Numerical Methods
Summary

September 18, 2020

Chapter 1

Computing with Matrices and Vectors

Elementary Operations: +,-,*,\, lowest level of real arithmetic available on computers usually implemented in hardware
Elementary Linear Algebra operations: The next level real arithmetic which is the computation on finite arrays of real numbers
Complex Algorithms: involves iterations and approximations

1.1 Fundamentals

$\mathbb{K}$ notation for a generic field of numbers i.e from $\mathbb{R}$ or $\mathbb{C}$

Vectors: one-dimensional array of real/complex numbers, the default for this lecture are column vectors:

$$\underbrace{\begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{K}^n}_{\text{column vector}} \quad \underbrace{\begin{bmatrix} x_1 \cdots x_n \end{bmatrix} \in \mathbb{K}^{1,n}}_{\text{row vector}}$$

Unless stated otherwise, in mathematical formulas vector components are indexed from 1

Notations:

- column vectors: bold small roman letters e.g **x,y,z**
- row vectors: x^T, y^T, z^T
- Addressing vector components:

$$x = [x_1, ..., x_n]^T \rightarrow x_i, i = 1, ..., n$$
$$x \in \mathbb{K}^n \rightarrow (x)_i, i = 1, ..., n$$

- Selecting sub-vectors: $(x)_{k:l} = [x_k, ..., x_l]^T, 1 \leq k \leq l \leq n$
- j-th unit vector: $(e_j)_i = \sigma_{ij}$ where $sigma_{ij} := 1$ if $i = j$ 0 else (Kronecker symbol)

1.2 Software and Libraries

Eigen: A header-only C++ template library designed to enable easy, natural and efficient numerical linear algebra

Header-Only A library is called header-only if the full definitions of all macros functions and classes comprising the library are visible to the compiler in a header file form. Header-only libraries do not need to be separately compiled, packaged and installed in order to be used. All that is required is to point the compiler at the location of the headers, and then include the header files into the application source

Header File form: Many programming languages and other computer files have a directive, often called inlcude (or copy, import) that causes the contents of a second file to be inserted into the original file. These included files are called copybooks or header files.

