C++ 及其他语言的科学编程项目

Victor Eijkhout

2020

目录

目录

第一部分

简单项目

第 1 章

质数

本章你将完成一系列关于质数的练习，这些练习彼此构建。每节列出了所需的先决条件。反之，这里的练习也会被前面章节所引用。

1.1 算术

练习 1.1. 读取两个数字并打印它们的模数。模运算符是 x%y。

- 你能在不使用运算符的情况下计算模数吗?
- 在两种情况下，负输入会得到什么结果?
- 在输出前将所有结果赋值给一个变量。

1.2 Conditionals

练习 1.2. 读取两个数字并打印一条消息，说明第二个数字是否为第一个数字的除数:

代码:

```
1 ///divisiontest.cpp
2   整型 数值, 除数; b li
3   io♦_ _s a d v sor;
4     /* ... */
5   if (
6     /* ... */
7       ) {
8     cout <<"确实, "<< 除数
9          << " is a divisor of "
10         << number << '\n';
11  } else {
12    cout << "No, " << divisor
13         << " is not a divisor of "
14         << number << '\n';
15  }
```

输出

```
[质数] 除法:

(回显 6 ; 回显 2 )  |
     除法测试
输入一个数字:
输入一个试验除数:
事实上, 2 是 6 的除数

(回显 9 ; 回显 2 )  |
     除法测试
输入一个数字:
Enter a trial divisor:
No, 2 is not a divisor of 9
```

1.3 循环

练习 1.3. 读取一个整数，并通过测试较小的数是否能整除该数来设置一个布尔变量以确定其是否为质数。

打印最终消息

您的数字是质数

or

您的数字不是质数：它可以被 .... 整除

在此情况下你只需报告找到的一个因数。

向屏幕输出信息几乎从不是严肃程序的核心目的。因此在上一练习中，我们假设数字的素数性（或非素数性）结论将被程序后续部分使用。所以你需要存储这一结论。

练习 1.4. 用布尔变量表示输入数字的素数性，重写前一个练习。

练习 1.5. 读入一个整数 r。如果是素数，打印提示信息；若非素数，则找出整数 $p \leq q$ 使得 $r = p \cdot q$ 成立，且 p 与 q 尽可能接近。例如对于 $r = 30$ 应输出 `5,6` 而非 `3,10`。允许使用函数 `sqrt`。

1.4 函数

chapter]ch:function

之前你编写了几行代码来测试一个数是否为质数。现在我们将这段代码转化为一个函数。

练习 1.6. 编写一个函数 `is_prime`，该函数接收一个整数参数，并返回一个布尔值，表示该参数是否为质数。

```
int main(){bool
isprime;isprime= 是_质数
(13);
```

编写一个主程序，读取输入的数字，并打印该布尔值的值。（布尔值如何渲染？参见教材第 12.2.2 节。）

你的函数包含一个还是两个 `return` 语句？你能想象另一种可能性是什么样子吗？你对此有何支持或反对的论点？

1.5 While 循环

练习 1.7. 利用你的素数检测函数 *is_prime*，编写一个能输出多个素数的程序：

- 从输入读取一个整数 *how_many*，表示需要输出多少个（连续）素数。
- 输出相应数量的连续素数，每个素数独占一行。
- （提示：维护一个变量 *number_of_primes_found*，每当发现新素数时递增该变量。）

1.6 类与对象

Exercise 1.8. Write a class *primegenerator* that contains:

- Methods *number_of_primes_found* and *nextprime*;
- Also write a function *isprime* that does not need to be in the class.

你的主程序应如下所示：

```
// /6primesbyclass.cpp
  cin >> nprimes;
  primegenerator sequence;
  while (sequence.number_of_primes_found()<nprimes) {
    int number = sequence.nextprime();
    cout << "Number " << number << " is prime" << '\n';
  }
```

在前面的练习中，你定义了 *primegenerator* 类，并创建了该类的一个对象：

素数生成器序列；

但你可以创建多个生成器，它们各自拥有独立的内部数据，因此彼此互不影响。

练习 1.9. The *Goldbach* 猜想 指出，从 4 开始的每个偶数都是两个素数之和 $p+q$。编写一个程序来测试这一点，针对你读取的边界内的所有偶数。使用你在练习 *primegenerator* 类中开发的 45.8 (教材)。

这是一个采用自顶向下方法的绝佳练习！

1. 对偶数 e 进行外层循环。2. 对于每个 e，生成所有质数 p。3. 由 $p+q=e$ 可推导出 $q=e-p$ 为质数：需验证该 q 是否为质数。

对于每个偶数 e，则打印 e, p, q，例如：

数字 10 是 3+7

若存在多种可能性，仅打印首个找到的结果。

哥德巴赫猜想的一个有趣推论是，每个从 5 开始的素数都与其他两个素数等距。

哥德巴赫猜想 指出每个偶数 $2n$ （从 4 开始）都可表示为两个质数 $p+q$ 之和：

$$2n = p + q.$$

等价地说，每个数 n 与两个质数的距离相等：

$$n = \frac{p+q}{2} \qquad \text{or} \qquad q - n = n - p.$$

特别地，这对每个质数都成立：

$$\forall_{r\,\text{prime}} \exists_{p,q\,\text{prime}} : r = (p+q)/2 \text{ is prime.}$$

练习 1.10.

编写程序验证此猜想。至少需要一个循环来测试所有质数 r；对于每个 r，需找到与之等距的质数 p,q。为此需要使用两个生成器，还是一个就足够？是否需要三个生成器来处理 p, q,r?

对于每个 r 值，当程序找到 p, q 值时，打印 p, q,r 三元组并继续处理下一个 r。

1.6.1 异常

练习 1.11. 重新审视素数生成器类（练习 45.8 （教材）），并使其在候选数字过大时抛出异常。（您可以硬编码此最大值，或使用限制；参见教材第 24.2 节。）

代码：

```
1 ///genx.cpp
2   尝试 {
3     do {
4       auto cur = primes.nextprime();
5       cout << cur << '\n';
6     } while (true);
7   } 捕获 (   string s ) {
8     cout << s << '\n';
9   }
```

输出[
ipr mes] genx:

```
9931
9941
9949
9967
9973
达到最大整数值
```

1.6.2 质因数分解

Design a class 整数 将其值存储为其质因数分解形式 omposition. For instance,

$$180 = 2^2 \cdot 3^3 \cdot 5 \qquad \Rightarrow \qquad \texttt{[2:2, 3:2, 5:1]}$$

你可以将此分解本身实现为一个向量（ i-th 位置存储第 i 个质数的指数），但我们改用映射。

练习 1.12. 编写一个从 整数 构造 `int` 的构造函数，以及将分解转换回传统形式的方法 `as_int` / `as_string`。开始时假设每个质因数仅出现一次。

代码:

```
// /decomposition.cpp
i2
cou        .as_string() << ": "
      << i2.as_int() << '\n';

整数 i6(6); 常量 t<< i6 赋
值为 s_
u          .     tring() << ": "
      << i6.as_int() << '\n';
```

输出
[质数] 分解 26:

2^1 : 2
2^1 3^1 : 6

练习 1.13. 扩展前一练习，为质因数添加 > 1 重数。

代码:

```
/// deCom osition.c
i180(180);
cout << i180.as_string() << ": "
      << i180.as_int() << '\n';
```

输出[
i pr mes] decomposition180:

2 的 2 次方 × 3 的 2 次方 × 5
的 1 次方 = 180

实现整数的加法与乘法运算。

实现一个表示有理数的 `Rational` 类，该类由两个整数对象构成。该类需包含加法与乘法的运算方法，若已掌握运算符重载，请通过重载运算符实现。

确保始终对分子和分母进行公约数约分。

1.7 范围

练习 1.14. 编写基于范围的代码来测试

$$\forall_{\text{prime } p} : \exists_{\text{prime } q} : q > p$$

Exercise 1.15. Rewrite exercise 1.10, using only range expressions, and no loops.

练习 1.16. 在上述哥德巴赫猜想练习中，你可能需要两个不同起始值的素数序列。能否修改代码使其读取

```
all_of( primes_from(5) /* et cetera */
```

1.8 其他

The following exercise requires `std::optional`, which you can learn about in section 24.6.2 (textbook).

练习 1.17. 编写一个函数 *first_factor*，可选择性地返回给定输入的最小因数。

```
// /optfactor.cppauto factor =first_factor(number);if
(factor.has_value())cout <<"找到因数:"<< factor.value()
<< '\n';
```

1.9 埃拉托斯特尼筛法

埃拉托斯特尼筛法是一种用于筛选质数的算法，它逐步过滤掉所发现质数的所有倍数。

1. 从整数 2 开始：$2, 3, 4, 5, 6, \ldots$2. 第一个数字 2 是质数：记录它并移除所有倍数，得到

$$3, 5, 7, 9.11, 13, 15, 17 \ldots$$

3. 第一个剩余数字 3 是素数：记录它并移除所有倍数，得到

$$5, 7, 11, 13, 17, 19, 23, 25, 29 \ldots$$

4. 第一个剩余数字 5 是素数：记录它并移除所有倍数，得到

$$7, 11, 13, 17, 19, 23, 29, \ldots$$

1.9.1 数组实现

The sieve can be implemented with an array that stores all integers.

练习 1.18. 读入一个整数作为待测试的最大数值。创建一个等长的整数数组，将元素初始化为连续整数。应用筛法算法找出素数。

1.9.2 流实现

使用数组的缺点在于需要预先分配数组空间。更重要的是，数组大小取决于待测试整数的数量，而非要生成的素数数量。我们将采用前文提到的生成器对象思路，将其应用于筛法算法：现在会有多个生成器对象，每个以前一个生成器为输入，从中剔除特定的倍数。

练习 1.19. 编写一个流类来生成整数并通过指针使用它。

代码:

```
// /ints.cpp
  for (int i=0; i<7; ++i)
    cout <<"下一个整数: "
         << the_ints->next() << '\n';
```

输出[筛] 整数:

```
下一个整数 : 2
下一个整数 : 3
下一个整数 : 4
下一个整数 : 5
下一个整数 : 6
下一个整数 : 7
下一个整数 : 8
```

接下来，我们需要一个以另一个流作为输入并从中过滤出值的流。

练习 1.20. 编写一个类 *filtered stream*，其构造函数为 _

```
filtered_stream(int filter,shared_ptr<stream> input);
```

that

1. 实现 *next*，提供过滤后的值，2. 通过调用输入流的 *next* 方法并过滤掉不需要的值。

代码:

```
// /odds.cpp
  auto integers =
    make_shared<stream>();
  auto odds =
    shared_ptr<stream>
    ( new filtered_stream(2,integers) );
  for (int step=0; step<5; ++step)
    cout <<"下一个奇数 :"
         << odds->next() << '\n';
```

输出[sieve] odds:

```
下一个奇数 : 3
下一个奇数: 5
下一个奇数: 7
下一个奇数: 9
下一个奇数: 11
```

现在你可以为每个质数创建过滤后的 _ 流来实现埃拉托斯特尼筛法。

练习 1.21. 编写一个按如下方式生成质数的程序。

- 维护一个当前的流，初始时即为素数流。
- 重复执行以下操作:
  - 记录来自当前流的首个项，即为新发现的素数；- 并将当前流设置为一个新流，该新流以当前流为输入，过滤掉刚找到素数倍数的项。

1.10 范围实现

若我们写出素数的定义

$$\begin{aligned} D(n,d) &\equiv n|d=0 \\ P(n) &\equiv \forall_{d\leq\sqrt{N}}: \neg D(n,d) \end{aligned}$$

可看出这涉及我们迭代处理的两个流:

1. 首先是所有 d 的集合，使得 $d^2 \leq n$；然后 2. 我们有一组布尔值
用于测试这些 d 值是否为除数。

练习 1.22。 使用 *iota* 范围视图生成从 2 到无穷大的所有整数，并找到一个范围视图，在最后一个可能的除数处截断序列。

然后使用 *all_of* 或 *any_of* 范围化算法来测试这些潜在除数中是否有任何一个实际上是除数，从而判断你的数字是否为质数。

练习 1.23。 使用 *filter* 视图从 *iota* 视图中筛选出那些是质数的元素。

练习 1.24。 创建一个可以范围化的 *primes* 类：

代码：

```
// /rangeclass.cpp
  primegenerator allprimes;
  for ( auto p : allprimes ) {
    cout << p << ", ";
    if (p>100) break;
  }
  cout << '\n';
```

输出 [primes] 范围：

缺失片段
../code/primes/range.runout

1.11 用户友好性

使用 *cxxopts* 包（第 63.2 节（教材））为某些素数判定程序添加命令行选项。

Exercise 1.25. Take your old prime number testing program, and add commandline options:

- -h 选项应打印用法信息；
- 指定单个整数 --test 1001 应打印该数字以下的所有素数；
- 指定一组整数 --tests 57,125,1001 应测试这些数字的素性。

第 2 章

几何

在这组练习中，你将编写一个小的 ‘几何’ 包：用于操作点、线、形状的代码。这些练习主要使用教材第 9 节的内容。

2.1 基本函数

练习 2.1. 编写一个函数，输入为（float 或 double） x,y，返回点 (x,y) 到原点的距离。

测试以下点对：1,0；0,1；1,1；3,4。

练习 2.2. 编写一个函数，输入为 x,y,θ，该函数通过旋转点 (x,y) 角度 θ 来改变 x 和 y。

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

你的代码应如下所示：

代码：

```
// /rotate.cpp
  const float pi = 2*acos(0.0);
  float x{1.}, y{0.};
  rotate(x,y,pi/4);
  cout << "Rotated halfway: ("
       << x << "," << y << ")" << '\n';
  旋转(x,y,pi/4);t<< "
  Rttdt      o a e    o the y-axis: ("
       << x << "," << y << ")" << '\n';
```

输出

```
[几何] 旋转:
Rotated halfway:
      (0.707107, 0.707107)
旋转至y-轴：(0,1)
```

2.2 Point 类

一个类可以包含基础数据。在本节中，你将创建一个 Point 类来建模笛卡尔坐标及基于坐标定义的函数。

练习 2.3. 创建带构造函数的 Point 类

```
点（ 浮点数 x 坐标， 浮点数 y 坐标）;
```

编写以下方法:

- distance_to_origin 返回一个浮点数。
- angle 计算向量 (x,y) 与 x 轴的夹角。

Exercise2.4. 扩展上一练习中的 Point 类，添加一个方法：distance，用于计算该点与另一点之间的距离：若 p 、 q 为 Point 对象,

```
p.距离（q）
```

计算两者之间的距离。

Exercise2.5. 编写一个方法 *halfway*，给定两个 Point 对象 p 、 q，构造位于两者中间的点，即 $(p+q)/2$:

```
点 p(1,2.2),q(3.4,5.6); 点
h=p.中点（q）;
```

你可以直接编写此函数，也可以先编写 Add 和 Scale 函数再组合使用。（后续你将学习运算符重载。）

如何打印一个 *Point* 以确保正确计算中点?

