Beginner's Guide to the FFT-objects in Pd

Author: Frank Barknecht

Abstract

The Fourier transformation (FFT) is a powerful operation in digital signal processing. Pd includes objects that can be used to apply FFT, modify signals in the frequency domain, and transform the data back. This guide will try a hands-on explanation of how to use the [rfft~] and [rifft~] objects and give you at least a basic understanding of the process. This guide however is not meant to be a deep explanation of the underlying math, it will just give you enough knowledge to get started. For a deeper explanation I would recommend the wonderful DSP-Guide.

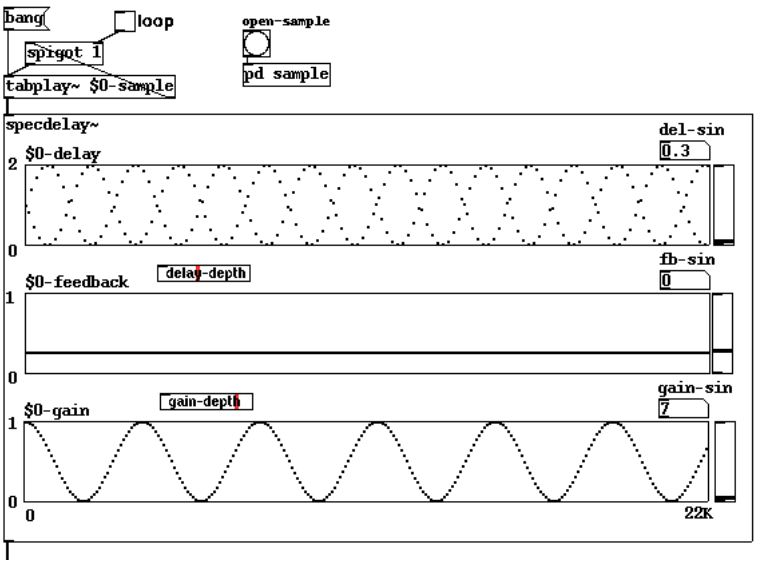

Figure 1: A spectral delay built with 98 Pd objects and free of charge.

So lets first have a look at what [rfft~] does: It will give you two signals. One is called the real, the other the imaginary part, but lets forget about this for now and look at it from a bit afar:

Generally a FFT will do a spectral analysis. It will calculate, what sine waves you need to add up to get the same signal as that played in the current signal block. Basically it will tell you the

frequencies and phases (first and second inlets) and amplitudes of a lot of [osc~] objects that, if you add them all up, would resynthesize your current signal. (You cannot directly use these [osc~] objects to resynthesize what comes from rfft~ but lets for a moment assume that we could.)

How many [osc~] objects you can control, will depend on the block-size: The FFT will generate control data for blocksize/2 oscillators. So with a blocksize of 64, you get frequencies, phase and amplitudes for 32 osc~s.

Now for some deep mathematical reasons all these [osc~] objects have fixed tunings: They all are multiples (harmonics) of Samplerate/Blocksize. So it starts at f0 = 0 Hertz, the next [osc~] would have a frequency f1= 1 * SR/BS, the next at f2=2*SR/BS up to the final one: f_final = (BS/2) * SR/BS == SR/2 or the Nyquist-frequency. For a blocksize of 16 and a samplerate of 48000 Hz this would be:

```
f0: 0
f1: 1 * 48000/16 = 3000
f2: 2 * 48000/16 = 6000
...
f8: 8 * 48000/16 = 24000
```

(Actually of course these are bs/2 + 1 frequencies, but 0 and Nyquist are special anyway so I thought I could cheat a bit. ;))

Because the frequencies are fixed and known, the rfft~ object doesn't need to specify them explicitly. It only needs to calculate the amplitude and the phase of every partial [osc~].

Now the tricky parts to understand are these:

[rfft~] will not directly output the amplitudes and the phases, but this strange thing called real and imaginary part. These carry exactly the same information about amplitude and phase, but encoded a bit differently than you are probably used to from working with [osc~]:

They are specified in a kind of polar coordinate system, where the amplitude is the radius (or distance from origin) and the phase is the angle of the polar coordinates. Re and Im however are cartesian coordinates (in the complex plane).

You can convert re/img-pairs to amplitude and phase using these formulas:

```
amp = sqrt(re^2 + im^2)
phs = arctan(im/re)
```

This is a standard cartesian to polar conversion, which you can read a bit more about in the wonderful DSP-Guide Chapter 8.

Most of the time you can skip calculating the phase, but more on that later.

The amplitude calculation in Pd lingo looks like this:

```
amp
```

```
 [rfft~]
 |\    |\
 [*~]  [*~]
 |    /
 |   /
 [+~ ] <= just inserted for clarity, you can also di-
rectly go to sqrt~
 |
 [sqrt~] or [q8_sqrt~], which is much faster.
```

The real and imaginary part (or the phase and amplitudes) are encoded inside the signal blocks, that [rfft~] outputs. The first pair of samples of the left and right outlet~s of [rfft~] contains the info about amplitude and phase for the first [osc~] in our big oscillator bank, that has frequency f0. Each second sample pair contains info for the next osc~ with frequency f1 and so on up to the sample pair number "blocksize/2", which contains the amp and phase for the final oscillator at Nyquist frequency. The rest of the block always is zero, as we don't have oscillators for that.

