6 Errors and Numerical Stability of ODEs

6.1 Errors

Typically numerical computation of any sort will generate errors and it is one of the tasks of
the numerical analyst is to be able to assess and be aware of the errors arising from numerical
computation. The different types of errors can be categorised into the following main types:

- Roundoff errors.
- Errors in modelling.
- Programming errors.
- Truncation and discretization errors.

6.1.1 Roundoff errors

These arise when a computer is used for doing numerical calculations. Some typical examples
include the inexact representation of numbers such as \( \pi \), \( \sqrt{2} \). Roundoff and chopping errors arise
from the way numbers are stored on the computer, and in the way arithmetic operations are
carried out. Whereas most of the time the way numbers are stored on a computer is not under
our control, the way certain expressions are computed de

initely is. A simple example will illustrate the point. Consider the computation of the roots
of a quadratic equation \( ax^2 + bx + c = 0 \) with the expressions

\[
x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}
\]

Let us take \( a = c = 1 \), \( b = -28 \). Then

\[
x_1 = 14 + \sqrt{195}, \quad x_2 = 14 - \sqrt{195}
\]

Now to 5 significant figures we have \( \sqrt{195} = 13.964 \). Hence \( x_1 = 27.964 \) \( x_2 = 0.036 \). So what can
we say about the error. We need some way to quantify errors. There are two useful measures for
this.
Absolute error
Suppose that $\phi^*$ is an approximation to a quantity $\phi$. Then the absolute error is defined by $|\phi^* - \phi|$.

Relative error
Another measure is the relative error and this is defined by $|\phi^* - \phi|/|\phi|$ if $\phi \neq 0$.

For our example above we see that $|x_1^* - x_1| = 2.4 \times 10^{-4}$ and $|x_2^* - x_2| = 2.4 \times 10^{-4}$ which look small. On the other hand the relative errors are $\frac{|x_1^* - x_1|}{|x_1|} = 8.6 \times 10^{-6}$ and $\frac{|x_2^* - x_2|}{|x_2|} = 6.7 \times 10^{-3}$. Thus the accuracy in computing $x_2$ is far less than in computing $x_1$. On the other hand if we compute $x_2$ via

$$x_2 = 14 - \sqrt{195} = \frac{1}{14 + \sqrt{195}}$$

we obtain $x_2 = 0.03576$ with a much smaller absolute error of $3.4 \times 10^{-7}$ and a relative error of $9.6 \times 10^{-6}$.

6.1.2 Errors in numerical modelling

These arise for example when the equations being solved are not the proper equations for the problem in question. An example is using the Euler equations for calculating the solution of flows where viscosity effects are important. No matter how accurately the solution has been computed, it may not be close to the real physical solution because viscous effects have been neglected through the computation.

6.1.3 Programming errors, and bugs

These are errors entirely under the control of the programmer. To eliminate these requires careful testing of the code and logic, as well as comparison with previous work. Even then, for your problem for which there may not be previous work to compare with, one has to do numerous self-consistency checks with further analysis as necessary.
6 ERRORS AND NUMERICAL STABILITY OF ODES

6.1.4 Truncation and discretization errors

These errors arise when we take the continuum model and replace it with a discrete approximation. For example, suppose we wish to solve

$$\frac{d^2 U}{dx^2} = f(x).$$

Using Taylor series we can approximate the second derivative term by

$$\frac{u(x_{i+1}) - 2u(x_i) + u(x_{i-1})}{h^2}$$

where we have taken a uniform grid with spacing $h$ say and node points $x_i$. As far the approximation of the equation is concerned we will have a truncation error given by

$$\tau(x_i) = \frac{d^2 U(x_i)}{dx^2} - f(x_i) = \frac{h^4}{12} \frac{d^4 Y(x_i)}{dx^4} + \ldots$$

Even though the discrete equations may be solved to high accuracy, there will be still an error of $O(h^2)$ arising from the discretization of the equations. Of course with more points, we would expect the error to diminish.

