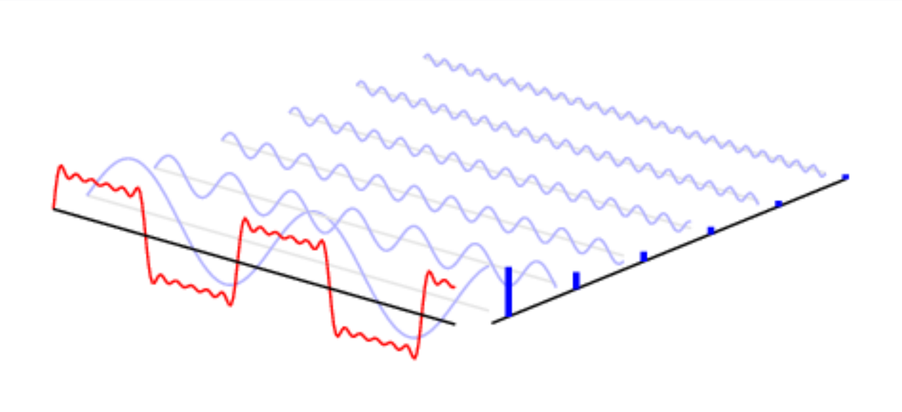
スクリプト言語プログラミング
Pythonによる数値解析

第14回講義資料

箕原辰夫

フーリエ変換

- フーリエ変換・逆フーリエ変換
- 離散フーリエ変換・高速フーリエ変換
- 音声データの高速フーリエ変換
- 画像データの高速フーリエ変換



フーリエ変換について

- フーリエ（Jean Baptiste Joseph Fourier）は、1796年代数方程式の実数解に関する「フーリエの定理」を発表した。これは、周期関数を正弦（sine関数）や余弦（cosine関数）の和を用いて現れるもので、周期関数を解析して、時間関数を周波数の軸に投影できる、所謂「周波数スペクトル」による様々な物理化学的な解析を可能とするものであった。これを「フーリエ級数」と呼ぶ。
- 実数の関数だけに留まらず、オイラー（Leonhard Euler）の以下の複素数に関する公式を用いて、複素数も含めて展開する方が簡単に記述できる。複素数を含めた形で記述されたものをフーリエ変換（Fourier Transform）と呼ぶ。
  - $e^{i\theta} = \cos\theta + i\sin\theta$
- これに対して、実数のcos関数だけを用いて変換するものを、cos変換（Cosine Transform）と呼ぶ。cos変換は、対象となる関数が偶関数（$f(t) = f(-t)$）の場合に使用される。偶関数の場合、フーリエ変換の虚数成分は0になるため、実数成分（cos成分）のみを考慮すれば良く
- フーリエ変換は、周期関数を周波数領域に投影できるため、周期関数、これは複雑な波動関数のことが多いが、その解析のために工学的には幅広く用いられている。
- フーリエ変換の逆を行なう逆フーリエ変換（Inverse Fourier Transform）は、周波数領域から時間関数を生成できるために、周期的な波動による関数を合成するために、同じように工学的に様々な用途に用いられている。

フーリエ変換・逆フーリエ変換の概略

- フーリエ変換：波形を、Cosine, Sine波の重なりに分解することができる（周期・振幅・位相を変える）
- 逆フーリエ変換：すべての波形は、Sine波の合成で再現できる

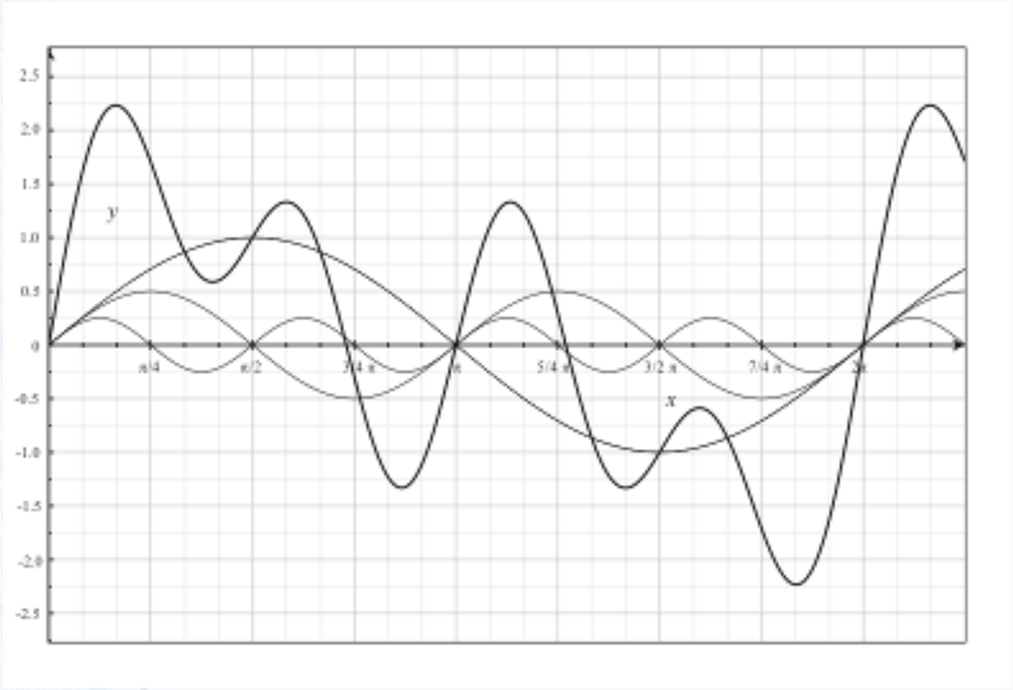
$$\hat{f}(\xi) := \int_{-\infty}^{\infty} f(x)\, e^{-2\pi i x \xi}\,dx$$

