

Note Set 2 – Programming

2.1 – Overview

In this note set, we will cover programming in `c/c++` for numerical applications. We will start by covering the basic of the language, with particular emphasis on the difference in programming strategies between a mid-level language like `c/c++` and matrix programming languages like `r` and `matlab`. We next describe the principles of object-oriented programming and how these techniques can make `c++` “feel like” `matlab` for numerical applications. Finally, we will discuss strategies for writing efficient code for both `c/c++` and matrix programming languages.

2.2 – Program Structure

2.2.1 - General

For numerical applications, we will focus on console applications, or applications that run from the command line (as opposed to windows applications, for example). If you are using an IDE (Integrated Development Environment), make sure that when you

create a new project, you create your project as a console application. In a console application, the main routine control program flow. The simplest possible c/c++ program is the hello world program:

```
#include <iostream>
using namespace std;

int main()
{
    cout << "hello world\n";
    return 0;
}
```

This program will simply print “hello world” to the command line. The line “#include <iostream>” tells the compiler to include the input/output stream library, which allows calling the `cout` (console output) command. In c/c++, the set of built in functions/operations/keywords is quite small. Thus, even for some very fundamental things, we need to tell the compiler to look in the write place to find code to perform these functions. In c/c++, even some very basic functions have to be manually included. For example, if we want to use the exponential function, we must include `<cmath>`.

In c++, every user defined keyword falls into a namespace. The default namespace is the `std` namespace. If we had instead written, “using namespace `std2`;”, the above code would not compile because the compiler won’t understand `cout`, since `cout` has not been defined in the `std` namespace. This feature is included to avoid conflicts between libraries. If two libraries defined the function `eval_all(double x)`, these libraries would conflict. If the first library used the name `Namespace1` and the second used `Namespace2`, then these would not conflict, but we would have to call these functions as `Namespace1::eval_all(x)` and `Namespace2::eval_all(x)`. If we had defined `<iostream>` to fall in the namespace `std2`, we would have to write,

```
std2::cout << hello world\n";
```

As a general rule, you should define everything in the `std` namespace to make your life easier, since you are not writing code that other people will have to compile (and thus do not have to worry about keyword conflicts).

Program flow is specified in the main routine. The main routine may take command line arguments by modifying the first line,

```
int main(int argc, char *argv[])  
{  
    ...  
}
```

This is another one of these features that you won't be using much. The main function always returns an integer value. This integer values doesn't really do anything, so the only function of "return 0" is to stop program flow. For example, consider the program,

```
#include <iostream>  
using namespace std;  
  
int main()  
{  
    cout << "hello world\n";  
    return 0;  
    cout << "goodbye world\n";  
}
```

This program will print hello world, but not goodbye world. Some compilers will allow you to specify main and a function that returns a void type (e.g. nothing). Keep in mind, however, that if you don't include a return in the program, the program may fail to compile, it may generate a warning, or it may simply crash!

2.2.2 – Data Types

In c/c++, any variable you would like to use must be “declared”, and you must specify the type of variable. This performs the function of reserving memory for the variable. The common list of types includes:

int: (integer, four bytes, -2,147,483,647 – 2,147,483,647)
float: (floating point, four bytes, ~six digits of precision)
double: (floating point, eight bytes, ~ten digits of precision)
bool: (Boolean, one byte)
char: (character, one byte)
string: (string)

All but string are part of c, while string is part of the c++ STL (standard template library).

Note that there is no guarantee that these keywords will stay the same across platforms. For example, on my computer int is four bytes, but on some machines it may be two bytes. The keyword long double will create an 80-bit floating point number of some machines, but will create a 64-bit double on my machine. In practice, we don't have to worry about this if we are working with Intel/AMD/Mac machines.

Lesson 1 - You have to be somewhat careful:

You have to be somewhat careful. For example, on my compiler, the code,

```
int i = 25423459740;  
cout << "i: " << i << "\n";
```

return the error message, “main.cpp:28: error: integer constant is too large for “long” type”, but the code,

```
int i = 2542345974;
cout << "i: " << i << "\n";
```

produces the output, “i: -1752621322”. Similarly, adding two very large integers may cause the sum to overflow without producing an error.

2.2.3 – Program Flow

In c/c++, the curly brackets, ‘{’ and ‘}’ are used to define scope. For example, in the code,

```
int k = 3;
for(int i=0; i<n; i++)
{
    int j = 2;
    [ some code here ]
}
```

j is defined in the for loop, but not outside of it. Alternatively, k is defined both inside and outside of the loop.

In matrix programming languages, all variables are global- a variable declared anywhere can be accessed anywhere. In c/c++, variables are only available where they are called and below. This is determined by { and }. For example, consider the program,

```
int main()
{
    {
        double x = 3.0 + 5.0;
    }
    cout << "x: " << x;
}
```

This code will not compile because x is only declared within the inner parentheses.