练习 2.6. 为点类创建默认构造函数:

```
Point() { /* default code */ }
```

which you can use as:

```
Point p;
```

但需表明该点未定义:

代码:

```
// /linear.cpp   Pointp3; cout <<"未初始化的点: "
        <<'\n';
p3.printout(); cout<< "使用未初始化的点:
"        <<'\n'; auto p4 =Point(4,5)+
p3;   p4.printout();
```

输出

```
[geom] linearnan:
未初始化的点: 点: nan,nan 使
用未初始化的点: 点: nan,nan
```

提示: 参见章节 26.3.3 (教材)。

练习 2.7。 重新完成练习 46.2 (教材)，使用 Point 类。你的代码现在应如下所示:

```
新点 = 点 . 旋转 (alpha);
```

练习 2.8. 进阶。你能创建一个 `Point` 类，使其能适应任意空间维度吗？提示：使用 `vector`；参见教材第 10.3 节。你能创建一个不显式指定空间维度的构造函数吗？

2.3 在一个类中使用另一个类

Exercise 2.9. Make a class `LinearFunction` with a constructor:

```
线性函数 ( 点输入 _p1, 点输入 _p2);
```

以及一个成员函数

```
浮点数 评估 _ 在 ( 浮点数 x);
```

可将其用作:

```
LinearFunction line(p1,p2);cout << "在4.0 处的值 :"<<
line.evaluate_at(4.0) << endl;
```

练习 2.10. 创建一个类 LinearFunction，包含两个构造函数:

```
线性函数 ( 点输入 _p2 ); 线性函数 ( 点输入 _p1, 点输入 _p2 );
```

其中第一个构造函数表示通过原点的直线。
再次实现 evaluate 函数，以便

```
线性函数 line(p1,p2);cout << " 在 4.0 处的值 :"<<line.evaluate_
at(4.0) << endl;
```

Exercise2.11. Revisit exercises 46.2 (textbook) and 46.7 (textbook), introducing a `Matrix` 类。您的代码现在可以这样写

```
新点 = 点 . 应用 ( 旋转 _ 矩阵 );
```

or

```
新点 = 旋转 _ 矩阵。应用 ( 点 );
```

你能支持其中一种方式吗?

假设你想编写一个 Rectangle 类，它可能包含诸如 float Rectangle::area() 或 bool Rectangle::contains(Point) 等方法。由于矩形有四个角，你可以在每个 Rectangle 对象中存储四个 Point 对象。然而，这里存在冗余：你只需要三个点就能推断出第四个点。让我们考虑边为水平和垂直的矩形情况；那么你只需要两个点。

预期 API:

```
float Rectangle::area();
```

It would be convenient to store width and height; for

```
bool Rectangle::contains(Point);
```

存储左下角 / 右上角点会很方便。

Exercise 2.12.

1. Make a class `Rectangle` (sides parallel to axes) with a constructor:

   ```
   矩形（点 左下角，浮点型宽度，浮点型高度）;
   ```

   The logical implementation is to store these quantities. Implement methods:

   ```
   float 面积(); float 右边缘_x();float 上边缘_y();
   ```

   and write a main program to test these.
2. Add a second constructor

   ```
   矩形（点 左下角，点 右上角）;
   ```

   你能弄清楚如何使用成员初始化列表来编写构造函数吗?

练习 2.13。复制上一练习的解决方案，并重新设计你的类，使其存储两个 Point 对象。你的主程序应保持不变。

上一个练习阐明了一个重要观点：对于设计良好的类，你可以更改其实现（例如出于效率考虑），而使用该类的程序无需变动。

2.4 继承关系（Is-a 关系）

练习 2.14. 取出你之前定义的通过一个点、宽度和高度构建 Rectangle 的代码。

创建一个继承自 Rectangle 的类 Square。它应具备从 Rectangle 继承而来的 area 函数定义。

首先思考：Square 的构造函数应该如何设计?

练习 2.15. 重新审视 LinearFunction 类。为其添加 slope 和 intercept 方法。

现在将 LinearFunction 泛化为 StraightLine 类。两者几乎相同，除了垂直线的情况。斜率和截距不适用于垂直线，因此设计 StraightLine 时应内部存储定义点。让 LinearFunction 继承该设计。

2.5 指针

以下练习略显人为设计。

练习 2.16. 创建一个 `DynRectangle` 类，该类由两个指向 `Point` 对象的共享指针构造：

```
// /dynrectangle.cpp
  auto
    origin  = make_shared<Point>(0,0),
    fivetwo = make_shared<Point>(5,2);
  DynRectangle lielow( origin,fivetwo );
```

计算面积，缩放右上角点，并重新计算面积：

代码：

```
1// / 动态矩形 .cpp
2   cout << "面积 :"<<       lielow. 区域 () <<
      '\n';
3     /*
4       / // 将 'fivetwo' 点放大两倍
5   cout << "面积 :"<< lielow.area() <<
      '\n';
```

输出 [指针] dynrect:

```
面积 : 10
面积 : 40
```

You can base this off the file `pointrectangle.cpp` in the repository

2.6 更多内容

Rectangle 类最多存储一个角点，但有时拥有所有四个角点的数组可能会更方便。

练习 2.17. 添加一个方法

```
const vector<Point> &corners()
```

向 Rectangle 类添加该方法。结果将返回一个包含所有四个角点的数组（顺序不限）。通过编译错误演示该数组不可被修改。

练习 2.18. 重访练习 2.5 并将其中的 `add` 和 `scale` 函数替换为运算符重载。提示：对于 `add` 函数可能需要使用 'this' 。

2. 几何学

第 3 章

零点求解

3.1 二分法求根

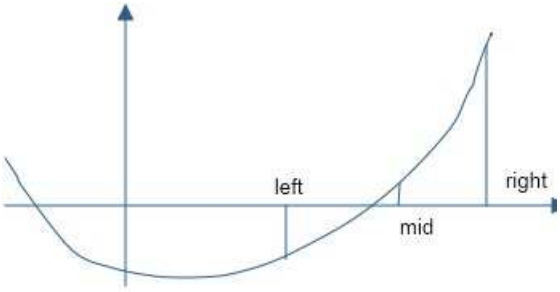


图 3.1：区间二分法求根

对于许多函数 f，要找到它们的零点（即使 $f(x) = 0$ 成立的 x 值）往往无法通过解析方法实现。此时必须借助数值求根方案。本项目将逐步开发一个简单方案的复杂实现：通过二分法求根。

该方案从函数符号相反的两个点开始，根据中点的函数符号，将左端点或右端点移动到中点位置。示意图见 3.1。

在 3.2 节中，我们将探讨牛顿法。

此处我们并不关注这些方法之间的数学差异（尽管它们很重要），而是利用这些方法来练习某些编程技巧。

3.1.1 简单实现

让我们逐步开发第一个实现版本。为确保代码正确性，我们将采用测试驱动开发（ TDD ）方法：针对每个功能模块，先编写测试用例验证其正确性，再将其集成到主代码中。（关于 TDD 及 Catch2 框架的更多内容，请参阅 68.2 节（教材）。）

3.1.2 多项式

首先，我们需要一种表示多项式的方法。对于一个 d 次多项式，需要 $d+1$ 个系数：

$$f(x) = c_0 x^d + \cdots + c_{d-1} x^1 + c_d \tag{3.1}$$

我们通过将系数存储在向量 <double> 中实现这一点。我们做出以下任意决定

1. 令该向量的第一个元素为最高次幂的系数；2. 为了使这些系数正确定义一个多项式，该首项系数必须非零。

让我们从一个固定的测试多项式开始，该多项式由函数 *set_coefficients* 提供。为了使该函数能提供一个正确的多项式，它必须满足以下测试：

```
// /testzeroarray.cppTEST_CASE("系数代表多项式"[1]")
{vector<double>coefficients = { 1.5, 0., -3 };
REQUIRE(coefficients.size()>0 );
REQUIRE(coefficients.front()!=0. );}
```

习题 3.1. 编写一个名为 *set coefficients* 的例程，用于构建系数向量：_

```
// /findzeroarray.cpp
  vector<double> coefficients = set_coefficients();
```

并使其满足上述条件。

首先编写一个硬编码的系数集，然后尝试从命令行读取它们。

习题 3.2. 附加题：使用 *cxxopts* 库（参考教材第 63.2.2 节）通过命令行指定系数。

前文我们提出了两个条件，一个数字数组必须满足这些条件才能作为多项式的系数。你的代码很可能会对此进行测试，因此让我们引入一个布尔函数 *is_proper_polynomial*：

- 此函数返回 **true**，如果数字数组满足这两个条件；
- 如果任一条件不满足，则返回 **false**。

In order to test your function *is_proper_polynomial* you should check that

- 它能识别正确的多项式，且
- 对于未正确定义多项式的不当系数，它会返回失败。

练习 3.3. 编写一个函数 *is_proper_polynomial*，并为其编写通过和失败的单元测试：

```
vector<double>good = /* 正确的系数 */;REQUIRE(is_
proper_polynomial(good) );vector<double>notso = /* 不
正确的系数 */;REQUIRE(not is_proper_
polynomial(notso) );
```

接下来我们需要多项式求值。我们将构建一个函数 *evaluate_at*，其定义如下：

```
// /findzerolib.hpp
double evaluate_at( const std::vector<double>& coefficients,double x);
```

你可以通过（至少）两种方式解释系数数组，但根据方程（3.1），我们规定了其中一种特定的解释方式。

因此我们需要一个测试来验证系数确实是以最高次项系数优先的方式解释，而非以最低次项系数优先。例如：

```
// /testzeroclass.cpp 多项式 second( {2,0,1} ); // 正确解释：2x^2 + 1
REQUIRE( second.is_proper() ); REQUIRE( second.evaluate_at(2)
== Catch::Approx(9) ); // 错误解释：1x^2 + 2
REQUIRE(second.evaluate_at(2) != Catch::Approx(6) );
```

（此处我们省略了 *TEST_CASE* 头部。）

现在我们来编写通过这些测试的函数：

练习 3.4. 编写一个函数 *evaluate_at* 用于计算

$$y \leftarrow f(x).$$

并确认它通过了上述测试。

```
double evaluate_at( polynomial coefficients,double x);
```

For bonus points, look up *Horner's rule* and implement it.

实现了多项式函数后，我们可以开始着手算法部分。

3.1.3 左 / 右搜索点

假设 x_-,x_+ 满足以下条件

$$x_- < x_+, \quad \text{and} \quad f(x_-) \cdot f(x_+) < 0,$$

即左右端点的函数值符号相反。那么区间 (x_-,x_+) 内必存在一个零点；如图 3.1 所示。

但如何找到搜索的这些外部边界呢?

若多项式为奇数次，通过从任意两个起始点向左和向右足够远的位置，可以找到 x_-,x_+。对于偶数次多项式，则不存在如此简单的算法（实际上，可能根本不存在零点），因此我们放弃这一尝试。

我们首先编写一个函数 *is_odd* 来测试多项式是否为奇数次。

3. 零点求解

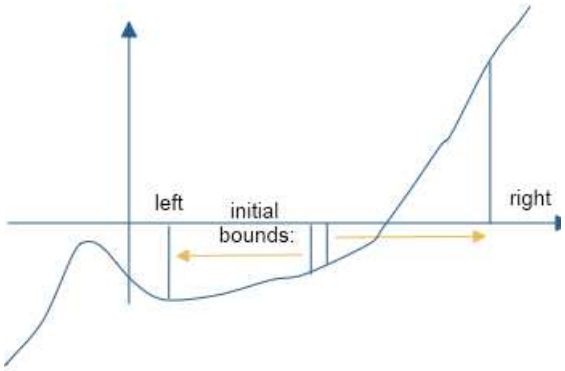


图 3.2: 设置初始搜索点

练习 3.5. 使以下代码正常工作:

```
// /findzeroarray.cppif( not is_odd(coefficients) ) {cout <<" 本程序
仅适用于奇次多项式 \n";exit(1);}
```

你可以按如下方式测试上述代码:

```
// /testzeroarray.cpp多项式 second{2,0,1}; //
2x^2 + 1要求(非为_奇(second) );多项式
third{3,2,0,1}; // 3x^3 + 2x^2 + 1要求(为_奇
(third) );
```

现在我们可以找到 x_-,x_+: 从某个区间开始，将端点向外移动，直到函数值符号相反。

练习 3.6. 编写一个函数 *find_initial_bounds*，用于计算 x_-,x_+ 使得

$$f(x_-) < 0 < f(x_+) \quad \text{or} \quad f(x_+) < 0 < f(x_-)$$

解决以下问题:

1. 该函数的理想原型是什么? 2. 如何将点移动足够远以满足此条件? 3. 能否更简洁地计算上述测试?

