Processingを用いた信号処理
フーリエ変換のフルスクラッチ

フーリエ変換（Fourier Transform）

- 時間tに関する周期関数f(t)が、周波数Ωに関する関数g(Ω)に変換する数学的処理

$$g(\Omega)=\int_{-\infty}^{+\infty} f(t)\exp(-j\Omega t)\,dt$$

- f(t)がどのようなsin波の組み合わせでできているのかを調べる処理ともとれる

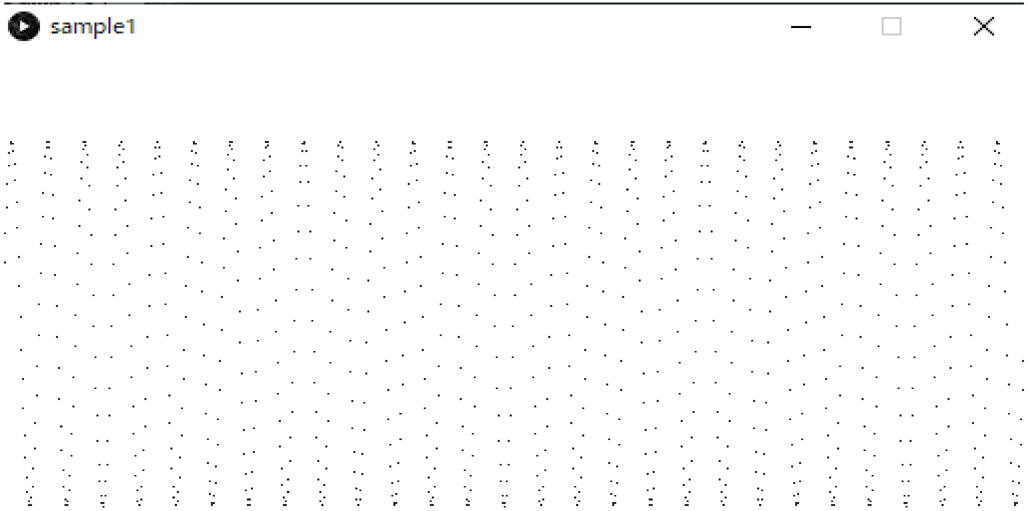
フーリエ変換の基本的な考え方

- すべての関数はsin波の合成波としてあらわされる。

$$f(t)=a_0+\sum_{k=1}^{\infty}\left(a_k\cos\left(\frac{2\pi kt}{T_0}\right)+b_k\sin\left(\frac{2\pi kt}{T_0}\right)\right)$$

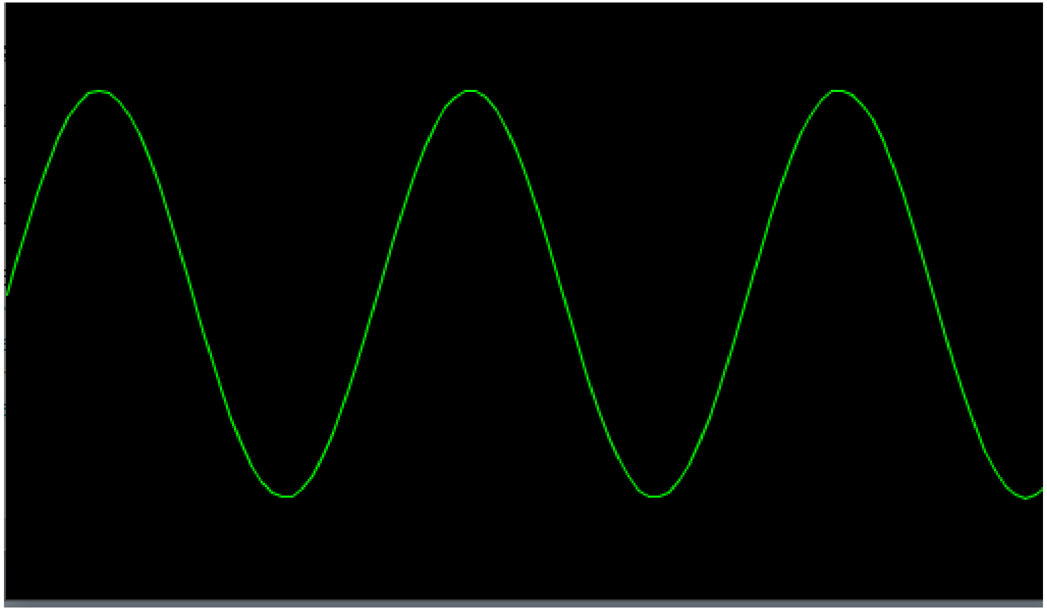
プログラミング１：sin波の表示

Sample1.pde



補足：ちょっと(個人的に)かっこよく

- Sample1-2.pde



確認すべきこと

- A, fを変えるとどうなるか

    (Aは今回は「見やすさ」重視で適宜値を変えています)