$$f(x) := \int_{-\infty}^{\infty} \hat{f}(\xi)\, e^{2\pi i x \xi}\,d\xi$$

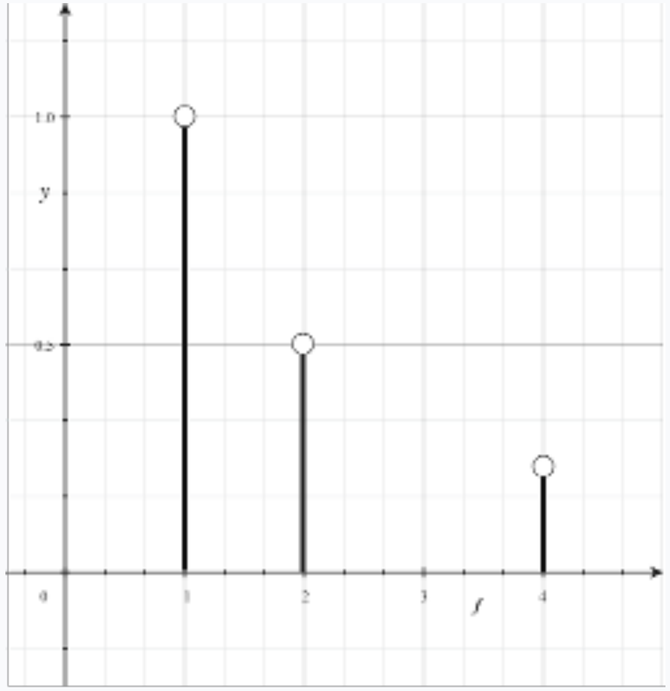
- Wikipediaの「フーリエ変換」のアニメーション参照

三角関数による分解と線（離散）スペクトル

- 三角関数による合成波形・分解

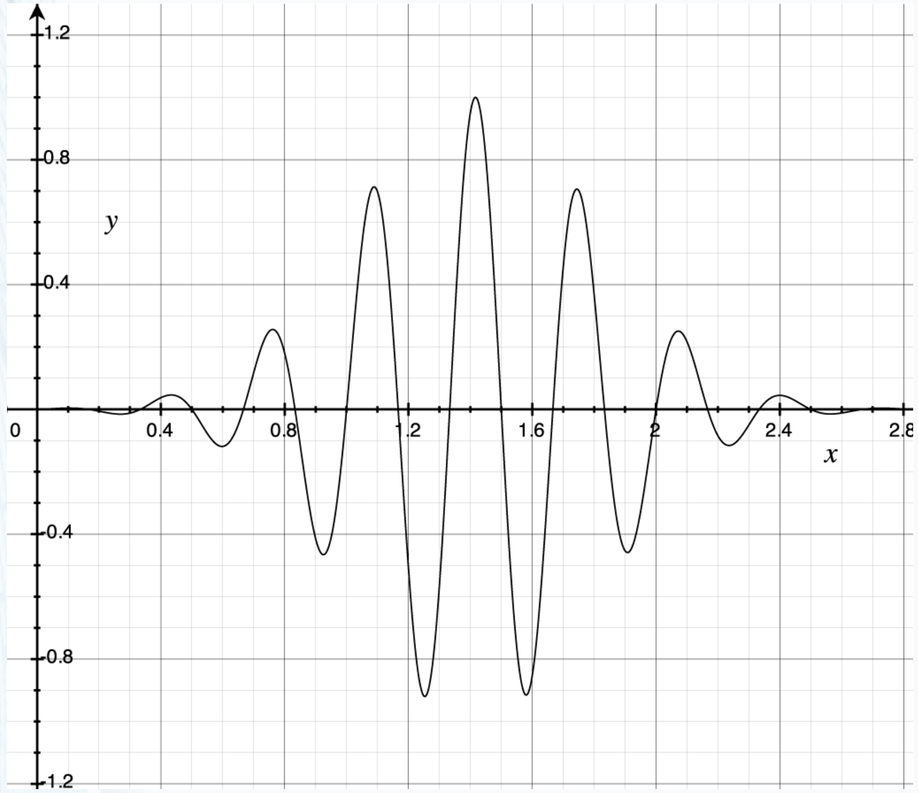


- 周波数領域での線スペクトル

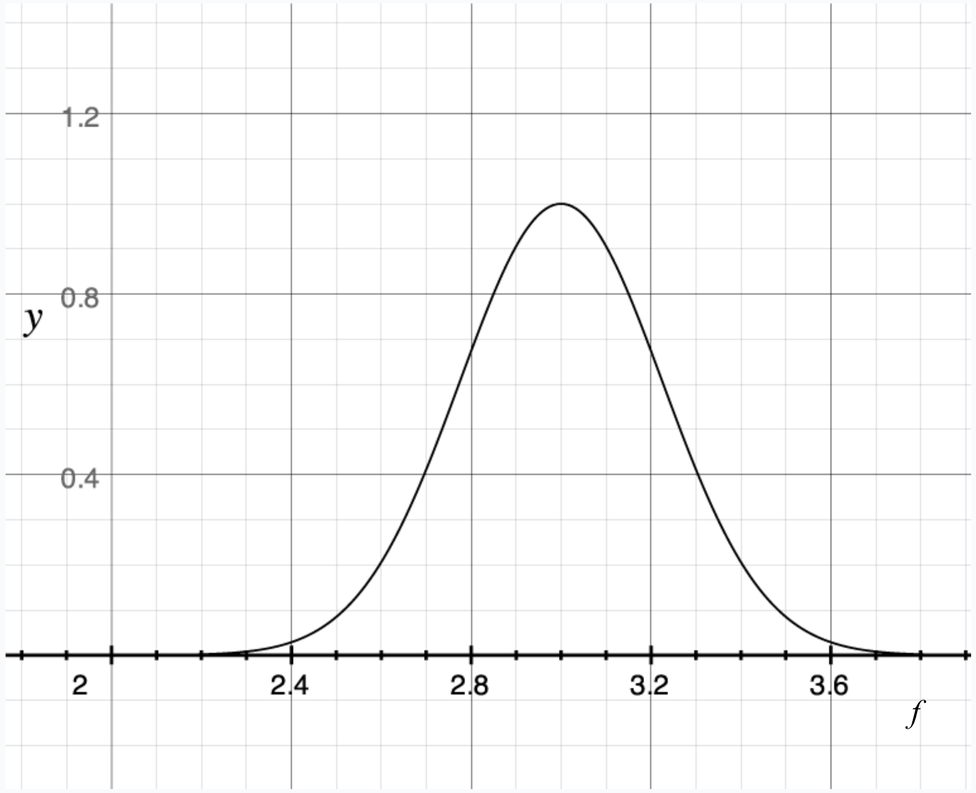


波動関数と連続スペクトル

- 元の波動関数



- 連続スペクトル



離散フーリエ変換について

- 様々な録音・録画機器で得られた映像・画像・音声（録音）については、アナログの波動を標本化（sampling）し、量子化（quantization）して、離散化情報、即ちデジタル情報として表現されている。そのため、コンピュータ上では、離散化された周期波動関数に対する離散フーリエ変換・逆変換（Discrete Fourier Transform/Inverse Transform：DFT/IDFT）が用いられている。これは、フーリエ級数の応用版になっている。
- 離散フーリエ変換・逆変換においては、いくつかの高速化のアルゴリズムが考えられており、それらは、総称して「高速フーリエ変換・逆変換」（略称FFT・IFFT）と呼ばれている。これは、高速フーリエ変換の計算量$O(n^2)$を、$O(n \log n)$に落とせるために、多くの場合は、離散フーリエ変換はFFTで実装されているいることが多い。
- それぞれの立ち位置をまとめると以下のようになる
  - 実数のフーリエ級数→複素数のフーリエ変換→
    複素数の離散フーリエ変換→高速フーリエ変換

離散フーリエ変換の概略

- 離散フーリエ変換（DFT: Discrete Fourier Transform）
  - 離散化されたデジタル信号の周波数解析などに使われるフーリエ変換
  - 離散フーリエ変換

$$F(t) = \sum_{x=0}^{N-1} f(x) e^{-i \frac{2\pi t x}{N}}$$

  - 逆離散フーリエ変換

$$f(x) = \frac{1}{N} \sum_{t=0}^{N-1} F(t) e^{i \frac{2\pi x t}{N}}$$

- 高速フーリエ変換（FFT: Fast Fourier Transform）は、離散フーリエ変換を高速に行なうアルゴリズム

離散コサイン変換

- 複素数領域において、虚数軸を含めた計算になると、複素数を計算しなければならず、リアルタイム（実時間）でメディアを合成・解析するためには、かなりの負荷となる。
- 実関数を元にした変換においては、エルミート対称になることがわかっている。
- 離散フーリエ変換を実数の領域だけで行なうのが、離散コサイン変換・逆変換である。この変換は以下のような公式で変換が可能になる。
- これらメディアの大量のデータサイズの圧縮に用いられることが多く、JPEG/MPEGといった画像・映像・音声のデジタル表現形式の圧縮方法の符号化および復号に用いられている。
- 以下は、DCT-IIとDCT-IIIと分類された離散コサイン変換で、 離散コサイン変換と逆変換を表している

$$X_k = \sum_{j=0}^{N-1} x_j \cos\left(\frac{\pi}{N}\left(j+\frac{1}{2}\right)k\right)$$

$$x_i = \frac{1}{2}X_0 + \sum_{k=1}^{N-1} \cos\left(\frac{\pi}{N}\left(j+\frac{1}{2}\right)k\right)$$

離散フーリエ変換の定義