6.2 Initial Value Problems

Here we will look into the solution of ordinary differential equations of the type, say

$$\frac{dy}{dx} = f(x, y), \quad a \leq x \leq b,$$

subject to an initial condition

$$y(a) = \alpha$$

Most methods that are used can generalise readily to systems of differential equations of the type

$$\frac{dY}{dx} = F(x, Y), \quad a \leq x \leq b,$$

where

$$Y = (Y_1(x), Y_2(x), \ldots, Y_N(x))^T,$$

$$F = (f_1(x, Y), f_2(x, Y), \ldots, f_N(x, Y))^T,$$
with initial data

\[ Y(a) = \alpha, \quad (4) \]
say, where \( \alpha = (\alpha_1, \alpha_2, \ldots, \alpha_N)^T \). One question which arises immediately is, what about second and third order differential equations? Consider for example

\[ g''' + gg'' + \frac{1}{2}(1 - g'^2) = 0, \quad g(0) = g'(0) = g'(\infty) - 1 = 0, \quad (5) \]

which arise in many different applications. The answer to this is that if we take any differential equation, we can always write it as a system of first order equations. In (5) let

\[ Y_1 = g(x); Y_2 = g'(x); Y_3 = g''(x) \]

then (5) becomes

\[
\begin{pmatrix}
    Y_2 \\
    Y_3 \\
    -Y_1 Y_3 - \frac{1}{2}(1 - Y_1^2)
\end{pmatrix} = F 
\]

The boundary conditions in (5) are not of the form as in (4) because we have conditions given at two points \( x = 0 \) and \( x = 1 \). These are called boundary value problems and will be explained in more detail if you choose the ODE mini-project.

Consider equation (1). There are various mathematical results that we should be aware of, although we will not go too deeply into the implications or theory behind some of these. Suppose we define \( D \) to be the domain

\[ D = \{(x, y) \mid a \leq x \leq b, \quad -\infty < y < \infty\} \]

and \( f(x, y) \) is continuous on \( D \). If \( f(x, y) \) satisfies a Lipschitz condition on \( D \) then (1) has a unique solution for \( a \leq x \leq b \). Recall \( f(x, y) \) satisfies a Lipschitz condition on \( D \) means that there exists a constant \( L > 0 \) (called the Lipschitz constant) such that

\[ |f(x_1, y_1) - f(x_2, y_2)| \leq L|y_1 - y_2| \]

whenever \((x_1, y_1), (x_2, y_2)\) belong to \( D \).
6.3 Euler’s Method

This is the simplest of techniques for the numerical solution of (1). It is good for proving various results, but it is rarely used in practice because of far superior methods. For simplicity define an equally spaced mesh

\[ x_j = a + jh, \quad j = 0, \ldots, N \]

where \( h = (b - a)/N \) is called the step size.

We can derive Euler’s method as follows. Suppose \( y(x) \) is the unique solution to (1), and is twice differentiable. Then by Taylor’s theorem we have

\[ y(x_{i+1}) = y(x_i + h) = y(x_i) + y'(x_i)h + \frac{h^2}{2}y''(\xi) \]

(6)

where \( x_i \leq \xi \leq x_{i+1} \). But from the differential equation \( y'(x_i) = f(x_i) \), and \( y_i = y(x_i) \). This suggests the scheme

\[ w_0 = \alpha, \quad w_{i+1} = w_i + hf(x_i, w_i), \quad i = 1, 2, \ldots, N - 1, \]

(7)

for calculating the \( w_i \) which will be our approximate solution to the equation. This is called Euler’s method.

6.4 Truncation error for Euler’s method

Suppose that \( y_i = y(x_i) \) is the exact solution (1) at \( x = x_i \). Then the truncation error is defined by

\[ \tau_{i+1}(h) = \frac{y_{i+1} - (y_i + hf(x_i, y_i))}{h} = \frac{y_{i+1} - y_i}{h} - f(x_i, y_i). \]

(8)

for \( i = 0, 1, \ldots, N - 1 \). From earlier (6) we find that

\[ \tau_{i+1}(h) = \frac{h}{2}y''(\xi_i) \]
for some $\xi_i$ in $(x_i, x_{i+1})$. So if $y''(x)$ is bounded by a constant $M$ in $(a, b)$ then

$$|\tau_{i+1}(h)| \leq \frac{h}{2}M$$

and thus we see that the truncation error for Euler’s method is $O(h)$. In general if $\tau_{i+1} = h^p$ we say that the method is of order $h^p$. In principle if $h$ decreases, we should be able to achieve greater accuracy, although in practice roundoff error limits the smallest size of $h$ that we can take.