- sinをcosに変えるとどうなるか

プログラミング２：合成波の作成１

- 例えば、sinとcosの合成波を作ってみる(sample2)
  - y = sin(x)+cos(x)
  - これを表示するには、sample1-2(もしくはsample1)を次のように変更すればよい
    - 変更前:
      - y[t] = A * sin(2 * PI * t / (T0 * fs));
    - 変更後:
      - y[t] = A * sin(2 * PI * t / (T0 * fs)) + A * cos(2 * PI * t / (T0 * fs)); Aの値は適宜適切に変更する

sinとcosの割合(それぞれの振幅)を変えてみる

- Sample2-2
  - 例えば、新しい変数Bを導入し、cosの振幅をBにする
    - Aの定義辺りに追加：float B = （Aとは違う好きな値）;
    - A * cos(2 * PI * t / (T0 * fs)); を
      B * cos(2 * PI * t / (T0 * fs)); に変更して実行

プログラミング２を通して

- 同じ周波数のsin波とcos波を足すと、スタート位置のずれたsin波になる

- つまり、$f(t)=a_0+\sum_{k=1}^{\infty}\left(a_k\cos\left(\frac{2\pi kt}{T_0}\right)+b_k\sin\left(\frac{2\pi kt}{T_0}\right)\right)$

  この式は結局sin波だけを足し合わせている……とみることができる

プログラミング３：合成波の作成２
周波数の違うsin波を足し合わせてみる

- Sample3
  - sample2-2に以下の２変数を追加
    - float f_2 = 880;
    - float T0_2 = 1 / f_2;
  - さらに、以下のように変更してみる
    - 変更前：
      - y[t] = A * sin(2 * PI * t / (T0 * fs)) + B * sin(2 * PI * t / (T0 * fs);
    - 変更後：
      - y[t] = A * sin(2 * PI * t / (T0 * fs)) + B * sin(2 * PI * t / (T0_2 * fs));

確認すべきこと

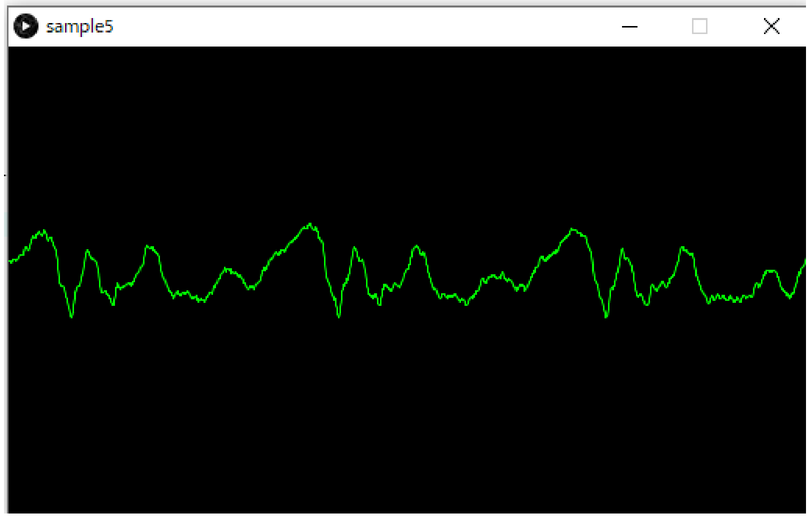
- A、Bの値をそれぞれ同じにしたり、変えたりするとどうなるか
- f_2の値を変えてみるとどうなるか
- ３つ目のsin波（振幅C、周波数f_3）をたしてみるとどうなるか

大量のsin波の合成で様々な関数を作ってみよう

- 矩形波
  - sample4_SquareWave.pde
- のこぎり波
  - sample4_SawWave.pde
- 三角波
  - sample4_TriangleWave.pde

