Unit 0.3 Vector and Matrix Classes

Numerical Analysis

EE/NTHU

Mar. 9, 2020

VEC and MAT Classes

- Two more class examples are given below.
  - VEC class for vectors.
  - MAT class for matrices.
- These classes can be used in most of the topics of this course.
- Basic functions of these two classes are declared in this section.
- Examples of function definitions are also given.
  - You should be able to complete all function definitions yourself.
- More functions will be added during this course to solve different problems.
- These two example classes can be used to implement numerical algorithm in a more direct way.
  - But, the efficiency can still be improved
  - It is a challenge to you to improve the efficiency.

VEC.h (1/2)

```
// vector class
#ifndef VEC_H
#define VEC_H
class VEC {
  private:
    int dim;                        // vector length
    double *val;                    // array to store vector
  public:
    VEC(int n);                     // uninit constructor, val set to 0
    VEC(const VEC &v1);             // copy constructor
    VEC(int n, double *v);          // init constructor
    ~VEC();                         // destructor
    int len();                      // dimension of the vector
    VEC &operator-();               // unary operator, negative value
    VEC &operator=(const VEC v1);   // assignment
    VEC &operator+=(const VEC v1);  // V += v1;
    VEC &operator-=(const VEC v1);  // V -= v1;
    VEC &operator*=(double a);      // V *= dbl;
    VEC &operator/=(double a);      // V /= dbl;
```


VEC.h (2/2)

```
    VEC operator+(const VEC v1);    // V + v1
    VEC operator-(const VEC v1);    // V - v1
    double operator*(VEC v1);       // inner product
    VEC operator*(double a);        // V * dbl
    VEC operator/(double a);        // V / dbl
    double &operator[](int n);      // indexing
    friend VEC operator*(double a, const VEC v1);     // dbl x V
    friend VEC *newVEC(int n);      // create dynamic VEC
};
VEC operator*(double a,const VEC v1);
VEC *newVEC(int n);                 // create dynamic VEC
#endif
```

- Example of a vector class declaration.
- The length of the vector is not fixed, thus the array needs to be allocated using dynamic memory allocation.
- Note the declaration of indexing operator `[]`.
  - This enable the accessing of a single element of the vector for both `read` and `write` operations.

VEC.cpp (1/4)

```
// VEC class functions
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "VEC.h"
VEC::VEC(int n)                        // uninit constructor
{
    dim = n;
    val = (double *)calloc(n, sizeof(double));
}
VEC::VEC(const VEC &v1)                // copy constructor
{
    dim = v1.dim;
    val = (double *)calloc(dim, sizeof(double));
    for (int i = 0; i < dim; i++) {
        val[i] = v1.val[i];
    }
}
```


VEC.cpp (2/4)

```
VEC::VEC(int n, double *v)       // init constructor
{
    dim = n;
    val = (double *)calloc(n, sizeof(double));
    for (int i = 0; i < n; i++) val[i] = v[i];
}
VEC::~VEC()                      // destructor
{
    free(val);
}
int VEC::len()                   // return dimension of the vector
{
    return dim;
}
VEC &VEC::operator-()            // unary operator - : negative value
{
    for (int i = 0; i < dim; i++) val[i] = -val[i];
    return *this;
}
```

VEC.cpp (3/4)

```
VEC &VEC::operator=(const VEC v1)   // assignment
{
    dim = v1.dim;
    for (int i = 0; i < dim; i++) {
        val[i] = v1.val[i];
    }
    return *this;
}
VEC &VEC::operator+=(const VEC v1)  // V += v1
{
    for (int i = 0; i < dim; i++) {
        val[i] += v1.val[i];
    }
    return *this;
}
VEC VEC::operator+(const VEC v1)    // V + v1
{
    VEC s(*this);
    for (int i = 0; i < dim; i++) s.val[i] += v1.val[i];
    return s;
}
```


VEC.cpp (4/4)

```
double &VEC::operator[](int n)      // indexing
{
    if (n < 0) n = 0;
    else if (n >= dim) n = dim - 1;
    return val[n];
}
VEC *newVEC(int n)                  // allocate a dynamic VEC
{
    VEC *vptr;
    vptr = (VEC *)malloc(sizeof(VEC));
    vptr->dim = n;
    vptr->val = (double*)calloc(n, sizeof(double));
    return vptr;
}
```

- Using return type of `double &`, the specific entry of the vector is returned.
  - Not its value.
  - Thus the entry can be modified, in addition to simple read access.

```
    VEC v1(10);             // uninit constructor, all entries set to 0
    for (int i = 0; i < 10; i++)
        v1[i] = i;          // change the values of the vector
```

MAT.h (1/2)

```
// matrix class
#ifndef MAT_H
#define MAT_H
#include "VEC.h"
class MAT {
  private:
    int n;                              // define nxn matrix
    VEC **va;                           // array of n pointers to vectors
  public:
    MAT(int dim);                       // uninit constructor
    MAT(const MAT &m1);                 // copy constructor
    MAT(int dim, double *v);            // init constructor
    ~MAT();                             // destructor
    int dim();                          // return dimension of the matrix
    MAT tpose();                        // transpose
    MAT &operator-();                   // unary operator, negative value
    MAT &operator=(MAT m1);             // assignment
    MAT &operator+=(MAT &m1);           // m += m1;
    MAT &operator-=(MAT &m1);           // m -= m1;
    MAT &operator*=(double a);          // m *= dbl;
```


