電気電子計算工学及演習

三軒家 佑将（さんげんや ゆうすけ）
3 回生
1026-26-5817
a0146089

以下のレポートにおいては、プログラミング言語として Go 言語 (https://goo.gl/pclkeC) を用いた。

また、ソースコードは巻末にまとめて添付した。

1　採用したアルゴリズム

1.1　行列とベクトルの演算を行う関数の実装

行列ベクトル積の演算を行う関数 MatVec と、行列積の演算を行う関数 MatMlt を、ソースコード 1 のとおりに実装した。

MatVec

行列ベクトル積の演算は、「引数の行列の各行ベクトル」と、引数のベクトルの内積を並べたものと考えることができる。この考えに則り、ベクトルの内積を計算する補助関数 dot を定義し、それを用いて行列ベクトルの計算を行った。

MatMlt

行列積の演算は、引数 1 の行列と「引数 2 の行列の各列ベクトル」の行列ベクトル積を並べたものと考えることができる。この考えに則り、MatVec 関数を用いて行列積の計算を行った。

1.2　バッタ G の移動

ある時刻 t の存在確率ベクトル（地点 0,1,2,3,4,5 にいる確率を並べたもの）$\mathbf{p}_t$ は、

$$\mathbf{p}_t = \mathbf{A}\mathbf{p}_{t-1}$$

によって求めることができる。ただし $\mathbf{A}$ は、表 1 の数値部分を行列と考え、さらにそれの転置を取ったものである。

	一秒後にこの地点に移動する確率					
現在地	地点 0	地点 1	地点 2	地点 3	地点 4	地点 5
地点 0	s+(1-s)(1-c)	(1-s)c	0	0	0	0
地点 1	(1-s)(1-c)	s	(1-s)c	0	0	0
地点 2	0	(1-s)(1-c)	s	(1-s)c	0	0
地点 3	0	0	(1-s)c	s	(1-s)(1-c)	0
地点 4	0	0	0	(1-s)c	s	(1-s)(1-c)
地点 5	0	0	0	0	(1-s)c	s+(1-s)(1-c)

表 1　バッタ G の振る舞い

この考え方に則って、バッタ G が、ある時刻 $0 \leq t \leq 60$ に地点 0,1,2,3,4,5 にいる確率を計算した (ソースコード 2)。

1.3　パラメータ c によるバッタ G の振る舞いの変化

ソースコード 3 のプログラムを用いて、s=0.15 とし、c=0.7,0.5,0.45 の 3 つの場合について、前節と同様の考え方に則り、時刻 $0 \leq t \leq 60$ において各地点に G がいる確率を計算した。

1.4　ニュートン法による非線形方程式の解

以下の 2 つの非線形方程式について、ソースコード 4 のプログラムを用いて、ニュートン法によって定められた範囲の解を求めた。

$$-2.2x^4 + 3.5x^3 + 4.1x^2 + 3.3x - 2.7 = 0, (0 \leq x \leq 1) \tag{1}$$
$$-\cos(2x+2) + \exp(x+1) - 2x - 30 = 0, (0 \leq x \leq \pi) \tag{2}$$

大部分は授業で示されたアルゴリズムを実装しただけである。ただし、終了条件については以下のようにした。

- 修正量が ϵ より小さくなり、かつ、f(x) が ϵ より小さくなったとき終了
- 上の条件が満たされないまま、$k = 10$ まで近似解を求めたとき終了

また、ϵ としては、Go 言語の float64 型の中で、最も絶対値が小さい値である math.SmallestNonzeroFloat64 を用いた。これにより、修正量と f(x) の値がどちらも 0 とならない限りは、後者の条件によって終了することになる。

前者のような終了条件を設定したのは、片方の条件のみでは収束しているように見えても、もう片方の条件を見ると収束していないような場合があると考えられるからである。また、後者のような終了条件を設定したのは、実際にはどちらの方程式も $k = 10$ 程度までで x の値が収束したから

である。

2　課題を解くために作成したプログラムに関する情報

プログラムのファイル名については、各ソースコード中に記載した。

また、どのプログラムに関しても、 ”go run *filename*” コマンドにて実行できる。

定義した関数の機能や呼び出し方法は、ソースコード中にコメントで注釈があるので、それを参照のこと。

使用上の注意としては、ソースコード 1,2 の冒頭は、”package matmlt” などとなっているが、これを単体として実行するときには、この部分を”package main” に書き換える必要がある。

