = (phaseB / (double)(2 * M_PI)) * MaxRGB;
75     }
76
77     // free memory
78     fftw_destroy_plan(planR);
79     fftw_destroy_plan(planG);
80     fftw_destroy_plan(planB);
81     fftw_free(inR); fftw_free(outR);
82     fftw_free(inG); fftw_free(outG);
83     fftw_free(inB); fftw_free(outB);
84 }
```

A first Test

If you havent been writing the code up to now here's what your `main.cpp` should be looking like:

Devino programator

 link-academy.com/Programare



main.cpp

```
1  #include <fftw3.h>
2  #include <Magick++.h>
3  #include <math.h>
4  #include <complex>
5  #include <iostream>
6  #include <string>
7  using namespace Magick;
8  using namespace std;
9
10 void fft(int squareSize, PixelPacket* pixels, PixelPacket* outMag, PixelPacket* outCol)
11 {
12     fftw_plan planR, planG, planB;
13     fftw_complex *inR, *inG, *inB, *outR, *outG, *outB;
14
15     // allocate input arrays
16     inR = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
17     inG = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
18     inB = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
19
20     // allocate output arrays
21     outR = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
22     outG = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
23     outB = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
24
25     // create plans
26     planR = fftw_plan_dft_2d(squareSize, squareSize, inR, outR, FFTW_FORWARD, FFTW_CONJUGATE);
27     planG = fftw_plan_dft_2d(squareSize, squareSize, inG, outG, FFTW_FORWARD, FFTW_CONJUGATE);
28     planB = fftw_plan_dft_2d(squareSize, squareSize, inB, outB, FFTW_FORWARD, FFTW_CONJUGATE);
29
30     // assign values to real parts (values between 0 and MaxRGB)
```

```
31     for(int i = 0; i < squareSize * squareSize; i++) {
32         PixelPacket current = *(pixels + i);
33         double red = current.red;
34         double green = current.green;
35         double blue = current.blue;
36         // save as real numbers
37         inR[i][0] = red;
38         inG[i][0] = green;
39         inB[i][0] = blue;
40     }
41
42     // perform FORWARD fft
43     fftw_execute(planR);
44     fftw_execute(planG);
45     fftw_execute(planB);
46
47     // transform imaginary number to phase and magnitude and save to output
48     for(int i = 0; i < squareSize * squareSize; i++) {
49         // normalize values
50         double realR = outR[i][0] / (double)(squareSize * squareSize);
51         double imagR = outR[i][1] / (double)(squareSize * squareSize);
52
53         double realG = outG[i][0] / (double)(squareSize * squareSize);
54         double imagG = outG[i][1] / (double)(squareSize * squareSize);
55
56         double realB = outB[i][0] / (double)(squareSize * squareSize);
57         double imagB = outB[i][1] / (double)(squareSize * squareSize);
58
59         // magnitude
60         double magR = sqrt((realR * realR) + (imagR * imagR));
61         double magG = sqrt((realG * realG) + (imagG * imagG));
62         double magB = sqrt((realB * realB) + (imagB * imagB));
63     }
```

```
64         // write to output
65         (*(outMag + i)).red = magR;
66         (*(outMag + i)).green = magG;
67         (*(outMag + i)).blue = magB;
68
69         // std::complex for arg()
70         complex<double> cR(realR, imagR);
71         complex<double> cG(realG, imagG);
72         complex<double> cB(realB, imagB);
73
74         // phase
75         double phaseR = arg(cR) + M_PI;
76         double phaseG = arg(cG) + M_PI;
77         double phaseB = arg(cB) + M_PI;
78
79         // scale and write to output
80         (*(outPhase + i)).red = (phaseR / (double)(2 * M_PI)) * MaxRGB;
81         (*(outPhase + i)).green = (phaseG / (double)(2 * M_PI)) * MaxRGB;
82         (*(outPhase + i)).blue = (phaseB / (double)(2 * M_PI)) * MaxRGB;
83     }
84
85     // free memory
86     fftw_destroy_plan(planR);
87     fftw_destroy_plan(planG);
88     fftw_destroy_plan(planB);
89     fftw_free(inR); fftw_free(outR);
90     fftw_free(inG); fftw_free(outG);
91     fftw_free(inB); fftw_free(outB);
92 }
93
94 void ift(int squareSize, PixelPacket* inMag, PixelPacket* inPhase, PixelPacket* outMag)
95 {
96 }
```

