

CindyScript / CindyJS Reference

CinderellaJapan

2024年3月26日

目次

1	CindyScript の全般的仕様	6
1.1	関数型プログラミング言語	6
1.2	コードの書式	6
2	制御命令	7
2.1	繰り返し	7
2.1.1	while	7
2.1.2	repeat	7
2.1.3	forall	8
2.2	条件分岐 if	9
2.3	トリガー trigger	10
2.4	強制的評価 eval	10
3	変数	10
3.1	大域的変数と局所変数	11
3.2	変数の作成と削除	12
3.3	局所変数の宣言	13
3.4	オブジェクトのキー変数	13
4	算術関数	15
4.1	四則演算と累乗	15
4.2	剰余	16
4.3	角度演算子	16
4.4	絶対値	16
4.5	距離演算子	17
4.6	標準的な算術関数	17

4.7	四捨五入	18
4.8	床関数と天井関数	18
4.9	複素数	18
4.10	乱数	18
4.11	ブール関数	19
4.11.1	比較	19
4.11.2	ブール代数	19
4.12	演算子一覧	20
5	型の判定	21
6	文字列関数	22
6.1	文字列の結合	22
6.2	文字列への変換	22
6.3	文字列の長さ	22
6.4	文字列の抜き出し	23
6.5	文字列の検索	23
6.6	文字列の分解	23
6.7	文字列の置換	24
6.8	文字列の解析	24
6.9	文字列の比較と並べ替え	25
7	リスト	25
7.1	リストの作成とアクセス	25
7.2	リストの操作	27
7.3	リストの要素の走査	28
7.4	要素の組み合わせ	29
7.5	要素の整列	30
7.6	要素の総和・積	31
7.7	最大と最小	32
8	ベクトルと行列	32
8.1	ベクトルと行列の定義	32
8.2	和と積	33
8.3	ベクトルと行列の演算	33
8.4	線形代数の演算	34
8.5	3次元の凸多面体を作る	35

9	幾何学要素へのアクセス	36
9.1	要素にその名前でアクセスする	36
9.2	幾何学要素のリスト	37
9.3	幾何学要素のパラメータ	37
9.4	インスペクタの要素	42
9.5	要素の作成と消去	43
9.6	幾何要素の操作	44
10	図形の描画	46
10.1	修飾子	46
10.2	色や大きさの設定	47
10.3	色の関数	48
10.4	描画関数	49
11	関数プロット	51
11.1	実数関数のグラフを描く	51
11.2	動的な色と透明度	54
11.3	カラープロット	55
11.4	ベクトル場	57
11.5	グリッド	60
11.6	オシログラフ	62
11.7	文字と表	64
12	TeX 記法 Cinderella TeX	67
13	画像の操作とレンダリング	72
13.1	メディアブラウザ	72
13.2	画像の表示と変換	72
13.3	カスタム画像を作る	81
13.4	キャンバスに描く	81
14	シェイプ	85
14.1	シェイプの初歩	85
14.2	シェイプの結合	85
14.3	シェイプの使い方	85
15	座標系と基底	88
15.1	座標変換	88
15.2	射影基底との関係	90

15.3	基底スタック	92
15.4	レイヤー	93
15.5	スクリーン境界の決定	95
16	幾何学的演算	96
16.1	リストと座標	96
16.2	基本的な幾何学関数	96
16.3	有用な線形代数関数	97
16.4	変換とオブジェクトの型	98
16.5	幾何学変換と基底	99
17	計算	100
17.1	微分と接線	100
17.2	高度な計算	101
18	音声出力 Syntherella	102
18.1	概要	102
18.2	単音	102
18.3	旋律	103
18.4	音色	107
18.5	サウンド関数	109
19	ファイル管理	111
19.1	データの読み込み	111
19.2	データの書き出し	111
19.3	HTML との連携	112
19.4	ネットワーク	113
19.5	コンソールへの出力	114
20	時間とアニメーション	114
20.1	時間	114
20.2	アニメーションのコントロール	115
21	ユーザー入力	115
21.1	マウスとキーボード	115
21.2	加速度センサにおける AMS データ	116
22	CindyLab との連携	116
22.1	シミュレーション環境	116

23	Cindy3D	118
23.1	設定	118
23.2	描画関数	119
23.3	光の當て方と表現	126
24	CindyJS	132
24.1	CindyJS の HTML 基本構造	132
24.1.1	CindyJS ランタイムのロード	133
24.1.2	スクリプトの記述	133
24.1.3	CindyJS の初期化	134
24.2	CindyJS の HTML 文書の実例	135
25	索引	138

1 CindyScript の全般的仕様

1.1 関数型プログラミング言語

CindyScript のすべての計算は関数によって行われる。いろいろな計算は、基本的な関数によってなされる。通常の計算のように $a+b$ のようにして計算をすることもでき、「+」は演算子と呼ばれるが、CindyScript では、これを挿入演算子と呼んでいる。同じ計算が、関数 `add(a,b)` によってもできる。そこで、このマニュアルでは、「演算子」と「関数」をほぼ同義なものとして使っている。したがって、`add(a,b)` のような関数も「演算子」と呼ぶことがある。

さらに、if 文のような一見手続き風の命令文が、関数として認識される。例えば、

```
if(x<y,print("Mine"),print("Yours"))
```

という文は、3つの引数を持つ if 関数の例である。第 1 の引数 `x` の条件を評価し、その結果により 第 2 または第 3 の結果 `print("Mine")` または `print("Yours")` を実行し、戻り値とする。これは次の命令文と同じである。`print(if(x<y,"Mine","Yours"))`

1.2 コードの書式

CindyScript では、命令文の区切りにセミコロン ; を書く。これは必須である。単なる改行は命令文の続きと見なされる。したがって、関数を使うときの括弧をそのままブロックを表すものとして、その内部で改行することもできる。ブロック内のインデントは特に必要はないが、可読性のためににはつけるとよい。

命令文の最後にはセミコロンをつけなくてもよい。

【例】

`repeat(9,i, j=i*i; draw([i,j]))` は、次のようにブロックとして書くことができる。

```
repeat(9,i,  
       j=i*i;  
       draw([i,j]);  
)
```

2 制御命令

2.1 繰り返し

2.1.1 while

```
while(<bool>,<expr>)
```

while 関数は <bool> が真である間、<expr> を評価する。

【例】：0 から 4 までの和を求める。

```
x=0;  
sum=0;  
while(x<4,  
      x=x+1;  
      sum=sum+x;  
);  
println(sum);
```

参考：while は関数なので戻り値がある。戻り値は最後の評価結果である。上の例では sum の値が戻る。

2.1.2 repeat

```
repeat(<int>,<expr>)
```

<expr> を <int> 回繰り返し、最後の評価結果が戻る。<expr> が評価されている間、実行変数 # は繰り返しの回数を数える。

【修飾子】次の修飾子を任意に組み合わせることができる。

- start ループの開始の値を設定する。
- stop ループの終了の値を設定する。
- step ステップの値を設定する。

【例】スクリプト

```
repeat(6, println(#+" "))  
repeat(6, start->4, println(#+" "))
```

結果

1 2 3 4 5 6

4 5 6 7 8 9

repeat(6, stop->2, println(#+" "))	-3 -2 -1 0 1 2
repeat(6, step->3, println(#+" "))	1 4 7 10 13 16
repeat(6, stop->12,step->4, println(#+" "))	-8 -4 0 4 8 12
repeat(6, start->3,step->2, println(#+" "))	3 5 7 9 11 13
repeat(6, start->3,stop->4, println(#+" "))	3 3.2 3.4 3.6 3.8 4
repeat(6, start->0,stop->-3, println(#+" "))	0 -0.6 -1.2 -1.8 -2.4 -3
repeat(6, start->3,stop->4,step->0.4, println(#+" "))	3 3.4 3.8

`repeat(<int>,<var>,<expr>)`

この関数は、実行変数が `<var>` に割り当てられるという 1 点を除いては、`repeat(<int>,<expr>)` と同じ。これは、異なる実行変数を使って、ループを入れ子にすることを考慮している。

【例】 10 × 10 の点列を描く

```
repeat(10,i,
  repeat(10,j,
    draw((i,j));
  );
)
```

2.1.3 forall

リストを使って繰り返しをする。

`forall(<list>,<expr>)`

この関数の第 1 引数はリスト。`<expr>` がリストのそれぞれの要素に対して評価される。実行変数# はリストのそれぞれの要素になる。

【例】 リストの全要素を改行してコンソールに表示する。

```
a=["this","is","a","list"];
forall(a,println(#))
```

`forall(<list>,<var>,<expr>)`

実行変数が `<var>` であること以外は `forall(<list>,<expr>)` と同じ。

2.2 条件分岐 if

```
if(<bool>,<expr>)
```

<bool> が真であれば、<expr> が評価 (実行) され、その値が返される。

【例】

```
if(x<0,println("x は負です"))
```

このコードでは、x が負の値であれば、"x は負です" が印字 (表示) され、その文字列がそのまま戻り値として返される。負の値でなければ、何も印字されず、 が返される。

```
if(<bool>,<expr1>,<expr2>)
```

条件 <bool> が真であれば <expr1> が評価され、そうでなければ <expr2> が評価され、その値が返される。したがって、この if 文は、`if then else` に相当する。この条件分岐の典型的な使用法は 2 つある。まず、条件によって場合分けをするときに使う。

【例】

```
if(x<0,
    println("x は負です")
    ,
    if (x>0,
        println("x は正です")
        ,
        println("x は零です")
    )
)
```

このコードでは、x の値が正か、負か、ゼロかを表示する。

第 2 の使い方は、<bool> の条件により値を返したいときで if 関数の戻り値を利用する。これは関数の定義としてよく使う。

【例】

```
f(x):=if(x>0,x,-x)
```

このコードは、x の絶対値を返す関数 f(x) を定義する。

【例】

```
A.color = if(A.x > 0, (1, 0, 0), (0, 0, 1))
```

このコードでは、点 A の色を、その x 座標の正負により赤か青に設定する。

2.3 トリガー trigger

```
t trigger(<bool>, <expr>)
```

trigger は動的意味あいを持つ条件判断である。<bool> が false から true に変わったときに評価される。たとえば、何かの要素をドラッグしているとき、<bool> が false から true に変化したことを察知して <expr> を評価（実行）する。この演算子の目的は、要素をドラッグしているときに何らかのイベントが起きたかどうかを判断することである。次のコードはそのような例である。

【例】

```
trigger(A.x < 0, println("A は x が負の半平面に入りました"));
trigger(A.x > 0, println("A は x が正の半平面に入りました"))
```

このコードでは、点 A が y 軸を横切るとメッセージを出す。

2.4 強制的評価 eval

```
eval(<expr>, <modif1>, <modif2>, ...)
```

この関数は、<expr> の内容を強制的に評価する。<expr> 内の変数は modifier のリストにある変数で置き換えられる。置き換える変数は局所変数。実行後はもとの値に書き戻される。

【例】次のコードは、評価値を 7 とする。

```
eval(x+y, x->2, y->5)
```

3 変数

CindyScript の基本的な型は次の通り。

- ・ 数値 : 数値の型は、整数、浮動小数点数、複素数
- ・ リスト : 任意の値のリスト。ベクトルや行列はリストで表される
- ・ 文字列 : 文字列
- ・ 幾何要素 : 幾何学的要素

- ・ブール値 : `true` または `false` のブール値

しかしながら、CindyScript は値の型を明示しない。変数は特定の型を持たず、どんな型のどんな値でも変数に代入することができる。例えば、関数 `f(x,y)` を定義する次のコード

```
f(x,y):=x+y;
```

は、整数、複素数、ベクトル、行列のいずれでも使えるが、関数が意味不明なことをすると、「意味がない」ことを表す `---` 印を返す。したがって、上の例で、`f([1,2],[3,4])` ならばベクトルの和として `[4,6]` を返すが、`f(4,[3,4])` では意味がなく、`---` を返す。

3.1 大域的変数と局所変数

変数に代入した値がどこまで有効かを表すのに、「大域的 (global)」と「局所的 (local)」という概念がある。どこで使われていても変数の値が変化しないのが「大域的」、関数の内部や繰り返しループなどの内部だけで有効な場合を「局所的」という。CindyScript の変数は、基本的には大域的である。したがって、関数定義などで使用した変数がすでに使われていてなんらかの値が代入されていることがある。

【例】

```
f(x):=(  
    x+y;  
)  
x=1;  
y=1;  
println(f(5));
```

この例では、`x` は引数なので、`f(5)` を実行すると `x=5` が代入される。`y` の値は関数内部では代入されていないが、呼び出す前に `y=1` としているので、この値が使われる。結果は

```
6  
1
```

となる。

繰り返しの `repeat` で使う変数は局所的である。

【例】

```
x=5;  
repeat(2,x,  
    println(x);  
)
```

```
println(x);
```

結果は

```
1  
2  
5
```

となる。

3.2 変数の作成と削除

変数の作成: `createvar(<varname>)`

変数の削除: `removevar(<varname>)`

これらの関数は、局所変数の生成について手動で管理するのを助ける。

`createvar(x)` は新しい変数 `x` を作り、前の値はスタックに入れられる。

`removevar(x)` は局所変数を削除して、スタックの値を書き戻す。

通常、変数は明示的に作っておく必要はない。それらは、最初に使われるときに自動的に作られる。`createvar` と `removevar` は、コードの特定の部分に、ある名前で変数を予約したいときだけ使われる。

【例】

```
x=10;  
println("x is now "+x);  
createvar(x);  
x=5;  
println("x is now "+x);  
removevar(x);  
println("x is now "+x);
```

このコードの実行結果は次の通り。

```
x is now 10  
x is now 5  
x is now 10  
  
clear()
```

引数なしで、すべての変数をクリアする。特定の変数をクリアする場合は `clear(<var>)` と引数に渡す。

3.3 局所変数の宣言

`regional(name1,name2,...)`

関数内で局所変数を使う。

この関数は、ユーザーが定義する関数の最初に使う。引数にある複数の変数を局所変数として定義する。`local` 関数と似ているが、関数内での処理が終わると自動的に消滅する。したがって、`release` などで消去する手続きは不要で `local` 関数より便利に使える。

【例】

```
f(x):=(  
    regional(y);  
    y=2;  
    x+y;  
)  
x=1;  
y=1;  
println(f(5));
```

この例では、`y` の値は関数内部で 2 が代入されているが、この `y` は局所変数として扱われる所以、呼び出す前の `y` の値は変わらず 1 のままである。

結果は

```
7  
1
```

となる。

関数を呼び出すことで、その前の `y` の値が変わってしまうのを防ぐことができる。

3.4 オブジェクトのキー変数

`<key>=<something>` で宣言した `<object>` のキー変数のリストを得る。

【例】オブジェクトのキー変数を割り当てる。

```
A:"age"=34;
```

```
A:"haircolor"="brown";
```

割当をした結果はつぎのような構文で利用できる。

```
println(A:"age");
println(A:"haircolor");
```

`keys` 関数は、オブジェクトに割り当てられた変数のリストを取得する。したがって、次のコードではリスト `["age", "haircolor"]` が表示される。

```
println(keys(A));
```

4 算術関数

4.1 四則演算と累乗

四則演算は、数学の基本的な演算子 $+$, $-$, $*$, $/$, $^$ がそのまま使える。これらは、「挿入演算子」と呼ばれ、数とリストに適用できるが、演算の意味は、適用されるものによって異なる。たとえば、 $5+7$ の結果は 12 であり、 $[2,3,4]+[3,-1,5]$ の結果は $[5,2,9]$ となる。これらの演算子のほかに、関数でも演算ができる。

演算	演算子	関数
和	$+$	<code>add(<expr1>, <expr2>)</code>
差	$-$	<code>sub(<expr1>, <expr2>)</code>
積	$*$	<code>mult(<expr1>, <expr2>)</code>
商	$/$	<code>div(<expr1>, <expr2>)</code>
累乗	$^$	<code>pow(<expr1>, <expr2>)</code>

商については、 \div でも演算ができる。（ \div は日本語入力で入力する。Unicode）

数（整数、実数、複素数）のべき乗では、指数は整数に限らず、実数、複素数でも可能。正式には `exp(b*ln(a))` が計算される。`(ln(a))` は複素変数の対数関数）

`ln()` は周期 $2\pi i$ で定義されるので、 a^b は一般には多値関数。 b が整数でない場合には a^b は最初の主要な値を返す。

【例】	式	結果	
	$2.3 + 5.9$	8.2	実数の和
	<code>add(5, 6)</code>	11	整数の和
	$(2+3*i) + (5+9*i)$	$7+i*12$	複素数の和
	$(2+3*i) - (5+9*i)$	$-3 - i*6$	複素数の差
	<code>sub([3,5]), [1,2])</code>	[2,3]	リストの差
	$[2,3,6] - [3,2,4]$	[-1,1,2]	リスト（ベクトル）の差
	$[2,3,[1,2]] + [3,4,[1,3]]$	[5,7,[2,4]]	リストの和
	$[2,3,4] + [3,4,[1,2]]$	[5,7,___]	（第3要素が未定義値となる）
	$7 * 8$	56	整数の積
	$(1+i) * (2+i)$	$1+3*i$	複素数の積
	$2 * [5,3,2]$	[10,2,4]	ベクトルの実数倍
	$[5,3,2] * 2$	[10,2,4]	ベクトルの実数倍
	$[2,2,3] * [3,4,6]$	32	ベクトルの内積
	$[[1,2],[3,4]] * [1,2]$	[5,11]	行列と列ベクトルの積

$[1, 2] * [[1, 2], [3, 4]]$	$[7, 10]$	行ベクトルと行列の積
$[[1, 2], [3, 4]] * [[1, 2], [3, 4]]$	$[[7, 10], [15, 22]]$	行列の積
$[6, 8, 4]/2$	$[3, 4, 2]$	ベクトルの実数倍と同じ
$\text{mult}(2, [3, 4])$	$[6, 8]$	ベクトルの実数倍
$\text{mult}([4, 5], [3, 4])$	32	ベクトルの内積
$8/2$	4	整数の商
$8 \div 2$	4	整数の商
$\text{div}(8, 2)$	4	整数の商
$(2-i) / (1+i)$	$0.5 - i*1.5$	複素数の商
6^2	36	整数の整数乗
$\text{pow}(6, 2)$	36	整数の整数乗
$2^{(1/2)}$	$1.4142 \dots$	整数の分数乗
2^i	$0.7692 + i*0.639$	整数の i 乗

4.2 剰余

`mod(<expr1>, <expr2>)` 関数は、`<expr1>`を `<expr2>`で割った余りを求める。

4.3 角度演算子

`<数>°` は、数に $\pi/180$ をかける。度数法(60分法)による角度を弧度法に変換する。
「°」は日本語入力モードで入力する。

【例】`sin(30°)` の結果は 0.5

4.4 絶対値

| `<obj>` | 演算子、および関数 `abs(obj)` は、`obj` の絶対値を求める。`obj` は実数、複素数、ベクトルのいずれか。

| `<obj>` | 演算子を $\|3 + |x - 2|\|$ のように入れ子にして使うことはできない。入れ子にしたい場合は `abs()` 関数を用いる。

`abs()` は、実数、複素数、ベクトルなどのノルムを計算する。

【例】 `abs([1, 3, 1, 2, 1])` の結果は 4

4.5 距離演算子

$|<\text{obj1}, \text{obj2}>|$ 演算子, および `dist(obj1,obj2)` 関数は, 2つのオブジェクトの距離を計算する。オブジェクトは, 実数, 複素数, ベクトルのいずれかで, 2つとも同じ種類のものとする。

【例】 $|[1,1],[4,5]|$ の結果は 5

$|<\text{obj1}, \text{obj2j}>|$ 演算子を入れ子にして使うことはできない。入れ子にしたい場合は `dist()` 関数を用いる。

4.6 標準的な算術関数

平方根	<code>sqrt(<expr>)</code>
指数関数	<code>exp(<expr>)</code>
自然対数:	<code>log(<expr>)</code>
三角関数	<code>sin(<expr>)</code>
三角関数	<code>cos(<expr>)</code>
三角関数	<code>tan(<expr>)</code>
sine の逆関数	<code>arcsin(<expr>)</code>
cosine の逆関数	<code>arccos(<expr>)</code>
tangent の逆関数	<code>arctan(<expr>)</code>
ベクトルのなす角	<code>arctan2(<実数 1, 実数 2>)</code>
ベクトルのなす角	<code>arctan2(<vector>)</code>

三角関数は複素関数なので引数に複素数を与えることができる。

`arc` は原則として多値だが, 関数は $-\pi$ から π の間の解をひとつ返す。

平方根は演算子として $\sqrt{ }$ も使える。 $\sqrt{ }$ は日本語入力モードで入力する。

【例】 `sin(2+3*i)` の結果は 9.1545 - i*4.1689

`arctan2(1,1)` と `arctan2([1,1])` の結果はいずれも 45°

Cinderella で作図した点を A としたとき, `arctan2(A)` は OA が x 軸の正の方向となす角を返す。(O は原点)

$\sqrt{2}$ の結果は 1.412 · · ·

4.7 四捨五入

`round(<expr>)` は、小数点以下を四捨五入する。小数点以下の桁を指定して四捨五入する関数は用意されていないので、たとえば、小数点以下第 4 位を四捨五入する場合には、1000 倍して四捨五入したのち 10000 で割る。

リストにも適用できる。複素数に対しては、実部と虚部のそれぞれに対して適用される。

【例】

```
round(pi*1000)/1000      結果は 3.142  
round([3.2,7.8,3.1+i*6.9]) 結果は [3,8,3+i*7]
```

4.8 床関数と天井関数

床関数：その数以下の最大の整数: `floor(<expr>)`

天井関数：その数以上の最小の整数: `ceil(<expr>)`

複素数に対しては、実部と虚部のそれぞれに対して適用される。

4.9 複素数

複素数の実部: `re(<expr>)`

複素数の虚部: `im(<expr>)`

共役複素数: `conjugate(<expr>)`

4.10 亂数

<code>random()</code>	0 より大きく 1 未満の一様乱数
<code>random(<数>)</code>	0 より大きく <数> 未満の実数の一様乱数
<code>randomint(<数>)</code>	0 以上 <数> 未満の整数の一様乱数
<code>randomnormal()</code>	(0,1) 正規乱数
<code>randombool()</code>	ブール値の乱数 true または false
<code>seedrandom(<数>)</code>	乱数発生器の初期化

`random` 関数は、負の数や複素数を引数にすることもできる。例えば、`random(-5)` は -5 から 0 までの乱数を発生する。`randomint(6+i*10)` は実部が 0 から 6 まで、虚部が 0 から 10 までの複素数をランダムに発生する。

疑似乱数は、常に予測不可能な新しい乱数列を発生する。なんらかの理由で、スクリプトを実行するたびに、同じ乱数列を発生させたい場合は、`seedrandom()` 関数を使う。引数に

与えた整数に対し、同じ乱数列が発生する。異なる引数 (seed) を与えれば別の乱数列が発生する。

4.11 ブール関数

条件付き分岐では真偽の判定が必要になるが、これに関する演算がブール演算である。演算子もしくは関数により、true または false が返される。

4.11.1 比較

`<expr1> == <expr2>` 2つの式の評価結果が等しい。
`<expr1> != <expr2>` 2つの式の評価結果が等しくない。

以下は、ともに実数であれば、通常の大小関係。

ともに文字列であれば、辞書順に比較。比較できない値の場合は `___` が返される。

`<expr1> > <expr2>` expr1 が、expr2 より大きい。
`<expr1> < <expr2>` expr1 が、expr2 より小さい。
`<expr1> >= <expr2>` expr1 が、expr2 より大きいか等しい。
`<expr1> <= <expr2>` expr1 が、expr2 より小さいか等しい。

ファジー (あいまい) な比較：`~=`, `~!=`, `~<`, `~>`, `~>=`, `~<=` (チルダと比較記号)

CindyScript は、比較を ファジー (あいまい) におこなう演算子を持っている。この演算子では、条件が ε の限界内かどうかを調べる。したがって、 $a \sim= 0$ は、値 a が $+ \varepsilon$ から $- \varepsilon$ の範囲にあるかどうかを調べる。小さな値 ε は 0.0000000001 に設定される。この演算子は、数値計算における誤差を吸収するのに役に立つ。たとえば、 $x == y$ が論理的に等しいにもかかわらず、`if(x == y, • •)` が正しく動作しない場合、`if(x ~= y, • •)` とすることにより誤差を吸収して動作するようになる。

4.11.2 ブール代数

	演算子	関数
論理積	<code><bool1> & <bool2></code>	<code>and(<bool1>, <bool2>)</code>
論理和	<code><bool1> % <bool2></code>	<code>or(<bool1>, <bool2>)</code>
否定	<code>!<bool></code>	<code>not(<bool>)</code>
排他的論理和	—	<code>xor(<bool1>, <bool2>)</code>

引数がブール表現でない場合は `___` を返す。

4.12 演算子一覧

括弧内は優先順位。

:	自身の定義にアクセスする
.	あらかじめ定義されたデータフィールドにアクセスする
°	度数法を弧度法に変換する
-	リスト内の要素にアクセスする
^	べき乗
*	乗法 (ベクトル, 行列も含む)
/	除法 (ベクトル, 行列をスカラーで割る場合も含む)
+	加法 (ベクトル, 行列も含む)
-	減法 (ベクトル, 行列も含む)
!	論理否定
==	相等
>	より大きい
<	より小さい
>=	以上
<=	以下
!=	等しくない
~=	ほぼ等しい
~<	ほぼ小さい
~>	ほぼ大きい
~>=	ほぼ以上である
~<=	ほぼ以下である
=:=	equals after evaluation
&	論理積
%	論理和
!=	等しくない
~!=	ほぼ等しくない
..	a から b までの範囲の数のリスト
++	リストの連結
--	リストの差
~~	2つのリストの共通要素
:>	リストの後方に要素を追加する
<:	リストの前方に要素を追加する
=	変数への代入

<code>:=</code>	関数の定義
<code>:=_</code>	定義の消去
<code>-></code>	修飾子の宣言
<code>,</code>	リストと関数の区切り
<code>;</code>	命令文の区切り

5 型の判定

次の情報関数は、式が特定の種類に属するかどうかを調べる。戻り値は bool 値。

<code>isinteger(<expr>)</code>	整数かどうかを調べる。
<code>isreal(<expr>)</code>	実数かどうかを調べる。なお、整数は実数。
<code>iscomplex(<expr>)</code>	複素数かどうかを調べる。なお、実数は複素数。
<code>iseven(<expr>)</code>	偶数かどうかを調べる。
<code>isodd(<expr>)</code>	奇数かどうかを調べる。
<code>islist(<expr>)</code>	リストかどうかを調べる。
<code>ismatrix(<expr>)</code>	行列かどうかを調べる。
<code>isnumbervector(<expr>)</code>	すべて数(実数または複素数)からなるベクトルかどうかを調べる。
<code>isnumbermatrix(<expr>)</code>	成分がすべて数(実数または複素数)の行列かどうかを調べる。
<code>isstring(<expr>)</code>	文字列かどうかを調べる。
<code>isgeometric(<expr>)</code>	幾何学要素かどうかを調べる。
<code>isselected(<expr>)</code>	選択された幾何学要素かどうかを調べる。
<code>ispoint(<expr>)</code>	幾何の点かどうかを調べる。
<code>isline(<expr>)</code>	幾何の直線かどうかを調べる。
<code>iscircle(<expr>)</code>	幾何の円かどうかを調べる。
<code>isconic(<expr>)</code>	が幾何の円錐曲線かどうかを調べる。
<code>ismass(<expr>)</code>	CindyLab の質点かどうかを調べる。
<code>issun(<expr>)</code>	CindyLab の恒星かどうかを調べる。
<code>isspring(<expr>)</code>	CindyLab のバネかどうかを調べる。
<code>isundefined(<expr>)</code>	未定義要素(____)を返すかどうかを調べる。

6 文字列関数

6.1 文字列の結合

<文字列> + <expr>

+ 演算子は、ひとつの文字列に expr をつけ加える。expr が文字列でない場合も文字列に変換する。文字列と expr は逆順でもよい。

【例】	式	結果
	"Cindy"+"Script"	"CindyScript"
	"4 たす 3 は "+(4+3)	"4 たす 3 は 7"
	""+(4+3)	"7"

6.2 文字列への変換

`text(<expr>)`

expr を評価し、結果を文字列に変換する。

`format(<数>, <整数>)`

第1引数の数の小数点以下を、第2引数で指定された桁まで、文字列として整形する。最高15桁まで可能。引数がリストの場合は、各要素に対して整形がなされる。`format()` は、出力としての整形を行なうだけで、整形された値は文字列として扱われる所以計算はできない。

【例】

式	結果
<code>text(12*3)</code>	"36"
<code>text([1,2]*[2,-1])</code>	"0"
<code>format(sqrt(2),4)</code>	"1.4142"
<code>format(pi,14)</code>	"3.14159265358979"
<code>format([sin(30°),cos(30°)],3)</code>	["0.5","0.866"]

6.3 文字列の長さ

`length(<文字列>)`

文字列の長さを返す。

6.4 文字列の抜き出し

`substring(<文字列>, <整数 1>, <整数 2>)`

<文字列> の <整数 1> 文字目の次から <整数 2> 文字目までの文字列を抜き出す。

【例】 `substring("abcdefg", 3, 6)` は "def" を返す。

6.5 文字列の検索

`indexof(<文字列 1>, <文字列 2>)`

文字列 <文字列 2> が <文字列 1> の中にあるかどうかを検索し、最初に見つかった位置を返す。なければ 0 を返す。

`indexof(<文字列 1>, <文字列 2>, <整数>)`

文字列 <文字列 2> が <文字列 1> の <整数> 文字目以降にあるかどうかを検索し、最初に見つかった位置を返す。なければ 0 を返す。

【例】	式	結果
	<code>indexof("CindyScript", "i")</code>	2
	<code>indexof("CindyScript", "y")</code>	5
	<code>indexof("CindyScript", "z")</code>	0
	<code>indexof("CindyScript", "i", 1)</code>	2
	<code>indexof("CindyScript", "i", 3)</code>	9
	<code>indexof("CindyScript", "i", 10)</code>	0

6.6 文字列の分解

`tokenize(<文字列>, <expr>)`

この関数は、引数 <文字列> の部分文字列のリストを作成する。2 番目の引数 <expr> は、文字列か、文字列のリストでなければならない。もし <expr> が文字列であれば、<文字列> からこの文字列を検索する。この文字列は <文字列> を分解するときの標識となる。

もし <expr> が文字列のリストであれば、このリストを再帰的に使って <文字列> の部分文字列のリストを生成する。

【例】	式	結果
	<code>tokenize("one:two--three:four", ":")</code>	<code>["one", "two--three", "four"]</code>

```

 tokenize("one:two--three:four","-")          ["one:two", "", "three:four"]
 tokenize("one:two--three:four","--")          ["one:two", "three:four"]
 tokenize("one:two--three:four",["-",":])      [[["one", "two"], [], ["three", "four"]]]
 tokenize("one:two--three:four",["--",":])      [[["one", "two"], ["three", "four"]]]

```

6.7 文字列の置換

`replace(<文字列 1>, <文字列 2>, <文字列 3>)`

<文字列 1> の中の <文字列 2> をすべて <文字列 3> で置き換える。L-System の構築に便利。`replace(<文字列>, <list>)`

<list> は <文字列> に対して、置き換えをする文字列の組合せからなるリスト。リストの置き換え規則は 1 つでもよく、変数に代入して使うこともできる。

【例】	式	結果
	<code>replace("one:two--three:four", "o", "XXX")</code>	"XXXne:twXXX--three:fXXXur"
	<code>replace("F", "F", "F+F")</code>	"F+F"
	<code>replace("F+F", "F", "F+F")</code>	"F+F+F+F"
	<code>replace("XYX", [[{"X": "one"}, {"Y": "two"}]])</code>	"onetwoone"
	<code>s="ABC"; replace(s, [{"A": "X"}])</code>	"XBC"
	<code>s="ABC"; t= [{"A": "X"}, {"B": "Y"}]; replace(s, t)</code>	"XYC"

6.8 文字列の解析

`parse(<文字列>)`

文字列の内容を解析して、その結果を評価する。この関数は、テキストフィールドから入力された文字列を処理するのに特に有効。

`guess(<数>)`

浮動小数点で表された数を、高い精度でその浮動小数点数を生み出す数式に変えようとする。

【例】	式	結果
	<code>parse("3+7")</code>	10
	<code>text="sin(x)+cos(x)"; f(x):=parse(text);</code>	<code>f(x)=sin(x)+cos(x)</code> と同じ
	<code>parse(Text0.text);</code>	Text0 に入力された文字を解析

