Monaca

「情報Ⅰ」に向けたプログラミング研修会

～モデル化とシミュレーション演習（JavaScript版）～

アシアル株式会社
アシアル情報教育研究所
岡本 雄樹

はじめに

自己紹介

- 名前
  - 岡本雄樹(アシアル情報教育研究所 所長)
- 著書
  - イラストでよくわかるPHP
  - WordPressプロフェッショナル養成読本
  - Monacaで学ぶはじめてのプログラミング
- メッセージ
  - 「コンピューター」「インターネット」「プログラミング」私は高校生の時にそれらと出会うことで人生が拓けました。先生方とMonacaによるアプリ開発を通じて、情報技術の活用方法や作品作りの楽しさを広めてまいります。

はじめに

■ 教材サイトのご案内

- あんこエデュケーション
  - https://anko.education/
- 教科情報研修資料
  - https://anko.education/joho

はじめに

■ プロジェクトのインポート

- 教材サイトに各種モデルのサンプルプロジェクトがあります
- Monacaでインポートして進めます
- 予めMonacaのログインと空きプロジェクト数の確保をお願いします

■ 学習16 確定モデルと確率モデル

- 複利法による預金の複利計算(確定モデルのシミュレーション)
- サイコロによる確率モデルのシミュレーション
- モンテカルロ法による円周率の計算

■ 学習17 自然現象のモデル化とシミュレーション

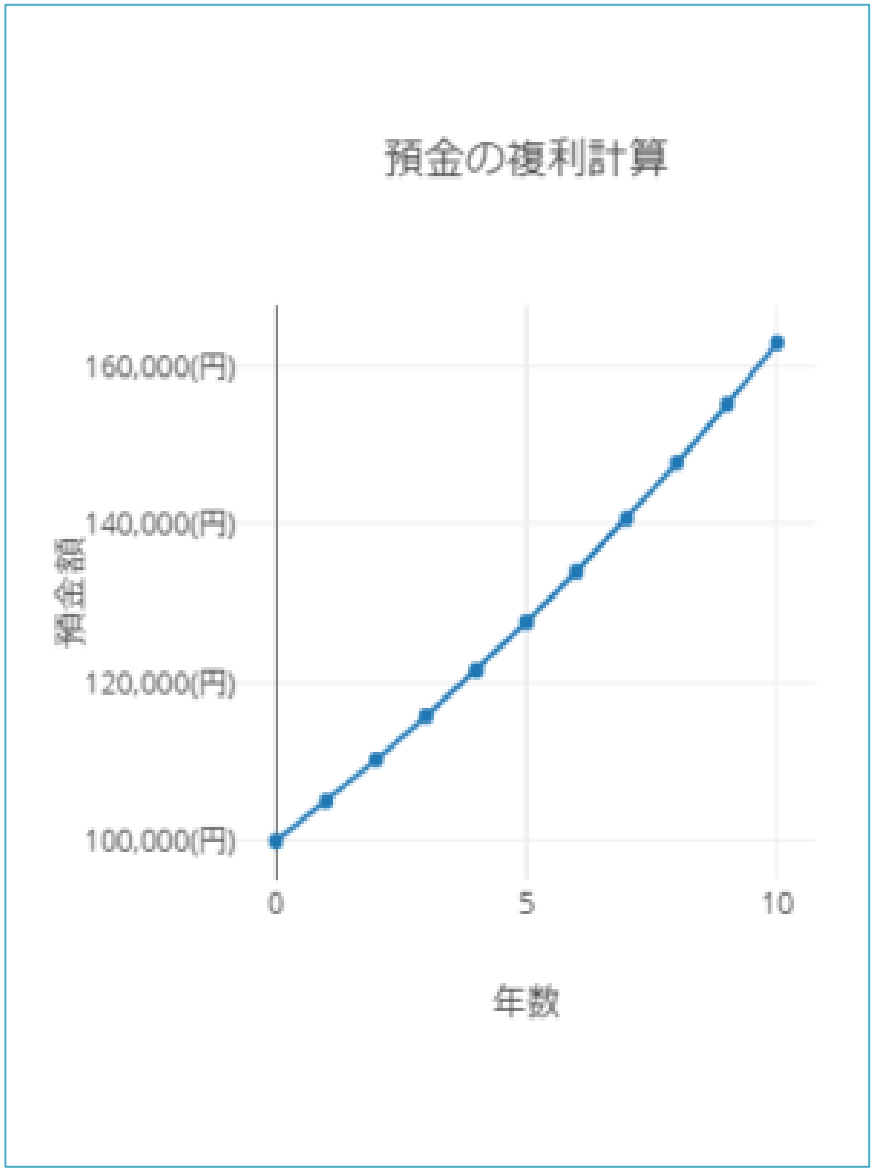
- 物体の放物運動のモデル化(斜方投射)
- 生命体の増加シミュレーション(ロジスティクス曲線)
- ランダムウォークのシミュレーション

確定モデルと確率モデル

確定モデル

複利法による預金の複利計算

- モデルを使ってN年後の預金額を予測（シミュレーション）してみましょう。
- 金利が一定なら確定モデルでシミュレーション可能です



実習：<script>の中にプログラムを記述

```
let yokin = 100000;
let riritsu = 0.05;
let risoku;
let term = 10;

for (let i = 0; i < term; i++) {
    risoku = yokin * riritsu;
    yokin = yokin + risoku;
    document.write(i + 1, "年目:", yokin, "<br>");
}
```

実行例：プレビュー表示

```
1年目:105000
2年目:110250
3年目:115762.5
4年目:121550.625
5年目:127628.15625
6年目:134009.5640625
7年目:140710.042265625
8年目:147745.54437890626
9年目:155132.82159785158
10年目:162889.46267774416
```

