連載

SUPER COLLIDER チュートリアル (2)
SUPER COLLIDER TUTORIALS (2)

美山 千香士
Chikashi Miyama
ケルン音楽舞踏大学
Hochschule für Musik und Tanz Köln

概要

本連載では、リアルタイム音響合成環境の SuperCollider(SC) の使い方を、同ソフトを作品創作や研究のために利用しようと考えている音楽家、メディア・アーティストを対象にチュートリアル形式で紹介する。
SuperCollider(SC) is a realtime programming environment for audio synthesis. This article introduces SC to musicians and media artists who are planning to utilize the software for their artistic creations and researches.

1. 今回の目標：SC によるメロディの演奏

前回は SC[1] のインストール、ソフトウェアの構成、音の出し方などのプログラミングの基礎を勉強した。今回は、前回の内容を発展させ、簡単なメロディを自動的に演奏するプログラムを作り、それを通して SC で音楽を作る上で必要不可欠な以下の 3 項目を順に学習していく。

- **Env** と **EnvGen** を用いた「音符」作り
- **SynthDef** を用いた「楽器」作り
- **Routine** を用いた「楽譜」作り

2. 「音符」を作る

前回、SinOsc などの UGen を用いて SC で音を出す方方法を学習したが、それらはストップコマンド [Cmd+.] などで強制的に終了させない限り音が止まらない「持続音」であった。これでは今回の目標である「メロディ」を演奏するのは難しい。故に、まずはじめに一定時間で発音が止まる音価のある音、「音符」を作る必要がある。このために SC では**エンベロープ**を用いる。エンベロープを作るには **Env** というクラスと **EnvGen** という UGen を組み合わせる。Env はエンベロープの「設計」を行い、EnvGen は Env を使って設計したエンベロープを生成し、音に適用できるようにする役割を担う。

2.1. エンベロープの設計

SC では、エンベロープは、目標とする値とその値に達する推移時間の組み合わせによって定義する事ができる[1]。例えば、リスト 1 は、Env を用いて 0.5 秒のアタックと 1 秒のサステイン、0.5 秒リリースのみの簡単なエンベロープを設計したものである。

リスト 1. エンベロープの設計

```
1 Env.new([0,1,1,0],[0.5,1.0,0.5]).plot;
```

Env を使ってこのようにエンベロープを設計し、.plot というメッセージを送ると、エンベロープがプロット (図示) される。Env の第 1 引数の [ ] の中の数値がそれぞれエンベロープのノード (節点) の値を、続く第 2 引数の [ ] 内の数値がノード間の時間を指定している (図 1)。

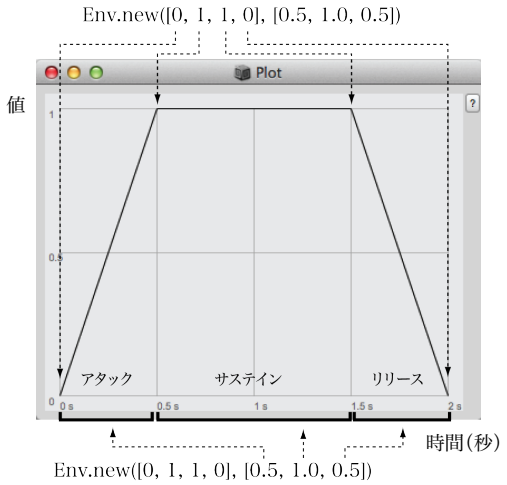

図 1. エンベロープ

[1] 本稿では音の長さの決まった固定長のエンベロープのみを扱うが、可変長のエンベロープを用いる事も可能である。この場合は Env.adsr と Env.asr などによりエンベロープを定義する。詳しくは Env のヘルプファイルを参照のこと。

また、リスト 2 のように、さらに.test を.plot の前に挟むと、自動的に SC がこのエンベロープを正弦波の振幅に適用したものを試奏するので、エンベロープを聴感で吟味したいときに便利である。尚、.test を使う時は SC Server を事前に起動する必要がある。

リスト 2. .test の利用

```
1 e=Env.new([0,1,1,0],[0.5,1,0.5]);
2 e.test.plot;
```

2.2. Array-複数のデータをまとめて保存する

リスト 1 で、Env は 2 つの引数を与えられているが、そのいずれもが [ ] に囲まれ、その中に複数の数値が列挙されている。このように SC では、複数のデータをコンマで区切って列挙し、[ ] で囲ったものを Array(配列) という。Array は複数のデータを集合として扱うときに重宝する。例えば、リスト 3 の 1 行目の [5,8,2,4,9] は 5 つの数値を Array を用いてひとまとめにしている。

