admindojo

tutorials and apps

Discrete Fourier Transform in C++ with fftw

Posted on March 12, 2011 by fmauro

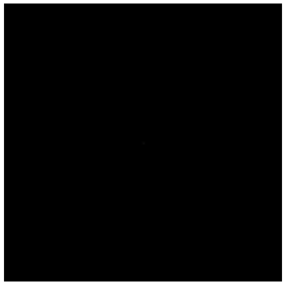
This tutorial will give you a short introduction to using libfftw in C++ under Linux.

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

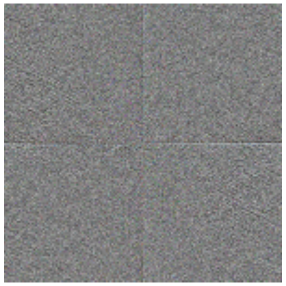
Some knowledge of complex numbers is required to understand the transform itself but we'll keep the math to the bare minimum.

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

Input image

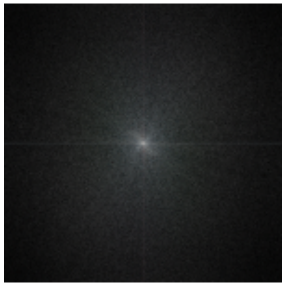


Magnitude image



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:

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

or get it on the web:
[libfftw]
[Imagemagick]

Create a folder and source structure:

```
mkdir fourier
touch fourier/Makefile
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

```
CXXFLAGS=`Magick++-config --cppflags --cxxflags --ldflags --libs` `pkg-config
CXX=g++

all: fourier

fourier: main.o
	$(CXX) -o $@ $(CXXFLAGS) $<

main.o: main.cpp
	$(CXX) -c -o $@ $(CXXFLAGS) $<

clean:
	rm main.o

install:
	cp fourier /usr/bin/

uninstall:
	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

```cpp
#include <fftw3.h>
#include <Magick++.h>
#include <math.h>
#include <complex>
#include <iostream>
#include <string>
using namespace Magick;
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

```
int main(int argc, char** argv)
{
    if(argc < 4) {
        cout << "usage:\t" << argv[0] << " in outMag outPhase" << endl
                << "\t" << argv[0] << " -ift inMag inPhase out" << endl;
        return 0;
    }

    string arg = argv[1];

    if(arg != "-ift") {
        // FORWARD TRANSFORM
    }
    else {
        // BACKWARD TRANSFORM
    }
    return 0;
}
```

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

```
Image img("myfilename");

// lock image for modification
img.modifyImage();
Pixels pixelCache(img);
PixelPacket* pixels;

// get desired area in image
pixels = pixelCache.get(0, 0, sizeX, sizeY);

// read/write pixel (12,3)
*(pixel + 12 + (3 * sizeX)) = Color("Blue");

// write changes
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

```
void fft(int squareSize, PixelPacket* pixels, PixelPacket* outMag, PixelPacket
{
}

void ift(int squareSize, PixelPacket* inMag, PixelPacket* inPhase, PixelPacket
{
}
```

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

```
int main(int argc, char** argv)
{
    if(argc < 4) {
        cout << "usage:\t" << argv[0] << " in outMag outPhase" << endl
                << "\t" << argv[0] << " -ift inMag inPhase out" << endl;
        return 0;
    }

    string arg = argv[1];

    if(arg != "-ift") {
        // FORWARD FOURIER TRANSFORM
        Image img(argv[1]);

        // get the length of the longer side of the image
        int squareSize = img.columns() < img.rows() ? img.rows() : img.column

        // the geometry of our padded image
        Geometry padded(squareSize, squareSize);
        padded.aspect(true);

        // make image square
        img.extent(padded);

        // create templates for magnitude and phase
        Image mag(Geometry(squareSize, squareSize), "Black");
```

```
        Image phase(Geometry(squareSize, squareSize), "Black");

        // get image pixels
        img.modifyImage();
        Pixels pixelCache(img);
        PixelPacket* pixels;
        pixels = pixelCache.get(0, 0, squareSize, squareSize);

        // get magnitude pixels
        mag.modifyImage();
        Pixels pixelCacheMag(mag);
        PixelPacket* pixelsMag;
        pixelsMag = pixelCacheMag.get(0, 0, squareSize, squareSize);

        // get phase pixels
        phase.modifyImage();
        Pixels pixelCachePhase(phase);
        PixelPacket* pixelsPhase;
        pixelsPhase = pixelCachePhase.get(0, 0, squareSize, squareSize);

        // perform fft
        fft(squareSize, pixels, pixelsMag, pixelsPhase);

        // write changes
        pixelCache.sync();
        pixelCacheMag.sync();
        pixelCachePhase.sync();

        // save files
        mag.write(argv[2]);
        phase.write(argv[3]);
    }
    else {
```

```
        // BACKWARD FOURIER TRANSFORM
    }
    return 0;
}
```