終価係数で10年後の預金を計算すると以下のような式になります

終価係数 = $(1 + 0.05)^{10}$

10年後の預金額 = 10万 * 上記終価係数

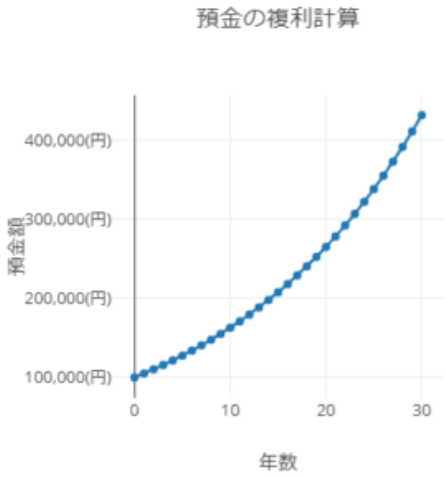
グラフ化に挑戦

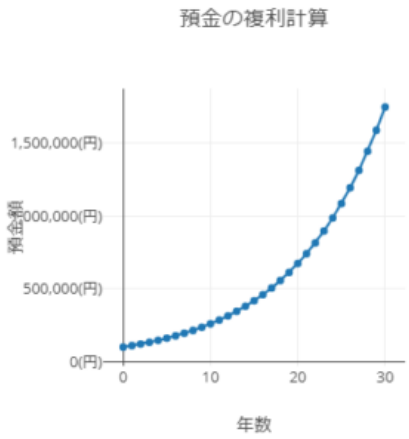
- グラフライブラリの活用
  - グラフの種類やタイトルなどを設定する
  - Y軸やX軸に目盛りやタイトルを設定する
- 配列技術の習得
  - 複数の値を一つの配列に格納する
  - 連想配列などの応用

key	value
0	100000
1	105000
2	110250
3	115762.5
4	121550.6
5	127628.2
6	134009.6
7	140710
8	147745.5
9	155132.8
10	162889.5

■ 預金の複利計算をグラフで表示する

- サンプルプロジェクトのインポート
- 演習
  - パラメーターを変更しながら利息計算の特徴について考える





変数定義

- 預金の初期値を配列で持つ

```
let yokin = [100000];
let riritsu = 0.05;
let risoku;
let term = 10;
```

ループ

- 配列.push()命令で要素を追加
- 現在の値を元に次の要素の値を計算

```
for (let i = 0; i < term; i++) {
    risoku = yokin[i] * riritsu;
    yokin.push(yokin[i] + risoku);
}
```

グラフ設定

- 横軸(x)のタイトルを年数
- 縦軸(y)のタイトルを預金額
- tickとは目盛りのこと
- suffixは接尾辞
- preffixは接頭辞(今回はない)
- exponentは指数のこと

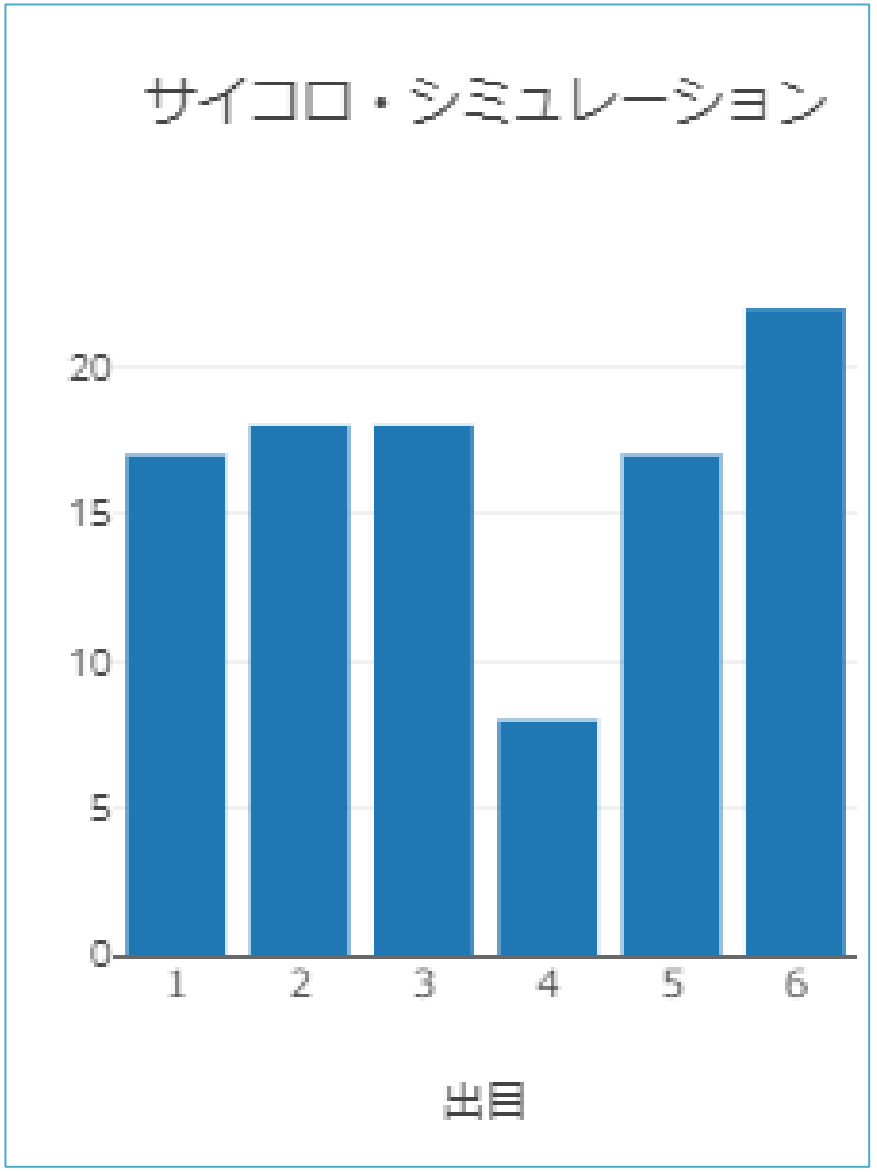
```
xaxis: {
    title: "年数"
},
yaxis: {
    title: "預金額",
    ticksuffix:"(円)",
    exponentformat:"none",
},
```

確率モデル

サイコロによる確率モデルのシミュレーション

プログラミング言語の機能で乱数を発生させサイコロをシミュレーションします

- サイコロのモデルをプログラミング
- 出目をカウント
- グラフ化して傾向を確認



■ JavaScriptで乱数を発生させる方法

JavaScriptではMath.random()関数で乱数を発生させることができます。Math.random()は0～0.999…の範囲でランダムな値を返します。なお、乱数にも幾つか種類があるのですが、このような乱数を一様乱数と呼びます。