リスト 3. Array の例

```
1 a=[5,8,2,4,9];
2 a.at(3);
```

Array は数値などと同様に、変数に代入する事ができるため、複数のデータをまとめて 1 つの変数名で扱えるという利点がある。変数に代入した Array の *n* 番目の要素を参照するには、リストの 2 行目のように Array を格納した変数に.at(*n*) を送る。図 2 の示すように、Array 内の要素のインデックス (カウント) は 0 から始まるため、.at(3) を送ると Array から 4 が返される。SC では実行したコードの最終行の評価結果が自動的にポスト・ウインドウに表示されるので、リスト 3 を実行するとポスト・ウインドウに「4」が表示される。

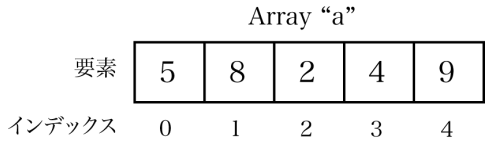

図 2. Array の要素とインデックスの関係

Array 内の各要素を順番に参照し、ポスト・ウインドウに表示させる場合は、リスト 4 のように記述する。Array の.do を実行すると、.do の引数として渡された {} で囲まれたコードが、Array の要素数の回数 (この場合は 5 回) 繰り返し実行される。関数が実行される度に、引数 item には Array の各要素が順番に代入される。このように.do を使って Array 内の各要素を順番に評価していく手法を**イテレーション**という。また、プログラム中の item.postln は「item の中身をポスト・ウインドウに表示せよ」という意味である。前述のように、プログラムの最後に実行したコードの評価はポスト・ウインドウに自動的に表示されるが、そうでない場合は、このように.postln を使って強制的にポスト・ウインドウへの表示を促すことができる。

リスト 4. イテレーション

```
1 a = [5,8,2,4,9];
2 a.do({
3   arg item;
4   item.postln;
5 });
```

Array には、整数や少数などの数値だけでなく、リスト 5 のように様々なタイプのデータを混ぜて格納することも可能である。さらに、Array の中に Array を入れ子のように格納することもできる。このような場合は、リスト 5 の 2 行目のように、a.at(3) で Array a のインデックス 3 の要素、[2,10,-1] を参照し、それにさらに.at(2) を送ることで、Array の中の Array のインデックス 2 の要素、-1 を取り出す事ができる。

リスト 5. Array に数値以外のものを格納する

```
1 a=["JSSA",-17,0.995,[2,10,-1]];
2 a.at(3).at(2);
```

Array は.at の他にも様々な命令を受けつける、例えば.sum は Array 内の全ての要素の合計を計算、.sort は Array 中の要素を昇順に並べ替え、.rotate は Array の要素を () で指定した数だけ Array の最後尾から先頭に移動、.indexOf は () 内で指定した要素が Array の何番目に格納されているのかを返す (リスト 6)。5 行目のように、Array 対して四則演算を行う事も可能で、この場合は Array 内の全ての要素に 10 が加算され、Array が [15,18,12,14,19] となる。

リスト 6. Array の受け付ける様々な命令

```
1 [5,8,2,4,9].sum.postln;
2 [5,8,2,4,9].sort.postln;
3 [5,8,2,4,9].rotate(2).postln;
4 [5,8,2,4,9].indexOf(9).postln;
5 ([5,8,2,4,9]+10).postln;
```

Env

エンベロープを設計するためのクラス。

.new(*levels*, *times*, *curve*, *releaseNode*, *loopNode*, *offset*)

levels · · · エンベロープのノードの値。Array で指定する
times · · · ノード間の補間時間。Array で指定する
curve · · · ノード間の補間方法。
(残りの引数の解説はヘルプファイルを参照のこと)

2.3. エンベロープの適用

さて、Array の役割と使用法を学んだので、次に Array を使って設計したエンベロープを使って音作りを行う。エンベロープを音に適用するには、**EnvGen** という UGen を用いてエンベロープをオーディオ信号として生成し、それを適用の対象となる信号と乗算する。リスト 7 は Saw.ar を使って生成したノコギリ波に Env で設計したエンベロープを EnvGen を用いて適用したものである。