Eigen Cheet Sheet:

```
// A simple quickref for Eigen. Add anything that's missing.
// Main author: Keir Mierle

#include <Eigen/Dense>

Matrix<double, 3, 3> A;                // Fixed rows and cols. Same as Matrix3d.
Matrix<double, 3, Dynamic> B;          // Fixed rows, dynamic cols.
Matrix<double, Dynamic, Dynamic> C;    // Full dynamic. Same as MatrixXd.
Matrix<double, 3, 3, RowMajor> E;      // Row major; default is column-major.
Matrix3f P, Q, R;                      // 3x3 float matrix.
Vector3f x, y, z;                      // 3x1 float matrix.
RowVector3f a, b, c;                   // 1x3 float matrix.
VectorXd v;                            // Dynamic column vector of doubles
double s;

// Basic usage
// Eigen          // Matlab           // comments
x.size()          // length(x)        // vector size
C.rows()          // size(C,1)        // number of rows
C.cols()          // size(C,2)        // number of columns
x(i)              // x(i+1)           // Matlab is 1-based
C(i,j)            // C(i+1,j+1)       //

A.resize(4, 4);   // Runtime error if assertions are on.
B.resize(4, 9);   // Runtime error if assertions are on.
A.resize(3, 3);   // Ok; size didn't change.
B.resize(3, 9);   // Ok; only dynamic cols changed.

A << 1, 2, 3,     // Initialize A. The elements can also be
     4, 5, 6,     // matrices, which are stacked along cols
     7, 8, 9;     // and then the rows are stacked.
B << A, A, A;     // B is three horizontally stacked A's.
A.fill(10);       // Fill A with all 10's.

// Eigen                                    // Matlab
MatrixXd::Identity(rows,cols)               // eye(rows,cols)
C.setIdentity(rows,cols)                    // C = eye(rows,cols)
MatrixXd::Zero(rows,cols)                   // zeros(rows,cols)
C.setZero(rows,cols)                        // C = zeros(rows,cols)
MatrixXd::Ones(rows,cols)                   // ones(rows,cols)
C.setOnes(rows,cols)                        // C = ones(rows,cols)
MatrixXd::Random(rows,cols)                 // rand(rows,cols)*2-1            // MatrixXd::Random returns uniform random numbers in (-1, 1).
C.setRandom(rows,cols)                      // C = rand(rows,cols)*2-1
VectorXd::LinSpaced(size,low,high)          // linspace(low,high,size)'
v.setLinSpaced(size,low,high)               // v = linspace(low,high,size)'
VectorXi::LinSpaced(((hi-low)/step)+1,      // low:step:hi
                    low,low+step*(size-1))  //


// Matrix slicing and blocks. All expressions listed here are read/write.
// Templated size versions are faster. Note that Matlab is 1-based (a size N
// vector is x(1)...x(N)).
// Eigen                           // Matlab
x.head(n)                          // x(1:n)
x.head<n>()                        // x(1:n)
x.tail(n)                          // x(end - n + 1: end)
x.tail<n>()                        // x(end - n + 1: end)
x.segment(i, n)                    // x(i+1 : i+n)
x.segment<n>(i)                    // x(i+1 : i+n)
P.block(i, j, rows, cols)          // P(i+1 : i+rows, j+1 : j+cols)
P.block<rows, cols>(i, j)          // P(i+1 : i+rows, j+1 : j+cols)
P.row(i)                           // P(i+1, :)
P.col(j)                           // P(:, j+1)
P.leftCols<cols>()                 // P(:, 1:cols)
P.leftCols(cols)                   // P(:, 1:cols)
P.middleCols<cols>(j)              // P(:, j+1:j+cols)
P.middleCols(j, cols)              // P(:, j+1:j+cols)
P.rightCols<cols>()                // P(:, end-cols+1:end)
P.rightCols(cols)                  // P(:, end-cols+1:end)
P.topRows<rows>()                  // P(1:rows, :)
P.topRows(rows)                    // P(1:rows, :)
P.middleRows<rows>(i)              // P(i+1:i+rows, :)
P.middleRows(i, rows)              // P(i+1:i+rows, :)
P.bottomRows<rows>()               // P(end-rows+1:end, :)
P.bottomRows(rows)                 // P(end-rows+1:end, :)
P.topLeftCorner(rows, cols)        // P(1:rows, 1:cols)
P.topRightCorner(rows, cols)       // P(1:rows, end-cols+1:end)
P.bottomLeftCorner(rows, cols)     // P(end-rows+1:end, 1:cols)
P.bottomRightCorner(rows, cols)    // P(end-rows+1:end, end-cols+1:end)
P.topLeftCorner<rows,cols>()       // P(1:rows, 1:cols)
P.topRightCorner<rows,cols>()      // P(1:rows, end-cols+1:end)
P.bottomLeftCorner<rows,cols>()    // P(end-rows+1:end, 1:cols)
P.bottomRightCorner<rows,cols>()   // P(end-rows+1:end, end-cols+1:end)
```

```
// Of particular note is Eigen's swap function which is highly optimized.
// Eigen                           // Matlab
R.row(i) = P.col(j);               // R(i, :) = P(:, j)
R.col(j1).swap(mat1.col(j2));      // R(:, [j1 j2]) = R(:, [j2, j1])

// Views, transpose, etc;
// Eigen                           // Matlab
R.adjoint()                        // R'
R.transpose()                      // R.' or conj(R')        // Read-write
R.diagonal()                       // diag(R)                // Read-write
x.asDiagonal()                     // diag(x)
R.transpose().colwise().reverse()  // rot90(R)               // Read-write
R.rowwise().reverse()              // fliplr(R)
R.colwise().reverse()              // flipud(R)
R.replicate(i,j)                   // repmat(P,i,j)


// All the same as Matlab, but matlab doesn't have *= style operators.
// Matrix-vector.  Matrix-matrix.   Matrix-scalar.
y  = M*x;          R  = P*Q;        R  = P*s;
a  = b*M;          R  = P - Q;      R  = s*P;
a *= M;            R  = P + Q;      R  = P/s;
                   R *= Q;          R  = s*P;
                   R += Q;          R *= s;
                   R -= Q;          R /= s;

// Vectorized operations on each element independently
// Eigen                       // Matlab
R = P.cwiseProduct(Q);         // R = P .* Q
R = P.array() * s.array();     // R = P .* s
R = P.cwiseQuotient(Q);        // R = P ./ Q
R = P.array() / Q.array();     // R = P ./ Q
R = P.array() + s.array();     // R = P + s
R = P.array() - s.array();     // R = P - s
R.array() += s;                // R = R + s
R.array() -= s;                // R = R - s
R.array() < Q.array();         // R < Q
R.array() <= Q.array();        // R <= Q
R.cwiseInverse();              // 1 ./ P
R.array().inverse();           // 1 ./ P
R.array().sin()                // sin(P)
R.array().cos()                // cos(P)
```