◆命令：ランダム関数

```
Math.random()
```

◆例：ランダム関数でサイコロをモデル化

```
let min = 1;
let max = 6;
let length = max - min + 1;
let value = Math.floor(Math.random() * length) + min;

console.log(value);
```

■ サイコロを再現する関数の作成

サイコロを関数化します。

◆例：サイコロを関数化

```
function random(min = 1, max = 6) {
    let length = max - min + 1;
    let value = Math.floor(Math.random() * length) + min;
    return value;
}
```

■ 複数のサイコロを一斉にふる関数の作成

グラフ化に備えて、複数のサイコロを一斉に振る関数を作成します。

◆例：複数のサイコロを一斉に振るための関数

```
function random_values(min = 1, max = 6, number = 1) {
    let result = [];
    for (let i = 0; i < number; i++) {
        result.push(random(min, max));
    }
    return result;
}
```

◆例：実行例

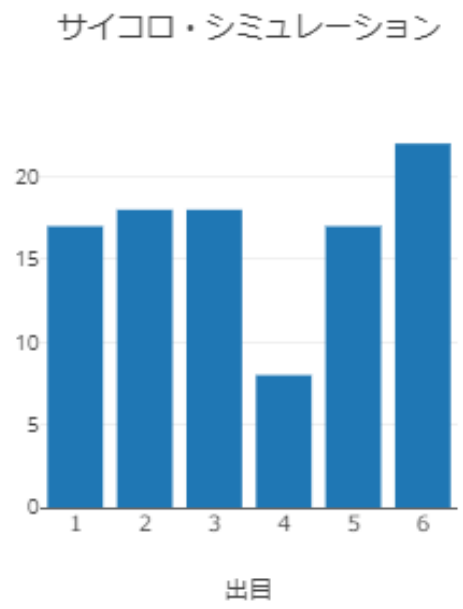
```
console.log(random_values(1, 6, 100));
```

◆例：実行結果

```
(100) [5, 5, 2, 5, 6, 4, 5, 1, 6, 4, 2, 6, 4, 2, 6, 4, 2, 1, 4, 2, 3, 3, 1, 1, 5, 3,
6, 3, 3, 6, 3, 3, 6, 1, 1, 5, 5, 6, 6, 2, 5, 3, 1, 6, 1, 5, 2, 3, 1, 6, 2, 5, 4, 5,
4, 4, 5, 2, 1, 1, 4, 3, 1, 3, 5, 3, 6, 2, 1, 6, 6, 5, 3, 3, 4, 5, 4, 1, 5, 3, 3, 4,
2, 4, 6, 2, 4, 6, 1, 1, 4, 4, 1, 5, 2, 6, 4, 5, 2, 3]
```

■ 【実習】サイコロの出目をグラフで表示する

- サンプルプロジェクトのインポート
- 演習
  - サイコロを振る回数を1000や10000などに増やして実験



変数定義

- サイコロを100回振った結果をdeme配列に格納
- 1～6の値が何回ずつ出たのかを集計するための連想配列としてresultを用意
    - 全部0回として初期化

```
let deme = random_values(1, 6, 100);
let result = {
    1:0,
    2:0,
    3:0,
    4:0,
    5:0,
    6:0
}
```

ループ

- 1～6の値が何回出たのかをresult連想配列に集計

```
for (let i = 0; i < deme.length; i++ )
{
    result[deme[i]]++;
}
```

グラフ描画

- 横軸(x)に全角で１～６を指定
- 縦軸(y)に出目の回数を指定
- Object.values()は連想配列の値を配列で返す命令

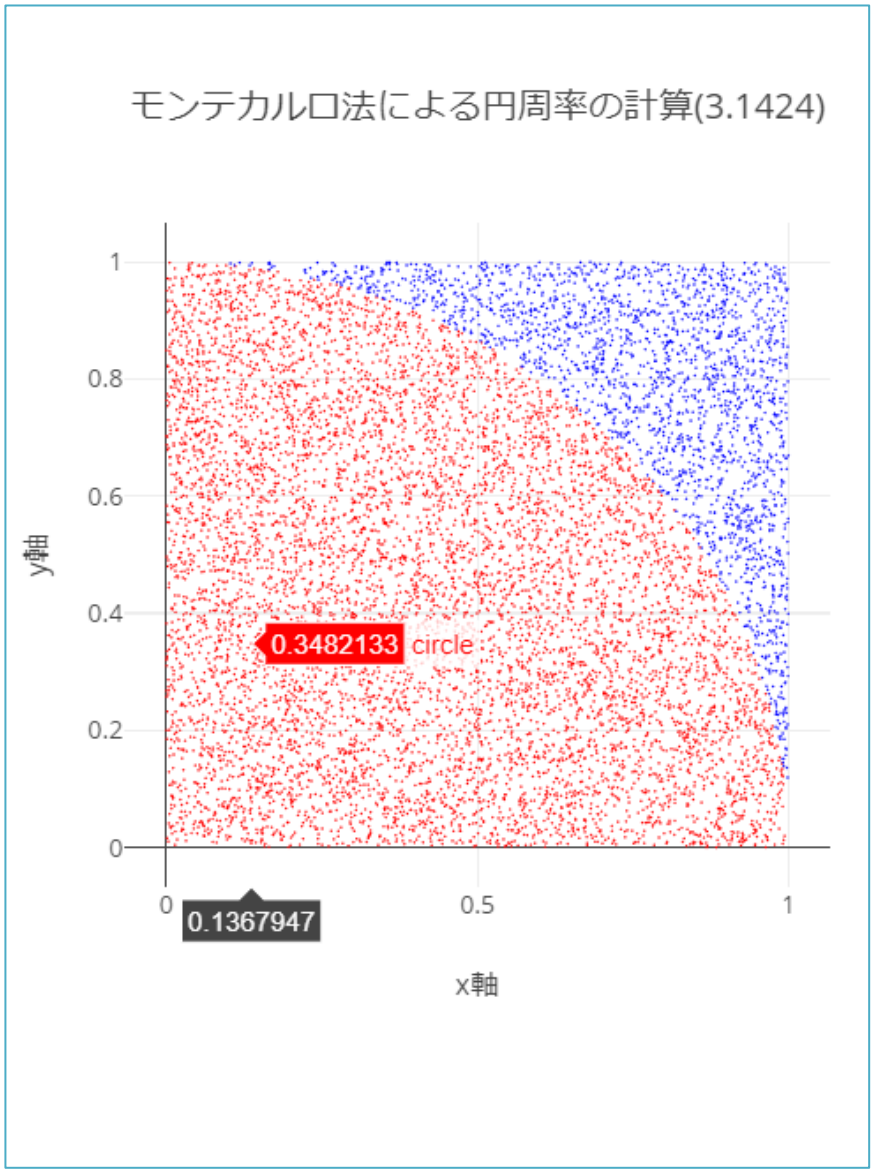
```
let x_memori = ["１", "２", "３", "４", "５", "６"];
let trace = {
    x: x_memori,
    y: Object.values(result),
    type: "bar"
};
let data = [trace];
```