リスト 7. エンベロープの適用

```
1 {
2   e=Env.new([0,1,1,0],[0.5,1,0.5]);
3   Saw.ar(880,0.1)*EnvGen.ar(e);
4 }.play;
```

EnvGen

エンベロープを生成する UGen。

.ar(*envelope*, *gate*, *levelScale*, *levelBias*,*timeScale*, *doneAction*)
.kr(*envelope*, *gate*, *levelScale*, *levelBias*,*timeScale*, *doneAction*)

Env · · · Env によって設計したエンベロープ
gate · · · ゲート値。0 より大きい値が入力されるとエンベロープの生成を開始する
levelScale · · · レベルの伸縮値
levelBias · · · レベルのオフセット値
timeScale · · · エンベロープの補間時間の伸縮値
doneAction · · · エンベロープ終了時の処理を指定する値

2.4. エンベロープ終了時の処理指定-doneAction

SC ウインドウの右下にあるステータス・バーは現在のサーバの状況を表示しており、左から順番に「CPU 使用率 (平均)」「CPU 使用率 (ピーク)」「使用中の UGen 数」「使用中の Synth 数」「使用中の Group 数」「定義された SynthDef(後述) の数」が表示されている (図 3)。この図のステータス・バーはサーバを立ち上げた直後のもので、何も演奏していないので、UGen の数が 0u となっている。

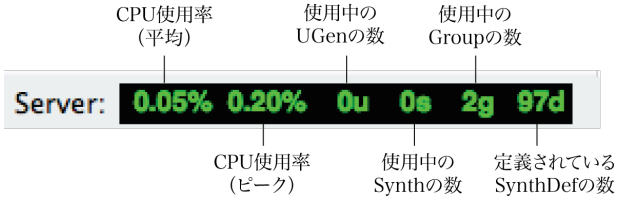

図 3. ステータス・バーの項目

リスト 7 を実行すると、SC のステータスバーが、11u、1s、2g を表示し、プログラムが、いくつかの UGen を使って音を生成しているのが分かるが[2]、2 秒後にエンベロープが終わり、無音状態になっても、この表示が変わらず、UGen を使い続けており、SC が無用な計算を続けていることがわかる (図 4)。

図 4. エンベロープ開始から 2 秒以降のステータス

このようなリソースの無意味な消費を回避するために、エンベロープが終了すると同時に SC に計算を停止させる必要がある。これを行うには EnvGen の引数の末尾に **doneAction:2** を加え、リスト 8 のように変更する。このコードを実行すると、発音の開始から 2 秒後に UGen の数が 11u から 0u に自動的に変化し、発音のために使用した UGen がエンベロープの終了に同期して解放されたのが分かる。

リスト 8. doneAction を指定

```
1 {
2   e=Env.new([0,1,1,0],[0.5,1,0.5]);
3   Saw.ar(880,0.1) * EnvGen.ar(e,doneAction
      :2);
4 }.play;
```

このようにエンベロープの終了 (done) の時の振る舞い (Action) を設定するのが doneAction である。これに「2」を指定する事により、終了と同時にこの音に関するプロセスを停止するので、無用な計算リソースの消費を防ぐ事が可能になる。doneAction には 2 以外にも 0 から 14 までの数を与える事で、終了時の振る舞いを様々に指定する事ができる。詳しくはヘルプファイル、「UGen done-actions」を参照されたい。

2.5. キーワードによる引数の指定

前項において、「doneAction:2 」という特殊な表記によって EnvGen の引数を指定した。SC では、UGen などに引数を渡すときには、リスト 9 のように、それぞれの引数が何のパラメータをコントロールするのかをヘル

[2] リストでは EnvGen と Saw の 2 つの UGen しか使用していないが、ステータスバーには 11u と表示される。これは、SC が適切なオーディオ・チャンネルに音を送る処理などのために自動的に幾つかの UGen を使うためである。後述する SynthDef を用いた場合は、リスト中の UGen の数とステータス・バーの UGen の数は一致するようになる。

プファイルで調べ、順番に引数を渡していくのが一般的である。

リスト 9. 一般的な引数の指定

```
{
  SinOsc.ar(880,0.0,0.1,0.0);
}.play;
```

例えば、リスト 9 において、SinOsc.ar の引数は freq, phase, mul, add であるので、SinOsc.ar(880, 0.0, 0.1, 0.0) のように書くことでそれぞれ freq = 880, phase = 0.0, mul = 0.1, add = 0.0 を指定できる。だが、**キーワード** (引数の名前) を使って特定の引数だけを設定する事もできる。キーワードですべてのパラメータを指定した場合は、引数の順番に留意する必要がなくなる。例えばリスト 10 では、freq の前に mul を指定しているが、正常に作動する。