- アナログ値における有限区間でのフーリエ変換と同様にが区間０～ 以外で０であると仮定し, 均等な回の標本化 (サンプリング) を行なう。サンプリング時間の列　を仮定するとフーリエ変換は以下のように記述できる。

$$X(f_k) = \frac{T^{\omega}}{N} \sum_{j=1}^{N-1} x(t_j) e^{-i2\pi kj/N}$$

- この式から、係数 T^{ω}/Nを取り除いた

$$y(f_k) = N/T^{\omega} \times X(f_k)$$

を 離散フーリエ変換と呼ぶ。

- T^{ω}は「サンプリング区間」、N/T^{ω}は「サンプリング周波数」と呼ばれる。また更に, $y(f_k) = y_k$ および $x(t_j) = x_j$と置くと，離散フーリエ変換は，以下のように書き表すことができる。

$$y_k = \sum_{j=0}^{N-1} \omega_N^{kj} x_j$$

- この式において、ω_Nは 回転因子(twiddle factor) と呼ばれ、以下のように定義される。

  回転因子は $\boldsymbol{y}$ と$\boldsymbol{x}$をベクトル表記した場合、ω_N^{kj}を行列としても書き表すことが可能である

$$\omega_N = e^{-i2\pi/N}$$

離散フーリエ変換の逆変換と各周波数のスペクトラム

- 離散フーリエ逆変換は、$y_k = y(f_k)$と$x_j = x(t_j)$、および回転因子$\omega_N = e^{-i2\pi/N}$を用いて以下のように記述することができる。

$$x_j = \frac{1}{N}\sum_{k=0}^{N-1} \omega_N^{-jk} y_k \quad (j = 0,1,\cdots,N-1)$$

- 標本化定理から、標本化周波数f_sを$f_s = N/T^{\omega}$と置くと、$f_n = f_s/2$までの周波数成分を持つ波動が離散フーリエ逆変換によって再現可能となる。

- f_nは　ナイキスト周波数（Nyquist　frequency）と呼ばれる。

- $y_k = y(f_k)$が複素数になることから、各周波数に関して、それぞれ以下の式によって求める。
  - 振幅スペクトラム：

$$\left|y(f_k)\right| = \sqrt{Re(y_k)^2 + Im(y_k)^2}$$

  - パワー・スペクトラム：

$$\left|y(f_k)\right|^2 = Re(y_k)^2 + Im(y_k)^2$$

  - 位相角スペクトラム：

$$\arg(y(f_k)) = \frac{Im(y_k)}{Re(y_k)}$$

離散フーリエ変換から高速フーリエ変換へ

- 離散フーリエ変換を回転因子の行列を用いてベクトル表記で表すと次のような式になる。

$$\boldsymbol{y} = F_N \boldsymbol{x} \text{ ただし, } \boldsymbol{y} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \end{bmatrix}, \boldsymbol{x} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \end{bmatrix}, F_N = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_N^1 & \omega_N^2 & \cdots & \omega_N^{N-1} \\ 1 & \omega_N^2 & \omega_N^4 & \cdots & \omega_N^{2(N-1)} \\ 1 & \vdots & \vdots & & \vdots \\ 1 & \omega_N^{N-1} & \omega_N^{2(N-1)} & \cdots & \omega_N^{(N-1)(N-1)} \end{bmatrix}$$