Alternatively, the following code will compile.

```
int main()
{
    double x = 3.0 + 5.0;
    {
        cout << "x: " << x;
    }
}
```

You can declare two (different) variables to have the same name, but don't do this unless you have a good reason.

Variables declared outside of the main function are global variables, and are accessible anywhere. As a general rule, it is a good practice to declare variables in the innermost nest possible, as this will allow the compiler to alert you if it is accessed elsewhere (a likely error). It is also a good practice to avoid global variables, though sometimes global variables are unavoidable.

For example, consider the following code,

```
int main()
{
    double Sum = 0.0;
    for(int i=0; i<5; i++)
    {
        double Temp = sqrt((double)i);
        Sum += Temp + Temp * Temp;
    }
}
```

as opposed to,

```
int main()
{
    double Sum = 0.0;
    int i;
    double Temp;
    for(i=0; i<5; i++)
    {
        Temp = sqrt((double)i);
    }
}
```

```
        Sum += Temp + Temp * Temp
    }
}
```

These codes will generally compile to the same machine code, but the compiler is more likely to catch bugs in your program if you use the first strategy. An exception to this rule occurs when declaring objects that much allocate memory (such as vectors). In this case, we may want to avoid the cost of allocating memory repeatedly.

2.2.4 - Input and Output

One of the most fundamental things we do is output to the screen. This is accomplished using the `cout` operator. The library `iostream` must be included. You can write to a file using similar syntax. For example,

```
#include <fstream>
using namespace std;

int main()
{
    ofstream fout1("c:\\a.dat");
    fout1 << "hello world\n";
    fout1.close();
    return;
}
```

This program will write “hello world” to the text file “a.dat” located in the root directory, overwriting any file that it already there. Input is a more complicated topic.

2.2.5 – Functions

Functions are an important components of c/c++. A function takes on multiple arguments and returns a single argument (or no arguments, in case the return type is void). Function parameters may be “passed by value” or “passed by reference”. For example,

```
double func(double x)
{
    x = x* x;
    return x;
}

int main()
{
    double x = 5.0;
    double y = func(x);
    cout << "x: " << x << "\n";
    cout << "y: " << y << "\n";
    return 0;
}
```

will return,

```
x: 5
y: 25
```

Alternatively, if the function were defined as,

```
double func(double &x)
{
    x = x* x;
    return x;
}
```

we would have,

```
x: 25
y: 25
```

A third alternative,

```
double func(const double &x)
```

```
{
    x = x* x;
    return x;
}
```

would not compile.

As a general strategy, declaring all object arguments by value, and as const, unless they need to be modified is a good idea. It ensures that we don't waste time copying. For more basic types, declaring them by value allows efficient use of the stack however.

Now consider the following function definition,

```
int double_it(double &x)
{
    x *= 2;
}

int main()
{
    int i = double_it(2.3);
    return 0;
}
```

This will give an error because the computer is requesting a return value from `double_it(2.3)`, but doesn't get any.

2.3 – Object Oriented Programming

2.3.1 - Classes

Suppose that c++ did not have a built in representation of complex numbers and we wanted to create one. We could do this using a class. We would create the object,

```
Class ComplexDouble
{
```

```

public:

double Real; // Real part
double Imag; // Imaginary part

/* Constructor */
ComplexDouble()
{
    Real = 0.0;
    Imag = 0.0;
}

/* Destructor */
~ComplexDouble()
{
}
};

```

To make this more useful, we might overload the << operator:

```

ostream & operator << (ostream &os, Complex x)
{
    os << x.Real;
    if(x.Imag > 0.0) os << " + " << x.Imag << "*i";
    else if(x.Imag < 0.0) os << " - " << -x.Imag << "*i";
    return os;
}

```

We could also overload the addition operator,

```

ComplexDouble operator + (const ComplexDouble &x, const ComplexDouble
&y)
{
    ComplexDouble z;
    z.Real = x.Real + y.Real;
    z.Imag = x.Imag + y.Imag;
    return z;
}

```

2.3.2 – Templates

Perhaps we want a complex integer type in addition to the complex double type. It would be convenient if we did not have to define it separately. We can accomplish this using template,

```

template <class T>
Class Complex
{
    public:

    T Real;
    T Imag;

    /* Constructor */
    Complex()
    {
        Real = 0.0;
        Imag = 0.0;
    }

    /* Destructor */
    ~Complex()
    {
    }
}

```

Here, T is a variable type. We could create instances of each calls using,

```

Complex<double> x;
x.Real = 1.7;
x.Imag = 1.2;

Complex<int> y;
x.Real = 7;
x.Imag = 3;

```

Using c++ efficiently (as in man hours, not computation time) involves collecting libraries that perform these tasks as automatically as possible. For example, suppose we wanted to computing the polynomial root of $(x^4 - 7x + 3)(x^3 + 3x - 1)$. It would be nice if we could write,

```

Polynomial p = "x^4 - 7*x + 3";
Polynomial q = "x^3 + 3*x - 1";
Polynomial r = p * q;
Vector< Complex<double> > Roots = r.GetRoots();

```

It is in this sense that for an experienced user, C++ can be easier to use than any matrix programming language. Because it is so easily extendable, anything you do often, you can automate using class, templates, overloading operators, etc.