```
97
98  int main(int argc, char** argv)
99  {
100      if(argc < 4) {
101          cout << "usage:\t" << argv[0] << " in outMag outPhase" << endl
102              << "\t" << argv[0] << " -ift inMag inPhase out" << endl;
103          return 0;
104      }
105
106      string arg = argv[1];
107
108      if(arg != "-ift") {
109          // FORWARD FOURIER TRANSFORM
110          Image img(argv[1]);
111
112          // get the length of the longer side of the image
113          int squareSize = img.columns() < img.rows() ? img.rows() : img.columns();
114
115          // the geometry of our padded image
116          Geometry padded(squareSize, squareSize);
117          padded.aspect(true);
118
119          // make image square
120          img.extent(padded);
121
122          // create templates for magnitude and phase
123          Image mag(Geometry(squareSize, squareSize), "Black");
124          Image phase(Geometry(squareSize, squareSize), "Black");
125
126          // get image pixels
127          img.modifyImage();
128          Pixels pixelCache(img);
129          PixelPacket* pixels;
```

```
130         pixels = pixelCache.get(0, 0, squareSize, squareSize);
131
132         // get magnitude pixels
133         mag.modifyImage();
134         Pixels pixelCacheMag(mag);
135         PixelPacket* pixelsMag;
136         pixelsMag = pixelCacheMag.get(0, 0, squareSize, squareSize);
137
138         // get phase pixels
139         phase.modifyImage();
140         Pixels pixelCachePhase(phase);
141         PixelPacket* pixelsPhase;
142         pixelsPhase = pixelCachePhase.get(0, 0, squareSize, squareSize);
143
144         // perform fft
145         fft(squareSize, pixels, pixelsMag, pixelsPhase);
146
147         // write changes
148         pixelCache.sync();
149         pixelCacheMag.sync();
150         pixelCachePhase.sync();
151
152         // save files
153         mag.write(argv[2]);
154         phase.write(argv[3]);
155     }
156     else {
157         // BACKWARD FOURIER TRANSFORM
158     }
159     return 0;
160 }
```

Open a shell, enter your fourier directory and type make and sudo make install

```
1 user@host:~/fourier$ make
2 g++ -c -o main.o `Magick++-config --cppflags --cxxflags --ldflags --libs` `pkg-
3 g++ -o fourier `Magick++-config --cppflags --cxxflags --ldflags --libs` `pkg-co
4 user@host:~/fourier$ sudo make install
5 cp fourier /usr/bin/
```

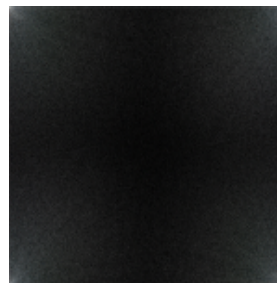
Now you can run `fourier` with the right input parameters to get a magnitude and phase image.

Save the image at the top of this tutorial to a directory of your choice. In this directory execute the following:

```
1 user@host:~/fft$ fourier in.png mag.png phase.png
2 user@host:~/fft$ convert mag.png -contrast-stretch 0 -evaluate log 10000 mag-ld
```

The first command will do the FFT and save the images. The second command will scale the magnitude image logarithmically as seen at the top of this page.

You may notice that the images **do not match**. Indeed our transform so far produces something like this:



Current magnitude image

The picture above is the correct representation of the magnitudes of our input images' frequencies although the zero-frequency is not in the center, but in the corners. The zero frequency is often moved to $(\text{sizeX}/2, \text{sizeY}/2)$ so it

converges to the middle. We get the desired result if we swap the four quadrants of the image diagonally.

The method `swapQuadrants` will do just that:

`main.cpp`