- この変換行列を計算するのには計算量として$O(N^2)$掛かる。この計算量を少なくする方式として, 高速フーリエ変換 (Fast Fourier Transform) のアルゴリズムがいくつか提案されている。
- 主に次の方式に分かれる
  - 時間間引き方式
  - 周波数間引き方式

- 時間間引き方式のCooley-Turkey型FFTアルゴリズムでは、サンプリングされた値が$N = 2^m$個あるときに、段数mで係数を計算していくため$O(N \log_2 N)$で計算量が収まる。
- 離散フリーエ変換の式において、偶数列と奇数列で分けると以下のようになる。

$$\begin{aligned} y_k &= \sum_{r=0}^{N/2-1} x_{2r} \omega_N^{(2r)k} + \sum_{r=0}^{N/2-1} x_{2r+1} \omega_N^{(2r+1)k} \\ &= \sum_{r=0}^{N/2-1} x_{2r} \omega_N^{2rk} + \omega_N^k \sum_{r=0}^{N/2-1} x_{2r+1} \omega_N^{2rk} \end{aligned}$$

- ここで以下の等式が成り立つ。

$$\omega_N^2 = e^{-i2\cdot 2\pi/N} = e^{-i2\pi/(N/2)} = \omega_{N/2}$$

高速フーリエ変換の定義

- 前のスライドの最後の等式を用いて、離散フーリエ変換は以下のように記述することができる。

$$y_k = \sum_{j=0}^{N-1} x_j \omega_N^{kj} = X_k^{(0)} + \omega_N^k X_k^{(1)} \quad (k = 0,1,\cdots,N-1)$$

$$\left.\begin{aligned} X_{k'}^{(0)} &= \sum_{r=0}^{N/2-1} x_{2r} \omega_{N/2}^{rk'} \quad X_{k'+N/2}^{(0)} = X_{k'}^{(0)} \\ X_{k'}^{(1)} &= \sum_{r=0}^{N/2-1} x_{2r+1} \omega_{N/2}^{rk'} \quad X_{k'+N/2}^{(1)} = X_{k'}^{(1)} \end{aligned}\right\} \quad (k' = 0,1,\cdots,\frac{N}{2}-1)$$

- この式において, および は, データ点数が の離散フーリエ変換となっている。 これが第m段目の計算になる。 これを再帰的に第 1 段目まで適用すると、すべての値を求めることができる。

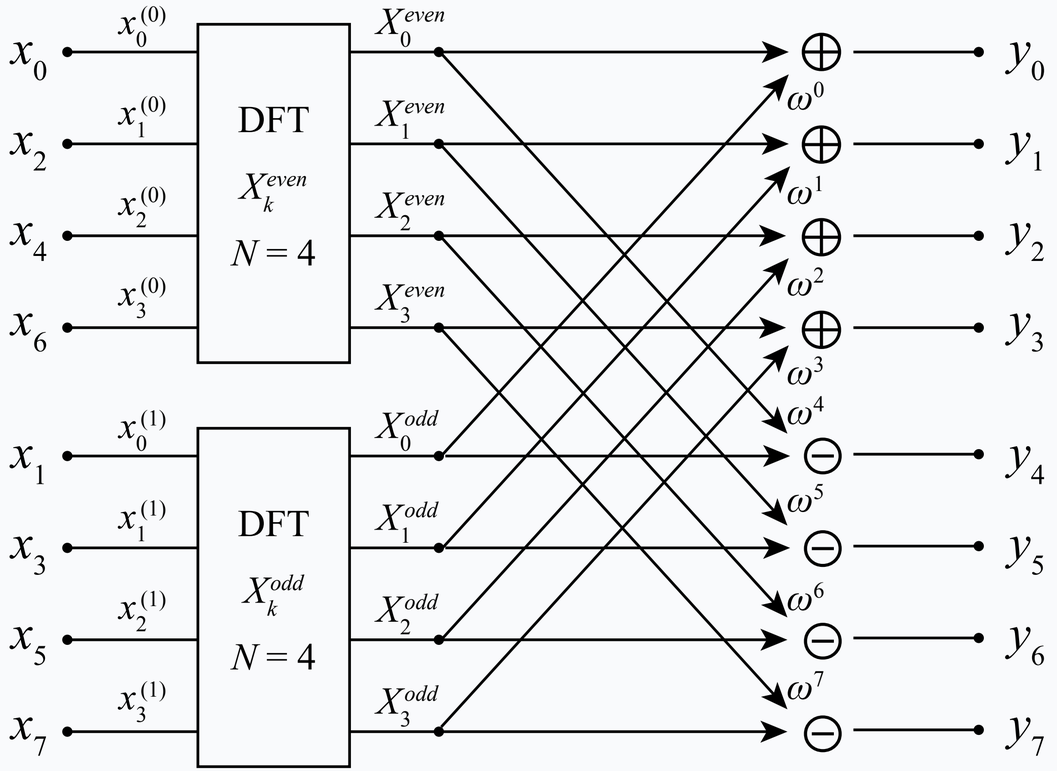
- データ総数が$N = 2^m$であるときに，入力データx_kの順序を、その添え字を 2 進数表記で表したときに、最上位桁から最下位桁を反対にして並べ替えた結果を、$X_0, X_1, \cdots, X_{N-1}$と表記すれば, p段目ではデータ点数が$N_p = 2^p$のグループが $N/N^p = 2^{m-p}$ 個あり、 各グループの要素は$\omega_{N_p}^{k+N_p/2} = -\omega_{N_p}^k$を用いて、 以下のように偶数列と奇数列に分解される。

$$\left.\begin{aligned} X_k &= X_k^{even} + \omega_{N_p}^k X_k^{odd} \\ X_{k+n_p/2} &= X_k^{even} - \omega_{N_p}^k X_k^{odd} \end{aligned}\right\} \quad (k = 0,1,\cdots,N_p/2-1)$$

