変奏

モチーフと変奏

モチーフ(動機)は、人間の聴覚的チャンク(3秒)程度の長さをもつ旋律の最小の意味単位である。モチーフはしばし反復され、さらに「変奏（variation）」を通じて変化する。

旋律を生成する際の出発点となるモチーフのことを「原型（original）」という。原型のパラメータ列(リスト)を何らかの数値的操作によって変形することで、変奏をつくることができる。原型が持っている音響的、音楽的な喚起力を変奏を通じて展開・発展することで、反復や緊張、解決(弛緩)という機能を与え、作品の部分間の関連性や全体的な統一感をコントロールすることができる。

変奏の基本形

変奏を生み出すためのメソッドは、大きく以下4種類に分けられる。

値(value)による変奏 : パラメータ列の要素の数や順序は入れ替えず、それぞれの要素の値を変形するもの

順序(order)による変奏 : パラメータ列の要素の数や値は変化させず、要素の順序を入れ替えるもの

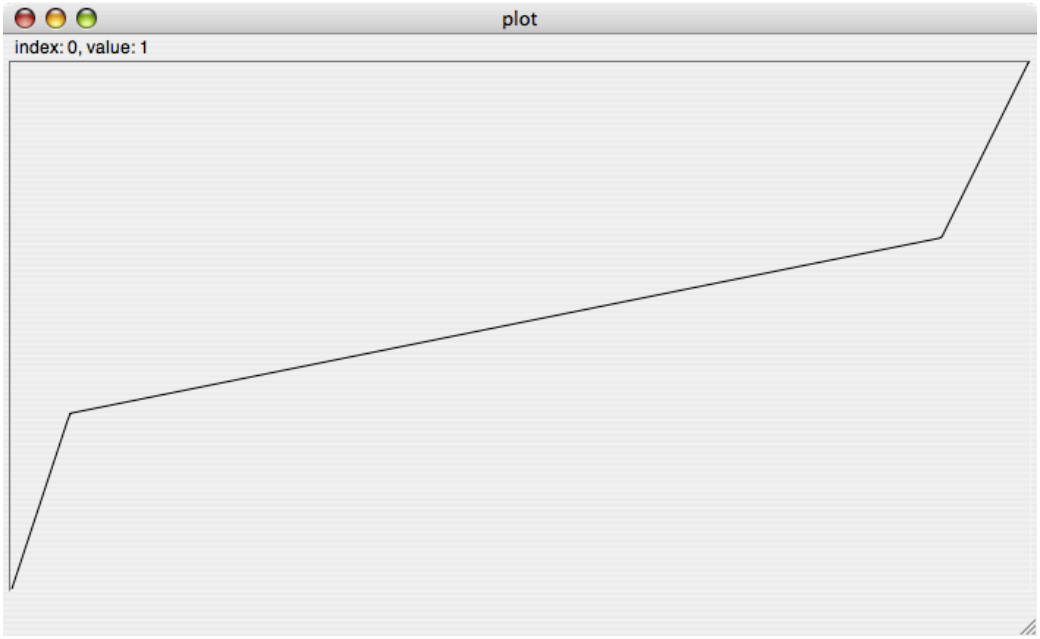
要素(item)による変奏 : パラメータ列に新たな要素を追加したり、すでにある要素を削除するもの

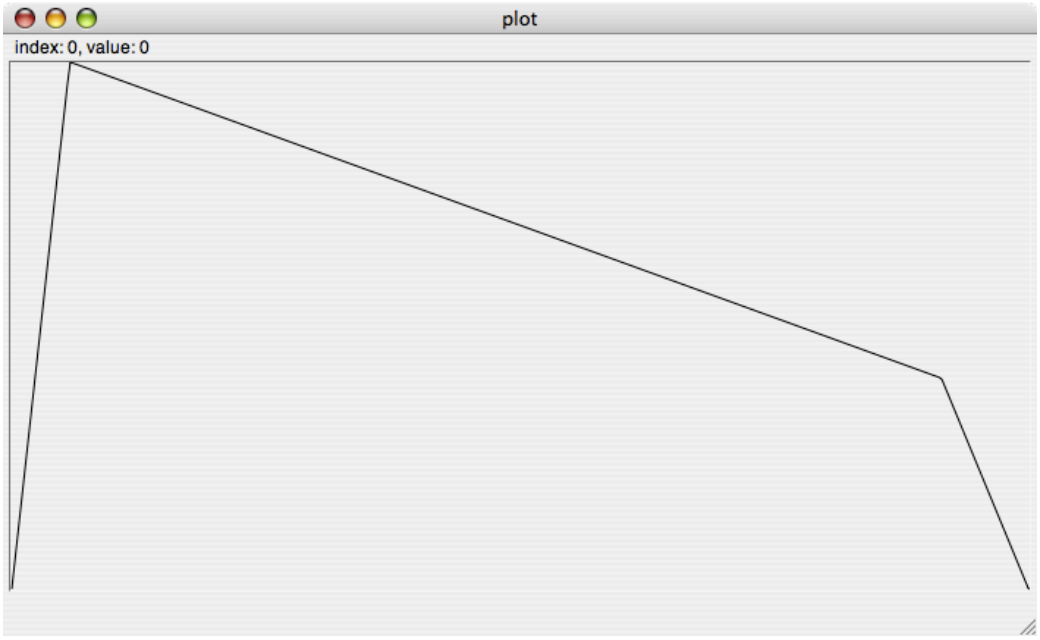
リスト(list)による変奏 : 2つのパラメータ列から、新たなパラメータ列を生成するもの

1本の線による旋律の原型を考える。

```
// 原型(リスト)
[1, 3, 2, 4].postln;
```

```
// 原型(エンヴェロープ)
(
~out = { var f0, a0, t0, envf, enva;
    f0 = [1, 2, 3, 4]; // 周波数のパラメータ列
    a0 = [0, 0.5, 0.2, 0]; // 振幅のパラメータ列
    t0 = [0.2, 3, 0.3]; // 継続時間のパラメータ列
    envf = Env.new(f0, t0).plot; // 周波数エンヴェロープ
    enva = Env.new(a0, t0).plot; // 振幅エンヴェロープ
    SinOsc.ar(100 * EnvGen.kr(envf, doneAction:2), 0, EnvGen.kr(enva, doneAction:2)).dup }
)
```





値による変奏

パラメータ列の値全体に対する操作の基本は、ある定数(係数)の加減算と乗除算である。原型としてのパラメータ列をp0、aを任意の定数とすると、これらのメソッドは以下のような式で書ける。

加減算：p0 + a

乗除算：p0 * a

これらはそれぞれ、

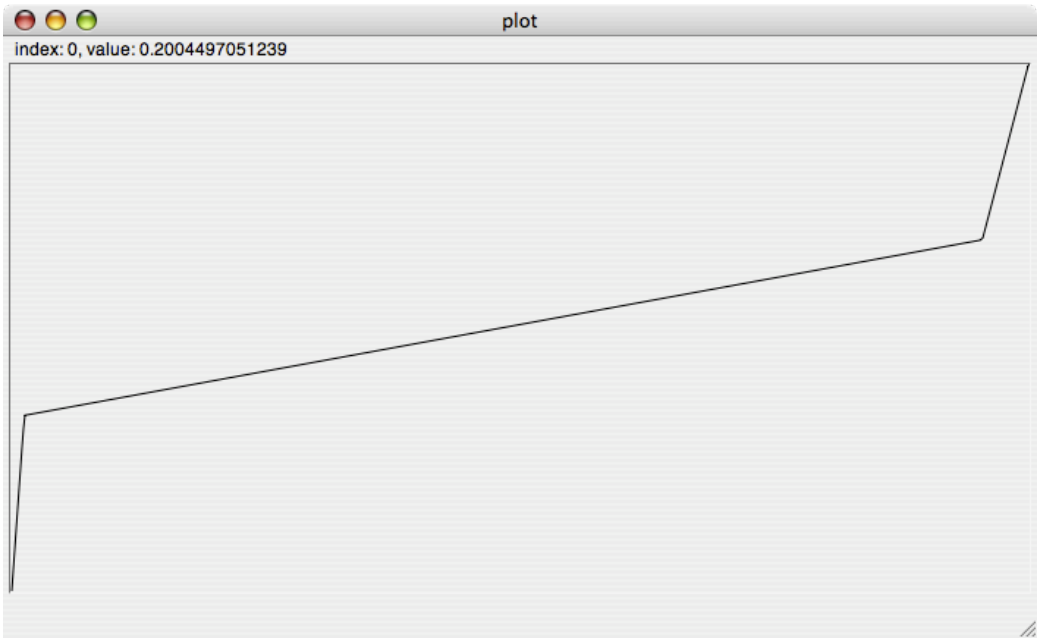
移高（transposition）：パラメータ列を平行移動する(加減算)。

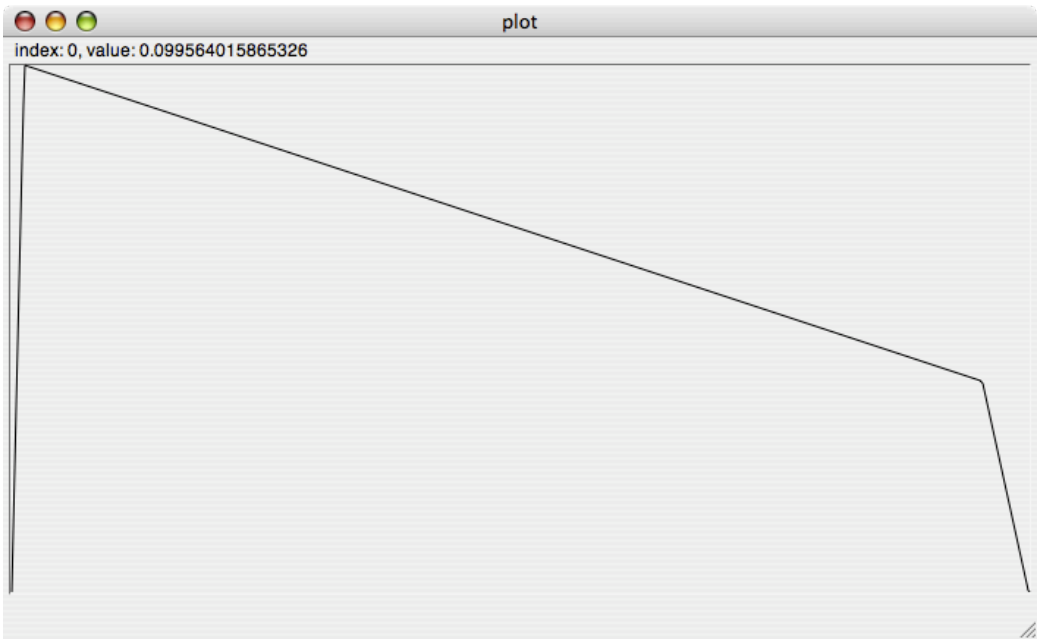
拡大（expansion）・**縮小**（compression）：パラメータ列のスケールを変化させる(乗除算)。