```
1  void swapQuadrants(int squareSize, PixelPacket* pixels)
2  {
3      int half = floor(squareSize / (double)2);
4
5      // swap quadrants diagonally
6      for(int i = 0; i < half; i++) {
7          for(int j = 0; j < half; j++) {
8              int upper = j + (squareSize * i);
9              int lower = upper + (squareSize * half) + half;
10
11              PixelPacket cur0 = *(pixels + upper);
12              *(pixels + upper) = *(pixels + lower);
13              *(pixels + lower) = cur0;
14
15              PixelPacket cur1 = *(pixels + upper + half);
16              *(pixels + upper + half) = *(pixels + lower - half);
17              *(pixels + lower - half) = cur1;
18          }
19      }
20 }
```

Add this code just after the includes so we can use it within `fftw`.

Calling the `swapQuadrants` on magnitude and phase output before saving them in our `fftw` method will then give us the desired image

main.cpp

```
1  void fft(int squareSize, PixelPacket* pixels, PixelPacket* outMag, PixelPacket* outPhase)
2  {
3      fftw_plan planR, planG, planB;
4      fftw_complex *inR, *inG, *inB, *outR, *outG, *outB;
5
6      // allocate input arrays
7      inR = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
8      inG = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
9      inB = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
10
11     // allocate output arrays
12     outR = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
13     outG = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
14     outB = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
15
16     // create plans
17     planR = fftw_plan_dft_2d(squareSize, squareSize, inR, outR, FFTW_FORWARD, FFTW_MEASURE);
18     planG = fftw_plan_dft_2d(squareSize, squareSize, inG, outG, FFTW_FORWARD, FFTW_MEASURE);
19     planB = fftw_plan_dft_2d(squareSize, squareSize, inB, outB, FFTW_FORWARD, FFTW_MEASURE);
20
21     // assign values to real parts (values between 0 and MaxRGB)
22     for(int i = 0; i < squareSize * squareSize; i++) {
23         PixelPacket current = *(pixels + i);
24         double red = current.red;
25         double green = current.green;
26         double blue = current.blue;
27
28         // save as real numbers
29         inR[i][0] = red;
30         inG[i][0] = green;
```

```
31         inB[i][0] = blue;
32     }
33
34     // perform FORWARD fft
35     fftw_execute(planR);
36     fftw_execute(planG);
37     fftw_execute(planB);
38
39     // transform imaginary number to phase and magnitude and save to output
40     for(int i = 0; i < squareSize * squareSize; i++) {
41         // normalize values
42         double realR = outR[i][0] / (double)(squareSize * squareSize);
43         double imagR = outR[i][1] / (double)(squareSize * squareSize);
44
45         double realG = outG[i][0] / (double)(squareSize * squareSize);
46         double imagG = outG[i][1] / (double)(squareSize * squareSize);
47
48         double realB = outB[i][0] / (double)(squareSize * squareSize);
49         double imagB = outB[i][1] / (double)(squareSize * squareSize);
50
51         // magnitude
52         double magR = sqrt((realR * realR) + (imagR * imagR));
53         double magG = sqrt((realG * realG) + (imagG * imagG));
54         double magB = sqrt((realB * realB) + (imagB * imagB));
55
56         // write to output
57         (*(outMag + i)).red = magR;
58         (*(outMag + i)).green = magG;
59         (*(outMag + i)).blue = magB;
60
61         // std::complex for arg()
62         complex<double> cR(realR, imagR);
63         complex<double> cG(realG, imagG);
```

```
64         complex<double> cB(realB, imagB);
65
66         // phase
67         double phaseR = arg(cR) + M_PI;
68         double phaseG = arg(cG) + M_PI;
69         double phaseB = arg(cB) + M_PI;
70
71         // scale and write to output
72         (*(outPhase + i)).red = (phaseR / (double)(2 * M_PI)) * MaxRGB;
73         (*(outPhase + i)).green = (phaseG / (double)(2 * M_PI)) * MaxRGB;
74         (*(outPhase + i)).blue = (phaseB / (double)(2 * M_PI)) * MaxRGB;
75     }
76
77     // move zero frequency to (squareSize/2, squareSize/2)
78     swapQuadrants(squareSize, outMag);
79     swapQuadrants(squareSize, outPhase);
80
81     // free memory
82     fftw_destroy_plan(planR);
83     fftw_destroy_plan(planG);
84     fftw_destroy_plan(planB);
85     fftw_free(inR); fftw_free(outR);
86     fftw_free(inG); fftw_free(outG);
87     fftw_free(inB); fftw_free(outB);
88 }
```

After compiling, installing and executing the `fourier` and `convert` commands, you should get an image similar to that at the top of the page.

Implementing the iDFT

As this part is very much the same as everything described above we will keep it short.

All the `ift` method has to do is:

- Swap the quadrants back into their original place
- Scale the phase values back to $[0, 2\pi]$
- Shift the phase values back to $[-\pi, \pi]$
- Transform phase and magnitude back to real and imaginary parts
- Perform backward fft on these values
- Store real parts in output

The final `main.cpp` doing all this looks like so:

`main.cpp`