- これを$p = 1$から始めて、$p = m$になるまで行なえば良い。

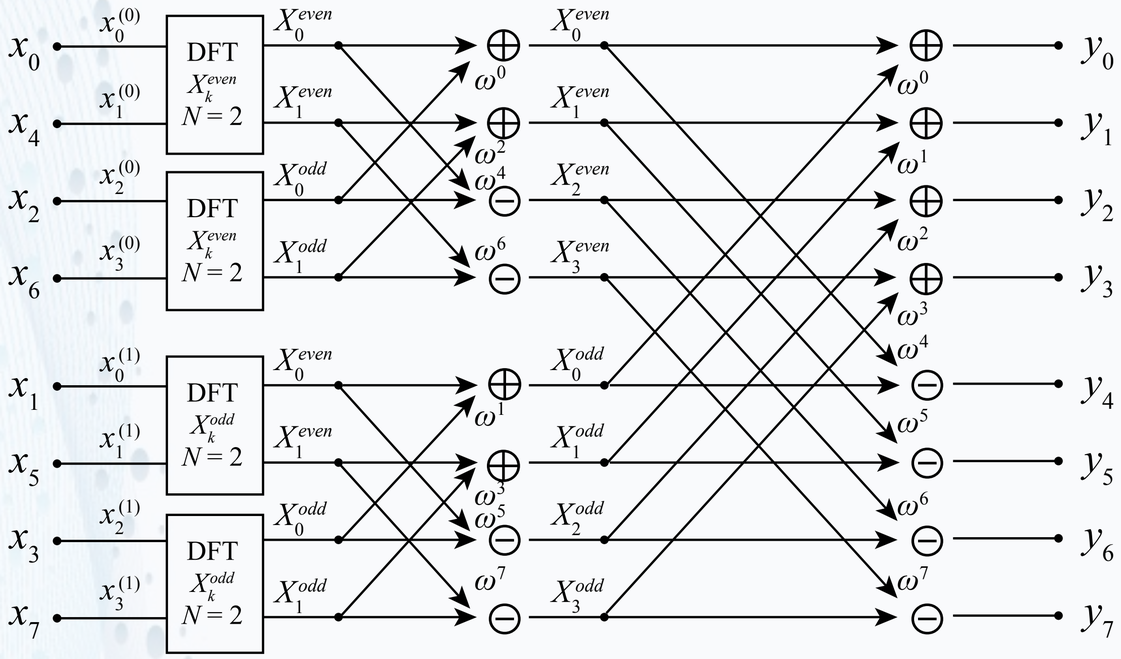
高速フーリエ変換の1つの段の模式図

- いまN=8のときに、偶数列と奇数列に分けてy_kを求める模式図が以下のようになる。DFT（N=4）の内部も同じように、交差で求められる



高速フーリエ変換の複数段

- N=8でDFT（N=2）を4つ用いた場合の模式図



- N=8のときの入力のビット反転（bit reversal：ビットの前後の反転）

入力並び	2進数表示	ビット反転	データ並び
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

高速フーリエ変換のプログラム概略

- 高速フーリエ変換の概略を以下のように、添え字のビット反転をする関数と、第m段の計算をする関数に分けて、書き表すことができる。

```
import math

# 添え字のビットを反転する関数
#（引数は添え字の数とビット長）
def bitInverse( number, n ):
    p = int( math.log( n, 2 ) )
    bi = 0
    for i in range( p ):
        base = 2**(p–1–i)
        orgbit = number & 1
        reversebit = base * orgbit
        bi = bi + reversebit
        number >>= 1
    return bi
```

```
# FFTを計算する関数（引数mは段数）
def FFT( m ):
    global X, N      # X, Nは大域変数とする
    for p in range( 1, m+1 ):
        Np = 2 ** p
        for i in range( N // Np ):
            for j in range( Np // 2 ):
                theta = j * 2 * math.pi / Np
                w = complex( math.cos( theta ),
                             –math.sin( theta ) )
                k = Np * i + j
                Xeven = X[ k ]
                Xodd = X[ k + Np//2 ]
                X[ k ] =Xeven + w * Xodd
                X[ k + Np//2 ] = Xeven – w * Xodd
```

scipyでの音声ファイルのフーリエ変換

- 以下のようにして、Sample.wavファイルからの音声データをフーリエ変換した結果を、matplotlibでグラフにすることができる

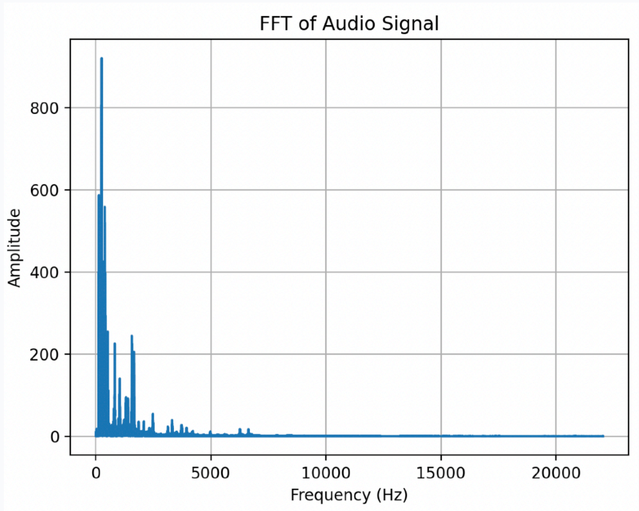
```
import numpy as np
from scipy.io import wavfile
from scipy.fft import fft, fftfreq
import matplotlib.pyplot as plt

# 音声ファイルの読み込み（WAV形式）
rate, data = wavfile.read("sample.wav")

# モノラル化（ステレオの場合）
if data.ndim > 1:  data = data.mean(axis=1)

# FFTの実行
N = len(data)
T = 1.0 / rate
yf = fft(data)
xf = fftfreq(N, T)[:N//2]

# スペクトル表示
plt.plot(xf, 2.0/N * np.abs(yf[0:N//2]))
plt.xlabel("Frequency (Hz)")
plt.ylabel("Amplitude")
plt.title("FFT of Audio Signal")
plt.grid()
plt.show()
```