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

```
fftw_plan plan;
fftw_complex *in, *out;

// allocate memory for input and output arrays
in = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * sizeX * sizeY);
out = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * sizeX * sizeY);

// create the plan
plan = fftw_plan_dft_2d(sizeX, sizeY, in, out, FFTW_FORWARD, FFTW_ESTIMATE);

// fill the array
for(int i = 0, i < sizeY; i++)
    for(int j = 0; j < sizeX; j++)
        in[j + (sizeX * i)][0] = randomNumber; // in[][0] is the real part of

// execute the specified transform
```

```
fftw_execute(plan);

// use output
[...]

// free allocated memory
fftw_destroy_plan(plan);
fftw_free(out);
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

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

    // allocate input arrays
    inR = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squa
    inG = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squa
    inB = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squa

    // allocate output arrays
    outR = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squ
    outG = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squ
    outB = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squ

    // create plans
```

```
    planR = fftw_plan_dft_2d(squareSize, squareSize, inR, outR, FFTW_FORWARD,
    planG = fftw_plan_dft_2d(squareSize, squareSize, inG, outG, FFTW_FORWARD,
    planB = fftw_plan_dft_2d(squareSize, squareSize, inB, outB, FFTW_FORWARD,

    // TODO: assign color-values to input arrays

    // perform FORWARD fft
    fftw_execute(planR);
    fftw_execute(planG);
    fftw_execute(planB);

    // TODO: calculate output mag/phase

    // free memory
    fftw_destroy_plan(planR);
    fftw_destroy_plan(planG);
    fftw_destroy_plan(planB);
    fftw_free(inR); fftw_free(outR);
    fftw_free(inG); fftw_free(outG);
    fftw_free(inB); fftw_free(outB);
}
```

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

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

```

```
    // allocate input arrays
    inR = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squa
    inG = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squa
    inB = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squa

    // allocate output arrays
    outR = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squ
    outG = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squ
    outB = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squ

    // create plans
    planR = fftw_plan_dft_2d(squareSize, squareSize, inR, outR, FFTW_FORWARD,
    planG = fftw_plan_dft_2d(squareSize, squareSize, inG, outG, FFTW_FORWARD,
    planB = fftw_plan_dft_2d(squareSize, squareSize, inB, outB, FFTW_FORWARD,

    // assign values to real parts (values between 0 and MaxRGB)
    for(int i = 0; i < squareSize * squareSize; i++) {
        PixelPacket current = *(pixels + i);
        double red = current.red;
        double green = current.green;
        double blue = current.blue;

        // save as real numbers
        inR[i][0] = red;
        inG[i][0] = green;
        inB[i][0] = blue;
    }

    // perform FORWARD fft
    fftw_execute(planR);
    fftw_execute(planG);
    fftw_execute(planB);

```