由于对于偶数次多项式，找到左右两点之间存在零点并不总是可能，我们完全拒绝这种情况。在以下测试中，我们为偶数次多项式抛出异常（参见教材第 23.2.2 节，特别是 23.2.2.3 节（教材））:

```
// /testzeroarray.cppright = left+1;多项式 second{2,0,1}; //
2x^2 + 1要求_抛出( 查找_初始_边界(second,left,right));
```

```
多项式三次{3,2,0,1}; // 3x^3 + 2x^2 + 1要求_不抛出( 查找_
初始_边界(三次,左,右));要求( 左<右);
```

确保你的代码通过这些测试。对于函数值，你需要添加什么测试?

3.1.4 求根法

寻根过程的全局视图如下:

- 你从函数符号相反的点 x_-,x_+ 开始；由此可知它们之间存在一个零点。
- 二分法寻找该零点时，会考察中点位置，并根据中点处的函数值:
- 将其中一个边界移动到中点，使得函数在左右搜索点再次具有相反的符号。

代码结构如下:

```
double find_zero( /*something */ ) {while ( /*左
右边界相距过远 */) {// 将左右边界向中间靠拢 }return
something;}
```

我们再次单独测试所有功能。在此情况下，这意味着移动边界应是一个可测试的步骤。

Exercise 3.7. Write a function *move_bounds_closer* and test it.

```
void移动边界靠近_      _(std::vector<
double> coefficients,double&
left,double& right );
```

对此函数实施一些单元测试。

最终，我们将所有内容整合到顶层函数 *find_zero* 中。

练习 3.8. 使此调用生效:

```
// /findzeroarray.cppauto zero= find_zero( coefficients, 1.e-8 );
cout << "找到根"<< zero<< "其值为"<< evaluate_at(coefficients,zero)
<< '\n';
```

设计单元测试（包括对达到精度的测试），并确保你的代码通过它们。

3.1.5 对象实现

重新审视章节 3.1.1 的练习，并引入一个多项式 类来存储多项式系数。现在，多个函数成为该类的成员。

同时更新单元测试。

一些进一步的建议：

1. 你能让你的多项式类看起来像一个函数吗?

```
class Polynomial {
  /* ... */
}
main () {
  Polynomial p;
  float y = p(x);
}
```

   参见章节 **??**。
2. Can you generalize the polynomial class, for instance to the case of special forms such as $(1+x)^n$?
3. Templatize your polynomials: see next subsection.

3.1.6 模板化

在目前的实现中，我们使用了 `double` 作为数值类型。请制作一个模板化版本，使其同时适用于 `float` 和 `double`。

你能看出这两种类型在可达到的精度上有何差异吗?

3.2 牛顿法

本节我们将探讨牛顿法。这是一种用于寻找函数 f 零点的迭代方法，即它计算一系列值 $\{x_n\}_n$，使得 $f(x_n) \to 0$。该序列由以下公式定义

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

其中 x_0 可任意选择。详情请参阅 HPC 书籍 [6]，第 23 节 .

虽然在实际应用中牛顿法用于复杂函数，但这里我们将看一个简单的例子，这实际上与计算历史有关。早期计算机没有计算平方根的硬件，而是使用牛顿法来实现。

假设你有一个正值 y，想要计算 $x = \sqrt{y}$。这相当于寻找以下函数的零点

$$f(x) = x^2 - y$$

其中 y 是固定的。为了表示这种对 y 的依赖关系，我们将写成 $f_y(x)$。牛顿方法随后通过迭代计算来寻找零点

$$x_{\text{next}} = x - f_y(x)/f'_y(x)$$

直到猜测足够精确，即直到 $f_y(x) \approx 0$。

我们不会深入讨论如何为迭代选择复杂的停止测试；这是数值分析课程的内容，而非编程课程。

3.2.1 函数实现

编写这个特定案例的代码当然很简单，大约需要 10 行代码。然而，我们希望有一个通用的代码，可以接受任意两个函数 f, f'，然后使用牛顿法找到我们特定函数 f 的零点。

练习 3.9.

- 编写函数 f(x,y) 和 deriv(x,y)，用于计算 $f_y(x)$ 和 $f'_y(x)$，根据上述 f_y 的定义。
- 读取一个值 y 并迭代直到 $|f(x,y)| < 10^{-5}$。打印 x。
- 第二部分：编写一个函数 newton_root 用于计算 $\sqrt{y}$。

3.2.2 使用 lambda 表达式

上文已编写符合以下规范的函数：

```
// /newton-fun.cpp
double f(double x);
double fprime(double x);
```

及算法实现：

```
// /newton-fun.cppdouble x{1.};while (true) {  auto fx=
f(x);  cout <<"f( " <<  x <<  " ) = " = "<<  fx <<  ’ \n’ ;  if
(std::abs(fx) <1 < 1.e-10) break;  x =  = x - fx/fprime(x);}
```

Exercise 3.10. Rewrite your code to use lambda functions for `f` and `fprime`.

你可以基于代码仓库中的 *newton.cpp* 文件进行开发

接下来，我们通过编写一个通用函数 `newton_root` 来实现代码模块化，该函数包含前一个练习中的牛顿方法。由于它需要适用于任何函数 f, f'，你必须将目标函数及其导数作为参数传入：

```
double root = newton_root( f,fprime );
```

Exercise 3.11. 重写上述牛顿练习以使用一个函数 used as:

```
double root = newton_root( f,fprime );
```

调用该函数

1. 首先使用您已创建的 lambda 变量；2. 但采用更优方案，直接以 lambda 表达式作为参数传递，即无需将其赋值给变量。

接下来我们扩展功能，但并非通过修改求根函数实现：而是采用更通用的方式来指定目标函数及其导数。

练习 3.12. 扩展牛顿法练习以循环计算根：

```
// /newton-lambda.cpp
  for (int n=2; n<=8; ++n) {
    cout << "sqrt(" << n << ") = "
         << newton_root(
    /* ... */
                        )
         << '\n';
```

若不使用 lambda 表达式，你需要定义一个函数

```
double 平方差_减去_n( double x,int n) { 返回
x*x-n;}
```

然而，`newton_root` 函数仅接受一个实数参数的函数。通过捕获使 `f` 依赖于整数参数。

Exercise 3.13. 你不需要将梯度作为显式函数：可以将其近似为

$$f'(x) = \big(f(x+h) - f(x)\big)/h$$

对于某个 h 的值。

编写一个仅接受目标函数的求根函数版本：

```
double newton_root( function< double(double)> f )
```

你可以使用固定值 `h=1e-6`。

不要重新实现整个牛顿方法：而是为梯度创建一个 lambda 并将其传递给你之前编写的函数 `newton_root`。

练习 3.14. 附加题：你能通过牛顿方法来计算对数吗?

3.2.3 模板实现

牛顿方法对复数同样有效，如同对实数一样。

练习 3.15. 重写你的牛顿程序，使其适用于复数：

// /newton-complex.cpp 复数 <double> z{.5,.5}; while (true) { auto fz= f(z); cout <<"f("<< z << ") "= "<< fz << ' \n' ; if (std::abs(fz)<1.e-10) break; z= z - fz/fprime(z); }

你可能会遇到一个问题，即无法直接在复数与 **float** 或 **double** 之间进行操作。使用 **static_cast**；参见章节 **??**。

那么你是否需要编写两个独立的实现，一个用于实数，另一个用于复数？（或许你还需要为 `float` 和 `double` 分别编写两个独立的实现！）
这正是模板派上用场的地方；参见教材第 22 章。

你可以将牛顿函数及其导数模板化：

```
// /newton-double.cpp模板<
typename T>T f(T x) { return
x*x- 2; };模板 <typename T>T
fprime(T x) { return 2 *x; };
```

然后编写

```
// /newton-double.cpp
  double x{1.};
  while ( true ) {
    auto fx = f<double>(x);
    cout << "f( " << x << " ) = " << fx << '\n';
    if (std::abs(fx)<1.e-10 ) break;
    x = x - fx/fprime<double>(x);
  }
```

练习 3.16. 使用模板更新你的牛顿法程序。如果它已能处理 `double`，尝试改用 `complex<double>`。它能正常运行吗？

练习 3.17. 用你的复数牛顿法计算 $\sqrt{2}$。它能正常运行吗？$\sqrt{}$

How about -2?

练习 3.18. 你能将使用了 lambda 表达式的牛顿法代码模板化吗？此时函数头应为：

```
// /lambda-complex.cpp
template<typename T>
T newton root_
    ( 函数 < T(T) > f, 函数 <
    T(T) >fprime, T 初始值 ) {
```

例如，您会这样计算 $\sqrt{2}$：

```
// /lambda-complex.cpp
  cout << "sqrt -2 = " <<
       newton_root<complex<double>>
            ( [] (complex<double> x) {
                 return x*x + static_cast<complex<double>>(2); },
              [] (complex<double> x) {
                return x * static_cast<complex<double>>(2); },
              complex<double>{.1,.1}
```

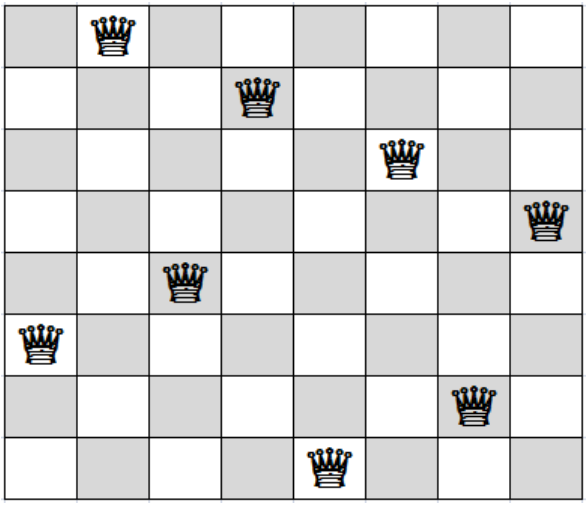
3. 零点求解

```
                )
        << '\n';
```

第 4 章

八皇后问题

递归编程中一个著名的练习是八皇后问题：能否按照国际象棋规则，在一个棋盘上放置八个皇后，使得任意两个皇后之间不会互相 ' 威胁 '?



4.1 问题陈述

' 八皇后问题 ' 的准确表述是：

- 在一个 8×8 棋盘上放置八个棋子，且任意两个棋子不在同一方格；同时满足
- 任意两个棋子不在同一行，
- 任意两个棋子不在同一列，且
- 任意两个棋子不在同一对角线上。

系统化的解决步骤如下：

1. 在第一行任意位置放置一个棋子；2. 针对第一行的每个选择，尝试第二行的所有可能位置；3. 针对前两行的所有选择组合，尝试第三行的所有位置；4. 当所有八行都放置棋子后，评估棋盘是否满足条件。

练习 4.1. 该算法将生成所有 8_8 种棋盘布局。你能想到至少一种加速搜索的方法吗?

由于数字 8 目前是固定的，你可以将此代码编写为八层嵌套循环。然而这种方式并不优雅。例如，上述阐述中数字 8 的唯一原因是这是国际象棋棋盘的传统尺寸。该问题更抽象地表述为在 $n \times n$ 棋盘上放置 n 个皇后，对于 $n \geq 4$ 存在解。

4.2 解决八皇后问题的基本方法

此问题要求你了解数组 / 向量；参见教材第 10 章。此外，关于测试驱动开发 (TDD)，参见教材第 68 章。最后，关于 std::optional 的使用，参见教材第 *24.6.2* 节。

基本策略是：通过每次指定下一皇后占据的列，逐步填充连续的行。采用面向对象 (OO) 策略，我们创建一个类 `ChessBoard`，用于存储部分填充的棋盘。

The basic solution strategy is recursive:

- 设`current`为当前部分填充的棋盘；
- 调用 `current.place_queens()` 尝试完成棋盘；

然而，使用递归时，此方法仅填充一行，然后在新棋盘上调用 `place_queens` 。

所以棋盘 : : 放置 _ 皇后 () {// 对于列 c = 1 ... 列数: // 创建棋盘副本 // 在副本的下一行第 c 列放置皇后 // 并调用放置 _ 皇后 () 于该副本; // 分析结果 .....}

该例程返回一个解，或指示无解可能。 .