また、上記の注意とは別に、ソースコード 2,3,4 は、他のソースコードをモジュールとして読み込んで動作するので、単体で動作させることはできない。また、ソースコード 2,3 を実行するためには、CSV ファイルを書き出すディレクトリとして、実行時のディレクトリ以下に’res’ という名前のディレクトリが存在していないといけない。これらの理由から、正しく動作させるためには、以下のようなディレクトリ構成とファイルの配置にする必要がある。

```
.
├── 1_3.go
├── 1_4.go
├── locust
│   └── locust.go
├── matmul
│   └── matmul.go
└── res
```

3　結果

3.1　行列とベクトルの演算を行う関数の実装

ソースコード 1 に含まれる動作確認により、MatVec 関数と MatMlt 関数が正常に動作していそうなことを確認した。

3.2　バッタ G の移動

ある時刻 t において、地点 1, 4 にバッタ G がいる確率をグラフに描画したのが、図 1,2,3 である。

図 1,2,3 を見ると、s が大きくなるに連れて、オーバーシュートが大きくなり、また、定常値に落ち着くまでの時間が長くなっている事がわかる。

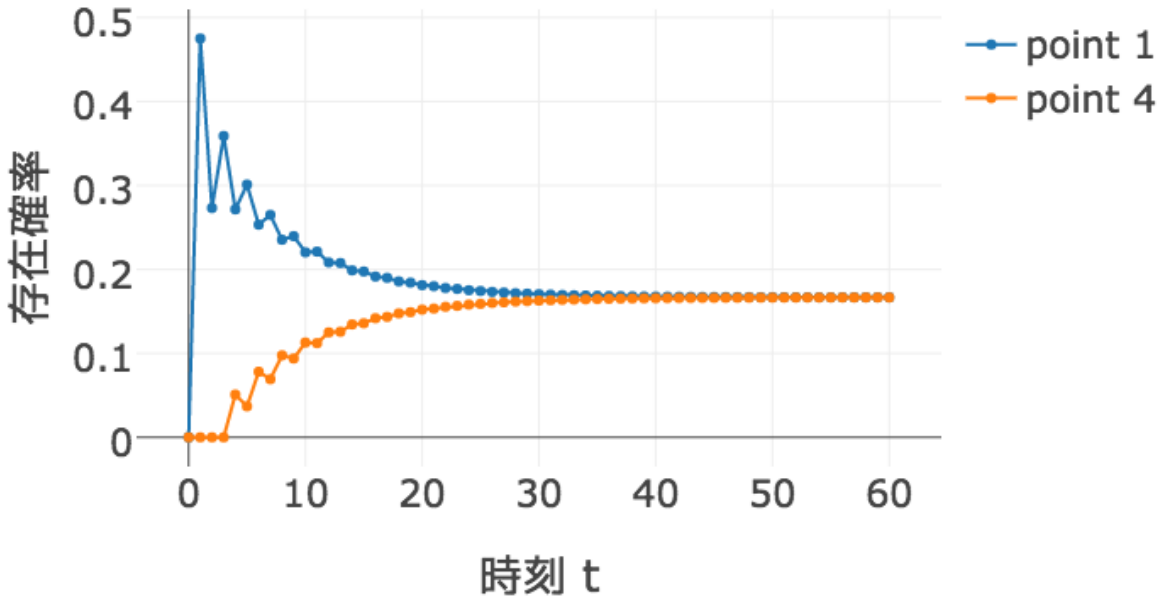

図 1　時刻 t における地点 1,4 でのバッタ G の存在確率 (s=0.05)