```
    // TODO: calculate output mag/phase

    // free memory
    fftw_destroy_plan(planR);
    fftw_destroy_plan(planG);
    fftw_destroy_plan(planB);
    fftw_free(inR); fftw_free(outR);
    fftw_free(inG); fftw_free(outG);
    fftw_free(inB); fftw_free(outB);
}
```

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 $[0, \mathrm{MaxRGB}]$ and scale/shift it back when doing the inverse DFT. Also libfftw for performance reasons scales the output by `(sizeX * 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

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

    // allocate input arrays
    inR = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squa
    inG = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squa
    inB = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squa

    // allocate output arrays
    outR = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squ
    outG = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squ
    outB = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squ

    // create plans
    planR = fftw_plan_dft_2d(squareSize, squareSize, inR, outR, FFTW_FORWARD,
    planG = fftw_plan_dft_2d(squareSize, squareSize, inG, outG, FFTW_FORWARD,
    planB = fftw_plan_dft_2d(squareSize, squareSize, inB, outB, FFTW_FORWARD,

    // assign values to real parts (values between 0 and MaxRGB)
    for(int i = 0; i < squareSize * squareSize; i++) {
        PixelPacket current = *(pixels + i);
        double red = current.red;
        double green = current.green;
        double blue = current.blue;

        // save as real numbers
        inR[i][0] = red;
```

```
            inG[i][0] = green;
            inB[i][0] = blue;
        }

        // perform FORWARD fft
        fftw_execute(planR);
        fftw_execute(planG);
        fftw_execute(planB);

        // transform imaginary number to phase and magnitude and save to output
        for(int i = 0; i < squareSize * squareSize; i++) {
            // normalize values
            double realR = outR[i][0] / (double)(squareSize * squareSize);
            double imagR = outR[i][1] / (double)(squareSize * squareSize);

            double realG = outG[i][0] / (double)(squareSize * squareSize);
            double imagG = outG[i][1] / (double)(squareSize * squareSize);

            double realB = outB[i][0] / (double)(squareSize * squareSize);
            double imagB = outB[i][1] / (double)(squareSize * squareSize);

            // magnitude
            double magR = sqrt((realR * realR) + (imagR * imagR));
            double magG = sqrt((realG * realG) + (imagG * imagG));
            double magB = sqrt((realB * realB) + (imagB * imagB));

            // write to output
            (*(outMag + i)).red = magR;
            (*(outMag + i)).green = magG;
            (*(outMag + i)).blue = magB;

            // std::complex for arg()
            complex<double> cR(realR, imagR);
```

```
        complex<double> cG(realG, imagG);
        complex<double> cB(realB, imagB);

        // phase
        double phaseR = arg(cR) + M_PI;
        double phaseG = arg(cG) + M_PI;
        double phaseB = arg(cB) + M_PI;

        // scale and write to output
        (*(outPhase + i)).red = (phaseR / (double)(2 * M_PI)) * MaxRGB;
        (*(outPhase + i)).green = (phaseG / (double)(2 * M_PI)) * MaxRGB;
        (*(outPhase + i)).blue = (phaseB / (double)(2 * M_PI)) * MaxRGB;
    }

    // free memory
    fftw_destroy_plan(planR);
    fftw_destroy_plan(planG);
    fftw_destroy_plan(planB);
    fftw_free(inR); fftw_free(outR);
    fftw_free(inG); fftw_free(outG);
    fftw_free(inB); fftw_free(outB);
}
```

A first Test

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

main.cpp

```
#include <fftw3.h>
#include <Magick++.h>
#include <math.h>
#include <complex>
#include <iostream>
#include <string>
using namespace Magick;
using namespace std;

void fft(int squareSize, PixelPacket* pixels, PixelPacket* outMag, PixelPack
{
    fftw_plan planR, planG, planB;
    fftw_complex *inR, *inG, *inB, *outR, *outG, *outB;

    // allocate input arrays
    inR = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * sq
    inG = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * sq
    inB = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * sq

    // allocate output arrays
    outR = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * s
    outG = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * s
    outB = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * s

    // create plans
    planR = fftw_plan_dft_2d(squareSize, squareSize, inR, outR, FFTW_FORWARD
    planG = fftw_plan_dft_2d(squareSize, squareSize, inG, outG, FFTW_FORWARD
    planB = fftw_plan_dft_2d(squareSize, squareSize, inB, outB, FFTW_FORWARD

    // assign values to real parts (values between 0 and MaxRGB)
```

```
    for(int i = 0; i < squareSize * squareSize; i++) {
        PixelPacket current = *(pixels + i);
        double red = current.red;
        double green = current.green;
        double blue = current.blue;
        // save as real numbers
        inR[i][0] = red;
        inG[i][0] = green;
        inB[i][0] = blue;
    }

    // perform FORWARD fft
    fftw_execute(planR);
    fftw_execute(planG);
    fftw_execute(planB);

    // transform imaginary number to phase and magnitude and save to output
    for(int i = 0; i < squareSize * squareSize; i++) {
        // normalize values
        double realR = outR[i][0] / (double)(squareSize * squareSize);
        double imagR = outR[i][1] / (double)(squareSize * squareSize);

        double realG = outG[i][0] / (double)(squareSize * squareSize);
        double imagG = outG[i][1] / (double)(squareSize * squareSize);

        double realB = outB[i][0] / (double)(squareSize * squareSize);
        double imagB = outB[i][1] / (double)(squareSize * squareSize);

        // magnitude
        double magR = sqrt((realR * realR) + (imagR * imagR));
        double magG = sqrt((realG * realG) + (imagG * imagG));
        double magB = sqrt((realB * realB) + (imagB * imagB));