常微分方程式の数値的解法

- 常微分方程式（ODE: Ordinary Differential Equation）は、解析解（積分形が求められること）が求められないことが多い
- そのため、数値を具体的に微分方程式に当てはめていき、その結果の数値解を求めることが多い
- 物理シミュレーションや物理的な挙動予測などにおいて使われている
- 方式は、次のような３つの方式が使われることが多い
  - オイラー法
  - 改良オイラー法
  - ルンゲ・クッタ法（この発展として、Dormand-Prince法がある）
- それ以外に、以下のような方式もある
  - ホイン法
  - リチャードソン補外（精度を増すために用いられる）
  - 暗黙法（剛性問題：急激に解が変化するような問題に用いられる）

常微分方程式の表現

- 一般的には以下の形で離散化する

$$x'(t) = f(t)$$

$$x(t) = x_0 + \int_0^t f(s)ds$$

$$x(t_n) = x_0 + \int_0^{t_n} f(s)ds$$

$$x'(t) = f(t, x(t)), (a \le t \le b), x(a) = x_0$$

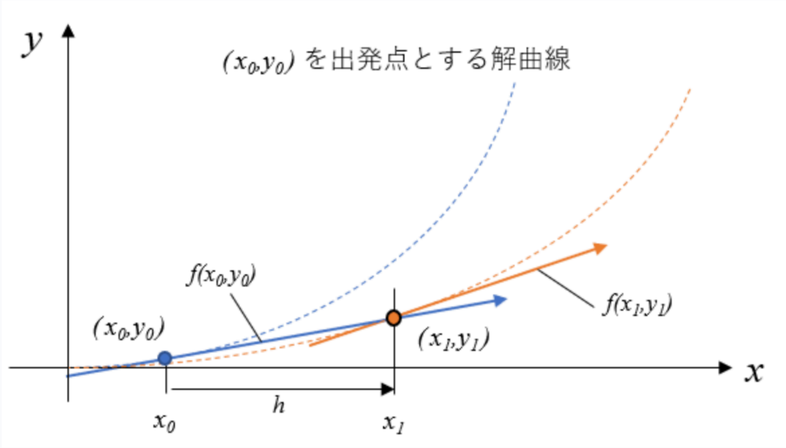
$$x(t_n) = x_0 + \int_a^{t_n} f(s, x(s))ds$$

オイラー法と修正オイラー法

- オイラー(Euler)法
- xの初期値X_0とする

$$t_{n+1} = t_n + h$$

$$X_{n+1} = X_n + hf(t_n, X_n)$$



- 修正オイラー法

$$X_{n+1} = X_n + hf(t_n + \frac{h}{2}, X_n + \frac{h}{2} f(t_n, X_n))$$

https://automatic-browsing.com/1階常微分方程式の数値解-オイラー法、ホイン法

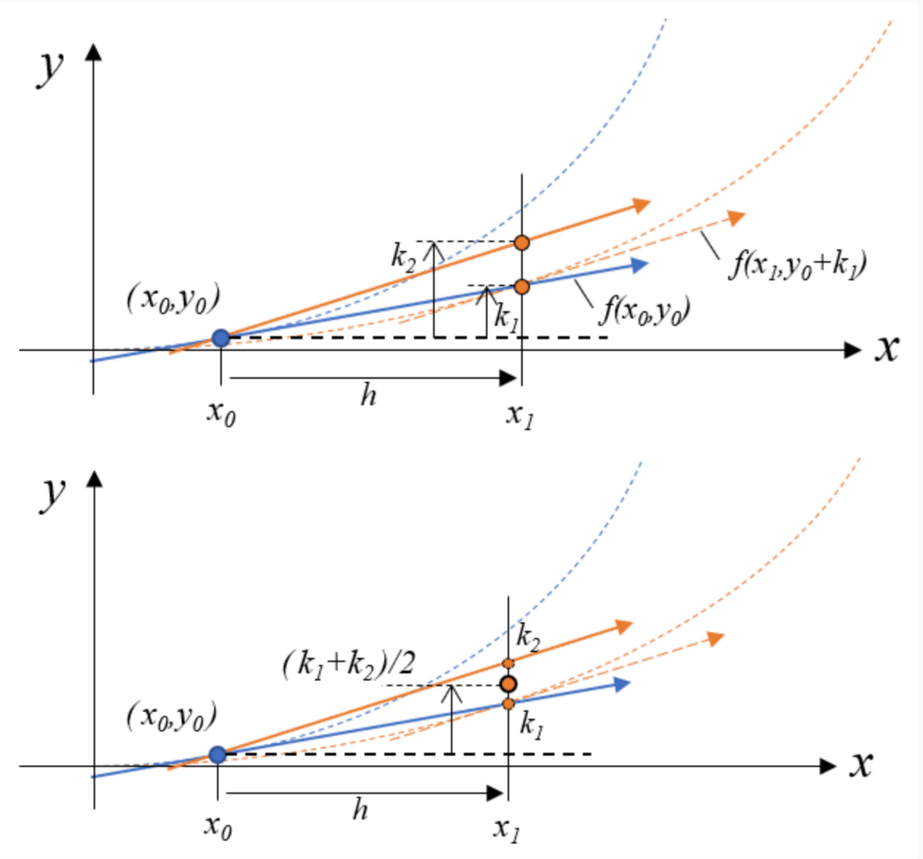
ホイン法

- 2次の公式

$$K_1 = f(t_n, X_n)$$

$$K_2 = f(t_n + h, X_n + hK_1)$$

$$X_{n+1} = X_n + \frac{h}{2}(K_1 + K_2)$$



https://automatic-browsing.com/1階常微分方程式の数値解-オイラー法、ホイン法

リチャードソン補外の適用

- リチャードソン補外
  - ある数値近似A(h)が真の値Aに対して次のような誤差を持つと仮定

    $$A(h) = A + Ch^p + O(h^p)$$
  - ここで、
    - h: ステップ幅（刻み幅）
    - C: 誤差項の係数
    - p: 数値解法の精度の次数（たとえば、オイラー法では1次）
    - $O(h^p)$: より高次の誤差項
  - このとき、2つの異なるステップ幅 h および h/2 で得られた解を用いて、高次精度の近似値を次のように得られる