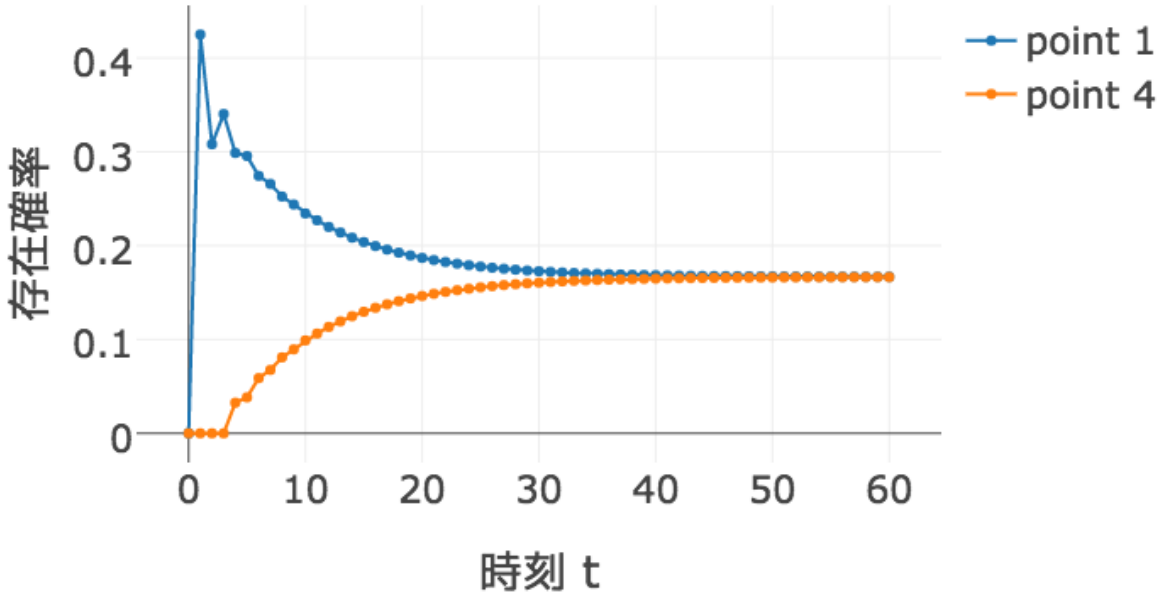

図 2　時刻 t における地点 1,4 でのバッタ G の存在確率 (s=0.15)

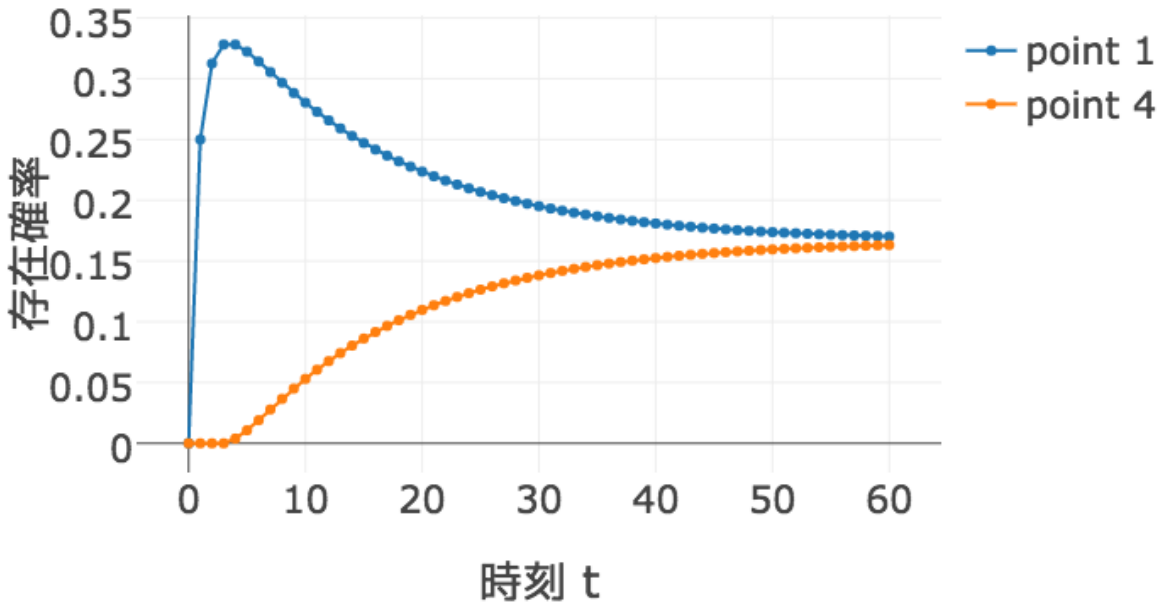

図 3　時刻 t における地点 1,4 でのバッタ G の存在確率 (s=0.5)