```

```
        // write to output
        (*(outMag + i)).red = magR;
        (*(outMag + i)).green = magG;
        (*(outMag + i)).blue = magB;

        // std::complex for arg()
        complex<double> cR(realR, imagR);
        complex<double> cG(realG, imagG);
        complex<double> cB(realB, imagB);

        // phase
        double phaseR = arg(cR) + M_PI;
        double phaseG = arg(cG) + M_PI;
        double phaseB = arg(cB) + M_PI;

        // scale and write to output
        (*(outPhase + i)).red = (phaseR / (double)(2 * M_PI)) * MaxRGB;
        (*(outPhase + i)).green = (phaseG / (double)(2 * M_PI)) * MaxRGB;
        (*(outPhase + i)).blue = (phaseB / (double)(2 * M_PI)) * MaxRGB;
    }

    // free memory
    fftw_destroy_plan(planR);
    fftw_destroy_plan(planG);
    fftw_destroy_plan(planB);
    fftw_free(inR); fftw_free(outR);
    fftw_free(inG); fftw_free(outG);
    fftw_free(inB); fftw_free(outB);
}

void ift(int squareSize, PixelPacket* inMag, PixelPacket* inPhase, PixelPack
{
}
```

```
int main(int argc, char** argv)
{
    if(argc < 4) {
        cout << "usage:\t" << argv[0] << " in outMag outPhase" << endl
                << "\t" << argv[0] << " -ift inMag inPhase out" << endl;
        return 0;
    }

    string arg = argv[1];

    if(arg != "-ift") {
        // FORWARD FOURIER TRANSFORM
        Image img(argv[1]);

        // get the length of the longer side of the image
        int squareSize = img.columns() < img.rows() ? img.rows() : img.colum

        // the geometry of our padded image
        Geometry padded(squareSize, squareSize);
        padded.aspect(true);

        // make image square
        img.extent(padded);

        // create templates for magnitude and phase
        Image mag(Geometry(squareSize, squareSize), "Black");
        Image phase(Geometry(squareSize, squareSize), "Black");

        // get image pixels
        img.modifyImage();
        Pixels pixelCache(img);
        PixelPacket* pixels;
```

```
            pixels = pixelCache.get(0, 0, squareSize, squareSize);

            // get magnitude pixels
            mag.modifyImage();
            Pixels pixelCacheMag(mag);
            PixelPacket* pixelsMag;
            pixelsMag = pixelCacheMag.get(0, 0, squareSize, squareSize);

            // get phase pixels
            phase.modifyImage();
            Pixels pixelCachePhase(phase);
            PixelPacket* pixelsPhase;
            pixelsPhase = pixelCachePhase.get(0, 0, squareSize, squareSize);

            // perform fft
            fft(squareSize, pixels, pixelsMag, pixelsPhase);

            // write changes
            pixelCache.sync();
            pixelCacheMag.sync();
            pixelCachePhase.sync();

            // save files
            mag.write(argv[2]);
            phase.write(argv[3]);
        }
        else {
            // BACKWARD FOURIER TRANSFORM
        }
        return 0;
    }
```

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

```
user@host:~/fourier$ make
g++ -c -o main.o `Magick++-config --cppflags --cxxflags --ldflags --libs` `pkg
g++ -o fourier `Magick++-config --cppflags --cxxflags --ldflags --libs` `pkg-c
user@host:~/fourier$ sudo make install
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:

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

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 pictude 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 `(sizeX/2, 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

```
void swapQuadrants(int squareSize, PixelPacket* pixels)
{
    int half = floor(squareSize / (double)2);

    // swap quadrants diagonally
    for(int i = 0; i < half; i++) {
        for(int j = 0; j < half; j++) {
            int upper = j + (squareSize * i);
            int lower = upper + (squareSize * half) + half;

            PixelPacket cur0 = *(pixels + upper);
            *(pixels + upper) = *(pixels + lower);
            *(pixels + lower) = cur0;