リスト 10. キーワードによる引数の指定

```
{
  SinOsc.ar(mul:0.1,freq:880);
}.play;
```

また、リスト 11 のように順番に引数を与えていく手法とキーワードによる引数の指定は混在してもかまわない。だが、この場合キーワード指定でない引数は、必ず順番に先頭から記述する必要がある。

リスト 11. 引数指定法の混在

```
{
  SinOsc.ar(880,mul:0.1);
}.play;
```

リスト 8 の EnvGen.ar(e, doneAction:2) は、このように 2 つの引数の指定方法を混在させた例である。UGen には EnvGen のように非常に多くの引数をとれるものもあり、その際にこの事を知っていると、より簡潔かつ柔軟にプログラムを書くことができる。

2.6. エンベロープの応用

エンベロープはオシレータからの波形の振幅をコントロールする使い方が最も一般的だが、これ以外にも例えば周波数や、フィルタのパラメータをコントロールする事もできる。リスト 12 では、ノコギリ波の周波数の上げ下げをエンベロープを用いて行っている。このように、エンベロープはさまざまなパラメータのコントロールに柔軟に利用できる。

リスト 12. エンベロープの応用

```
{
  e=Env.new([220,880,440],[1,0.5]);
  Saw.ar(EnvGen.ar(e, doneAction:2), 0.1);
}.play;
```

3. 「楽器」を作る

3.1. 自分だけの楽器を定義する-SynthDef

これまで勉強してきたように、SC では様々な UGen を組み合わせて、独自の音作りをすることができる。そして、SC には気に入った音が出るプログラムを自分独自の **Synth**(楽器) として定義・保存し、定義した Synth を様々な作品で再利用できる仕組みが用意されている[3]。

Synth を定義するには、リスト 13 の要領で **SynthDef** を使う。SynthDef とは文字通り Synth を定義 (Define) するクラスで、第 1 引数 (”jssa1”) でその Synth の名前を設定し、第 2 引数はその Synth を定義する。この定義の中で使われている UGen、**Out** は音を送る先のチャンネルを指定している。リストの 4 行目のように、Out の第 2 引数に UGen を代入した変数を指定し、第 1 引数に 0 を指定すると左チャンネルに、1 を指定すると右チャンネルに音を送る設定となる。

リスト 13. SynthDef の使用

```
SynthDef.new("jssa1", {
  a=Saw.ar(440,0.1);
  e=Env.new([0,1,1,0],[0.05,0.1,0.1]);
  Out.ar(0,a*EnvGen.ar(e, doneAction:2));
}).load;
```

定義した Synth を実際に生成して使用するにはリスト 14 のように、SynthDef で定義した Synth の名前を Synth の第 1 引数に指定する。

リスト 14. 定義した Synth を使う

```
Synth.new("jssa1");
```

現段階では、この Synth は 440Hz の音しか出せないが、Synth の周波数を自由に設定・変更できるようにすることもできる。そのためには、SynthDef の中に arg を用いて freq という引数をリスト 15 のように宣言する。このように arg を使うと、Synth が外部から引数を取る事ができるようになる。

リスト 15. SynthDef の使用法

```
 SynthDef.new("jssa2",{
   arg freq;
   a=Saw.ar(freq, 0.1);
   e=Env.new([0,1,1,0],[0.05,0.1,0.1]);
   Out.ar(0,a*EnvGen.ar(e,doneAction:2));
 }).load;
```

周波数を指定して Synth を使用するには、Synth の第 2 引数として Array を与え、その中に「引数名」「数値」のペアを記述する。

[3] SC では今回の例のように発音を担うものはもちろん、エフェクタ、アナライザー、ミキサーなど様々なものを Synth として定義できる

リスト 16. arg の指定

```
1 Synth.new("jssa2",["freq",880]);
```

これを応用すれば、和音を SC に演奏させるなども容易である。例えば以下の 3 行を選択し、同時に実行すれば長三和音が生成される。

リスト 17. 和音

```
1 Synth.new("jssa2",["freq",60.midicps]);
2 Synth.new("jssa2",["freq",64.midicps]);
3 Synth.new("jssa2",["freq",67.midicps]);
```

arg は SynthDef の中で複数個宣言する事もできる。リスト 18 の SynthDef では周波数の他に振幅も arg を介して指定できるようにしている。それぞれの arg を指定するにはリスト 19 のように、Synth の第 2 引数の Array 内に引数名と値を列挙していく。また、SynthDef 内で、arg には freq = 440, amp = 0.1 のように初期値を設定しておくことができる。この初期値は arg に対しパラメータが与えられなかった時に用いられる。