The Fundamental type of Eigen is the Matrix. There are two types of matrices:

- **fixed size:** size known at compile time
- **dynamic:** size known only at run time

Tensor product A column vector multiplied with a row vector

Dot Product Row vector multiplied with a column vector

1.3 1.2.3 Matrix Storage Formats

The entries of a (generic, dense i.e every entry matters) $A \in \mathbb{K}^{m,n}$ are stored in a contiguous linear array of size $m \cdot n$. An exception is structured/sparse matrices i.e matrices which have a certain structure or few non zero entries.

By Default Eigen stores matrices in Column major format as all the elements in this format are contiguous in memory. In Row major format the elements of the Matrix are scattered which results in cache misses. Hence Column major format is more efficient

Accessing Arrays in Eigen:

- A(i) $\rightarrow$ reference to the i-th element of the array
- A.data() $\rightarrow$ raw pointer

We declare a nxn matrix in Eigen like:

Eigen:: MatrixXd A = Eigen:: MatrixXd::Random(n,n)

Row/Col Access (j-th row):

A.row(j)
A.col(j)

1.3.1 Raw Pointers

A pointer is a type of variable. It stores the address of an object in memory and is used to access that object in memory and is used to access that object.
A **Raw Pointer** is a pointer whose lifetime is not controlled by an encapsulating object . A raw pointer can be assigned the address of another non-pointer variable, or it can be assigned a value of nullptr.

```
int* p = nullptr; // declare pointer and initialize it
                  // so that it doesn't store a random address
int i = 5;
p = &i; // assign pointer to address of object
int j = *p; // dereference p to retrieve the value at its address
```

1.4 1.4 Computational Effort

Traditional: number of elementary operations

Modern: The computational effort involved in a run of a numerical code is only loosely related to the overall execution time on modern computers. It is mainly determined by the memory access pattern and vectorization/pipelining.

1.4.1 1.4.1 Asymptotic complexity

Characterises the worst-case dependence of its computational effort on one or more problem size parameters when these tend to ∞

Notation: Landau-O notation

We define the computational effort as Cost(n) $= (O)(n^{\alpha}), \alpha > 0 \iff \exists C > 0, n_0 \in \mathbb{N} : cost(n) \leq Cn^{\alpha} \forall n > n_0$

Tacit Sharpness assumption: $cost(n) \neq \mathbb{O}(n^{\beta}), \forall \beta < \alpha$

Asymptotic complexity predicts the dependence of runtime on problem size for large problems, because for small problem sizes we can use the caches and hence we do not have the memory access bottleneck.