Some real world data might be useful: Assume we have a blocksize of 8. Then a block of samples might look like this, when print~ed:

```
orig:
0.13004  0.26951  0.40352  0.52934  0.64446  0.74649  0.83341  0.90344
```

If you send this through [rfft~] you will get this:

```
img:
0        1.0317   0.41679  0.17191  0        0        0        0


re:
4.4602   -0.58717 -0.46243 -0.44167 -0.43737 0        0        0
```

Sending these two to [rifft~] and dividing by blocksize 8 will give you the the original signal block back.

You can also calculate the amplitudes like above, which of course is easy for our first sample:

```
amp = sqrt(4.4602^2 + 0^2) = 4.4602
```

For the next two, lets use the handy calculator that is the Python command line:

```
>>> import math
>>> math.sqrt(1.0317 * 1.0317 + -0.58717 * -0.58717)
1.1870861379445048
>>> math.sqrt(0.41679 * 0.41679 + -0.46243 * -0.46243 )
0.62253948388837155
```

and so on.

Actually to get the correct amplitudes you would need to normalize the re/im pairs here as well, I just skipped that. (Normalizing the amplitude needs to take into account that we added both imaginary and real part in sqrt(re^2 + im^2) so we need to dividing them by an additional 2, which in our example is: $4 == 8/2$)

Here's the full scoop as reported by Pd:

```
amp:
4.4602    1.1871    0.62254   0.47394   0.43737   0         0         0
```

See attached fft-up-close.pd to try this on your own.

This means, that resynthesizing this signal at SR=48000 would be similar to using oscillators like this:

```
[osc~ 0]
|
[*~ 4.4602]

[osc~ 6000]
|
[*~ 1.1871]

[osc~ 12000]
|
[*~ 0.62254]

[osc~ 18000]
|
[*~ 0.47394]

...
```

and so on (Note that without normalizing these values are too loud.)

However: All these oscillators would also need to have their phases set accordingly, so you cannot just use above oscillator bank directly in real life.

The inverse FFT objects like [rifft~] will accept the amplitude and phase information in the real/imaginary format directly. This means, you can think of the [rifft~] as a resynthesis bank of blocksize/2 oscillators like above with real and imaginary inputs instead of amplitude and phase input, and every oscillator inside [rifft~] is spaced Samplerate/Blocksize Hertz apart.

As fft~-help.pd and my calculation above shows, connecting an [rfft~] to a [rifft~] will just pass the signal practically unchanged (it's just a bit louder afterwards, that's why you normally normalize

it by dividing the output by the blocksize like [/~ 64]). Depending on Windowing and Overlap you need to use a different normalization factor.

Of course it will only get interesting if we wreck havoc to the re/im frequency data in the meantime.

A simple FFT-based modification is shown in I03.resynthesis.pd: Here every re/im sample pair (or every amplitude/phase-info for the respective "oscillator" in rifft~) is multiplied by some value retrieved from the gain table through tabreceive~. If this table has a 1 at a certain sample, this data is passed unchanged, if it has a 0 at another sample, than that oscillator is muted. This is a filter operation, and it only affects the amplitudes of the internal oscillators.

You might ask: "Why only the amplitudes? What about the phases? You said, they are also encoded in the re/im data? Are you cheating again?!" Read on.

If we scale the re/im pair by a (positive) value x, then the amplitudes will be scaled by (the absolute of) x as well:

```
amp(x*re,x*im) = sqrt((x * re)^2 + (x * im)^2)
               = sqrt (x^2 * (re^2+im^2))
               = sqrt(x^2) * sqrt(re^2+im^2)
               = |x| * amp(re,im)
```

However the phases will stay the same! Proof:

```
phs(x*re,x*im) = atan(x*im/x*re) = atan(im/re) = phs(re,im)
```

Get it? That's why for such modifications you can omit the phase calculation with atan etc.

Note that you need to do this multiplication *on every block* again and again, because the data coming out of [rfft~] is constantly updated - it still is an audio signal! That's why a [tabreceive~] is used: Although the table received is not changing all the time, we still need to read it again on every block and make a signal out of it.

Now for a simple, amplitude-dependent gating or filtering, you first need to calculate the actual amplitude using the formula above. Then compare it to a value and multiply the original re/im-pairs with 0 or 1 depending on the result to change the amplitudes used in the resynthesis.

Attached specgate.pd illustrates this and also has a comparison of the windowed and unwindowed fft, that affects the quality of the result and also your normalization factors.

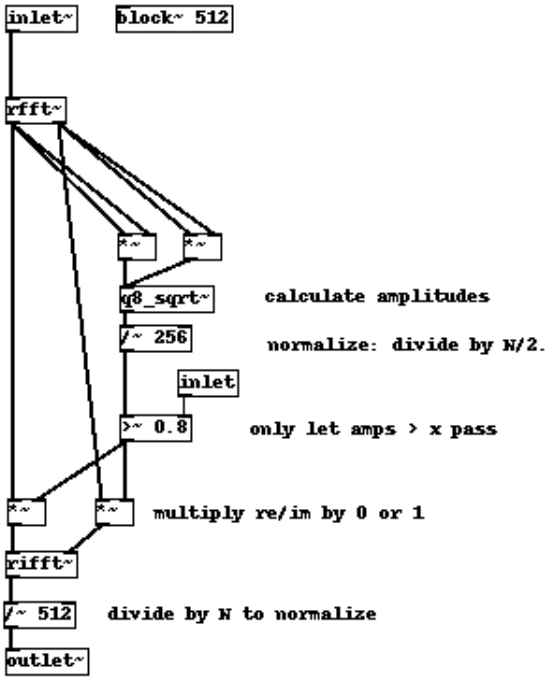

Figure 2: Spectral gating in Pd, unwindowed version.