と呼ばれている。

```
// 移高(リスト)
([1, 3, 2, 4] + 3).postln;
```

```
[ 4, 6, 5, 7 ]

// 移高(エンヴェロープ)
(
~out = { var f0, a0, t0, envf, enva;
    f0 = [1, 2, 3, 4] + 1.0.rand2; // 周波数のパラメータ列を移高
    a0 = [0, 0.5, 0.2, 0] + 0.5.rand; // 振幅のパラメータ列を移高
    t0 = [0.2, 3, 0.3] + 0.2.rand2; // 継続時間のパラメータ列を移高
    envf = Env.new(f0, t0).plot; // 周波数エンヴェロープ
    enva = Env.new(a0, t0).plot; // 振幅エンヴェロープ
    SinOsc.ar(100 * EnvGen.kr(envf, doneAction:2), 0, EnvGen.kr(enva, doneAction:2)).dup }
)
```






```
// 拡大(リスト)
([1, 3, 2, 4] * 3).postln;

[ 3, 9, 6, 12 ]
```

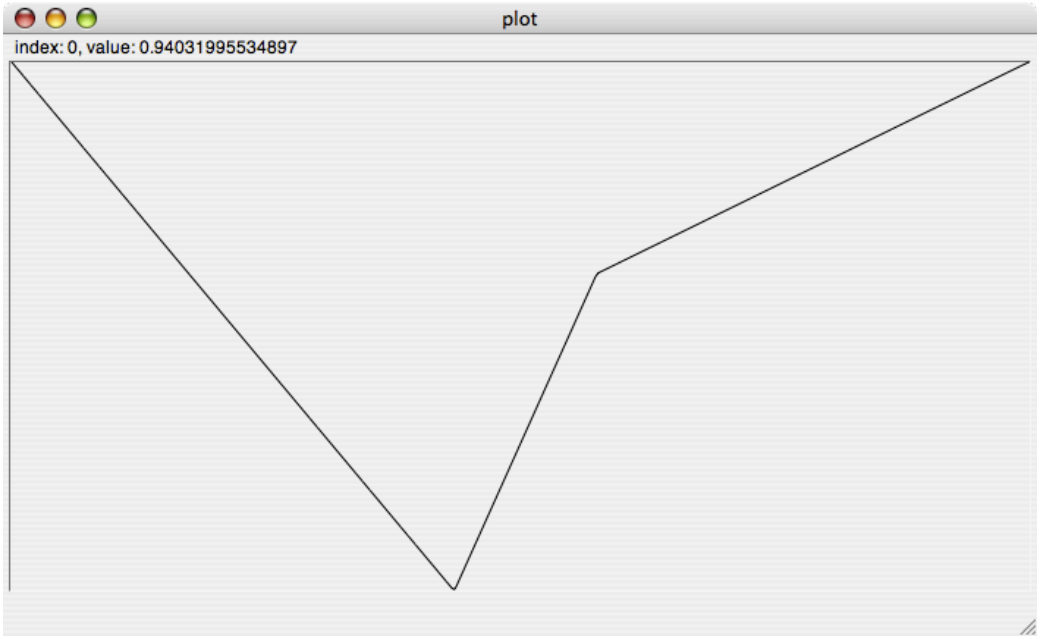
```
// 縮小(リスト)
([1, 3, 2, 4] * 0.5).postln;

[ 0.5, 1.5, 1, 2 ]


// 拡大・縮小(エンヴェロープ)
(
~out = { var f0, a0, t0, envf, enva;
    f0 = [1, 2, 3, 4] * 2.0.rand; // 周波数のパラメータ列を拡大・縮小
    a0 = [0, 0.5, 0.2, 0] * 2.0.rand; // 振幅のパラメータ列を拡大・縮小
    t0 = [0.2, 3, 0.3] * 2.0.rand; // 継続時間のパラメータ列を拡大・縮小
    envf = Env.new(f0, t0).plot; // 周波数エンヴェロープ
    enva = Env.new(a0, t0).plot; // 振幅エンヴェロープ
    SinOsc.ar(100 * EnvGen.kr(envf, doneAction:2), 0, EnvGen.kr(enva, doneAction:2)).dup }
)
```

拡大・縮小の係数aが負になると、値の並びの大小が反転される。係数aが-1の場合は「反行」と呼ばれる。パラメータの値を正の数にするために、しばしある定数を加えるため、反行は拡大・縮小と移高の組み合わせである。

反行（inversion）：パラメータ列の上下(大小)関係を反転させる。

反行：-p0 + a

```
// 反行(リスト)
(7 - [1, 3, 2, 4]).postln;

[ 6, 4, 5, 3 ]


// 反行(エンヴェロープ)
(
~out = { var f0, a0, t0, envf, enva;
    f0 = rrand(4.0, 10.0) - [1, 2, 3, 4]; // 周波数のパラメータ列を反行
    a0 = rrand(0.5, 1.0) - [0, 0.5, 0.2, 0]; // 振幅のパラメータ列を反行
    t0 = rrand(3.0, 6.0) - [0.2, 3, 0.3]; // 継続時間のパラメータ列を反行
    envf = Env.new(f0, t0).plot; // 周波数エンヴェロープ
    enva = Env.new(a0, t0).plot; // 振幅エンヴェロープ
    SinOsc.ar(100 * EnvGen.kr(envf, doneAction:2), 0, EnvGen.kr(enva, doneAction:2)).dup }
)
```



順序による変奏

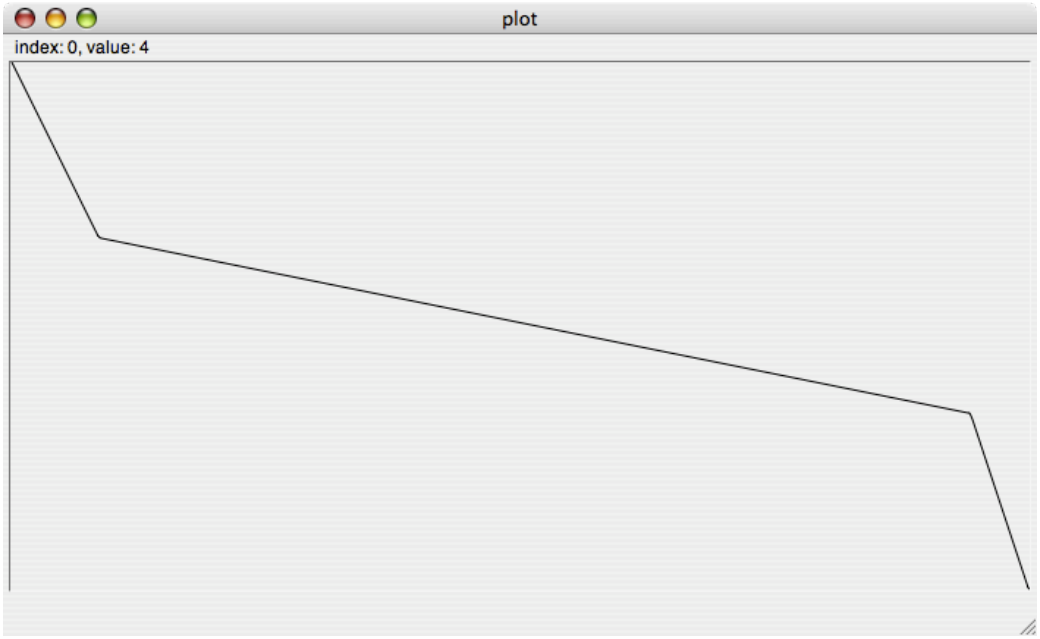
パラメータ列の並びを逆にすることを

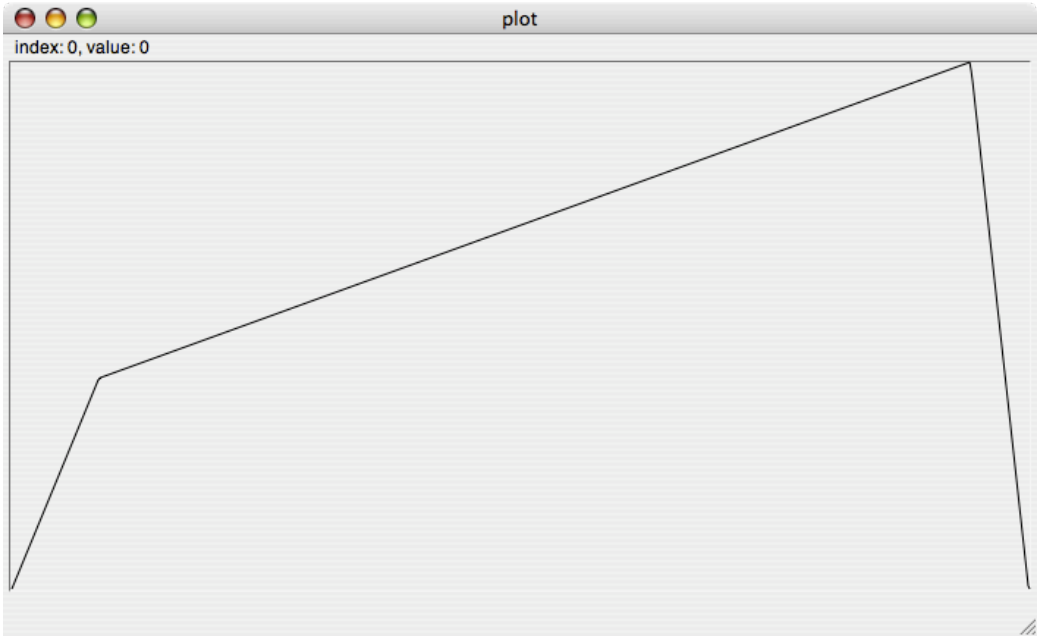
逆行（retrograde）：パラメータ列の時間的な並びを反転させる。

という。逆行は以下のように書くことができる。

逆行：p0.reverse（reverseは配列の要素を逆順にするメソッド）

```
// 逆行(リスト)
([1, 3, 2, 4].reverse).postln;

[ 4, 2, 3, 1 ]

// 逆行(エンヴェロープ)
(
~out = { var f0, a0, t0, envf, enva;
    f0 = [1, 2, 3, 4].reverse; // 周波数のパラメータ列を逆行
    a0 = [0, 0.5, 0.2, 0].reverse; // 振幅のパラメータ列を逆行
    t0 = [0.2, 3, 0.3].reverse; // 継続時間のパラメータ列を逆行
    envf = Env.new(f0, t0).plot; // 周波数エンヴェロープ
    enva = Env.new(a0, t0).plot; // 振幅エンヴェロープ
    SinOsc.ar(100 * EnvGen.kr(envf, doneAction:2), 0, EnvGen.kr(enva, doneAction:2)).dup }
)
```