在下一节中，我们将以 TDD 方式系统地开发一个解决方案。

4.3 采用 TDD 方法开发解决方案

我们现在逐步采用测试驱动开发的方法，为八皇后问题构建面向对象的解决方案。

The board 我们首先构建一个棋盘，其构造函数仅需指定问题规模：

```
// /queens.hpp
  ChessBoard(int n);
```

这是一个大小为 $n \times n$ 的 ‘广义棋盘’，初始状态应为空。

Exercise 4.2. 编写该构造函数，用于创建大小为 $n \times n$ 的空棋盘。

请注意棋盘的具体实现完全由您决定。后续将提供需要满足的功能测试用例，但任何能通过这些测试的实现都是正确解。

Bookkeeping: what's the next row? 假设我们逐行填充棋盘，需要一个辅助函数返回待填充的下一行：

```
// /queens.hpp
  int next_row_to_be_filled()
```

这为我们提供了第一个简单的测试：在空棋盘上，待填充的行是第零行。

Exercise 4.3. Write this method and make sure that it passes the test for an empty board.

```
// /queentest.cppTEST_CASE("空棋盘","[1]" )
{constexprint n=10;ChessBoard
empty(n);REQUIRE(empty.next_行_待_填充_位置
()==0 );}
```

根据 TDD 的规则，你可以实际编写该方法，使其仅满足空棋盘的测试。稍后，我们将测试该方法在填充几行后是否给出正确结果，那时你的实现当然需要具有通用性。

放置一个皇后 接下来，我们有一个函数来放置下一个皇后，无论这是否会给出一个可行的棋盘（意味着没有棋子可以互相捕获）：

```
// /queens.hpp
  void place_next_queen_at_column(int i);
```

该方法首先应捕获错误的索引：我们假设放置例程会对无效列号抛出异常。

```
ChessBoard::place_next_queen_at_column( intc )
{if ( /*c is outside the board */ )throw(1); //或
其他异常处理
```

（假设您未测试错误的索引。您能否在任何规模下构造一个简单的 ‘作弊’ 解决方案？）

Exercise4.4. 编写此方法，并确保它通过以下针对有效和无效列号的测试：

```
// /queentest.cpp
  REQUIRE_抛出（ empty.放置_下一个_皇后_于_列(-1));REQUIRE_抛
  出（ empty.放置_下一个_皇后_于_列(n));REQUIRE_不抛出（
  empty.放置_下一个_皇后_于_列(0));REQUIRE(empty.下一个_待填充
  _行_为_()==1 );
```

(从现在起我们只展示测试的主体部分。)

现在是时候开始编写一些真正重要的内容了。

Is a (partial) board feasible? 如果你有一个棋盘，即使是部分完成的，你会想要测试它是否可行，这意味着已经放置的皇后不能互相攻击。

该方法的原型为：

```
// /queens.hpp
  bool feasible()
```

该测试需从简单案例开始验证：空棋盘是可行的，仅放置一枚棋子的棋盘同样可行。

```
// /queentest.cpp棋盘空置 (n);
要求 ( 空置 . 可行 () );

// / ueentest.c pp 棋盘一 = 空 ; 一 . 放置 _ 下一
  个 _ 皇后 _ 于 _ 列 (0); 要求 ( 一 . 下一个 _ 行 _ 应 _
  为 _ 已填充 ()==1 ); 要求 ( 一 . 可行 () );
```

Exercise 4.5. Write the method and make sure it passes these tests.

我们不应仅进行成功测试（有时称为代码的 ‘快乐路径’ ）。例如，若将两个皇后置于同一列，测试应当失败。

练习 4.6. 基于初始尝试 —— 在位置 (0,0) 放置皇后后，在下一行的第零列添加另一个皇后。验证其是否通过测试：

```
// /queentest.cppChessBoard collide= one;//
在 ‘冲突’ 位置放置皇后 collide.place_next_
queen_at_column(0);// 并测试此操作不可行
REQUIRE(not collide.feasible() );
```

自行添加更多测试用例。（提交脚本不会运行这些测试，但它们可能对你有用。）

测试配置 若要测试非平凡配置的可行性，能够 ‘创建’ 解决方案是个好主意。为此我们需要第二种构造函数，通过棋子的位置构造一个完全填满的棋盘。

```
// /queens.hpp
  棋盘 ( 整型 n, 向量 < 整型 > 列 ); 棋盘 ( 向量 <
  整型 > 列 );
```

- 若构造函数仅传入一个向量，则描述一个完整棋盘。
- 添加一个整数参数表示棋盘大小，向量仅描述已填充的行。

练习 4.7. 编写这些构造函数，并测试显式给出的解是否为可行棋盘：

```
// /queentest.cpp棋盘五 (
{0,3,1,4,2} );要求 ( 五 . 可行 () );
```

关于实现这一点的优雅方法，请参阅委托构造函数；教材第 9.4.1 节。

最终我们必须编写复杂的部分。

4.4 递归求解方法

主函数

```
// /queens.hpp
  可选 < 棋盘 >  放置 _ 皇后 ()
```

接收一个棋盘（无论是否为空），并尝试填充剩余行。

一个问题在于，该方法需要能够传达：给定某些初始配置时无解的可能性。为此，我们将 `place_queens` 的返回类型设为 `optional<ChessBoard>`：

- 如果能够完成当前棋盘并得到一个解，则返回填充后的棋盘；
- 否则返回 `{}`，表示无解。

根据 4.2 节讨论的递归策略，该放置方法大致具有以下结构：

```
place_queens() {for( int col=0; col<n; col++ )
{ChessBoard next =*this;// 在 'next' 棋盘的 col 列放置皇后 // 若
可行且填满，则得到解 // 若可行但未填满，则递归 }}
```

该行

```
棋盘下一步 = * 此 ;
```

复制当前所在的对象。

Remark 1 *Another approach would be to make a recursive function*

```
boolplace_queen( const ChessBoard& current, ChessBoard &next );//
返回 true 表示可行，false 表示不可行
```

The final step 之前你已经编写了方法 `feasible` 来测试棋盘是否仍是解的候选。由于此例程适用于任何部分填充的棋盘，你还需要一个方法来测试是否已完成。

练习 4.8. 编写一个方法

```
bool filled();
```

并为其编写测试，包括正向和负向测试。

既然你能识别解决方案，现在就该编写解决方案例程了。

Exercise4.9. 编写该方法

```
// /queens.hpp
  optional<ChessBoard> place_queens()
```

由于函数 *place_queens* 是递归的，整体测试起来有些困难 .

我们从更简单的测试开始：如果你几乎已经得到解决方案，它可以完成最后一步。

练习 4.10. 使用构造器 r

```
棋盘 ( intn,vector<int> cols )
```

生成一个除最后一行外已填满且仍可行的棋盘。测试你是否能找到解决方案：

```
// /queentest.cpp棋盘几乎 ( 4, {1,3,0} );
自动 解决方案 = 几乎。放置 _ 皇后 (); 要求 ( 解决
方案 . 拥有 _ 值 () ); 要求 ( 解决方案 -> 已填充
() );
```

由于此测试仅填充最后一行，因此仅执行一次循环，因此可以打印诊断信息，而不会被大量输出淹没。

Solutions and non-solutions 现在您有了解决方案例程，测试它从空棋盘开始工作。例如，确认没有 3×3 解决方案：

```
// /queentest.cpp
TEST_CASE( "no 3x3 solutions","[9]" ) {
  ChessBoard three(3);
  auto solution = three.place_queens();
  REQUIRE( not solution.has_value() );
}
```

另一方面， 4×4 解决方案确实存在：

```
// /queentest.cppTEST_CASE( "存在 4x4 的解决方案",
"[10]") {ChessBoard four(4);auto solution=
four.place_queens();REQUIRE(solution.has_
value() );}
```

练习 4.11. （可选）你能修改代码使其统计所有可能的解吗？

练习 4.12. （可选）求解时间如何随 n 变化？

第二部分

研究项目

第 5 章

传染病传播模拟

本节包含一系列逐步构建的练习，最终将形成一个较为真实的传染病传播模拟。

5.1 模型设计

虽然可以通过统计方法模拟疾病传播，但此处我们将构建一个显式模拟：明确维护人群中所有个体的描述，并追踪每个人的状态。

我们将采用一个简单模型，其中个体的状态可分为：

- sick （患病者）：当他们患病时，会传染给其他人；
- susceptible （易感者）：他们健康但可能被感染；
- recovered （康复者）：他们曾患病但已不携带病原体，且不会被二次感染；
- vaccinated （接种者）：他们健康、不携带病原体且不会被感染。

在更复杂的模型中，个体可能仅在患病期间部分时间具有传染性，或可能存在其他疾病的继发感染等情况。此处我们简化模型：任何患病者在患病期间均能传染他人。

在后续练习中，我们将逐步构建疾病从初始传染源扩散的模型。程序将逐日追踪人群状态，无限运行直至无人患病。由于不存在重复感染，模拟总会终止。后续我们会加入变异机制以延长疫情持续时间。

5.1.1 其他建模方式

与在代码中逐人追踪的 ' 接触网络 ' 模型不同，可采用常微分方程（ ODE ）方法进行疾病建模。此时只需用单个标量表示感染人群的百分比，并推导该标量与其他标量之间的关系 [2, 9]。

这被称为 ' 隔室模型 '，其中 SIR 三状态（易感者、感染者、移除者）各自构成一个隔室：即人群的一个子集。接触网络模型与隔室模型都反映了部分事实。实际上二者可以结合，例如将国家视为多个城市的集合，

5. 传染病传播模拟

其中人员可在任意城市间流动。我们在城市内部采用分室模型，城市之间则通过接触网络连接。

本项目仅使用网络模型。

5.2 编码实现

以下章节将逐步开发本项目代码，后续将讨论如何将其作为科学实验运行。

Remark2 多处需要使用随机数生成器。可采用 *C* 语言随机数生成器（参见教材第 *24.7.5* 节），或使用新版标准模板库（*STL*）中的生成器（教材第 *24.7* 节）。

5.2.1 人员基础信息

疾病模拟最基本的组件是让人感染疾病，并观察该感染的时间发展过程。因此你需要一个人员类和一个疾病类。

在开始编码前，先思考这些类需要支持哪些行为 .

- 人员类。人员的基本方法包括：1. 被感染；2. 接种疫苗；3. 时间推进一天。此外，你可能需要查询人员状态：健康、患病还是已康复?

- 疾病类。目前疾病本身不需要太多功能（后续项目中你可能需要为其添加变异方法）。但你可能需要查询某些属性：

  1. 传播概率；2. 人员感染后的患病天数。

针对单个人的测试可能产生如下输出：

第 10 天，*Joe* 处于易感状态 第 11 天，*Joe* 处于易感状态 第 12 天，*Joe* 处于易感状态 第 13 天，*Joe* 处于易感状态 第 14 天，*Joe* 患病 (剩余 5 天) 第 15 天，*Joe* 患病 (剩余 4 天) 第 16 天，*Joe* 患病 (剩余 3 天) 第 17 天，*Joe* 患病 (剩余 2 天) 第 18 天，*Joe* 患病 (剩余 1 天) 第 19 天，*Joe* 康复

练习 5.1. 编写一个 `Person` 类，包含以下方法：

- `status_string()`：以字符串形式返回该对象的状态描述 `string`;
- `one_more_day()`：将该对象的状态更新至下一天;
- `infect(s)`：用疾病对象感染该对象

```
Disease s(n);
```

is specified to run for n days.

Your main program could for instance look like:

```
// /person.cpp
  for ( int step = 1; ; ++step ) {

    joe.one_more_day();
    /* ... */
    cout << "On day " << step << ", Joe is "
         << joe.status_string() << '\n';
    if (joe.is_recovered())
      break;
  }
```

其中感染部分已被省略。

您主要关注的是如何模拟一个人的内部状态。这实际上是两个独立的问题：

1. 状态，以及 2. 如果生病，需要多少天才能康复。

您可以找到用单个整数实现这一点的方法，但使用两个更好。此外，编写足够的支持方法，例如 `is_recovered` 测试。

5.2.1.1 人员测试

编写看似正确但在所有情况下行为都不正确的代码很容易。因此，对代码进行一些系统性测试是个好主意。

Make sure your `Person` objects pass these tests:

- 感染一种 100% 可传播的疾病后，他们应登记为患病状态。
- 若已接种疫苗或康复，且接触此类疾病，则保持原有状态不变。

- 若某疾病传播概率为 50%，且有多人接触，约半数人应会患病。此项测试编写起来可能稍有难度。

C你能使用 *Catch2* 单元测试框架吗？参见 63.3 节（教材）。

5.2.2 交互

接下来我们模拟人与人之间的互动：其中一人健康，另一人感染，当两者接触时疾病可能传播。

```
// /interaction.cpp人员感染，健
康；已感染．感染（流感）;/* ...
*/健康．接触（已感染）;
```

疾病传播存在特定概率，因此需要指定该概率。可将声明设为：

```
流感疾病 ( 5, 0.3 );
```

where the first parameter is the number of days an infection lasts, and the second the transfer probability.

练习 5.2. 为 `Disease` 类添加传播概率，并为 `Person` 类添加一个 `touch` 方法。设计并运行一些测试。

Exercise 5.3. Bonus: can you get the following disease specification to work?

```
// /interaction.cpp流感疾病 ;
flu.duration()  = 20;flu.transfer_
probability() =p;
```

为什么你可以认为这比之前建议的语法更好?

5.2.2.1 交互测试

调整上述测试，但现在是一个人接触感染者，而非直接接触疾病。

5.2.3 群体

接下来我们需要一个群体类，其中群体包含一个向量，由人对象组成。最初我们只感染一个人，且疾病不会传播。

The `Population` class should at least have the following methods:

- 随机 _ 感染，以初始设定一个被感染的群体部分；
- 随机接种，以初始设定一定数量的已接种个体。_
- 计数函数统计 _ 感染者和统计 _ 已接种者。

要运行一个真实的模拟，你还需要一个 `one_more_day` 方法，该方法将人口带过一天。这是你代码的核心，我们将在下一节逐步开发它。

5.2.3.1 人口测试

大多数人口测试将在下一节完成。现在，请确保你通过以下测试：

- 当疫苗接种率为 100% 时，每个人都确实应该接种疫苗。

5.3 疫情模拟

为了模拟疾病在人口中的传播，我们需要一个更新方法，该方法将人口推进一天：

- 患病者会与一定数量的其他民众接触；
- 且所有人的年龄都会增长一天，主要意味着患病者距离康复又近了一天。

我们将分几个步骤展开这一过程。

5.3.1 无接触

首先假设人群之间无接触，因此疾病仅止于最初感染的人群。

追踪输出应类似于：

```
Size of
population?
In step   1 #sick:    1 :  ? ? ? ? ? ? ? ? ? + ? ? ? ? ? ? ? ?
In step   2 #sick:    1 :  ? ? ? ? ? ? ? ? ? + ? ? ? ? ? ? ? ?
In step   3 #sick:    1 :  ? ? ? ? ? ? ? ? ? + ? ? ? ? ? ? ? ?
In step   4 #sick:    1 :  ? ? ? ? ? ? ? ? ? + ? ? ? ? ? ? ? ?
In step   5 #sick:    1 :  ? ? ? ? ? ? ? ? ? + ? ? ? ? ? ? ? ?
In step   6 #sick:    0 :  ? ? ? ? ? ? ? ? ? - ? ? ? ? ? ? ? ?
Disease ran its course by step 6
```

备注 3 此类显示有助于验证程序行为的合理性。若在报告中包含此类显示，请务必使用等宽字体，并避免使用需要换行的大规模人口数据。在后续测试中，应使用大规模人口，但无需包含这些显示内容。

练习 5.4. 编写一个无感染人群的程序。

- 编写 *Population* 类。构造函数接收人数参数：

  人口 *population*(*npeople*)；

- 编写一个方法来随机感染若干人 le:

  ```
  // /pandemic.cpp
  人口．随机_感染（发热，初始_感染）；
  ```

- 编写一个 *count_infected* 方法用于统计被感染人数。
- 编写一个 *one_more_day* 方法来更新群体中所有人的状态。
- 循环执行 *one_more_day* 方法直到无人被感染：*Population*::*one_more_day* 方法应对种群中所有人应用 *Person*::*one_more_day*。

编写一个例程来显示群体状态，例如：？表示易感者，+ 表示感染者，- 表示康复者。

5.3.1.1 测试

测试在疾病持续期间，感染人数保持恒定，且健康人数与感染人数之和等于群体总数量。

5.3.2 传染

之前的练习过于简单化：最初的零号病人是唯一患病的人。现在让我们引入传染机制，研究疾病从单个感染者开始的传播过程。

我们从一个非常简单的感染模型开始。

练习 5.5. 编写一个模拟程序，其中每一步感染者可直接传染其相邻个体。

使用不同人口规模和传染概率运行多次模拟。是否存在人群完全免疫的情况?