```
1  #include <fftw3.h>
2  #include <Magick++.h>
3  #include <math.h>
4  #include <complex>
5  #include <iostream>
6  #include <string>
7  using namespace Magick;
8  using namespace std;
9
10 void swapQuadrants(int squareSize, PixelPacket* pixels)
11 {
12     int half = floor(squareSize / (double)2);
13
14     // swap quadrants diagonally
15     for(int i = 0; i < half; i++) {
16         for(int j = 0; j < half; j++) {
```

```
17         int upper = j + (squareSize * i);
18         int lower = upper + (squareSize * half) + half;
19
20         PixelPacket cur0 = *(pixels + upper);
21         *(pixels + upper) = *(pixels + lower);
22         *(pixels + lower) = cur0;
23
24         PixelPacket cur1 = *(pixels + upper + half);
25         *(pixels + upper + half) = *(pixels + lower - half);
26         *(pixels + lower - half) = cur1;
27     }
28 }
29 }
30
31 void fft(int squareSize, PixelPacket* pixels, PixelPacket* outMag, PixelPacket* outPhase)
32 {
33     fftw_plan planR, planG, planB;
34     fftw_complex *inR, *inG, *inB, *outR, *outG, *outB;
35
36     // allocate input arrays
37     inR = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
38     inG = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
39     inB = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
40
41     // allocate output arrays
42     outR = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
43     outG = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
44     outB = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
45
46     // create plans
47     planR = fftw_plan_dft_2d(squareSize, squareSize, inR, outR, FFTW_FORWARD, FFTW_ESTIMATE);
48     planG = fftw_plan_dft_2d(squareSize, squareSize, inG, outG, FFTW_FORWARD, FFTW_ESTIMATE);
49     planB = fftw_plan_dft_2d(squareSize, squareSize, inB, outB, FFTW_FORWARD, FFTW_ESTIMATE);
```

```
50
51 // assign values to real parts (values between 0 and MaxRGB)
52 for(int i = 0; i < squareSize * squareSize; i++) {
53     PixelPacket current = *(pixels + i);
54     double red = current.red;
55     double green = current.green;
56     double blue = current.blue;
57
58     // save as real numbers
59     inR[i][0] = red;
60     inG[i][0] = green;
61     inB[i][0] = blue;
62 }
63
64 // perform FORWARD fft
65 fftw_execute(planR);
66 fftw_execute(planG);
67 fftw_execute(planB);
68
69 // transform imaginary number to phase and magnitude and save to output
70 for(int i = 0; i < squareSize * squareSize; i++) {
71     // normalize values
72     double realR = outR[i][0] / (double)(squareSize * squareSize);
73     double imagR = outR[i][1] / (double)(squareSize * squareSize);
74
75     double realG = outG[i][0] / (double)(squareSize * squareSize);
76     double imagG = outG[i][1] / (double)(squareSize * squareSize);
77
78     double realB = outB[i][0] / (double)(squareSize * squareSize);
79     double imagB = outB[i][1] / (double)(squareSize * squareSize);
80
81     // magnitude
82     double magR = sqrt((realR * realR) + (imagR * imagR));
```