3.3　パラメータ c によるバッタ G の振る舞いの変化

$t = 60$ において各地点に G がいる確率をグラフにしたのが、図 4,5,6 である。

図 4,5,6 を見ると、c が大きくなるに連れて、中央に近い地点の存在確率が高くなっていることがわかる。

3.4　ニュートン法による非線形方程式の解

計算結果として、

- (1) に対しては $x = 0.4685126936655117$
- (2) に対しては $x = 2.6107790395825665$

という解が得られた。

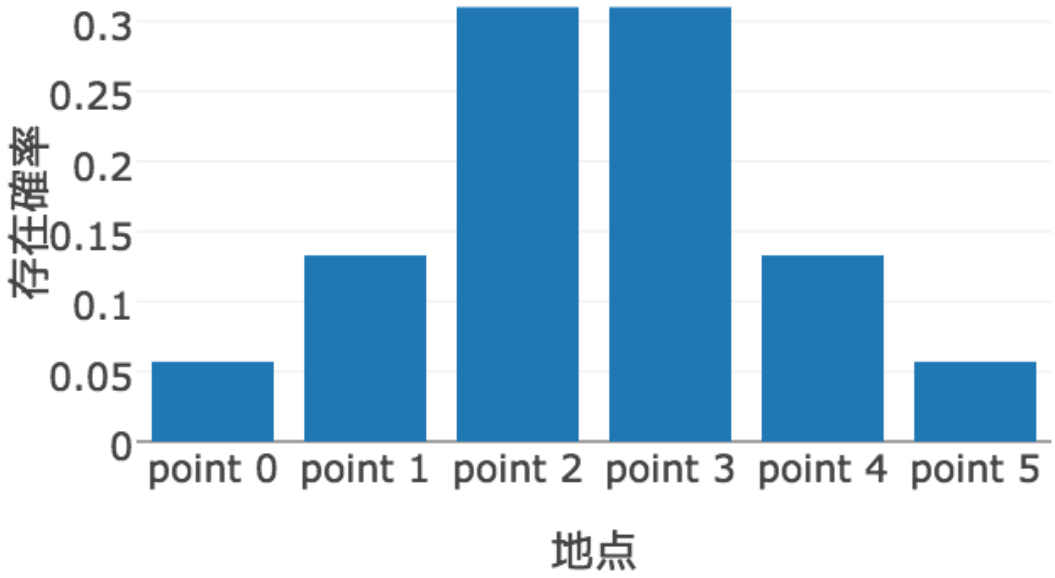

図 4　t=60 における各地点での G の存在確率 (c=0.7)

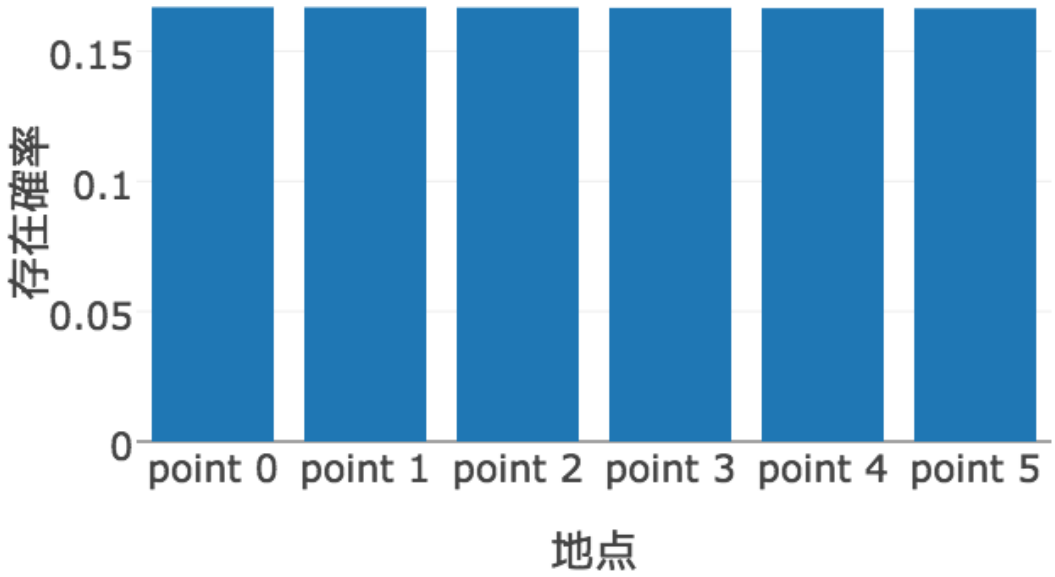

図 5　t=60 における各地点での G の存在確率 (c=0.5)