5.3.2.1 测试

执行基础测试验证：

- 若初始 1 人感染 $p = 1$ 疾病，次日应有 3 例新增 —— 除非感染者为首尾个体，则新增 2 例。
- 若人员 0 被感染，且 $p = 1$，则模拟运行天数应等于人口规模。
- 若 $p = 0.5$，前例将如何变化?

5.3.3 疫苗接种

练习 5.6. 引入疫苗接种：读取另一个代表已接种疫苗人口百分比的数值，并随机选择该群体成员。

Describe the effect 疫苗接种人群对疾病传播的影响。为何 his model unrealistic?

5.3.4 传播

为使模拟更贴近现实，我们让每位病患每天与固定数量的随机人群接触。这大致对应 *SIR* 模型；https://en.wikipedia.org/wiki/Epidemic_ 模型。

将每人每日接触人数设置为 6 左右（也可设为随机值的上限，但这不会实质改变模拟结果）。你已编程实现接触感染者后的患病概率。再次从单例感染者开始模拟。

练习 5.7. 编写随机交互代码。现在运行多组模拟，变量包括

- 人群接种疫苗的百分比，以及
- 疾病在接触时的传播几率。

记录疾病在人群中传播的时长。在固定接触次数和传播概率下，该数值如何随疫苗接种比例变化?

将此函数以表格或图表形式报告。确保有足够的数据点以得出有意义的结论。使用现实的人口规模。也可进行多次运行并报告平均值，以消除随机数生成器的影响。

练习 5.8. 研究 “ 群体免疫 ” 问题：若足够多的人接种疫苗，则部分未接种者仍不会患病。假设你希望未接种疫苗却永不生病的概率超过 95%。研究实现这一目标所需的疫苗接种比例，作为疾病传染性的函数。

与前一练习相同，请确保您的数据集足够大。

Remark4 您之前使用的屏幕输出适用于小规模问题的完整性检查。但对于实际模拟，您需要考虑现实的群体规模。若以大学校园作为人群模型，其中个体可能随机相遇，那么模拟该场景需要多大的群体规模？您所居住的城市规模又如何？

同理，若测试不同疫苗接种率时，您采用何种粒度？以 *5%* 或 *10%* 为增量时虽可将所有结果打印至屏幕，但可能遗漏细节。不必畏惧生成大量数据并直接输入绘图程序。

5.3.5 变异

新冠疫情年份已证明原始病毒变异的重要性。接下来，您可在项目中引入变异模块。我们按如下方式建模：

- 每隔一定数量的传播，病毒就会变异成一种新变体。
- 从一种变体中康复的人仍可能感染其他变体。
- 为简化模型，假设每种变体导致患者生病的天数相同，且
- 疫苗接种是全有或全无的：一剂疫苗足以抵御所有变体；
- 另一方面，从一种变体中康复并不能对其他变体产生免疫力。

从实现角度而言，我们按如下方式建模。首先需要一个 `Disease` 类，以便能用显式病毒感染个体；

```
// /infect_lib.hpp
void接触( 常量 人员 &, 长整型 参数 =0 );void 感染
( 常量 疾病 & );
```

一个 `Disease` 对象现在携带了诸如传播概率或一个人保持不适状态时长等信息。模拟变异有点棘手。您可以按如下方式操作：

- 有一个全局的 `variants` 计数器用于记录新病毒变种，以及一个全局的 `transmissions` 计数器。
- 每当一个人感染另一个人时，新被感染者会获得一个新的 `Disease` 对象，包含当前变种信息，同时 transmissions 计数器会更新。
- 存在一个参数决定经过多少次传播后疾病会发生变异。若发生变异，全局的 `variants` 计数器会更新，此后所有感染都将基于新变种。（注：此模型并不十分真实。您可以自由设计更好的模型。）
- 每个 `Person` 对象都有一个变异体向量，表示他们从中康复；从一种变异体康复仅使他们对该特定变异体免疫，而非其他。

练习 5.9. 向你的模型添加突变。试验不同的突变率：随着突变率增加，疾病应在人群中留存更长时间。你之前观察到的与疫苗接种率的关系是否发生变化？

5.3.6 无疫苗疾病：埃博拉与新冠

本节为可选内容，完成可获得加分

目前该项目适用于已有疫苗的疾病，例如麻疹、腮腺炎和风疹的 MMR 疫苗。若疾病尚无疫苗（如本文撰写时的埃博拉和新冠肺炎），分析方式则截然不同。

此时需在代码中引入 ‘社交距离’ 机制：人们不再随机接触他人，仅与极有限社交圈内成员互动。设计一个距离函数模型，并探索不同设置。

The difference between Ebola and Covid-19 is how long an infection can go unnoticed: the *incubation period*. With Ebola, infection is almost immediately apparent, so such people are removed from the general population and treated in a hospital. For Covid-19, a person can be infected, and infect others, for a number of days before they are sequestered from the population.

将该参数加入模拟，探索疾病行为随其变化的规律。

5.4 伦理

传染病与疫苗接种议题充满伦理拷问。核心问题在于小概率事件难以降临我身，为何不能稍越规则？。这种逻辑最常出现在疫苗接种场景，人们因各种理由拒绝接种。

深入探讨这个问题以及其他你可能想到的问题：显然，如果每个人都违反规则将带来灾难性后果，但如果只有少数人这么做呢?

5.5 附加内容：测试

阅读 https://sinews.siam.org/Details-Page/the-mathematics-of-mass-testing-for-cov 你能验证该文章提出的假设吗?

5.6 项目报告与提交

5.6.1 程序文件

在本项目中，你已编写了多个主程序，但部分代码在多个程序间共享。请按以下方式组织代码：每个主程序对应一个文件，并创建一个单独的 ‘库’ 文件存放类方法。

你有两种实现方式：

1. 创建一个 ‘库’ 文件，例如 infect_lib.cc，每个主程序中包含一行

   ```
   #include "infect_lib.cc"
   ```

   这不是最佳解决方案，但目前可以接受。
2. 更优方案要求使用分离编译 来构建程序，并需要一个头文件 。此时需分别编译 infect_lib.cc 库文件，并在库文件与主程序中包含 infect_lib.h 头文件：

```
#include "infect_lib.h"
```

详见 19.2.2 (教材) 以获取更多信息。

提交所有源代码文件，并附上如何构建所有主程序的说明。你可以将这些说明放在一个描述性名称的文件中，例如 README 或 INSTALL，或者使用一个 *makefile*。

5.6.2 报告撰写

在报告中，描述你进行的 ‘ 实验 ’ 以及从中得出的结论。上述练习为你提供了许多需要回答的问题。

对于每个主程序，请包含一些示例输出，但请注意这不能替代用完整句子写出你的结论。

Theexercisesin section49.3.4(教材)要求你探索程序行为作为一个或多个参数的函数。请用表格报告你发现的行为。你可以使用 Matlab 或 Python 中的 Matplotlib （甚至 Excel ）来绘制数据，但这不是必须的。

5.7 附加内容：数学分析

SIR 模型也可以通过耦合差分或微分方程进行建模。

1. 在时间 i 时，易感人群数量 S_i 会减少一个比例

$$S_{i+1} = S_i(1 - \lambda_i\, dt)$$

其中 λ_i 是感染人数与一个反映会面次数和疾病传染性的常数的乘积。我们写成：

$$S_{i+1} = S_i(1 - \lambda I_i\, dt)$$

2. 感染人数同样会增加 $\lambda S_i I_i$，但也会因康复（或死亡）而减少：

$$I_{i+1} = I_i(1 + \lambda S_i\, dt - \gamma\, dt).$$

3. 最后， ‘ 移除 ’ 人数等于最后一项：

$$R_{i+1} = R_i(1 + \gamma I_i).$$

Exercise 5.10. Code this scheme. What is the effect of varying dt?

练习 5.11. 要使疾病成为流行病，新增感染人数必须大于康复人数。即，

$$\lambda S_i I_i - \gamma I_i > 0 \Leftrightarrow S_i > \gamma/\lambda.$$

你能在模拟中观察到这一点吗?

参数 γ 有一个简单的解释。假设一个人患病 δ 天后康复。如果 I_t 相对稳定，意味着每天感染和康复的人数相同，因此每天有 $1/\delta$ 比例的人康复。因此，γ 是个人感染持续时间的倒数。

5. 传染病传播模拟

第 6 章

谷歌 PageRank 算法

6.1 基本概念

我们将模拟互联网的运行机制。具体而言，我们将模拟 *Pagerank* 算法 ——*Google* 正是通过该算法判定网页的重要性。

首先定义几个基础类：

- 一个 Page （页面）包含标题、谷歌数据中心的全局编号等信息，同时还包含一组链接集合。
- 我们用一个指向页面的指针来表示链接。理论上可以设计一个 Link 类，包含诸如点击概率或点击次数等更多信息，但目前仅用一个指针即可满足需求。
- 最终我们需要一个 Web 类来包含多个页面及其链接。这个 web 对象最终还将包含诸如页面相对重要性等信息。

这个应用场景天然适合使用指针。当你在网页上点击链接时，浏览器会从当前页面跳转到另一个页面。可以通过维护一个指向页面的指针来实现此功能，点击操作更新该指针的值即可。

练习 6.1. 创建一个 Page 类，初始仅包含页面名称。编写一个显示页面的方法。由于我们将频繁使用指针，以下代码用于测试验证：

```
// /web2.cpp
auto homepage= make_shared<Page>("My Home Page");
cout << "主页尚无链接 :"<< '\n';cout << homepage->as_
string() <<'\n';
```

接下来，为页面添加链接。链接是指向另一个页面的指针，由于可以有任意数量的链接，你需要一个向量来存储它们。编写一个名为 click 的方法来跟踪链接。预期代码如下：

```
// /web2.cpp
auto shared_Page<Page>(make_主页 "); homepage->add_link(utexas); auto
searchpage= make_shared<Page>("google"); homepage->add_
link(searchpage); cout << homepage->as_string() <<'\n';
```

练习 6.2. 在你的主页上添加更多链接。为 Page 类编写一个 random_click 方法。预期代码：

```
// /web2.cpp
  for (int iclick=0; iclick<20; ++iclick) {
    auto newpage = homepage->random_click();
    cout << "To: " << newpage->as_string() << '\n';
  }
```

你如何处理没有链接的页面情况?

6.2 随机点击模拟

练习 6.3. 现在创建一个 Web 类，其主要包含一组页面（技术上：一个 vector ）。更准确地说：包含指向页面的指针。由于我们不想手动构建整个互联网，让我们编写一个 create_random_links 方法，用于生成随机数量的指向随机页面的链接。预期代码：

```
// /web2.cpp 网络互联网（网络规模）; 互联网 . 创建
_ 随机 _ 链接（平均链接数）;
```

现在我们可以开始模拟了。编写一个方法 Web::random_walk，该方法接收一个页面和行走长度，模拟在当前页面上随机点击多次后的结果。（当前页面，而非起始页面。）

让我们开始着手 PageRank。首先看看是否存在比其他页面更受欢迎的页面。你可以通过在每页上启动一次随机行走来实现这一点，或者可能进行几次。

练习 6.4. 除了互联网的规模外，你的测试还有哪些其他设计参数？你能给出它们影响的粗略估计吗?