なお、原型のパラメータ列に逆行を接続(追加)することを鏡像という。

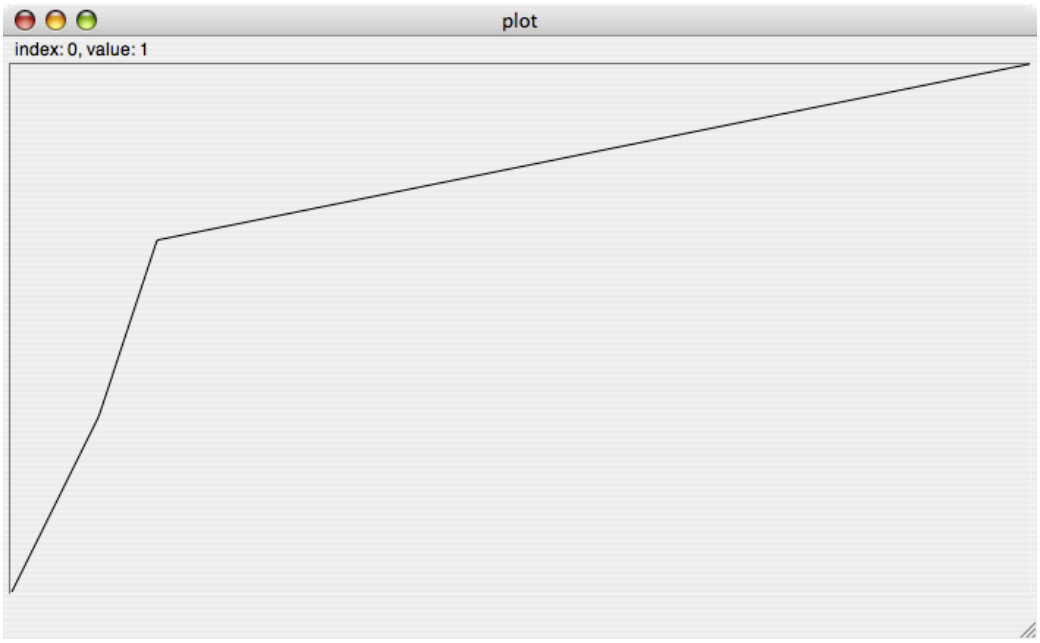
```
// 鏡像(原型+逆行)

([1, 3, 2, 4].mirror).postln; // 原型の最後の要素を重複させない

[ 1, 3, 2, 4, 2, 3, 1 ]

([1, 3, 2, 4].mirror2).postln; // 原型の最後の要素を重複させる

[ 1, 3, 2, 4, 4, 2, 3, 1 ]
```

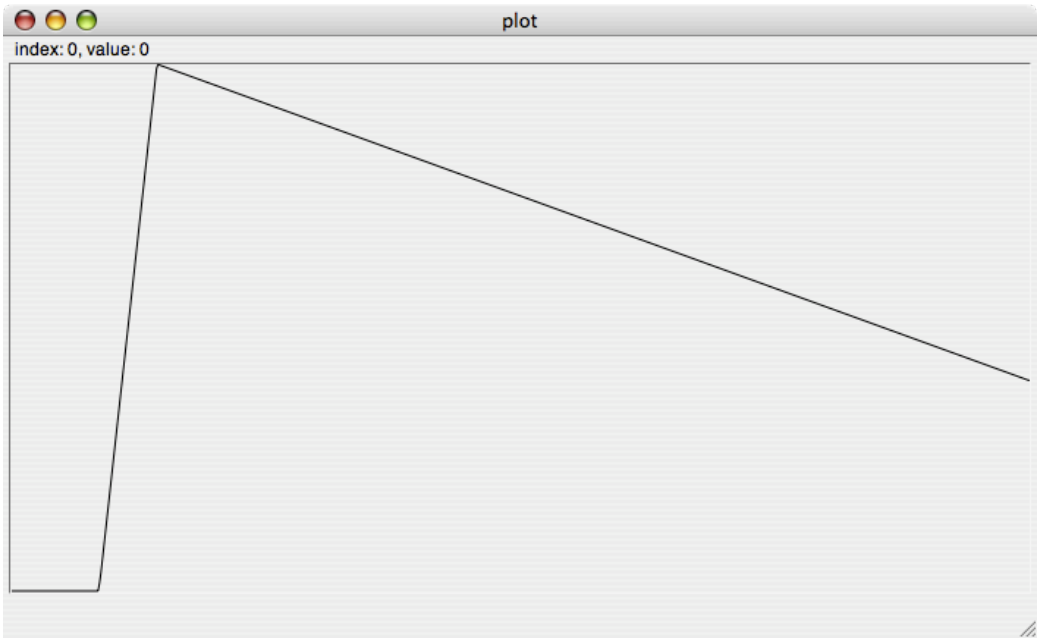
パラメータ列の並びをシフト(ずらす)のが「回転」である。シフト量をn(右シフトが+、左シフトが-)とすると、回転は以下のように書くことができる。

回転：p0.rotate(n)

```
// 回転(リスト)
([1, 3, 2, 4].rotate(1)).postln;
```

```
[ 4, 1, 3, 2 ]

([1, 3, 2, 4].rotate(-2)).postln;

[ 2, 4, 1, 3 ]


// 回転(エンヴェロープ)
(
~out = { var f0, a0, t0, envf, enva;
    f0 = [1, 2, 3, 4].rotate(4.rand); // 周波数のパラメータ列を回転
    a0 = [0, 0.5, 0.2, 0].rotate(4.rand); // 振幅のパラメータ列を回転
    t0 = [0.2, 3, 0.3].rotate(3.rand); // 継続時間のパラメータ列を回転
    envf = Env.new(f0, t0).plot; // 周波数エンヴェロープ
    enva = Env.new(a0, t0).plot; // 振幅エンヴェロープ
    SinOsc.ar(100 * EnvGen.kr(envf, doneAction:2), 0, EnvGen.kr(enva, doneAction:2)).dup }
)
```





パラメータ列の要素の順番を入れ替えることを「交換」という。i番目の要素とj番目の要素を入れ替える交換を以下のように書くことができる。

交換：p0.swap(i,j)

```
// 交換(リスト)
([1, 3, 2, 4].swap(1, 3)).postln;
```

```
[ 1, 4, 2, 3 ]
```

```
// 交換(エンヴェロープ)
(
~out = { var f0, a0, t0, envf, enva;
    f0 = [1, 2, 3, 4].swap(4.rand, 4.rand); // 周波数のパラメータ列をランダムに交換
    a0 = [0, 0.5, 0.2, 0].swap(4.rand, 4.rand); // 振幅のパラメータ列をランダムに交換
    t0 = [0.2, 3, 0.3].swap(3.rand, 3.rand); // 継続時間のパラメータ列をランダムに交換
    envf = Env.new(f0, t0).plot; // 周波数エンヴェロープ
    enva = Env.new(a0, t0).plot; // 振幅エンヴェロープ
    SinOsc.ar(100 * EnvGen.kr(envf, doneAction:2), 0, EnvGen.kr(enva, doneAction:2)).dup }
)
```

要素の順序のランダムな並べ替えを「スクランブル」という。

スクランブル：p0.scramble

```
// スクランブル
([1, 3, 2, 4].scramble).postln;
```

```
[ 2, 4, 1, 3 ]
```

```
// ランダムな交換と比較する
[1, 2, 3, 4].swap(4.rand, 4.rand)
```

```
[ 3, 2, 1, 4 ]
```

```
// ランダムな交換を繰り返す
10.do{[1, 2, 3, 4].swap(4.rand, 4.rand).postln}
```

```
[ 2, 1, 3, 4 ]
[ 1, 3, 2, 4 ]
[ 1, 4, 3, 2 ]
[ 1, 3, 2, 4 ]
[ 1, 2, 3, 4 ]
[ 1, 4, 3, 2 ]
[ 4, 2, 3, 1 ]
[ 1, 2, 3, 4 ]
[ 3, 2, 1, 4 ]
[ 1, 4, 3, 2 ]
```

要素による変奏

パラメータ列の要素そのものに対する操作の基本は「挿入」と「削除」である。

```
// 挿入
([1, 3, 2, 4].insert(1, 5)).postln;
```

```
[ 1, 5, 3, 2, 4 ]
```

```
([1, 3, 2, 4].add(5)).postln;
```

```
[ 1, 3, 2, 4, 5 ]
```

```
// 挿入の繰り返しによる反復
(
x = [1, 3, 2, 4];
10.do{x = x.insert(1, 5)};
x.postln;
)
```

```
[ 1, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 3, 2, 4 ]
```

```
// コピーと挿入による部分的な反復
(
var x = [1, 3, 2, 4];
(x.insert(1, x.copyRange(1, 3))).flatten.postln;
)

[ 1, 3, 2, 4, 3, 2, 4 ]


// 削除
(
x = [1, 3, 2, 4];
x.removeAt(1);
x.postln;
)

[ 1, 2, 4 ]

([1, 3, 2, 4].removeAt(1)).postln;
```