In this case, we would have to create a polynomial class. Overload the = operator to read string, parse the expression, and create a polynomial object, overload the * operators, and link some root finding code.

As another example, the C++ library Gambit will solve for the equilibrium of a game in rational arithmetic (rather than floating point arithmetic), by calling linear algebra templates.

2.4 – Pointers and Arrays

2.4.1 – Pointers

In C/C++, it is possible to declare a variable that points to a particular location in memory. This is a very useful feature, but also one of the most dangerous features (and a frequent source of bugs). Direct access to memory is one of the primary ways C/C++ improve speed over alternative programming languages.

The code,

```
double *p;
```

Create a pointer to a double. The code,

```
double *p = 8.7;
```

creates a pointer to a double, stores 8.7 in memory, and set the pointer to the location in memory where the 8.7 is stored.

Now, the code,

```
p++;
```

would increment the pointer so that it would point to a location in memory 8 bytes in front of where the 8.7 is stored. The code,

```
*p++;
```

would instead add one to the value 8.7 stored in memory. Literally, the * operators (distinct from the * multiplication operator) is the value-in operator. If x is a pointer to an address in memory, then *x is the value in memory in the location where x points. The operator & is the address-of memory. For example, x is a double, &x might contain something like "0x2fe17".

Now here is where the danger comes in- there is nothing to prevent us from accessing a location in memory where garbage is stored, some of our other data is stored, or worse yet, part of the operating system is stored.

The operator & return the address of. For example, if we write,

```
double x = 3.0;
double *y = &x;
cout << "x: " << x << "\n";
cout << "&x: " << &x << "\n";
cout << "y: " << y << "\n";
cout << "*y: " << *y << "\n";
```

This code will return,

```
x: 3
&x: 0x22ff48
y: 0x22ff48
*y: 3
```

Alternatively, we could write,

```
double x = 3.0;
double *y = &x;
y += 10000;
*y = -189.6;
```

This code will save a location in memory and store the number 3 there. Next, it will create a pointer to a double, which points to the location where the 3 is stored. Next, the pointer will move over $10000 * 8$ bytes in memory. Next, it will assign to this location the number -189.6. The problem is that the location that y is pointing to could contain some of your other data, part of your operating system, etc. It may cause your whole computer to crash, one of your other programs to crash, or most likely, just the program you are running to crash.

2.4.2 – Static Arrays

In c, there is no built in dynamic array functionality. We can declare an array as,

```
double a[10];
```

The notation `a[i]` is used to access elements of an array. Arrays in c/c++ are zero-indexed. This means that `a[0]` contains the first element of the array and `a[5]` contains the sixth element of the array. Literally, the statement above reserves 80 sequential bytes in memory and creates a pointer to a double which points to the start of this. Typing,

```
cout << a << "\n"
```

would print out the address in memory where the array starts, as would,

```
cout << &a[0] << "\n";
```

Hence, the `[i]` operator functions as, the value in the location `i` units forward from `a`.

There is not check on the bounds of an array. For example, `a[-7]` and `a[345987345]` are valid entries, and `a[-7]` would return the value in memory 7*8 bytes to the left of the beginning of the array.

Two-dimensional arrays can be created using,

```
double a[10][12];
```

Elements can be accessed using the `a[i][j]` notation. Indexing always starts as zero.

This will literally create a 120 elements array of doubles and a 10 elements array of pointers to doubles. Each elements of the pointer array would point to `&a[0]`, `&a[12]`, `&a[24]`, etc., in memory. The notation `a[i]` will return a pointer to the start of the *i*th row in the larger array (and not the value!). The notation `*a[2]` would return the value of the 24th elements in the array.

2.4.3 – Dynamic Arrays

The code,

```
double a[10];
```

will create an array with 10 doubles, but the size of an array must be specified at compile time. For example, the following will not work:

```
int sz = 10;  
double a[sz];
```

In c, to declare a dynamic array, we must use the `malloc` command to allocate memory and the `delloc` command to unallocated memory. C++ introduced the commands `new` and `delete`. For example, consider the code,

```
int sz = 10;
```

```
double *x = new(sz);
for(int i=0; i<sz; i++) x[i] = 5 + i;
delete x;
```

This is starting to get complicated. First, the type `double *` is a pointer to a double (literally, a variable containing the address in memory where we can find a double). The 10 elements in memory starting from the location `*x` will contain the elements of an array. The notation `x[i]` tells us to start at `*x`, and move over `i` units.