练习 6.5. 你的第一次模拟是在每个页面上启动若干次，并统计最终停留的位置。预期代码：

```
//web2.cpp vector<int> landing_counts( 互联网 . 数量 _ 的 _ 页面 (), 0);for (auto 页面 :
互联网 . 所有 _ 页面 ()) {  for (int 步数 =0; 步数 <5; ++ 步数 ) {  auto 终止页面 = 互联网 .
随机 _ 游走 ( 页面 , 2* 平均链接数 , 追踪 );  landing_counts.at( 终止页面 -> 全局 _ID())
++;  }}
```

展示结果并进行分析。你可能会发现自己在某些页面上停留次数过多。这是怎么回事？请修正这个问题。

6.3 图算法

有许多算法依赖于逐步遍历网络。例如，任何图都可以是连通的。测试方法如下：

- 任选一个顶点 v。建立一个 ‘可达集合’ $R \leftarrow \{v\}$。
- 现在观察从可达集合能到达哪些地方：

$$\forall_{v \in V} \forall_{w \text{ neighbour of } v} : R \leftarrow R \cup \{w\}$$

- 重复上一步骤，直到 R 不再发生变化。

该算法结束后， R 是否等于你的顶点集合？若是，则你的图被称为（完全）连通的。若不是，则你的图具有多个连通组件。

练习 6.6. 编写上述算法代码，记录到达每个顶点 w 所需的步数。这就是单源最短路径算法（针对无权图）。

直径定义为最大最短路径。编写代码实现此功能。

6.4 网页排名

PageRank 算法提出的问题是：如果持续随机点击，最终停留在特定页面的概率分布如何。我们通过概率分布来计算这一点：为每个页面分配一个概率，所有概率之和为 1。我们从随机分布开始：

代码：

```
// /web2.cpp
    概率分布

    随机_状态（互联网．编号_的_页                ges());
    随机_状态．设置_随机();
    cout <<"初始分布："                          <<
    random_state.as_string() << '\n';
```

输出 [谷歌] pdf 设置：

```
初始分布：
    0:0.00, 1:0.02, 2:0.07
    3:0.05, 4:0.06, 5:0.08,
    6:0.04, 7:0.04, 8:0.04,
    9:0.01, 10:0.07, 11:0.05,
    12:0.01, 13:0.04,
    14:0.08, 15:0.06,
    16:0.10, 17:0.06,
    18:0.11, 19:0.01
```

Exercise6.7. 实现一个类 ProbabilityDistribution，用于存储浮点数向量。编写以下方法：

- 访问特定元素，
- 将整个分布设置为随机，并且
- 进行归一化处理，使得概率之和为 1。
- 一个将分布渲染为字符串的方法可能也会很有用。

接下来我们需要一个方法，给定一个概率分布，给出对应于执行一次点击的新分布。（这与马尔可夫链有关；参见 HPC 书籍 [6]，章节 10.2.1。）

练习 6.8. 编写该方法

```
概率分布 Web::全局点击
    ( 概率分布 当前状态 );
```

测试方法

- 从一个仅在单一页面非零的分布开始;
- 打印对应一次点击的新分布;
- 对多个页面执行此操作并直观检查结果。

然后从一个随机分布开始并运行几次迭代。该过程收敛速度如何？将结果与上述随机游走练习进行比较。

练习 6.9. 在随机游走练习中，你必须处理某些页面没有出链的情况。那时你会跳转至随机页面。但全局点击方法中缺乏这一机制。请设法将其整合进来。

让我们模拟一些简单的 ‘搜索引擎优化’ 技巧。

练习 6.10. 添加一个你人为使其显得重要的页面：设置若干页面全部指向该页面，但这些页面自身不被任何页面链接。（由于随机点击，它们仍会被偶尔访问到。）

计算人为炒作页面的排名。你是否成功欺骗谷歌将此页面排名提升？你需要添加多少链接?

示例输出:

```
互联网拥有 5000 个页面 最高得分 : 109:0.0013, 3179:0.0012, 4655:0.0010, 3465:0.0009,
4298:0.0008 含虚假页面时 : 互联网拥有 5051 个页面 最高得分 : 109:0.0013, 3179:0.0012,
4655:0.0010, 5050:0.0010, 4298:0.0008 炒作页面得分位于 4
```

6.5 图与线性代数

概率分布本质上是一个向量。你也可以将网络表示为一个矩阵 W，其中 $w_{ij} = 1$ 如果页面 i 链接到页面 j。如何用这些术语解释 globalclick 方法?

练习 6.11. 添加 Web 对象的矩阵表示并重新实现全局点击方法。测试其正确性。

进行时间性能对比。

你上面为寻找稳定概率分布所做的迭代对应于线性代数中的 ‘幂方法’。查阅 Perron-Frobenius 理论，看看它对页面排名有何启示。

第 7 章

选区重划

本项目将探讨 ‘杰利蝾螈’ 现象 —— 通过策略性划分选区使少数群体获得多数选区席位 [1]。

7.1 基本概念

我们将涉及以下核心概念：

- 州级行政区被划分为既定的普查区域。根据普查数据（收入、族裔、年龄中位数），通常能较准确预测该选区的整体投票倾向。
- 国会选区数量是预先确定的，每个选区由人口普查区组成。国会选区并非随机划分：这些普查区必须在地理上相互接壤。
- 为适应人口变化，每隔几年会重新划定选区边界，这一过程称为选区重划（redistricting）。

选区重划具有较大自由度：通过调整（国会）选区边界，可以使整体处于少数地位的群体获得多数选区席位。这种做法被称为杰利蝾螈（*gerrymandering*）。

背景阅读资料请参阅 https://redistrictingonline.org/。

为进行小规模计算机模拟杰利蝾螈现象，我们需要做出一些简化假设。

- 首先，我们跳过人口普查区概念：假设选区直接由选民构成，且已知其政治倾向。实践中通常通过代理指标（如收入和教育水平）来预测选民倾向。
- 接着，我们假设一个一维状态。这足以构建能揭示问题本质的示例：考虑一个由五位选民组成的州，我们将他们的投票记为 AAABB。将其划分为三个（连续的）选区可以这样操作：AAA|B|B，这样会形成一个 ‘A’ 选区与两个 ‘B’ 选区。
- 我们也允许选区的规模可以是任意正值，只要选区的数量是固定的。

1. 本项目显然基于北美政治体系。希望此处的解释足够清晰。如果您知道其他采用类似体系的国家，请联系作者。

7.2 基本功能

7.2.1 选民

我们摒弃人口普查区域的概念，将所有内容以选民为基准进行表述，并假设选民的投票行为已知。因此，我们需要一个选民类（Voter class），用于记录选民 ID 及其党派归属。我们假设存在两个党派，并为未决定投票意向的选民保留选项。

练习 7.1. 实现一个选民类（Voter class）。例如，可用 ± 1 代表 A/B 党派，0 代表未决定。

```
// /linear.cppcout <<"选民 5 持肯定态度: "<< '
\n';Voter nr5(5,+1);cout <<nr5.print() << '
\n';


// /linear.cppcout <<"选民 6 为负面: "<< '\n';
选民编号 6(6,-1);cout << nr6.print() <<'\n';



// /linear.cpp
    cout <<"选民 7 号行为异常: "<< '\n'; 选民编
    号 7(7,3);cout << nr7.print() <<'\n';
```

7.2.2 群体

Exercise 7.2. 实现一个模拟选民群体的 District 类。

- 你可能希望从一个选民或一组选民向量创建选区。若能有一个接受字符串表示的构造函数会更方便。
- 编写方法 majority 以给出确切的多数或少数，以及 lean 来评估该选区总体上属于 A 党还是 B 党。
- 编写一个子方法来创建子集 .

  ```
  District District::sub(int first,int last);
  ```

- 为了调试和报告，拥有一个方法可能是个好主意

  ```
  string District::print();
  ```

代码：

```
1 ///linear.cpp
2     cout <<"创建包含一个 B 的选区
      votèr"; 选民 mr5,5,+1);
3
4     第九选区 ( nr5);t <<"
5     cou      ·· 大小:  "  <<  九. 大小 ()    <<
      '\n';
6     cout << ".. lean: " << nine.lean() <<
      '\n';
7     /* ... */
8     cout <<"Making district ABA" << <style id='7>' \n' ;
9     第九区 ( v        ector<选民 >
10                    {{1,-1},{2,+1},{3,-1}}
      } );
11    cout << ".. 大小:  " <<  九. 大小 ()    <<
      '\n';
12    cout << ".. lean: " << nine.lean() <<
      '\n';
```

输出
[gerry] 选区：

```
创建仅含一名 B 类选民的选区
..  规模： 1
..  倾向： 1

创建 ABA 型选区
..  大小： 3
..  精益： -1
```

练习 7.3. 实现一个 Population 类，最初用于模拟整个州的情况。

代码：

```
1 // /linear.cpp
2     string pns( "-++--" );
3     种群一些 (pns);
4        cout <<"从字符串生成种群 pns <<'\n'; " <<

5     cout << ".. size: " << 某些 . 尺寸 ()<<
      '\n';
6     cout << ".. lean:  " 一些 . 精简 ()    <<
      '\n';
7     种群组=some.sub(1,3);"b   li   13"
8     cout <<  su  popu at on  --   << '\n';
9     cout << ".. size: " << group.size()
      << '\n';
10    cout << ".. lean: " << group.lean()
      << '\n';
```

输出
[gerry] 群体：

```
来自 字符串的群体              -++--
.. 规模：5
..  倾斜： -1
子群体 1--            3
.. 规模：2
..  精益： 1
```

除了显式创建外，还需编写一个构造函数，用于指定人数及多数比例：

```
// /linear.cpp
  Population( int population_size,int majority,bool trace=false )
```

使用随机数生成器精确实现指定的多数比例。

7.2.3 选区划分

下一层级的复杂性在于处理一组选区。由于我们将逐步创建此结构，因此需要一些方法来扩展它。

练习 7.4. 编写一个存储地区对象向量的类 Districting。编写 size 和 lean 方法：

代码：

```
// /linear.cpp
    cout <<"制作单一选民
    人口 B"<< '\n';
    人口人数 (              vect 或 <选民>{
    选民(0,+1) } );t <<"
    cou        .. 规模:  " <<  人数 . 规模 ()
    << '\n';
    cout << ".. lean: " << people.lean()
    << '\n';

    选区划分 gerry;cout <<" 从
                          empty
    分区: "<< '\n';
    cout <<".. 选区数量 :                  " <<
    gerry.size() << '\n';
```

输出

```
[ 格里 ]  格里空区:

Making single voter population B
.. size: 1
..  精益 : 1tith
Star  w  em t  districting:
    py.. 选区数量 :0
```

Exercise 7.5. Write methods to extend a `Districting`:

```
// /linear.cpp
    cout <<" 添加一名 B 选民 :" << '\n'; gerry = gerry.extend_with_new_
    district( people.at(0) ); cout << ".. 选区数量 : "<< gerry.size() << '\n'; cout
    << ".. 倾向 : " << gerry.lean() <<'\n'; cout <<" 添加 A A:" << '\n'; gerry=
    gerry.extend_last_district( Voter(1,-1) ); gerry= gerry.extend_last_
    district( Voter(2,-1) ); cout << ".. 选区数量 : "<< gerry.size() << '\n'; cout
    << ".. 倾向 : " << gerry.lean() <<'\n';


    cout <<" 添加两个 B 选区 :" << '\n'; gerry= gerry.extend_with_new_
    district( Voter(3,+1) ); gerry= gerry.extend_with_new_district( Voter(4,+1) );
    cout <<".. 选区数量 : " << gerry.size() << '\n'; cout <<".. 倾向 : " <<
    gerry.lean() << '\n';
```

7.3 策略

现在我们需要一种方法来划分人口区域：

```
划分人口 :: 少数群体 _ 规则 ( int ndistricts );
```

与其生成所有可能的人口分区方案，我们采用一种渐进式方法（这与名为动态规划的解决策略相关）：

- 核心问题是如何将人口最优划分到 n 个区域；
- 我们通过递归实现这一点，首先解决子人口在 $n-1$ 个区域的划分问题，
- 并将其余人口作为一个区域扩展。

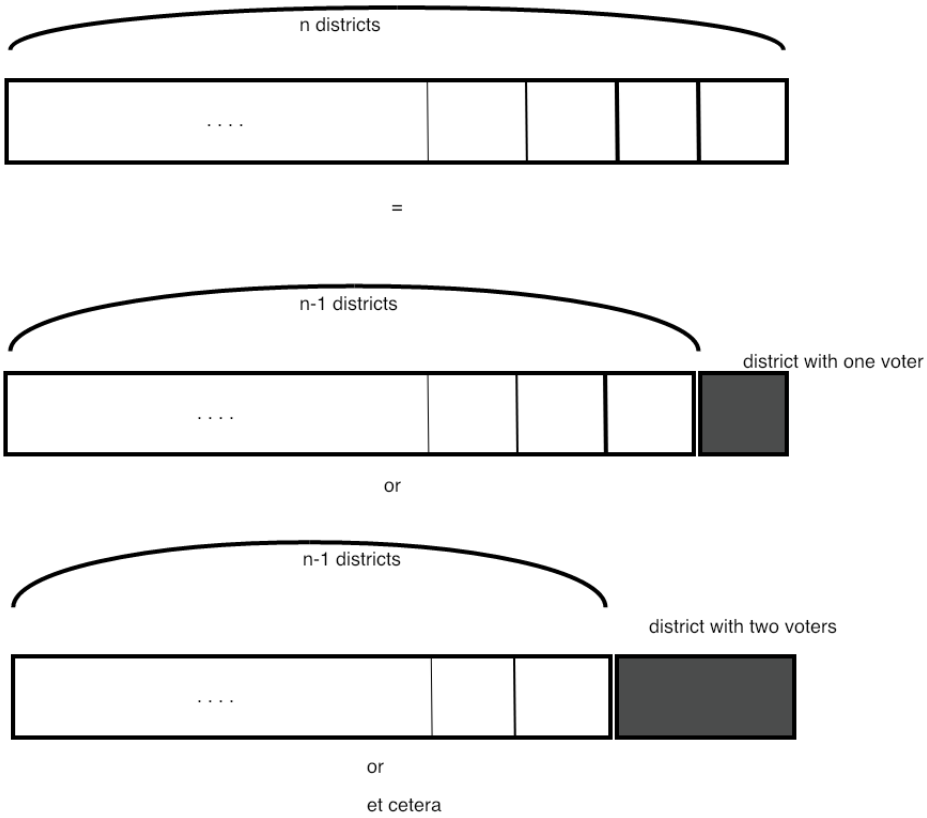


图 7.1：划分人口的多种方式

这意味着你需要考虑所有将 ‘剩余’ 人口划分到一个选区的方式，这表示你需要在递归之外对所有人口划分方式进行循环遍历；参见图 51.1 （教材）。

- 对于所有 $p = 0, \ldots n-1$ 考虑将州划分为 $0, \ldots, p-1$ 和 $p, \ldots, n-1$。
- 使用第一组的最佳选区划分方案，并将最后一组单独划为一个选区。
- 保留在所有 p 值中能产生最强少数派规则的选区划分方案。

现在您可以实现上述简单示例：

```
AAABB => AAA|B|B
```

Exercise 7.6. Implement the above scheme.