MAT.h (2/2)

```
    MAT &operator/=(double a);               // m /= dbl;
    MAT operator+(MAT m1);                   // m1 + m2
    MAT operator-(MAT m1);                   // m1 - m2
    MAT operator*(MAT m1);                   // m1 * m2
    VEC & operator[](int m);                 // m'th row
    VEC operator*(VEC v1);                   // m x v1
    MAT operator*(double a);                 // m * dbl
    MAT operator/(double a);                 // m / dbl
    friend MAT operator*(double a, MAT &m1); // dbl x m
    friend VEC operator*(VEC &v1, MAT &m1);  // vT x m
};
MAT operator*(double a, const MAT &m1);      // dbl x m
VEC operator*(VEC &v1, MAT &m1);             // vT x m
#endif
```

- Note the indexing operator returns a reference to the VEC
- Can combine with VEC indexing function to access an element of an array

$$\texttt{A[i][j]} = A_{ij}$$

MAT.cpp (1/6)

- Example function definitions

```
// MAT class functions
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "MAT.h"
MAT::MAT(int dim)                    // uninit constructor
{
    n = dim;
    va = (VEC **)malloc(n * sizeof(VEC*));
    for (int i = 0; i < n; i++) {
        va[i] = newVEC(n);
    }
}
```


MAT.cpp (2/6)

```
MAT::MAT(const MAT &m1)              // copy constructor
{
    VEC **vsrc = m1.va;          // to get around not indexing const MAT
    n = m1.n;
    va = (VEC **)malloc(n * sizeof(VEC*));
    for (int i = 0; i < n; i++) {
        va[i] = newVEC(n);
        (*va[i]) = (*vsrc[i]);       // VEC assignment
    }
}
MAT::MAT(int dim, double *v)         // init constructor
{
    n = dim;
    va = (VEC **)malloc(n * sizeof(VEC*));
    for (int i = 0; i < n; i++) {
        va[i] = newVEC(n);
        for (int j = 0; j < n; j++) {
            (*va[i])[j] = *(v++);    // array indexing + VEC indexing
        }
    }
}
```

MAT.cpp (3/6)

```
MAT::~MAT()                          // destructor
{
    for (int i = n - 1; i >= 0; i--)
        (*va[i]).~VEC();
    free(va);
}
int MAT::dim()                       // return dimension of the matrix
{
    return n;
}
MAT MAT::tpose()                     // matrix transpose
{
    MAT mnew(n);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            mnew[i][j] = (*va[j])[i];
        }
    }
    return mnew;
}
```


MAT.cpp (4/6)

```
MAT &MAT::operator-()                // unary operator - : negative value
{
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            (*va[i])[j] = -(*va[i])[j];
    return *this;
}
MAT &MAT::operator=(MAT m1)          // assignment
{
    for (int i = 0; i < n; i++)
        (*va[i]) = m1[i];            // VEC assignment
    return *this;
}
MAT &MAT::operator+=(MAT &m1)        // m += m1
{
    for (int i = 0; i < n; i++)
        (*va[i]) += m1[i];           // VEC += operation
    return *this;
}
```

MAT.cpp (5/6)

```
MAT MAT::operator+(MAT m1)          // addition
{
    MAT s(n);
    for (int i = 0; i < n; i++)
        s[i] = (*val[i]) + m1[i];   // VEC addition and assignment
    return s;
}
MAT MAT::operator*(MAT m1)          // matrix-matrix product
{
    MAT z(n);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            z[i][j] = 0;
            for (int k = 0; k < n; k++)
                z[i][j] += ((*va[i])[k] * m1[k][j]);
        }
    }
    return z;
}
```


MAT.cpp (6/6)

```
VEC MAT::operator*(VEC v1)          // M * v
{
    VEC s(n);
    for (int i = 0; i < n; i++) {
        s[i] = (*va[i]) * v1;       // VEC inner product
    }
    return s;
}
VEC operator*(VEC &v1, MAT &m1)     // vT x M
{
    VEC v2(m1.n);
    for (int i = 0; i < m1.n; i++) {
        v2[i] = 0;
        for (int j = 0; j < m1.n; j++) {
            v2[i] += v1[j] * m1[j][i];
        }
    }
    return v2;
}
```

MAT Class

- Using these VEC and MAT classes most of the operators for vector and matrix operations are defined.
- Given an $n \times n$ matrix $\mathbf{A}$, an n-vector $\mathbf{r}$ and a real number α, then the equation

$$\alpha = \frac{\mathbf{r}^T\mathbf{r}}{\mathbf{r}^T\mathbf{A}\mathbf{r}}$$

  can be coded as

```
VEC r(n);    // r should be initialized properly
MAT A(n);    // A should be initialized properly
double alpha;

alpha = (r * r) / (r * (A * r));
```

- Note that the denominator can also be written as `r*A*r` with the same result and similar efficiency.
- But in some cases, parentheses should be used to get better efficiency.