<code>guess(8.125)</code>	"65/8"
<code>guess(0.774596669241483)</code>	"sqrt(3/5)"

6.9 文字列の比較と並べ替え

数と同様、文字列も順序付けができる。したがって、演算子 `>`, `<`, `>=`, `<=`, `==`, `!=` を使って比較ができる。これらの演算子についてはブール関数を参照のこと。

文字の順序としては、辞書順を使う。たとえば、次のようにする。

```
"a" < "abd" < "abe" < "b" < "blue" < "blunt" < "xxx"
```

リストの並べ替え: `sort(<list>)`, `sort(<list>, <expr>)`, `sort(<list>, <var>, <expr>)`

文字列を含むリストを並べ替える。文字列の順序は常に辞書順。文字列の長さで並べ替えるように、ユーザーが順序を定義することもできる。

【例】	式	結果
	<code>sort(["one", "two", "three", "four"])</code>	<code>["four", "one", "three", "two"]</code>
	<code>sort(["one", "two", "three", "four"], length(#))</code>	<code>["one", "two", "four", "three"]</code>

指標関数: `<文字列>_<整数>`

リストの項目にアクセスする挿入演算子 `_` は、文字列内の指定した位置にある文字にアクセスするのにも使える。この演算子で、文字を返したり、文字をセットしたりすることができる。

【例】	式	結果
	<code>"CindyScript"_5</code>	<code>"y"</code>
	<code>"CindyScript"_12</code>	<code>---</code>
	<code>a="CindyScript"; a_5="rella";</code>	<code>"CinderellaScript"</code>

7 リスト

7.1 リストの作成とアクセス

整数列の作成: `<整数 1>..<整数 2>`

`<整数 1>` から `<整数 2>`までの連続する整数のリストを作る。`<整数 1>` が`<整数 2>`より大きい場合は空のリストを返す。

【例】	式	結果
-----	---	----

4..9	[4,5,6,7,8,9]
-2..2	[-2,-1,0,1,2]

リストの要素へのアクセス

演算子では `<list>_<整数>`

関数では `take(<list>, <整数>)`

第2引数のインデックスは1から始まる整数。インデックスが0か、リストの要素数より大きいときは、`---`が返され、「Index out of range」という Warning メッセージが出る。

インデックスは計算式で与えることもできる。また、入れ子にしたリストの要素にもアクセスできる。インデックスを負の整数にすると、リストの末尾から逆順にアクセスする。

整数のリストを使って、まとめてアクセスすることもできる。結果はリストで返される。

【例】 式 結果

[2 ,5 ,7 ,3]_3	7
<code>take([2 ,5 ,7 ,3], 2)</code>	5
[2 ,5 ,7 ,3]_5	---
[[2,[4,5]],1]_1	[2,[4,5]]
[[2,[4,5]],1]_(7-5)	1
[[2,[4,5]],1]_1_2	[4,5]
[[2,[4,5]],1]_1_2_2	5
[[2,[4,5]],1]_1_2_2_2	---
[2 ,5 ,7 ,3]_(-1)	7
<code>take([2 ,5 ,7 ,3], (-3))</code>	5
[[2,6] ,5 ,7 ,3]_(-4)_(-1)	6
[2 ,5 ,7 ,3]_[2,3]	[5,7]
[2 ,5 ,7 ,3]_[-1,1,1]	[3,2,2]

リストが変数に代入されると、個々の要素は`_`演算子によってアクセスしたのち、それを設定することができる。たとえば、次のスクリプトの結果、`a`は`[[2,["B",5]], "A"]`になる。

```
a=[[2,[4,5]],1];
a_2="A";
a_1_2_1="B";
```

リストの長さ: `length(<list>)`

【例】 式 結果

<code>length([2, 5, 7, 3])</code>	4
<code>length([2, [5, 4, 5], 7, 3]_2)</code>	3
<code>length(1..1000)</code>	1000

内容のテスト: `contains(<list>, <expr>)`

`<list>` の中に`<expr>`があるかどうかで、`true` または `false` を返す。

Unicode の数学記号 \in , \notin を使うこともできる。

`println(4 \in [1,3,4,5])` の結果は `true`

`println(4 \notin [1,3,4,5])` の結果は `false`

7.2 リストの操作

リストの連結

演算子では `<リスト 1> ++ <リスト 2>` または `<リスト 1> ∪ <リスト 2>`
`∪` は Unicode の数学記号。

関数では `concat(<リスト 1>, <リスト 2>)`

【例】 式	結果
<code>concat(["a", "b"], ["c", "d"])</code>	<code>["a", "b", "c", "d"]</code>
<code>["a", "b"] ++ ["c", "d"]</code>	<code>["a", "b", "c", "d"]</code>
<code>["a", "b"] ∪ ["c", "d"]</code>	<code>["a", "b", "c", "d"]</code>

リストからの要素の削除

演算子では `<リスト 1> -- <リスト 2>`

関数では `remove(<リスト 1>, <リスト 2>)`

【例】 式	結果
<code>remove([1,3,4,5,1,5,6], [1,3,7])</code>	<code>[4,5,5,7]</code>
<code>[1,3,4,5,1,5,6] -- [1,3,7]</code>	<code>[4,5,5,7]</code>

リストの共通部分

演算子では `<リスト 1> ~~ <リスト 2>` または `<リスト 1> ∩ <リスト 2>`
`∩` は Unicode の数学記号。

関数では `common(<リスト 1>, <リスト 2>)`

【例】 式	結果
<code>common([1,3,4,5,1,5,6], [1,3,7])</code>	<code>[1,3]</code>

$[1,3,4,5,1,5,6] \sim [1,3,7]$	$[1,3]$
$[1,3,4,5,1,5,6] \cap [1,3,7]$	$[1,3]$

要素の後方追加

演算子では $<\text{list}> :> <\text{expr}>$
 関数では $\text{append}(<\text{list}>, <\text{expr}>)$

要素の前方付加

演算子では $<\text{expr}> <:> <\text{list}>$
 関数では $\text{prepend}(<\text{expr}>, <\text{list}>)$

【例】	式	結果
	$["a", "b", "c"] :> "d"$	$["a", "b", "c", "d"]$
	$\text{append}(["a", "b", "c"], "d")$	$["a", "b", "c", "d"]$
	$"d" <:> ["a", "b", "c"]$	$["d", "a", "b", "c"]$
	$\text{prepend}("d", ["a", "b", "c"])$	$["d", "a", "b", "c"]$

7.3 リストの要素の走査

全要素走査

$\text{forall}(<\text{list}>, <\text{expr}>)$ 実行変数は #
 $\text{forall}(<\text{list}>, <\text{var}>, <\text{expr}>)$ 実行変数を ver にする

第1引数の $<\text{list}>$ を走査し、それぞれの要素を $<\text{expr}>$ で評価する。

【例】

$a = ["this", "is", "a", "list"]$; のとき、 $\text{forall}(a, \text{println}(\#))$ とすれば、 $\text{repeat}()$ で繰り返しをしなくともリストのすべての要素がコンソールに表示される。

式の適用

$\text{apply}(<\text{list}>, <\text{expr}>)$ 実行変数は #
 $\text{apply}(<\text{list}>, <\text{var}>, <\text{expr}>)$ 実行変数を ver にする

操作 $<\text{expr}>$ をリストのすべての要素に適用し、その結果からなるリストを作成する。

【例】

式	結果
---	----

$\text{apply}([1, 2, 3, 4, 5], \# * 2)$	$[2, 4, 6, 8, 10]$
---	--------------------

```

apply([1, 2, 3, 4, 5], t, t+5)      [6, 7, 8, 9, 10]
apply(1..5, #[#,*2])                [[1, 2], [2, 4], [3, 6], [4, 8], [5, 10]]

```

リストの要素の選択

```

select(<list>,<boolexpr>)        実行変数は #
select(<list>,<ver><boolexpr>)  実行変数を ver にする

```

この関数は、条件 <boolexpr>を満たすすべての要素を選び出す。この条件は <bool> 値をとる。

【例】 式 結果

select(1..10, isodd(#))	[1, 3, 5, 7, 9]
select(0..10, #+# == #^2)	[0,2]

【例】 次のスクリプトは x の約数のリストを得る関数 `divisors` を定義し、100 以下の素数のリストを表示する。

```

divisors(x):=select(1..x,mod(x,#)==0);
primes(n):=select(1..n,length(divisors(#))==2);
println(primes(100))

```

7.4 要素の組み合わせ

ペアを作る: `pairs(<list>)`

リストに含まれるすべての要素から 2つずつを組み合わせたペアの部分リストを要素とするリストを作る。

チエーンを作る: `consecutive(<list>)`

引数 <list> の連続する 2つずつの要素のペアからなるリストを作る。

輪を作る: `cycle(<list>)`

チエーンに加え、最後の要素と最初の要素もつなげる。

3つの組合せを作る: `triples(<list>)`

リストに含まれるすべての要素から 3つずつを組み合わせた部分リストを要素とするリストを作る。

2つのリストの直積を作る: `directproduct(<リスト 1>,<リスト 2>)`

<リスト 1> の要素を第 1 成分に、<リスト 2> の要素を第 2 成分とした、すべてのペアからなるリストを作る。

【例】	式	結果
pairs([1, 2, 3, 4])		[[1, 2], [1, 3], [1, 4], [2, 3], [2, 4], [3, 4]]
consecutive([1, 2, 3, 4, 5])		[[1, 2], [2, 3], [3, 4], [4, 5]]
cycle([1, 2, 3, 4, 5])		[[1, 2], [2, 3], [3, 4], [4, 5], [5, 1]]
triples([1, 2, 3, 4])		[[[1, 2, 3], [1, 2, 4], [1, 3, 4], [2, 3, 4]]]
directproduct([1,2,3], ["A", "B"])		[[[1, "A"], [1, "B"]], [2, "A"], [2, "B"], [3, "A"], [3, "B"]]]

入れ子リストの平坦化: flatten(<list>)

リストが入れ子になっているときに、第 2 の入れ子を解除して平坦化したリストを返す。修飾子を使うと、再帰的な操作が可能。また、平坦化のレベルを制御できる。

修飾子 levels は、"all" ですべてを平坦化する。整数を与えるとその回数だけ再帰的に平坦化する。flatten(..., levels->1) は修飾子を付けない場合と同じ。

【例】次のリストを用意する。

```
list=[[1,2],[3,[4,5],[6,[7,8]]],6];
```

このリストに対して平坦化を行なった結果は次の通り。

式	結果
flatten(list)	[1,2,3,[4,5],[6,[7,8]],6]
flatten(list,levels->0)	[[1,2],[3,[4,5],[6,[7,8]]],6]
flatten(list,levels->1)	[1,2,3,[4,5],[6,[7,8]],6]
flatten(list,levels->2)	[1,2,3,4,5,6,[7,8],6]
flatten(list,levels->3)	[1,2,3,4,5,6,7,8,6]
flatten(list,levels->"all")	[1,2,3,4,5,6,7,8,6]

7.5 要素の整列

逆順にする: reverse(<list>)

<list> の要素を逆順にする。

並べ替え: sort(<list>)

次の順序で要素を並べる。

ブール値 | 数 | 文字列 | リスト

並べ替え: `sort(<list>, <expr>)` : 実行変数は #

並べ替え: `sort(<list>, <ver>, <expr>)` : 実行変数を ver に指定する

リストの各要素を `<expr>` の内容によって評価し、その結果に従って並べ替える。

リストの要素を单一化した集合 (set) を作る: `set(<list>)`

リストのすべての要素を分類し、同一のものを取り除いて並べ替える。すなわち、リストをオブジェクトの集合とみなして唯一の表現にする。

【例】 式

結果

<code>reverse([1, 2, 3, 4])</code>	[4, 3, 2, 1]
<code>sort([4.5, 1.3, 6.7, 0.2])</code>	[0.2, 1.3, 4.5, 6.7]
<code>sort(["one", "two", "three", "four", "five"])</code>	["five", "four", "one", "three", "two"]
<code>sort([-4.5, 1.3, -6.7, 0.2], abs(#))</code>	[0.2, 1.3, -4.5, -6.7]
<code>sort(["one", "two", "three", "four"], length(#))</code>	["one", "two", "four", "three"]
<code>set([3, 5, 2, 4, 3, 5, 7])</code>	[2, 3, 4, 5, 7]
<code>set([3, 5, 2]++[4, 5, 2])</code>	[2, 3, 4, 5]

7.6 要素の総和・積

総和関数

`sum(<list>)` 要素の総和を求める。

`sum(<list>, <expr>)` 各要素に `<expr>` を実行した結果の総和を求める。

実行変数は #

`sum(<list>, <var>, <expr>)` `sum(<list>, <expr>)` で実行変数を `<var>` とする。

積関数

`product(<list>)` 要素をすべて掛け合わせる。要素は数であることが前提。

`product(<list>, <expr>)` 各要素に `<expr>` を実行した結果をすべて掛け合わせる。

実行変数は #

`product(<list>, <var>, <expr>)` `product(<list>, <expr>)` で実行変数を `<var>` とする。

【例】 式

結果

<code>sum(1..10)</code>	55
<code>sum([4, 6, 2, 6])</code>	18
<code>sum([[3, 5], [2, 5], [5, 6]])</code>	[10, 16]
<code>sum(["h", "e", "l1", "o"])</code>	"hello"
<code>sum(1..100, #^2)</code>	338350

応用例

`average(x) := sum(x)/length(x)` で平均を計算する関数が定義できる。x はリスト。

`fac(x) := product(1..x)` で階乗を計算する関数が定義できる。x は自然数。

7.7 最大と最小

最大値関数

<code>max(<list>)</code>	要素の最大値を求める。
<code>max(<list>, <expr>)</code>	<code><expr></code> をリストのすべての要素に実行した結果の最大値 実行変数は#
<code>max(<list>, <var>, <expr>)</code>	<code>max(<list>, <expr>)</code> で実行変数 <code><var></code> にする。

最小値関数

<code>min(<list>)</code>	要素の最小値を求める。
<code>min(<list>, <expr>)</code>	<code><expr></code> をリストのすべての要素に実行した結果の最小値 実行変数は#
<code>min(<list>, <var>, <expr>)</code>	<code>min(<list>, <expr>)</code> で実行変数を <code><var></code> にする。

8 ベクトルと行列

ベクトルや行列はリストで表現される。幾何学要素の座標もリストで表現される。したがって、これらの間での計算は、リストとしての計算で行われる。

8.1 ベクトルと行列の定義

数のリストは”数ベクトル”と呼ばれる。あるリストが数ベクトルかどうかは `isnumbervector` 関数によって確かめられる。

リストの要素がまたリストであり、しかもそれらがすべて同じ長さであれば、そのようなリストは 行列と呼ばれる。あるリストが行列であるかどうかは、`ismatrix` 関数によって確かめられる。さらに、行列の個々の要素がすべて数であれば、この行列は数行列と呼ばれる。ある行列が数行列かどうかは `isnumbermatrix` 関数によって確かめられる。行列の要素は同じ長さのベクトルとも考えられ、このようなベクトルは行列の行ベクトルである。したがって、行列が n 個の、長さ m の行ベクトルからなるならば、 $n \times m$ 行列である。

8.2 和と積

リストが同じ形を持つときは、加法と減法ができる。これは、リストが同じ長さでいくつかの要素がやはり同じようなリストであれば、和・差に対応する要素は同じ形になるということである。

リストのかけ算は、数学的に意味があれば許される。次の表は、かけ算が許される場合についてまとめたものである。

要素1	要素2	結果	意味
数	数	数	通常のかけ算
数	長さ r のベクトル	長さ r のベクトル	ベクトルの実数倍
長さ r のベクトル	数	長さ r のベクトル	ベクトルの実数倍
長さ r のベクトル	長さ r のベクトル	数	ベクトルの内積
$n \times r$ 行列	長さ r のベクトル	長さ n のベクトル	行列 \times 列ベクトル
長さ n のベクトル	$n \times r$ 行列	長さ r のベクトル	行ベクトル \times 行列
$n \times r$ 行列	$r \times m$ 行列	$n \times m$ 行列	行列の積

8.3 ベクトルと行列の演算

行列の次数	<code>matrixrowcolumn(<matrix>)</code>
転置行列	行と列の数を、2つの数の要素からなるリストとして返す。 <code>transpose(<matrix>)</code> 転置行列を返す。
行列の行抽出	<code>row(<matrix>, <整数>)</code> <i><整数></i> 番目の行をベクトルとして返す。
行列の列抽出	<code>column(<matrix>, <整数>)</code> <i><整数></i> 番目の列をベクトルとして返す。
小行列	<code>submatrix(<matrix>, <整数 1>, <整数 2>)</code> 第 <i><整数 1></i> 列と 第 <i><整数 2></i> 行を削除した小行列を返す。
ベクトルから行列への変換	<code>rowmatrix(<vector>)</code> ベクトルを1行とする行列を返す。
ベクトルから行列への変換	<code>columnmatrix(<vector>)</code> ベクトルを1列とする行列を返す。
零ベクトルの生成	<code>zerovector(<整数>)</code> 長さが <i><整数></i> の零ベクトルを作る。
零行列の生成	<code>zeromatrix(<整数 1>, <整数 2>)</code>

<整数 1> 行 <整数 2> 列の零行列を作る。

【例】 式	結果
matrixrowcolumn([[1,2],[3,2],[1,3],[5,4]])	[4,2]
transpose([[1,2],[3,2],[1,3],[5,4]])	[[1,3,1,5],[2,2,3,4]]
transpose([[1],[3],[1],[5]])	[[1,3,1,5]]
transpose([[1,3,1,5]])	[[1],[3],[1],[5]]
row([[1,2],[3,2],[1,3],[5,4]],2)	[3,2]
column([[1,2],[3,2],[1,3],[5,4]],2)	[2,2,3,4]
submatrix([[1,2,4],[3,2,3],[1,3,6],[5,4,7]],2,3)	[[1,4],[3,3],[5,7]]
rowmatrix([1,2,3,4])	[[1,2,3,4]]
columnmatrix([1,2,3,4])	[[1],[2],[3],[4]]

8.4 線形代数の演算

正方行列の行列式	det(<matrix>)
ベクトルの大きさ	vec
2つのベクトルの距離	<ベクトル 1>,<ベクトル 2>
ベクトルの距離	dist(<ベクトル 1>,<ベクトル 2>)
エルミート内積	hermiteanproduct(<ベクトル 1>,<ベクトル 2>)
正方行列の逆行列	inverse(<matrix>)
余因子行列	adj(<matrix>)
固有値	eigenvalues(<matrix>)
固有ベクトル	eigenvectors(<matrix>)
1次方程式の解	linearsolve(<matrix>,<vector>) または linearsolve(<matrix>,<matrix>)

エルミート内積は内積 $\langle \text{ベクトル } 1 \rangle * \langle \text{ベクトル } 2 \rangle$ と似ているが、2番目のベクトルは掛けられる前に、共役複素数にされる。特に、hermiteanproduct(a,a) は常に非負。

次のコードは、内積とエルミート内積の違いを示す。

```
a=[2+3*i,1-i];
println(hermiteanproduct(a,a));
println(a*a);
結果は次の通り。
```

15

-5 + i*10

`inverse(<matrix>)` は、逆行列を持たない場合は、未定義のオブジェクトを返す。

`eigenvalues(<matrix>)` の結果は値のリストとして返される。n 次の正方行列からは n 個の固有値を返す。実数の行列であっても、固有値は複素数の範囲で求めるのが普通。

【例】

```
m1=[[1,1,0],[0,1,0],[0,0,.5]];
println(eigenvalues(m1));
m2=[[1,1,0],[-1,1,0],[0,0,.5]];
println(eigenvalues(m2));
結果は次の通り
[1,1,0.5]
[1 + i*1,1 - i*1,0.5]
```

`eigenvectors(<matrix>)` は、正方行列の固有ベクトルの基底を計算する。結果としてベクトルのリストを返す。このリストの順序は、`eigenvalues` 関数における固有ベクトルの順序と同じ。もし、行列が対角化可能でない場合は、この結果は意味がない。

`linearsolve(A,b)` は、方程式 $Ax=b$ の解 x を計算する。行列 A は正方行列で逆行列を持たなければならない。 b は n 次のベクトルか、n 行の行列。 A が逆行列を持たないか、次数が合わない場合は未定義値を返す。

【例】

```
m=[[1,1,0],[0,1,0],[0,1,1]];
x=linearsolve(m,[2,3,4]);
println(x);
println(m*x);
結果は次の通り。
[-1,3,1]
[2,3,4]
```

8.5 3 次元の凸多面体を作る

`convexhull3d(<ベクトルのリスト>)`

この関数は、3 次元ベクトルのリストを与えると凸多面体を作る。戻り値は 2 つのリスト

からなるペア。第1の要素は凸多面体の頂点リスト、第2の要素は面リスト。それぞれの面は第1要素の頂点によって与えられる。

【例】 次の点のリストは、立方体の頂点と中心を示すリストである。

```
[[1,1,1],[1,1,-1],[1,-1,1],[1,-1,-1],[-1,1,1],  
[-1,1,-1],[-1,-1,1],[-1,-1,-1],[0,0,0]]
```

このリストに convexhull3d 関数を適用すると、次の出力を得る。

```
[ [[1,1,1],[1,1,-1],[1,-1,1],[1,-1,-1],[-1,1,1],[-1,1,-1],[-1,-1,1],[-1,-1,-1]],  
[[6,5,1,2],[3,1,5,7],[3,4,2,1],[8,7,5,6],[8,6,2,4],[8,4,3,7]] ]
```

9 幾何学要素へのアクセス

9.1 要素にその名前でアクセスする

すべての幾何学要素には個々の名前(識別子)がある。CindyScriptにおいて、幾何学要素はあらかじめ定義された変数として扱われる。それぞれのパラメータは.(ドット)演算子によって読み書きできる。例えば、次のコードでは、点Aの大きさを20に設定する。

```
A.size=20
```

もし、点や線(の名前)が算術関数に含まれるときは、自動的にその位置を表すベクトルに変換される。点は2次元座標で表される [x,y] ベクトルに変換される。直線は、同様に同次座標で表される [x,y,z] ベクトルに変換される。しかし、座標を設定する場合は、ドット演算子を用いて明示的に設定されなければならない。幾何学要素が算術関数で使われないのであれば、依然として幾何学的要素として扱われます。この概念が微妙なので、少し例を挙げてはっきりさせておこう。

AとB,CがCinderellaの点であると仮定する。次のコードはAを線分BCの中点に設定する。

```
A.xy=(B+C)/2
```

この2点B,Cは算術関数に含まれるので、[x,y]ベクトルとして処理される。しかし、点Aの位置は.xyパラメータによって明示的に設定されなければならない。

次のコードでは3つの点の色をすべて緑色にする。

```
pts=[A,B,C];
```

```
forall(pts, p, p.color=[0,1,0]);
```

このコードでは、点の名前はリスト pts へのハンドルとしてそのまま扱われる。`forall` 関数で走査されたハンドルは一度変数 p に代入され、そこから色のパラメータにアクセスされる。

9.2 幾何学要素のリスト

Cinderella で作図した幾何学要素をリストアップするような場合、次のような要素のリストを返す関数が使える。

```
allmasses allsprings
```

すべての点のリスト	<code>allpoints()</code>
すべての直線のリスト	<code>alllines()</code>
すべての線分のリスト	<code>allsegments()</code>
すべての円のリスト	<code>allcircles()</code>
すべての円錐曲線のリスト	<code>allconics()</code>
すべての質点のリスト	<code>allmasses()</code>
すべてのバネのリスト	<code>allsprings()</code>
他の要素に付随する要素のリスト	<code>allsegments(<幾何要素>)</code>

【例】 次のスクリプトは、y 軸 の左右で点の色を変える。

```
pts=allpoints();
forall(pts,p,
  if(p.x<0, p.color=[1,1,0], p.color=[0,1,0]);
);
```

9.3 幾何学要素のパラメータ

ドット演算子によってアクセスできるパラメータ。以下の表において、パラメータの型は次の通り。

real : 実数
int : 整数
bool : true または false
string : 文字列
2-vector : 2 次元ベクトル

3-vector : 3 次元ベクトル

3x3-matrix : 3 行 3 列行列

すべての幾何学要素についてのパラメータ

名前	読み出し	書き込み	型	目的
color	可	可	3-vector	オブジェクトの色 (赤, 緑, 青)
colorhsb	可	可	3-vector	オブジェクトの色 (色相, 彩度, 輝度)
isshowing	可	可	bool	オブジェクトの表示/非表示
visible	可	可	bool	オブジェクトの表示/非表示
alpha	可	可	real	オブジェクトの透明度
labelled	可	可	bool	ラベルの表示/非表示
name	可	不可	string	オブジェクトの名称
caption	可	可	string	オブジェクトのラベル
trace	可	可	bool	軌跡 (足跡) の表示/非表示
tracelength	可	可	int	足跡の長さ
selected	可	可	bool	オブジェクトが選択されているかどうか

パラメータの書き込みは、時には自由要素に限られることがある。該当する項で "free" の語を使ってそれを表す。

それぞれの幾何学要素は個別の識別名を持つ。識別名は .name パラメータによってアクセスできる。たとえば、 A.name は文字列 "A" を返す。その名前は、画面上に表示される ラベル (caption) とは異なる場合がある。ラベルが設定されていなければ A.caption は空の文字列。また、ラベルはインスペクタで変更することができる。

isshowing と visible の違いは、 isshowing ではオブジェクトに依存するすべての要素に引き継がれるのに対し、 visible では従属するオブジェクトからは引き継がれない点。

直線のパラメータ

名前	読み出し	書き込み	型	目的
homog	可	free	3-vector	直線の同次座標
angle	可	free	real	直線の角度
slope	可	free	real	直線の傾き
size	可	可	int	直線の幅 (0 から 10))

点のパラメータ

名前	読み出し	書き込み	型	目的
x	可	free	real	点の x 座標
y	可	free	real	点の y 座標
xy	可	free	2-vector	点の xy 座標
coord	可	free	2-vector	点の xy 座標
homog	可	free	3-vector	点の同次座標
angle	可	free	real	円周上の点の角度 (円周上の点にのみ適用)
size	可	可	int	点の大きさ (0 から 40)
imagerot	可	可	real	点が画像で置き換えられている場合、回転角度

円と円錐曲線 (2次曲線)

名前	読み出し	書き込み	型	目的
center	可	free	real	円の中心
radius	可	free	real	円の半径
matrix	可	不可	real	円または二次曲線を記述する行列
size	可	可	int	線の幅 (0 から 10)

テキストのパラメータ

名前	読み出し	書き込み	型	目的
text	可	可	string	文字列の内容
pressed	可	可	boolean	文字列の状態。ボタンならば true
xy	可	可	2-vector	文字列の位置

アニメーションのパラメータ

名前	読み出し	書き込み	型	目的
run	可	可	bool	アニメーションの実行/非実行
speed	可	可	real	アニメーションの相対的な速さ

変換のパラメータ

名前	読み出し	書き込み	型	目的
matrix	可	不可	3x3 matrix	変換の同次行列
inverse	可	不可	3x3 matrix	逆変換の同次行列

CindyLab オブジェクトのパラメータ

すべての CindyLab 要素のパラメータ

名前	読み出し	書き込み	型	目的
simulate	可	可	bool	オブジェクトがシミュレーションに加わるかどうか

質点のパラメータ

名前	読み出し	書き込み	型	目的
mass	可	可	real	オブジェクトの質量
charge	可	可	int	オブジェクトの電荷
friction	可	可	real	オブジェクトの摩擦
radius	可	可	real	質点を球とみなすときの半径
posx	可	可	real	質点の x 座標
posy	可	可	real	質点の y 座標
pos	可	可	2-vector	質点の位置ベクトル
vx	可	可	real	速度の x 成分
vy	可	可	real	速度の y 成分
v	可	可	2-vector	速度ベクトル
fx	可	不可	real	粒子にかかる力の x 成分
可	不可	real	粒子にかかる力の y 成分	
f	可	不可	2-vector	粒子にかかる力のベクトル
kinetic	可	不可	real	粒子の運動エネルギー
ke	可	不可	real	粒子の運動エネルギー

質点間に力を定義したい場合がある。そのときは Integeration Tick スロットにコードを書く。質点の位置が内部的に通常の幾何学要素と異なるタイムスケールで動くときは pos, posx および posy によってその位置にアクセスする必要がある。

バネとクーロン力のパラメータ

名前	読み出し	書き込み	型	目的
l	可	不可	real	バネの現在長
lrest	可	不可	real	バネの自然長
ldiff	可	不可	real	現在の長さと自然長との差
strength	可	可	real	バネ定数
f	可	不可	real	バネにかかる力
amplitude	可	可	real	振幅
speed	可	可	real	運動の速さ
phase	可	可	real	運動の段階（位相）(0.0 ~ 1.0)
potential	可	不可	real	バネのポテンシャルエネルギー
pe	可	不可	real	バネのポテンシャルエネルギー

速度のパラメータ

名前	読み出し	書き込み	型	目的
factor	可	可	real	図で表されている速度と実際の速度との掛け率

重力のパラメータ

名前	読み出し	書き込み	型	目的
strength	可	可	real	重力場の強さ
potential	可	不可	real	重力場におけるすべての質点のポテンシャルエネルギー
pe	可	不可	real	重力場におけるすべての質点のポテンシャルエネルギー

恒星のパラメータ

名前	読み出し	書き込み	型	目的
mass	可	可	real	恒星の質量
potential	可	不可	real	恒星の場におけるすべての質点のポテンシャルエネルギー
pe	可	不可	real	恒星の場におけるすべての質点のポテンシャルエネルギー

電磁場のパラメータ

名前	読み出し	書き込み	型	目的
strength	可	可	real	電磁場の強さ
friction	可	可	real	磁力のかかるフィールドでの摩擦

床と反射壁のパラメータ

名前	読み出し	書き込み	型	目的
xdamp	可	可	real	x 方向の吸収率
ydamp	可	可	real	y 方向の吸収率

環境のパラメータ

環境は内蔵関数の `simulation()` によって次の項目がアクセスできる。

名前	読み出し	書き込み	型	目的
gravity	可	可	real	全体にかかる重力
friction	可	可	real	全体にかかる摩擦力
kinetic	可	不可	real	全体的な運動エネルギー
ke	可	不可	real	全体的な運動エネルギー
otential	可	不可	real	全体的なポテンシャルエネルギー
pe	可	不可	real	全体的なポテンシャルエネルギー

9.4 インスペクタの要素

`inspect 属性のリスト: inspect(<幾何要素>)`

【例】 点 A が描かれているとき, `inspect(A)` とすれば, 次のようなリストが返される。

[name,definition,color,visibility,drawtrace,tracelength,traceskip,tracedim,render,isvisible,
text.fontfamily,pinning,incidences,labeled,textsize,textbold;textitalics,ptsizes,pointborder,
printname,point.image,point.image.rotation,freetpt.pos]

`属性の読み出し: inspect(<幾何要素>,<文字列 1>)`

【例】 `inspect(A,"text.fontfamily")` とすれば

SansSerif

が返される。

`属性の設定: inspect(<幾何要素>,<文字列 1>,<文字列 2>)`

変更可能な属性の値を設定する。

【例】 A のフォント Serif に設定する。

`inspect(A,"text.fontfamily","Serif")`

`ユーザー属性の設定: attribute(<幾何要素>,<文字列 1>,<文字列 2>)`

<幾何要素> の属性を <文字列 1> と <文字列 2>で設定する。

ユーザー属性の取得: attribute(<幾何要素>, <文字列>)

<文字列> で定義される幾何要素 <幾何要素> のユーザー属性を取得する。

この 2 つのユーザー属性に関する関数は Visage で主に使用する。

9.5 要素の作成と消去

createpoint 自由点を作る: createpoint(<文字列>, <位置>)

この関数は <文字列> を識別名とする点を <位置> に作る。すでに同じ名前の点がある場合には新しく作らず、単に指定した位置に移動する。

create 幾何要素を作る: create(<リスト 1>, <文字列>, <リスト 2>)

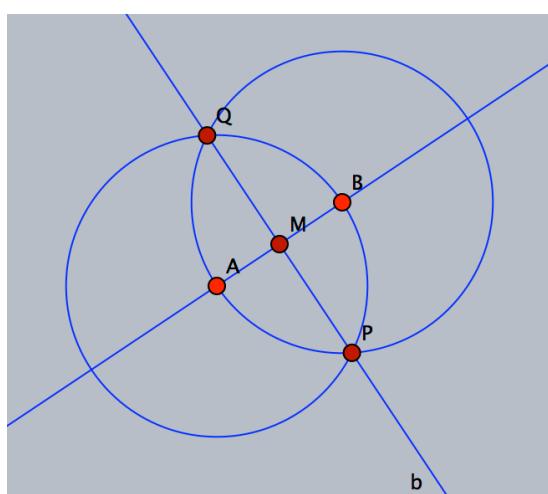
この関数は、任意の幾何要素を作る。アルゴリズムがいくつか微妙な点を引き起こすので、特別な用途に限られる。