挿入と削除を組み合わせることで、要素の「置換」を行うことができる。

```
// 置換(挿入+削除)
(
x = [1, 3, 2, 4].insert(1, 5); // 挿入
x.removeAt(2); // 削除
x.postln;
)

[ 1, 5, 2, 4 ]

// これをひとつのメソッドで書くこともできる
([1, 3, 2, 4].put(1, 5)).postln;

[ 1, 5, 2, 4 ]
```

リストによる変奏

2つ(以上)のパラメータ列(リスト)による変奏も可能である。

```
// 2つのリストの演算(対応する要素同士)
(
var x = [1, 2, 3, 4]; // リスト1
var y = [5, 6, 7, 8]; // リスト2
(x + y).postln; // 2つのリストの加算
(x - y).postln; // 2つのリストの減算
(x * y).postln; // 2つのリストの乗算
(x / y).postln; // 2つのリストの除算
)

[ 6, 8, 10, 12 ]
[ -4, -4, -4, -4 ]
[ 5, 12, 21, 32 ]
[ 0.2, 0.33333333333333, 0.42857142857143, 0.5 ]


// 2つのリストの演算(すべての要素の組み合わせ)
(
var x = [1, 2, 3, 4]; // リスト1
var y = [5, 6, 7, 8]; // リスト2
(x +.x y).postln; // 2つのリストの加算
(x -.x y).postln; // 2つのリストの減算
(x *.x y).postln; // 2つのリストの乗算
(x /.x y).postln; // 2つのリストの除算
)

[ 6, 7, 8, 9, 7, 8, 9, 10, 8, 9, 10, 11, 9, 10, 11, 12 ]
```

```
[ -4, -5, -6, -7, -3, -4, -5, -6, -2, -3, -4, -5, -1, -2, -3, -4 ]
[ 5, 6, 7, 8, 10, 12, 14, 16, 15, 18, 21, 24, 20, 24, 28, 32 ]
[ 0.2, 0.16666666666667, 0.14285714285714, 0.125, 0.4, 0.33333333333333, 0.28571428571429, 0.25, 0.6, 0.5,
0.42857142857143, 0.375, 0.8, 0.66666666666667, 0.57142857142857, 0.5 ]


// 2つのリストの交配(離散的な混合)
(
var x = [1, 2, 3, 4]; // リスト1
var y = [5, 6, 7, 8]; // リスト2
var i = 4.rand; // インデックス
(x.put(i, y.at(i))).postln;
)

[ 1, 6, 3, 4 ]

// 2つのリストのブレンド(連続的な混合)
(
var x = [1, 2, 3, 4]; // リスト1
var y = [5, 6, 7, 8]; // リスト2
var a = 1.0.rand; // インデックス
((a * x) + ((1 - a) * y)).postln;
)

[ 2.3667421340942, 3.3667421340942, 4.3667421340942, 5.3667421340942 ]
```

エンヴェロープ自体を直接混合する方法もある。

```
// 2つのエンヴェロープのブレンド(連続的な混合)
(
var a = Env([1, 2, 3, 4], [1, 1, 1]); // 原型
var b = Env([1, 2, 3, 4].scramble, [1, 1, 1]); // スクランブルによる変奏
blend(a, b, 0.0).plot; // 原型 : 変奏 = 1 : 0
blend(a, b, 0.25).plot; // 原型 : 変奏 = 0.75 : 0.25
blend(a, b, 0.5).plot; // 原型 : 変奏 = 0.5 : 0.5
blend(a, b, 0.75).plot; // 原型 : 変奏 = 0.25 : 0.75
blend(a, b, 1.0).plot; // 原型 : 変奏 = 0 : 1
)
```

変奏の意味

こうしたさまざまなメソッドを用いて、アルゴリズミックに変奏をつくりだすことはたやすい。しかし個々の要素としての変奏に反復、緊張、解決などの機能を与え、それらを有機的に組み合わせ統一感を生み出したり、変奏を通じて原型の音響的換気力を全体に波及・浸透させることは非常に難しい。原型の旋律を変形しすぎて、聴き手が原型と変奏の関係を見失なうようでは、せっかくの変奏も変奏でなくなってしまう。

変奏を生みだす数値的なメソッドそのものが重要なのではない。変奏が変奏であるためには、そのフレーズが原型の変奏であると認知できなければならない。したがって変奏は通常、原型となる旋律の輪郭(方向や形といった全体的な形態認知)を保持するように、そのあまり重要「でない」特徴に対して行われる。いいかえれば、いい変奏をデザインするためには、旋律の輪郭パターンの知覚や認知のされ方に対する意識と感覚が必要とされる。

変奏の聴覚的、音響的な有効性(あるいは無効性)を判断するのは、やはり耳とその使い方(聴取)である。耳を有効に使うためには、原型の輪郭とそこから数値的に生みだされた変奏の関係を、パラメトリック・リスニング、ランダム・リスニング、インタラクティヴ・リスニングなどの方法を用いて、注意深く聴き込む必要がある。

旋律とゲシュタルト

旋律の輪郭は、時間的に変動するスペクトルとしての音響を、耳がグルーピング(群化)して認知し、全体的なパターンとして抽象することにより生まれる。それは知覚を既定する全体的な枠組み(ゲシュタルト、形態)と深く関連している。聴覚的なゲシュタルトとしては、以下のようなものがある。

・近接(proximity)

離れているもnoよりも、近接しているもの同士の方が、ひとまとまりになりやすい。

```
100.do{ if (0.5.coin, {"|".post}, {" ".post})};

|     || ||  | ||| ||||||||| || |||||    |||  |||  |  ||| | |     |||| ||  | ||| ||   || | || |   |
```

周波数の近接によるグルーピング(と分裂)

```
// 基本周波数882[Hz]から周波数差44.1[Hz](5%)で毎秒10回高さが変動する線(継続時間10秒)
(
~out = {
    var env = Env.new(Array.fill(101, {|i| i % 2}), Array.fill(100, 0.1), \step).plot;
    SinOsc.ar(882 * EnvGen.kr(env, levelScale:0.05, levelBias:1.0, doneAction:2), 0, 0.2).dup}
)

// 基本周波数882[Hz]から周波数差441[Hz](50%)で毎秒10回高さが変動する線(継続時間10秒)
(
~out = {
    var env = Env.new(Array.fill(101, {|i| i % 2}), Array.fill(100, 0.1), \step).plot;
    SinOsc.ar(882 * EnvGen.kr(env, levelScale:0.5, levelBias:1.0, doneAction:2), 0, 0.2).dup}
)
```

グルーピングに対する変動時間の影響

```
// 基本周波数882[Hz]から周波数差441[Hz](50%)で毎秒1回高さが変動する線(継続時間10秒)
(
~out = {
    var env = Env.new(Array.fill(11, {|i| i % 2}), Array.fill(10, 1), \step).plot;
    SinOsc.ar(882 * EnvGen.kr(env, levelScale:0.5, levelBias:1.0, doneAction:2), 0, 0.2).dup}
)

// 基本周波数882[Hz]から周波数差441[Hz](50%)で毎秒20回高さが変動する線(継続時間10秒)
(
~out = {
    var env = Env.new(Array.fill(201, {|i| i % 2}), Array.fill(200, 0.05), \step).plot;
    SinOsc.ar(882 * EnvGen.kr(env, levelScale:0.5, levelBias:1.0, doneAction:2), 0, 0.2).dup}
)

// 2つの異なる速度の変動を重ねる。
(
~out = {
    var env1 = Env.new(Array.fill(11, {|i| i % 2}), Array.series(10, 1, 0), \step).plot;
    var env2 = Env.new(Array.fill(201, {|i| i % 2}), Array.series(200, 0.05, 0), \step).plot;
    SinOsc.ar(882 * EnvGen.kr(env1, levelScale:0.5, levelBias:1.0, doneAction:2), 0, 0.2).dup
    + SinOsc.ar(882 * EnvGen.kr(env2, levelScale:0.5, levelBias:1.0, doneAction:2), 0, 0.2).dup}
)
```

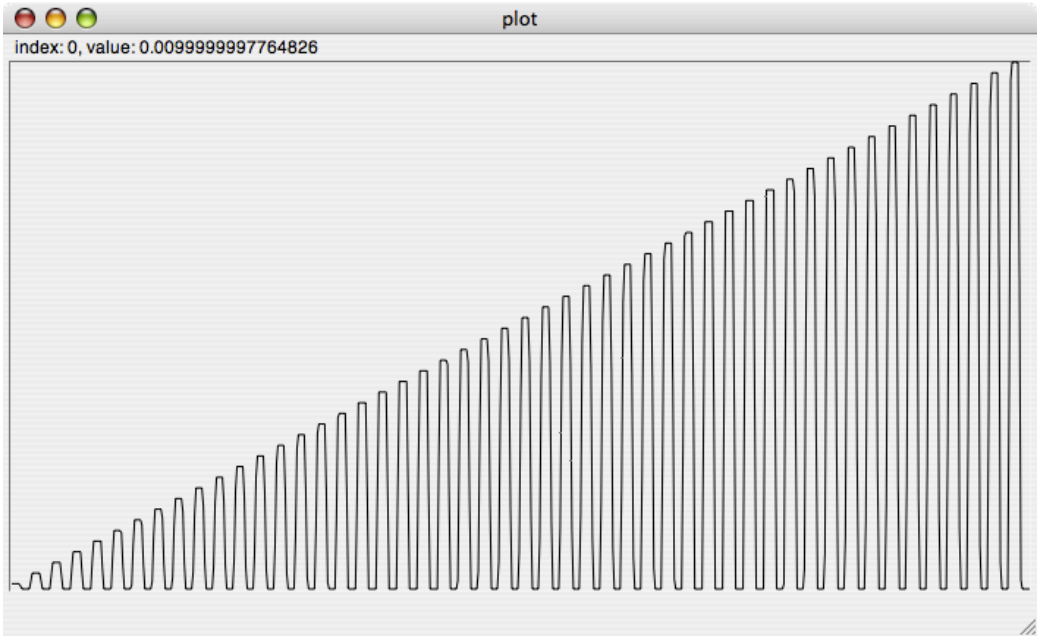
周波数と時間の変動間隔をインタラクティヴに変化させる。

```
// マウスのX軸を変動周波数の間隔、Y軸を変動周期に対応させる。
~out = { SinOsc.ar(LFPulse.kr(MouseY.kr(0.0, 40.0), 0, 0.5, MouseX.kr(0.0, 441.0), 441), 0, 0.3).dup};
```

・よい連続(good continuation)