6.5 Stability

In practice most the Euler method should work reasonably well on standard problems. However certain types of problems (stiff problems) can cause difficulty and care needs to be exercised when choosing a method. Typically boundary layer type problems (say with a small parameter multiplying a highest derivative) are examples of stiff problems. This is where one needs to be aware of the stability properties of a particular method.

6.5.1 Consistency

A method is said to be consistent if the local truncation error tends to zero as the step size $\to 0$, i.e

$$\lim_{h \to 0} \max_i |\tau_i(h)| = 0.$$ 

This condition will almost always be met. It is a local condition.

6.5.2 Convergence

A method is said to be convergent with respect to the equation it approximates if

$$\lim_{h \to 0} \max_i |w_i - y(x_i)| = 0,$$

where $y(x)$ is the exact solution and $w_i$ an approximation produced by the method. This is the condition we are most interested in satisfying.

6.5.3 Stability

A stable method is one whose results depend continuously on the initial data. It can be proven that if the difference method is consistent with the differential equation, then the method is stable in this sense if and only if the method is convergent.
6.5.4 Absolute Stability

Is it enough just to have stability as defined above? Consider the solution of
\[
\frac{dy}{dx} = -30y, \quad y(0) = 1/3.
\]
The Euler method, although stable, has difficulty in computing the accurate solution of this problem. This means that we need something more than just the idea of stability defined above. Now consider
\[
\frac{dy}{dx} = \kappa y, \quad y(0) = \alpha, \quad \kappa < 0. \tag{9}
\]
The exact solution of this is \(y(x) = \alpha e^{\kappa x}\), so any approximation \(w_n\) should tend to zero as \(x\) tends to infinity. If we take the Euler method approximation and apply it to this equation we obtain
\[
w_{i+1} = Q(h\kappa)w_i.
\]
Then if \(w_0 = \alpha\) this will give
\[
w_n = Q(h\kappa)w_{n-1} = Q(h\kappa)^nw_0,
\]
So to satisfy the condition \(w_n \to 0\) as \(n \to \infty\) we have
\[
\lim_{n \to \infty} Q(h\kappa)^n = 0, \quad \text{or} \quad |Q(h\kappa)| < 1.
\]
We define the region \(R\) of absolute stability for the Euler method as the region in the complex plane satisfying:
\[
R = \{ h\kappa \in \mathbb{C}, |Q(h\kappa)| < 1 \}. \tag{10}
\]

A numerical method is A-stable if \(R\) contains the entire left half plane. In practice the above conditions place a limit on the size of the step size which we can use. For the Euler method, we have
\[
|Q(h\kappa)| = |1 + h\kappa| < 1
\]
which (since \(\kappa < 0\)) implies the condition
\[
h < -\frac{2}{\kappa}
\]
7 The Standard Libraries: Data Structures and Algorithms

In this section we give some brief examples of the uses of data types from the standard libraries. More detailed information can be found from web sources.

7.1 Strings

The C++ standard library `string` contains the a data structure to store arrays and characters (a string) and perform operations on them.

7.1.1 Declaration and Initialisation

First let us state that a `string` should be enclosed in double quotes, as oppose to a `char` that is enclosed by single quotes. A `string` can be initialised by another `string` or by a C-style sting, not by a `char` or an integer. An example of some declarations:

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

int main()
{
    string empty_string;  // an empty string
    string init_string="What\n";  // initialisation
    string a_string = init_string;
    std::cout << init_string << a_string << "\n";
}
```

Operations and Functions

Certain operations have been overloaded for strings to make them easier to work with. For instance, `+` will add one string to the end of another, and `+=` can be used to append to the end of a string. All of the usual comparison operators are provided. There are also many member function of `string` such as `size()` to return the size of a string, or `empty` to return `true` if it is empty. Since a `string` stores an array of characters, individual characters can be accessed via the `[]` operator.

Some example code and outputs:
### Example: Adding strings