            PixelPacket cur1 = *(pixels + upper + half);
            *(pixels + upper + half) = *(pixels + lower - half);
            *(pixels + lower - half) = cur1;
        }
    }
}
```

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

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

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

    // allocate input arrays
    inR = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squa
    inG = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squa
    inB = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squa

    // allocate output arrays
    outR = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squ
    outG = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squ
    outB = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squ

    // create plans
    planR = fftw_plan_dft_2d(squareSize, squareSize, inR, outR, FFTW_FORWARD,
    planG = fftw_plan_dft_2d(squareSize, squareSize, inG, outG, FFTW_FORWARD,
    planB = fftw_plan_dft_2d(squareSize, squareSize, inB, outB, FFTW_FORWARD,

    // assign values to real parts (values between 0 and MaxRGB)
    for(int i = 0; i < squareSize * squareSize; i++) {
        PixelPacket current = *(pixels + i);
        double red = current.red;
        double green = current.green;
        double blue = current.blue;

        // save as real numbers
        inR[i][0] = red;
        inG[i][0] = green;
```

```
            inB[i][0] = blue;
        }

        // perform FORWARD fft
        fftw_execute(planR);
        fftw_execute(planG);
        fftw_execute(planB);

        // transform imaginary number to phase and magnitude and save to output
        for(int i = 0; i < squareSize * squareSize; i++) {
            // normalize values
            double realR = outR[i][0] / (double)(squareSize * squareSize);
            double imagR = outR[i][1] / (double)(squareSize * squareSize);

            double realG = outG[i][0] / (double)(squareSize * squareSize);
            double imagG = outG[i][1] / (double)(squareSize * squareSize);

            double realB = outB[i][0] / (double)(squareSize * squareSize);
            double imagB = outB[i][1] / (double)(squareSize * squareSize);

            // magnitude
            double magR = sqrt((realR * realR) + (imagR * imagR));
            double magG = sqrt((realG * realG) + (imagG * imagG));
            double magB = sqrt((realB * realB) + (imagB * imagB));

            // write to output
            (*(outMag + i)).red = magR;
            (*(outMag + i)).green = magG;
            (*(outMag + i)).blue = magB;

            // std::complex for arg()
            complex<double> cR(realR, imagR);
            complex<double> cG(realG, imagG);
```

```
            complex<double> cB(realB, imagB);

            // phase
            double phaseR = arg(cR) + M_PI;
            double phaseG = arg(cG) + M_PI;
            double phaseB = arg(cB) + M_PI;

            // scale and write to output
            (*(outPhase + i)).red = (phaseR / (double)(2 * M_PI)) * MaxRGB;
            (*(outPhase + i)).green = (phaseG / (double)(2 * M_PI)) * MaxRGB;
            (*(outPhase + i)).blue = (phaseB / (double)(2 * M_PI)) * MaxRGB;
        }

    // move zero frequency to (squareSize/2, squareSize/2)
    swapQuadrants(squareSize, outMag);
    swapQuadrants(squareSize, outPhase);

    // free memory
    fftw_destroy_plan(planR);
    fftw_destroy_plan(planG);
    fftw_destroy_plan(planB);
    fftw_free(inR); fftw_free(outR);
    fftw_free(inG); fftw_free(outG);
    fftw_free(inB); fftw_free(outB);
}
```