モンテカルロ法による円周率の計算

モンテカルロ法による円周率の計算

乱数を使って円周率(π)の近似値を求めます

- 円周率について学習
- ランダムにプロットしつつ円の中に入った点の数を記録
- 割合から円周率を求める



■ モンテカルロ法による円周率の計算

- サンプルプロジェクトのインポート
  - 「グラフなし版」と「グラフあり版」をご用意
- 演習
  - プロット回数を増やすと精度が上がるかを確認する

JavaScriptでモンテカルロ法による円周率計算を行う

下記のようなプログラムを実行すれば円周率を求めることができます。

```
// 計算に使う変数の定義
let totalcount = 10000;
let incount = 0;
let x,y,distance,pi;

// ランダムにプロットしつつ円の中に入った数を記録
for (let i = 0; i < totalcount; i++) {
    x = Math.random();
    y = Math.random();
    distance = x ** 2 + y ** 2;

    if (distance < 1.0){
        incount++;
    }
    console.log("x:" + x + " y:" + y + " D:" + distance);
}

// 円の中に入った点の割合を求めて4倍する
pi = (incount / totalcount) * 4;
document.write("円周率は" + pi);
```

円周率とは

円の面積や円の円周の長さを求めるときに使う、3.14…の数字です、π(パイ)のことです。

πは数学定数の一つです。

JavaScriptではMathオブジェクトのPIプロパティで円周率を取ることができます。

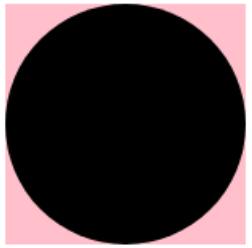
```
alert(Math.PI)
```

正方形の四角形の面積と円の面積

- 正方形の四角形の面積は縦と横の長さが分かれば求めらる。
- 円の面積も直径が分かれば求められる。
- 仮に長さ100の正方形と直径100の円を比較した場合、正方形の法が面積は大きくなる。
    - 下図は直径100ですが、プログラムの方では直径2、半径1です

10000 =
(100 x 100)

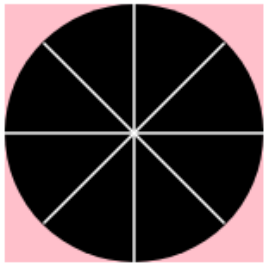
7500 ≒
(50 x 50 x π)

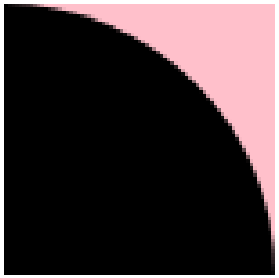


正方形 > 円

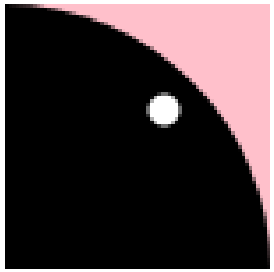
どうやって円周率を求めるか？

- 円の中心から円周に向かって放射線を引く(長さは半径)
- 適当に点をプロットする
- 中心からの距離が一定に収まっている点は円の中と判定
    - 中心からの距離は三平方の定理で求められる
- 点を沢山プロットし、円の中に落ちた割合を円周率とする

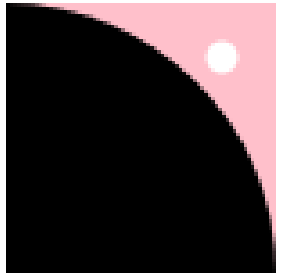




半径^2 = 2500 =
50-2



座標(30,30)
距離^2 = 1800 =
30^2 + 30^2



座標(40,40)
距離^2 = 3200 =
40^2 + 40^2

変数定義

- プロット回数を宣言
- 円の中にプロットした回数をincountに宣言してカウント(初期値は0)
- その他の変数を宣言

```
let totalcount = 10000;
let incount = 0;
let x,y,distance,pi;
```

ループ

- プロット位置(x,y)をランダムに決める
- プロット位置が円の中心からどれくらい離れているのかを計算
    - 半径を1とした場合、1の二乗も1なので1と比較すれば良い
- 位置が半径よりも近ければ変数incountを増やす

```
for (let i = 0; i < totalcount; i++) {
    x = Math.random();
    y = Math.random();
    distance = x ** 2 + y ** 2;

    if (distance < 1.0 ** 2){
        incount++;
    }
    console.log("x:" + x + " y:" + y + " D:" + distance);
}
```

仕上げ

- 円の中に収まった点の割合を求める
    - 仮に2500/10000なら、0.75
- 四半円なので4倍して円にする
    - 仮に0.75なら3
- 求めた円周率を出力

```
pi = (incount / totalcount) * 4;
document.write("円周率は" + pi);
```

変数定義

- 点の座標を配列に記録できるように宣言

```
let circle_x = [];
let circle_y = [];
let outer_x = [];
let outer_y = [];
```

ループ

- 位置が半径よりも近ければ変数incountを増やす
- ついでにプロット用の配列に位置を記録

```
if (distance < 1.0 ** 2){
    incount++;
    circle_x.push(x);
    circle_y.push(y);
} else {
    outer_x.push(x);
    outer_y.push(y);
}
```

グラフのレイアウト

- サイズやタイトルを指定
- showlevend はグラフの凡例を出すかどうかのフラグ
    - 横幅が狭くなるので今回はfalse

```
let layout = {
    height: 500,
    width:  500,
    showlegend:false,
    title:"モンテカルロ法による円周率の計算(" + pi + ")",
    xaxis: {
        title: 'x軸'
    },
    yaxis: {
        title: 'y軸'
    },
};
```