単純な形としてよく連続しているものはひとつとして見られやすい。

```
// 基本周波数882[Hz]から周波数差が0〜882[Hz]まで変化量が次第に増加していく線(変動周期10[Hz]、継続時間10秒)
(
~out = {
    var env = Env.new(Array.fill(101, {|i| (i % 2) * i / 100}), Array.fill(100, 0.1), \step).plot;
    SinOsc.ar(882 * EnvGen.kr(env, levelScale:1.0, levelBias:1.0, doneAction:2), 0, 0.2).dup}
)
```




```
// 831[Hz], 741[Hz], 330[Hz]の3つの周波数をランダムに毎秒10回変動する線(継続時間20秒)
(
~out = {
    var env = Env.new(Array.fill(201, {[831, 741, 330].choose}), Array.fill(200, 0.1), \step).plot;
    SinOsc.ar(EnvGen.kr(env, doneAction:2), 0, 0.2).dup}
)
```


```
// 基本周波数882[Hz]から周波数差88.2[Hz](10%)で次第に変動周波数が低くなっていく線
(
~out = {
    var env = Env.new(Array.fill(101, {|i| i % 2}), Array.fill(100, {|i| 0.05 + ((i / 100)**10)}),
\step).plot;
    SinOsc.ar(882 * EnvGen.kr(env, levelScale:0.1, levelBias:1.0, doneAction:2), 0, 0.2).dup}
)
```

・**類同**(similarity)

幾つかの刺激がある時、同種のもの同士がひとまとまりになりやすい。

```
100.do{ if (0.5.coin, {"|".post}, {"-".post})};

-||--|--|-|--|||---||||--|||||-----|-|---|-||||-|-|--||||--|-----|-||--|-------|-|-|-|-||||-|-------
```

```
// 2つの異なる変動速度の点列と点列を重ねる。
(
~out = {
    var env1 = Env.new(Array.fill(11, {|i| i % 2}), Array.series(10, 1, 0), \step);
    var env2 = Env.new(Array.fill(201, {|i| i % 2}), Array.series(200, 0.05, 0), \step);
    Impulse.ar(882 * EnvGen.kr(env1, levelScale:0.5, levelBias:1.0, doneAction:2), 0, 0.2).dup
    + Impulse.ar(882 * EnvGen.kr(env2, levelScale:0.5, levelBias:1.0, doneAction:2), 0, 0.2).dup}
)


// 2つの異なる変動速度の点列と線を重ねる。
(
~out = {
    var env1 = Env.new(Array.fill(11, {|i| i % 2}), Array.series(10, 1, 0), \step);
    var env2 = Env.new(Array.fill(201, {|i| i % 2}), Array.series(200, 0.05, 0), \step);
    Impulse.ar(882 * EnvGen.kr(env1, levelScale:0.5, levelBias:1.0, doneAction:2), 0, 0.2).dup
    + SinOsc.ar(882 * EnvGen.kr(env2, levelScale:0.5, levelBias:1.0, doneAction:2), 0, 0.2).dup}
)

(
~out = {
    var env1 = Env.new(Array.fill(11, {|i| i % 2}), Array.series(10, 1, 0), \step);
    var env2 = Env.new(Array.fill(201, {|i| i % 2}), Array.series(200, 0.05, 0), \step);
    SinOsc.ar(882 * EnvGen.kr(env1, levelScale:0.5, levelBias:1.0, doneAction:2), 0, 0.2).dup
    + Impulse.ar(882 * EnvGen.kr(env2, levelScale:0.5, levelBias:1.0, doneAction:2), 0, 0.2).dup}
)


// ランダムに変動する点列と点列を重ねる。
(
~out = {
    var env1 = Env.new(Array.rand(101, 0.0, 1.0), Array.series(100, 0.1, 0), \step);
    var env2 = Env.new(Array.rand(201, 0.0, 1.0), Array.series(200, 0.05, 0), \step);
    Impulse.ar(441 * EnvGen.kr(env1, levelScale:0.5, levelBias:1.0, doneAction:2), 0, 0.2).dup
    + Impulse.ar(441 * EnvGen.kr(env2, levelScale:0.5, levelBias:1.0, doneAction:2), 0, 0.2).dup}
)



// ランダムに変動する点列と線を重ねる。
(
~out = {
    var env1 = Env.new(Array.rand(101, 0.0, 1.0), Array.series(100, 0.1, 0), \step);
    var env2 = Env.new(Array.rand(201, 0.0, 1.0), Array.series(200, 0.05, 0), \step);
    SinOsc.ar(441 * EnvGen.kr(env1, levelScale:0.5, levelBias:1.0, doneAction:2), 0, 0.2).dup
    + Impulse.ar(441 * EnvGen.kr(env2, levelScale:0.5, levelBias:1.0, doneAction:2), 0, 0.2).dup}
)
```

ゲシュタルト崩壊

ゲシュタルト崩壊とは、全体性を失って、個別のみを認識するようになることである。旋律の過剰な反復などによって、ゲシュタルトを崩壊させることで、旋律の異化効果を生みだすことができる。

```
// 同じパターンを長時間聴き続けるとどうなるか?
(
var flist = Array.geom(12, 1, 2**(1/12)).scramble;
~out = Pbind(
        \instrument, \line,
        \amp, 0.5,
        \freq, Pseq(flist * 100, inf),
        \sustain, 0.1,
        \dur, 0.1,
        \pan, 0.0
    );
```

)

パラメータの絶対性

こうしたゲシュタルトによる旋律の輪郭は、主に音の高さや長さ(間隔)の関係と深く関連している。例えば、古典的な旋律の輪郭は、移調によって変化しないとされているが、これは旋律の輪郭にとって、相対的な音の高さの関係が、音の絶対的な高さよりも重要であることを示している。同様に、旋律のテンポを変化させても旋律の同一性を認識できることは、相対的な音の長さの関係が、音の絶対的な長さよりも重要であることを示している。

それに対して音響旋律の場合は、音の高さや長さの相対的な関係よりも、音そのものが持っている絶対的な高さや長さが優先する。40[Hz]の線による音響旋律の高さを100[Hz]に移動させたり、3秒の面による音響旋律の長さを10秒に引き延ばすことで、音響旋律全体が有している聴覚的な経験や意味が変化する。そうした各々の音が持っているパラメータの絶対値は、広義の音色としての音響全体に対して、本質的な役割を担っている。そこに音響旋律が「音響」旋律たる所以があり、音楽をいわゆる(古典的な)旋律やリズム、和音の集まりとしてではなく、常に音を音色や響きを核とした音そのものとして捉え、そこから旋律やリズム、和音の意味や機能を再構築していくことの具体的なアプローチがある。

線の思考

変奏は旋律という音響の水平構造を拡大する。原型としての旋律を変奏することは、ある旋律を時間軸に沿って、直列に延長していくことでもある。こうした水平的、直列的な思考を、音響要素のレイアウトに関する「線の思考」と呼ぶ。「線の思考による旋律の連鎖」は、聴き手の意識の流れや、音響の時間的な持続をコントロールするための、最もシンプルで基本的な方法である。

▼エクササイズ：線の思考による4つの変奏

3秒の原型に4つの変奏を付加することで15秒の旋律をデザインせよ。

最初に原型として3秒の旋律を設定し、それに4つの変奏を繋げていくことで、原型を線の思考で発展させていく。

「反行の移高」のように、いくつかのメソッドを組み合わせてもよいし、異なるパラメータに同じメソッドを適用することもできる。原型を操作してつくられた変奏を第一世代の変奏だとすれば、さらにその変奏をつくることで、第二世代、第三世代というように、変奏の子孫を次々と生み出していくこともできる。変奏の可能性は多様である。

最も単純な連鎖の方法として、同種の変奏が反復していく一次元的な流れを考える。例えば、

・原型→原型の変奏1→原型の変奏2→原型の変奏3→原型の変奏4（第一世代の変奏の連鎖）
・原型→原型の変奏1→変奏1の変奏2→変奏2の変奏3→変奏3の変奏4（変奏の世代の連鎖）

のような組み合せが考えられる。

```
// フェイドアウト時間を0秒に設定
p.fadeTime = 0;
```

点列による原型の拡大・縮小による4つの変奏

```
// 原型の定義
(
~out = { arg base=21, fls=1, flb=0, als=1, alb=0, ts=1;
    var farray, aarray, tarray, envf, enva;
    farray = Array.rand(11, 0.0, 1.0).postln; // ランダムな周波数列
    aarray = ([0] ++ Array.rand(9, 0.5, 0.9) ++ [0]).postln; // ランダムな振幅列
    tarray = Array.fill(10, 0.3); // 一定の継続時間列
    envf = Env.new(farray, tarray).plot; // 周波数エンヴェロープの生成
    enva = Env.new(aarray, tarray).plot; // 振幅エンヴェロープの生成
    Impulse.ar(base * EnvGen.kr(envf, 1, fls, flb, ts, 2), 0, EnvGen.kr(enva, 1, als, alb, ts, 2)).dup }
)

// TaskProxyの定義
x = TaskProxy.basicNew;

// TaskProxyの開始と終了
x.play;
x.stop;


// 拡大による4つの変奏
(
var fls;
x.source = {
    5.do ({ |i|
        fls = 2 ** i; // 周波数エンヴェロープが倍々に拡大していく
        fls.postln;
```

```
        ~out.spawn([\fls, fls]);
        3.wait;
    })
}
)

// 縮小による4つの変奏
(
var ts;
x.source = {
    5.do ({ |i|
        ts = 0.5 ** i; // 時間スケールが半々に縮小していく
        ts.postln;
        ~out.spawn([\ts, ts]);
        (3 * ts).wait;
    })
}
)


// 別の原型を定義する
(
~out = { arg base=4410, fls=1, flb=0, als=1, alb=0, ts=1;
    var farray, aarray, tarray, envf, enva;
    farray = Array.geom(8, 0.5, 1.25).postln; // 一定の比で上昇する周波数列
    aarray = Array.rand(8, 0.2, 0.5); // ランダムな振幅列
    tarray = Array.fill(4, 0.4) ++ Array.fill(3, 0.2); // 継続時間のパターン
    envf = Env.new(farray, tarray).plot;
    enva = Env.new(aarray, tarray).plot;
    Impulse.ar(base * EnvGen.kr(envf, 1, fls, flb, ts, 2), 0, EnvGen.kr(enva, 1, als, alb, ts, 2)).dup }
)


// 継続時間のパターンが一定のまま、周波数と振幅がランダムに拡大・縮小する変奏の繰り返し
(
var fls, als;
x.source = {
    loop {
        fls = rrand(0.1, 2.0);
        als = rrand(0.1, 2.0);
        fls.postln;
        als.postln;
        ~out.spawn([\fls, fls, \als, als]);
        2.5.wait;
    }
}
)
```