第 1 引数の <リスト 1> は生成する要素の名前のリスト。第 2 引数の <文字列> は幾何学的なアルゴリズムの内部的な名前。第 3 引数の <リスト 2> は、定義のために必要なパラメータのリスト。

【例】

```
create(["A"], "FreePoint", [[1, 1, 1]]);
create(["B"], "FreePoint", [[4, 3, 1]]);
create(["a"], "Join", [A, B]);
create(["X"], "CircleMP", [A, B]);
create(["Y"], "CircleMP", [B, A]);
create(["P", "Q"], "IntersectionCircleCircle", [X, Y]);
create(["b"], "Join", [P, Q]);
create(["M"], "Meet", [a, b]);
```

この一連のコードで、次の図のように幾何要素を作る。6 番目の命令で円の交点を作るとき、2 つの点の名前のリストがなければならない。



removeelement 幾何要素の消去: removeelement(<幾何要素>)

幾何要素を、それに関連するものも含めて消去する。

inputs 幾何要素を構成する要素: inputs(<幾何要素>)

幾何要素を定義するために必要な要素のリストを返す。

algorithm 幾何要素の作図手順: algorithm(<幾何要素>)

この関数は<幾何要素>の作図手順を文字列で返す。

【例】次のスクリプトはすべての要素に対してその作図手順を示す。

```
els=allelements();  
data=apply(els,([#[.name],algorithm(#),inputs(#)]));  
たとえば、垂直二等分線の作図手順が次のように出力される。
```

```

[[["A"], "FreePoint", [[4,4,-4]]],  

[["B"], "FreePoint", [[4,-3,1]]],  

[["a"], "Join", [A,B]],  

[["C"], "Mid", [A,B]],  

[["b"], "Orthogonal", [a,C]]  

]

```

9.6 幾何要素の操作

自由要素を動かす: `moveto(<幾何要素>,<位置>)`

この関数では<幾何要素>に幾何の自由要素を、<位置>にこれを動かしたい位置を記述する。この関数によって自由要素の動きをシミュレートする。

<幾何要素>が自由点であれば、<位置>は2つの数のリスト [x,y]（ユークリッド座標）か

3つの数のリスト [x,y,z] (同次座標)。

<幾何要素>が自由直線であれば、<位置>は3つの数のリスト [a,b,c] でなければならず、直線 $ax + by + c = 0$ に設定される。

【例】 次のコードは、幾何の要素をどのように動かせるかをまとめたものである。要素のデータに直接アクセスして動かすものも含む。

```
// A は自由点
moveto(A,[1,4]);           // 点 A をユークリッド座標 (1,4) に置く
A.xy=[1,4];                // 点 A をユークリッド座標 (1,4) に置く
A.x=5;                     // A の x 座標を 5 にする。y 座標は変化しない
A.y=3;                     // A の y 座標を 3 にする。x 座標は変化しない
moveto(A,[2,3,2]);         // A を同次座標 [2,3,2] に置く
A.homog=[2,3,2];           // A を同次座標 [2,3,2] に置く

// a は自由曲線
moveto(a,[2,3,4]);         // a を同次座標 [2,3,4] に置く
a.moveto=[2,3,4];           // a を同次座標 [2,3,4] に置く

// b は傾きつき直線
a.slope=1;                  // 傾きを 1 にする

// C は半径が自由な円
C.radius=1;                 // 半径を 1 にする
```

動いた要素: mover()

この関数は、マウスによって今動かされた要素が何であるか(そのハンドル)を返す。

マウスのあるところの要素: elementsatmouse()

この関数は、現在マウスカーソルの近くにある要素のリストを返す。

【例】 次のスクリプトを Mouse Move スロットに置いて実行すると、マウスカーソルの近くにある要素が消える。マウスカーソルが遠ざかれればまた現れる。

```
apply(allelements(),#.alpha=1);
apply(elementsatmouse(),#.alpha=0);
repaint();
```

オブジェクトへのインシデント:incidences(<幾何要素>)

この関数は、幾何要素とインシデントなすべての要素のリストを返す。

軌跡上の点: `locusdata(<軌跡>)`

この関数は、`<軌跡>`によって与えられる軌跡上の点のxy座標のリストを返す。

その名前の要素を取り出す: `element(<文字列>)`

この関数は、`<文字列>`の名前の幾何要素を返す。

【例】`element` 関数は、要素名が無効であるか、あるいはすでに使われているような場合に使われる。たとえば、`i`という名前の直線の色にアクセスしようとすると、`i`は虚数単位の名前として予約されているので、`i.color=[1,1,1]`と書くことはできない。このような場合、次のように書く。

```
element("i").color=[1,1,1]
```

再描画をする: `repaint()`

この関数は描画している図を強制的に再描画する。これは、スクリプトが図の更新を必要とするときに実行される。この関数は `draw` あるいは `move` スロットにおいてはいけない。

再描画その2: `repaint(<real>)`

`repaint` 命令を、パラメータで与えられたミリ秒だけ遅らせて実行する。

10 図形の描画

10.1 修飾子

描画関数では、色や大きさなどをオプションとして与えることができる。これを「修飾子」と呼んでいる。次のような修飾子があり、関数によって使えるものは異なる。修飾子はコンマで区切って、`draw([1,1],size->8,color->[1,1,0])` のように指定する。

修飾子	型	効果
pointsize	<real>	点の大きさを設定する。
linesize	<real>	線の幅を設定する
size	<real>	点の大きさと線の幅を設定する
pointcolor	[R,G,B]	RGB 値による点の色を設定する
linecolor	[R,G,B]	RGB 値による線の色を設定する
color	[R,G,B]	RGB 値による点と線の色を設定する
alpha	<real>	透明度を設定する
noborder	<bool>	true なら点の境界線の表示をしない
border	<bool>	true なら点の境界線の表示をする
dashtype	<real>	破線パターンを 0 から 4 で指定する
dashing	<real>	破線を描く。与えた数により細かさが変わる
dashpattern	<list>	個別の破線パターンを指定する

10.2 色や大きさの設定

色の設定

色は、赤/緑/青 (R/G/B) の 3 つの実数のリストで指定する。それぞれの値は 0 から 1 までで、0 が最も暗く 1 が最も明るくなる。ある実数が色のコードと解釈される場合、0 以下の数は 0 に、1 以上の数は 1 に置き換えられる。

初期状態の色は、

```
pointcolor(<colorvec>),
linecolor(<colorvec>),
textcolor(<colorvec>)
```

の 3 つの関数で設定できる。さらに、`color(<colorvec>)` 関数はすべてのオブジェクトの色を同時に設定する。

個々に色を指定するときは、修飾子 `color->[R,G,B]` を用いる。

【例】

```
pointcolor([0,1,1]);           これ以降、点の色を水色（シアン）にする。
draw([2,2],color->[1,1,0]);   座標 [2,2] に黄色で点を打つ。
drawtext([2,2],"注意",color->[1,0,0])座標 [2,2] に赤で「注意」と表示する。
```

透明度の設定

透明度は 0 から 1 までの実数で指定する。ここで、0 は完全に透明で、1 は完全に不透

明になる。この範囲外の値は 0 か 1 のどちらかに設定される。初期状態の透明度は、関数 `alpha(<数>)` によりすべての要素に設定される。個々に設定する場合は、拡張子 `alpha->数` を用いる。

【例】

```
alpha(0.5);          これ以降、透明度を 0.5 にする。  
draw([2,2],alpha->0.8);    座標 [2,2] に透明度 0.8 で点を打つ。
```

大きさの設定

初期状態の大きさは `pointsize(<数>)`, `linesize(<数>)`, `textsize(<数>)` の 3 つの関数で設定できる。大きさは、実数で指定する。点と線の大きさは 1 から 20 までの整数で、ピクセルのサイズ。

個々に大きさを指定するときは、修飾子 `size->数` を用いる。

【例】

```
textsize(24);          これ以降、文字の大きさを 24 にする。  
draw([2,2],size->8);    座標 [2,2] に大きさ 8 の点を打つ。  
drawtext([2,2],"文字",size->36);    座標 [2,2] に大きさ 36 で文字を表示する。
```

10.3 色の関数

次の関数は、指定した輝度の色コード (`[R,G,B]`) を返す。たとえば、`red(0.8)` は `[0.8,0,0]`

赤色 : `red(<数>)`

緑色 : `green(<数>)`

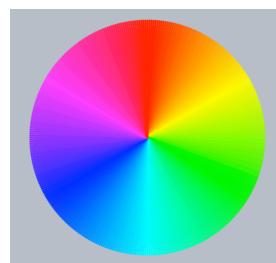
青色 : `blue(<数>)`

灰色 : `gray(<数>)`

虹色 : `hue(<数>)`

`hue` は、すべての色のうちの一つを表す色コードを作る。<数> は 0 から 1 までの数で色相環の範囲を表す。1 より大きな値の場合には、色相環を何周かして色を決める。次のコードと実行結果はその例。

```
n=360;  
ind=(1..n)/n;  
linesize(2);  
forall(ind,  
color(hue(#));  
draw((0,0),(sin(#*2*pi),cos(#*2*pi)));  
);
```



10.4 描画関数

点を描く: `draw(<expr>)`

`draw([x,y])` は点 $[x,y]$ に点を打つ。`draw([x,y,z])` は同次座標が $[x,y,z]$ である点を打つ。

$[x,y,z]$ が点の同次座標であるとき、対応する点は、xy-座標が $[x/z, y/z]$ の点。 $z=0$ の点は”無限遠点”。この無限遠点は、Cinderella の通常のユークリッド表示では見えないが、球面表示や局所的に射影基底が設定されている場合は表示される。

線分・直線を描く: `draw(<expr>,<expr>)`

引数に2点を与えると、線分を描く。この2点はリストにしてもよく、座標はユークリッド座標でも、同次座標でもよい。

作図した幾何点の識別子を点の座標として用いることもできる。

直線を描くには同次座標を用いる。3つの実数 $[a,b,c]$ で表される直線は、方程式 $a * x + b * y + c = 0$ で表されるものと考えることができる。ユークリッド座標 $[x,y]$ の点は、この方程式を満たすときに限りこの直線上にある。同次座標 $[x,y,z]$ の点は、方程式 $a * x + b * y + c * z = 0$ を満たすときに限りこの直線上にある。

3つの実数 $[a,b,c]$ が点を表すのか直線を表すのかを CindyScript が知るために、直線が引かれた場合には内部フラグが立つようになっている。たとえば、`join(A,B)` 命令は、2点 A と B を通る直線を計算して、直線であるという内部フラグを立てる。その結果、直線が引かれる。関数 `line()` を使うことによって、直線のフラグを強制的に立てることもできる。

【例】2点 A(1,1) と B(4,5) を作図してあるとする。次の描画命令はいずれも2点 $[1,1]$ と $[4,5]$ を結ぶ線分を描く。

```
draw([1,1],[4,5]);
draw([[1,1],[4,5]]);
draw(A,B);
draw([A,B]);
```

【例】次の2つの描画命令はいずれも直線を描く。

```
draw(line([1,1,0.5]));
draw(join([1,2],[2,-1]));
```

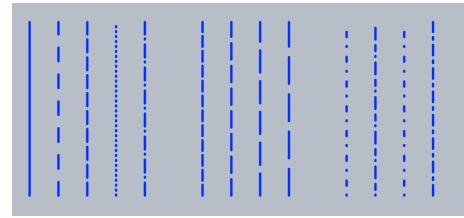
【例】線分（直線）では、修飾子によって幾つかの線種が扱える。次はその例。

```
draw([0,0],[0,6],dashType->0);
draw([1,0],[1,6],dashType->1);
```

```

draw([2,0],[2,6],dashtype->2);
draw([3,0],[3,6],dashtype->3);
draw([4,0],[4,6],dashtype->4);
draw([6,0],[6,6],dashing->4);
draw([7,0],[7,6],dashing->6);
draw([8,0],[8,6],dashing->8);
draw([9,0],[9,6],dashing->10);
draw([11,0],[11,6],dashpattern->[0,4,2,4]);
draw([12,0],[12,6],dashpattern->[0,2,2,2,4,2]);
draw([13,0],[13,6],dashpattern->[0,4,2,4]);
draw([14,0],[14,6],dashpattern->[4,2,1,2]);

```



リストのオブジェクトを描画する: drawall(<list>)

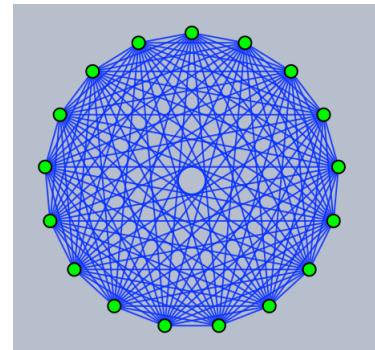
draw() 関数で描画する点・線分をリストにして描画することができる。これにより、repeat() で繰り返しを行うことなく、まとめて描画できる。

【例】次のコードは、正 17 角形の辺と対角線をすべて描く。2 行目で単位円上に点を打つ関数を定義する。3 行目は、変数 steps を単位円を 17 等分する角度のリストとする。これらに対応する点が、変数 pts に 4 行目でリストとして代入され、5 行目で、変数 segs を pts のすべてのペアのリストとする。最後の 2 つの行で 2 つのリストにある点と線分を描く。

```

n=17;
f(x):=[sin(x),cos(x)];
steps=2*pi*(1..n)/n;
pts=apply(steps,f(#));
segs=pairs(pts);
drawall(segs,alpha->0.9);
drawall(pts,size->4);

```



点をつなげる: connect(<list>)

この関数は、点のリストを受け取って、それらを順に結んだ線分を描く。

多角形を描く: drawpoly(<list>)

この関数は、点のリストを受け取り、点を順に結んだ多角形を描く。点のリストは閉じている必要はなく、自動的に閉じて描く。たとえば、draw([A,B,C]) とすると、A,B,C,A をこの順に結んで多角形とする。使える修飾子は color と alpha。size は使えないで、線の太さを指定したい場合は、閉じたリストにして connect() を用いる。

中を塗った多角形を描く: `fillpoly(<list>)`

この関数は、点のリストを受け取って中が塗られた多角形を描く。使える修飾子は `color` と `alpha`。

円を描く: `drawcircle(<点>, <半径>)`

`<点>` を中心とする半径 `<半径>` の円を描く。点の座標はユークリッド座標か同次座標で与える。

円盤を描く: `fillcircle(<点>, <半径>)`

`<点>` を中心とする、半径 `<半径>` の、内部を塗りつぶした円を描く。点の座標はユークリッド座標か同次座標で与える。使える修飾子は `color` と `alpha`。

11 関数プロット

CindyScript には、数学的な関数をプロットするいくつかの関数がある。簡単な関数のプロットの他に、極値、零点、変曲点に関する情報も表示できる。

11.1 実数関数のグラフを描く

関数のプロット: `plot(<expr>)`

`plot` 関数は、関数をプロットするのに使う。`<expr>` に関数式を与える。この式は実行変数 # を含まなくてはならず、実数の入力値 # に対して、ひとつの実数か 2 次元ベクトルを計算します。一つの実数の場合には、`plot` 関数は単純な関数のグラフを描く。ベクトルの場合には、媒介変数形のグラフを描く。座標系は、幾何の表示画面の座標系に依存する。# の代わりに他の実行変数を使うことができる。自由変数が一つだけあればそれを実行変数とみなす。複数の自由変数がある場合は、x, y, t, z の順に探して実行変数とみなす。

関数を定義しておいてそのグラフを描くこともできる。

<code>plot(sin(#));</code>	
<code>plot(sin(x));</code>	<code>plot(sin(#))</code> と同じ結果になる
<code>f(x):=2*sin(x); plot(f(#));</code>	<code>f(x)</code> で定義したグラフを描く
<code>plot([2*cos(t), 3*sin(t)]);</code>	媒介変数表示で橙円を描く

媒介変数表示の場合、変数の値の初期値は 0 から 100。この範囲は、修飾子 `start`, `stop` によって指定できる。

`plot()` 系の関数では次の修飾子が使える。曲線の外観に関するもの、定義域や解像度に関するもの、特別な点のマーク、線種がある。

修飾子	型	効果
外観の表現		
color	[R,G,B]	色を設定
size	<real>	線の幅を設定
alpha	<real>	不透明度を設定
connect	<real>	関数値の飛んでいるところをつなぐ
繰り返しの制御		
start	<real>	描き始めの値を設定
stop	<real>	描き終わりの値を設定
steps	<real>	プロットする点の間隔 (媒介変数のみ)
pxlres	<real>	曲線をプロットする解像度 (real 関数のみ)
特別な点		
extrema	<bool>	すべての極値をマークする
extrema	[R,G,B]	すべての極値を指定した色でマークする
minima	<bool>	すべての極小値をマークする
minima	[R,G,B]	すべての極小値を指定した色でマークする
maxima	<bool>	すべての極大値をマークする
maxima	[R,G,B]	すべての極大値を指定した色でマークする
zeros	<bool>	すべての零点をマークする
zeros	[R,G,B]	すべての零点を指定した色でマークする
inflections	<bool>	すべての変曲点をマークする
inflections	[R,G,B]	すべての変曲点を指定した色でマークする
破線		
dashing	<real>	破線パターンの幅 (初期値 5)
dashtype	<int>	特定の破線タイプ (値は 0 から 4 まで)
dashpattern	<list>	個々の破線パターンを指定する

定義域

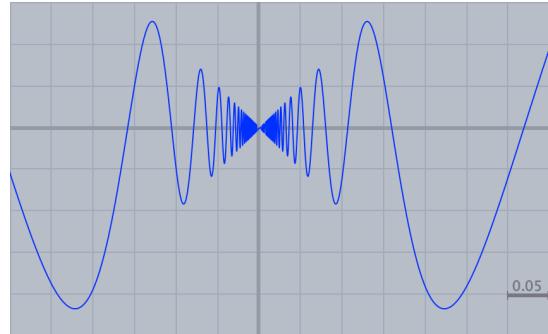
定義域の左端と右端を決める, start , stop は幾何点で制御することもできる。

点 A,B を x 軸上に置き, 次のスクリプトでグラフを表示すれば, 点をドラッグしてインタラクティブに定義域を変えることができる。

```
plot(f(#), start->A.x, stop->B.x)
```

プロットの解像度

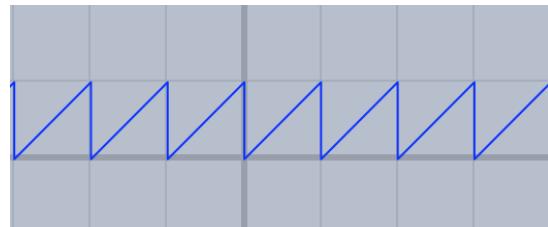
グラフのプロットは、定義域を分割して点の座標を計算して線分でつないでいる。媒介変数表示の場合は、この間隔を `steps` で変更できる。分割数が大きくなると解像度が高くなる。また、`pxlres` 修飾子はこの解像度の指定だが、`plot()` 関数は解像度を自動的に調整し、特異点の近くでは解像度を高めるようになっている。次の例は `plot(sin(1/#)*#)` の結果を示すが、原点付近での解像度が高くなっている。



`pxlres` 修飾子は、`colorplot()` 関数で有用になる。

不連続点を結ぶ

通常は、関数値が飛んでいるところが発見されても、初期状態ではその点は結ばれない。ノコギリ波のようにそれらの点を結ぶときには `connect->true` という修飾子を使う。次の例は、`plot(x-floor(x), connect->true)` で、 $f(x) = x - |x|$ のグラフの不連続点を結んだもの。



特異点の表示

修飾子を用いて、零点、極大値、極小値、変曲点を表示することができる。たとえば、`extrema->true` とすると、すべての極値を点で表示する。点の色を指定する場合は `extrema->[R,G,B]` とする。

線種

線分の線種と同様に、関数プロットでも `dashtype` などの修飾子で線種を指定できる。

関数のプロット その2: `plot(<expr>, <var>)`

実行変数を指定してプロットする。実行変数として有効な変数を2つ以上含む式のとき、どの変数についての関数かを指定できる。

11.2 動的な色と透明度

関数プロットの色と透明度は、関数値によって変化させることができる。次の図は

```
plot(sin(x),color->hue(x/(2*pi)),size->3)
```

の実行例。ここで、周期的な虹色を作るために `hue` を使っている。



積分イメージのプロット: `fillplot(<expr>)`

積分の説明などで関数のグラフと x 軸の間をハイライトさせたいときに用いる。`plot()` 関数と同様な引数（実行変数など）を使う。最も簡単な使い方では、 x 軸をはさんだ部分をハイライトする。この関数ではグラフそのものは描かない。グラフも描きたい場合は `plot()` 関数と併用するか修飾子 `graph` を用いる。

【例】

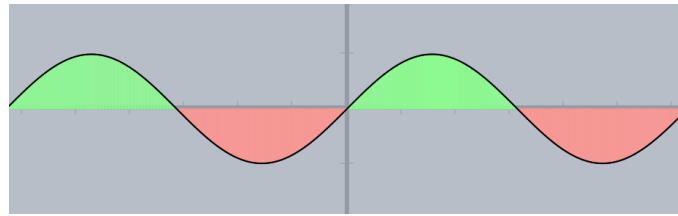
```
f(x):=1/(x^2+1)*sin(4*x);
fillplot(f(x));
plot(f(x)); または fillplot(f(x),graph->true);
```

注意： `fillplot(...)` における特異点の処理は、`plot()` 関数ほど精密ではない。そのため、`plot()` での修飾子とは使い分ける必要がある。

この関数では通常の `color, alpha` のほかに、次の修飾子が使える。

修飾子	型	効果
pluscolor	[R,G,B]	関数値が正のときの色
minuscolor	[R,G,B]	関数値が負のときの色
start	<real>	定義域の左端
stop	<real>	定義域の右端
graph	<bool>	関数のグラフも描く
graph	[R,G,B]	特定の色で関数のグラフを描く
size	<reall>	関数のグラフの太さ

【例】`fillplot(sin(x),graph->true,pluscolor->(0.5,1,0.5),minuscolor->(1,0.5,0.5));`



積分イメージのプロット: `fillplot(<expr>, <expr>)`

この関数は `fillplot(...)` と同様だが、2つの関数のグラフではさまれた部分をハイライトする。

【例】 $\sin x$ と $\cos x$ のグラフで挟まれた部分を交互に色を変えて塗る。

```
fillplot(sin(x),cos(x),graph->true,pluscolor->(0.5,1,0.5),minuscolor->(1,0.5,0.5))
```

11.3 カラープロット

カラープロットは平面上の各点に、関数によって色を割り当てることができる。

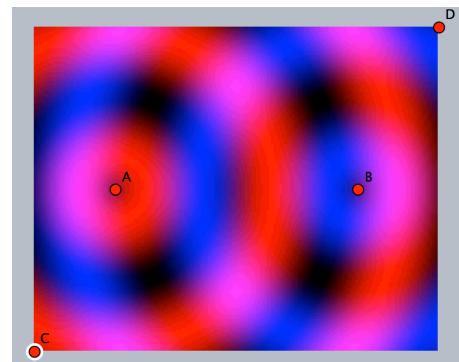
場の色表現: `colorplot(<expr>, <vector>, <vector>)`

この関数は、関数によって長方形の各々の点に、カラーコードを割り当てる。ここでも `<expr>` の関数の中で実行変数 # が使われるが、この変数は平面上の点である。 `<expr>` の値は、実数か（その場合は灰色）、3つの実数のベクトル (RGB)。2番目と3番目の引数は、描画領域の左下と右上の位置を決定する。

【例】 次のコードと2点 A, B は右の図を描く。1行目で2点間の距離の正弦を計算する関数を定義する。（値は区間 [0,1] に入るよう縮小・シフトしている）

`colorplot` 関数の第1の引数は、実行変数 # から計算される色のベクトル。赤は A からの同心円状の波、緑は 0、青は B からの同心円状の波。最後に、C と D は描画領域の長方形の2つの頂点。

```
f(A,B):=((1+sin(2*dist(A,B)))/2);
colorplot((f(A,#),0,f(B,#)),C,D);
```



実行変数

通常は # を colorplot の実行変数として使う。しかし、別の変数を用いることもできる。どんな変数が実行変数として使えるかは、次の順序で考える。

- ・<expr> の中で 1つだけの自由変数が使われている場合、その変数が実行変数として解釈される。2次元ベクトルの場合もある。
- ・<expr> が # を含む場合、# が実行変数とされる。
- ・<expr> が x と y を自由変数として含む場合、ベクトル (x,y) として扱われる。
- ・1つの自由変数が明確に割り当てられない場合、ベクトル (x,y) が用いられる。
- ・上記のいずれでもない場合、点 p と複素数 z が実行変数としてチェックされる。

修飾子

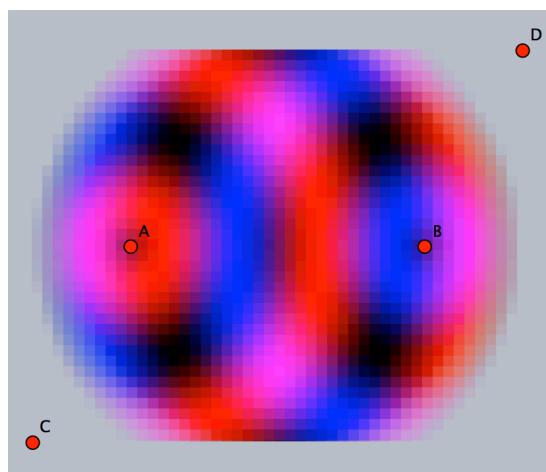
colorplot 関数では 3つの修飾子が扱える。pxlres は、色をプロットする四角形のピクセルの大きさを決定する整数。上の図は pxlres->2 の場合で、これが初期値。pxlres を 2 から 1 にすると精密な図ができる。しかしながら、四角形の中のピクセルごとに、colorplot が計算されるので、pxlres が 1 次関数的に減らされても計算は 2 次関数的に増加する。したがって、効率を上げるためにこの修飾子を調整するとよい。

解像度を動的に変えることもできる。そのため、解像度を段階的に変える startres がある。たとえば、startres->16 と pxlres->1 を使うとインタラクティブな動きをする。十分な時間があれば最もよい解像度になるように再計算をする。

さらに colorplot 関数では、透明度をコントロールする alpha 修飾子が使える。この修飾子は、実行変数をパラメータに用いることもできる。

【例】

```
f(A,B):=((1+sin(2*dist(A,B)))/2);  
colorplot((f(A,#),0,f(B,#)),C,D, p xlres->4, alpha->-abs(#)+5);
```



CindyJS での colorplot

CindeyJS での `colorplot` では、矩形領域を示す点を略すと画面全体が対象となる。また、`pxlres` は 1 である。Cinderella では矩形領域を示す点を略すことはできない。

11.4 ベクトル場

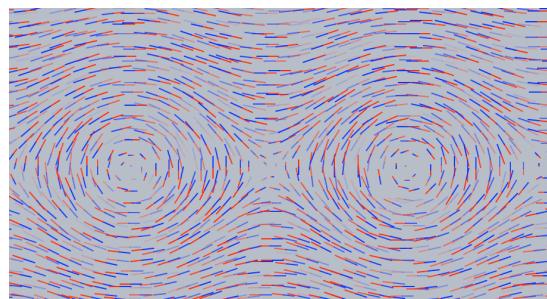
ベクトル場の描画: `drawfield(<expr>)`

`drawfield` 関数は、式 `<expr>` で与えられるベクトル場を描画するのに用いる。この式は実行変数 `#` を含んだ 2 次元ベクトルである必要がある。その結果も 2 次元ベクトル。式を `drawfield` 関数に適用すると、対応するベクトル場を描く。ベクトル場は動的である。たとえば、画面上でマウスをドラッグすると絵が動く。したがって、`drawfield` 関数を Draw スロットでなく "timer tick" スロットに入れるのが有効である。すると、アニメーションコントローラが表示され、アニメーションを走らせると、ベクトル場が自動的に動く。実行変数の考え方は `colorplot()` と同様。特に、自由変数 `x` と `y` を 2 次元ベクトル `(x,y)` として使うことができる。

【例】

関数 $f(x,y) = (y, \sin(x))$ によって定義されるベクトル場を考える。CindyScript での定義のしかたと `drawfield` 関数の使い方は次の通り。

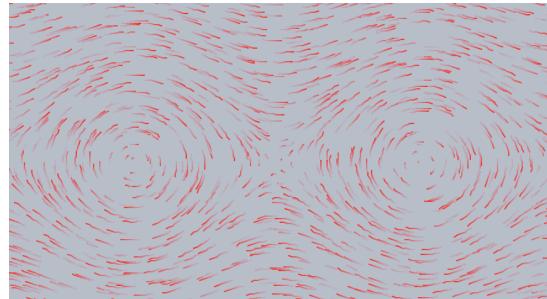
```
f(v):=[v.y,sin(v.x)];  
drawfield(f(#));
```



`drawfield((y,sin(x)))` でも同じ図が得られる。

図を描くのに、針のような図形が画面上に描かれる。これらの針はベクトル場によって配置される。アニメーションの間、針はベクトル場によって動く。計算時間はかかるが、針のかわりに小さな蛇のような図形でベクトル場の印象を高めることもできる。これは、`stream`修飾子で指定する。

```
f(v):=[v.y,sin(v.x)];  
drawfield(f(#),stream->true);
```



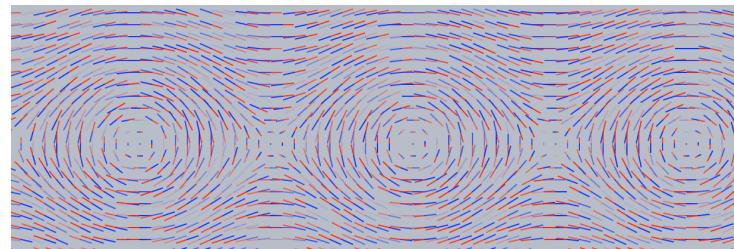
修飾子

drawfield 関数は、ベクトル場の生成プロセスを制御する多くの修飾子を備えている。それらの理解を助けるために、どのように図が描かれるかをもう少し詳しく説明する。

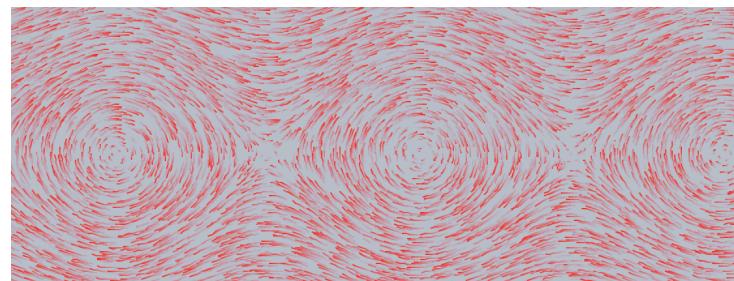
図は、ベクトル場の影響を受けているいくつかのテストオブジェクトの動きを示すことで生成される。初期状態では、テストオブジェクトは針状。それらはまず格子状に置かれる。これは通常いくつかの副産物を発生し、格子点の周りにある半径でランダムにゆがめられる。アニメーションの間、針は力の場の方向に動く。針の長さは場の強さを表す。

修飾子	型	効果
resolution	<int>	ピクセル単位での最初の格子の大きさ。
jitter	<int>	テストオブジェクトのゆがみ
needlesize	<real>	針の最大長
factor	<real>	場の強さの拡大要素
stream	<bool>	針状か流体か
move	<real>	オブジェクトを動かす速さ
color	[R,G,B]	流体もしくは針の始めの方の色
color2	[R,G,B]	針のあとの方の色

次の図は最初の格子の様子。`move->0` と `jitter->0` を指定した。垂直または水平方向から不自然な配置になっている。



次の図は、`resolution->5` と `stream->true` で描写したもの。

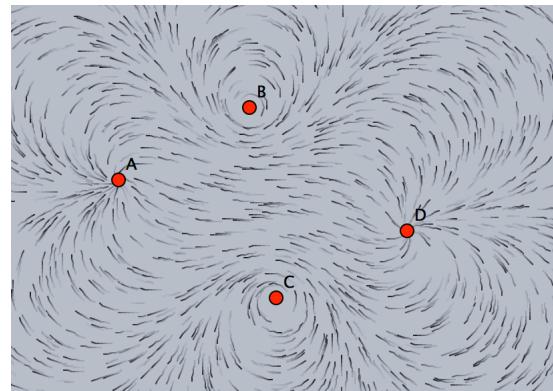


複素ベクトル場の描画: `drawfieldcomplex(<expr>)`

この関数は `drawfield` とよく似ている。しかし、これは入力値を 1 次元の複素数とする関数である。実部と虚部はベクトル場の x 成分と y 成分とみなされる。それ以外は、`drawfield` と同様。