1.4.2 1.4.2 Computational Cost of basic numerical LA operations

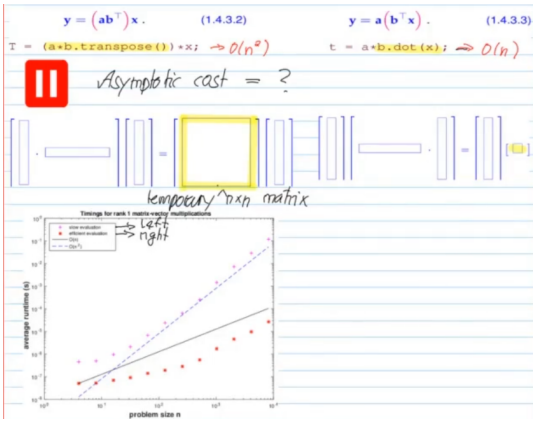
operation	description	#mul/div	#add/sub	asymp. complexity
dot product	$(\mathbf{x} \in \mathbb{R}^n, \mathbf{y} \in \mathbb{R}^n) \mapsto \mathbf{x}^H \mathbf{y}$	n	$n-1$	$O(n)$
tensor product	$(\mathbf{x} \in \mathbb{R}^m, \mathbf{y} \in \mathbb{R}^n) \mapsto \mathbf{x}\mathbf{y}^H$	nm	0	$O(mn)$
Matrix×vector	$(\mathbf{x} \in \mathbb{R}^n, \mathbf{A} \in \mathbb{R}^{m,n}) \mapsto \mathbf{A}\mathbf{x}$	mn	$(n-1)m$	$O(mn)$
matrix product[(*)]	$(\mathbf{A} \in \mathbb{R}^{m,n}, \mathbf{B} \in \mathbb{R}^{n,k}) \mapsto \mathbf{A}\mathbf{B}$	mnk	$mk(n-1)$	$O(mnk)$

The matrix multiplication is implemented with a triple loop and hence not optimal.

A doubly logarithmic plot is scaled logarithmically on the x and y axis. If Cost(n) = $\mathbb{N}(n^\alpha)$ then the data points aligned in a doubly logarithmic plot correspond to a straight line.

1.4.3 1.4.3 Tricks to improve complexity

- **Exploit associativity**



Chapter 2

C++

2.1 Templates and Template Classes in C++

Templates: Let you define the behavior of the class without actually knowing what datatype will be handled by the operations of the class i,e a templated class does not depend on the datatype it deals with. The basic syntax for declaring a templated class:

```
template <class a_type> class a_class {...};
```

a_type is not a keyword, its an identifier that during the execution of the program will represent a single datatype

When defining a function as a member of a templated class, it is necessary to define it as a templated function

```
template<class a_type> void a_class<a_type>::a_function(){...}
```

specialization: An instantiated object of a templated class
Example:

```
class calc
{
  public:
    int multiply(int x, int y);
    int add(int x, int y);
 };
int calc::multiply(int x, int y)
{
  return x*y;
}
int calc::add(int x, int y)
{
  return x+y;
}
```

```
template <class A_Type> class calc
{
  public:
    A_Type multiply(A_Type x, A_Type y);
    A_Type add(A_Type x, A_Type y);
};
template <class A_Type> A_Type calc<A_Type>::multiply(A_Type x,A_Type y)
{
  return x*y;
}
template <class A_Type> A_Type calc<A_Type>::add(A_Type x, A_Type y)
{
  return x+y;
}
```

Keyword: typename In a template declaration "typename" can be used as an alternative to class to declare type template parameters

2.2 Scope Resolution Operator "::"

The scope Resolution Operator is used for one of the following:

- Access the global variable, if there is a local variable with the same name
- Define a function outside the class
- Access a class's static variable
- Multiple Inheritance
- Refer to a class inside another class

2.3 Namespaces

Namespaces provide a method for preventing name conflicts in large projects. Symbols declared inside a namespace block are placed in a named scope that prevents them from being mistaken for identically-named symbols in other scopes

using-declaration:

using $ns_$name::name

Makes the symbol name from the namespace *ns_name* accessible for unqualified lookup as if declared in the same class scope, block scope or namespace as where this using-declaration appears

unqualified lookup

2.4 Name Lookup

The procedure by which a name, when encountered in a program, is associated with the declaration that introduced

Unqualified name lookup An unqualified name, is a name that does not appear to the right of a scope resultion operator. Name lookup examines the scopes as described below, until it finds at least one declaration of any kind at which time the lookup stops and no further scopes are examined.