円の内側のトレース

- circle_xとcircle_yの値を元にグラフを用意
- 点の色やサイズを決めたり、名前を定義できる

```
let circle = {
    x: circle_x,
    y: circle_y,
    name:"circle",
    mode: 'markers',
    type: 'scatter',
    marker:{
        color:'red',
        size:1
    }
};
```

※ 円の外側のトレースも同様に用意

プロット

- グラフ用のデータが揃ったらplot命令で図をプロット

```
let data = [circle,outer];
Plotly.newPlot(graph, data, layout);
```

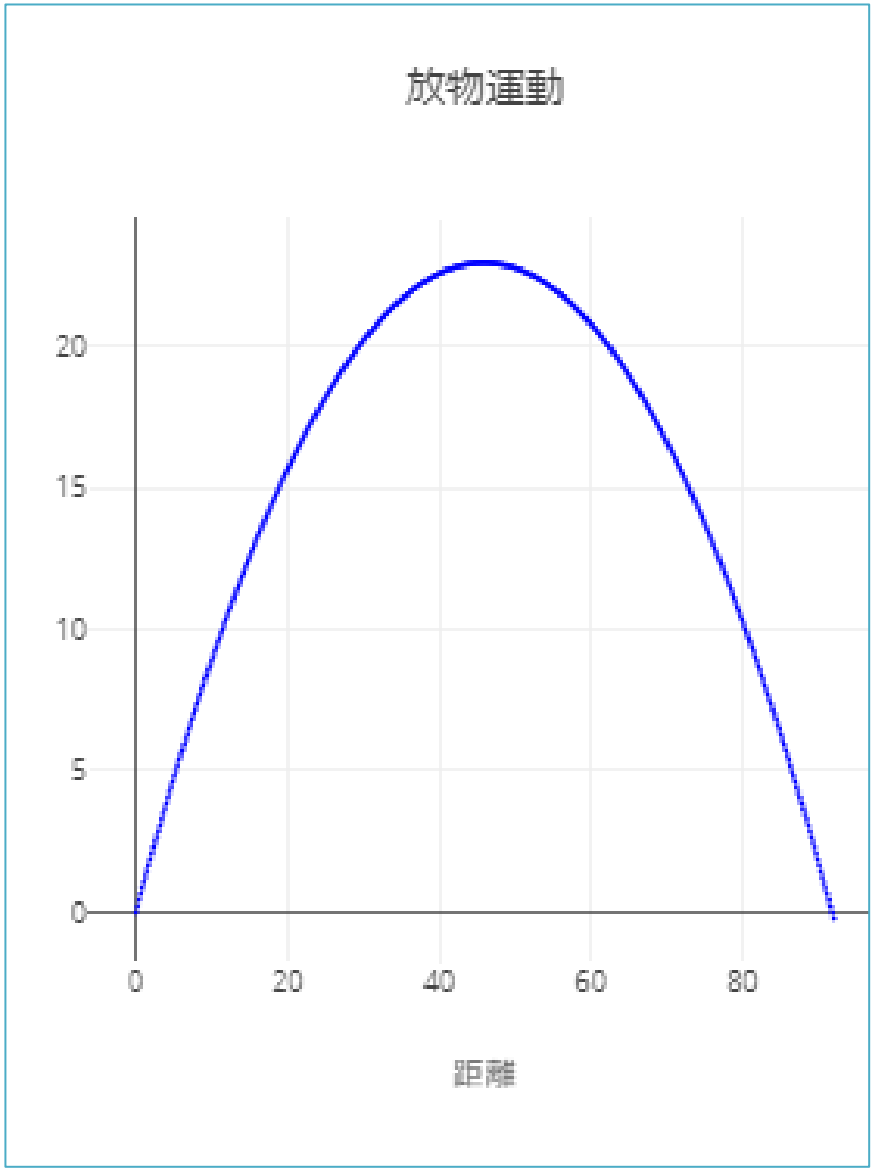
自然現象のモデル化とシミュレーション

物体の放物運動のモデル化（斜方投射）

物体の放物運動のモデル化

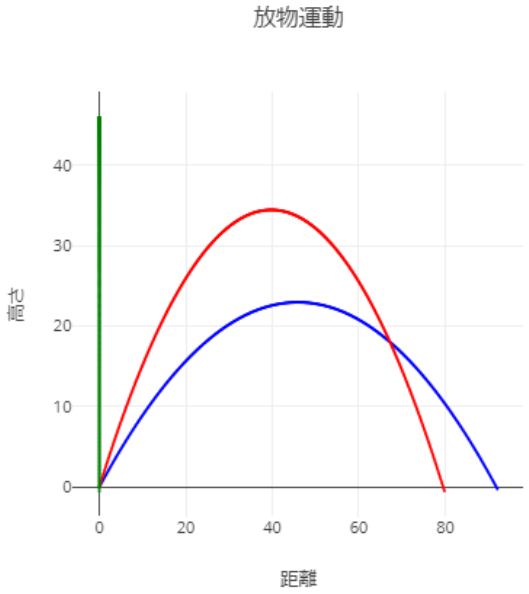
物体を投射したときの高さと距離をシミュレーションします

- 空気抵抗は省きます
- 地面に着地したら終了
- 初速度によって距離は変化
- 角度によって高さや距離は変化



放物運動とは

物体を投げたとき、物体は放物線を描くように一定の高さまで上昇し、途中で重力で落下して着地します。強く投げれば初速度も速くなり、高く遠くへ飛ぶはずです。また、角度によっても高やさ飛距離は変わると考えられます。それと、真上に上げた場合には飛距離は0になるはずです。



水平方向(X)の移動(つまり飛距離)

今回は空気抵抗を省いて考えますので、水平方向への移動を妨げるものはありません。等速運動になります。重力によって地面に着地するまでずっと同じスピードで飛んでいきます。

水平方向の速度は「初速度」と「角度」が分かれば三角関数のコサインを使って求められます。コサインはMathの機能で呼び出せるため以下のような記述で求められます。

```
v0    = 30;                          // 初速度
deg   = 60;                          // 角度(deg)
angle = deg * Math.PI / 180.0;       // 投げ上げ角度(ラジアン)
vx    = v0 * Math.cos(angle)         // 水平「方向」の初速度
```