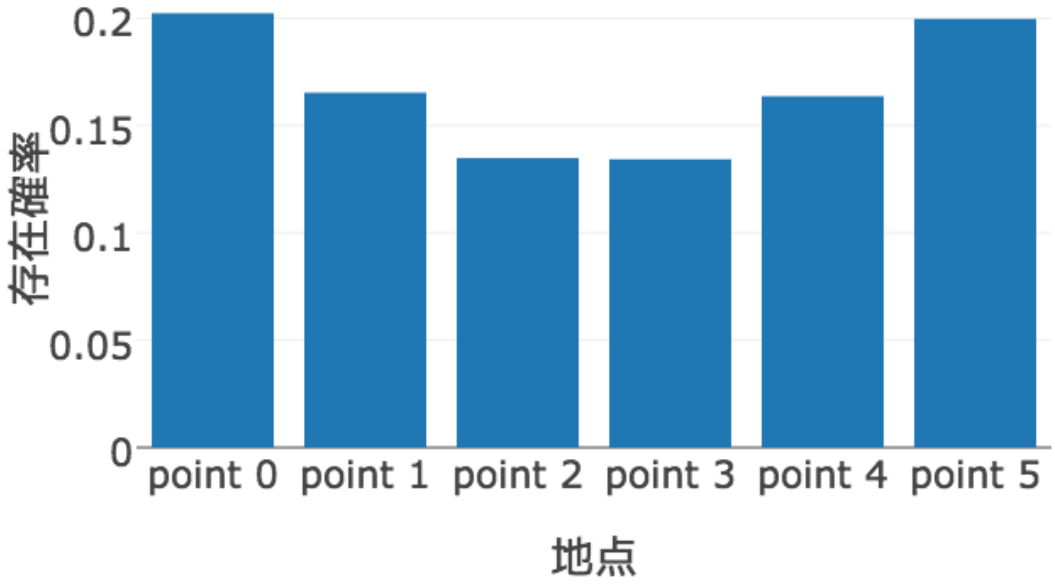

図 6　t=60 における各地点での G の存在確率 (c=0.45)

4　考察

4.1　行列とベクトルの演算を行う関数の実装

理解しやすいように、またデバッグがしやすいように、抽象度の高い関数を組み合わせて 2 つの関数を実装したが、科学技術計算において実用レベルのものとしては速度が不安である。現実の行列演算ライブラリとして用いられるものを実装するのであれば、C 言語で拡張ライブラリを作成する、関数呼び出しを伴わずに実装する、などの工夫が必要となると考えられる。

4.2　バッタ G の移動

表 1 を見ると、s というパラメータは、前の時刻にいた場所に、次の時刻にもいる確率を表していることがわかる。その場にとどまる確率 (s) が低いほど、t が小さいときに G が地点 0 から地点 1 に移動してくる確率が高いため、オーバーシュートが大きくなると考えられる。また、その場にとどまる確率 (s) が高いほど、ある状態から次の状態に移行する確率が低くなり、変化にはより多くの時間がかかると考えられる。

4.3 パラメータ c によるバッタ G の振る舞いの変化

表 1 について、各列の確率を足し合わせると、

- 点 0,5 に移動してくる確率のそれぞれの合計 $P_x = s + 2(1-s)(1-c)$
- 点 1,4 に移動してくる確率のそれぞれの合計 $P_y = s + 2(1-s)1 = 1$
- 点 2,3 に移動してくる確率のそれぞれの合計 $P_z = s + 2(1-s)c$

のように表せる。

これを見るとわかるように、$c = 0.5$ のとき、P_x, P_y, P_z のどれについても 1 となるので、定常状態においては、どの点にいる確率も等しくなると予想できる。

また、$c > 0.5$ のとき、

$$P_x < P_y < P_z$$

となるので、定常状態においては、点 0,5 にいる確率が一番小さく、点 2,3 にいる確率が一番大きくなると予想できる。

同様に、$c < 0.5$ のとき、

$$P_x > P_y > P_z$$

となることから、それぞれ、定常状態においては、点 0,5 にいる確率が一番大きく、点 2,3 にいる確率が一番小さくなると予想できる。

4.4 ニュートン法による非線形方程式の解

以下、便利のため

$$f_1(x) = -2.2x^4 + 3.5x^3 + 4.1x^2 + 3.3x - 2.7$$
$$f_2(x) = -\cos(2x+2) + \exp(x+1) - 2x - 30$$

とおく。

f_1 は、$|f_1(x_k)| < \epsilon$ も $|x_k - x_{k-1}| < \epsilon$ も満たして終了している。f_1 については、$f_1(x)$ が 0 になってしまったため、それ以上の修正が行われなくなったと考えられる。

f_2 は、$|x_k - x_{k-1}| < \epsilon$ は満たされているが、$|f_2(x_k)| < \epsilon$ は満たされず、実行回数の上限によって終了している。しかし、終了直前については、x の値は変化しておらず、float64 型の精度の上限いっぱいまで正確な近似解を求めていると考えられる。

このことから、$|f_2(x_k)| < \epsilon$ が満たされないのは、f_2 の中には指数関数が入っているため、x の小さな変化に対しても、鋭敏に $f_2(x)$ の値が変化してしまうことが原因であると考えられる。float64 型の誤差は 10 進数でおおよそ 10^{-16} 程度であるので、$|x_k - x_{k-1}| < \epsilon$ が成立する x_k の近傍での f_2 の誤差 Δf_2 は、

$$\Delta f_2 \approx \frac{\partial f_2}{\partial x} \Delta x$$

$$\approx e^{2.6107790395825665+1} \times 10^{-16}$$
$$\approx 3.7 \times 10^{-15}$$

