```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
}
```