```
83     double magG = sqrt((realG * realG) + (imagG * imagG));
84     double magB = sqrt((realB * realB) + (imagB * imagB));
85
86     // write to output
87     (*(outMag + i)).red = magR;
88     (*(outMag + i)).green = magG;
89     (*(outMag + i)).blue = magB;
90
91     // std::complex for arg()
92     complex<double> cR(realR, imagR);
93     complex<double> cG(realG, imagG);
94     complex<double> cB(realB, imagB);
95
96     // phase
97     double phaseR = arg(cR) + M_PI;
98     double phaseG = arg(cG) + M_PI;
99     double phaseB = arg(cB) + M_PI;
100
101     // scale and write to output
102     (*(outPhase + i)).red = (phaseR / (double)(2 * M_PI)) * MaxRGB;
103     (*(outPhase + i)).green = (phaseG / (double)(2 * M_PI)) * MaxRGB;
104     (*(outPhase + i)).blue = (phaseB / (double)(2 * M_PI)) * MaxRGB;
105 }
106
107 // move zero frequency to (squareSize/2, squareSize/2)
108 swapQuadrants(squareSize, outMag);
109 swapQuadrants(squareSize, outPhase);
110
111 // free memory
112 fftw_destroy_plan(planR);
113 fftw_destroy_plan(planG);
114 fftw_destroy_plan(planB);
115 fftw_free(inR); fftw_free(outR);
```



```

116     fftw_free(inG); fftw_free(outG);
117     fftw_free(inB); fftw_free(outB);
118 }
119
120 void ift(int squareSize, PixelPacket* inMag, PixelPacket* inPhase, PixelPacket* outMag, PixelPacket* outPhase)
121 {
122     // move zero frequency back to corners
123     swapQuadrants(squareSize, inMag);
124     swapQuadrants(squareSize, inPhase);
125
126     fftw_plan planR, planG, planB;
127     fftw_complex *inR, *inG, *inB, *outR, *outG, *outB;
128
129     // allocate input arrays
130     inR = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
131     inG = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
132     inB = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
133
134     // allocate output arrays
135     outR = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
136     outG = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
137     outB = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squareSize);
138
139     // create plans
140     planR = fftw_plan_dft_2d(squareSize, squareSize, inR, outR, FFTW_BACKWARD);
141     planG = fftw_plan_dft_2d(squareSize, squareSize, inG, outG, FFTW_BACKWARD);
142     planB = fftw_plan_dft_2d(squareSize, squareSize, inB, outB, FFTW_BACKWARD);
143
144     // transform magnitude/phase to real/imaginary
145     for(int i = 0; i < squareSize * squareSize; i++) {
146         double magR = inMag[i].red;
147         double phaseR = ((inPhase[i].red / (double)MaxRGB) * 2 * M_PI) - M_PI;
148         inR[i][0] = (magR * cos(phaseR));

```

```
149         inR[i][1] = (magR * sin(phaseR));
150
151         double magB = inMag[i].blue;
152         double phaseB = ((inPhase[i].blue / (double)MaxRGB) * 2 * M_PI) - M_PI;
153         inB[i][0] = (magB * cos(phaseB));
154         inB[i][1] = (magB * sin(phaseB));
155
156         double magG = inMag[i].green;
157         double phaseG = ((inPhase[i].green / (double)MaxRGB) * 2 * M_PI) - M_PI;
158         inG[i][0] = (magG * cos(phaseG));
159         inG[i][1] = (magG * sin(phaseG));
160     }
161
162     // perform ifft
163     fftw_execute(planR);
164     fftw_execute(planG);
165     fftw_execute(planB);
166
167     // save real parts to output
168     for(int i = 0; i < squareSize * squareSize; i++) {
169         double magR = outR[i][0];
170         double magG = outG[i][0];
171         double magB = outB[i][0];
172
173         // make sure it's capped at MaxRGB
174         (*(outPixels + i)).red = magR > MaxRGB ? MaxRGB : magR;
175         (*(outPixels + i)).green = magG > MaxRGB ? MaxRGB : magG;
176         (*(outPixels + i)).blue = magB > MaxRGB ? MaxRGB : magB;
177     }
178
179     // free memory
180     fftw_destroy_plan(planR);
181     fftw_destroy_plan(planG);
```