継続時間のパターンが一定であれば、周波数や振幅の大きな変奏が知覚されやすい。

▼エクササイズ：モチーフとしてのエンヴェロープ

あるひとつのエンヴェロープ(リスト)をモチーフとし、それを別のパラメータに適用することで変奏を展開せよ。

```
// 上昇するモチーフを周波数、振幅、時間に対してランダム適用することによる変奏の繰り返し
(
~out = { arg base=8810, fls=1, flb=0, als=0.1, alb=0, ts=0.2;
    var array0, array1, tarray0, tarray1, envf, enva;
    array0 = Array.rand(11, 0.9, 1.1); // ランダムにゆらぐエンヴェロープ
    array1 = Array.series(11, 1.0, 0.1); // モチーフとしての上昇するエンヴェロープ
    tarray0 = Array.fill(10, 1.0);
    tarray1 = Array.series(10, 1.0, 0.1);
    envf = Env.new([array0, array1].choose, [tarray0, tarray1].choose); // 2つの周波数列をランダムに選択
    enva = Env.new([array0, array1].choose, [tarray0, tarray1].choose); // 2つの振幅列をランダムに選択
    SinOsc.ar(base * EnvGen.kr(envf, 1, fls, flb, ts, 2), 0, EnvGen.kr(enva, 1, als, alb, ts, 2)).dup }
)

// 変奏の3秒間隔での繰り返し
x.source = { loop { ~out.rebuild; ~out.spawn; 3.wait; }}
```

▼エクササイズ：変奏メソッドの探求(1)

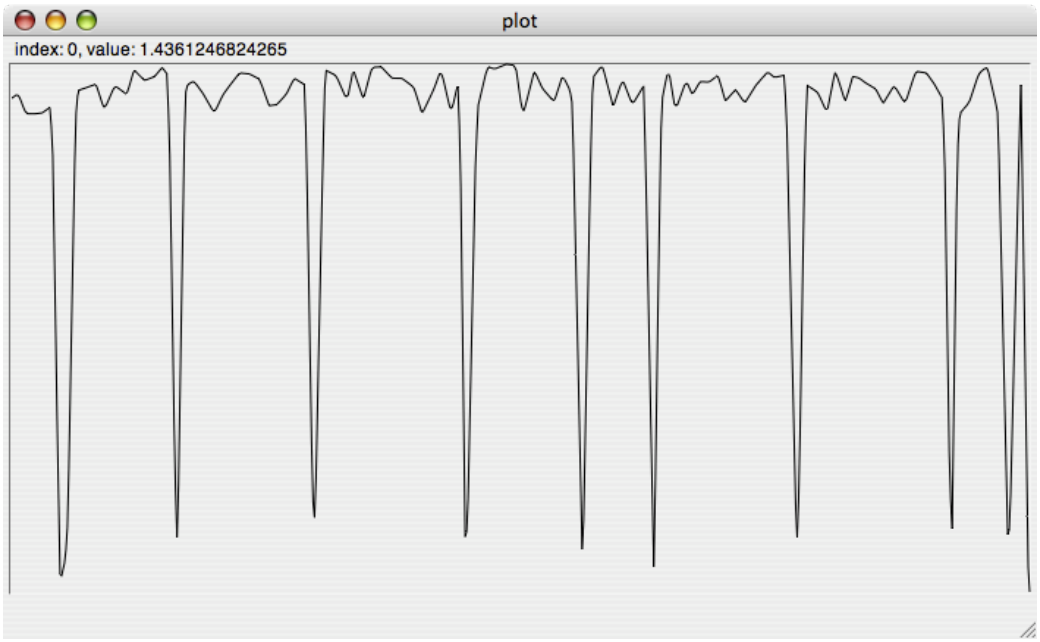
移高や拡大・縮小、置換やなどの値によるメソッドの変奏を徹底的に探求せよ。微細な変化から極端な変化まで、さまざまな変奏を試みよ。

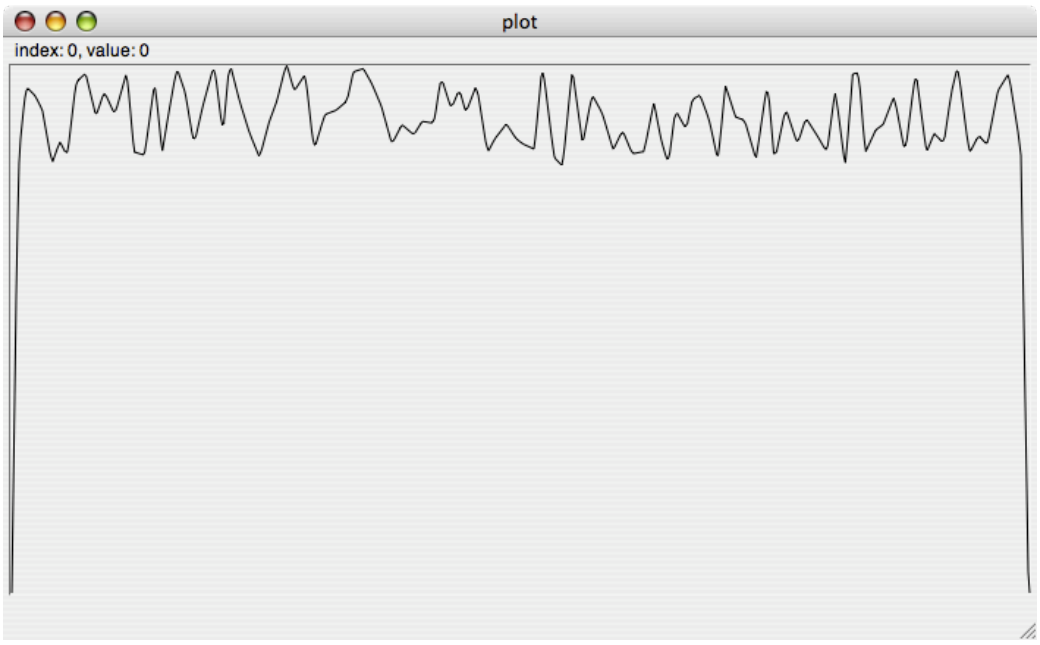
```
// 点列の極端/微細な移高と拡大・縮小による4つの変奏
(
~out = { arg base=10, fls=1, flb=0, als=1, alb=0, ts=1;
    var farray, aarray, tarray, envf, enva;
    farray = Array.rand(11, 0.0, 1.0); // ランダムな周波数列
    aarray = [0] ++ Array.rand(9, 0.4, 0.5) ++ [0]; // ゆるやかに変化する振幅列
    tarray = Array.rand(10, 0.2, 0.4); // ゆるやかに変化する継続時間列
    envf = Env.new(farray, tarray);
    enva = Env.new(aarray, tarray);
    Impulse.ar(base * EnvGen.kr(envf, 1, fls, flb, ts, 2), 0, EnvGen.kr(enva, 1, als, alb, ts, 2)).dup }
)

// 継続時間のパターンが一定のまま、周波数と振幅が大きく拡大・縮小するランダムな変奏の繰り返し
(
var fls, als;
x.source = {
    loop {
        fls = [ rrand(1, 1.1), rrand(1000.0, 1100.0)].choose;
        als = [ rrand(0.1, 0.2), rrand(1.2, 1.5)].choose;
        fls.postln;
        als.postln;
        ~out.spawn([\fls, fls, \als, als]);
        3.wait;
    }
}
)
```

順序や要素、リストによるメソッドで変奏を行ってみよ。前エクササイズ同様、微細な変化から極端な変化まで、さまざまな変奏を試みよ。

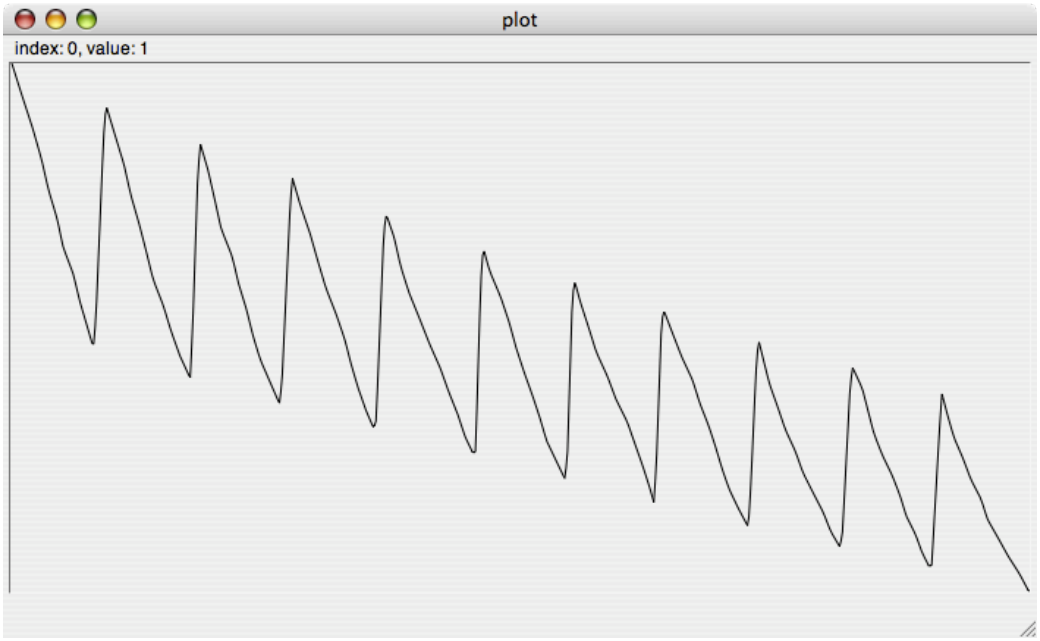
```
// 変奏の3秒間隔での繰り返し
x.source = { loop { ~out.rebuild; ~out.spawn; 3.wait; }}
x.play;
x.stop;

// ランダムな要素の回転による変奏
(
~out = { arg base=11050, fls=1, flb=0, als=1, alb=0, ts=1;
    var farray, aarray, tarray, envf, enva;
    farray = Array.series(11, 1.0, -0.01); // ゆるやかに下行する周波数列
    aarray = ([0] ++ Array.rand(9, 0.1, 0.3) ++ [0]); // ランダムな振幅列
    tarray = Array.fill(10, 0.3); // 一定の継続時間列
    envf = Env.new(farray.rotate(farray.size.rand.postln), tarray); // 周波数列をランダムに回転する
    enva = Env.new(aarray.rotate(farray.size.rand.postln), tarray); // 振幅列をランダムに回転する
    SinOsc.ar(base * EnvGen.kr(envf, 1, fls, flb, ts, 2), 0, EnvGen.kr(enva, 1, als, alb, ts, 2)).dup }
)


// 時間要素のスクランブルによる変奏
(
~out = { arg base=14700, fls=1, flb=0, als=1, alb=0, ts=0.2;
    var farray, aarray, tarray, envf, enva;
    farray = Array.fill(11, {|i| i % 2}); // ジグザグに振動する周波数列
    aarray = Array.fill(11, 0.3); // 一定の振幅列
    tarray = Array.fill(10, {|i| (i % 2) + 1}); // 一定の継続時間列
    envf = Env.new(farray, tarray.scramble, \step); // 継続時間列をスクランブル
    enva = Env.new(aarray, tarray.scramble); // 継続時間列をスクランブル
    SinOsc.ar(base * EnvGen.kr(envf, 1, fls, flb, ts, 2), 0, EnvGen.kr(enva, 1, als, alb, ts, 2)).dup }
)


// 要素の挿入(追加)による変奏の生成
(
~out = { arg base=11050, fls=1, flb=0, als=1, alb=0, ts=1;
    var farray, aarray, tarray, envf, enva;
    farray = Array.rand(11, 0.5, 0.6); // ランダムな周波数列
    100.do{farray = farray.insert(farray.size.rand, rrand(1.4, 1.5))}; // 100個の要素をランダムに挿入
```