リスト 18. SynthDef の使用法

```
1 SynthDef.new("jssa3",{
2    arg freq = 440,amp = 0.1;
3    a=Saw.ar(freq,amp);
4    e=Env.new([0,1,1,0],[0.05,0.1,0.1]);
5    Out.ar(0,a*EnvGen.ar(e,doneAction:2));
6 }).load;
```

リスト 19. arg の指定

```
1 Synth.new("jssa3",["freq",880,"amp",0.5]);
```

Out

信号の出力先のバスを指定するための UGen

.ar(*bus*, *channelsArray*)
.kr(*bus*, *channelsArray*)

bus ··· 出力先のバス

channelsArray ··· バスに送る信号。複数の信号を送る場合は Array を渡す。Array が渡された場合、*bus* で指定されたバスから順番に信号が送られる。バスの概念については回を改めて詳説する。

3.2. SynthDef ファイル

SynthDef を用いて Synth を命名、定義した後に、定義した Synth を SC サーバに登録する必要がある。リスト 18 の最後の.load メッセージは、SC サーバへの登録を行うためのコマンドであり、これを指定しないと定義した Synth を使う事はできない。

また、一度 SynthDef に.load を送ると、その定義が書かれた SynthDef という定義ファイルが、Mac では「~/Library/Application Support/SuperCollider/synthdefs」に保存される。一度ここに保存された SynthDef は SC サーバを再度立ち上げた時に自動的にサーバに登録され、再定義する必要なくなる。このため、自分の気に入った音を出す Synth を予め定義し、SynthDef ファイルに保存しておくと、複数のプロジェクトで 1 つの Synth を流用する事などが容易となる。

このように.load を送ると、SC サーバへの Synth の登録と、SynthDef ファイルへの定義の書き出しが同時に行われるが、SynthDef で定義した Synth を SynthDef ファイルを保存することなく、使用したい場合は、.load メッセージの代わりに.send を送る (図 5)。

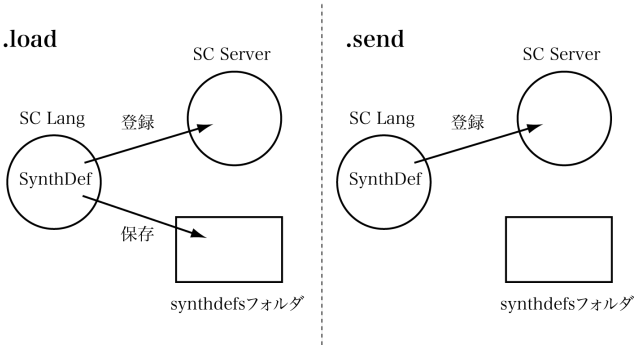

図 5. .load と.send の違い

SynthDef

Synth を定義し、保存するためのクラス。

.new(*name*, *ugenGraphFunc*, *rates*, *prependArgs*, *variants*, *metadata*)

name ··· 名称。Synth により参照される

ugenGraphFunc ··· Synth の定義

(全ての引数はヘルプファイルを参照のこと)

Synth

SynthDef で定義した Synth を使用する際に用いる。
.new(*defname*, *args*, *target*, *addAction:'addToHead'*)

name ··· 事前に定義した SynthDef の名称

args ··· Synth に渡すパラメータを列挙した Array

(全ての引数はヘルプファイルを参照のこと)

4. 「楽譜」を作る

前項で「楽器」を Synth として定義した事で、より柔軟に任意の音のパラメータをコントロールできるようになったので、あとはこの Synth の発音のタイミングを時間軸上にスケジュールする、いわば「楽譜」をプログラムする事で、SC にメロディを奏でさせることが可能となる。

4.1. 任意のタイミングで音を出す-Routine

SC でユーザの任意の時間に音が鳴るようにするには **Routine** を使う。使い方は、リスト 20 のように Routine の引数の {} の中に Synth を使った発音を促すコードを書き、その間に処理を遅延させる.wait を挟んでいく。