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

```cpp
#include <fftw3.h>
#include <Magick++.h>
#include <math.h>
#include <complex>
#include <iostream>
#include <string>
using namespace Magick;
using namespace std;

void swapQuadrants(int squareSize, PixelPacket* pixels)
{
    int half = floor(squareSize / (double)2);

    // swap quadrants diagonally
    for(int i = 0; i < half; i++) {
        for(int j = 0; j < half; j++) {
```

```
            int upper = j + (squareSize * i);
            int lower = upper + (squareSize * half) + half;

            PixelPacket cur0 = *(pixels + upper);
            *(pixels + upper) = *(pixels + lower);
            *(pixels + lower) = cur0;

            PixelPacket cur1 = *(pixels + upper + half);
            *(pixels + upper + half) = *(pixels + lower - half);
            *(pixels + lower - half) = cur1;
        }
    }
}

void fft(int squareSize, PixelPacket* pixels, PixelPacket* outMag, PixelPack
{
    fftw_plan planR, planG, planB;
    fftw_complex *inR, *inG, *inB, *outR, *outG, *outB;

    // allocate input arrays
    inR = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * sq
    inG = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * sq
    inB = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * sq

    // allocate output arrays
    outR = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * s
    outG = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * s
    outB = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * s

    // create plans
    planR = fftw_plan_dft_2d(squareSize, squareSize, inR, outR, FFTW_FORWARD
    planG = fftw_plan_dft_2d(squareSize, squareSize, inG, outG, FFTW_FORWARD
    planB = fftw_plan_dft_2d(squareSize, squareSize, inB, outB, FFTW_FORWARD
```

```

    // assign values to real parts (values between 0 and MaxRGB)
    for(int i = 0; i < squareSize * squareSize; i++) {
        PixelPacket current = *(pixels + i);
        double red = current.red;
        double green = current.green;
        double blue = current.blue;

        // save as real numbers
        inR[i][0] = red;
        inG[i][0] = green;
        inB[i][0] = blue;
    }

    // perform FORWARD fft
    fftw_execute(planR);
    fftw_execute(planG);
    fftw_execute(planB);

    // transform imaginary number to phase and magnitude and save to output
    for(int i = 0; i < squareSize * squareSize; i++) {
        // normalize values
        double realR = outR[i][0] / (double)(squareSize * squareSize);
        double imagR = outR[i][1] / (double)(squareSize * squareSize);

        double realG = outG[i][0] / (double)(squareSize * squareSize);
        double imagG = outG[i][1] / (double)(squareSize * squareSize);

        double realB = outB[i][0] / (double)(squareSize * squareSize);
        double imagB = outB[i][1] / (double)(squareSize * squareSize);

        // magnitude
        double magR = sqrt((realR * realR) + (imagR * imagR));
```

```
            double magG = sqrt((realG * realG) + (imagG * imagG));
            double magB = sqrt((realB * realB) + (imagB * imagB));

            // write to output
            (*(outMag + i)).red = magR;
            (*(outMag + i)).green = magG;
            (*(outMag + i)).blue = magB;

            // std::complex for arg()
            complex<double> cR(realR, imagR);
            complex<double> cG(realG, imagG);
            complex<double> cB(realB, imagB);

            // phase
            double phaseR = arg(cR) + M_PI;
            double phaseG = arg(cG) + M_PI;
            double phaseB = arg(cB) + M_PI;

            // scale and write to output
            (*(outPhase + i)).red = (phaseR / (double)(2 * M_PI)) * MaxRGB;
            (*(outPhase + i)).green = (phaseG / (double)(2 * M_PI)) * MaxRGB;
            (*(outPhase + i)).blue = (phaseB / (double)(2 * M_PI)) * MaxRGB;
        }

        // move zero frequency to (squareSize/2, squareSize/2)
        swapQuadrants(squareSize, outMag);
        swapQuadrants(squareSize, outPhase);

        // free memory
        fftw_destroy_plan(planR);
        fftw_destroy_plan(planG);
        fftw_destroy_plan(planB);
        fftw_free(inR); fftw_free(outR);
```

```
    fftw_free(inG); fftw_free(outG);
    fftw_free(inB); fftw_free(outB);
}

void ift(int squareSize, PixelPacket* inMag, PixelPacket* inPhase, PixelPack
{
    // move zero frequency back to corners
    swapQuadrants(squareSize, inMag);
    swapQuadrants(squareSize, inPhase);

    fftw_plan planR, planG, planB;
    fftw_complex *inR, *inG, *inB, *outR, *outG, *outB;

    // allocate input arrays
    inR = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squ
    inG = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squ
    inB = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * squ

    // allocate output arrays
    outR = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * s
    outG = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * s