代码:

```
1 // /linear.cpp
2     Population five("+++--");
3     cout << "Redistricting population: "
      << '\n'
4          << five.print() << '\n';
5     cout << ".. majority rule: "
6          << five.rule() << '\n';
7     int ndistricts{3};
8     auto gerry =
      five.minority_rules(ndistricts);
9     cout << gerry.print() << '\n';
10    cout << ".. minority rule: "
11         << gerry.rule() << '\n';
```

Output

[gerry] district5:

```
Redistricting population:
[0:+,1:+,2:+,3:-,4:-,]
.. majority rule: 1
[3[0:+,1:+,2:+,],[3:-,],[4:-,],]
.. minority rule: -1
```

注意：上述给定的 p 范围并不完全准确：例如，初始部分的人口基数需要足够大以容纳 $n-1$ 选民。

练习 7.7. 测试多种人口规模；在仍给予政党 A 多数席位的同时，你能给予政党 B 多大的多数优势。

7.4 效率：动态规划

如果思考刚刚实现的算法，你可能会注意到初始部分的选区划分被重复计算多次。针对此问题的优化策略称为记忆化。

练习 7.8. 通过存储并复用初始子种群的结果来改进你的实现 .

在某种程度上，我们是逆向解决这个问题的：先考虑用最后若干选民组成一个选区，然后递归地解决前面若干选民的较小规模问题。但在这个过程中，我们确定了如何最优地将前 1 个、前 2 个、前 3 个选民等分配到选区。实际上，对于超过一个选民的情况，比如五个选民，我们找到了将这些五个选民分配到一个、两个、三个、四个选区时，可达到的最佳少数派规则的结果。这种正向计算 ‘最佳’ 选区划分的过程被称为动态规划。其基本假设是，你可以利用中间结果并扩展它们，而无需重新考虑之前的问题。

例如，假设你已经考虑了将十个选民划分到最多五个选区的情况。现在，十一个选民和五个选区的多数派是最小值

- 十位选民和五个选区，新增选民被分配到最后一个选区；或
- 十位选民和四个选区，新增选民独立成为一个新选区。

练习 7.9. 编写动态规划算法解决选区重划问题 .

7.5 扩展

当前项目基于多项简化假设。

- 国会选区需要大致规模相当。能否对规模比例设定上限？少数群体是否仍能获得多数席位？

练习 7.10. 最大的假设当然是我们考虑了一维状态。在二维情况下，你有更多自由度来塑造选区形状。实现一个二维方案；使用完全正方形的州，其中人口普查区形成规则网格。将国会选区的形状限制为凸形。

效率差距 是衡量一个州选区划分 ‘ 公平性 ’ 的指标。

练习 7.11. 查阅效率差距（及 ‘ 无效票 ’ ）的定义，并在代码中实现它。

7.6 伦理

选区重划活动的初衷是赋予民众公平的代表权。但在其畸变形式 —— 杰利蝾螈（ Gerrymandering ）中，这种公平理念被公然违背，因为其明确目标就是让少数派获得多数选票。探索如何消除这种不公现象。

在上述探索中，选民唯一特征是其对政党 A 或 B 的偏好。但实际上，选民可被视为社区组成部分。《投票权法案》关注 ' 少数族裔选票稀释 ' 问题。能否举例说明不考虑种族因素的选区划分会对某些社区造成负面影响?

7. 选区重划

第 8 章

亚马逊配送卡车调度

本节包含一系列逐步构建配送卡车调度模拟的练习。

8.1 问题描述

配送卡车路线调度是一个被深入研究的问题。例如，最小化卡车需行驶的总距离对应着旅行商问题 *(TSP)*。然而，在亚马逊配送卡车调度场景中，该问题出现了一些新特点：

- 客户被承诺了一个可进行配送的日期窗口。因此，卡车可将地点列表拆分为多个子列表，其总距离比一次性遍历整个列表更短。
- 除了 *Amazon prime* 客户需要保证次日送达的包裹。

8.2 基础编码实现

在尝试寻找最佳路线之前，我们先搭建基础框架以生成任意可行路线。

8.2.1 地址列表

你可能需要一个类 `Address` 来描述需要投递的房屋位置。

- 为简化起见，我们为房屋赋予 (i, j) 坐标。
- 我们可能需要一个距离函数来计算两个地址之间的距离。我们可以假设两个房屋之间可以直线通行，或者城市建立在网格上，从而应用所谓的曼哈顿距离。
- 地址可能还需要一个字段来记录最晚可能的交付日期。

练习 8.1. 编写一个 `Address` 类来实现上述功能，并进行测试。

8. 亚马逊配送卡车调度

代码：

```
// /route.cpp
    地址一（1.,1.),
      two(2.,2.);
    cerr << "Distance: "
         << 一. 距离（二）
         << '\n';
```

输出
[amazon] 地址：

```
地址
距离：1.41421
.. 地址
地址 1 应最接近
    仓库。 检查：1

从仓库到仓库的路线：
    (0,0) (2,0) (1,0) (3,0)
    (0,0)
 长度为 8: 8
贪婪调度：  (0,0)(1,0)
    (2,0) (3,0) (0,0)
 长度应为 6:6

Square5
按顺序移动： 24.1421
平方路线：(0,0) (0,5)
    (5,5) (5,0) (0,0)
 长度为 20
.. square5

原始列表：(0,0) (-2,0)
    (-1,0) (1,0) (2,0) (0,0)
 长度 =8
翻转中间两个地址：
    (0,0) (-2,0) (1,0) (-1,0)
    (2,0) (0,0)
 长度 =12
更好：(0,0) (1,0) (-2,0)
    (-1,0) (2,0) (0,0)
 长度 =10

百户住宅
按顺序排列的路线长度为
    25852.6
基于简单列举的 TSP 算法
    长度： 2751.99 相较于朴素算法
    25852.6
单一路线长度为： 2078.43
 .. 新路线已接受
    长度 2076.65
最终路线长度为 2076.65
    相较于初始值 2078.43
TSP 路线长度为 1899.4
    初始值超过 2078.43

两条路线
Route1 :(0 0) (2 0) (3 2)      ,
    (2,3) (0,2) (0,0)
route2:  (0,0) (3,1) (2,1)
    (1,2) (1,3) (0,0)
total length 19.6251
start with 9.88635,9.73877
Pass 0
.. down to 9.81256,8.57649
Pass 1
Pass 2
Pass 3
Pass 4
TSP Route1:  (0,0) (3,1) (3,2)
    (2,3) (0,2) (0,0)
route2:  (0,0) (2,0) (2,1)
    (1,2) (1,3) (0,0)
total length 18.389
```

接下来我们需要一个类 *AddressList* 来包含地址列表。

练习 8.2. 实现一个类 *AddressList*；它可能需要以下方法：

- *add_address* 用于构建列表；
- *length* 给出按顺序访问所有地址需行驶的距离；
- *index_closest_to* 返回列表中与另一个（假设不在列表中的）地址最接近的地址。

8.2.2 添加一个仓库点

接下来，我们模拟路线需要从仓库点出发并最终返回该点的场景，该仓库点被随意设置在坐标 (0,0) 处。我们可以构建一个 *AddressList*，将仓库点作为首尾元素，但这可能会引发问题：

- 若对列表进行重新排序以最小化行驶距离，首尾元素可能无法保持原位。
- 我们可能希望列表中的元素具有唯一性：同一地址出现两次意味着需在同一地点进行两次交付，因此 *add_address* 方法会检查该地址是否已存在于列表中。

我们可以通过创建一个继承自 *AddressList* 的类 *Route* 来解决此问题，该类的相关方法会保持列表首尾元素固定不变。

8.2.3 路线的贪心算法构建

接下来我们需要构建一条路线。不同于求解完整的旅行商问题（TSP），我们首先采用一种贪婪搜索策略：

> 给定一个点，通过某种局部最优性测试（如最短距离）找到下一个点。绝不回头重新审视已构建的路线。

这种策略很可能会带来改进，但大概率不会给出最优路线。

Let's write a method

```
路径::路径贪婪_路径();
```

该方法会生成一个新的地址列表，包含相同的地址，但经过重新排列以缩短旅行距离。

练习 8.3. 为 AddressList 类编写贪婪_路线方法。

1. 假设路线从位于 (0,0) 的仓库出发。然后通过以下步骤逐步构建新列表：2. 维护一个表示当前位置的地址变量 *we_are_here*；3. 不断寻找距离 *we_are_here* 最近的地址。

通过处理不包含最后一个元素的子向量，将其扩展为 *Route* 类的一个方法。

在此示例上进行测试：

Code:

```
1 // /route.cpp
2     配送路线;
3     配送点.添加_地址( 地址(0,5)
      );
4     配送点.添加_地址( 地址(5,0)
      );
5     deliveries.add_address( Address(5,5)
      );
6     cerr <<"按顺序行进:"<<
      配送。长度() <<'\n';
7     assert(deliveries.size()==5 );
8     auto route =
      配送.贪婪_路线();
9     断言(路线.大小()==5 );
10    auto len = route.length();
11    cerr << "Square route: " <<
      route.as_string()
12         << "\n has length " << len <<
      '\n';
```

输出
[亚马逊] square5:

```
Travel in order: 24.1421
方形路线:(0,0) (0,5)
      (5,5) (5,0) (0,0)
 长度为 20
```

重组列表可以通过多种方式实现。

- 首先，可以尝试在原地进行修改。这会遇到一个反对意见，即可能需要保留原始列表；此外，虽然交换两个元素可以通过 `insert` 和 `erase` 方法完成，但更复杂的操作则较为棘手。
- 另一种方法是逐步构建一个新列表。此时的主要问题在于跟踪原始列表中哪些元素已被处理。可以通过为每个地址添加布尔字段 `done` 来实现，也可以复制输入列表并移除已处理的元素。为此，需研究 `erase` 方法针对 `vector` 对象的用法。

8.3 优化路线

上述每次寻找最近地址的建议被称为贪心搜索策略。它并不能给出 TSP 问题的最优解。寻找 TSP 的最优解编程难度较高 —— 可通过递归实现 —— 且随着地址数量增加会耗费大量时间。事实上，TSP 可能是 *NP* 难问题类别中最著名的一例，这类问题通常被认为其运行时间增长速度超过问题规模的多项式级别。

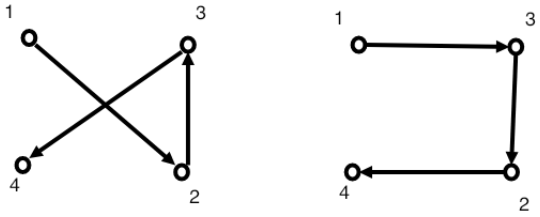


图 8.1：展示 ‘opt2’ 理念中反转部分路径的示意图

然而，你可以通过启发式方法近似求解。一种方法，即 Kernighan-Lin 算法 [12], 基于 *opt2* 理念：若路径存在 ‘自交叉’，通过反转可使其缩短

部分内容。图 52.1（教材）显示路径 $1--2--3--4$ 可以通过反转部分路径缩短，得到 $1--3--2--4$。由于识别路径自交叉点可能很困难，甚至对于没有关联笛卡尔坐标的图来说不可能实现，我们采用一种尝试所有可能反转的方案：

> 对于路径 `[1..N]` 上的所有节点 m<n：创建一条新路径，从`[1..m-1]` + `[m--n].reversed` + `[n+1..N]`若新路径更短，则保留它

练习 8.4。编写 opt2 启发式算法：编写一个方法来反转部分路径，并编写循环以尝试不同起点和终点的反转。通过简单测试案例验证你的代码是否按预期工作。

让我们探讨复杂度问题。（关于复杂度计算的介绍，参阅 HPC 书籍 [6], 第 20 节。） TSP 属于 *NP* 完全问题类，非正式地说，这意味着不存在比尝试所有可能性更优的解决方案。

练习 8.5. 使用 opt2 启发式解决方案的运行时复杂度是多少？通过考虑所有可能性找到最佳解的运行时复杂度又会如何？对两种策略在不同问题规模下的运行时间做一个非常粗略的估计：$N = 10, 100, 1000, \ldots$。

练习 8.6. 之前你已经编写了贪婪启发式算法。比较从 opt2 启发式算法中获得的改进，两者均从给定的地址列表开始，并使用贪婪遍历。

为了贴近现实，你会在路线上设置多少个地址？一名送货司机在典型的一天中会处理多少个地址？

8.4 多辆卡车

如果引入多辆送货卡车，我们就得到了 ‘多旅行商问题’ [3]。通过这种方式，我们可以模块化处理多辆卡车在同一天外出送货的情况，或者一辆卡车分散在几天内完成送货的情况。目前我们不对这两种情况进行区分。

首要问题是如何划分地址。

1. 我们可以通过某种几何测试将列表一分为二。这适用于多辆卡车同一天出勤的场景。但若将其视为同一卡车多日连续出勤的模型，则会忽略第一天可能新增地址的情况，从而打乱原本清晰划分的路线。2. 因此，更合理的假设可能是所有卡车获取的地址列表本质上都是随机分配的。

能否将 opt2 启发式算法扩展到多路径场景？参考教材图 52.2：不同于修改单一路径，我们可以在两条路径间交换片段。编写代码时需注意另一条路径可能是反向行驶的！这意味着基于第一条和第二条路径的分割点，你需要考虑四条可能的修改后路径。