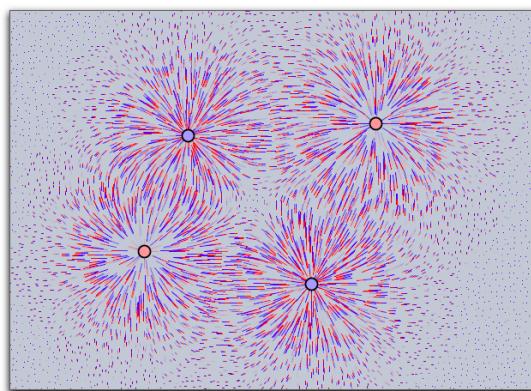
【例】 次は、描かれた 4 点によって零値が決まる複素多項式を使った例。

```
f(x):=(x-complex(A))*(x-complex(B))*(x-complex(C))*(x-complex(D));
drawfieldcomplex(f(#),stream->true,resolution->5,color->(0,0,0))
```



力の場の表示: `drawforces()`

この関数も `drawfield` とよく似ているが、今度は CindyLab における物理シミュレーションに関連している。引数は必要でなく、画面のいろいろな地点にある潜在的な試験電荷による力を示す。試験電荷は、質量 1、電荷 1、半径 1。しかし、他のいかなる粒子もこれと相互作用は起こさない。ときどき、力の場を説明するのに、`factor` 修飾子を使うことが必要になる。次の図は 4 つの荷電粒子間の相互作用を示す。



点による力の場の描画: `drawforces(<mass>)`

固定された質量を持つ粒子に関して力の場を描く他の関数がある。粒子自身は力の計算に入らない。こうして、特定の粒子に作用する力を表現することができる。

11.5 グリッド

グリッドは平面から平面への写像を視覚化するのに用いることができる。グリッドで写像による変形が示される。

正方グリッドのマッピング: `mapgrid(<expr>)`

この関数は、正方グリッドをとって `|expr|` による関数で変形する。初期状態ではもとのグリッドはその平面における単位正方形とみなされる。この正方形の境界は修飾子によって変更できる。また `complex->true` 修飾子によって、複素写像を視覚化できる。

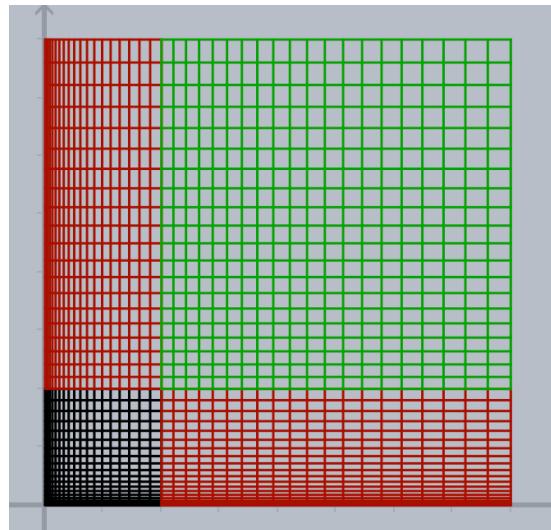
修飾子

修飾子	型	効果
color	[R,G,B]	色
alpha	<real>	透明度
size	<real>	グリッド線のサイズを指定する。
xrange	[<real>,<real>]	もとの正方形の x の範囲
yrange	[<real>,<real>]	もとの正方形の y の範囲
resolution	<int>	両方向のグリッド線の数
resolutionx	<int>	x-方向のグリッド線の数
resolutiony	<int>	y-方向のグリッド線の数
step	<int>	両方向のステップ数
stepx	<int>	x-方向のステップ数
stepy	<int>	y-方向のステップ数
complex	<bool>	複素関数を使う

【例】次のコードは `mapgrid` 関数の使い方の例。修飾子 `xrange` と `yrange` による効果も示す。x 座標と y 座標をそれぞれ 2乗する 2次元の関数による効果を示す。

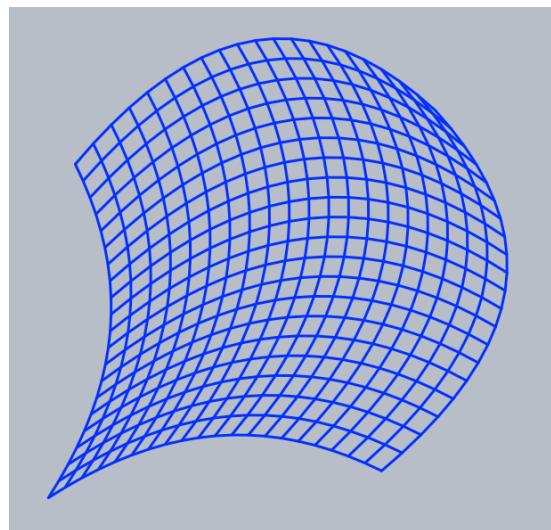
```
f(v):=(v_1^2,v_2^2);
linesize(1.5);
mapgrid(f(v),color->[0,0,0]);
mapgrid(f(v),xrange->[1,2],color->[0.6,0,0]);
mapgrid(f(v),yrange->[1,2],color->[0.6,0,0]);
```

```
mapgrid(f(v),xrange->[1,2],yrange->[1,2],color->[0,0.6,0]);
```



次の例は、グリッドの線がまっすぐであったり平行であったりしない例。

```
f(v):=(v_1*sin(v_2),v_2*sin(v_1));
linesize(1.5);
mapgrid(f(v),xrange->[1,2],yrange->[1,2]);
```



次の例は、複素関数による mapgrid の使い方の例。

```
mapgrid(z^2,complex->true);
```

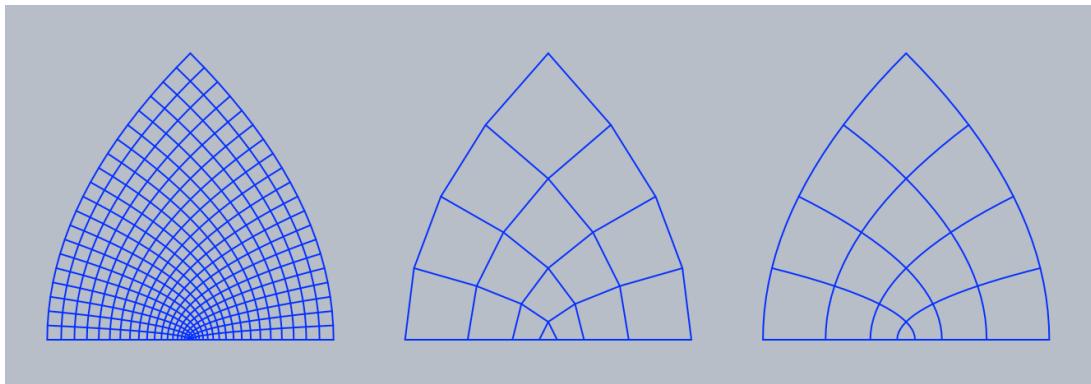
resolution 修飾子によって生成される格子の密度を指定できる。

```
mapgrid(z^2,complex->true,resolution->4);
```

初期状態では `mapgrid` 関数では直接格子点をつなぐ。これは数学的に正しくない図を作るかもしれない。`step` 修飾子で格子点の間に段階を追加する。

```
mapgrid(z^2,complex->true,resolution->4,step->5);
```

以上の 3 つのコードの結果は左から順に次の図。



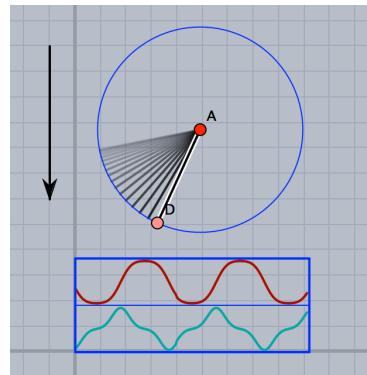
11.6 オシログラフ

物理的大きさのグラフを描く: `drawcurves(<vector>, <list>)`

物理シミュレーションにおいて、大きさが時間とともにどのように変化するかをグラフで示したい場合がよくある。このために、`drawcurves` 関数が作られた。ここで、`<vector>` は表示領域の左下の角の位置を表す 2 次元ベクトルで、`<list>` は観察する値のリスト。アニメーションが始まると値が更新され、対応する曲線が描かれる。

【例】次の図は `drawcurves` 関数の簡単な使い方を示している。CindyLab で振り子を作っている。D は自由質点。重力を加えて振り子が動く。次のコードは、動く点 D の x 座標と x 方向の速度を曲線で表す。

```
drawcurves([0,0], [D.x,D.vx])
```



修飾子

修飾子	型	効果
width	<real>	プロット範囲の幅 (ピクセル)
height	<real>	それぞれの曲線の高さ (ピクセル)
border	<bool>	表の枠を表示する
back	<bool>	背景を表示する
back	[R,G,B]	背景を指定した色で塗りつぶす
backalpha	<real>	背景の不透明度
colors	[< 色 1>,< 色 2>,< 色 3>,...]	曲線ごとに色を指定する
texts	[< 文字列 1>,< 文字列 2>,< 文字列 3>,...]	曲線ごとに表題を付ける
showrange	<bool>	曲線の最大と最小を表示する
range	< 文字列 >	欄外参照
range	[< 文字列 1>,< 文字列 2>,< 文字列 3>,...]	それぞれの曲線の peek/auto

range の文字列は

"peek" : 絶対値が最大のときの大きさ

"auto" : 現在表示されている部分の大きさ

次のスクリプトは、修飾子の使い方を示す。連結振動子のエネルギー状態を表す。

```

linecolor((1,1,1));
textcolor((0,0.8,0));
drawcurves((-7,-3),
[A.x,B.x,A.ke,B.ke,a.pe+b.pe+c.pe],
height->50,
color-(1,1,1),
back-(0,0,0),
backalpha->1,
range->"peek",width->400,
colors-[

[1,0.5,0.5],
[0.5,1,0.5],
[1,0.5,0.5],
[0.5,1,0.5],
[0.5,0.5,1]],

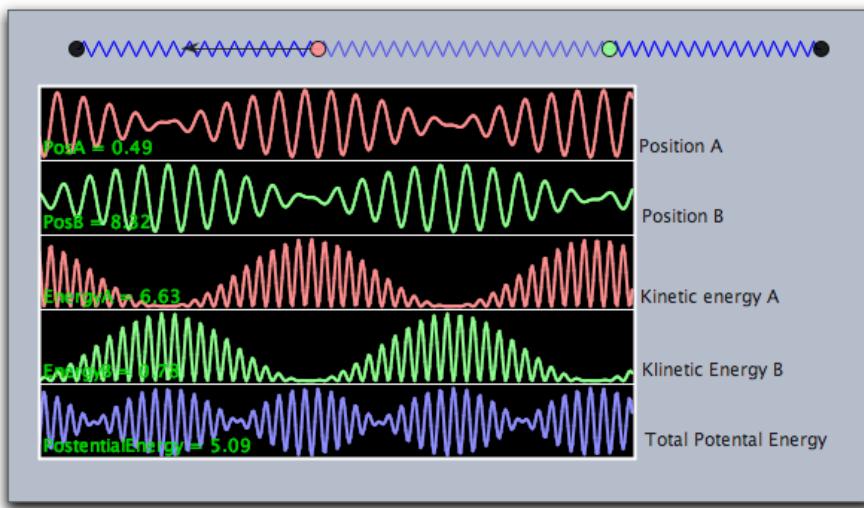
```

```

texts->[
    "PosA = "+ A.x,
    "PosB = "+B.x,
    "EnergyA = "+A.ke,
    "EnergyB = "+B.ke,
    "PotentialEnergy = "+(a.pe+b.pe+c.pe)
]
);

```

対応する図は次の通り。



11.7 文字と表

文字の表示: `drawtext(<vector>, <expr>)`

`drawtext(<vector>, <文字列>)` 関数は `<文字列>` で与えられる文字列を `<vector>` の指定する位置に表示する。この位置ベクトルは、ユークリッド xy 座標か、同次座標で与える。文字列中に改行コードがあると、文字通りそこで改行される。改行コードは不可視だが、スクリプト中で単に Enter キーで改行するときに入る。これにより、複数行からなる文章を書くことができる。

【例】`drawtext([0,0], "Hello World")` で、文字列 "Hello World" を [0, 0] の位置から書き始める。

修飾子

修飾子	型	効果
size	<real>	文字の大きさの設定
color	[R,G,B]	文字の色を設定
alpha	<real>	文字の透明度を設定
xoffset	<real>	文字と基準点の x 方向のピクセル単位の隔たりを設定
yoffset	<real>	文字と基準点の y 方向のピクセル単位の隔たりを設定
offset	<real>,<real>	文字と基準点の xy ピクセル単位の隔たりを設定
align	< 文字列 >	left,right,mid で文字の配置を設定
bold	<bool>	太字
italics	<bool>	斜体
family	< 文字列 >	フォントの指定

利用できるフォント: `fontfamilies()`

この関数では、そのコンピュータで利用できるすべてのフォントのリストを作成する。次のコードはそれらのフォントの名前をその字体で表示する。

```

families=fontfamilies();
t=0;
while(t<length(families),t=t+1;
    drawtext((mod(t,5)*7,round(t/5)),families\_t,family->families\_t);
);

```

Unicode: `unicode(<文字列>)`

Cinderella では Unicode が扱える。Unicode で文字を表示するには `unicode(<文字列>)` 関数を使う。引数は、16 進数の Unicode 列。戻り値はそれに対応する文字。修飾子によって 16 進数ではない形式の引数にもできる。

修飾子 `base -> <整数>` を用いると、Unicode のコード表現を変更できる。

【例】`unicode("0041")` と `unicode(65,base->10)` はどちらも文字 "A"を返す。

文字列の表示可/不可: `candisplay(<文字列>)`

この関数は、与えられた文字列が、現在選択されているフォントで表示できるかどうを調べる。戻り値はブール値。

表を描く: `drawtable(<位置>,<list>)`

リストで表を作る。表の外観のために、修飾子も使える。

【例】次のコードで表ができる。

```
x=1..10;  
table=apply(x,(#,#^2,#^3,#^4));  
drawtable([0,0],table);
```

修飾子

修飾子	型	効果
width	<int>	セル幅をピクセル単位で指定する
height	<int>	セルの高さをピクセル単位で指定する
flip		行と列を入れ替える
border	<bool>	背景と線を描くかどうかの指定
size	<実数>	文字の大きさを設定
color	[R,G,B]	文字の色を設定
alpha	<real>	文字の透明度を設定
offset	[<real>,<real>]	文字と基準点の xy ピクセル単位の隔たりを設定
align	<文字列>	セル内での横位置を center,right,left の文字で指定
back	<bool>	表の背景を描くかどうかを指定
back	[R,G,B]	表の背景を指定した RGB の色で描くかどうかを指定
backalpha	<real>	表の背景の透明度

【例】次のコードは drawtable 関数を用いた例。2番目の表は見出し。

```
x=1..9;  
tab=apply(x,(#,#^2,#^3,#^4));  
tab1=("x","x^2","x^3","x^4");  
linecolor([0,0,.8]);  
drawtable([0,0],tab,  
         width->50,  
         back->[1,0,0],  
         backalpha->0.1,  
         align->"right",  
         size->12
```

x	x^2	x^3	x^4
1	1	1	1
2	4	8	16
3	9	27	81
4	16	64	256
5	25	125	625
6	36	216	1296
7	49	343	2401
8	64	512	4096
9	81	729	6561

```

);
linecolor([0,0,0]);
drawtable([0,7.1],tab1,flip->true,
width->50,
height->33,
back->[0,0,1],
backalpha->0.4,
align->"mid",
size->16,
color->[1,1,1]
);

```

12 TeX 記法 Cinderella TeX

現在の Cinderella ではおよそ 95% の Tex 記法が使える。ここでは、いくつかの例と、標準的な TeX との違いについて記述する。TeX 記法の詳細については TeX の解説書などを参照されたい。また、例については実行画面を示さないものもあるが、適宜実行してみてもらいたい。

TeX 記法は CindyScript の `drawtext()` で表示する文章と、幾何学要素のラベル、「文字列を追加する」ボタンによる文字列に使える。特別メニューの文字列入力による文章には使えない。

TeX 記法の宣言

通常の文章（文字列）の中で、ドル記号ではさんだ部分が TeX 記法によるものと解釈される。

【例】 `drawtext([0,2],"$f(x)=x^3-3x^2-3x+1$");`

サブスクリプトとスーパースクリプト

サブスクリプト（上付き文字）とスーパースクリプト（添え字）は、それぞれ `_` と `^` 記号を用いる。複雑なサブ・スーパースクリプトを用いたい場合は、波括弧でくくるが、通常の TeX スクリプトと異なり、数だけからなるものであれば、括弧はいらない。（つけてもよい）

【例】

```

drawtext([0,0], "$A_1$");
drawtext([2,0], "$A_{123}$");

```

```

drawtext([4,0], "$A_1^{12}$");
drawtext([6,0], "$A_{1_2}^{\frac{1}{2}}$");
drawtext([8,0], "$A_{1_2}^{\sqrt{x^2+y^2}}$");

```

$$A_1 \quad A_{123} \quad A_1^{12} \quad A_{1_2}^{\frac{1}{2}} \quad A_{1_2}^{\sqrt{x^2+y^2}}$$

和, 積分などの記号

TeX ではバックスラッシュ \ を用いて制御コードを書く。 \sum と \int では, 自動的に和の記号や積分記号, 上付き文字と下付き文字などを使って式を表現する。

和, 積分記号の結果は添字のつき方が通常の TeX と Cinderella TeX では異なる。

```

drawtext([0,0], "$\sum_{i=1}^n (i^2+1)$");
drawtext([3,0], "$\sqrt{x^2+y^2}$");
drawtext([6,0], "$\int_a^b f(x)dx$");

```

を実行すると, 通常の TeX では

$$\sum_{i=1}^n (i^2 + 1) \sqrt{x^2 + y^2} \int_a^b f(x) dx$$

となるが, Cinderella TeX では次のようになる。

$$\sum_{i=1}^n (i^2 + 1) \sqrt{x^2 + y^2} \int_a^b f(x) dx$$

積分記号で, 上端下端の位置を調整したい場合は, \int のあとにスペースをいれ, _ と ^ を2重にするとそれらしくなる。

```
drawtext((6,0), "$\int_{a}^{b} f(x)dx$");
```

$$\sum_{i=1}^n (i^2 + 1) \sqrt{x^2 + y^2} \int_a^b f(x) dx$$

括弧

Cinderella TeX では 4 種類の括弧を式の中で使う。

丸い括弧 : (....)

角括弧 : [....]

波括弧 : \{...\}

縦線 : |...|

異なる大きさの括弧を使う場合は, \big, \Big, \bigg, \Bigg に続いて括弧を書

く。`\left` と `\right` を使うと、ちょうどよい大きさの括弧でくくることができる。`\left` と `\right` はちゃんと入れ子状になっている必要がある。

【例】

```
drawtext([0,0],"$\\Bigg( \\bigg( \\Big( \\big( (\\ldots) \\big) \\Big) \\bigg) \\Bigg)$");
drawtext([5,0],"$\\Bigg[ \\bigg[ \\Big[ \\big[ [\\ldots] \\big] \\Big] \\bigg] \\Bigg]$");
drawtext([10,0],"$\\Bigg\\{\\bigg\\{\\Big\\{\\big\\{\\{\\ldots\\}\\big\\}\\Big\\}\\bigg\\}\\Bigg\\}$");
drawtext([15,0],"$\\Bigg| \\bigg| \\Big| \\big| |\\ldots| \\big| \\Big| \\bigg| \\Bigg|$");
drawtext([20,0],"$\\left[\\sum_{i=1}^n \\left(\\sqrt{\\sin(i)}\\right)^2\\right]^2$");
```

分数

分数のような特殊な形の式は、Cinderella TeX は次の式をサポートしている。

`\frac`, `\over`, `\choose`, `\binom`
`\cfrac`, `\dfrac` はサポートしていない。

【例】

```
drawtext([0,0],"${1+n^2}\over{1-n^2}$");
drawtext([3,0],"${2}\choose{3}$");
drawtext([6,0],"${a+b}\over{x^2}$");
drawtext([9,0],"${a+b}\choose{x^2}$");
```

$$\frac{1+n^2}{1-n^2} \quad \binom{2}{3} \quad \frac{a+b}{x^2} \quad \binom{a+b}{x^2}$$

空白

空白と改行は、よく式を区切るために使われ、式のレイアウトのためにはあまり使われない。`\,`, `\;`, `\quad`, `\qquad`, `\!` を使えば、空白の大きさを変えられる。現在のフォントの”m”の大きさが空白の単位になる。

`\qquad`: 2.0 単位の空白
`\quad`: 1.0 単位の空白
`\;`: 5/18 単位の空白
`\,`: 3/18 単位の空白
`\!`: -5/18 単位の空白 (マイナスなので前の文字と重なる)

【例】 `drawtext([0,0],"$A\!A A \;,A\;A\quad A \qquad A\$")`

上線と下線

Cinderella TeX は式の上または下に矢印などの線を引くいくつかの方法をサポートしている。

\overline, \underline, \overleftarrow, \overrightarrow, \vec, \hat, \tilde
これらの記号のあとに、線を引きたい部分を波括弧でくくる。

【例】

```
drawtext([0,0],"$\\overline{A}\\;\\cap\\;\\overline{B}\\;=\\;\\overline{A\\;\\cup\\; B}$");  
drawtext([6,0],"$|\\overrightarrow{(x,y)}|\\;=\\; \\sqrt{x^2+y^2}$");  
drawtext([13,0],"$\\tilde{X}+\\hat{Y}\\;=\\;\\underline{X\\oplus Y}$");
```

$$\overline{A} \cap \overline{B} = \overline{A \cup B} \quad |(\overrightarrow{x,y})| = \sqrt{x^2+y^2} \quad \tilde{X} + \hat{Y} = \underline{X \oplus Y}$$

色

Cinderella TeX は色名を使って \color{...} で式に色をつけることができる。現在用意されている色の名前は

white, black, red, green, blue, darkred, darkgreen, darkblue, magenta, yellow, cyan, orange

色記号とそれに続いて書かれた文全体を波括弧でくくると、それらが指示した色で表示される。

【例】

```
drawtext([0,0],size->20,color->(0,0,0),  
"Sum formula: $  
\\sum_{\\color{darkgreen}i=1}^{\\color{darkgreen}n} {\\color{darkred}i^2}\\quad =\\quad \\color{blue}{2\\cdot n^3+4\\cdot n^2+n\\over 6}\\}$"  
);
```

$$\text{Sum formula: } \sum_{i=1}^n i^2 = \frac{2 \cdot n^3 + 4 \cdot n^2 + n}{6}$$

通常の文

TeX では半角スペースは無視される。半角スペースの入った文を表示する場合は、\mbox

を使う。

【例】

```
drawtext([0,0],"$\\sum_{\\mbox{All i not equal to j}}(i^2+j^2)$");
```

Unicode

特定の記号が TeX で表現できないことがある。その場合、Cinderella TeX では Unicode を使う方法がある。`\unicode{...}` か `\unicodex{...}` という記号を用いて TeX のなかで使う。第 1 の式では、Unicode を 10 進数で指定する。2 番目の式では 16 進数で指定する。次の例では、まず `sum(...)` 関数を用いて Unicode の列を作り、それを TeX で表示している。

```
chess=sum(0..11,i,"\\;\\unicode{+(9812+i)+}"");  
drawtext((0,0),size->30,"$"+chess+"$");
```



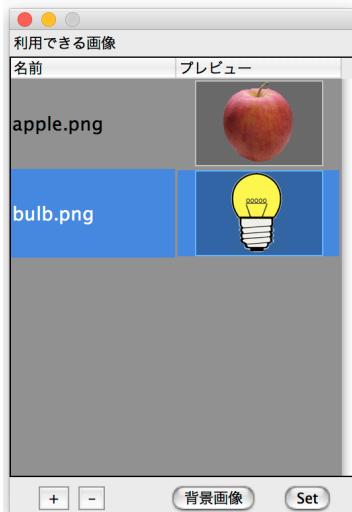
このほか、行列の表記、特殊記号、数学記号の表記については、通常の TeX と同様なので、TeX の説明書などを参照されたい。

13 画像の操作とレンダリング

PNG または JPG 形式の画像ファイルを読み込み、幾何要素の点や直線を利用して表示することができる。画像は、移動、回転、拡大・縮小、射影変換ができる。

13.1 メディアブラウザ

Cindyscript では、メディアブラウザを用いて画像を管理する。ファイルメニューの「メディアブラウザ」を選ぶとポップアップウィンドウが出て、+ボタンを押すと画像ファイルを読み込むことができてリストに追加できる。-ボタンを押せばリストから削除される。読み込んだ画像は、内部での名前とプレビューが表示される。画像の名前の初期値は読み込んだファイルの名前だが、ダブルクリックして名前を変更することができる。これは内部的な名前で、CindyScript ではこの名前を使う。



なお、以下の画面例では、Cindyscript のマニュアルのものをそのまま使用している。これらの絵の著作権は Terzio Verlag external にある。

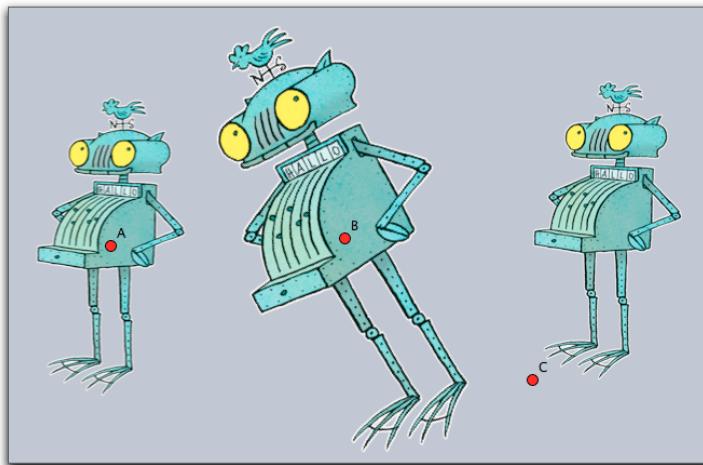
13.2 画像の表示と変換

画像の表示: `drawimage(<位置>, <画像名>)`

この関数は、メディアブラウザから画像を取り出して、第1引数で示した位置に表示する。位置は、座標もしくは幾何要素の点の識別名。初期状態では指定した座標が画像の中心になるが、これは修飾子によって変更することができる。また、画像の拡大縮小と配置も変えることができる。

【例】

```
drawimage(A, "myimage");
drawimage(B, "myimage", scale->1.5, angle->30° );
drawimage(C, "myimage", ref->"lb");
```

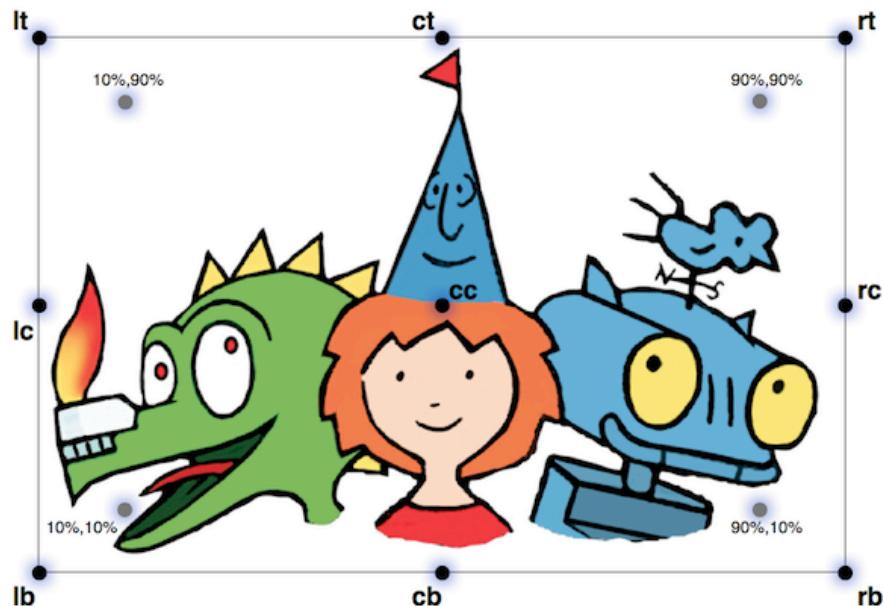


参照点

指定した点に対して画像をどこに置くかを、何通りか指定できる。このとき、この点を「参照点」という。修飾子 `ref`, `refx`, `refy` を用いて参照点の位置を指定する。参照情報は3つの異なる方法で与えられる。

- ・ もとの画像の画素数への絶対参照。たとえば、元の画像の画素数が 400×800 のとき、修飾子 `ref->[100, 200]` により、左下 4 分の 1 のところが参照点になる。
- ・ 比率で参照。この場合、百分率で位置を示す。たとえば、`ref->["25%", "25%"]` とすれば左下 4 分の 1 のところの参照点を置く。
- ・ 2 つの文字で記号的に。ここで x 方向を l, c, r で、順に 左, 中央, 右, y 方向を letters b, c, t で、下, 中央, 上とする。すると左下は `ref->"lb"` または `ref->["l", "b"]` で表される。

位置情報は2つの方向に分けて指定することもできる。たとえば、`refx->"l"`, `refx->100` あるいは `refx->"10%"` のように。次の図はいくつかの参照点を示す。



修飾子

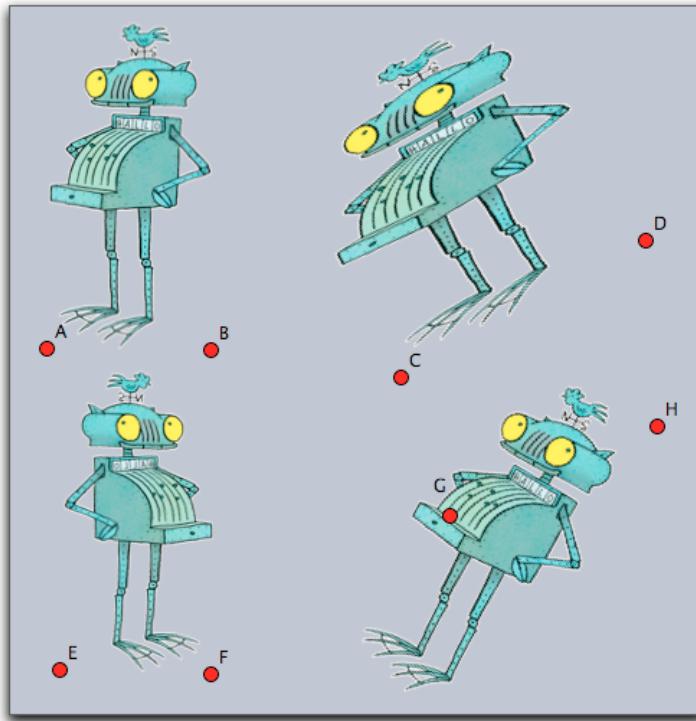
修飾子	型	効果
alpha	0.0 ... 1.0	画像の透明度
angle	real	参照点周りの回転角
rotation	real	angle と同じ
scale	real	拡大縮小
scale	ベクトル	それぞれの方向に拡大縮小
scalex	real	x 軸方向へ拡大縮小
scaley	real	y 軸方向へ拡大縮小
flipx	bool	垂直方向に反転
flipy	bool	水平方向に反転
ref	上記説明参照	参照点の位置指定
refx	上記説明参照	参照点の x 軸方向の位置指定
refy	上記説明参照	参照点の y 軸方向の位置指定
rendering	文字列	fast または nice の文字で表示品質

画像表示: `drawimage(<位置>, <位置>, <画像名>)`

位置と大きさを 2 つの参照点で指定して画像を表示する。初期状態では 2 つの参照点は画像の下辺の両端となる。この参照点を画像内の他の点にすることもできる。その方法は先ほどのものと同じ。

【例】

```
drawimage(A,B,"MyImage");
drawimage(C,D,"MyImage",aspect->1);
drawimage(E,F,"MyImage",flipx->true);
drawimage(G,H,"MyImage",refx1->"20\%",refy1->"50\%",ref2->"rt");
```