```
182     fftw_destroy_plan(planB);
183     fftw_free(inR); fftw_free(outR);
184     fftw_free(inG); fftw_free(outG);
185     fftw_free(inB); fftw_free(outB);
186 }
187
188 int main(int argc, char** argv)
189 {
190     if(argc < 4) {
191         cout << "usage:\t" << argv[0] << " in outMag outPhase" << endl
192              << "\t" << argv[0] << " -ift inMag inPhase out" << endl;
193         return 0;
194     }
195
196     string arg = argv[1];
197
198     if(arg != "-ift") {
199         // FORWARD FOURIER TRANSFORM
200         Image img(argv[1]);
201
202         // get the length of the longer side of the image
203         int squareSize = img.columns() < img.rows() ? img.rows() : img.columns();
204
205         // the geometry of our padded image
206         Geometry padded(squareSize, squareSize);
207         padded.aspect(true);
208
209         // make image square
210         img.extent(padded);
211
212         // create templates for magnitude and phase
213         Image mag(Geometry(squareSize, squareSize), "Black");
214         Image phase(Geometry(squareSize, squareSize), "Black");
```

```
215
216 // get image pixels
217 img.modifyImage();
218 Pixels pixelCache(img);
219 PixelPacket* pixels;
220 pixels = pixelCache.get(0, 0, squareSize, squareSize);
221
222 // get magnitude pixels
223 mag.modifyImage();
224 Pixels pixelCacheMag(mag);
225 PixelPacket* pixelsMag;
226 pixelsMag = pixelCacheMag.get(0, 0, squareSize, squareSize);
227
228 // get phase pixels
229 phase.modifyImage();
230 Pixels pixelCachePhase(phase);
231 PixelPacket* pixelsPhase;
232 pixelsPhase = pixelCachePhase.get(0, 0, squareSize, squareSize);
233
234 // perform fft
235 fft(squareSize, pixels, pixelsMag, pixelsPhase);
236
237 // write changes
238 pixelCache.sync();
239 pixelCacheMag.sync();
240 pixelCachePhase.sync();
241
242 // save files
243 mag.write(argv[2]);
244 phase.write(argv[3]);
245 }
246 else {
247 // BACKWARD FOURIER TRANSFORM
```

```
248     Image mag(argv[2]);
249     Image phase(argv[3]);
250
251     // get size
252     int squareSize = mag.columns();
253     Image img(Geometry(squareSize, squareSize), "Black");
254
255     // get image pixels
256     img.modifyImage();
257     Pixels pixelCache(img);
258     PixelPacket* pixels;
259     pixels = pixelCache.get(0, 0, squareSize, squareSize);
260
261     // get magnitude pixels
262     mag.modifyImage();
263     Pixels pixelCacheMag(mag);
264     PixelPacket* pixelsMag;
265     pixelsMag = pixelCacheMag.get(0, 0, squareSize, squareSize);
266
267     // get phase pixels
268     phase.modifyImage();
269     Pixels pixelCachePhase(phase);
270     PixelPacket* pixelsPhase;
271     pixelsPhase = pixelCachePhase.get(0, 0, squareSize, squareSize);
272
273     // perform ifft
274     ift(squareSize, pixelsMag, pixelsPhase, pixels);
275
276     // write changes
277     pixelCache.sync();
278
279     // save file
280     img.write(argv[4]);
```

```
281 |     }  
282 |  
283 |     return 0;  
284 | }
```

You can get an archive with all the sources [here](#).

Remember to remove the `-g` and `-O0` flags from the makefile when actually using it. (These are there for debug purposes)

Executing the command

```
1 | user@host:~/fftw$ fourier -ift mag.png phase.png out.png
```

will produce and save the image obtained from the inverse DFT in out.png.

I take no responsibility in how you use this code and make no guarantee as to its fitness for production environments.

Have fun experimenting with libfftw in C++.

Like

11

Tweet

[+](#) Share / Save [f](#) [t](#) [↗](#)

This entry was posted in [Linux](#) and tagged [c++](#), [dft](#), [fft](#), [fftw](#), [fourier](#), [libfftw](#), [linux](#). Bookmark the [permalink](#).

admindojo

Proudly powered by WordPress.