    $$A_{extrap} = \frac{2^p A(h/2) - A(h)}{2^p - 1}$$
  - これにより、Chpを消去し、より高精度な近似が得られる

リチャードソン補外の特徴（ChatGPT-4o）

- 利点
  - 精度向上：低次数の手法でも高精度化できる
  - モジュール化：元の数値解法を変更せずに補外できる
  - 複雑な解析問題への応用が可能（例：物理シミュレーション）
- 欠点・注意点
  - 計算コスト：複数のステップを必要とするため、計算回数が増える
  - 数値的不安定性：高次補外を繰り返すと丸め誤差の影響が出やすい
  - 剛性（解が急激に変化する）問題には向かないことがある

ルンゲ・クッタ法

- ４次の公式

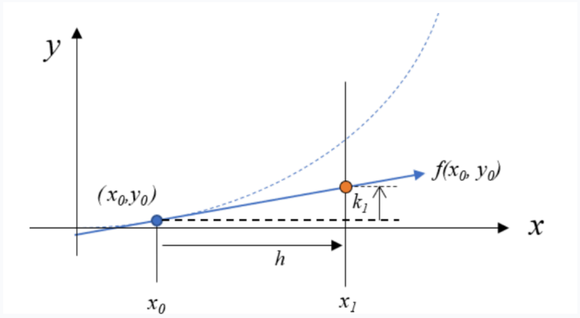
$$K_1 = f(t_n, X_n)$$

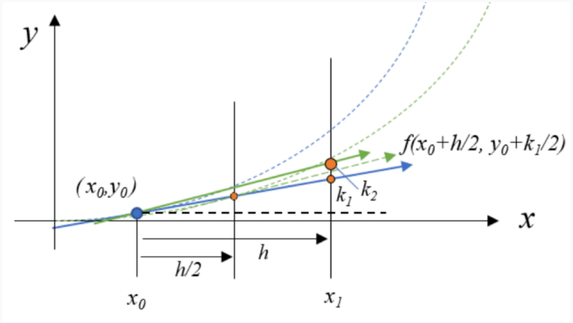
$$K_2 = f(t_n + \frac{h}{2}, X_n + \frac{h}{2}K_1)$$

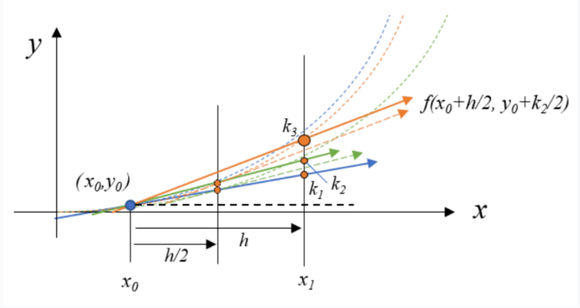
$$K_3 = f(t_n + \frac{h}{2}, X_n + \frac{h}{2}K_2)$$

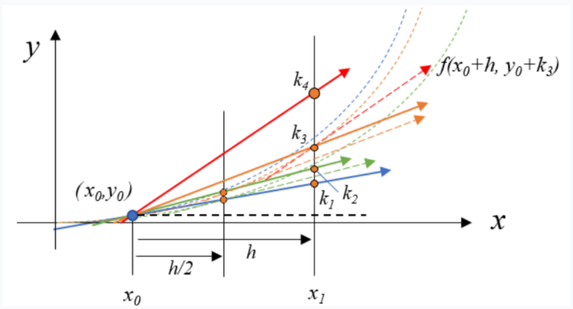
$$K_4 = f(t_n + h, X_n + hK_3)$$

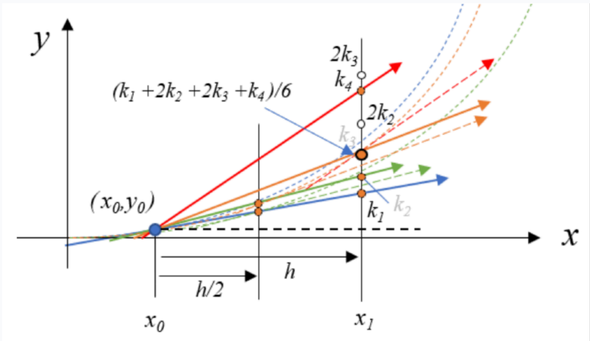
$$X_{n+1} = X_n + \frac{h}{6}(K_1 + 2K_2 + 2K_3 + K_4)$$











https://automatic-browsing.com/1階常微分方程式の数値解-オイラー法、ホイン法

ルンゲ=クッタ法とリチャードソン補外（ChatGPT-4o）

- Runge-Kutta法にリチャードソン補外を組み合わせることで、複数のステップ結果を使って局所誤差を制御する「多段階補外法（extrapolation methods）」が構成される。剛性のない（解が急激に変化することがない）問題で非常に効率的に機能する。
- 例：
  - Gragg-Bulirsch-Stoer法（GBS法）

    高精度なODE解法で、リチャードソン補外を中心に設計されており、計算コストを最小限にしつつ高精度を実現。

    - ステップ1: 修正中点法（modified midpoint method） でいくつかの近似解を得る
    - ステップ2: リチャードソン補外 によって高精度化する