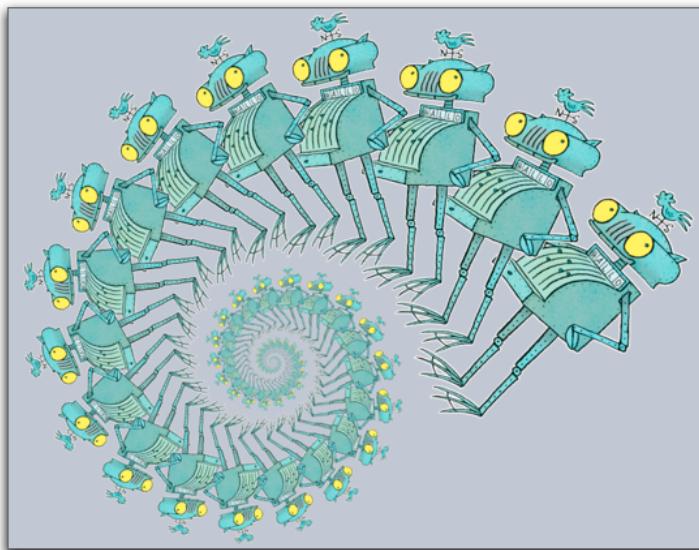


修飾子

修飾子	型	効果
alpha	0.0 ... 1.0	画像の透明度
flipx	bool	垂直方向に反転
flipy	bool	水平方向に反転
aspect	real	アスペクト比
ref1	上記参照	1つめの参照点の位置指定
refx1	上記参照	1つめの参照点の x 軸方向の位置指定
refy1	上記参照	1つめの参照点の y 軸方向の位置指定
ref2	上記参照	2つめの参照点の位置指定
refx2	上記参照	2つめの参照点の x 軸方向の位置指定
refy2	上記参照	2つめの参照点の y 軸方向の位置指定
rendering	文字列	fast または nice で表示品質

【例】次の例では、変換行列によって2つの参照点を反復的に変化させている。それぞれのステップで画像を表示していき、対数螺旋を描く。多くの画像を表示するために一つ問題がある。それは表示速度である。場合によってはきれいに表示するか、速く表示するかを選択する必要があるだろう。そのための修飾子が `rendering` で、"nice" か "fast" を設定する。初期状態では "nice" になっている。

```
a=[1,0];
b=[2,-1];
w=30° ;
m=[[cos(w),-sin(w)],[sin(w),cos(w)]]*0.9;
repeat(100,
  drawimage(a,b,"MyImage");
  a=m*a;
  b=m*b;
);
```

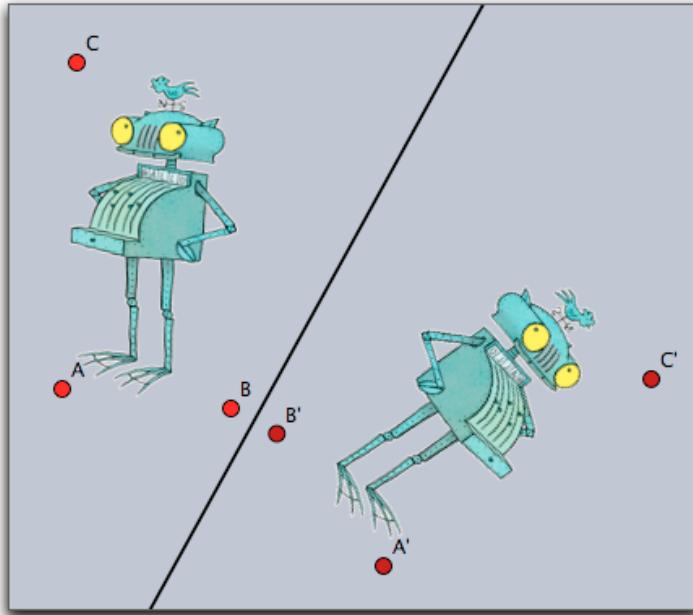


画像の表示: `drawimage(<位置>, <位置>, <位置>, <画像名>)`

3つの参照点を用いてアフィン変換をする。初期状態では、画像の底辺の左右と左上の点。参照点は修飾子で変更することができる。

この関数は、反転や回転のような幾何学的変換をして画像を表示するのに適している。そのためには、あらかじめ参照点を変換した点を用意する。次の例では点 A' , B' , C' は点 A,B,C を直線に関して鏡像変換（反転）した点である。

```
drawimage(A,B,C,"MyImage"));
drawimage(A',B',C',"MyImage"))
```



修飾子

この関数には次の修飾子が追加される。

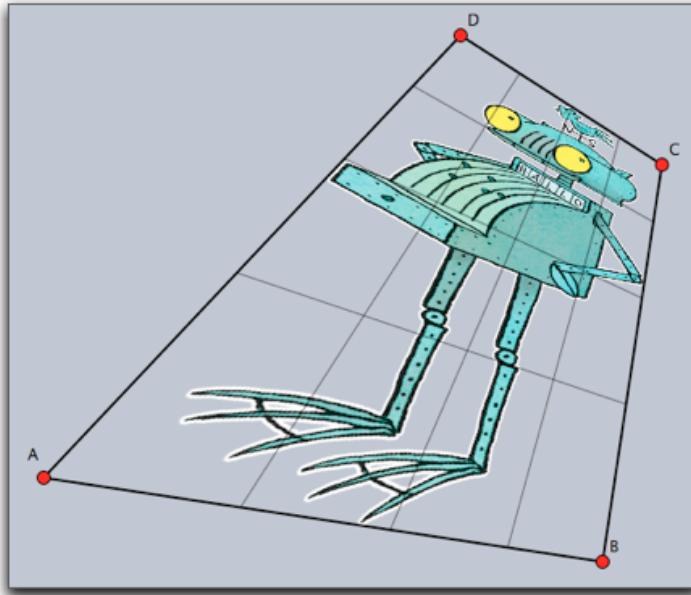
修飾子	型	効果
ref3	上記参照	3つめの参照点の位置指定
refx3	上記参照	3つめの参照点の x 軸方向の位置指定
refy3	上記参照	3つめの参照点の y 軸方向の位置指定

画像の表示: `drawimage(<位置>, <位置>, <位置>, <位置>, <画像名>)`

4つの参照点を指定して射影変換をする。初期状態では参照点は左下から4つの角の点を反時計回りにとられる。

【例】 `drawimage(A,B,D,C,"myimage")`

次の図ではわかりやすくするために格子を書き加えている。



修飾子

この関数には次の修飾子が追加される。

修飾子	型	効果
ref4	上記参照	4つめの参照点の位置指定
refx4	上記参照	4つめの参照点の x 軸方向の位置指定
refy4	上記参照	4つめの参照点の y 軸方向の位置指定

画像を変形する: `mapimage(<画像名>, <function>)`

この関数は、引数の関数を用いて画像を変形させる。通常、この関数は、ある関数を用いて2次元ベクトルを2次元ベクトルに写すものとする。`complex` 修飾子を用いて複素数平面から複素数平面へ写すこともできる。関数の範囲は `x-range` と `y-range` 修飾子で指定するが、指定がなければ 0.0 と 1.0 の間とされる。この範囲が画像の長方形の縦横幅に対応する。この範囲ですべての点が写される。この機能は基本的に `mapgrid` 関数と類似している。

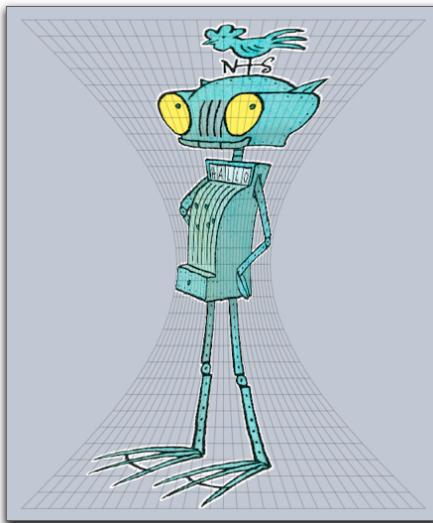
【例】ここでは、x 軸方向を `sin` 関数を使って変形する。まず、その関数を定義する。次に `mapimage` 関数で変形する。図では、わかりやすくするために方眼を描いている。

```
f(z):=(z\_1*(sin(z\_2)+1.3),z\_2);
mapimage("MyImage",f(\verb|#!|,
    xrange->(-1,1),
    yrange->(0,pi),resolution->30
);
mapgrid(f(\verb|#!|),color->(0,0,0),alpha->0.3,
```

```

xrange->(-1,1),
yrange-(0,pi),resolution->30
);

```

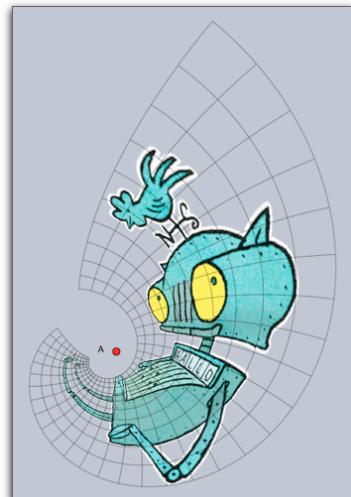


次の例は前のものと類似している、関数として複素関数を用いている。

```

r=complex(A);
f(x):=exp(r*x);
mapimage("myimage",f(\verb|\#|),
complex->true,
xrange-(0,1),
yrange-(0,pi),resolution->30
);
mapgrid(f(\verb|\#|),complex->true,
xrange-(0,1),
yrange-(0,pi),
color-(0,0,0),alpha->0.5,
resolutiony->30,
resolutionx->10,
step->10,size->1
);

```



修飾子

この関数には次の修飾子がある。

修飾子	型	効果
alpha	0.0 ... 1.0	画像の透明度
xrange	ベクトル	x 軸方向の範囲
yrange	ベクトル	y 軸方向の範囲
complex	bool	複素関数を用いるフラグ
resolution	int	変換後の画像の解像度

画像の大きさを取得する: `imagesize(<画像名>)`

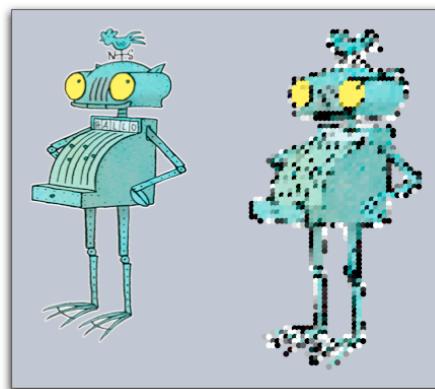
画像の大きさを返す。戻り値は幅と高さの画素数を示す 2 つの整数。

画素情報を取得する: `imagergb(<画像名>, <整数>, <整数>)`

`imagergb(<画像名>, x, y)` で、(x, y) の位置にある画素の色情報を取得する。戻り値は 4 次元のベクトルで、はじめの 3 つは RGB 値 (0~255), 4 番目はアルファ値。

【例】次のコードでは、まず画像を表示してその大きさを取得する。次に、両方向に標本点を取り、その色と透明度で、粗っぽい点からなる画像を作成する。

```
drawimage(A, "MyImage", scale->2);
dim=imagesize("MyImage");
forall((0..dim\_1/10)*10,i,err(i);
      forall((0..dim\_2/10)*10,j,
            col=imagergb("MyImage",i,j);
            draw((i,-j)*.03,color->(col\_1,col\_2,col\_3)/255,alpha->col\_4,border->false);
      );
)
```



13.3 カスタム画像を作る

ここまででは、メディアブラウザであらかじめ読み込まれた画像についての操作であったが、Cinderella の描画画面から画像を作ることもできる。いったん画像として作ってしまえば、CindyScript の `canvas` 関数でそれを操作することができる。これにより、描画のための命令などは見えなくなる。しかし、カスタム画像が作られればそれを描画手順のうちの一つとして画面上に描くことができる。画像表示は描画と似て、画面の任意の場所に描くことができます。この概念はとても強力だが、基本的な使い方を示すだけにする。

カスタム画像を作る: `createimage(<画像名>, <整数>, <整数>)`

`createimage(<画像名>, width, height)` 関数は、指定された画像のイメージバッファを作成する。最初はそこには何もない。イメージバッファはメディアブラウザのもとで指定した名前でアクセス可能となり、`drawimage(...)` 関数で使えるようになる。

画像の内容を消去: `clearimage(<画像名>)`

この関数は、画像からすべての内容を消去する。消去後も画像は存在するが、描画された内容はなく、いわば透明な状態である。

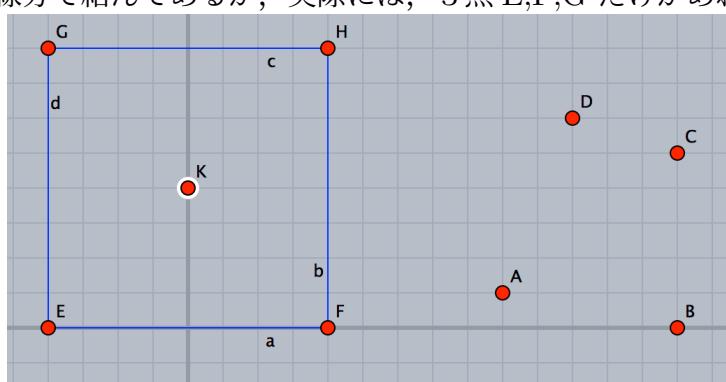
画像の削除: `removeimage(<画像名>)`

この関数はメディアブラウザから画像を取り除く。これ以降アクセスすることはできなくなる。

13.4 キャンバスに描く

描画画面をキャンバスのようにして、描いた図を CindyScript で画像化することができる。できた画像は、読み込んだ画像と同様な変形ができる。ここでは、その手順を例示する。

まず、図のように点を取る。点 A,B,C,D はできた画像を射影変形して表示するための点、矩形 EFGH はキャンバスの領域、点 K は作図のための点である。矩形 EFGH は、わかりやすくするために線分で結んであるが、実際には、3 点 E,F,G だけがあればよい。

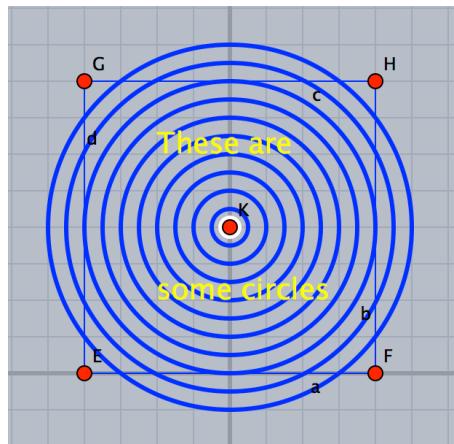


initialization スロットに、次のようなコードを書くことによって、キャンバスが用意できる。"image" という名称で、 400×400 のサイズの画像を作ることになる。

```
createimage("image",400,400);
```

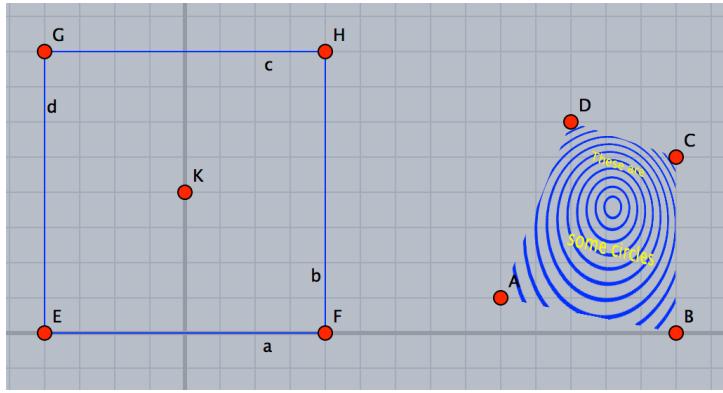
次に、Draw スロットで例として同心円を描く。

```
linesize(3);
repeat(10,t,drawcircle(K,t*.5));
drawtext(K+[-2,2],"These are",size->20,color->[1,1,0]);
drawtext(K+[-2,-2],"some circles",size->20,color->[1,1,0]);
```



この図で、矩形 EFGH で囲まれた部分（実際には E,F,G の 3 点で指定した長方形の中）が画像化される。図を確かめたら、上の描画スクリプトを、canvas 関数の中に入れる。このとき、図は表示されなくなるので、できた画像を drawimage で表示する。

```
clearimage("image");
canvas(E,F,G,"image",
       linesize(3),
       repeat(10,t,drawcircle(K,t*.5)),
       drawtext(K+[-2,2],"These are",size->20,color->[1,1,0]),
       drawtext(K+[-2,-2],"some circles",size->20,color->[1,1,0]));
);
drawimage(A,B,C,D,"image");
```



この例では4点を指定して射影変換を行っているが、1点だけで表示することももちろんできる。また、キャンバスの画像はビットマップであることに注意する。したがって適切な解像度を指定しなければならない。上の例で初期値として 80×80 のサイズにすると、小さな画像を大きく表示することになり結果として粗い画像になる。

この例では、3点でキャンバス領域を指定したが、これらの点を「参照点」といい、1個ないし3個の参照点を使うことができる。

キャンバスに描く（参照点1つ）: `canvas(<位置>, <画像名>, <drawing code>)`

この関数は `|画像名|` の画像をキャンバスとして用い、1つの参照点に対して画像を作る。修飾子は `drawimage(<位置>, <画像名>)` と同様。

修飾子

修飾子	型	効果
angle	real	参照点周りの回転角
rotation	real	angleと同じ
scale	real	拡大縮小
scale	ベクトル	それぞれの方向に拡大縮小
scalex	real	x軸方向へ拡大縮小
scaley	real	y軸方向へ拡大縮小
flipx	bool	垂直方向に反転
flipy	bool	水平方向に反転
ref	上記説明参照	参照点の位置指定
refx	上記説明参照	参照点のx軸方向の位置指定
refy	上記説明参照	参照点のy軸方向の位置指定

キャンバスに描く（参照点2つ）: `canvas(<位置>, <位置>, <画像名>, <drawing code>)`

この関数は `|画像名|` の画像をキャンバスとして用い、2つの参照点に対して画像を作る。

修飾子は `drawimage(<位置>, <画像名>)` と同様, `ref2, refx2, refy2` が追加される。

キャンバスに描く (参照点3つ): `canvas(<位置>, <位置>, <位置>, <画像名>, <drawing code>)`

この関数は `|画像名|` の画像をキャンバスとして用い, 2つの参照点に対して画像を作る。

修飾子は `drawimage(<位置>, <画像名>)` と同様, `ref3, refx3, refy3` が追加される。

14 シェイプ

直線や多角形、円などの基本的な描画に加えて CindyScript によってこれらを結合した図形—シェイプーを描くことができる。シェイプは直接は目に見えないが、充填材やアウトラインの描画、クリッピングとして使える。シェイプは和集合、積集合、補集合といった集合の論理演算によって作られる。ここでは、まず基本的な演算について説明し、そのあと、より精巧な例を示す。

14.1 シェイプの初步

円形のシェイプ: `circle(<点>, <半径>)`

円形のシェイプを作る。

多角形シェイプ: `polygon(<list>)`

`<list>` に頂点のリストを渡して多角形のシェイプを作る。

半平面シェイプ: `halfplane(<直線>, <点>)`

半平面のシェイプを作る。半平面は境界線 `<直線>` と、点 `<点>` によって定義される。境界線のどちら側であるかがその点で決まり、点が含まれる方がその半平面である。`<直線>` は3次元の同次座標か直線オブジェクト。

スクリーンのシェイプ: `screen()`

すべてのアクティブな描画要素を覆うのに十分に大きい長方形を作る。

14.2 シェイプの結合

それぞれのシェイプは、論理演算によって結合することができる。それには次の3つの演算がある。

<code><シェイプ 1> ++ <シェイプ 2></code>	2つのシェイプの併合
<code><シェイプ 1> ^^ <シェイプ 2></code>	2つのシェイプの共通部分
<code><シェイプ 1> -- <シェイプ 2></code>	2つのシェイプの差

14.3 シェイプの使い方

シェイプは、充填と境界線描画、クリッピングの3つの方法で使われる。

シェイプによる充填: `fill(<シェイプ>)`

この関数ではシェイプを与えられた色で塗りつぶす。使える修飾子は、`color` と `alpha`。

シェイプを描く: `draw(<シェイプ>)`

この関数はシェイプの輪郭線を描く。この関数は `draw` 関数の拡張で、`draw` と同じ修飾子が使える。

クリップパスの設定: `clip(<シェイプ>)`

この関数は、シェイプのクリップパスを設定する。これ以降に描画するものはすべてこのクリップパスで止めらる。クリップパスは表現スタックに加えられるが、それは `grestore()` あるいは `greset()` で取り除くことができる。

【例】

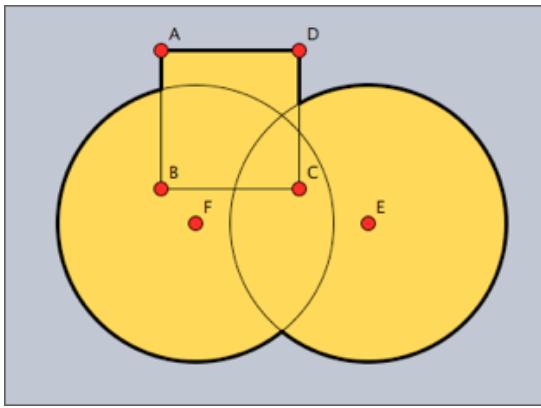
まず「点を加える」モードで点 A~F をとっておく。次のコードで3つの異なるシェイプ1つの正方形と2つの円が定義され、それらが論理演算によって結合されている。小さなシェイプは、細い線でアウトラインが描かれている。

```
shape1=circle(E,4);
shape2=circle(F,4);
shape3=polygon([A,B,C,D]);
color([0,0,0]);
shape=shape1++shape2++shape3;
fill(shape,color->[1,0.8,0]);
draw(shape,size->3);
draw(shape1);
draw(shape2);
draw(shape3);
```

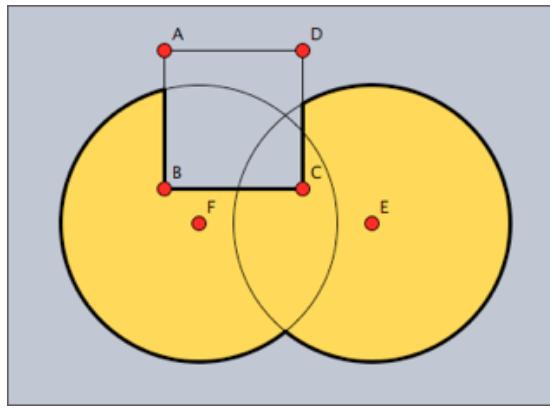
次の最初の図がこの結果である。ほかの3つの図は、それぞれ図の上に示した演算で作成したシェイプ。

`shape1++shape2++shape3`

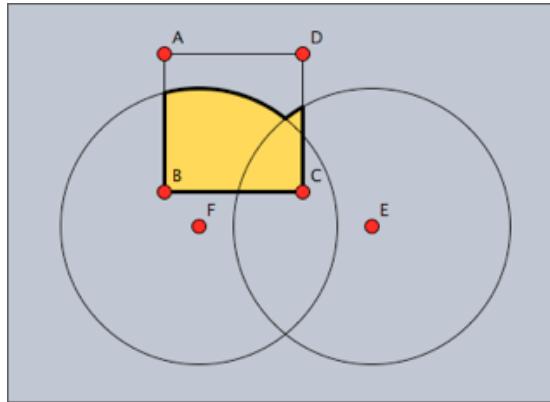
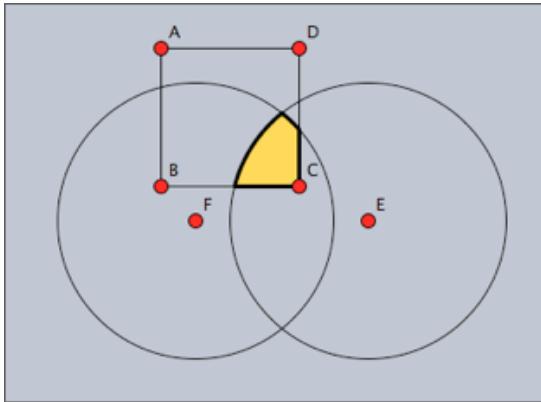
`shape1~~shape2~~shape3`



shape1++shape2--shape3

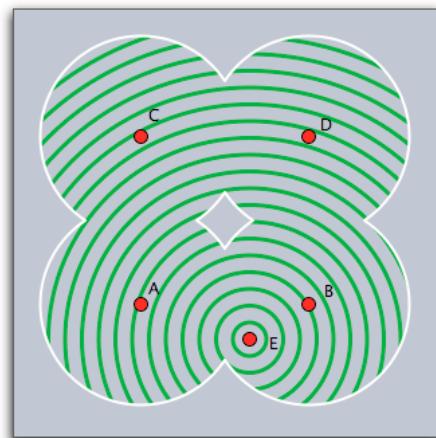


shape1++shape2~~shape3



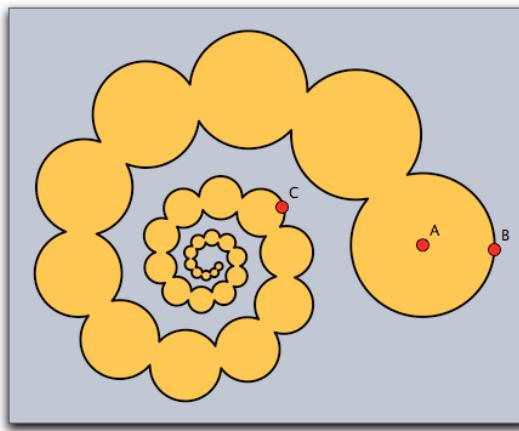
次のコードはクリップパスの使い方を示す。「点を加える」モードで5つの点 A,B,C,D,E をとっておく。スクリプトでは、まず4つの円を合併したシェイプを定義する。すると、このシェイプが、点Eを中心とした同心円のクリッピングパスとして働く。

```
r=3;
shape=circle(A,r)++circle(B,r)++circle(C,r)++circle(D,r);
clip(shape);
repeat(60,t,
    drawcircle(E,t/2,color->[0,.6,0],size->3);
);
greset();
draw(shape,color->[1,1,1],size->2);
```



シェイプは、次の例に示すような反復系のオブジェクトにもなる。ただし、あまり複雑にすると破綻をきたすので注意。複雑な形は Cinderella の動作を遅くしてしまう。

```
a=complex(A);
b=complex(B);
z=complex(C);
shape=circle([0,0],0);
repeat(50,
    shape=shape++circle(gauss(a),|a-b|);
    a=a*z;
    b=b*z;
);
fill(shape,color->[1,0.7,0]);
draw(shape,color->[0,0,0],size->2);
```



15 座標系と基底

通常、CindyScript の座標系は Cinderella の図形描画での座標系と同じである。しかし、特殊関数によって座標系を変えることができる。そのあとは、この修正された画面に対して描画が行われる。座標変換を行うときは、どこで実際の描画を行っているかを決定するのが難しいかも知れない。そこで、変換の関数を使いやすくするために、CindyScript は `gsave` と `grestore` の関数を用意している。それは、現在の描画エンジンをスタックに push/pop する。これは、図形の表現情報の他に、現在の座標変換を含む。

15.1 座標変換

座標系の平行移動: `translate(<list>)`

この関数は `[<real>, <real>]` 型の `<list>` を与えて、このベクトルにより描画する座標

系を移動する。

座標系の回転: `rotate(<real>)`

この関数は、実数 `<real>` の角だけ座標系を回転する。角は弧度法。度数法で与えたい場合は $^\circ$ 演算子を使う。たとえば、`rotate(30°)` とすれば、座標系を 30° 回転する。

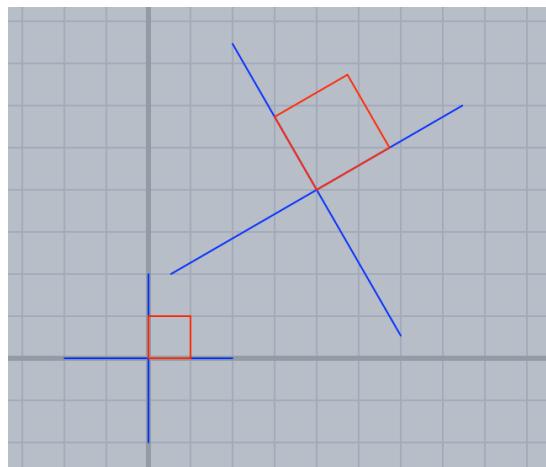
拡大・縮小: `scale(<real>)`

この関数は `<real>` によって、拡大・縮小を行う。

これらの変換は、続けて行うこともできる。変換の順序によって結果は異なる。

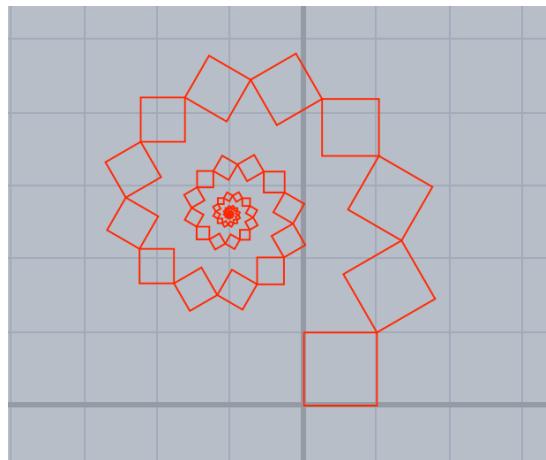
【例】 座標軸と、頂点の座標が $[0,0], [0,1], [1,1], [1,0]$ である正方形を描く関数 `drawaxis()` を定義し、座標変換を行って描画する。座標変換は 2 倍して、x 軸方向に 1, y 軸方向に 1 だけ平行移動、 30° 回転、という順序でおこなう。描画した図形を移動するのではなく、座標変換を行ってから描画することに注意。

```
drawaxis():=(  
    draw([-2,0],[2,0]);  
    draw([0,-2],[0,2]);  
    drawpoly([[0,0],[0,1],[1,1],[1,0]],color->[1,0,0]);  
);  
drawaxis();  
scale(2);  
translate([2,2]);  
rotate(30° );  
drawaxis();
```



【例】繰り返して座標変換を行い、正方形を描く。

```
repeat(90,  
    drawall(square);  
    translate((1,1));  
    scale(0.92);  
    rotate(30° );  
)
```

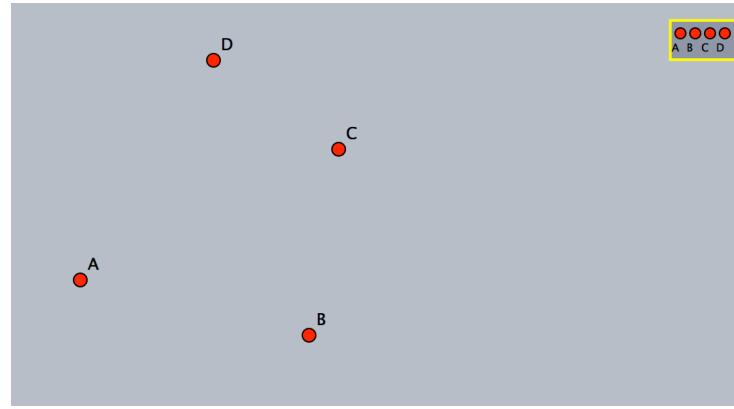


15.2 射影基底との関係

```
setbasis(<basis>)
```

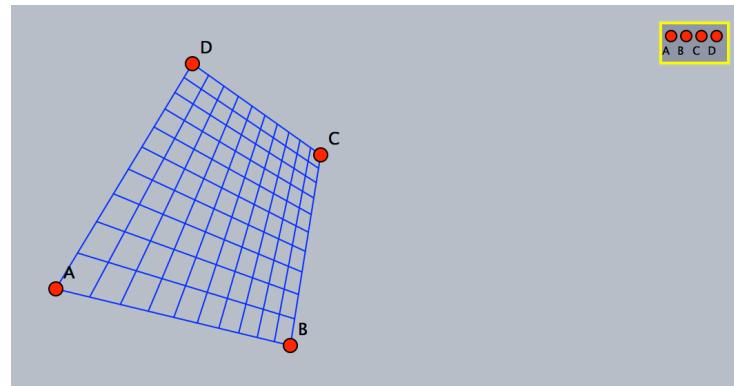
Cinderella の幾何学部分は、すべての描画が関わる基底を定められるようになっている。これらの基底とは、平行移動、相似、アフィン、射影で、CindyScript でも使える。`setbasis()` 関数は Cinderella で使われる基底を定義する。引数は Cinderella で使われる基底のラベルでなければならない。`setbasis` 関数を適用したあとは、CindyScript の以前の座標変換は使われなくなるが、それらは `gsave` で保存しておく、`grestore` で呼び出すことができる。