となり、10^{-15} 程度は発生することになる。実際、$|x_k - x_{k-1}| < \epsilon$ が満たされたときの $|f(x)|$ の値は、10^{-15} 程度の値であり、f_2 の誤差程度の値までは小さくなっていることがわかる。

```go
// 後で他のソースコードから関数をモジュールとして参照するので、パッケージをexportしている。
// Go言語では、初めの文字が大文字の関数が公開され、小文字の関数は公開されない。
package matmul

// Println関数を使うためにパッケージをimportしている。
import "fmt"

// #define N 6 の代わり
// main関数内の動作確認を作動させるためにはN=3で十分だが、
// 後のバッタGの移動に際してはN=6である必要があるため、ここではN=6としている。
const N = 6

/* これ以降、行列の添字としては、行 => i, 列 => jを用いる。
    ex:)

    行列を、
        [[1,2,3]
        ,[4,5,6]
        ,[7,8,9]]
    として、6を参照するには
        i = 1
        j = 2
    とする。
*/

// この関数の中で動作確認をしている
func main() {
    // 行列を定義
    /*
        |1 4 7|
        |2 5 8|
        |3 6 9|
    */
    A := [N][N]float64{
        {1, 4, 7},
        {2, 5, 8},
        {3, 6, 9},
    }
    // ベクトルを定義
    b1 := [N]float64{1, 2, 3}
    b2 := [N]float64{4, 5, 6}
    b3 := [N]float64{7, 8, 9}

    // 行列ベクトル積を計算し、
    // 結果のベクトルから行列を作成している
    /* 転置を取っていることについて。
        転置を取らずに行列を作成すると、行列ベクトル積の結果をv1, v2, v3としたとき、
            (v1 v2 v3)
        のような行列が作成されてしまう。得たい行列は
            |v1|
            |v2|
            |v3|
        のような行列である。この２つの行列は、転置の関係にあるので、転置を取ってA(b1 b2 b3)の計算結果としている。
    */
    Abs := Transpose([N][N]float64{
        MatVec(A, b1),
        MatVec(A, b2),
        MatVec(A, b3),
    })

    // 行列積を計算
    AA := MatMlt(A, A)

    // Goの==は、Arrayに適用された場合は、
    // ２つのArrayの対応する要素の値がすべて等しいときにtrueを返す。
    // 参考： https://golang.org/ref/spec#Comparison_operators
    if (AA == Abs) {
        fmt.Println("AA == Abs")
    }

    // 行列積を取ってからベクトル積を取った場合
    AA_b1 := MatVec(MatMlt(A, A), b1)
```

```
    // ベクトル積を取ってから、更にもう一度ベクトル積を取った場合
    A_Ab1 := MatVec(A, MatVec(A, b1))

    // Goの==は、Arrayに適用された場合は、
    // ２つのArrayの対応する要素の値がすべて等しいときにtrueを返す。
    if (AA_b1 == A_Ab1) {
        fmt.Println("AA_b1 == A_Ab1")
    }
}

// 行列ベクトル積を計算する関数
func MatVec(mat [N][N]float64, vec [N]float64) [N]float64 {
    var ret_vec [N]float64

    for i := 0; i < N; i++ {
        // 行列の各行ベクトルと、ベクトルの内積を取る
        ret_vec[i] = dot(mat[i], vec)
    }

    return ret_vec
}

// 行列積を計算する関数
func MatMlt(mat1 [N][N]float64, mat2 [N][N]float64) [N][N]float64 {
    var ret_mat [N][N]float64

    for k := 0; k < N; k++ {
        // 行列1と、行列2のj列ベクトルの行列ベクトル積が、
        // 返す行列のj列成分であることを利用して計算している
        ret_mat[k] = MatVec(mat1, fetch_col(mat2, k))
    }

    /*
        ret_mat[k] = MatVec(mat1, fetch_col(mat2, k))
        としているので、本来列ベクトルとなるはずのMatVec(mat1, fetch_col(mat2, k))が、
        行ベクトルとしてret_matに格納されているため、
        ret_matはこの時点では、求める行列を転置したものである。
        したがって、計算結果として返却するのは、ret_matの転置を取ったものでなければならない。
    */
    return Transpose(ret_mat)
}

// ベクトルの内積を取る関数
func dot(vec1 [N]float64, vec2 [N]float64) float64 {
    var sum float64 = 0
    for n := 0; n < N; n++ {
        sum += vec1[n] * vec2[n]
    }
    return sum
}

// 行列から列ベクトルを取り出す関数
func fetch_col(mat [N][N]float64, col_ind int) [N]float64 {
    var col [N]float64

    for i := 0; i < N; i++ {
        col[i] = mat[i][col_ind]
    }

    return col
}

// 行列の転置を取る関数
func Transpose(mat [N][N]float64) [N][N]float64 {