If you use `new`, you have to remember to free the memory using `delete`, otherwise, the memory will not free up until your program finishes executing. You have to make sure you don't call `delete` too soon. The vast majority of "malware" is due to unchecked pointers (so called buffer overruns).

2.4.4 –Arrays as Objects

In the days of `c`, there is really nothing we could do about this problem. In `c++` there is an alternative- objected oriented programming. Consider the following vector object from numerical recipes in `c++`,

```
template <class T>
class NRvector {
private:
    int nn;        // size of array. upper index is nn-1
    T *v;
public:
    NRvector();
    explicit NRvector(int n);           // Zero-based array
    NRvector(int n, const T &a); //initialize to constant value
    NRvector(int n, const T *a); // Initialize to array
    NRvector(const NRvector &rhs);      // Copy constructor
    NRvector & operator=(const NRvector &rhs); //assignment
    typedef T value_type; // make T available externally
    inline T & operator[](const int i); //i'th element
    inline const T & operator[](const int i) const;
    inline int size() const;
    void resize(int newn); // resize (contents not preserved)
```

```

        void assign(int newn, const T &a); // resize and assign a
constant value
        ~NRvector();
};

// NRvector definitions

template <class T>
NRvector<T>::NRvector() : nn(0), v(NULL) {}

template <class T>
NRvector<T>::NRvector(int n) : nn(n), v(n>0 ? new T[n] : NULL) {}

template <class T>
NRvector<T>::NRvector(int n, const T& a) : nn(n), v(n>0 ? new T[n] :
NULL)
{
    for(int i=0; i<n; i++) v[i] = a;
}

template <class T>
NRvector<T>::NRvector(int n, const T *a) : nn(n), v(n>0 ? new T[n] :
NULL)
{
    for(int i=0; i<n; i++) v[i] = *a++;
}

template <class T>
NRvector<T>::NRvector(const NRvector<T> &rhs) : nn(rhs.nn), v(nn>0 ?
new T[nn] : NULL)
{
    for(int i=0; i<nn; i++) v[i] = rhs[i];
}

template <class T>
NRvector<T> & NRvector<T>::operator=(const NRvector<T> &rhs)
// postcondition: normal assignment via copying has been performed;
//           if vector and rhs were different sizes, vector
//           has been resized to match the size of rhs
{
    if (this != &rhs)
    {
        if (nn != rhs.nn) {
            if (v != NULL) delete [] (v);
            nn=rhs.nn;
            v= nn>0 ? new T[nn] : NULL;
        }
        for (int i=0; i<nn; i++)
            v[i]=rhs[i];
    }
    return *this;
}

template <class T>
inline T & NRvector<T>::operator[](const int i) //subscripting
{

```

```

#ifdef _CHECKBOUNDS_
if (i<0 || i>=nn) {
    throw("NRvector subscript out of bounds");
}
#endif
return v[i];
}

template <class T>
inline const T & NRvector<T>::operator[](const int i) const
    //subscripting
{
#ifdef _CHECKBOUNDS_
if (i<0 || i>=nn) {
    throw("NRvector subscript out of bounds");
}
#endif
return v[i];
}

template <class T>
inline int NRvector<T>::size() const
{
    return nn;
}

template <class T>
void NRvector<T>::resize(int newn)
{
    if (newn != nn) {
        if (v != NULL) delete[] (v);
        nn = newn;
        v = nn > 0 ? new T[nn] : NULL;
    }
}

template <class T>
void NRvector<T>::assign(int newn, const T& a)
{
    if (newn != nn) {
        if (v != NULL) delete[] (v);
        nn = newn;
        v = nn > 0 ? new T[nn] : NULL;
    }
    for (int i=0;i<nn;i++) v[i] = a;
}

template <class T>
NRvector<T>::~~NRvector()
{
    if (v != NULL) delete[] (v);
}

// end of NRvector definitions

```

Here, we create a class called `NRvector`. We could create an instance of `NRvector` using the command,

```
NRvector v(10);
```

This will execute the code,

```
NRvector<T>::NRvector(int n) : nn(n), v(n>0 ? new T[n] : NULL) {}
```

Here, this tells the computer to allocate a dynamic array of size `n` and assign it to `v` if `n>0`, and assign a `NULL` pointer to `v` otherwise (a null pointer tells the computer that the pointer points to nowhere in memory).