【例】まず、図のように4点 A,B,C,D をとり、モードメニューの「特別」－「基底の定義」を選ぶ。ガイドに従って、A,B,C,D の4点をクリックし、選択モードにしてこのモードを抜けると、射影基底 Bas0 が定義され、右上にアイコンができる。



これは、一辺の長さが 1 である正方形(単位正方形)の 4 頂点 $[0,0], [0,1], [1,1], [1,0]$ が、点 A,B,C,D に対応するような基底であることを意味する。実際、次のスクリプトを実行すると、単位正方形内の格子を描く。

```
setbasis(Bas0);
x=(0..10)/10;
drawall(apply(x,([\verb|\#|,0],[\verb|\#|,1])));
drawall(apply(x,([0,\verb|\#|],[1,\verb|\#|]));
```



CindyScript の内部基底を、既存の点に基づく基底に直接関連付けることも可能である。すると、Cinderella での基底を変えなくて済む。これは、基底を定義するための点を与えて、次の関数で実行できる。

平行移動基底を設定する: `setbasis(<点 1>)`

相似基底を設定する: `setbasis(<点 1>,<点 2>)`

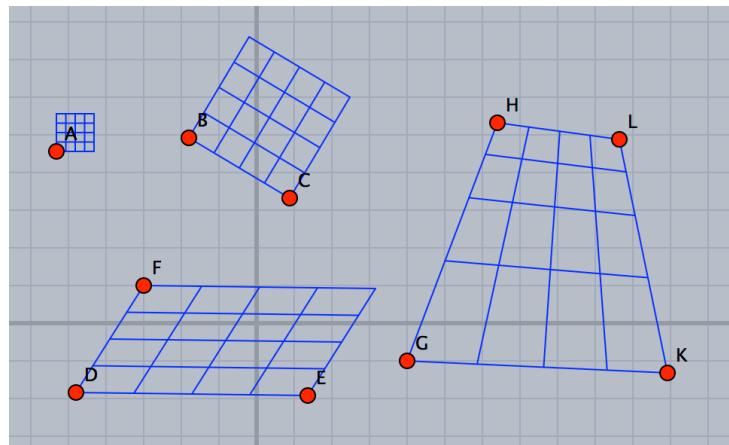
アフィン基底を設定する: `setbasis(<点 1>,<点 2>,<点 3>)`

射影基底を設定する: `setbasis(<点 1>,<点 2>,<点 3>,<点 4>)`

【例】次のコードはこの性質を示すものである。sq の部分は正方形の格子を描く関数の定義。

この格子をいろいろな基底によって描く。

```
sq:=(draw([0,0],[1,0]);
      draw([0,0.25],[1,0.25]);
      draw([0,0.5],[1,0.5]);
      draw([0,0.75],[1,0.75]);
      draw([0,1],[1,1]);
      draw([0,0],[0,1]);
      draw([0.25,0],[0.25,1]);
      draw([0.5,0],[0.5,1]);
      draw([0.75,0],[0.75,1]);
      draw([1,0],[1,1]));
);
setbasis(A);
sq;
setbasis(B,C);
sq;
setbasis(D,E,F);
sq;
setbasis(G,H,L,K);
sq;
```



15.3 基底スタック

座標系変換は全体的な表示に影響する。一時的に座標系変換を行い、その後以前の状態に戻したいことはよくある。CindyScriptにはそのための関数が用意されている。

図の状態の一時保存: `gsave()`

図の状態を戻す: grestore()

`gsave` 関数は、すべての図形の状態（座標変換、大きさ、色、透明度）についての情報をスタックに保存する。`grestor` はこの情報をスタックから引き出す。したがって、`gsave()` `grestore()` の間は、他のコードの部分に影響を与えることがない。

グラフの状態を消去する: greset()

この関数は、スタックに保存されていた座標系やすべての図形に関する色、大きさなどの情報を消去し、初期状態に戻す。

15.4 レイヤー

ある意味で、Cindy script で描画される図には 3 番目の次元がある。各々の図は特定の層（レイヤー）にある。レイヤーは数で指定される。図が描かれるときレイヤーは付随する番号の順に塗られる。レイヤーが指定されなければ、Cindy script で描画される図は背景層すなわち幾何要素のある層で描画される。初期状態では Cindyscript の `draw` が実行される前にレイヤーは消去される。しかしながら、自動消去されないレイヤーをマークすることも可能である。

描画レイヤーを設定する: layer()

この関数はレイヤーを指定したレベルに設定する。初期状態では描画が行われる前にレイヤーは消去される。通常、Cindyscript で描く図はスクリプトが書かれている順に描かれるので、図が重なる場合はあとから書かれた方が上になる。しかし、レイヤーを設定することにより、重ね合わせを変えることができる。

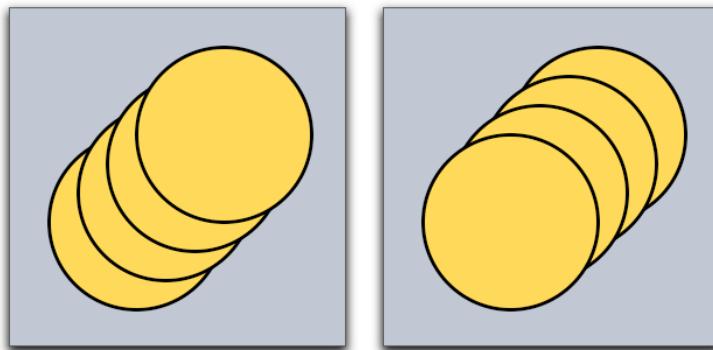
【例】次のスクリプトでは、通常の描画順で 4 つにオーバーラップして円盤を描く。（下図左）

```
cir(x,y):=(  
    fillcircle((x,y),3,color->(1,.8,0));  
    drawcircle((x,y),3,color->(0,0,0),size->3);  
);  
cir(0,0);  
cir(1,1);  
cir(2,2);  
cir(3,3);
```

それぞれの円盤を描画する前にレイヤー関数が呼ばれると、それぞれの円盤を逆順に描画することができる。（下図右）

```
layer(6);cir(0,0);
```

```
layer(5);cir(1,1);
layer(4);cir(2,2);
layer(3);cir(3,3);
```



レイヤーを空にする: emptylayer()

この関数はレイヤーを空にする。

レイヤーからすべての図を消去する: clrscr()

この関数は、スクリーンで実行されたすべての図を消去する。図が本当に消去された状態になっているかどうかを確認することは時々役に立つ。特に、他のスクリプトスロットから repaint() 関数が呼ばれているときに必要になるだろう。

レイヤーの自動消去: autoclearlayer(<整数>,<ブール値>)

レイヤーのフラグの自動消去はこの関数で行なえる。初期状態で、すべてのレイヤーは画面の再計算の間に自動的に消去される。この自動消去フラグを `false` にすることでこれを切り替えることができる。

レイヤーが自動消去されなければすべての描画命令が保存される。Cinderella はビットマップを保存しないであらゆるステップで画面を書き直すので、レイヤーのために多くの描画命令を使うとパフォーマンスに影響する。

【例】initialization スロットに次のコードを書くと、消去されないレイヤーを背景あるいは前面に描くことができ、 draw スロットに書くと、何か操作するたびにレイヤーを書き換える。

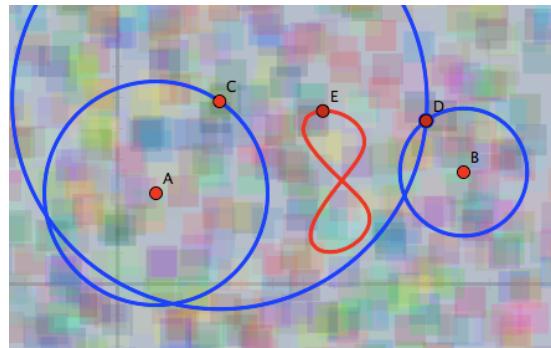
```
autoclearlayer(-4,false);
layer(-4);
repeat(1000,
  p = [random(20)-5,random(20)-5];
  color([random(),random(),random()]);
  
```

```

fillpolygon(apply([[0,0],[1,0],[1,1],[0,1]],p+#),alpha->.2);
);
layer(0);

```

このコードを Initialization スロットに描くと、レイヤー 4 で背景を描く。自動消去フラグが `false` なので、次の図のように、他の作業をする間も表示したままになる。これを `draw` スロットに書いて同じ効果を得るためにには、乱数によってできる異なる長方形と色をすべて保存しておかなければならない。



15.5 スクリーン境界の決定

`screenbounds()`

もし射影平面の有限部分だけ使い、球面表示を使わないのであれば、スクリーン境界を求めるに意味がある。この関数は、見える範囲の長方形を定義している 4 つの同次座標のリストを返す。この関数はユークリッド表示においてのみ有効。

画面解像度の決定: `screenresolution()`

双曲表示・球面表示と対照的に、ユークリッド表示はいたるところに同じ画面解像度を持つ。この関数は、原点と $[0,1]$ 間の画素数を与える。同じ図に対していくつかのユークリッド表示があるとき、この関数はすべての画面解像度の最大値を返す。

【例】 点に色をつけてチェックボードを作ることができる。次のコードは、現在の拡大率に関わりなくすべての画素に色をつける。画素単位で色をつけることを勧めているわけではないが、これが必要なことがあるかもしれない。ほとんどの場合、`colorplot` 関数が使いやすく、より高速である。

```

upperleft=(screenbounds()\_1).xy;
lowerright=(screenbounds()\_3).xy;
width in pixels=screenresolution()*(lowerright.x-upperleft.x);

```

```

height in pixels=screenresolution()*(upperleft.y-lowerright.y);
repeat(width in pixels/2, x,
       start->upperleft.x, stop->lowerright.x,
       repeat(height in pixels/2, y,
              start->upperleft.y, stop->lowerright.y,
              draw([x,y],border->false,size->.5,color->[0,0,0]);
       );
);

```

16 幾何学的演算

16.1 リストと座標

直線のための座標は常に同次座標である。すなわち、3つの数のリストで、 $[a,b,c]$ が、方程式 $ax + by + c = 0$ を表す。点の座標はユークリッド座標（2つの数のリスト $[x,y]$ ）か、または同次座標（3つの数のリスト $[x,y,z]$ で点 $[x/z, y/z]$ ）を表す）。戻り値は常に同次座標。以下の説明の中では、特に断りなく、点の形式の引数は<点>で、直線の形式の引数は<直線>で表す。<点>と<直線>は両方とも3つの数のリストで表現することができる所以、両者を区別する方法が必要になる。内部的には、リストは、それが幾何学的にどのような意味を持つのかというフラグを持っている。その情報は、`geotype(<list>)` 関数で得ることができる。この関数は、"Point", "Line", "None" のいずれかを返す。ベクトルが持つ幾何学的な意味に応じて、`draw` 関数は対応する図形を描く。

16.2 基本的な幾何学関数

2直線の交点: `meet(<直線 1>, <直線 2>)`

この関数は2本の直線の交点を計算し、その同次座標を返す。

2点を通る直線: `join(<点 1>, <点 2>)`

この関数は、2点を通る直線を計算し、その同次座標を返す。

平行線を求める: `parallel(<点>, <直線>)`

平行線を求める: `parallel(<直線>, <点>)`

この関数では、点と直線を引数として与え（順序は問わない）、その点を通りその直線に平行な直線を求める。戻り値は直線の同次座標。この関数はユークリッド幾何に対して適用される。双曲線幾何と楕円幾何はサポートしていない。この関数は、`para(...)` と略すこと

もできる。

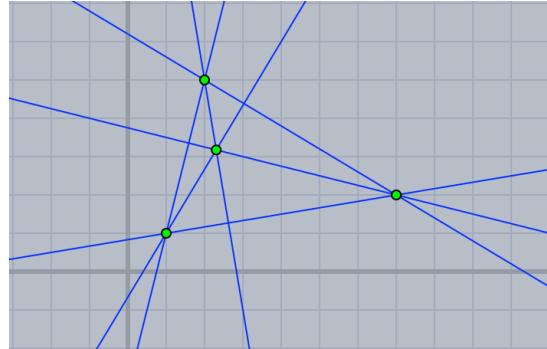
perp 垂線: perpendicular(<点>, <直線>)

垂線: perpendicular(<直線>, <点>)

この関数では、点と直線を引数として与え(順序は問わない)、その点を通りその直線に垂直な直線を求める。戻り値は直線の同次座標。この関数はユークリッド幾何に対して適用される。双曲線幾何と楕円幾何はサポートしていない。この関数は、`perp(...)`と略すこともできる。

組み合わせた例 次のコードを実行すると図のようになる。それぞれの演算の戻り値が図形のタイプ(点か直線か)を内在しているので、`drawall` 関数によって自動的に正確に描画される。

```
A=[1,1];
B=[2,5];
C=[7,2];
a=join(B,C);
b=join(C,A);
c=join(A,B);
ha=perpendicular(A,a);
hb=perpendicular(B,b);
hc=perpendicular(C,c);
X=meet(ha,hb);
drawall([a,b,c,d,ha,hb,hc,X,A,B,C]);
```



垂直なベクトル: perpendicular(<list>)

`perp` 関数の引数が、2つの数からなるリスト1つだけとする。その場合は、リスト [a,b] は [-b,a] に変換される。それは、もとのベクトルを 90° 回転したものである。

三角形の面積: area(<点1>, <点2>, <点3>)

この関数は、<点1>, <点2>, <点3>を3つの頂点とする三角形の面積を計算する。点の方向が反時計回りならば面積は正、時計回りならば面積は負になる。3点が同一直線上にあるときは面積はゼロ。

16.3 有用な線形代数関数

線形代数で使う次の関数は幾何でも有用である。特に3次元のベクトルに適用できる。その状況(行列の計算、内積など)については、ベクトルと行列の節を参照されたい。

3 点の位置ベクトルの行列式: `det(<ベクトル 1>, <ベクトル 2>, <ベクトル 3>)`

この関数は、3つの3次元ベクトル `<ベクトル 1>`, `<ベクトル 2>`, `<ベクトル 3>` からなる 3×3 行列の行列式を計算する。ベクトルと行列の節にある一般的な行列式と異なり、この方法はパフォーマンスが優れている。

2 点の位置ベクトルの外積: `cross(<ベクトル 1>, <ベクトル 2>)`

この関数は、2つの3次元ベクトルの外積を計算する。外積は3次元ベクトルで、もとの2つのベクトルと垂直。

16.4 変換とオブジェクトの型

オブジェクトの型: `geotype(<list>)`

この関数は、オブジェクトが明確な幾何学的意味を持つかどうかを決定する。戻り値は、文字列で "Point", "Line", "None" のいずれか。

この関数を、2つの数からなるリストに適用した場合は、常に `Point` を返す。3つの数からなるリストに適用した場合は、このリストの内部フラグによって "Point", "Line", または "None" のいずれかを返す。Cinderella で描画した幾何オブジェクトであれば、常にそのフラグを有している。`meet` 関数の戻り値は常に "Point"。`join`, `parallel`, `perpendicular` 関数の戻り値は常に "Line"。さらに、幾何学的意味は `line` と `point` 関数によって明確に設定することができる。

点の指定: `point(<vector>)`

この関数は、3つの数のベクトルを、はっきりと "Point" として位置づける。引数が3つの数からなるベクトルでなければ無効。

直線の指定: `line(<vector>)`

この関数は、3つの数のベクトルを、はっきりと "Line" として位置づける。引数が3つの数からなるベクトルでなければ無効。

点を複素数に変換: `complex(<点>)`

この関数は点の座標を複素数に変換する。ここで、ユークリッド座標系はガウスの複素数平面と同一視され、点 `[a, b]` は複素数 `a+i*b` に変換される。

複素数を点に変換: `gauss(<点>)`

複素数 `a+i*b` を2つの数のリスト `[a, b]` に変換する。

4点または直線の複比: `crossratio(<vector>, <vector>, <vector>, <vector>)`

4点の幾何複比を計算する。4点が共線であれば、通常の実数射影平面における複比が計算される。4点が共線でなければ、複素射影直線の対応した点における複比が計算される。共線的無限遠点に対しては2つの結果は同等。

4つの数の複比: `crossratio(<数>, <数>, <数>, <数>)`

4つの実数または複素数の複比 $(A/B)/(C/D)$ を計算する。

16.5 幾何学変換と基底

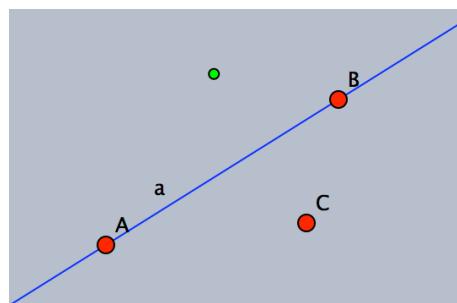
代数的な方法で幾何の変換を扱うことができる。変換は 3×3 行列で表される。実際の変換は、点の同次座標に変換行列を掛けて行われる。この変換行列を計算するためにいくつかの関数がある。

線対称: `linereflect(<直線>)`

<直線> に関する対称な点を求める行列を返す。

【例】次のコードは、作図した直線 a (AB) と点 C に対し、対称変換行列を作り、点 C の同次座標に掛けて対称点を求める。結果は緑色の点。

```
m=linereflect(a);  
draw(m*C.homog);
```



点対称: `pointreflect(<点>)`

<点> に関する対称点を求める行列を返す。

平行移動: `map(<点1>, <点2>)`

<点1> を <点2> に写す平行移動を表す行列を返す。

相似変換: `map(<点1>, <点2>, <点3>, <点4>)`

<点1> を <点2> に、<点3>を<点4> に写す相似変換を表す行列を返す。

アフィン変換: `map(<点 1>, <点 2>, <点 3>, <点 4>, <点 5>, <点 6>)`

<点 1>を<点 2>に, <点 3>を<点 4>に, <点 5>を<点 6> に写すアフィン変換を表す行列を返す。

射影変換: `map(<点 1>, <点 2>, <点 3>, <点 4>, <点 5>, <点 6>, <点 7>, <点 8>)`

<点 1>を<点 2>に, <点 3>を<点 4>に, <点 5>を<点 6>に, <点 7>を<点 8> に写す射影変換を表す行列を返す。

17 計算

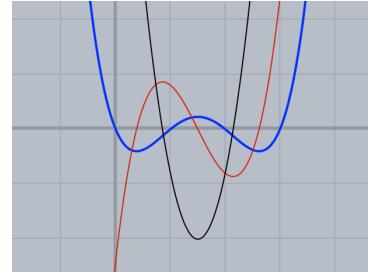
CindyScript は純粹に数的言語である。記号的計算は行なわない。にもかかわらず, 通常記号的システムだけが行えるようないくつかの手続きを行うことができる。たとえば, 関数の微分や接線を計算することができる。

17.1 微分と接線

微分: `d(<function>, <var>)`

この関数は, 第1引数 に与えられた関数の導関数を作る。微分は標準的な実行変数 # についておこなわれるが, 変数名を第2引数に指定しする。次のコードが使用例。青の線がもとの関数 $f(x)$, 赤の線が第1次導関数 $g(x)$, そして黒の線が第2次導関数 $h(x)$ のグラフ。

```
f(x):=(x-3)*(x-2)*(x-1)*x*.4;  
g(x):=d(f(#),x);  
h(x):=d(g(#),x);  
plot(f(x),size->2);  
plot(g(x),color->[0.8,0,0]);  
plot(h(x),color->[0,0,0]);
```



第2次導関数は第1次導関数を微分したものである。しかし注意事項がある。微分関数は完全に数の原則に基づいている。 $f(x)$ が微分可能ならば, 導関数は $d(f(#),x)$ で定義される。ここで, x が微分する変数を示している。導関数の値は, 式 $(f(x+eps)-f(x-eps))/2eps$ によって計算される。 eps は十分小さい数。これは, この点における実際の微分係数に十分近い値になる。しかし, 連続して何度もこの関数を使うと, かなり誤差を生じる。5回も微分すると結果は使えない。したがって第5次導関数ではもはや理にかなった計算はできない。

修飾子

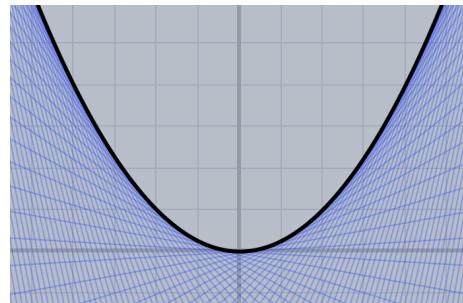
初期値として, 上記の式の eps は 0.0001 に設定されている。この値は, 高次導関数の信頼性と精度のバランス上十分理にかなった値である。この値は修飾子 `eps-><数>` によって

変更することができる。

接線: tangent(<関数>, <値>)

この関数は、微分とよく似ている。しかし、微分係数を計算する代わりに、ある点における接線の同次座標を計算する。その点は第2引数で与える。次のコードでは、放物線のいろいろな接線を計算している。

```
f(x):=(x^2)/4;  
repeat(250,start->-30,stop->30,x,  
t=tangent(f(#),x);  
draw(t,alpha->.3);  
);  
plot(f(x),size->3,color->[0,0,0]);
```



tangent 関数の戻り値は3次元ベクトルの同次座標である。加えて、直線の内部フラグが立っている。(「幾何学関数」参照)そのため、自動的に直線が引かれる。

17.2 高度な計算

実数値を推測する: guess(<数>)

推測関数は、浮動小数点数から記号的意味を取り戻すのに使われる。guess 関数は入力として数を予想し、文字列を返す。文字列は入力された数を記述する記号的な式。guess 関数は、次の形式の文字列を生み出そうとする。

```
a+b*sqrt(c)
```

ここで a, b, c は、約 1000 以下の分母と分子で表される有理数。入力された数が十分な精度で表現できるならば、上記の形で表す。そうでなければ、guess 関数はそのまま入力値を返す。

たとえば、guess(0.25) は "1/4" を返す。guess(3.14159) はそのまま "3.14159" を返す。

guess 関数の背後で働いているのは、いわゆる PSLQ と呼ばれるアルゴリズムである。実数 x, y, z が与えられたときに、整数関係 $ax + by + cz = 0$ を発見する実に巧妙なアルゴリズムである。実数 x が2次関係式の解であるかどうかを調べたいならば、 $a + bx + cx^2 = 0$ の整数関係を探さなくてはならない。これが、guess 関数が実装しているものである。解はその整数係数を使って再構成される。

n 次方程式の解: roots(<list>)

roots 関数は、1変数の多項式からなる方程式の解を示す。低次の項から高次の項に向かって順に係数を与える。結果は複素数のこともある。

【例】たとえば、方程式 $x^2 + 1 = 0$ の解を求める場合は、単に `roots([1,0,1])` とする。結果は複素数 $[-0-i*1, -0+i*1]$ である。

次のコードは、3次方程式の解を計算してグラフとともに表示する。

```
a=0.4;  
b=-0.4;  
c=-3;  
d=-1;  
f(x):=a*x^3+b*x^2+c*x+d;  
plot(f(x),size->2);  
r=roots([d,c,b,a]);  
forall(r,draw((#,0)))
```

18 音声出力 Syntherella

18.1 概要

Syntherella は、CindyScript による MIDI インターフェイスである。

音を出すには、中央の C を 60 として半音ずつプラス・マイナスし、`playtone(60)` とする。

音量は、これに `velocity` 修飾子を付ける。値は実数で 0 ~ 1。たとえば、
`playtone(60,velocity->0.5)`

楽器をチャンネルに割り当てるには、`midichannel(3, instrument->25)` あるいは
`instruments(25, channel->3)` のように記述する。

残響は音を鳴らす時間（秒）を指定する `duration` 修飾子を用いる。たとえば、
`playtone(60,duration->0.1,velocity->1)`。

タイミングを取るには、CindyScript の `wait(...)` 関数を使うか、MIDI シーケンサを使う。CindyScript でのシーケンサにアクセスするには `playmelody` 関数か `midiaddtrack, midistart` 関数を用いる。

トラックをシーケンサに追加するには、`midiaddtrack` 関数を用いる。追加されたトラックは `midistart` 関数によって有効になる。

18.2 単音

単音を鳴らす: `playtone(<整数>)`

単音を鳴らす。修飾子がないとき、初期設定の楽器音が使われ、鳴る時間は 1 秒で、音量は 0.5。

【例】 `playtone(72, duration->4);`

`bend` 修飾子によって、音程が半音単位で 2 半音まで上下できる。音の長さ `duration` が 0 以下のときは `stoptone()` で止められるまで鳴り続ける。そのほか、修飾子は次の通り

修飾子	型	効果
velocity	0.0 ... 1.0	音量（ピアノの鍵盤を叩くときの速度）
amp	0.0 ... 1.0	velocity と同じ
duration	<real>	音の長さ。秒単位。
channel	0..15	演奏するチャンネル
reverb	0.0 ... 1.0	残響効果
balance	- 1.0 ... 1.0	左右のバランス
bend	- 2.0 ... 2.0	半音の上げ下げ

音を止める: `stoptone(<整数>)`

特定のキーの音を止める。音が鳴り続けるときに有効。

特定の周波数で鳴らす: `playfrequency(<real>)`

この関数は引数を周波数 (Hz) (実数) で渡す。より物理的な `playsin` を使う方がよい場合もある。

修飾子は `playtone` と同様だが `bend` 修飾子はここでは無効である。

【例】

`playfrequency(440)` で 440 Hz の音を、現在選択されているチャンネルの楽器で鳴らす。`playfrequency` 関数は、インドのラーガやジャワのガムラン音楽のような非ヨーロッパ系の音階をシミュレートするときに使える。

18.3 旋律

旋律の演奏: `playmelody(<list>)`

`playmelody()` はリストをシーケンサに加えてすぐに演奏する。この関数が呼ばれるとき、他のトラックはシーケンサから削除される。リストは、音程と長さを表す整数か文字列、楽器や音量で構成される。リストの各要素は、音の高さと長さを表す。音の長さは拍で数えられる。初期状態では 1 分につき 60 の速さで演奏される。これは、`midispeed` 関数の修飾子で変更できる。

【例】ハ長調の音階

```
playmelody([["C",1],["D",1],["E",1],["F",1],["G",1],["A",1],["H",1],["c",5]])  
または
```

```
playmelody([[60,1],[62,1],[64,1],[65,1],[67,1],[69,1],[71,1],[72,5]])
```

単音ではなく和音を演奏することもできる。そのためには、同時に鳴らす音をリストにする。その形式は [[<音程 1>,<音程 2>,<音程 3>,...],<長さ>]。

次のコードは、次第に和音を構成する音を増やしながら演奏する。

```
playmelody([["C",1],[["C","E"],1],[["C","E","G"],1],[["C","E","G","c"],5]] )
```

最後に、-1か”P”または”p”を使うことで休符ができる。

ダイナミクスとフレージング

旋律を演奏するために、次のようなダイナミクス（音の強弱）やフレージングがある。それぞれ、その文字列だけのリストにする。

”ppp”: ピアノピアニッシモ

”pp”: ピアニッシモ

”p”: ピアノ

”mp”: メゾピアノ

”mf”: メゾフォルテ

”f”: フォルテ

”ff”: フォルテッシモ

”fff”: フォルテフォルテッシモ

”>”: アクセント

”staccato”: スタッカート

”st”: スタッカート

”legato”: レガート

”le”: レガート

”[“velocity”,<real>]: 音量を <real> にする。実数は 0.0 から 1.0

【例】次のコードは、音階の演奏をピアニッシモ・スタッカートで始めて、フォルテ・レガートで終わる。

```
playmelody([["st"],["pp"],["C",1],["D",1],["E",1],["F",1],["le"],["f"],  
["G",1],["A",1],["H",1],["c",5]])
```

位置合わせ

トラックが加えられたとき、シーケンサの演奏位置を示すポインタは 0 に設定される。

"goto" と "gorel" によって、ポインタは絶対位置か相対位置に移動できる。その構文は次の通り。

- ["goto", <real>]: ポインタを、最初から測って <real> の位置に置く。
非負の数とする。
- ["gt", <real>]: "goto" と同じ。
- ["gorel", <real>]: ポインタの位置を相対的に計算して <real> の位置に置く。
負の値も可だが、結果として負になるような数は禁じられている。
- ["gr", <real>]: "gorel" と同じ。
- ["||:"]: 繰り返し記号の始め。
- [":||"]: 繰り返し記号の終わり。
- ["1."]: 第1カッコ。
- ["2."]: 第2カッコ。

【例】

```
playmelody([
    ["c", .5], ["d", .5],
    ['$||$:', ["e", 1], ["g", 1], ["g", 1.5], ["a", .5], ["g", 1], ["e", 1], ["c", 1.5], ["d", .5],
     ["1."], ["e", 1], ["e", 1], ["d", 1], ["c", 1], ["d", 3], ["c", .5], ["d", .5], [":$||$"],
     ["2."], ["e", 1], ["e", 1], ["d", 1], ["d", 1], ["c", 4]
], speed->200
);
```

楽器のコントロール

- ["channel", <整数>]: チャンネルを (0...15) の範囲で変更する。
- ["ch", <整数>]: channel と同じ。
- ["instrument", <整数>]: チャンネルと関連して楽器を (1...128) の範囲で変更する。
- ["inst", <整数>]: instrument と同じ。
- [<整数>, <整数>, <整数>, <整数>]: 楽器のコントロールにアクセスする。
最初の数はコントロールコード、2番目はチャンネル、
との2つはデータ。
トラックが再開されるときは、楽器は初期値にリセットされる。

修飾子

修飾子	型	効果
channel	0..15	演奏するチャンネルの設定
instrument	1 ... 123	楽器の選択
speed	<real>	1分間あたりの拍数を設定
loop	0,1,2,3,4 ...	旋律の繰り返し
start	<real>	開始位置(拍数で)

修飾子 `loop` は演奏が終わったあとの巻き戻しを指定する。この修飾子に続く数は巻き戻しの回数。したがって、`loop->3` とすると、旋律は4回演奏される。`loop->-1` は無限ループと解釈される。これが呼ばれたあとは、新しい旋律（空の旋律 `playmelody([])` も可）を加えたときにループが終わりる。あるいは、`midistop()` 命令を使うこともできる。

シーケンサにトラックを追加する: `midiaddtrack(<list>)`

この関数はシーケンサに旋律を追加するがすぐに演奏はしない。シーケンサの開始は `midistart` 命令によって行われる。シーケンサが演奏を始めたら、トラックを追加してもシーケンサは演奏をやりなおさない。トラックはそのまま置き換えられる。さらに、シーケンサと比較して追加されたトラックのタイミングを変えていくいくつかの修飾子がある。

修飾子

修飾子	型	効果
channel	0..15	演奏するチャンネルを選ぶ
instrument	1 ... 123	楽器を選ぶ
speed	<real>	1分間あたりの拍数
track	0..10	シーケンサのトラックを選ぶ
start	<real>	開始位置(拍)
mode	<文字列>	add, replace, append の文字でトラックを追加したときの動作
stretch	<real>	旋律を追加したときの伸縮(シーケンサと関連)
offset	<real>	シーケンサと関連して、旋律を追加したときのシフト量
repeat	<real>	旋律がどのくらい追加されるか

シーケンサの開始: `midistart()`

シーケンサを開始する。修飾子は `playmelody` と同様で、`speed`, `loop`, `start` が使える。

シーケンサの停止: `midistop()`

MIDI シーケンサを直ちに停止する。

シーケンサの速度の設定: `midispeed(<real>)`

速度は 1 分あたりの拍数を実数で与える。初期値は 60 です。シーケンサがすでに動いていても速度を変えることができる。

シーケンサの速度を問い合わせる: `midispeed()`

現在のシーケンサの速度を返す。

シーケンサの位置を設定する: `midiposition(<real>)`

シーケンサの位置を特定の位置に設定する。

シーケンサの開始位置を問い合わせる: `midiposition()`

シーケンサの現在の位置を返す。シーケンサが動いているとき、この値は連続的に変化する。

18.4 音色

楽器を選ぶ: `instrument(<整数>)`

この関数によってチャンネルに楽器を割り当てる。チャンネルが指定されなければ初期値のチャンネルが使われる。各楽器には、1 から 128 までのコードが割り当てられている。楽器とコードの対応については、次の `instrumentnames` で説明する。修飾子によってチャンネルを指定すれば、そのチャンネルの楽器が変更される。