    var ret_mat [N][N]float64

    for i := 0; i < N; i++ {
        for j := 0; j < N; j++ {
            ret_mat[i][j] = mat[j][i]
        }
    }

    return ret_mat
}
```

```go
// 後で他のソースコードから関数をモジュールとして参照するので、パッケージをexportしている。
// Go言語では、初めの文字が大文字の関数が公開され、小文字の関数は公開されない。
package locust

import (
    "../matmul" // MatVec関数, MatMlt関数, Transpose関数, 定数N
    "encoding/csv" // 計算結果をCSVに出力するため
    "fmt" // Sprint関数を使うため
    "os" // 計算結果をCSVに出力するため
)

// #define N 3 の代わり。
const N = matmul.N

func main() {
    // c = 0.5, s = 0.05の場合の、各時刻における各地点の、バッタGの存在確率を計算している
    res1 := Calculation(0.5, 0.05)
    // c = 0.5, s = 0.15の場合の、各時刻における各地点の、バッタGの存在確率を計算している
    res2 := Calculation(0.5, 0.15)
    // c = 0.5, s = 0.5の場合の、各時刻における各地点の、バッタGの存在確率を計算している
    res3 := Calculation(0.5, 0.5)

    // 計算結果をCSVに出力している
    WriteCSV("0.05", res1)
    WriteCSV("0.15", res2)
    WriteCSV("0.5", res3)
}

// 書き出すファイル名と、書き出す内容を含んだArrayを受け取り、
// CSVファイルに書き出す関数
func WriteCSV(filename string, resVec [61][N]float64) {
    // CSVを書き出すファイルを指定
    file, _ := os.Create("res/" + filename + ".csv")
    writer := csv.NewWriter(file)
    // CSVのヘッダーを指定
    writer.Write([]string{
        "t",
        "point 0",
        "point 1",
        "point 2",
        "point 3",
        "point 4",
        "point 5",
    })
    for t := 0; t <= 60; t++ {
        // writer.Write 関数はstringのArrayしか受け取らないので、
        // resVec[t]の各要素をfmt.Sprint関数でstringに変換して、
        // そのArrayを生成し、それをwriter.Writeに渡している。
        writer.Write(
            []string{
                fmt.Sprint(t),
                fmt.Sprint(resVec[t][0]),
                fmt.Sprint(resVec[t][1]),
                fmt.Sprint(resVec[t][2]),
                fmt.Sprint(resVec[t][3]),
                fmt.Sprint(resVec[t][4]),
                fmt.Sprint(resVec[t][5]),
            }
        )
    }
    // バッファをファイルへ出力する
    writer.Flush()
}

// パラメーターcとsを受け取り、0 <= t <= 60の範囲のバッタGの
// 各地点での存在確率を計算する
func Calculation(c float64, s float64) [61][N]float64 {
    // 移動確率を表す行列を計算し、取得する
    var probMat [N][N]float64 = getProbMat(c, s)

    var resultVectors [61][N]float64 // resultVectors[t]: 時刻tに於ける、Gの地点0~5での存在確率を表すベクトル
    resultVectors[0] = [N]float64{1, 0, 0, 0, 0, 0} // t = 0のとき、バッタGの地点0~5での存在確率は
```

```
1,0,0,0,0,0である
    // t = 1, 2, ... , 60について、
    // Gの存在確率ベクトルを求める
    for t := 1; t <= 60; t++ {
        // ある時刻tのGの各地点での存在確率は、
        // 移動確率を表す行列と、t-1での存在確率ベクトルの、行列ベクトル積である。
        resultVectors[t] = matmul.MatVec(probMat, resultVectors[t-1])
    }
    return resultVectors
}

// パラメーターcとsを受け取り、バッタの移動確率を表す行列の値を計算する
func getProbMat(c float64, s float64) [N][N]float64 {
    // 問に示された「バッタGの振る舞い」行列を転置したものと、
    // 時刻tでのバッタの存在確率ベクトルとの行列ベクトル積を取ると、
    // その計算結果が時刻t+1でのバッタの存在確率ベクトルとなっている。
    return matmul.Transpose([N][N]float64{
        {s + (1-s)*(1-c), (1 - s) * c, 0, 0, 0},
        {(1 - s) * (1 - c), s, (1 - s) * c, 0, 0, 0},
        {0, (1 - s) * (1 - c), s, (1 - s) * c, 0, 0},
        {0, 0, (1 - s) * c, s, (1 - s) * (1 - c), 0},
        {0, 0, 0, (1 - s) * c, s, (1 - s) * (1 - c)},