PCから音を拾ってみる

- Minimライブラリを利用する
  - sample5.pde



sample5

```
import ddf.minim.*;
Minim minim;
AudioInput in;
int N = 1024;
float fs;
float[] y = new float[N];

void setup(){
  size(512, 300);
  background(0);

  minim = new Minim(this);
  in = minim.getLineIn(Minim.STEREO, N);
  fs = in.sampleRate();
}

void draw(){
  background(0);
  y = in.mix.toArray();
  stroke(0, 255, 0);
  for(int i=1; i<N; i++){
    line(map(i-1, 0, N, 0, width), height/2-y[i-1]*100, map(i, 0, N, 0, width), height/2-y[i]*100)
  }
}
```

離散フーリエ変換

- 離散時間上で機能するフーリエ変換
- PC上で扱うフーリエ変換はすべてこれ

$$g[\Omega] = \sum_{-\infty}^{\infty} f[t] e^{-j\Omega t}$$

自然対数eについて

- eは2.7……となる数字
- 1%のガチャを100回ひいて外れる確率の逆数とほぼ同じ数
  - 数式だと、 $\frac{1}{\left(\frac{99}{100}\right)^{100}}$
  - nをすごく小さい数としたとき、

    n%のガチャをn回引いて外れる確率の逆数がe
- この数のj(複素数)乗というのは、sinとcosに分けられる

オイラーの公式

- オイラーの公式から、

$$e^{jx}=\cos(x)+j\sin(x)$$

- なので、実数部分と虚数部分を分離できる
- これで離散フーリエ変換を書き直せば、

$$g[\Omega]=\sum_{-\infty}^{\infty} f[t](\cos(\Omega t)-jsin(\Omega t))$$

Ωについて

- 正規化角周波数
- 角周波数ω=2πfをサンプル数Nで割ったもの

$$\Omega=\frac{2\pi f}{N}$$

最終的な離散時間フーリエ変換の形

$$g[\Omega]=\sum_{f=0}^{N} f[t]\left(\cos\left(\frac{2\pi f}{N}t\right)-j sin\left(\frac{2\pi f}{N}t\right)\right)$$

ここで、$0 \le t \le N-1$の整数である

離散時間フーリエ変換の実装

- 以下の式を見ながら、実際にプログラムに起こしてみる
- 実装する式

$$g[\Omega]=\sum_{f=0}^{N} f[t]\left(\cos\left(\frac{2\pi f}{N}t\right)-j sin\left(\frac{2\pi f}{N}t\right)\right)$$

ただし、0≤t≤N-1である

離散時間フーリエ変換の発展

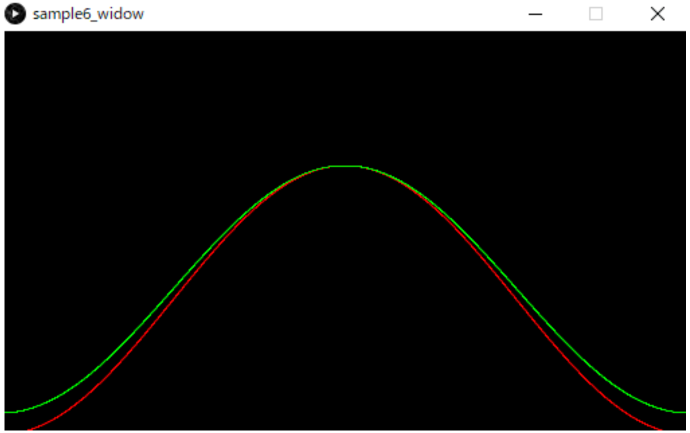
- FFT（高速フーリエ変換）
  - DFTの遅さをアルゴリズムによって解決している
  - 様々なアルゴリズムがある
- IDFT（逆離散時間フーリエ変換）
  - 周波数関数から時間関数に戻す、フーリエ変換の逆の作業

補足：窓関数について

- フーリエ変換は、理論上周期関数についてのみ行える処理である
- しかしマイクから拾ってきた音は完全な周期関数ではない
- そのため、マイクからの音を疑似的に周期関数にしてやる処理が必要
- その処理をする関数のことを窓関数と呼ぶ

窓関数の種類

- hamming窓(緑)
  - $w(x) = 0.54 - 0.46\cos(2\pi x),\ 0 \le x \le 1$
- hanning窓(赤)
  - $w(x) = 0.5 - 0.5\cos(2\pi x),\ 0 \le x \le 1$



窓関数を利用して
短時間フーリエ変換を完成させる

- 注意点
  - 元の時間関数が-1～１になるようにする
    - この作業を正規化という
  - 窓関数と時間関数は単純な掛け算をすればよい
  - あとの処理は先ほどの離散フーリエ変換と同じでよい