修飾子	型	効果
velocity	0.0 ... 1.0	音量 (ピアノの鍵盤を叩く速さ)
duration	<real>	音の長さ
bend	- 2.0 ... 2.0	半音単位で上げ下げ
channel	0..15	演奏するチャンネル
reverb	0.0 ... 1.0	残響効果
balance	- 1.0 ... 1.0	左右のバランス

利用できる楽器を知る: `instrumentnames()`

この関数は、使われているシピュータで利用できる楽器のリストを返す。標準 MIDI では次の楽器が使える。

Piano:			
1	Acoustic Grand Piano	33	Acoustic Bass
2	Bright Acoustic Piano	34	Electric Bass (finger)
3	Electric Grand Piano	35	Electric Bass (pick)
4	Honky-tonk Piano	36	Fretless Bass
5	Electric Piano 1	37	Slap Bass 1
6	Electric Piano 2	38	Slap Bass 2
7	Harpsichord	39	Synth Bass 1
8	Clavi	40	Synth Bass 2
Chromatic Percussion:			
9	Celesta	41	Violin
10	Glockenspiel	42	Viola
11	Music Box	43	Cello
12	Vibraphone	44	Contrabass
13	Marimba	45	Tremolo Strings
14	Xylophone	46	Pizzicato Strings
15	Tubular Bells	47	Orchestral Harp
16	Dulcimer	48	Timpani
Organ:			
17	Organ	49	String Ensemble 1
18	Percussive Organ	50	String Ensemble 2
19	Rock Organ	51	Synth Strings 1
20	Church Organ	52	Synth Strings 2
21	Reed Organ	53	Voice Aahs
22	Accordion	54	Voice Oohs
23	Harmonica	55	Synth Voice
24	Tango Accordion	56	Orchestra Hit
Guitar:			
25	Acoustic Guitar (nylon)	57	Trumpet
26	Acoustic Guitar (steel)	58	Trombone
27	Electric Guitar (jazz)	59	Tuba
28	Electric Guitar (clean)	60	Muted Trumpet
29	Electric Guitar (muted)	61	French horn
30	Overdriven Guitar	62	Brass Section
31	Distortion Guitar	63	Synth Brass 1
32	Guitar harmonics	64	Synth Brass 2
Bass:			
33	Acoustic Bass	65	Soprano Sax
34	Electric Bass (finger)	66	Alto Sax
35	Electric Bass (pick)	67	Tenor Sax
36	Fretless Bass	68	Baritone Sax
37	Slap Bass 1	69	Oboe
38	Slap Bass 2	70	English Horn
39	Synth Bass 1	71	Bassoon
40	Synth Bass 2	72	Clarinet
Reed:			
65	Soprano Sax	97	FX 1 (rain)
66	Alto Sax	98	FX 2 (soundtrack)
67	Tenor Sax	99	FX 3 (crystal)
68	Baritone Sax	100	FX 4 (atmosphere)
69	Oboe	101	FX 5 (brightness)
70	English Horn	102	FX 6 (goblins)
71	Bassoon	103	FX 7 (echoes)
72	Clarinet	104	FX 8 (sci-fi)
Synth Effects:			
97	FX 1 (rain)	105	Sitar
98	FX 2 (soundtrack)	106	Banjo
99	FX 3 (crystal)	107	Shamisen
100	FX 4 (atmosphere)	108	Koto
101	FX 5 (brightness)	109	Kalimba
102	FX 6 (goblins)	110	Bagpipe
103	FX 7 (echoes)	111	Fiddle
104	FX 8 (sci-fi)	112	Shanai
Ethnic:			
105	Sitar	113	Tinkle Bell
106	Banjo	114	Agogo Bells
107	Shamisen	115	Steel Drums
108	Koto	116	Woodblock
109	Kalimba	117	Taiko Drum
110	Bagpipe	118	Melodic Tom
111	Fiddle	119	Synth Drum
112	Shanai	120	Reverse Cymbal
Percussive:			
113	Tinkle Bell	121	Guitar Fret Noise
114	Agogo Bells	122	Breath Noise
115	Steel Drums	123	Seashore
116	Woodblock	124	Bird Tweet
117	Taiko Drum	125	Telephone Ring
118	Melodic Tom	126	Helicopter
119	Synth Drum	127	Applause
120	Reverse Cymbal	128	Gunshot
Sound effects:			
121	Guitar Fret Noise	122	Breath Noise
123	Seashore	124	Bird Tweet
125	Telephone Ring	126	Helicopter
127	Applause	128	Gunshot

チャンネルを選ぶ: `midichannel(<int>)`

この関数は、`playmelody` あるいは `playtone` で使うチャンネルを選ぶ。修飾子によって、選択したチャンネルの楽器や音色を変えることができる。

修飾子	型	効果
velocity	0.0 ... 1.0	音量 (ピアノの鍵盤を叩く速さ)
duration	<real>	音の長さ
bend	- 2.0 ... 2.0	半音単位での音の上げ下げ
instrument	0..15	演奏する楽器の指定
reverb	0.0 ... 1.0	残響効果
balance	- 1.0 ... 1.0	左右のバランス

チャンネルの音量を設定する: `midivolume(<real>)`

引数は 0.0 から 1.0 までの実数。通常は初期値のチャンネルが対象になるが、修飾子によってチャンネルは変更できる。

修飾子は、`channel` で 0~15 (チャンネル番号) あるいは "all" (すべてのチャンネル)

チャンネルのコントローラを設定する: `midicontrol(<整数>, <整数>)`

第 1 の引数はコントローラの番号 (0..127) で、第 2 の引数はコントローラにセットする値 (0..127)。コントローラは、残響効果やバランス、楽器に特有の音に影響する。修飾子は、`midivolume` と同じ。

18.5 サウンド関数

周波数を指定して音を鳴らす: `playsin(<real>)`

この関数では一定の振幅で 1 秒間音を鳴らします。速さや振幅などは修飾子で指定できる。

修飾子	型	効果
amp	0.0 ... 1.0	全体の振幅
damp	<real>	指数関数的に減衰する
harmonics	<list>	音のスペクトル
duration	<real>	鳴らす速さ
stop	<real>	duration と同じ
line	数または文字	音を関連付けるライン

【例】

```
playsin(440,damp->3,stop->5)
```

```
playsin(440,damp->3,stop->5,harmoics->[0.5,0.3,0.2,0.1])
```

フーリエ級数によって指数関数的に減衰する音を鳴らす。

```
0.5*sin(440*2*pi*x)+0.3*sin(2*440*2*pi*x)+0.2*sin(3*440*2*pi*x)
```

```
+0.1*sin(4*440*2*pi*x)
```

関数によって定義された音を鳴らす: `playfunction(<funct>)`

波形を入力して音を鳴らす。時間単位は関数の 1 単位が 1 秒にあたる。

修飾子	型	効果
amp	0.0 ... 1.0	全体の振幅
damp	<real>	指数関数的に減衰する
start	<real>	開始位置
stop	<real>	終了位置
duration	<real>	鳴らす速さ
mode	文字列	append または replace で新しい音のハンドル
line	数または文字	音を関連付けるライン
silent	<bool>	発音しない
export	<bool>	サンプルデータの書き出し

【例】440Hz の音 `playfunction(sin(440*x*pi*2));`

減衰する雑音 `playfunction(random(),damp->8);`

1 秒間の正弦波 `playfunction(sin(1000*x*2*pi),stop->1/1000,duration->1)`

正弦波の 4 4 の標本点のリストを作成し、再生する。

```
sample=playfunction(sin(1000*x*2*pi),stop->1/1000,silent->true,export->true);
playwave(sample,duration->1);
```

オーディオデータのリストを再生する: `playwave(<list>)`

この関数は、オーディオデータのリストを再生する。データの値は - 1.0 から 1.0 の範囲にあるものとする。サンプリングレートは 44100 Hz で、サンプル音は 1 秒間再生される。再生時間は `duration` 修飾子で指定できる。修飾子は、`amp`, `damp`, `duration`, `line` (`playfunction` の修飾子参照)

【例】次のコードでは、`playwave` で再生する 3 つのデータを作る。`wait` 関数によって、3 つの音が同じラインを使うのに時間差を設ける。`playwave` 関数を使う前に、オーディオデータの作成は完了している。

```
sample0=apply(1..200,sin(\verb|#|*2*pi/200));
sample1=apply(1..100,sin(\verb|#|*2*pi/100));
sample2=apply(1..50,sin(\verb|#|*2*pi/50));
```

```
playwave(sample0,duration->1,line->1);
wait(400);
playwave(sample1,duration->1,line->1);
wait(400);
playwave(sample2,duration->1,line->1);
```

再生の停止: `stopsound()`

すべての音の再生を停止する。

19 ファイル管理

19.1 データの読み込み

ディレクトリを設定する: `setdirectory(<文字列>)`

これ以降のファイルのディレクトリを設定する。ファイルの読み書きをするときは、対象となるディレクトリを設定してから行なう。設定しない場合はエラーとなる。

【例】

Windows : `setdirectory(C:¥Users¥ユーザー名¥Desktop);`

Macintosh : `setdirectory(" /Users/ユーザー名/Desktop");`

データの読み込み: `load(<文字列>)`

この関数の引数 はファイル名とする。ディレクトリ名を含んでもよい。ファイル名が正しければ、ファイルに含まれる全情報が文字列として返される。データは `setdirectory` で指定されたディレクトリから読まれる。

プログラムコードの読み込み: `import(<文字列>)`

この関数の引数 はファイル名とする。ディレクトリ名を含めることはできない。ファイルの内容は CindyScript のコードとして解析され、実行される。こうして、あらかじめ定義された関数ライブラリを読み込むことができる。`import` 関数は CindyScript の "Initialization" スロットに記述する。

19.2 データの書き出し

テキストデータの書き出しが次の手順で行われる。

1. `setdirectory` 関数を用いてディレクトリを指定する。
2. ファイルを開く

3. ファイルに書く
4. ファイルを閉じる

ファイルを開く: `openfile(<文字列>)`

指定した名前のファイルを開く。戻り値は書き出しに必要な<ファイル名>。

ファイルに書いて改行する: `println(<ファイル名>,<文字列>)`

`println` と同様だが、<ファイル名>で指定したファイルに書き出す。

ファイルに書く: `print(<ファイル名>,<文字列>)`

`print` と同様だが <ファイル名> で指定したファイルに書き出す。

ファイルを閉じる: `closefile(<ファイル名>)`

ファイルを閉じる。引数のファイル名は、`openfile()` の戻り値。

【例】次の例は、ファイルに書き出す一連の手順を示す。

```
setdirectory("path");
f=openfile("myFile");
println(f,"ここにいくつかの数があります。");
forall(1..15,print(f,\verb|#|+" "));
println(f,"");
closefile(f);
```

カレントディレクトリの取得

Cinderella のファイルを開いたとき、そのファイルのあるディレクトリが、変数 `loaddir` に入る。たとえば、デスクトップに置いたファイルを開き、

```
println(loaddir)
を実行すると、コンソールに
/Users/ユーザー名/Desktop/
と表示される。
```

【注】この内容は、オリジナルのマニュアルには記述されていない。

19.3 HTML との連携

Web ページを開く: `openurl(<文字列>)`

ブラウザを立ち上げ、<文字列>の Web ページを開く。ブラウザは、通常で使っているもの。

javascript を呼び出す: javascript(<文字列>)

書き出されたアプレットにおいて、ブラウザの Javascript 環境での命令を呼び出す。この命令の内容は <文字列> に書く。スタンダードアロンのアプリケーションでは何もしない。

【例】次のコードでは、ブラウザ上にポップアップウィンドウを表示してメッセージを出す。

```
javascript("alert('Hi from Cinderella!!!')");
```

19.4 ネットワーク

TCP ポートを開く: openconnection(<文字列>,<int>)

第1引数で指定されたサーバーへの TCP 接続を行い、第2引数で指定されたポートを開く。戻り値は、このネットワーク接続へのハンドル。

TCP へ書きだす: print(<handle>,<文字列>)

TCP へ書きだす: println(<handle>,<文字列>)

print と println 関数は、ファイルだけでなく openconnection によって作られるネットワークへも書き出しを行なう。

TCP ポートへの出力と消去: flush(<handle>)

与えられた接続への出力バッファを消去する。

TCP 接続からの読み込み: readln(<handle>)

与えられた接続から行を読む。もしデータがなければ 5 秒後にタイムアウトする。

TCP 接続を閉じる: closeconnection(<handle>)

与えられたハンドルの接続を閉じる。

【例】次のコードでは、Web サーバを開いて、そこから HTML コードを読み出す。

```
x=openconnection("cermat.org",80);
println(x,"GET /");
y="";
while(!isundefined(y),y=readln(x);println(y));
closeconnection(x);
```

19.5 コンソールへの出力

テキストの印字: `print(<expr>)`

この関数は を評価した結果をコンソールに表示する。

テキストの印字: `err(<expr>)`

この関数は, `<expr>` を評価した結果をコンソールに表示する。`<expr>` が変数であれば変数の値が表示される。

テキストの印字と改行: `println(<expr>)`

この関数は `<expr>` を評価して画面に表示して改行する。引数がなければ改行だけ行う。

画面クリア: `clearconsole()`

コンソール画面からすべての文字を消去してクリアする。

条件の印字: `assert(<bool>,<expr>)`

この関数は, エラーメッセージを作つて表示するのに用いられる。この関数は `if(<bool>,println(<expr>))` を実行するのと同じ。条件が満たされるかどうかを調べて, エラーメッセージを作成する。

【例】`assert(isinteger(k),"k is not an integer");`

ステータス行に表示: `message(<expr>)`

この関数は, ステータス行 (ツールバーの下), または, Cinderella・アプレットのためのブラウザのステータス行で`<expr>` を評価した結果を表示する。

20 時間とアニメーション

20.1 時間

time 時刻にアクセスする: `time()`

この関数は4つの整数からなるリスト `[h,m,s,ms]` を返す。4つの数はコンピュータの時計で "hour," "minute," "second," "millisecond" に相当する。

日にアクセスする: `date()`

この関数は3つの整数からなるリスト `[y,m,d]` を返す。3つの数は, コンピュータのカレンダーの "year," "month," "day" に相当する。

時刻表示: seconds()

この関数は、`resetclock()` の実行後に経過した時間を返す。時間は 1 単位が 1 秒になるよう調整される。時計の精度はミリ秒。

内部時計のリセット: resetclock()

`seconds` 関数の値をリセットする。

待ち時間: wait(<real>)

パラメータで与えられたミリ秒単位ですべてのスクリプトの実行を止める。

20.2 アニメーションのコントロール

アニメーションを始める: playanimation()

アニメーションを始める。CindyLab の物理シミュレーションも同様。

アニメーションの一時停止: pauseanimation()

アニメーションを一時停止する。

アニメーション停止: stopanimation()

アニメーションを停止する。幾何学要素などの状態は初めの状態に戻る。

21 ユーザー入力

Cindyscript により、マウスやキーボードによる入力ができる。そのとき、 "Mouse Down," "Mouse Up," "Mouse Click," "Mouse Drag," "Key Typed" の各スロットを使う。また、加速度センサが搭載されていればこれを利用することができる。

21.1 マウスとキーボード

マウスの位置: mouse()

マウスボタンが押されたとき、マウスの位置を表すベクトルを返す。ベクトルは同次座標。(無限遠点も考慮する) 2 次元のユークリッド座標が必要であれば、`mouse().xy` によって座標が得られる。

キー入力: key()

キーボードで打たれた文字列を返す。

あるキーが押されたか: iskeydown(<整数>)

この関数は、ある特定のキーが押されたならばブール値 true を返す。そのキーは整数の引数で与える。キーの番号は、たとえば 65, 66, 66, が 'A', 'B', 'C',... に対応する。'shift', 'crtl' と 'alt' は 16, 17, 18 である。

押されたキーのリスト: keydownlist()

この関数は、押されたキーのリストを返す。

21.2 加速度センサにおける AMS データ

AMS データの取得: amsdata()

この関数は AMS センサの生のデータを取得する。

調整された AMS データの取得: calibratedamsdata()

この関数は、AMS センサの調整されたデータを取得する。調整されたデータは、空間におけるパソコンの位置をベクトルデータとしたもの。

22 CindyLab との連携

22.1 シミュレーション環境

simulation()

この関数はシミュレーション環境へのハンドルを与える。この関数で、たとえば、運動や位置エネルギーにアクセスできる。

- friction シミュレーション全体の摩擦力 (実数, 読み書き可)
- gravity シミュレーション全体の重力 (実数, 読み書き可)
- kinetic シミュレーション全体の運動エネルギー (実数, 読み出しのみ)
- ke 同上 (実数, 読み出しのみ)
- potential シミュレーション全体の位置エネルギー (実数, 読み出しのみ)
- pe 同上 (実数, 読み出しのみ)

力の適用: addforce(<質量>,<vector>)

力を存在する質点に適用する。これは、Integration Tick スロットに置かなければならない。

力の設定: setforce(<質量>,<vector>)

力を存在する質点に設定する。これは、Integration Tick スロットに置かなければならない。

力の探索粒子: force(<vector>)

この関数は、特定の場所で粒子に及ぼす力をテストするのに使うことができる。ベクトル

は位置を表す。関数はこの位置での力を 2 次元ベクトルで返す。修飾子が使われないならば、探針となる粒子は、質量が 1, 電荷が 1, 半径が 1 であるとみなされる。

【注】もとのマニュアルには、ここに例が載っているが、仕様変更のためか、最近の版ではうまく動かないようなので割愛する。

修飾子: 質量と電荷、半径を明示することも可能。これらはその名前の修飾子で設定される。これらのうち少なくとも 1 つが設定されたならば、残りのものは 0 に設定される。したがって、`force((0,0),charge->2)` は、電荷 2, 質量 0, 半径 0 の粒子で、点 `point [0,0]` の力をテストする。

23 Cindy3D

Cinderella でレイトレーシングを用いた空間図形を描くには、同梱されている Cindy3D プラグインを用いる。Initailiazarion スロットの先頭に次の 1 行を書く。

```
use("Cindy3D");
```

これで Cindy3D の各関数が使えるようになる。スクリプトを書いて実行すると別ウィンドウが開いて描画がされる。

【重要な注意】

Cindy3D は、ファイルメニューの「HTML に書き出す」で HTML ファイルに書き出しても CindyJS では使えない。CindyJS で 3D を扱うには、HTML ファイルを直接編集する必要がある。

23.1 設定

以下の関数は初期値の設定なので、最初に一度だけ実行すればよく、Initialization スロットに `use("Cindy3D");` に続いて記述すればよい。

色の初期設定 : `color3d([R,G,B])`

すべてのオブジェクトの表示色の初期値を RGB 値に設定する。

【例】 `color3d([0.8,0.8,0]);` 表示色を少し暗い黄色に設定する。

点の色の初期設定 : `pointcolor3d([R,G,B])`

点の表示色の初期値を RGB 値に設定する。

線の色の初期設定 : `linecolor3d([R,G,B])`

線の表示色の初期値を RGB 値に設定する。

面の色の初期設定 : `surfacecolor3d([R,G,B])`

面の表示色の初期値を RGB 値に設定する。

透明度の初期設定 : `alpha3d(<real>)` または `surfacealpha3d(<real>)`

面の透明度の初期値を設定する。引数は 0 以上 1 以下の実数。

光沢の初期設定 : shininess3d(<real>)

すべてのオブジェクトの光沢の初期値を<real>に変更する。

点の光沢の初期設定 : pointshininess3d(<real>)

点の光沢の初期値を<real>に変更する。

線の光沢の初期設定 : lineshininess3d(<real>)

線の光沢の初期値を<real>に変更する。

面の光沢の初期設定 : surfaceshininess3d(<real>)

面の光沢の初期値を<real>に変更する。

サイズの初期設定 : size3d(<real>)

点と線のサイズの初期値を<real>に変更する。

点のサイズの初期設定 : pointsize3d(<real>)

点のサイズの初期値を<real>に変更する。

線の太さの初期設定 : linesize3d(<real>)

線のサイズの初期値を<real>に変更する。

23.2 描画関数

以下の関数は Draw スロットに記述する。

Cindy3D の描画開始 : begin3d()

Cindy3D の描画終了 : end3d()

Cindy3D の描画関数の使い始めと使い終わりを宣言する。Cindy3D の描画関数は、begin3d() から end3d() までの間に書かれたものが実行される。

現在の描画設定の保存 : gsave3d()

現在の描画に関する諸設定をスタックに保存する。

描画設定の復帰 : grestore3d()

スタックに保存された描画に関する諸設定を呼び出して、その状態に復帰する。

点を描く : draw3d(<point>)

座標 <point> に点を打つ。

修飾子	値	効果
size	< real >	点の大きさを指定する
color	[< real >, < real >, < real >]	色を RGB で指定する
shininess	< real >	光沢を指定する

線を描く : draw3d(<point1>,<point2>)

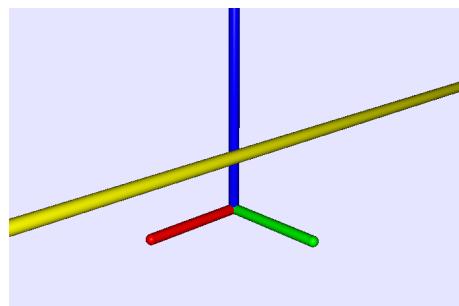
線分, 反直線, 直線を描く。線の種類は type 修飾子で指定する。指定がなければ線分が描かれる。2つの引数は、線の種類に応じて解釈される。

線種	point1	point2
線分	始めの端点	終わりの端点
反直線	始点	通る点
直線	直線上の点	直線上のもう一つの点

修飾子	値	効果
type	< string >	segment,ray,line のいずれか
size	< real >	線の太さを指定する
color	[< real >, < real >, < real >]	線の色を RGB 値でを指定する
shininess	< real >	光沢を指定する

【例】次の例は 3 種類の線を描画する。

- (0,0,0) と (1,0,0) を端点とする線分
`draw3d([0,0,0],[1,0,0],color->[1,0,0])`
- (0,0,0) と (0,1,0) を端点とする緑色の線分
`draw3d([0,0,0],[0,1,0],type->"segment",color->[0,1,0])`
- (0,0,0) を始点として、(0,0,1) を通る青の半直線
`draw3d([0,0,0],[0,0,1],type->"ray",color->[0,0,1])`
- (1,1,1) と (2,1,1) を通る黄色の直線
`draw3d([1,1,1],[2,1,1],type->"line",color->[1,1,0])`



点を結ぶ : connect3d(<list>)

<list>で与えられた各点を線分で結ぶ。

修飾子	値	効果
size	< real >	線の太さを指定する
color	[< real >, < real >, < real >]	色を RGB で指定する
shininess	< real >	光沢を指定する

多角形を描く : drawpoly3d(<list>)

<list>で与えられた各点を線分で結んで多角形を描く。

修飾子	値	効果
size	< real >	線の太さを指定する
color	[< real >, < real >, < real >]	色を RGB で指定する
shininess	< real >	光沢を指定する

多角形の面を描く : fillpoly3d(<list>)

<list>で与えられた各点を線分で結んで多角形の面を描く。

修飾子	値	効果
size	< real >	面の大きさを指定する
color	[< real >, < real >, < real >]	色を RGB で指定する
shininess	< real >	光沢を指定する
alpha	< real >	透明度を指定する

法線ベクトルを指定して多角形の面を描く : fillpoly3d(<list1>, <list2>)

ユーザー定義の法線ベクトルによる多角形の面を描く。法線ベクトルは、光の当たり方を計算するものである。

<list1>は多角形の頂点の座標、<list2>は多角形の各頂点の法線ベクトル。<list1>と<list2>の長さは一致する必要がある。

修飾子	値	効果
size	< real >	面の大きさを指定する
color	[< real >, < real >, < real >]	色を RGB で指定する
shininess	< real >	光沢を指定する
alpha	< real >	透明度を指定する

円盤を描く : fillcircle3d(<point>, <vec>, <real>)

<point>を中心、<vec>を法線ベクトル、<real>を半径とする円盤を描く。

修飾子	値	効果
size	< real >	面の大きさを指定する
color	[< real >, < real >, < real >]	色を RGB で指定する
shininess	< real >	光沢を指定する
alpha	< real >	透明度を指定する

【例】座標軸と円盤を描く。

Initialization スロットに次のコードを書く。以後の例も同様。

```
use("Cindy3D");
background3d([0.9,0.9,1]);
renderhints3d(quality->4);
lookat3d([4,4,4],[0,0,0],[-1,-1,0]);
```

draw スロットに次のコードを書く。

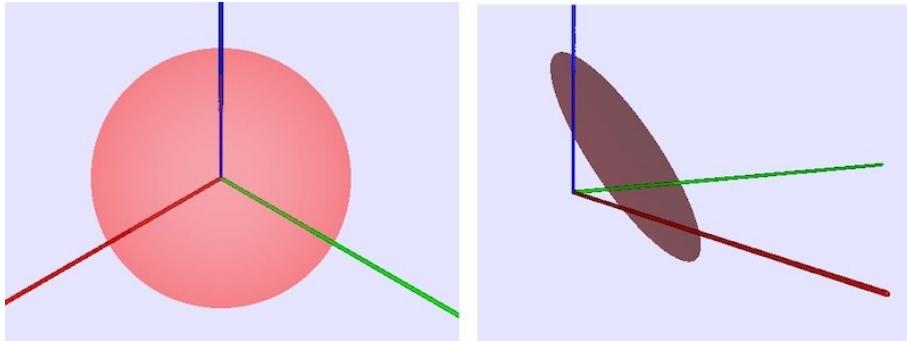
```
renderhints3d(quality->4);
begin3d();
draw3d([0,0,0],[3,0,0],color->[1,0,0],size->0.3); // 座標軸
```

```

draw3d([0,0,0],[0,3,0],color->[0,1,0],size->0.3);
draw3d([0,0,0],[0,0,3],color->[0,0,1],size->0.3);
fillcircle3d([sqrt(3)/6,sqrt(3)/6,sqrt(3)/6],[1,1,1],
  sqrt(3)/2,color->[1,0,0],alpha->0.3);
end3d()

```

`renderhints3d(quality->4)` はレンダリングの品質を指定する関数。(後述)
左が実行結果。右はマウスで画面上をドラッグし、回転したもの。



球面を描く : `drawsphere3d(<point>, <real>)`

`<point>`を中心、`<real>`を半径とする球面を描く。

修飾子	値	効果
<code>size</code>	<code>< real ></code>	面の大きさを指定する
<code>color</code>	<code>[< real >, < real >, < real >]</code>	色を RGB で指定する
<code>shininess</code>	<code>< real ></code>	光沢を指定する
<code>alpha</code>	<code>< real ></code>	透明度を指定する

網目上の曲面を描く : `mesh3d(<int1>, <int2>, <list>)`

曲面を、`m` 行 `n` 列の格子点でできる網目状に区切る。(メッシュモデル)

`<int1>` 格子点の行数 `m`

`<int2>` 格子点の列数 `n`

`<list>` 格子点のリスト

リストは 2 次元の格子点のリストを平坦化したもので、リストの長さは $m \times n$ 。

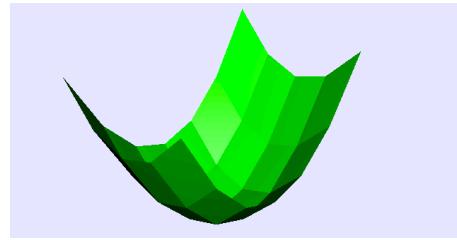
【例】図は、5行7列の格子点からなる放物線状の面。修飾子の効果を比較するため、かなり荒い網目にしてある。

```

begin3d();
pt=apply(-2..2,s,
         apply(-3..3,t,
               y = s/3;
               x = t/3;
               z = x^2+y^2;
               (x,y,z);
         );
);
pt=flatten(pt,levels->1);
mesh3d(5,7,pt);
end3d()

```

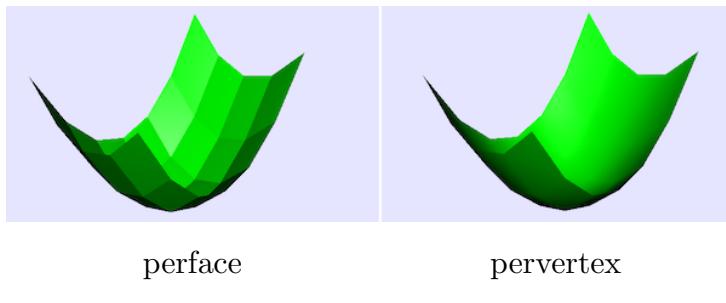
apply 関数のネストにより作成した格子点のリストを、 flatten() により平坦化して引数に与えている。



修飾子	値	効果
normaltype	< string >	法線のタイプを指定する。値は "perface" , "pervertex"
topology	< string >	topology を指定する。値は "open", "closerows", "closecolumns", "closeboth" のいずれか
size	< real >	面の大きさを指定する
color	[< real >, < real >, < real >]	色を RGB で指定する
shininess	< real >	光沢を指定する
alpha	< real >	透明度を指定する

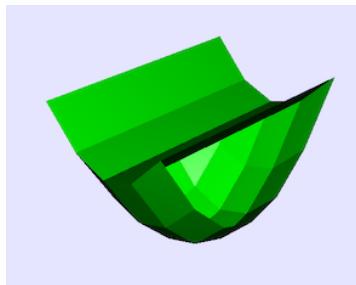
「normaltype」修飾子は、各面の法線の計算方法を指定する。指定がなければ「perface」として処理される。

nomaltytype	説明
perface	各面の法線はその面上の三角形の法線。その結果、面の端で光の当たり方が不連続になり、格子構造が明らかになる。
pervertex	各面の法線はその面上の三角形の頂点の3本の法線をとり、その一次結合によって計算される。その結果、面の端での光の当たり方が連続的になり、格子構造が見えなくなる。

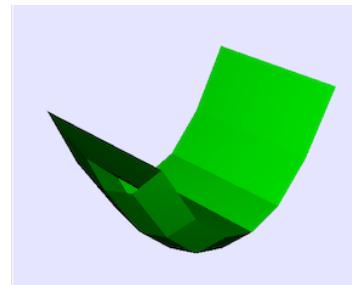


topology 修飾子は面の端の状態を指定する。指定がなければ、openとして処理される。

topology	説明
open	端点もしくは面の端の点までが面になる。その結果、 $(m-1) \times (n-1)$ 個の矩形ができる。面は両サイドと1つの境界を持つ。
closerows	各行の最初と最後の頂点が結合されて対応する面ができる。その結果、 $(m-1) \times n$ 個の矩形ができる。面は両サイドと2つの境界を持つ。
closecolumns	各列の最初と最後の頂点が結合されて対応する面ができる。その結果、 $m \times (n-1)$ 個の矩形ができる。面は両サイドと2つの境界を持つ。
closeboth	各行の最初と最後および各列の最初と最後の頂点が結合されて対応する面ができる。その結果、 $m \times n$ 個の矩形ができる。面は両サイドを持ち境界はない。



closerows



closecolumns

法線ベクトルを指定して網目を描く : `mesh3d(<int1>, <int2>, <list1>, <list2>)`

ユーザー定義による法線ベクトルによって網目を描く。

`<int1>` 格子点の行数 : m `<int2>` : 格子点の列数 n `<list1>` : 格子点のリスト

`<list2>` 各格子点での法線ベクトルのリスト。

リストの長さはいずれも $m \times n$ 。

修飾子	値	効果
topology	<code>< string ></code>	<code>topology</code> を指定する。 値は open, closerows, closecolumns, closeboth のいずれか
size	<code>< real ></code>	面の大きさを指定する
color	<code>[< real >, < real >, < real >]</code>	色を RGB で指定する
shininess	<code>< real ></code>	光沢を指定する
alpha	<code>< real ></code>	透明度を指定する

23.3 光の当て方と表現

背景の色を設定する : `background3d(<colorvec>)`

背景の色 `<colorvec>` を RGB 値で設定する。

カメラの位置 : `lookat3d(<point1>, <point2>, <vec>)`

3 D グラフィクスの表示は、空間に置いたカメラで物体を写していると考える。この関数は、カメラの位置と向きを設定する。カメラにはレンズがついていて、レンズの向き（視線の方向）で物を見ていると考える。

`<point1>` : カメラの位置 `<point2>` : 回転の中心 `<vec>` : 視線の方向

画角の設定 : fieldofview3d(<real>)

カメラの画角を設定する。実際のカメラの画角と同じ。画角が小さいと望遠（被写体が大きく写る）、大きいと広角（小さく写る）になる。初期設定は 45° 。

カメラ深度の設定 : depthrange3d(<real1>, <real2>)