上記の場合vxは「15」になります。
初速度30の角度60度で投げると水平方向の初速度は半減して15になるようです。

鉛直方向（Y）の移動

水平方向と異なり、鉛直方向への速度は重力があるので徐々に減少していきます。そして重力に負けて鉛直方向への速度はマイナスになり最後は地面に着地します。

鉛直方向の初速度も「初速度」と「角度」が分かれば三角関数のサインを使って求められます。サインもMathの機能で呼び出せるため以下のような記述で求められます。

```
v0    = 30;                          // 初速度
deg   = 60;                          // 角度(deg)
angle = deg * Math.PI / 180.0; // 投げ上げ角度(ラジアン)
vy    = v0 * Math.sin(angle);  // 鉛直「方向」の初速度
```

上記の場合vyは「25.980762113533157」という数字になります。
角度60度で投げた場合には鉛直方向の初速度の方が速いようですね。

重力加速度の影響

鉛直方向には重力があります。地球の重力加速度は9.8m/s²なので1秒ごとに速度が9.8m/s変化します。約25m/sの速度で「鉛直投げ上げ」した物体は約2.5秒（約25m/s）で落下しはじめます。

ちなみに、初速度30m/sというのは時速に直すと108km/hなので、時速108km・角度60度で投げた球が約2.5秒で落下しはじめ、約5秒で着地するという話とも言えます。

変数定義

変数名	意味
max	繰り返し上限数。文科省の教員研修資料ではこの値は変数化されておらずコード中に直接1000と指定されている。
dt	時間間隔。文科省版では0.01が代入されている。
v0	初速度。文科省版では30が指定されている。条件を変えてシミュレーションしたいときにはこの値を変えることになる。
g	重力加速度。地球の重力加速度は9.8m/s²なのでこの変数にも9.8m/s²が入る。月の斜方投射をシミュレーションしたいときにはこの値を減らすことになる。
angle	投げ上げ角度をラジアンで代入する。角度60度や45度はラジアンではなくdegなので 変換してから代入する必要あり。
x	水平位置。真上や後ろにでも投げない限りは時間とともに増加していく。
y	鉛直位置。投げはじめは上昇を続けるが重力によって途中から下がって最後は地面に着地する。
vx	水平速度。今回の設定では空気抵抗がないことになっているため繰り返し中では一切変化しない。
vy	鉛直速度。最初は整数だが重力加速度により徐々に減少し負数になっても加速し続ける。
vyavg	鉛直速度の「平均」。この値を先に求めないと鉛直位置が求められない。

変数定義

- 初速度や角度、時間間隔やループ上限などを宣言
- 時間間隔毎の水平位置や鉛直位置は配列に追加

```
let dt = 0.01; // 微小時間(時間間隔)
let max = 1000; // ループ上限
let v0 = 30; // 初速度
let g = 9.8; // 重力加速度
let angle = 45.0 * Math.PI / 180.0;  // 投げ上げ角度(ラジアン)

let x = [0]; // 水平位置の初期値は0(距離)
let y = [0]; // 鉛直位置の初期値は0(高さ)
let vx = [v0 * Math.cos(angle)]; // 水平「方向」の初速度
let vy = [v0 * Math.sin(angle)]; // 鉛直「方向」の初速度

let vyavg; // 鉛直方向の平均速度

console.log("x:" + x , " y:" + y + " vx:" + vx + " vy:" + vy);
```

※ 確認のためにconsole.log()で各値をログに書き出しています。

ループ

- 10秒経過するか着地するまで処理を実行
- 現在の速度や位置を元に0.01秒後(dt)の結果を計算

```
for (let i = 0; i < max; i++) {
    vx.push(vx[i]);
    vy.push(vy[i] - g * dt);
    x.push(x[i] + vx[i] * dt);
    vyavg = (vy[i] + vy[i + 1]) / 2.0;
    y.push(y[i] + vyavg * dt);

    // 高さが0未満になったら終了
    if (y[i] < 0) {
        console.log(i + "回目のループで着地");
        break;
    }
    console.log("x:" + x[i] , " y:" + y[i] + " vx:" + vx[i] + " vy:" +
vy[i]);
}
```

※ 着地による中断処理を行わないと、斜方投射した物体が高いところから落下し続ける様子をシミュレーションできます。

vt秒後の水平速度

水平速度は一定

```
vx.push(vx[i]);
```

vt秒後の鉛直速度

鉛直速度は重力加速度により減速、0以下になったらマイナス

```
x.push(x[i] + vx[i] * dt);
```

vt秒後の水平位置

水平位置は現在位置にdt秒後の移動距離を加算するだけ

```
x.push(x[i] + vx[i] * dt);
```

vt秒後の鉛直位置

鉛直位置もdt秒後の移動距離を加算する。ただし、鉛直速度はdt秒毎に変化しているため、鉛直速度の平均速度を求めてから移動距離を計算。

```
vyavg = (vy[i] + vy[i + 1]) / 2.0;
y.push(y[i] + vyavg * dt);
```

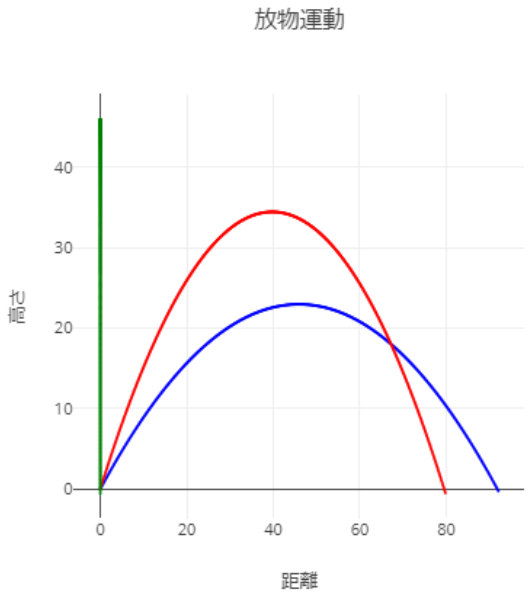
プロット

- グラフ用のデータが揃ったらplot命令で図をプロット

```
let data = [trace];
Plotly.newPlot(graph, data, layout);
```

複数の角度でグラフを描いて比較する

一番遠くに飛ぶ角度と一番高く飛ぶ角度を求めようという演習が教員研修資料では示されています。複数のグラフを描画するためにコピペで対応すると可読性や保守性が悪くなるので、先にプログラムを改造してシミュレーション部分を「関数化」してから描画してみることにします。