Now, after the last time the object is used, the computer will call the destructor, which is always the class name with a `~` before it. In this case, we have,

```
if (v != NULL) delete[] (v);
```

This tells the computer to free up the memory. Notice that this has taken care of the allocating and de-allocating memory problem for use.

Next, we make the `NRvector` object look and feel like a matlab-like array. We do this by overloading the `[]` operator. If `v` is an `NRvector`, then `v[i]` will tell the computing to start at `v` and move `i` spaces away in memory, and return the data in that location. To prevent us from accessing data where we shouldn't, the code

```
if (i<0 || i>=nn) {  
    throw("NRvector subscript out of bounds");  
}
```

keeps us in line. If the subscript is out of bounds, an “exception” is thrown. This will generally halt execution of the program, unless you have specifically written code to “catch” the exception. For our purposes, allowing the program to halt is usually satisfactory, as we will want to know that our program is doing something crazy.

Finally, we may want to overload various other operators to make the NRvector object behave like a matlab vector. For example, we might want to overload the << operator. We could write,

```
// Overload the cout operator for vectors
template <class T>
ostream & operator << (ostream &os, const Vector<T> &v)
{
    int n = v.Size();
    if(n > 0)
    {
        /* Print Vector as tab-delimited */
        os << "[";
        for(int i=0; i<n-1; i++) os << v[i] << "\t";
        os << v[n-1] << "];";
    }
    else os << "[]";
    return os;
}
```

If fact, I would suggest that you add this code into nr3.h, rather than printing a vector by writing,

```
for(int i=0; i<n-1; i++) cout << "[" << x[i] << ",";
cout << x[i] << "\n";
```

2.5 – Getting Started with C/C++

2.5.1 – Dev C++

If you are planning to program in c/c++, I recommend downloading the free IDE Dev-c++. This IDE using the open source gnu c++ compiler. The IDE is fairly easy to use- the only drawback is the compiler it uses produce fairly cryptic error messages. The IDE can be downloaded from:

<http://www.bloodshed.net/dev/devcpp.html>

Once you install this program, you should create a new project, which you should make sure is a console application. If libraries are not in the project directory, then you must include the path using the commands,

Project → Project Options → Directories → Include Directories

Select the folder where the libraries are included (I believe you must select subfolders as well) and don't forget to click 'Add'! This should hopefully be enough to get you started.

2.5.2 – Numerical Recipes in C++

Numerical recipes in c provides a good starting point. It is the most comprehensive source of c++ code that I know of, and it is fairly easy to get this code to compile on your machine. I would also highly recommend newmat (for matrices) and prob (for probability distributions). Links to this code can be found in the syllabus.

2.6 – Writing Efficient Code

In this section, we will consider a number of important tips for writing fast code. Being on the cutting edge of methodology requires solving very hard problems in reasonable time. Writing efficient code will allow you to solve harder problems than the 'competition'. We consider writing code in both low-level languages like c/c++ and high

level languages like matlab. In general what we see for c/c++ will hold for FORTRAN as well (though I can't imagine why anybody would be using FORTRAN these days) and what we say about matlab will hold true for gauss, r, stata, or most other matrix programming languages. The main difference is that efficient code in matlab will probably run faster than these other languages.

2.6.1 – Big-O Notation

Large O notation is one of the most useful tools in studying the efficiency of an algorithm. For example, suppose that n is the size of a problem. We say that an algorithm is $O(n^\alpha)$ if the execution time of the algorithm $e(n)$ satisfies $\frac{e(n)}{n^\alpha} \rightarrow c \neq 0$ as $n \rightarrow \infty$. For example, the operation $y \leftarrow A * x$ requires n^2 multiplications and additions. Hence, we have that $y \leftarrow A * x$ is $O(n^2)$. Similarly, the operation $y \leftarrow A * x + b$ requires n^2 multiplication and $n^2 + n$ additions. Hence, we still have that $y \leftarrow A * x + b$ is $O(n^2)$. The operation $y \leftarrow A * x + B * z$ may require $2n^2$ multiplications and additions, but we still say that it is $O(n^2)$.

Why do we only focus on the highest order terms? Why do we not care about the constant in front of the n^2 .

Example:

Suppose that we have a vector x , which is known to be sorted. Suppose we want to find whether the vector contains the element a . Consider the following two algorithms.

Algorithm 1 (Brute Force Search):

```
int Search(const Vector<double> &x, const double &a)
{
    int n = x.Size();
    for(int i=0; i<n; i++) if(x(i) == a) return i;
    return -1;
}
```

Algorithm 2 (Binary Search):

I won't write out the code, but describe it this way. First, look at the mid-element of the vector. If the element is greater than a , look in the lower part. If the element is less than a , look in the upper part. Continue this procedure until the element is found.

