Matlab:

```
clear all;

% Audio Portion
fid = fopen('lpf_260_400_44100_80db.bin', 'rb');
ndim = fread(fid, 1, 'int');
nchan = fread(fid, 1, 'int');
dim0 = fread(fid, 1, 'int');
dim1 = fread(fid, 1, 'int');
dim2 = fread(fid, 1, 'int');
h = fread(fid, inf, 'float');
fclose(fid);

% Calculate H(w)
N = 2^14; % FFT size
f = (0:N-1)*dim1/N; % Make frequency vector for plotting
H = abs(fft(h,N)).^2; % Compute the magnitude reponse

%Plot the filter response
figure(1);
subplot(2, 2, 1);
stem(h);
title('h[n]');
set(gca, 'FontSize', 16);
grid on;

subplot(2, 2, 2);
semilogx(f, 10*log10(H));
xlim([0 dim1/2]);
ylim([-100 10]);
title('H(w)');
xlabel('Frequncy (Hz)', 'FontSize', 18);
ylabel('Gain (db)', 'FontSize', 18);
set(gca, 'FontSize', 16);
grid on;

% Parse the audio
[x, fs] = audio2bin('fireflyintro.wav');
%sound(x, fs);

% Apply the filter
x2 = conv(x, h);
[x3, fs3] = bin2audio('fireflyintro_pfp.bin');
%sound(x2, fs);
%sound(x3, fs3);

% Plot the spectrograms
nfft = 2^8;
overlap = round(0.8*nfft);
window = hamming(nfft);

subplot(2, 2, 3);
spectrogram(x, window, overlap, nfft, fs);
title('Before Filter');
set(gca, 'FontSize', 16);
grid on;

subplot(2, 2, 4);
spectrogram(x3, window, overlap, nfft, fs);
title('After Filter');
set(gca, 'FontSize', 16);
grid on;
```

```
clear all;

% Video Portion
[x1] = image2bin('cameraman.tif');
[x2] = image2bin('John Fiddle.jpg');
```

main.cpp

```
#include <iostream>
#include "../Shared/dspf.hpp"

const std::string filter = "lpf_260_400_44100_80db.bin";
const std::string firefly = "output\\fireflyintro.bin";
const std::string firepfp = "output\\fireflyintro_pfp.bin";
const std::string firertp = "output\\fireflyintro_rtp.bin";
const std::string img1 = "output\\cameraman.bin";
const std::string img1_out = "output\\cameraman_edge.bin";
const std::string img2 = "output\\John Fiddle.bin";
const std::string img2_out = "output\\John Fiddle.bin";

void audio_full() {
	DSPFile
		fin(firefly),
		lpf(filter),
		fout(firepfp, DSP::Mode::Write);

	if (!fin.ready() || !lpf.ready() || !fout.ready()) {
		return;
	}

	dsig h = lpf.read_all();
	dsig x = fin.read_all();

	int h_size = h.size(),
		d_size = x.size(),
		o_size = d_size + (h_size - 1);

	dsig out(o_size, 0);
	fout.Header = fin.Header;
	fout.Header.dim0 = o_size;

	// Apparently array access on vectors is ridiculously slow (visual studio compiler)
	float
		* pout = out.data(),
		* ph = h.data(),
		* px = x.data();

	for (int j = 0; j < h_size; ++j) {
		if (ph[j] == 0) { continue; }
		for (int i = 0; i < o_size; ++i) {
			if (!(i - j < d_size)) { break; }
			if (!(i - j < 0)) {
				pout[i] += ph[j] * px[i - j];
			}
		}
	}

	fout.write_h();
	fout.write_d(out.data(), o_size);
}

void audio_realtime() {
	DSPFile
		fin(firefly),
		lpf(filter),
		fout(firertp, DSP::Mode::Write);
```

```
	dsig h = lpf.read_all();

	int buf = h.size();
	dsig x(buf, 0);

	// Fix the header
	fout.Header = fin.Header;
	//fout.Header.dim0; // The circular buffer chops the tails
	fout.write_h();

	// Apparently array access on vectors is ridiculously slow (visual studio compiler)
	float
		* ph = h.data(),
		* px = x.data();

	int k, i = buf - 1;
	px[i] = fin.read_1();
	while (fin.ready()) {
		float y = 0;
		for (k = 0; k < buf; ++k) {
			y += ph[k] * px[(k + i) % buf];
		}
		i = (i + buf - 1) % buf;

		fout.write_d(y);
		px[i] = fin.read_1();
	}
}

float* conv2(const float* x, int mx, int nx, const float* h, int mh, int nh) {

}

void image_grayscale() {
	DSPFile
		fin(img1),
		fout(img1_out, DSP::Mode::Write);

	dsig x = fin.read_all();

}

void image_color() {

}

int main() {
	//audio_full();
	//audio_realtime();
	image_grayscale();

	system("pause");
	return 0;
}
```

dspf.hpp:

```
#pragma once
#include <string>
#include <memory>
#include <vector>
#include <fstream>

namespace DSP {
	static enum Mode { Read = 1, Write = 2, RealTime = 4 };
	static enum Type { Audio = 1, Image = 2, Video = 3 };
	static struct color { float r, g, b; };
	static float gray(color c) {
		return (0.2989f * c.r) + (0.5870f * c.g) + (0.1140f * c.b);
```