関数化

引数　：初速度や角度、トレースの色情報を受け取ります

返り値：トレース情報を返します

```
function simulation (v0 = 30, deg = 45, color = "blue") {
  ...
  return trace;
}
```

実行

トレース情報をdata配列に詰めてプロットするだけです

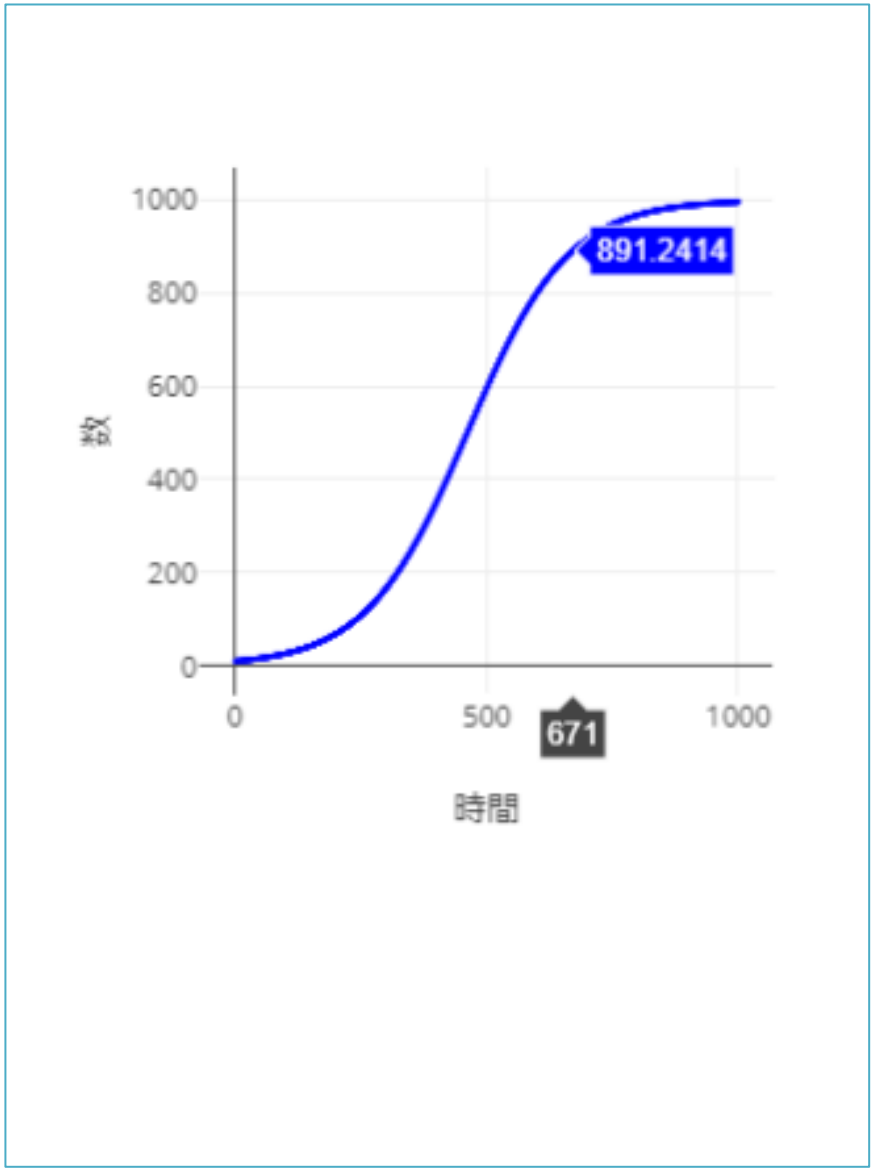
```
function plot() {
    let graph = "myDiv";
    let data = [
        simulation(30, 45, "blue"),
        simulation(30, 60, "red"),
        simulation(30, 90, "green"),
    ];
    let layout = {
        height: 500,
        width:  500,
        showlegend:false,
        title:"放物運動",
        xaxis: {
            title: "距離"
        },
        yaxis: {
            title: "高さ"
        },
    };

    Plotly.newPlot(graph, data, layout);
}
```

生命体の増加シミュレーション(ロジスティクス曲線)

生命体の増加シミュレーション

生命体がどのように増加するかをシミュレーションします

- 環境収容力は固定値
- 繁殖力の高い生命体は増加率が高い
- 個体数が増えると減少率も増えていく
- 個体数が環境収容力に達すると増加率と減少率が釣り合う
- 天敵はいないものとする



変数定義

変数名	意味
max	繰り返し上限数。
number	個体数の配列。時間帯毎にどんどん増える様を追加していきます。
zoukasuu	増加数。現在の個体数と増加率をかけ算して求めることができる。
zouka	増加率。繁殖力の強い生命の場合はこの値を大きくすれば良い。
gensyousuu	減少数。減少率を元に計算できる。
gensyou	減少率。減少率の式は先述の通り。
capacity	環境収容力。

変数定義

- 繰返し回数を宣言
- 生命体の初期値を設定
- 増加率と減少率、増加数と減少数の変数を宣言
    - 増加率は固定値、他は計算で求める
- 環境収容力を宣言

```
let max = 1000; // ループ上限
let number = [10]; // 個体数
let zoukasuu; // 増加数
let zouka = 0.01; // 増加率
let gensyousuu; // 減少数
let gensyou; // 減少率
let capacity = 1000; // 環境収容力
```

ループ

- 個体数と増加率から増加数を計算で求める
- 個体数と環境収容力と増加数から減少率を求める
- 減少率から減少数を求める
- 増加数と減少数を元に個体数の配列を追加する

```
for (let i = 0; i < max; i++) {
    zoukasuu = number[i] * zouka;
    gensyou = (number[i] / capacity) * zouka;
    gensyousuu = number[i] * gensyou;
    number.push(number[i] + (zoukasuu - gensyousuu));
    console.log("zoukasuu:" + zoukasuu + " gensyousuu:" + gensyousuu);
}
```