How would we analyze each of these using big-O notation? Which algorithm is better? We can see that Algorithm 1 is $O(n)$ and Algorithm 2 is $O(\log n)$. Hence, the second algorithm is better. In practice, this approach to analyzing algorithm speed is very effective. Ignoring the constant and lower order terms is unusually not necessary to get very good performance.

Consider the following code for multiplying two matrices,

```
for(int i=0; i<m; i++) for(int j=0; j<n; j++)
{
    C(i,j) = 0.0;
    for(int k=0; k<r; k++)
    {
        C(i,j) += A(i,k) + C(k,j);
    }
}
```

In this case, the code is $O(n^3)$ (assuming that m , n , and r are all roughly are same size.

2.6.2 – Writing Efficient Code in a Low Level Language

Here, we outline a number of principles for writing efficient code.

1. Find the bottleneck by finding the largest number of nested loops
2. Easy code can be used for the parts of the program that don't contribute much to overall execution time.
3. Array elements should be accessed sequentially, when possible
4. Pointers are fast
5. Exploit the memory hierarchy

Most of the computers you will use will have 4 levels of memory- registers, cache, ram, and virtual memory (e.g. the hard disk). These levels of memory differ in speed as well as capacity. A typical processor will have 32 1 byte registers. These are directly connected to the cpu. A processor will also have a certain amount of cache- 512 kilobytes is common these days. Your computer probably has between 0.5 gigabytes and 2 gigabytes of ram. If you need to use more ram, you can use the hard drive as virtual memory. The cpu can only access the registers directly. Hence, for the cpu to access something in another level of memory, it must travel to the register. Traveling from cache to the registers takes the least amount of time. Traveling from ram to the registers takes more time. Traveling from the hard disk to the register takes the largest amount of time.

One principle that is used to speed up computer programs is to minimize the amount of times data is moved from the higher levels of memory to the registers. This is sometimes called 'blocking'. Register-block minimize the number of times data must be compiled from cache or main memory to the registers. Cache-blocking is effective use of the cache. Memory blocking minimizes copies from virtual memory into main memory.

Two principles are used to implement blocking- loop-unrolling and matrix blocking.

1. Loop unrolling

The idea In the following example, we consider the operation $y = Ax$. This may seem like an operation so simple that there is no need to optimize. In fact, we will find that there is a huge difference between optimized

2. Blocking

In the following example, we consider the operation $y = Ax$. This may seem like an operation so simple that there is no need to optimize. In fact, we will find that there is a huge difference between optimized and un-optimized code. We will consider implement this in c++. On the course web page, I have provided a number of routines that we will use. Consider first this code.

```

void Prod1(Vector<double> &y, const Matrix<double> &A, const
Vector<double> &x);
void Prod1(Vector<double> &y, const Matrix<double> &A, const
Vector<double> &x)
{
    int m = A.Rows();
    int n = A.Cols();

    for(int i=0; i<m; i++) y(i) = 0.0;
    for(int j=0; j<n; j++)
        for(int i=0; i<m; i++) y(i) += A(i,j) * x(j);
}

```

This code is simple, but fairly naïve. One problem with this code is that it does not access memory sequentially. This problem is fixed below.

```

void Prod2(Vector<double> &y, const Matrix<double> &A, const
Vector<double> &x)
{
    int m = A.Rows();
    int n = A.Cols();

    for(int i=0; i<m; i++) y(i) = 0.0;
    for(int i=0; i<m; i++)
        for(int j=0; j<n; j++) y(i) += A(i,j) * x(j);
}

```

This code is not yet fully efficient. First, it still has overhead related to using the $A(i,j)$ call. Below, we eliminate this problem.

```

void Prod3(Vector<double> &y, const Matrix<double> &A, const
Vector<double> &x)
{
    int m = A.Rows();
    int n = A.Cols();

    for(int i=0; i<m; i++) y(i) = 0.0;
    int Curr = 0;
    for(int i=0; i<m; i++)
        for(int j=0; j<n; j++) y[i] += A[Curr++] * x[j];
}

```

Pointers make this code work even faster.

```

void Prod4(Vector<double> &y, const Matrix<double> &A, const
Vector<double> &x)
{
    int m = A.Rows();
    int n = A.Cols();

    double *y1 = y.Begin();
    const double *A1 = A.Begin();

    for(int i=0; i<m; i++)
    {
        *y1 = 0.0;
        double *x1 = x.Begin();
        for(int j=0; j<n; j++) *y1 += *A1++ * *x1++;
    }
}

```

As a final step, we unroll the loops.

```

void Prod5(Vector<double> &y, const Matrix<double> &A, const
Vector<double> &x)
{
    int m = A.Rows();
    int n = A.Cols();

    double *y1 = y.Begin();
    const double *A1 = A.Begin();

    for(int i=0; i<m; i++)
    {
        *y1 = 0.0;
        double *x1 = x.Begin();
        for(int j=0; j<n/5; j++)
        {
            *y1 +=    A1[0] * x1[0]
                    + A1[1] * x1[1]
                    + A1[2] * x1[2]
                    + A1[3] * x1[3]
                    + A1[4] * x1[4];

            A1+=5;
            x1+=5;
        }
        *y1++;
    }
}

```

I ran all of this code on my computer. The results were,

```

Prod1:      31.284 sec.
Prod2:      8.923 sec.
Prod3:      4.066 sec.
Prod4:      1.201 sec.
Prod5:      0.902 sec.