```cpp
std::string str1 = "Hello";
std::string str2 = "World";
std::string str3 = str1 + " " + str2;
str3 += "\n";
std::cout << str3 << str3;
```

Output:

```
Hello World!!
Hello World!!
```

### Example: Compare strings

```cpp
std::cin >> str1 >> str2;
if (str1 > str2) std::cout << "str1 comes after str2 in the dictionary\n";
else if (str1 == str2) std::cout << "The strings are the same...\n";
else std::cout << str2 << " comes after " << str1 << " in the dictionary\n";
```

Output:

```
Input 2 words...
help please
please comes after help in the dictionary
```

Some other functions that you may find use are `std::string::find(t)` to return the position of a substring t in the string s, or `std::string::substr(n,m)` to return the string with m characters starting at s[n]. You may also wish to convert a C++ string to a C-style string for use with C functions, this is provided by `std::string::c_str()`.

### 7.1.2 Input and Output

The stream operators have been overloaded to work with strings so they should work as you would expect. The only thing to mention is that spaces (on input) will terminate input by default. So

```cpp
std::cout << "What is your name?\n";
std::cin >> str;
std::cout << "Hi, \n" << str << "!\n";
```
will display:

```
What is your name?
Paul Johnson
Hi Paul!
```

So to ignore white spaces we need to use the `getline()` function.

```
cout << "What is your name?\n";
gcin , str); // read the whole line
cout << "Hi, " << str << "!\n"
```

Check web resources for more detailed information.

### 7.2 Complex Numbers

The library `<complex>` contains the definition of a template-based complex number class, with corresponding definitions of the basic arithmetic operators +,-,/,*. The functions `abs(complex<T> &)` and `arg(complex<T> &)` return the absolute value and argument of the complex number and the hyperbolic, logarithmic and circular functions have all been overloaded to handle complex numbers. The code segment below shows typical use of this class

```c++
#include <iostream>
#include <complex>
// complex actually lives in the std namespace
using namespace std;
int main()
{
    // Declare two complex numbers 1+i and 1-i
    complex<double> A(1,1), B(1,-1);
    // Declare a third complex number
    complex<double> C;
    // C is A divided by B (overloaded division operator here)
    C = A/B;
    // This will produce the output i (0,1)
    cout << C << endl;
    // Output the absolute value (1) and argument (pi/2)
    cout << abs(C) << " " << arg(C) << endl;
    // Logarithm of complex number
```
cout << log(B) << endl;
}

The output from the code is:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,1)</td>
<td>1</td>
<td>1.5708</td>
</tr>
<tr>
<td>(0.346574, -0.785398)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Standard Containers

A container is an object whose main purpose is to hold other objects, examples include vectors, sets and lists. Objects can be inserted into and removed from containers, and the objects in a container can be sorted and copied into other containers. The standard containers have different header files, usually the name of the container, and the most useful are shown in Table 6.

<table>
<thead>
<tr>
<th>Type</th>
<th>Example</th>
<th>Header File</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector&lt;T&gt;</td>
<td>a variable-sized 1D array</td>
<td>&lt;vector&gt;</td>
</tr>
<tr>
<td>list&lt;T&gt;</td>
<td>a doubly-linked list</td>
<td>&lt;list&gt;</td>
</tr>
<tr>
<td>set&lt;T&gt;</td>
<td>a set</td>
<td>&lt;set&gt;</td>
</tr>
<tr>
<td>map&lt;key,val&gt;</td>
<td>an associative array</td>
<td>&lt;map&gt;</td>
</tr>
<tr>
<td>stack&lt;T&gt;</td>
<td>a stack</td>
<td>&lt;stack&gt;</td>
</tr>
<tr>
<td>queue&lt;T&gt;</td>
<td>queue</td>
<td>&lt;queue&gt;</td>
</tr>
</tbody>
</table>

Table 6: Some of the standard container types in C++.

The standard containers are designed to be as similar as possible. For example, every container has a member function size() that returns the number of objects in the container. In the following sections, we shall examine some of the standard containers in more detail.

### 7.3 Vectors

A vector, v, contains an array of N objects indexed from 0 to N-1 and provides efficient, direct access to the ith element via subcripting, v[i], or (with range checking) v.at(i). A vector can be initialised in many different ways:
The function `resize()` changes the size of a vector, which is therefore a dynamic data structure – it can change in size during the execution of the program. A bonus of using the vector array, rather than writing your own dynamic arrays, is that memory allocation and de-allocation is handled by the class.

### 7.3.1 Stack Operations

Vectors also support stack operations, such as `push_back()`, which adds an entry at the end, `pop_back()`, which removes the last entry, `front()`, which returns the value of the first entry and `back()`, which returns the value of the last entry.
The output from the above program is:

```
10
1 10
8
```

### 7.3.2 Iterators

Every standard container provides iterators that can be used to step through the objects held in the container. An iterator is merely an object that is used to refer to the objects in the container in some particular order. It can be regarded as a pointer to the objects in the container, but its exact type depends upon the implementation and the container. For a standard container, C, the iterator is always called `C::iterator` and the functions `begin()` and `end()` return iterators to the first object in the container and the last+1 object, respectively, and allow forward traversal of the objects in the container. In order to access the object at the position of the iterator the syntax is like that of pointers.

```cpp
vector<int> vec(10, 5); // An array of 10 integers, all 5
// Print out the value of the first entry, using the iterator
cout << *vec.begin();
```

For a standard container, C, there is also a reverse iterator, `C::reverse_iterator` and the functions `rbegin()` returns a reverse iterator to the end of the array and `rend()` returns a reverse iterator to one before the start of the array. These allow reverse traversal of the objects in the container.

```cpp
#include <iostream>
#include <vector>
using namespace std;
int main() {
    // Declare a vector of ten integers
    vector<int> vec(10);
    // Fill the vector
    for (unsigned int i = 0; i < 10; i++) { vec[i] = 2 * i; }
    // Print out the value of the first entry, using the iterator
    cout << *vec.rbegin();
}```
The above code fragment illustrates the use of iterators for vectors and gives the following output

Traversing Forwards
0 2 4 6 8 10 12 14 16 18
Traversing Backwards
18 16 14 12 10 8 6 4 2 0

In fact, entries in a vector may be traversed by direct access (using subscripts e.g. `vec[i]`, but subscripts are not available in all containers. Using iterators is a generic way of traversing a standard container and should be used in standard algorithms.

### 7.3.3 List Operations

Vectors also support list-type operations `insert()` and `erase()` to add and delete elements at given positions. These operations are not efficient, but can be very convenient and their use is illustrated in the following program

```cpp
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    // Declare a vector of ten integers
```


```cpp
vector<int> vec(10);
// Fill the vector
for(unsigned int i=0; i<10; i++) { vec[i] = 2*i; }
// Insert the number 100 at the beginning of the vector
vec.insert(vec.begin(), 100);
// Insert two copies of 15 before the last entry in the vector
vec.insert(vec.end() - 2, 2, 15);
// Check the size of the vector --- it should be 13
cout << vec.size();
// Output the first entry
cout << " " << vec.front() << endl
// Now delete the 5th entry of the array
vec.erase(vec.begin()+4);
// Delete the last two entries in the array
vec.erase(vec.end()-2, vec.end());
// Check the size of the vector --- it should be 10
cout << vec.size() << endl;
// Output the last entry
cout << " " << vec.back() << endl
```

which produces the following output

```
13 100
10 15
```

### 7.4 List

A list is designed for efficient insertion and deletion of objects at given positions. It does not have a subscript [] operator, but otherwise has all the functionality of a vector. In addition, a list contains the `push_front()` and `pop_front()` functions for adding or removing entries from the start of the list efficiently. A list also contains the useful `remove()` function which can be used to remove all entries with a particular value.

```cpp
#include<iostream>
#include<list>
using namespace std;
int main()
```
The above program produces the output 4 2.