        {0, 0, 0, 0, (1 - s) * c, s + (1-s)*(1-c)},
    })
}
```

```go
package main

import (
    "./locust" // Calculation関数, WriteCSV関数
    "fmt" // Sprint関数
)

func main() {
    // s=0.15, c=0.7, 0.5, 0.45
    // の各場合について、バッタGの移動確率を求めている
    locust.WriteCSV(
        "c_0.7",
        locust.Calculation(0.7, 0.15)
    )
    locust.WriteCSV(
        "c_0.5",
        locust.Calculation(0.5, 0.15)
    )
    locust.WriteCSV(
        "c_0.45",
        locust.Calculation(0.45, 0.15)
    )
}
```

```go
package main

import (
    "fmt"  // Println関数
    "math" // SmallestNonzeroFloat64定数,
)

func main() {
    // math.SmallestNonzeroFloat64は、
    // 4.940656458412465441765687928682213723651e-324
    // を表す定数であり、Goのfloat64型の値の中で絶対値が最小のものである
    fmt.Println("f1")
    newton(f1, f1_prime, math.SmallestNonzeroFloat64, 0.1)
    fmt.Println("f2")
    newton(f2, f2_prime, math.SmallestNonzeroFloat64, 0.1)
}

// newton(f, f_prime, e, x0)の引数
//   f       : 対象関数
//   f_prim  : fの一次導関数
//   e       : "|x_(k+1) - x_k| < e && f(x_(k+1)) < e" が満たされたときに終了する
//   x0      : x0
// 表示される値
//   k    : 終了条件が満たされた、最初のkの値
//   x    : x_kの値
//   f(x) : f(x_k)の値
// 挙動の解説
//   1. "|x_(k+1) - x_k| < e && f(x_(k+1)) < e" が初めて満たされたときに、その時点でのk, x, f(x)を終了する。
//   2. "|x_(k+1) - x_k| < e"が満たされたとき、その時点でのk, x, f(x)を表示する。また、表示される行の末尾に":: delta x < e satisfied"が追加される。
//   3. "|f(x_(k+1))| < e"が満たされたとき、その時点でのk, x, f(x)を表示する。また、表示される行の末尾に":: f(x) < e satisfied"が追加される。
// なお、ステップの実行回数は10回までとした。
func newton(f func(float64) float64, f_prime func(float64) float64, e float64, x0 float64) {
    /*
        各ステップでのx_kを表しているのがx_prev変数
        各ステップでのx_k+1を表しているのがx_next変数
    */
    var x_prev float64
    var x_next float64

    k := 0
    x_prev = x0
    for k < 10 {
        // 修正量を計算し、それから反復列の次の近似解を求める
        x_next = x_prev - f(x_prev)/f_prime(x_prev)

        // 途中経過を表示する
        if math.Abs(f(x_next)) < e {
            // k, x, f(x)を表示する
            fmt.Printf("k=%v, x=%v, f(x)=%v :: f(x) < e satisfied\n", k, x_next, f(x_next))
        }
        if math.Abs(x_next-x_prev) < e {
            // k, x, f(x)を表示する
            fmt.Printf("k=%v, x=%v, f(x)=%v :: delta x < e satisfied\n", k, x_next, f(x_next))
        }

        // 終了条件を両方共満たしたとき終了する
        if math.Abs(x_next-x_prev) < e && math.Abs(f(x_next)) < e {
            break
        }
        x_prev = x_next
        k++
    }
    // 終了時のk, x, f(x)を表示する
    fmt.Printf("k=%v, x=%v, f(x)=%v :: finish\n", k, x_next, f(x_next))
}

// 問に挙げられた方程式のうち、4次方程式の方を表す関数
// xを受け取り、関数を評価した値を返す
func f1(x float64) float64 {
    return -2.2*math.Pow(x, 4) + 3.5*math.Pow(x, 3) + 4.1*math.Pow(x, 2) + 3.3*x - 2.7
```

```
71 }
72
73 // f1の一次導関数を表す
74 func f1_prime(x float64) float64 {
75     return -8.8*math.Pow(x, 3) + 10.5*math.Pow(x, 2) + 8.2*x + 3.3
76 }
77
78 // 問に挙げられた方程式のうち、余弦関数を含む方を表す関数
79 // xを受け取り、関数を評価した値を返す
80 func f2(x float64) float64 {
81     return -3*math.Cos(2*x+2) + math.Exp(x+1) - 2*x - 30
82 }
83
84 // f2の一次導関数を表す
85 func f2_prime(x float64) float64 {
86     return 6*math.Sin(2*x+2) + math.Exp(x+1) - 2
87 }
```