```

Amazingly, the naive code is about 50 times slower than the efficient code. Getting the loops in the wrong order makes the code slower by a factor of 4. Bounds checking and computing indexes accounts for a factor of 4. Overhead accounts for another factor of 4. Finally, loop-unrolling cuts off about 20% of the execution time.

If you have a very good compilers, you may not see any difference between 4 and 5 (some compilers will automatically attempt to unroll loops).

I reran the same code with $m = n = 5000$, for comparison with the matlab code below. Note that this code was run on a different cpu.

<i>Prod1:</i>	<i>1.688 sec.</i>
<i>Prod2:</i>	<i>1.109 sec.</i>
<i>Prod3:</i>	<i>0.656 sec.</i>
<i>Prod4:</i>	<i>0.141 sec.</i>
<i>Prod5:</i>	<i>0.094 sec.</i>

2.6.3 – Writing Efficient Code in a Matrix Programming Language

Writing efficient code in a high level language procedures by completely different principles. Languages such as matlab, r, gauss, stata, sas, etc. are scripted rather than compiled. Every operation entails a lot of overhead. Hence, these languages can be very slow, especially. Applying many of the principles we recommend for low level languages will actually lead a language like matlab to run even slower.

One recommendation that does carry over is to find the deepest loop. The main principle of writing efficient code in these languages is to ‘vectorize’. Eliminate the deepest loop and replace it with a vectorized operation.

For example, consider the same problem in the matlab environment. Suppose that we declare all the variables before-hand:

```
% Initialize
m = 5000;
n = 5000;
x = 3 * ones(m,1);
A = 0.001 * ones(m,n);
y = zeros(n,1);
```

Consider the following loop written in matlab:

```
% loop
tic;
for i=1:m
    y(i) = 0;
    for j=1:n
        y(i) = y(i) + A(i,j) * x(j);
    end
end
toc;
```

This loop took 44.021151 seconds to executed, which is much longer than the most naive c++ code. This is because matlab has a lot of overhead. The efficient way to multiply a vector by a matrix in matlab is the naïve way:

```
% Efficient Way
tic;
y = A * x;
toc;
```

This code took only 0.063978 seconds to run to completion. The reasons this is so much faster is because matlab there is little overhead. Matlab has A and x stored in memory, in the same way that an efficient c++ program would have. It then runs a pre-compiled routine that does something similar to what we suggested doing in c++. In fact, it call a version of BLAS that has been optimized for a typical windows cpu.

We note that we can get rid of most of the overhead by relying on vector operations. For example, consider the code:

```

% Almost Efficient
tic;
for i=1:m
    y(i) = A(i,:) * x;
end
toc;

```

This code took 0.446741 to run to completion. This step gets rid of a lot, but not all of the overhead. In general, efficient programming in matlab will rely on vector operations.

This example indicates that while programming efficiently in matlab can make a large difference (a factor of 100 in this case!), it does not offer the same degree of optimizability as programming in a lower level language. In fact, if matlab had not implemented this particular operation, we would lose a speed factor of 10.

Now, suppose that we didn't pre-allocate A, x, and y. Consider the code,

```

% Bad Way to Initialize
clear x;
clear y;
clear A;
tic;
for i=1:m
    x(i) = 3;
end
for i=1:m
    for j=1:n
        A(i,j) = 0.001;
    end
end
toc;

```

This last step took 820.755351 seconds! The reason this happens is because matlab is constantly reallocating memory.

As a final comparison, I will compare the most efficient matlab code to the most efficient c++ code. In this case, I will not use the timer provided by either matlab or c++, but will use a stopwatch to get a good comparison (I have no way of verifying that these two timers work in the same way). I will run the most efficient code for 100 times.

The c++ code took 47 seconds to run the code 500 times. The matlab code took 30 seconds to run the same thing. What this shows is that when you call a matlab routine, it is very efficient.