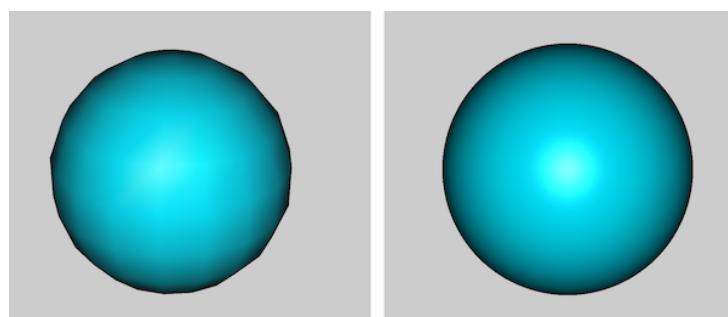
カメラの最小・最大深度を設定する。カメラの深度とは、カメラ面（カメラを通り、視線方向に垂直な面）と点の距離。カメラ深度に入らないものは表示されない。第 1 引数が最小値、第 2 引数が最大値。

レンダリングのヒントを設定する : renderhints3d()

レンダリングの過程における様々な比率のヒントを設定する。

修飾子	値	効果
quality	< int >	品質レベルを選ぶ。値は 0 以上 8 以下
renderMode	< string >	レンダリングモードを指定する。 値は "simple" か "raycated"
sampleRate	< int >	ピクセルごとのサンプル数を設定。1 以上の整数。
screenError	< real >	ピクセルにおける最大の screen space error を設定する。値は 0 より大きな実数

”quality” 修飾子は規定の品質レベルのいずれかを選ぶ。レベル 0 は最低の品質だが、最小のリソースですむ。最大の品質は 8 で、非常によい品質だが多くのリソースを必要とする。あらかじめ品質レベルが設定されているのは、個々のレンダリングヒントを操作することなく、全体の品質を簡単に管理できるようにするため。要請された品質レベルがサポートされない（例えばハードウェアの制限やリソースの制約のため）とき、Cindy3D は下位のレベルに移行するかもしれない。



quality->1

quality->4

”renderMode” 修飾子は、オブジェクトのレンダリングについて指定する。”simple” の場合は、すべてのオブジェクトは三角形の網目としてレンダリングされる。このモードではシェーディングは頂点ごとに行われ、上図左のように粗削りになる。”raycasted” の場合は、点・直線・球面はレイ・キャスティングを用いた連続面としてレンダリングされる。また、シェーディングは点ごとに行われる。上図右のようになる。”raycasted” モードでは高品質が得られるがハードウェアの条件によっては時間がかかる。

”screenError” 修飾子は、スクリーン・スペース・エラーをデティール・アルゴリズムのレベルに設定する。”simple” モードでは、点・直線・球面は三角形の網目で近似される。最適化のために、小さい、あるいは遠いオブジェクトはレンダリング時間を節約するために少しの三角形網目にする。これを”level of detail”と呼ぶ。Cindy3D は、それぞれのプリミティブに対して、異なる三角形網目ごとに決められた定数を用いている。あるプリミティブに対しどの定数を用いるかは、各網目をスクリーン上に仮想的に投影し、ピクセルごとに最大の三角形の大きさを計測して決定される。それから、最大と予測される三角形サイズが「screenError」の下にある最も小さな網目が、プリミティブを生成するために使われる。これは、「screenError」の低い値がより高い品質となることを意味する。この修飾子は”raycasted” レンダリングモードのもとでは無効になる。

”samplingRate” 修飾子は、オブジェクトのシルエットの滑らかさに影響する。サンプリング・レートは、出力イメージの各々のピクセルに対するサンプルの数を定める。最終的なピクセルの色はそれらのサンプルの平均値。サンプリングレートが高いほど、記憶領域と時間を消費するがオブジェクトシルエットはより滑らかになる。要請されたサンプリング・レートをサポートできないとき（例えばハードウェアの制限やリソースの制約のための）、Cindy3D は低いサンプリング・レートに移行するかもしれない。

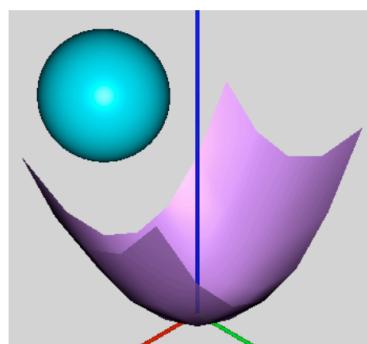
※訳者の実験では、品質は quality の例で示した 2 通りくらいで、数値を変えてあまり変化はなかった。（ハードウェアの制限などによるかもしれない）renderMode の”simple”，”raycasted” の違いも同様。あの 2 つの修飾子の効果については翻訳時点では不明だった。デフォルトでは quality->1 なので、renderhints3d(quality->4) もしくは renderhints3d(renderMode->”raycasted”) で運用するのがよさそうだ。

点光源の設定 : pointlight3d(<int>)

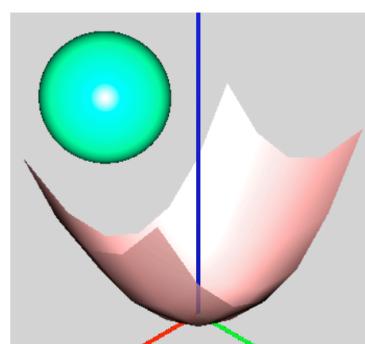
<int> は、光源の番号で 0 以上 7 以下の整数。

点光源を発生または修正する。指定された光源がすでに存在するならば修飾子によって指定された状態に修正し、利用可能にする。そうでなければ、指定された点光源を作る。修飾子がなければ初期値が使われる。点光源は 8 つまで作ることができ、それぞれに番号を振る。

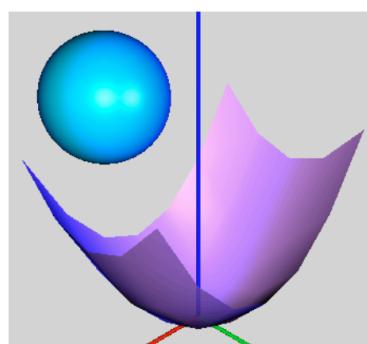
修飾子	値	効果
ambient	[R,G,B]	周囲の色を RGB 値で指定された色にする (初期値は [0,0,0])
diffuse	[R,G,B]	拡散する光の色を RGB 値で指定された色にする (初期値は [1,1,1])
specular	[R,G,B]	反射光の色を RGB 値で指定された色にする (初期値は [1,1,1])
position	< point >	点の位置 (初期値は [0,0,0])
frame	< string >	位置がカメラフレームに依存するか、絶対位置 かを指定する。値は”camera” か ”world” で、 初期値は ”camera”



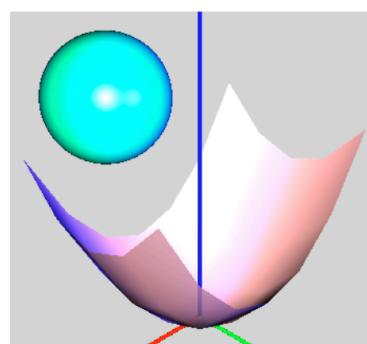
点光源の指定なし



点光源 1 diffuse->[1,1,0]



点光源 2 position->[8,0,0], diffuse->[0,0,1])



点光源 1 と 2

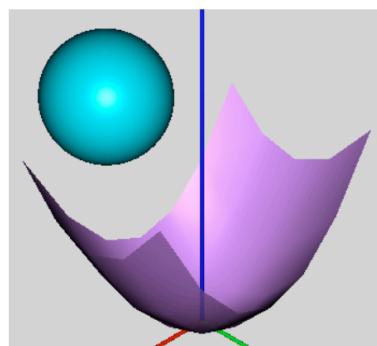
方向光源の設定 : directionallight3d(<int>)

<int>は、方向光源の番号で 0 以上 7 以下の整数。

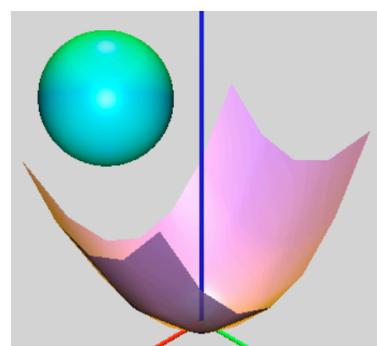
方向光源を発生または修正する。指定された方向光源がすでに存在するならば修飾子に

よって指定された状態に修正し、利用可能にする。そうでなければ、指定された方向光源を作る。修飾子がなければ初期値が使われる。

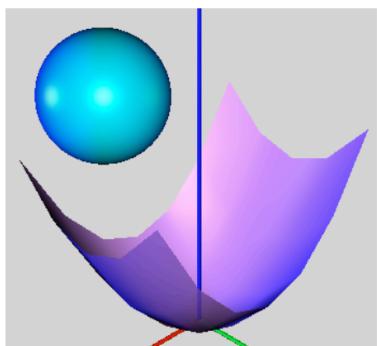
修飾子	値	効果
ambient	[R,G,B]	周囲の色を RGB 値で指定された色にする (初期値は [0,0,0])
diffuse	[R,G,B]	拡散する光の色を RGB 値で指定された色にする (初期値は [1,1,1])
specular	[R,G,B]	反射光の色を RGB 値で指定された色にする (初期値は [1,1,1])
direction	< vec >	光の方向 (初期値は [0,-1,0])
frame	< string >	方向がカメラフレームに依存するか、絶対的か を指定する。値は”camera” か ”world” で、初 期値は ”camera”



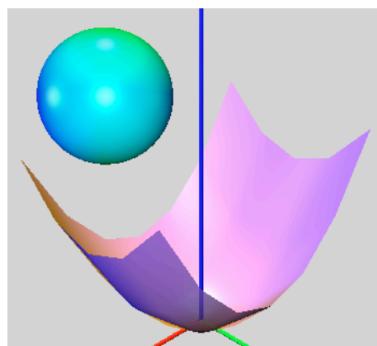
方向光源の指定なし



方向光源 1 diffuse->[1,1,0]



方向光源 2 direction->[8,0,0],diffuse->[0,0,1])



方向光源 1 と 2

スポットライトの設定 : spotlight3d(<int>)

<int>は、光源の番号で 0 以上 7 以下の整数。

スポットライトを発生または修正する。指定された光源がすでに存在するならば修飾子によって指定された状態に修正し、利用可能にする。そうでなければ、指定された光源を作る。修飾子がなければ初期値が使われる。

修飾子	値	効果
ambient	[R,G,B]	周囲の色を RGB 値で指定された色にする (初期値は [0,0,0])
diffuse	[R,G,B]	拡散する光の色を RGB 値で指定された色にする (初期値は [1,1,1])
specular	[R,G,B]	反射光の色を RGB 値で指定された色にする (初期値は [1,1,1])
position	< point >	点の位置 (初期値は [0,0,0])
direction	< vec >	光の方向 (初期値は [0,-1,0])
cutoffAngle	< real >	スポットコーンのカットオフ角。ラジアンで指定。0 から $\frac{\pi}{2}$ 初期値は $\frac{\pi}{4}$
exponent	< real >	減衰指数。値は 0 以上 128 未満。初期値は 0
frame	< string >	方向がカメラフレームに依存するか、絶対位置かを指定する。値は”camera” か ”world” で、初期値は ”camera”

光源を無効にする : disablelight3d(<int>)

<int>は、光源の番号で 0 以上 7 以下の整数。

与えられた番号の光源を無効にする。

24 CindyJS

CindyJS コンテンツを作成するには、Cinderella でコンテンツを作成してファイルメニューから「HTML に書き出す」か、直接 HTML ファイルをテキストエディタなどで編集する。ただし、Cinderella とは完全互換ではないので、Cinderella で作ったものがそのまま HTML で動くとは限らない。また、その逆もある。

24.1 CindyJS の HTML 基本構造

HTML ファイルをテキストエディタなどで編集する場合の最小構造は次の通りである。

```
<!DOCTYPE html>
<html>
  <head>
    <script type="text/javascript"
      src="https://cindyjs.org/dist/latest/Cindy.js"></script>
    <title>タイトル</title>
    <script id="csinit" type="text/x-cindyscript">
      初期設定
    </script>
    <script id="csdraw" type="text/x-cindyscript">
      メインコード
    </script>
    <script type="text/javascript">
      CindyJS({
        scripts: "cs*",
        autoplay: true,
        geometry: [
          幾何要素
        ],
        ports: [
          {
            id: "CSCanvas",
            width: 500,
            height: 500,
            transform: [
              visibleRect: [0, 1, 1, 0]
            ]
          }
        ]
      })
    </script>
  </head>
  <body>
    <div id="cindyjs"></div>
  </body>
</html>
```

```

        }]
    });
</script>
</head>
<body>
<div id="CSCanvas"></div>
</body>
</html>

```

以下に、この内容について解説する。

24.1.1 CindyJS ランタイムのロード

`<script type="text/javascript" src="https://cindyjs.org/dist/v0.8/CindyJS.js"></script>`で CindyJS のサイトから最新のランタイムをロードする。Cinderella から書き出した場合は、CSS をロードする次の行が入る。

```
<link rel="stylesheet" href="https://cindyjs.org/dist/v0.8/CindyJS.css">
```

CindyJS のサイトから `Cindy.js` と `CindyJS.css` をダウンロードして置いた場合は、同じディレクトリであれば

```
<link rel="stylesheet" href="CindyJS.css">
<script type="text/javascript" src="Cindy.js"></script>
```

と書けばよい。

ヘッダには、この他、スタイルシートの内容などの事項を書くことができる。

また、CindyGL などを使う場合も同様に記述する。

24.1.2 スクリプトの記述

```
<script id="csinit" type="text/x-cindyscript">
```

には、CindyScript の初期設定を書く。Cinderella で Initialization スロットに書く内容だ。内部時計を取得する `seconds()` を使う場合は、ここに `resetclock();` を書く。

```
<script id="csdraw" type="text/x-cindyscript">
```

には、draw スロットに書く内容を記述する。

この他次の id が使える。CindyScript の各スロットに応じている。

<code>csmove</code>	いくつかの要素が移動したとき呼び出される
<code>csmousedown</code>	マウスボタンが押された後に呼び出される
<code>csmousemove</code>	マウスが動かされたときに呼び出される
<code>csmousedrag</code>	マウスがドラッグされたときに呼び出される
<code>csmouseup</code>	マウスボタンが離されたら呼び出される
<code>csmouseclick</code>	マウスボタンをクリックすると呼び出されます
<code>cskeydown</code>	キーが押されたときに呼び出される
<code>cstick</code>	時間をとったアニメーションを実行する
<code>csmultidown</code>	マルチタッチで指が下がった後に呼び出される
<code>csmultidrag</code>	マルチタッチでドラッグしたときに呼び出される
<code>csmultiup</code>	マルチタッチで指が離された場合に呼び出される

24.1.3 CindyJS の初期化

`CindyJS()` は CindyJS の初期化関数。以下のパラメータを持つ。

scripts

`scripts: "cs*"` は、前述の `cs` で始まるすべてのスクリプトがアプレットに使用されることをいう。`"csinit"` などだ。

アニメーション `autoplay:true` は、新しいフレームがレンダリングされたときに常に Draw スクリプトを実行する必要があると述べている。このオプションがなければ、構造に何かが変わった場合にのみ、画像は再描画される。

この他、次のオプションがある。

- `controls` は、アニメーション制御ボタン (再生、一時停止、停止) を表示するかどうかを制御するブール値。
- `speed` はアニメーション速度。デフォルトは 1。

幾何要素

`geometry` ブロックは幾何要素についての記述で、作図ツールでとる点や線分などの情報が入る。たとえば、

```
{name:"A", kind:"P", type:"Free", pos:[-4,4]}
```

は、自由点 A を座標 [-4,4] に取ることを表す。

ports

`ports` ブロックは出力ポートを指定する。次の項目からなる。

<code>id</code>	描画キャンバスの要素 id
<code>element</code>	描画キャンバスの DOM 要素
<code>width, height</code>	キャンバスの寸法
<code>background</code>	キャンバスに使用する CSS の背景色。初期値は透明。
<code>transform</code>	座標系変換を指定する。オプションについては後述。省略可。
<code>fill</code>	”window” ウィンドウの innerWidth と innerHeight と一致するようにキャンバスのサイズを調整する。
<code>grid</code>	ユーザー単位でグリッドサイズを指定する。 欠落またはゼロの値は、グリッドが描画されない。
<code>snap</code>	ブール値でスナップモードの指定。デフォルトは false。
<code>axes</code>	座標軸を描画するかどうかを示すブール値。デフォルトは false。

ports の transform のオプション

- `{scale :<number>}`
スケール量
- `{translate:[<number>,<number>]}`
平行移動量
- `{scaleAndOrigin:[<number>,<number>,<number>]}`
事前状態を参照せずに変換を指定する。最初の数字はスケーリング係数で、他の 2 つは原点の位置
- `{visibleRect:[<number>,<number>,<number>,<number>] }`
ウィジェットのサイズを変更するうえで相互作用する方法で、以前の状態を参照せずに変換を指定する。指定された座標は、ユーザー座標で指定された目に見える長方形の左、上、右、下の座標。座標系は、この長方形が完全に見え、ウィジェット内の中間に配置されるよう選択される。

この他に、`defaultAppearance` などのパラメータがある。Cinderella で簡単なスクリプトを書いて、HTML 文書に書き出してみるとよい。

ここまでが HTML のヘッダで、その後に `<body>` を書く。

`<body>` には、`<div id="CSCanvas"></div>` の他に、説明文など通常の HTML の文書を書くことができる。

24.2 CindyJS の HTML 文書の実例

次の例は、マンデルブロ集合を描くための、CindyJS の HTML 文書である。

```

<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>mandelbloat set</title>
    <script type="text/javascript"
        src="https://cindyjs.org/dist/latest/Cindy.js"></script>

<script id="csdraw" type="text/x-cindyscript">
N = 30;
f(x):=(
    c = complex(x); // -0.75; //center -0.75+0*i
    z = 0;
    n = 0;
    repeat(N, k,
        if(|z| <= 2,
            z=z*z+c;
            n = k;
        );
    );
    hue(n/N)
);
colorplot(f(#));
</script>

<script type="text/javascript">
CindyJS({
    scripts: "cs*",
    autoplay: true,
    ports: [{
        id: "CSCanvas",
        width: 500,
        height: 500,
        transform: [(
            visibleRect: [-2, 2, 2, -2]
        )]
    }]
})

```

```

    });
</script>
</head>

<body>
<div id="CSCanvas"></div>
</body>
</html>

```

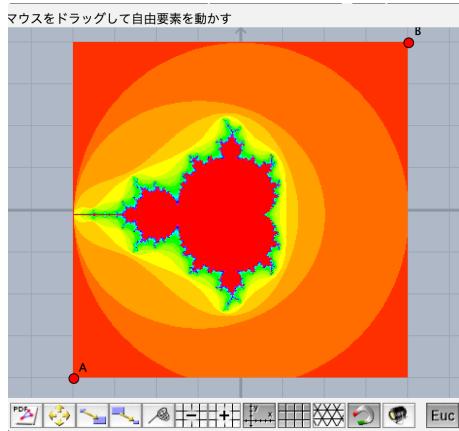
Cinderella で、2点 A(-2,-2) , B(2,2) を作図し、CIndyScript の Draw スロットに次のコードを書いて実行し、HTML に書き出したものと比較すると、Cinderella からの書き出しでどのような情報が追加されているかがわかるだろう。

```

N = 30;
f(x):=(

c = complex(x);
z = 0;
n = 0;
repeat(N, k,
  if(|z| <= 2,
    z=z*z+c;
    n = k;
  );
);
hue(n/N);
colorplot(f(#),A,B,pxlres->1);

```



25 索引

制御

if	もし～ならば～
trigger	条件を満たせば実行
while	While 繰り返し
repeat	繰り返し
forall	すべての要素に対して繰り返し
eval	式の強制的な評価

変数の管理

clear	変数をクリア
keys	オブジェクトまたは変数に関連したローカルキーのリストを返す
createvar	局所変数の作成
removevar	局所変数の削除
regional	1 局所変数を作る。引数の数は任意

出力

print	値を印字する
println	印字して改行する
clearconsole	コンソールをクリア
err	デバッグ用の印字
assert	エラーチェックの印字
format	数を整形する
message	ステータス行にメッセージを表示する

算術関数

sqrt	平方根
exp	指数関数
log	自然対数
sin	正弦
cos	余弦
tan	正接
arcsin	正弦の逆関数
arccos	余弦の逆関数
arctan	正接の逆関数
arctan2	ベクトル (x,y) のなす角
add	和
sub	差

mult	積
div	商
mod	剰余
pow	べき乗
abs	絶対値
round	四捨五入
floor	その数以下の最大の整数
ceil	その数以上の最小の整数
re	複素数の実部
im	複素数の虚部
conjugate	共役複素数
random	一様乱数
randominteger	整数の一様乱数
randombool	ブール値の乱数 true または false
randomnormal	正規乱数
seedrandom	乱数の種 (seed) の設定
ブール関数	
not	論理否定
and	論理積
or	論理和
xor	排他的論理和
型の判定	
isinteger	整数か
isreal	実数か
iscomplex	複素数か
iseven	偶数か
isodd	奇数か
islist	リストか
ismatrix	行列か
isnumbervector	数ベクトルか
isnumbermatrix	数行列か
isstring	文字列か
isgeometric	幾何学要素か
isselected	選択されているか
ispoint	幾何の点か
isline	直線か
iscircle	円か

isconic	円錐曲線か
ismass	質点か
issun	恒星か
isspring	バネか
isundefined	未定義要素か
文字関数	
text	文字列への変換
parse	CindyScript の式として文字列を解析
substring	文字列の抜きだし
replace	文字列の置換
tokenize	文字列の分解
indexof	文字列の検索
length	文字列の長さ
format	数を整形する
guess	文字列の解析
sort	リストの並べ替え
fontfamilies	使用可能なフォントのリストを得る
unicode	ユニコードを文字列に変換する
candisplay	文字列が現在のシステムフォントで表示できるかどうかテストする
微積分	
d	関数の微分
tangent	接線を計算する
guess	浮動小数点数から記号的意味を推測する
roots	n次方程式の解を与える
基本的なリスト処理	
take	リストの要素へのアクセス
length	リストの長さ
contains	内容のテスト
append	要素の後方追加
prepend	要素の前方追加
concat	リストの連結
common	リストの共通部分
remove	リストからの要素の削除
リストの要素の走査	
forall	全要素を走査する
apply	全要素に式を適用する
select	条件を満たす要素を選び出す

高度なリスト処理

pairs	リストの要素のすべてのペアを作る
triples	リストの要素のすべての 3 つの組を作る
directproduct	2 つのリストの直積
consecutive	リストの連続する要素のペアからなるチェーンを作る
cycle	連続する 2 つずつの要素のペアで輪を作る
reverse	リストの要素を逆順にする
set	リストの要素を单一化した集合を作る
sort	リストの要素を並べ替える
flatten	ネストされたリストの平坦化

幾何学要素のリスト

allelements	すべての要素のリストを作る
allpoints	すべての点のリストを作る
alllines	すべての直線のリストを作る
allsegments	すべての線分のリストを作る
allcircles	すべての円のリストを作る
allconics	すべての円錐曲線のリストを作る
allmasses	すべての質点のリストを作る
allsprings	すべてのバネのリストを作る

リストの算術演算

sum	要素の総和を求める
product	要素をすべて掛け合わせる
min	要素の最小値を求める
max	要素の最大値を求める

ベクトルと行列の操作

zerovector	零ベクトルを生成する
zeromatrix	零行列を作る
rowmatrix	行ベクトルから行列への変換
columnmatrix	列ベクトルから行列への変換
matrixrowcolumn	行と列の数を 2 つの数の要素からなるリストとして返す
row	行をベクトルとして返す
column	列をベクトルとして返す
submatrix	小行列を作る
transpose	転置行列を返す
dist	ベクトルの距離を返す
det	正方行列の行列式を返す
hermiteanproduct	2 つのベクトルのエルミート内積を返す

<code>inverse</code>	正方行列の逆行列を返す
<code>adj</code>	正方行列の余因子行列を返す
<code>eigenvalues</code>	正方行列の固有値を返す
<code>eigenvectors</code>	正方行列の固有ベクトルを返す
<code>linearsolve</code>	一次方程式の解を返す
<code>convexhull3d</code>	3次元の凸多面体を作る。

描画関数

<code>draw</code>	点, 線分を描く
<code>drawpoly</code>	多角形を描く
<code>fillpoly</code>	中を塗った多角形を描く
<code>drawcircle</code>	中心と半径を与えて円を描く
<code>fillcircle</code>	中を塗った円を描く
<code>drawall</code>	リストのすべての要素を描画する
<code>connect</code>	リストの点をつなぐ
<code>drawtext</code>	文字列を表示する
<code>drawtable</code>	表を描く
<code>repaint</code>	画面の再描画

描画の外観を設定する

<code>pointsize</code>	点の大きさ
<code>linesize</code>	線の太さ
<code>textsize</code>	文字の大きさ
<code>pointcolor</code>	点の色
<code>linecolor</code>	線の色
<code>textcolor</code>	文字の色
<code>color</code>	すべての色
<code>alpha</code>	すべての不透明度のアルファ値
<code>gsave</code>	色, アルファ値, 幅, 変換をスタックに入れる
<code>grestore</code>	色, アルファ値, 幅, 変換をスタックから出す
<code>greset</code>	色, アルファ値, 幅, 変換のスタックを空にする

色の関数

<code>red</code>	赤の RGB ベクトルを返す
<code>green</code>	緑の RGB ベクトルを返す
<code>blue</code>	青の RGB ベクトルを返す
<code>grey</code>	灰色の RGB ベクトルを返す
<code>hue</code>	色相の RGB ベクトルを返す

関数プロット

<code>plot</code>	関数をプロットする
-------------------	-----------

<code>fillplot</code>	積分のようにハイライトさせてプロットする
<code>colorplot</code>	関数の値によって色をプロットする
<code>drawfield</code>	ベクトル場を描画する
<code>drawfieldcomplex</code>	複素ベクトル場を描画する
<code>drawforces</code>	力の場を表示する
<code>drawcurves</code>	物理のオシログラフを描く
画像の操作	
<code>drawimage</code>	画像を表示する
<code>mapimage</code>	画像を変形する
<code>imagesize</code>	画像の大きさを取得する
<code>imagergb</code>	画素の色情報を取得する
<code>createimage</code>	カスタム画像を作る
<code>clearimage</code>	画像の内容を消去する
<code>removeimage</code>	画像を削除する
<code>canvas</code>	キャンバスに描く
シェイプ	
<code>circle</code>	円形のシェイプを作る
<code>polygon</code>	多角形シェイプを作る
<code>halfplane</code>	半平面シェイプを作る
<code>screen</code>	スクリーンのシェイプを作る
<code>fill</code>	シェイプを塗りつぶす
<code>draw</code>	シェイプの輪郭線を描く
<code>clip</code>	シェイプのクリップパスの設定
幾何変換	
<code>translate</code>	座標系全体を平行移動
<code>rotate</code>	座標系全体を回転
<code>scale</code>	座標系全体を拡大・縮小
<code>setbasis</code>	座標系全体の基底を変換
幾何関数	
<code>moveto</code>	点の位置を変える
<code>meet</code>	直線の交点を求める
<code>join</code>	2点を結ぶ
<code>perp</code>	垂直な2次元ベクトルを返す
<code>perp</code>	点を通り直線に垂直な直線を求める
<code>perpendicular</code>	点を通り直線に垂直な直線を求める
<code>para</code>	点を通り直線に平行な直線を求める
<code>parallel</code>	点を通り直線に平行な直線を求める

<code>cross</code>	2つの3次元ベクトルの外積を返す
<code>dist</code>	2点間の距離を返す
<code>area</code>	3点で与えられた三角形の面積を返す
<code>det</code>	3×3 行列の行列式を計算する
<code>crossratio</code>	4点の複比を計算する
<code>complex</code>	xy座標を複素数に変換する
<code>gauss</code>	複素数を点の座標に変換する
<code>point</code>	数ベクトルを点として位置づける
<code>line</code>	数ベクトルを直線として位置づける
<code>geotype</code>	ベクトルの幾何学的意味を返す
<code>map</code>	幾何変換を表す行列を返す
<code>pointreflect</code>	点に関する対称点を求める行列を返す
<code>linereflect</code>	直線に関して対称な点を求める行列を返す
<code>incidences</code>	幾何要素にインシデントである要素のリストを返す
<code>locusdata</code>	軌跡上の点の座標のリストを返す
インスペクタ	
<code>inspect</code>	利用できる属性のリストを作る
<code>inspect</code>	属性を読み出す・設定する
要素の作成と消去	
<code>createpoint</code>	位置とラベルを指定して点を加える
<code>create</code>	幾何要素を作成する
<code>removeelement</code>	幾何の要素をそれに従属する点とともに消去する
<code>algorithm</code>	幾何要素の作図手順を得る
<code>inputs</code>	幾何要素を定義するために必要な要素のリストを得る
<code>element</code>	引数の名前の幾何要素へのハンドルを得る
MIDI	
<code>playtone</code>	MIDIの単音を鳴らす
<code>stop tone</code>	音を止める
<code>playfrequency</code>	特定の周波数で鳴らす
<code>playmelody</code>	メロディを鳴らす
<code>midiaddtrack</code>	シーケンサにトラックを追加する
<code>midistart</code>	シーケンサの開始
<code>midistop</code>	シーケンサの停止
<code>midispeed</code>	シーケンサの速度設定
<code>midispeed</code>	シーケンサの速度を問い合わせる
<code>midiposition</code>	シーケンサの位置を設定する
<code>midiposition</code>	シーケンサの開始位置を得る

<code>instrument</code>	楽器を選ぶ
<code>instrumentnames</code>	利用できる楽器のリストを得る
<code>midichannel</code>	チャンネルを設定する
<code>midivolume</code>	チャンネルの音量を設定する
<code>midicontrol</code>	チャンネルのコントローラを設定する
サンプルーオーディオ	
<code>playsin</code>	特定の周波数の音を鳴らす
<code>playfunction</code>	数式で定義した音を鳴らす
<code>playwave</code>	オーディオデータのリストを再生する
<code>stopsound</code>	すべての音の再生を停止する
マウスとキーボードからの入力	
<code>mover</code>	最後に動かされた点を返す
<code>mouse</code>	マウスボタンが押されたときのマウスの位置を返す
<code>elementsatmouse</code>	マウスカーソルの近くにある要素のリストを返す
<code>key</code>	キーボードで打たれた文字列を返す
<code>iskeydown</code>	あるキーが押されたかどうか問い合わせる
<code>keydownlist</code>	押されたキーのリストを返す
<code>amsdata</code>	重力センサからデータを得る
<code>calibratedamsdata</code>	調整された AMS データを取得する
時間	
<code>resetclock</code>	内部時計のリセット
<code>seconds</code>	リセット後の経過時間
<code>time</code>	現在の時刻
<code>date</code>	現在の日付
<code>wait</code>	ミリ秒単位で指定された時間、実行を止めて待つ
ファイル入出力	
<code>setdirectory</code>	ファイルのディレクトリを設定する
<code>load</code>	データの読み込み
<code>import</code>	プログラムコードの読み込み
<code>openfile</code>	ファイルを開いてハンドルを返す
<code>closefile</code>	ファイルを閉じる
<code>print</code>	ファイルに書く
<code>println</code>	ファイルに書いて改行する
ネットワーク	
<code>openurl</code>	Web ページをブラウザで開く
物理シミュレーション	
<code>simulation</code>	シミュレーションへのハンドル提供

force	力の探索粒子を設定する
addforce	力を存在する質点に適用する
setforce	力を存在する質点に設定する
アニメーション	
playanimation	アニメーションを開始する
pauseanimation	アニメーションをいったん停止する
stopanimation	アニメーションを終わる
Cindy3D	
gsave3d	現在の描画設定の保存
grestore3d	描画設定の復帰
draw3d	点, 線を描く
connect3d	点を結ぶ
drawpoly3d	多角形を描く
fillpoly3d	多角形の面を描く
fillcircle3d	円盤を描く
drawsphere3d	球面を描く
mesh3d	網目状の曲面を描く
background3d	背景の色を設定する
lookat3d	カメラの位置
fieldofview3d	画角の設定
depthrange3d	カメラ深度の設定
renderhints3d	レンダリングのヒントを設定
pointlight3d	点光源の設定
directionallight3d	方向光源の設定
spotlight3d	スポットライトの設定
disablelight3d	光源を無効にする