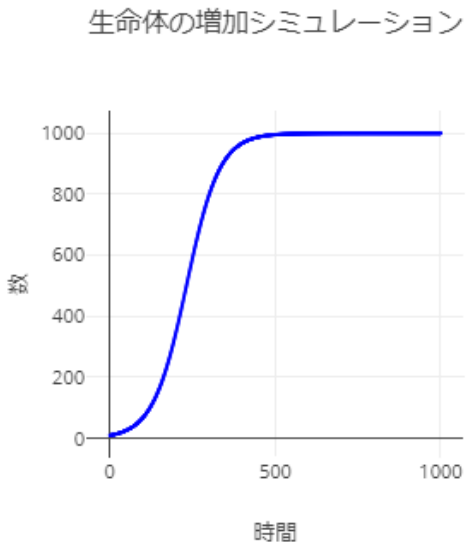
プロット

- グラフ用のデータが揃ったらplot命令で図をプロット

```
let trace = {
    y: number,
    mode: "markers",
    type: "scatter",
    marker:{
        color:"blue",
        size:2
    }
};
let data = [trace];

Plotly.newPlot(graph, data, layout);
```

増加率の変更

- -0.01ではなく別の数字に変更してみて下さい。



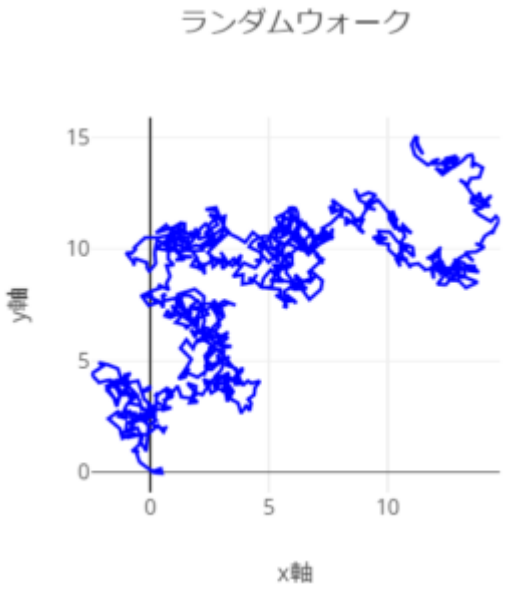

```
0.02
```

ランダムウォークのシミュレーション

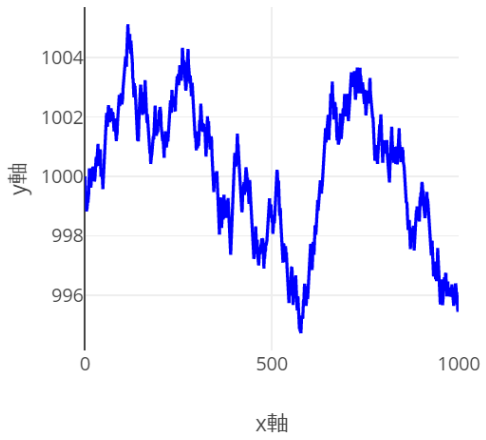
ランダムウォーク

不確実な現象（株価の変動など）をシミュレーションする際に用いられるランダムウォークのモデルを学習します

- 二次元のランダムウォークでは中心座標から毎回ランダムに点を移動させる
- 一次元のランダムウォークも可能



ランダムウォーク(一次元版)



変数定義

- 繰返し回数を宣言
- 開始位置の座標を宣言

```
let max = 1000; // ループ上限
let x = [0]; // xの初期値を0にする
let y = [0]; // yの初期値0にする
```

ループ

- 現在位置に乱数を加算する
  - Math.random() - 0.5により、-5～0.4999...の範囲の乱数が発生する
  - つまり、上下左右ランダムに座標が移動する

```
x.push(x[i] + Math.random() - 0.5);
y.push(y[i] + Math.random() - 0.5);
```

プロット

- グラフ用のデータが揃ったらplot命令で図をプロット

```
let trace = {
    x: x,
    y: y,
    mode: "lines",
    type: "scatter",
    marker:{
        color:"blue",
        size:2
    }
};
let data = [trace];

Plotly.newPlot(graph, data, layout);
```

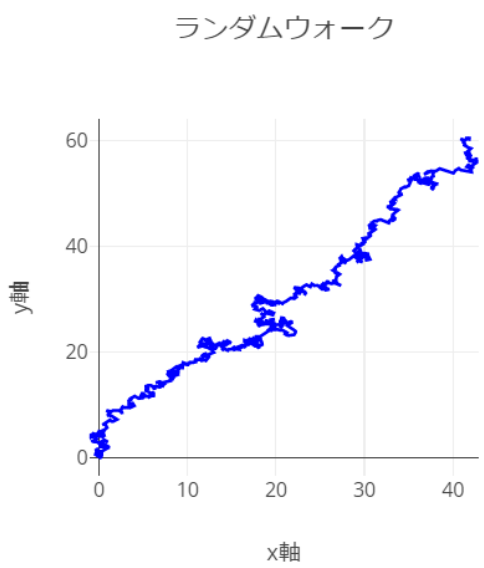
プロット(一次元版)

- x座標の情報を渡さなければ一次元のランダムウォークを描画できます

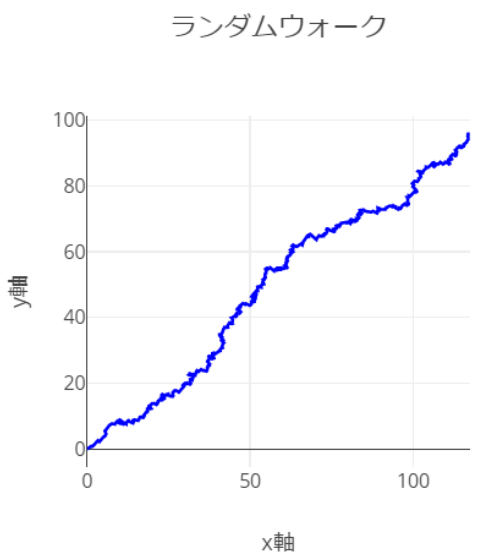
```
let trace = {
    y: y,
    mode: "lines",
    type: "scatter",
    marker:{
        color:"blue",
        size:2
    }
};
let data = [trace];

Plotly.newPlot(graph, data, layout);
```

乱数条件の変更

- -0.5ではなく-0.45などにすることで、右方向や上方向寄りのランダムウォークがおこなえます。



-0.45



-0.4