リスト 20. Routine の使用

```
Routine({
  Synth.new("jssa3",["freq",60.midicps]);
  0.5.wait;
  Synth.new("jssa3",["freq",64.midicps]);
  1.0.wait;
  Synth.new("jssa3",["freq",67.midicps]);
}).play;
```

リスト 20 を実行すると、MIDI ノート・ナンバーの「60」、つまりピアノの真ん中のドが演奏され、その 0.5 秒後に 64(ミ)、そのさらに 1 秒後に 67(ソ) が演奏される。このように.wait を使って処理を遅延させる事で任意のタイミングで SC に発音を促す事が可能となる。

4.2. 旋律を演奏する

リスト 21 は上記の Routine を利用して「メリーさんの羊」の冒頭部分を演奏するプログラムである。

リスト 21. メリーさんの羊

```
Routine({
  a = [[69,0.3],[67,0.1],[65,0.2],[67,0.2],
[69,0.2],[69,0.2],[69,0.2]];
  a.do({
    arg item;
    Synth.new("jssa3",["freq",item.at(0).
        midicps]);
    item.at(1).wait;
  });
}).play;
```

リストの冒頭では、各音の MIDI ノートナンバーと音価のペアを複数の Array として宣言し、それをさらに Array、a に格納している。そして、Array a の各要素を上述した.do を用いたイテレーションの手法を使って、順番に参照しており、item は [69, 0.3], [67, 0.1], [65, 0.3]... という順番でピッチと音価のペアの Array に置き換えられる。そして item の 0 番目の要素=ピッチのデータは Synth に渡され、1 番目の要素=音価のデータは.wait に用いられているため、任意のピッチの音を任意のタイミングに鳴らすこと、つまりメロディーを奏でる事が可能となっている。

4.3. 移調・テンポチェンジ

「メリーさんの羊」はリスト 21 ではヘ長調であるが、リスト 22 のように item.at(0) の後に-6 を追加する事によって、半音階で 6 度下のハ長調に移調する事もできる。また、このコードでは item.at(1) に 1.5 を乗算し、.wait による待機時間を延ばすことによって、テンポを遅くしている。

リスト 22. 移調

```
Routine({
  a = [[69,0.3],[67,0.1],[65,0.2],[67,0.2],
    [69,0.2],[69,0.2],[69,0.2]];
  a.do({
    arg item;
    Synth.new("jssa3",["freq",(item.at
        (0)-6).midicps]);
    (item.at(1)*1.5).wait;
  });
}).play;
```

このように若干の変更をプログラムに加えることで、音楽の特徴を大幅に変化させることが出来るところが、SC で音楽を作る醍醐味の 1 つといえるだろう。

5. 謝辞

本稿の執筆にあたり、校正を手伝って頂いた理化学研究所・岡ノ谷情動情報プロジェクトの濱野峻行氏に心からの感謝の気持ちとお礼を申し上げたい。

6. 参考文献

[1] *SuperCollider*, http://supercollider.sourceforge.net/(アクセス日 2013 年 7 月 3 日)

7. 著者プロフィール

美山 千香士 (Chikashi Miyama)

作曲家、電子楽器創作家、映像作家、パフォーマー。国立音楽大学音楽デザイン学科より学士・修士を、スイス・バーゼル音楽アカデミーよりナッハ・ディプロムを、アメリカ・ニューヨーク州立バッファロー大学から博士号を取得。Prix Destellos 特別賞、ASCAP/SEAMUS 委嘱コンクール 2 位、ニューヨーク州立大学学府総長賞、国際コンピュータ音楽協会賞を受賞。2004 年より作品と論文が国際コンピュータ音楽会議に 12 回入選、現在までに世界 18 カ国で作品発表を行っている。 2011 年、DAAD(ドイツ学術交流会) から研究奨学金を授与され、ドイツ・カールスルーエの ZKM で客員芸術家として創作活動に従事。近著に「Pure Data-チュートリアル＆リファレンス」(Works Corporation 社) がある。現在ドイツ・ケルン音楽大学、及びチューリッヒ芸術大学非常勤講師。バーゼル音楽大学客員講師。ICST(チューリッヒ コンピュータ音楽・音響技術研究所)SpatDIF プロジェクト研究員。Pure Data Japan (http://puredatajapan.info) 共同創設者。http://chikashi.net