A list does not have the subscript operator [], so iterators must be used to traverse the list. A subtlety is that the only operation that is permitted on a list iterator is the iteration operator ++. The “random-access” operations += and -= are not defined, so to find the iterator to the fourth entry in a list the following ugly loop must be used

```cpp
list<int>::iterator p = L.begin();
for (int i=1;i<=4;i++) p++;
```

Lists are not designed for this! If you need to do this a lot, you are using the wrong data structure. A much more common procedure is to use the standard algorithm \texttt{find()} to find the iterator to the position of the first instance of a particular value in a sequence. It returns \texttt{end()} if there is no such value.

```cpp
//Find the iterator to the value 100 in the list L
list<int>::iterator p = find(L.begin(), L.end(), 100);
```


7.5 Maps and Sets

A map is a container whose values are referenced by a key, which is a generalised version of an array index. Each entry of the map has two values, the key (first) and the actual value (second). It is best illustrated by example

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

int main()
{
    map<string, int> exam_result; // Entries are string, integer pairs
    // Set a couple of exam results
    exam_result["026578"] = 57;
    exam_result["5686"] = 10;
    // Traverse using iterators and print what we have
    for(map<string, int>::iterator p = exam_result.begin(); p!=exam_result.end(); p++)
    {
        cout << "ID: <" << p->first << "Result: " << p->second << endl;
    }
}
```

The output is

ID: 026578 Result: 57
ID: 5686 Result: 10

A set is a collection of objects with the proviso that each value can occur only once. It can be regarded as a map with keys (which must also be unique), but no values. An example of the use of sets is

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

int main()
{
    set<int> S; // An empty set of integers
    S.insert(10); // Add 10 to the set
    S.insert(5); // Add 5 to the set
```
7 STANDARD LIBRARIES

S.insert(10); // 10 is already in the set, so this will not be added
// Number of elements in set is two
cout << S.size() << endl;
S.erase(5); // Delete the entry 5 from the set
}

7.6 Stacks and Queues

A stack is a data structure that does three things efficiently: top() returns the entry at the top of the stack, push() adds an element to the top of the stack, and pop() removes and item from the top of the stack.

```cpp
#include <iostream>
#include <stack>
using namespace std;
int main()
{
    stack<double> S; // An empty stack of doubles
    S.push(2.5); // Add 2.5 to the stack
    S.push(3.1); // Add 3.1 to the stack
double x = S.top(); // Read the top entry from the stack, x = 3.1
    S.pop(); // Remove top entry, 3.1
}
```

A queue is a data structure that supports efficient addition of an entry at the end of the queue push(), and removal at the front pop().

```cpp
#include <iostream>
#include <queue>
using namespace std;
int main()
{
    queue<double> Q; // An empty queue of doubles
    Q.push(2.5); // Add 2.5 to the end of the queue
    Q.push(3.1); // Add 3.1 to the queue
double x = Q.front(); // Read the first entry from the queue, x = 2.5
double y = Q.back(); // Read the last entry from the queue, y = 3.1
    Q.pop(); // Remove front entry, 2.5
```
7.7 Standard Algorithms

A number of standard algorithms are provided in the C++ libraries, many of which are defined in the header `<algorithm>`. There are too many to cover here, but some of the most important are `sort()`, which sorts a sequence with good average efficiency, `reverse()`, which reverses the order of a sequence, and `rotate()`, which cyclically permutes a sequence. There are also the algorithms `max_element()`, which returns the iterator to the largest value in a sequence, `min_element()`, which returns the iterator to the smallest value in a sequence, and `count()`, which counts the number of occurrences of a value in a sequence. It is possible, in all cases, to overload the operator used for comparison, but the default is `>.

An example code is shown below:

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{
    // Define a vector and set the initial values
    vector<int> vec(5);
    // Sort the vector in ascending numerical order
    sort(vec.begin(), vec.end()); // {-10,-5,0,1,3}
    // Now reverse the vector
    reverse(vec.begin(), vec.end()); // {3,1,0,-5,-10}
    // Now cyclically permute so that the vec.begin()+2 (third element)
    // becomes the first element in the array
    rotate(vec.begin(), vec.begin()+2, vec.end()); // {0,-5,-10,3,1}
    for(int i=0; i<5; i++) cout << vec[i] << " ";
    cout << endl;
    cout << "Maximum element is " << *max_element(vec.begin(), vec.end()) << endl;
    cout << "Minimum element is " << *min_element(vec.begin(), vec.end()) << endl;
}
```
There are plenty of other algorithms out there. If you are interested, you should look at some of the many books and articles on the C++ standard template library. Typing “C++ standard template library” into google also leads to lots of useful information.