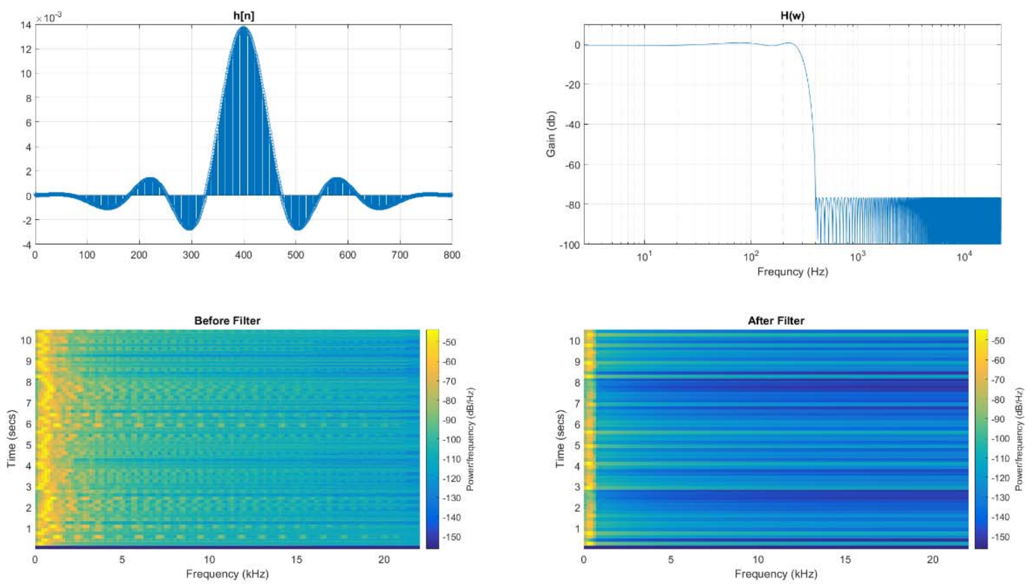
```
	};
}

typedef std::vector<float> dsig;
typedef std::vector<DSP::color> dpix;
typedef std::vector<std::vector<float>> dsig_block;
typedef std::vector<std::vector<DSP::color>> dpix_block;
struct dsh { int ndim, nchan, dim0, dim1, dim2; };

class DSPFile {
private:
	std::string file;
	std::fstream fid;
	DSP::Mode fmode;

	bool valid = true;
	void close() { fid.close(); valid = false; };

public:
	dsh Header;
	~DSPFile() { close(); }
	bool ready() { return valid; }
	DSPFile(std::string, DSP::Mode fm = DSP::Mode::Read);

	float read_1();
	dsig read_all();

	void write_h();
	void write_d(float*, int);
	void write_d(float);
};
```

dspf.cpp:

```
#include "dspf.hpp"
#include <iostream>
using namespace DSP;

DSPFile::DSPFile(std::string f, Mode fm) {
	file = f;
	fmode = fm;
	int mode = std::ios::binary;

	// TODO: Consider revising to handle read/write cases
	switch (fmode) {
	case Mode::Read:
		mode |= std::ios::in;
		break;
	case Mode::Write:
		mode |= std::ios::out | std::ios::trunc;
		break;
	};

	fid = std::fstream(file, mode);
	if (!fid) {
		std::cout << "Error fetching: " << file << std::endl;
		close();
		return;
	}

	if (fmode & Mode::Read) {
		fid.read(reinterpret_cast<char*>(&Header), sizeof(dsh));
	}
}

float DSPFile::read_1() {
	float data;
	fid.read(reinterpret_cast<char*>(&data), sizeof(float));
	valid &= !fid.eof();
```

```
        return data;
}


dsig DSPFile::read_all() {
        dsig data;
        float temp;
        while (true) {
                fid.read(reinterpret_cast<char*>(&temp), sizeof(float));
                if (fid.eof()) { break; }
                data.push_back(temp);
        }
        close();

        return data;
}

void DSPFile::write_h() {
        fid.write(reinterpret_cast<char*>(&Header), sizeof(dsh));
}

void DSPFile::write_d(float* data, int n) {
        fid.write(reinterpret_cast<char*>(data), sizeof(float) * n);
}
void DSPFile::write_d(float data) {
        fid.write(reinterpret_cast<char*>(&data), sizeof(float));
}
```

Output for Audio Portion



What is the benefit of zero-padding?

I'm still really confused on this point. It seems to me the only benefit would be to do in-place operations rather than allocating twice the memory. Perhaps some speed benefits from not having to check boundary conditions (resulting in conditional branches in a loop) but in modern architectures this effect would be small. What's more it changes the operation of convolution from addition to subtraction (really, how?!)

What is the benefit of using a circular buffer?

Easy. Shifting is expensive. Using a circular buffer and pointer arithmetic is much faster!