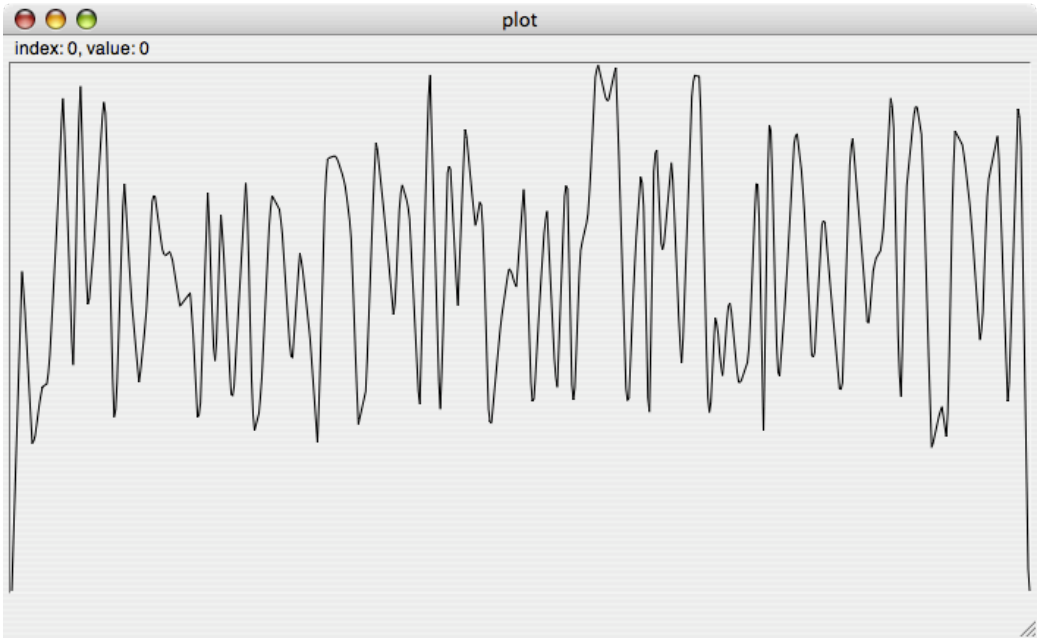
```
//  100.do{farray = farray.add(rrand(1.2, 1.5))}; // 100個の要素を追加
    aarray = [0] ++ Array.rand(farray.size-2, 0.4, 0.5) ++ [0]; // 小さく変動する振幅列
    tarray = Array.rand(farray.size-1, 0.02, 0.04); // 小さく変動する継続時間列
    envf = Env.new(farray, tarray); // 周波数エンヴェロープ
    enva = Env.new(aarray, tarray); // 振幅エンヴェロープ
    SinOsc.ar(base * EnvGen.kr(envf, 1, fls, flb, ts, 2), 0, EnvGen.kr(enva, 1, als, alb, ts, 2)).dup }
)
```






```
// リストの演算による変奏の生成
(
~out = { arg base=11050, fls=1, flb=0, als=1, alb=0, ts=1;
    var array1, array2, farray, aarray, tarray, envf, enva;
    array1 = Array.geom(11, 1.0, rrand(0.95, 1.05)); // 上行あるいは下行する周波数列
    array2 = Array.geom(11, 1.0, rrand(0.95, 1.05)); // 上行あるいは下行する周波数列
    farray = array1 *.x array2; // 2つの周波数列の乗算(すべての要素の組み合わせ)
    aarray = [0] ++ Array.rand(farray.size-2, 0.1, 0.4) ++ [0]; // ランダムに変動する振幅列
    tarray = Array.rand(farray.size-1, 0.02, 0.04); // 小さく変動する継続時間列
    envf = Env.new(farray, tarray); // 周波数エンヴェロープ
    enva = Env.new(aarray, tarray); // 振幅エンヴェロープ
    SinOsc.ar(base * EnvGen.kr(envf, 1, fls, flb, ts, 2), 0, EnvGen.kr(enva, 1, als, alb, ts, 2)).dup }
)
```





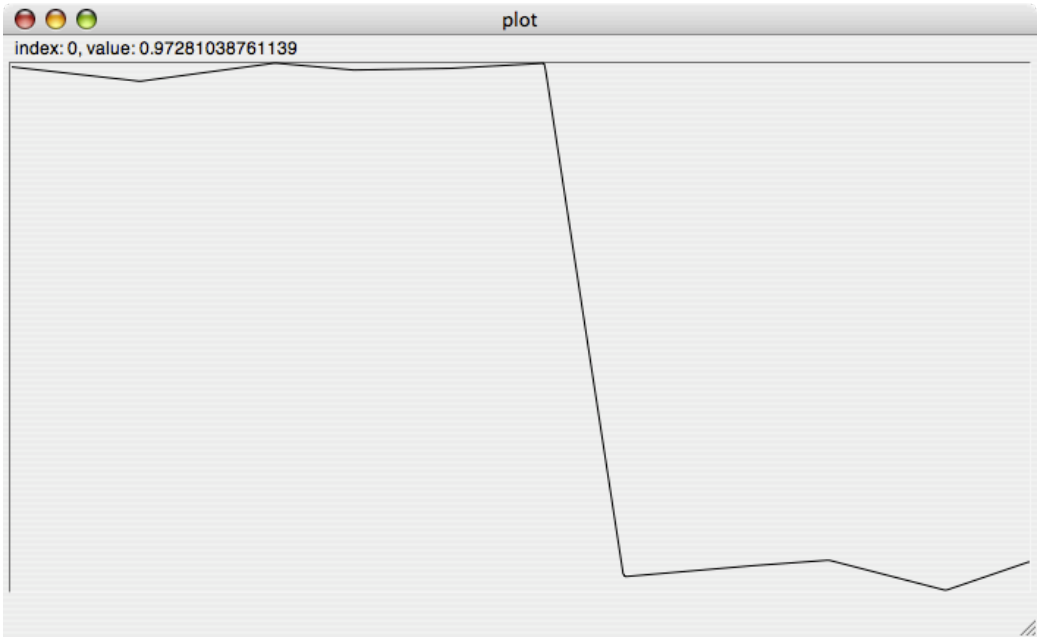
▼エクササイズ：変奏メソッドの探求(2)

インタラクティブ・リスニングの手法でさまざまな変奏の可能性を探求せよ。

値による変奏のインタラクティヴな探索

```
// エンヴェロープのトリガー用プロキシの定義
~trig.kr(1);
~trig = { Impulse.kr(0.2) }; // トリガー間隔=5秒

// エンヴェロープのパラメータ用プロキシの定義
~fls.kr(1);
```

```
~flb.kr(1);
~als.kr(1);
~alb.kr(1);
~ts.kr(1);

// 基本周波数用プロキシの定義
~base.kr(1);

// 原型の定義
(
~out = { var farray, aarray, tarray, envf, enva;
    farray = Array.rand(6, 0.9, 1.0) ++ Array.rand(5, 0.0, 0.1); // 原型の周波数列
    aarray = [0] ++ Array.series(farray.size-2, 0.3, 0.05) ++ [0]; // 原型の振幅列
    tarray = Array.rand(farray.size-1, 0.2, 0.4); // 原型の継続時間列
    envf = Env.new(farray, tarray).plot;
    enva = Env.new(aarray, tarray).plot;
    SinOsc.ar(~base.kr * EnvGen.kr(envf, ~trig.kr, ~fls.kr, ~flb.kr, ~ts.kr), 0,
    EnvGen.kr(enva, ~trig.kr, ~als.kr, ~alb.kr, ~ts.kr)).dup }
)
```



```
~fls = 1;
~flb = 0;
~als = 1;
~alb = 0;
~ts = 1;
~base = 100;

// マウスのX座標を周波数スケール(拡大・縮小)に対数マッピング
~fls = { MouseX.kr(10, 100, 1) };

// マウスのX座標を周波数バイアス(移高)に線形マッピング
~fls = 50.0;
~flb = { MouseX.kr(0, 100, 0) };

// マウスのY座標を継続時間スケール(拡大・縮小)に対数マッピング
~ts = { MouseY.kr(0.1, 2.0, 1) };
```