    outB = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * squareSize * s

    // create plans
    planR = fftw_plan_dft_2d(squareSize, squareSize, inR, outR, FFTW_BACKWARI
    planG = fftw_plan_dft_2d(squareSize, squareSize, inG, outG, FFTW_BACKWARI
    planB = fftw_plan_dft_2d(squareSize, squareSize, inB, outB, FFTW_BACKWARI

    // transform magnitude/phase to real/imaginary
    for(int i = 0; i < squareSize * squareSize; i++) {
        double magR = inMag[i].red;
        double phaseR = ((inPhase[i].red / (double)MaxRGB) * 2 * M_PI) - M_P
        inR[i][0] = (magR * cos(phaseR));
```

```
            inR[i][1] = (magR * sin(phaseR));

            double magB = inMag[i].blue;
            double phaseB = ((inPhase[i].blue / (double)MaxRGB) * 2 * M_PI) - M_I
            inB[i][0] = (magB * cos(phaseB));
            inB[i][1] = (magB * sin(phaseB));

            double magG = inMag[i].green;
            double phaseG = ((inPhase[i].green / (double)MaxRGB) * 2 * M_PI) - M_
            inG[i][0] = (magG * cos(phaseG));
            inG[i][1] = (magG * sin(phaseG));
        }

        // perform ift
        fftw_execute(planR);
        fftw_execute(planG);
        fftw_execute(planB);

        // save real parts to output
        for(int i = 0; i < squareSize * squareSize; i++) {
            double magR = outR[i][0];
            double magG = outG[i][0];
            double magB = outB[i][0];

            // make sure it's capped at MaxRGB
            (*(outPixels + i)).red = magR > MaxRGB ? MaxRGB : magR;
            (*(outPixels + i)).green = magG > MaxRGB ? MaxRGB : magG;
            (*(outPixels + i)).blue = magB > MaxRGB ? MaxRGB : magB;
        }

        // free memory
        fftw_destroy_plan(planR);
        fftw_destroy_plan(planG);
```

```
    fftw_destroy_plan(planB);
    fftw_free(inR); fftw_free(outR);
    fftw_free(inG); fftw_free(outG);
    fftw_free(inB); fftw_free(outB);
}

int main(int argc, char** argv)
{
    if(argc < 4) {
        cout << "usage:\t" << argv[0] << " in outMag outPhase" << endl
                << "\t" << argv[0] << " -ift inMag inPhase out" << endl;
        return 0;
    }

    string arg = argv[1];

    if(arg != "-ift") {
        // FORWARD FOURIER TRANSFORM
        Image img(argv[1]);

        // get the length of the longer side of the image
        int squareSize = img.columns() < img.rows() ? img.rows() : img.colum

        // the geometry of our padded image
        Geometry padded(squareSize, squareSize);
        padded.aspect(true);

        // make image square
        img.extent(padded);

        // create templates for magnitude and phase
        Image mag(Geometry(squareSize, squareSize), "Black");
        Image phase(Geometry(squareSize, squareSize), "Black");
```

```

        // get image pixels
        img.modifyImage();
        Pixels pixelCache(img);
        PixelPacket* pixels;
        pixels = pixelCache.get(0, 0, squareSize, squareSize);

        // get magnitude pixels
        mag.modifyImage();
        Pixels pixelCacheMag(mag);
        PixelPacket* pixelsMag;
        pixelsMag = pixelCacheMag.get(0, 0, squareSize, squareSize);

        // get phase pixels
        phase.modifyImage();
        Pixels pixelCachePhase(phase);
        PixelPacket* pixelsPhase;
        pixelsPhase = pixelCachePhase.get(0, 0, squareSize, squareSize);

        // perform fft
        fft(squareSize, pixels, pixelsMag, pixelsPhase);

        // write changes
        pixelCache.sync();
        pixelCacheMag.sync();
        pixelCachePhase.sync();

        // save files
        mag.write(argv[2]);
        phase.write(argv[3]);
    }
    else {
        // BACKWARD FOURIER TRANSFORM
```

```
        Image mag(argv[2]);
        Image phase(argv[3]);

        // get size
        int squareSize = mag.columns();
        Image img(Geometry(squareSize, squareSize), "Black");

        // get image pixels
        img.modifyImage();
        Pixels pixelCache(img);
        PixelPacket* pixels;
        pixels = pixelCache.get(0, 0, squareSize, squareSize);

        // get magnitude pixels
        mag.modifyImage();
        Pixels pixelCacheMag(mag);
        PixelPacket* pixelsMag;
        pixelsMag = pixelCacheMag.get(0, 0, squareSize, squareSize);

        // get phase pixels
        phase.modifyImage();
        Pixels pixelCachePhase(phase);
        PixelPacket* pixelsPhase;
        pixelsPhase = pixelCachePhase.get(0, 0, squareSize, squareSize);

        // perform ift
        ift(squareSize, pixelsMag, pixelsPhase, pixels);

        // write changes
        pixelCache.sync();

        // save file
        img.write(argv[4]);
```

```
    }

    return 0;
}
```

You can get an archive with all the sources here.
Remember to remove the `-g` and `-O0` flags form the makefile when actually using it. (These are there for debug purposes)

Executing the command

```
user@host:~/fft$ 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++.

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