Here is another example I got off the web (<http://web.cecs.pdx.edu/~gerry/MATLAB/programming/performance.html>):

A. Using vector operations instead of loops:

Consider the following loop, translated directly from Fortran or C,

```
dx = pi/30;
nx = 1 + 2*pi/dx;
for i = 1:nx
    x(i) = (i-1)*dx;
    y(i) = sin(3*x(i));
end
```

The preceding statements are perfectly legal MATLAB statements, but they are an inefficient way to create the x and y vectors. Recall that MATLAB allocates memory for variables on the fly. On the first time through the loop (i=1), MATLAB creates two row vectors x and y, each of length one. On each subsequent pass through the loop MATLAB appends new elements to x and y. Not only does this incur extra overhead in the memory allocation calls, the elements of x and y will not be stored in contiguous locations in RAM. Thus, any subsequent operations with x and y, even though these operations may be vectorized, will take a performance hit because of memory access overhead.

The preferred way to create the same two x and y vectors is with the following statements.

```
x = 0:pi/30:2*pi
y = sin(3*x);
```

The first statement creates the row vector, x , with $1 + \pi/15$ elements stored in contiguous locations in RAM. The second statement creates a new (matrix) variable, y , with the same number of rows and columns as x . Since x is a row vector, as determined by the preceding step, y is also a row vector. If x were, for example, a 5 by 3 matrix, then $y = \sin(3*x)$ would create a 5 by 3 matrix, y .

MATLAB is designed to perform vector and matrix operations efficiently. To take maximum advantage of the computer hardware at your disposal, therefore, you should use vectorized operations as much as possible.

B. Pre-allocating memory for vectors and matrices:

Though MATLAB will automatically adjust the size of a matrix (or vector) it is usually a good idea to pre-allocate the matrix. Pre-allocation incurs the cost of memory allocation just once, and it guarantees that matrix elements will be stored in contiguous locations in RAM (by columns). Consider the following (admittedly artificial) sequence of statements:

```
dx = pi/30;
nx = 1 + 2*pi/dx;
nx2 = nx/2;

for i = 1:nx2
    x(i) = (i-1)*dx;
    y(i) = sin(3*x(i));
end

for i = nx2+1:nx
    x(i) = (i-1)*dx;
```

```
y(i) = sin(5*x(i));  
end
```

Since we know the size of x and y , a priori, we can pre-allocate memory for these vectors. Pre-allocation involves creating a matrix (or vector) with one vectorized statement before any of the matrix elements are referenced individually. The `ones` and `zeros` functions are typically used to pre-allocate memory.

Here is an improvement of the preceding statements with pre-allocation of memory for x and y .

```
dx = pi/30;  
nx = 1 + 2*pi/dx;  
nx2 = nx/2;  
  
x = zeros(1,nx);      % pre-allocate row-vectors, x  
y = zeros(1,nx);      % and y  
  
for i = 1:nx2  
    x(i) = (i-1)*dx;  
    y(i) = sin(3*x(i));  
end  
  
for i = nx2+1:nx  
    x(i) = (i-1)*dx;  
    y(i) = sin(5*x(i));  
end
```

The statements $x(i) = \dots$, and $y(i) = \dots$ still do not take advantage of possibilities for vectorization, but at least the elements of x and y are stored contiguously in RAM.

We will improve the efficiency of the loops shortly. First, however, note that we could have written the pre-allocation statements as

```
x = zeros(1,nx);      % pre-allocate row-vectors, x  
y = x;                % and y
```

The statement $y = x$ does not mean that y will stay equal to x . It simply creates another matrix y with the same "shape" as x . Understanding that pre-allocation is important for

efficiency will help you understand these apparently confusing twists of MATLAB programming logic.

We can further improve our loops by pulling the assignment of x out of the loops.

```
x = 0:pi/30:2*pi;      % vectorized calculation of x
nx = length(x);
nx2 = nx/2;

y = x;                % pre-allocate memory for y

for i = 1:nx2
    y(i) = sin(3*x(i));
end

for i = nx2+1:nx
    y(i) = sin(5*x(i));
end
```

Finally, if we're obsessed with performance, we observe that the calculation of y can also be vectorized. To do this we use the colon notation to refer to segments of the x and y vectors.

```
x = 0:pi/30:2*pi;      % vectorized calculation of x
nx = length(x);
nx2 = nx/2;

y = x;                % pre-allocate memory for y

y(1:nx2) = sin(3*x(1:nx2)); % compute first part of y
y(nx2+1:nx) = sin(5*x(nx2+1:nx)); % and the second part
```

To those new to MATLAB programming, the preceding statements may appear unnecessarily obfuscated. The comment statements, of course, help, but the logic behind the logic comes from a true understanding of vectorization. Once you get the hang of MATLAB's colon notation you too will come to write code like this. Whenever the speed of MATLAB code is important, there is no substitute for vectorization.

2.7 – References

- [1] Schildt, Herb (1998). “C++: The Complete Reference”.
- [2] Stroustrup, Bjarne (1997). “The C++ Programming Language”.