エンヴェロープの混合のインタラクティヴな探索

```
// エンヴェロープの混合比用プロキシの定義
~bf.kr(1);

// 原型の定義
(
~out = { var farray, aarray, tarray, envf, enva;
    farray = Array.fill(11, {|i| i % 2}); // 原型の周波数列
    aarray = Array.series(farray.size-1, 0.3, 0.05) ++ [0]; // 原型の振幅列
    tarray = Array.rand(farray.size-1, 0.2, 0.4); // 原型の継続時間列
    envf = Env.new(farray, tarray).plot;
    enva = Env.new(aarray, tarray).plot;
    SinOsc.ar(~base.kr * EnvGen.kr(blend(envf, enva, ~bf.kr), ~trig.kr, ~fls.kr, ~flb.kr, ~ts.kr), 0,
    EnvGen.kr(blend(enva, envf, ~bf.kr), ~trig.kr, ~als.kr, ~alb.kr, ~ts.kr)).dup }
)

// マウスのX座標をエンヴェロープの混合パラメータにマッピング
~bf = { MouseX.kr(0.0, 1.0) };
```

変奏のデザイン

数式や数列を使って、特定の志向性を持たないランダム(確率的)な変奏から「音響旋律のコンポジション」へと発展させていく。

線の変奏(移高による変奏)

「線」を使って、高さ(周波数)のデザインを行う。

まず、単純な1本の線を一定の振幅でランダムに高さを変化させる。

```
// TaskProxyの定義と線の定義のアサインメント
x = TaskProxy.basicNew;
~out.awake = false;
~out = \line;
x.play;

// 4410[Hz]を初期値とし±5%の範囲でランダムに高さが変動する線(継続時間0.08秒、0.1秒間隔)
(
var a=0.3, f=4410.0, p=0.0, s=0.08;
x.source = {
    loop {
        f = (f * rrand(0.95, 1.05)).postln;
        ~out.spawn([\amp, a, \freq, f, \pan, p, \sustain, s]);
        0.1.wait;
    }
}
)
```

次に、線の高さの変化に上行、あるいは下行の方向性を持たせる。

```
// 441[Hz]を初期値とし0〜5%の範囲でランダムに上昇していく線(継続時間0.08秒、0.1秒間隔)
(
```

```
var a=0.3, f=441.0, p=0.0, s=0.08;
x.source = {
    loop {
        f = (f * rrand(1.0, 1.05)).postln;
        if (f > 22050) {f = 441.0};
        ~out.spawn([\amp, a, \freq, f, \pan, p, \sustain, s]);
        0.1.wait;
    }
}
)

// 8820[Hz]を初期値とし0〜5%の範囲でランダムに下降していく線(継続時間0.08秒、0.1秒間隔)
(
var a=0.3, f=8820.0, p=0.0, s=0.08;
x.source = {
    loop {
        f = (f * rrand(0.95, 1.0)).postln;
        if (f < 20) {f = 8820.0};
        ~out.spawn([\amp, a, \freq, f, \pan, p, \sustain, s]);
        0.1.wait;
    }
}
)
```

上下行を交互に繰り返す、すなわちジグザグに高さを変化させる。時間と共に次第にジグザグの振幅を大きくしていく。

```
// 4410[Hz]を初期値としランダムに上下行(ジグザグ)する線(継続時間0.08秒、0.1秒間隔)
(
var a=0.3, f1=4410.0, f2=4410.0, p=0.0, s=0.08;
x.source = {
    loop {
        f1 = (f1 * rrand(1.0, 1.05)).postln;
        f2 = (f2 * rrand(0.95, 1.0)).postln;
        if ((f1 > 22050) || (f2 < 20)) {f1 = 4410.0; f2 = 4410.0};
        ~out.spawn([\amp, a, \freq, f1, \pan, p, \sustain, s]);
        0.1.wait;
        ~out.spawn([\amp, a, \freq, f2, \pan, p, \sustain, s]);
        0.1.wait;
    }
}
)
```

数列(漸化式)を使って線の動きをコントロールする。

漸化式は一般的に

```
a(n+1) = r * a(n) + d
```

と書くことができる。

```
// 4410[Hz]を初期値とし漸化式によって動きがコントロールされる線(継続時間0.08秒、0.1秒間隔)
(
var a=0.3, f=4410.0, p=0.0, s=0.08, r=1.1, d= -442.0;
x.source = {
    loop {
        f = r * f + d;
        f.postln;
        if ((f > 22050) || (f < 20)) {f = 4410.0};
        ~out.spawn([\amp, a, \freq, f, \pan, p, \sustain, s]);
        0.1.wait;
    }
}
)

// 漸化式の係数をダイナミックに変化させ、自律的に動きをコントロールする線(継続時間0.08秒、0.1秒間隔)
(
var a=0.3, f=4410.0, p=0.0, s=0.08, r, d, fbase;
fbase = f;
r = rrand(0.9, 1.1);
d = rrand(-10.0, 10.0);
x.source = {
    loop {
        f = r * f + d;
        r = r * rrand(0.9, 1.1);
        d = (fbase - f) * rrand(0.9, 1.1);
        f.postln;
        if ((f > 22050) || (f < 20)) {f = 4410.0; r = rrand(0.9, 1.1); d = rrand(-10.0, 10.0)};
        ~out.spawn([\amp, a, \freq, f, \pan, p, \sustain, s]);
        0.1.wait;
    }
```

```
}
)

x.stop;
~out.awake = true;
```

▼エクササイズ：高さのデザイン

数列を使って、線の上下行を中心とした旋律をデザインせよ。点列を用いて同じアプローチで旋律をデザインせよ。

点の変奏(拡大・縮小による変奏)

「点」を使って、間隔(時間)のデザインを行う。

まず、単純な1つの点を一定の間隔で鳴らす。

```
// 点の定義のアサインメント
~out.awake = false;
~out = \points;
x.play;

// 1秒間隔の点(密度が一定の点列)
(
var a=0.8, f=1.0, p=0.0, s=0.01, t=1.0;
x.source = {
    loop {
        ~out.spawn([\amp, a, \freq, f, \pan, p, \sustain, s]);
        t.wait;
    }
}
)

// 点の間隔を0.1〜1秒の間でランダムに変化させる(密度がゆらぐ点列)
(
var a=0.8, f=1.0, p=0.0, s=0.01, t=1.0;
x.source = {
    loop {
        t = rrand(0.1, 1.0);
        ~out.spawn([\amp, a, \freq, f, \pan, p, \sustain, s]);
        t.wait;
    }
}
)

// 点の間隔を0.01秒から1秒までランダムに拡大していく(密度が減少していく点列)
(
var a=0.8, f=1.0, p=0.0, s=0.01, t=0.01;
x.source = {
    loop {
        t = t * rrand(1.0, 1.1);
        ~out.spawn([\amp, a, \freq, f, \pan, p, \sustain, s]);
        t.wait;
        if (t > 1.0) {t = 0.01};
    }
}
)

// 点の間隔を1秒から0.01秒までランダムに縮小していく(密度が増加していく点列)
(
var a=0.8, f=1.0, p=0.0, s=0.01, t=1.0;
x.source = {
    loop {
        t = t * rrand(0.9, 1.0);
        ~out.spawn([\amp, a, \freq, f, \pan, p, \sustain, s]);
        t.wait;
        if (t < 0.01) {t = 1.0};
    }
}
)
```

```
// 点の間隔が0.1秒を出発点にジグザグに拡大、縮小する
(
var a=0.8, f=1.0, p=0.0, s=0.01, t1=0.1, t2=0.1;
x.source = {
    loop {
        t1 = t1 * rrand(0.9, 1.0);
        t2 = t2 * rrand(1.0, 1.1);
        if ((t1 < 0.01) || (t2 > 1.0)) {t1 = 0.1; t2 = 0.1};
        ~out.spawn([\amp, a, \freq, f, \pan, p, \sustain, s]);
        t1.wait;
        ~out.spawn([\amp, a, \freq, f, \pan, p, \sustain, s]);
        t2.wait;
    }
}
)

// 0.1秒を初期値としランダムな係数の漸化式によってその間隔がコントロールされる点
(
var a=0.8, f=1.0, p=0.0, s=0.01, t=0.1, r=1.0, d=0.0;
x.source = {
    loop {
        r = rrand(0.9, 1.1);
        d = (t * 0.1).rand2;
        t = r * t + d;
        t.postln;
        if ((t > 1.0) || (t < 0.01)) {t = 0.1};
        ~out.spawn([\amp, a, \freq, f, \pan, p, \sustain, s]);
        t.wait;
    }
}
)

x.stop;
~out.awake = true;
```

▼エクササイズ：間隔のデザイン

数列を使って、点の間隔を変化させることによる旋律をデザインせよ。

点列の周波数を変化させることで、間隔と高さの両方を遷移する旋律のデザイン行ってみよ。

```
// 点の定義のアサインメント
~out.awake = false;
~out = \points;
x.play;


// ランダムな係数の漸化式によって点列の周波数を0.1[Hz](10秒間隔)から22050[Hz]まで変化させる
(
var a=0.8, f=10.0, p=0.0, s=0.1, t=0.1, r=1.0, d=0.0;
x.source = {
    loop {
        r = rrand(0.8, 1.3);
        d = (f * 0.5).rand2;
        f = r * f + d;
        f.postln;
        if ((f > 22050) || (f < 0.1)) {f = 10.0};
        ~out.spawn([\amp, a, \freq, f, \pan, p, \sustain, s]);
        t.wait;
    }
}
)
```

断続する面を用いて 持続と間隔のデザインを行なってみよ。

```
// 面のアサインメント
~out = \plane;

// ランダムな係数の漸化式によって面の持続と間隔を0.01秒から0.2秒まで変化させる
(
var a=0.2, p=0.0, s=0.1, t=0.1, rs=1.0, ds=0.0, rt=1.0, dt=0.0;
x.source = {
    loop {
```

```
        rs = rrand(0.8, 1.2);
        ds = (s * 0.5).rand2;
        s = rs * s + ds;
        rt = rrand(0.8, 1.2);
        dt = (t * 0.5).rand2;
        t = rt * t + dt;
        s.post; ", ".post; t.postln;
        if ((s > 0.2) || (s < 0.01)) {s = 0.1};
        if ((t > 0.2) || (t < 0.01)) {t = 0.1};
        ~out.spawn([\amp, a, \pan, p, \sustain, s]);
        t.wait;
    }
}
)

x.stop;
~out.awake = true;
```