暗黙法（ChatGPT-4o）

- 剛性のある（解が急激に変化する場所がある）問題では、暗黙法（Implicit method）を使う必要がある。
  - 暗黙解法は、常微分方程式（ODE）や偏微分方程式（PDE）などの数値的な時間積分において、現在の値だけでなく、未来の値も含めた式を解く手法
  - 未来の値y_{n+1}を 未知の状態で定義し、方程式を解く必要がある：
  - $y_{n+1} = y_n + hf(t_{n+1}, y_{n+1})$
  - これは後退オイラー法（Backward Euler）の例で、一般には非線形方程式を解く必要がある

暗黙法の使用例（ChatGPT-4o）

- 暗黙法は以下の特徴がある
  - 方程式を反復的に解く…Newton法や固定点反復法などが必要
  - 安定性が非常に高い…剛性問題でも大きなステップ幅で安定動作
  - 計算コストが高い…非線形（または線形）方程式を毎ステップ解くため計算量が増える
  - 精度の制御が容易…時間ステップの制御や高次精度拡張がしやすい
- また暗黙解法の種類と使用例は以下のようなものがある

名称	数式	特徴	備考
後退オイラー法	$y_{n+1} = y_n + hf(t_{n+1}, y_{n+1})$	一番基本的な暗黙法	1次精度、A-安定
暗黙中点法	$y_{n+1} = y_n + hf\left(t_n + \frac{h}{2}, \frac{y_n + y_{n+1}}{2}\right)$	対称性と精度に優れる	2次精度
Trapezoidal法（台形公式）	$y_{n+1} = y_n + \frac{h}{2}[f(t_n, y_n) + f(t_{n+1}, y_{n+1})]$	高精度な汎用暗黙法	2次精度、A-安定
BDF法（Backward Differentiation Formula）	多段階の暗黙法	剛性問題に強い	SciPyのBDFオプションで利用可能
• Radau IIA法	暗黙Runge-Kutta法の一種	高精度かつ剛性に非常に強い	SciPyのRadauメソッドで実装済み

scipyのODE解法アルゴリズム一覧（ChatGPT-4o）

- SciPyのsolve_ivp関数では、以下のメソッドを使用できる：

メソッド	アルゴリズム	特徴	向いている問題
RK45	Dormand-Prince (5(4)次)	明示的Runge-Kutta法。デフォルト。精度と速度のバランスが良い。	非剛性 (non-stiff) 問題
RK23	Bogacki–Shampine (3(2)次)	低精度ながら安定。初期の挙動の把握に適す。	非剛性で粗い精度でも良い場合
Radau	Radau IIA 法（Implicit Runge-Kutta）	高精度で剛性に強い。重い計算負荷。	剛性（stiff）問題
BDF	後退差分法（Backward Differentiation Formula）	多段法で剛性に強い。大規模系に強い。	剛性問題、大規模モデル
LSODA	自動切替型（Adams/BDF）	剛性・非剛性を自動判定。便利だがブラックボックス。	剛性かどうかわからない場合
DOP853	Dormand-Prince (8次)	高次のRunge-Kutta法。精度優先で遅い。	高精度が必要な非剛性問題

scipyのode関数LSODA

- 内部的には、Livermore Solver for Ordinary Differential Equations（LSODE）の拡張版LSODAを用いている。
- Alan C. HindmarshとAndrew H. Shermanが、Fortran言語用に書いたLSODEを踏襲（詳しい解説は、https://computing.llnl.gov/sites/default/files/ODEPACK_pub2_u113855.pdf）
- 通常の常微分方程式（ODE）系を数値的に解くための剛性・非剛性対応の基盤ソルバー。
- ソルバーとしては、以下の2つの手法を持つ：
  - Adams法（非剛性ODEに向く）
  - BDF法（剛性ODEに向く）
- LSODA = LSODE + 自動判定（Automatic switching）
  - LSODEの2つの手法（Adams法/BDF法）を内蔵しており、ODEの「剛性の有無」を内部で自動検出し、最適なアルゴリズムを自動で切り替える。

ScipyのODE解法アルゴリズムの比較（ChatGPT-4o）

- 1. 剛性（Stiffness）
  - 剛性とは、あるODE系に対し、解が急激に変化する成分と緩やかな成分を併せ持つこと。
  - RK45 や DOP853 は剛性に弱く、Radau や BDF は剛性に強い。
  - LSODA は自動で剛性の有無を判断し、アルゴリズムを切り替える点で優れている。
- 2. 精度
  - 高精度が必要な場合は DOP853 や Radau。
  - 精度よりも速度を優先するなら RK23 や RK45。
- 3. 速度
  - 明示的手法（RK45, RK23, DOP853）は非剛性に対して高速。
  - 暗黙的手法（Radau, BDF）は1ステップの計算が重いが、剛性が強いほど結果的に高速になる。
- 4. 問題サイズ
  - 多変数かつ剛性がある問題には BDF が向いている。
  - 小規模で試験的なシミュレーションなら RK45 で十分。

ScipyでのODE解法関数

- odeint関数
  - scipy.integrate.odeint(func, x0, t)という形で呼び出す
  - tはリストで、[tの初期値, …, tの最終値]という形にする
  - x0はリストで、[最初の値, 最初の微分値, …]という形にする
  - funcは、リストで、[微分形, ２階微分形, …]という形にする
- solve_ivp関数
  - scipy.integrate.solve_ivp(func, t, x0, method=アルゴリズム名)
  - methodとしては、"RK45", "RK23", "Radau", "BDF", "LSODA", "DOP853"が選択できる

空気抵抗のある弾道の計算

- ρは空気密度、Sは前面の面積、Cdは抗力係数

$$\frac{dy}{dt} = v_y$$

$$\frac{dv_y}{dt} = -\frac{g}{m} - kv_y^2$$

$$k = \frac{1}{2}\rho S C_d$$

$$\frac{dx}{dt} = v_x$$

$$\frac{dv_x}{dt} = -kv_x^2$$