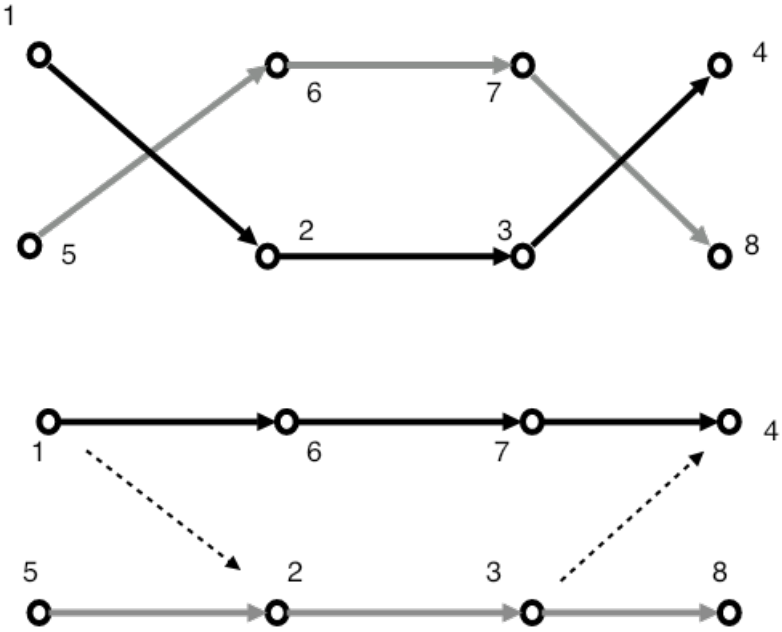


图 8.2：将 ‘opt2’ 思想扩展到多路径

练习 8.7. 编写一个函数，利用 opt2 启发式的多路径版本同时优化两条路径。测试案例可参考教材中的图 52.3 。

在此你有较大的自由度：

- 两条线段的起点应独立选择；
- 长度可独立选择，但并非必须；最后
- 每个片段都可以反转。

更高的灵活性也意味着程序运行时间更长。这值得吗？做一些测试并报告结果。

根据上述描述，将会有大量代码重复。确保为各种操作引入函数和方法。

8.5 Amazon prime

在 52.4 节（教材）中，你假设包裹在哪一天送达并不重要。这一假设因 *Amazon prime* 而改变，其要求包裹必须在次日保证送达。

练习 8.8. 探索一个场景：有两辆卡车，每辆车的路线中包含无法与另一路线交换的若干地址。总距离会增加多少？尝试调整 prime 与非 prime 地址的比例。

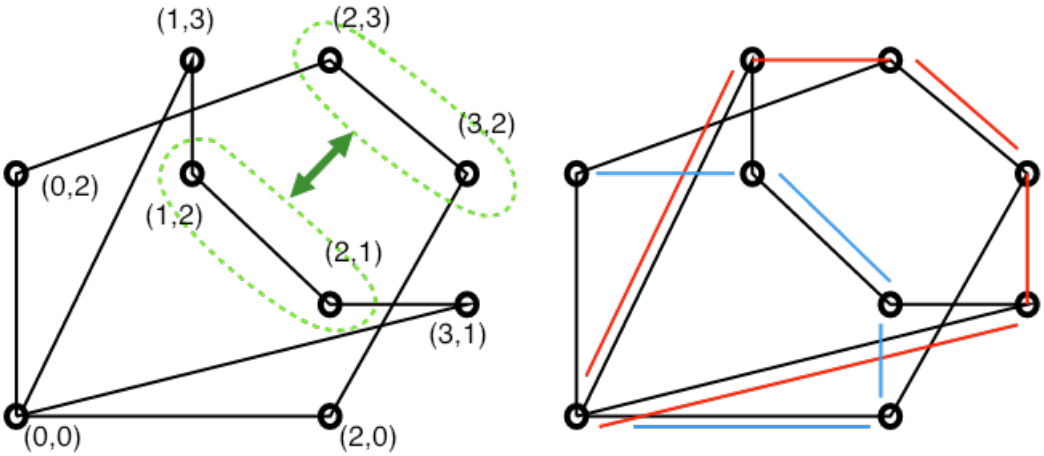


图 8.3：多路径测试用例

8.6 动态性

到目前为止，我们假设需要配送的地址列表是给定的。这当然不属实：新的配送需求会持续不断地需要被安排。

练习 8.9. 实现一个场景，每天随机数量的新配送任务被添加到列表中。探索策略和设计选择。

8.7 伦理

人们有时会批评亚马逊的劳动政策，包括对其司机的相关待遇。你能从你的模拟中对此提出什么见解吗?

第 9 章

大太平洋垃圾带

本节包含一系列逐步构建的练习，最终形成一个细胞自动机模拟：海洋中的海龟、对它们致命的垃圾，以及清理垃圾的船只。了解更多信息请访问：https://theoceancleanup.com/。

感谢 TACC 的 Ernesto Lima 提供此练习的创意和初始代码。

9.1 问题与模型解决方案

海洋中漂浮着大量塑料，这对鱼类、海龟和鲸类有害。在此你将模拟这些生物与

- 塑料，随机分布；
- 游动的海龟；它们繁殖缓慢，并会因吞食塑料而死亡；
- 清扫海洋塑料垃圾的船只。

Th我们使用的模拟方法是细胞自动机：

- We have a grid of cells;
- Each cell has a ‘state’ associated with it, namely, it can contain a ship, a turtle, or plastic, or be empty; and
- On each next time step, the state of a cell is a simple function of the states of that cell and its immediate neighbors.

本练习的目的是模拟若干时间步长，并探索参数间的相互作用：海龟会因多少垃圾而灭绝，需要多少船只才能保护海龟。

9.2 程序设计

基本思路是创建一个海洋对象，其中包含海龟、垃圾和船只。您的模拟将让海洋经历若干时间步长：

```
for (int t=0; t<time_steps; t++)
  ocean.update();
```

最终目的是研究海龟种群的发展：它是稳定的，还是会灭绝?

虽然您可以为此问题提供一个 ‘临时凑合’ 的解决方案，但部分评分将基于您对现代 / 简洁 C++ 编程技术的运用。以下提供若干建议。

9.2.1 网格更新

这里有一个需要注意的地方。你能看出完全原地更新有什么问题吗:

```
for ( i )
  for ( j )
    cell(i,j) = f( cell(i,j), .... other cells ... );
```

?

9.3 测试

测试该程序的正确性可能比较复杂。你能做的最好的事情就是尝试多种场景。为此，最好让你的程序输入灵活：使用 `cxxopts` 包 [4], 并通过 Shell 脚本驱动你的程序。

以下是一些你可以测试的内容列表。

1. 初始仅设置若干艘船只；运行 1000 个时间步后，验证数量是否保持不变。2. 同理测试海龟：若禁止繁殖与死亡，检查其数量是否恒定。3. 仅存在船只与垃圾时，是否全部被清理？4. 仅存在海龟与垃圾时，它们是否会全部死亡?

验证海龟与船只不会 ‘瞬移’ 而仅移动至相邻单元格较为困难。此时需通过视觉检查，参见章节 9.3.1。

9.3.1 动画图形

该程序的输出非常适合可视化呈现。事实上，某些测试（如 ‘确保海龟不会瞬移’ ）必须通过观察输出才能完成。首先制作 ASCII 字符渲染的海洋网格，如图 9.1 所示。

若能实现动态输出效果更佳。然而不同编程语言生成可视化输出的难易程度各异。C++ 虽拥有强大的视频 / 图形库，但使用门槛较高。这里推荐一种更简易的解决方案。

对于此类程序产生的简单输出，你可以制作低成本的基础动画。市面上所有终端模拟器 都支持 *VT100* 光标控制 [1]：只需向屏幕发送特定控制指令即可调整光标位置。

在每个时间步长中，您将

1. 将光标移至起始位置，通过此魔法指令 utput:

```
// /pacific.cpp
#include <cstdio>
    /* ... */
    // ESC [ i ; j H
    printf( "%c[0;0H",(char)27);
```

2. 如图 9.1 所示展示你的网格；3. 休眠一小段时间；参见章节 25.1.4 （教材）。

1. https://vt100.net/docs/vt100-ug/chapter3.html

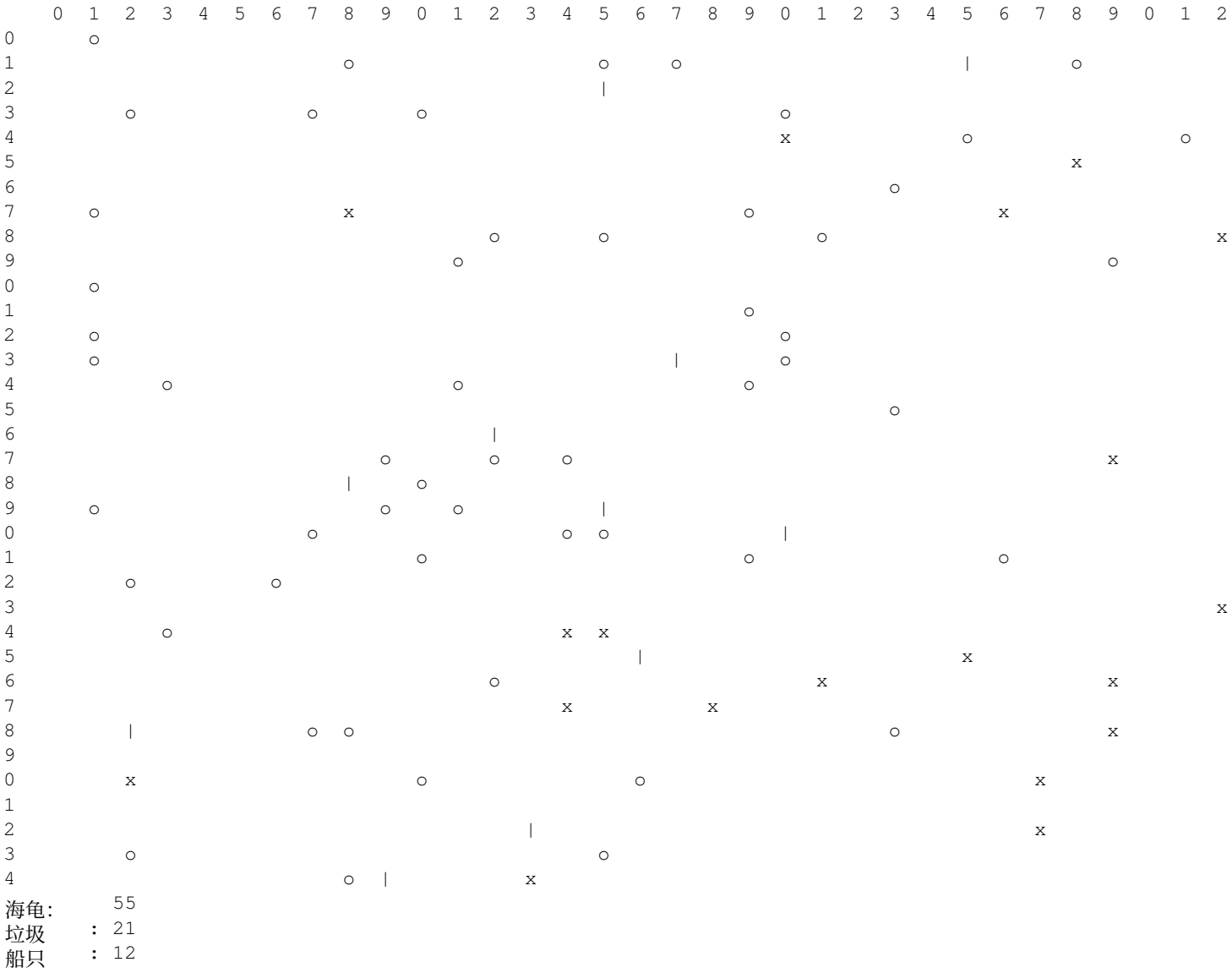


图 9.1：时间步的 ASCII 艺术打印输出

9. 大太平洋垃圾带

9.4 现代编程技术

9.4.1 面向对象编程

尽管只有一个海洋，您仍应创建一个 `ocean` 类，而非使用全局数组对象。所有函数都作为您创建的该类的单一对象的方法存在。

9.4.2 数据结构

通过忽略海洋深度和海岸线形状，我们几乎不会损失一般性，并将海洋建模为二维网格。

如果编写一个索引函数 `cell(i,j)`，可以使代码基本独立于实际选择的数据结构。请论证为何 `vector<vector<int>>` 并非最佳存储方案。

1. 你会使用什么替代方案？2. 当代码可运行时，能否通过计时证明你的选择确实更优？

9.4.3 单元格类型

代码中出现“魔法数字”（0 =empty，1 =turtle 等）不够优雅。建议创建一个 `enum` 或 `enum class`（参见教材第 24.10 节），以便为单元格内容命名：

```
// /pacific.cppcell(i,j) = 占据::
海龟;
```

若想打印海洋状态，若能直接 `cout` 单元格会很方便：

```
// /pacific.cpp for(int i=0; i<iSize; ++i) {
cout << i%10 <<" "; for (int j=0; j<jSize; ++j)
{ cout << setw(hs) <<cell(i,j); } cout
<< ’\n’; }
```

9.4.4 跨越海洋的范围测量

编写一个循环相当简单，比如

```
for (int i=0; i<iSize; i++)
  for (int j=0; j<jSize; j++)
    ... cell(i,j) ...
```

然而，并不推荐总是如此有序地遍历整个域。你能实现这个功能吗：

```
for ( auto [i,j] : permuted_indices() ) {
    ... cell(i,j) ...
```

? 参见教材章节 24.5 关于结构化绑定。

同样地，如果您需要统计海龟周围有多少片垃圾，您能让这段代码运行起来吗：

```
//pacific.cppintcount_around( int ic, int jc, occupy typ ) const {  i
nt count=0;  for ( auto [i,j] : neighbors(ic,jc) ) {  if (cell(i,j) ==
typ)++count;  }  return count;};
```

9.4.5 随机数

对于船只和海龟的随机移动，你需要一个随机数生成器。不要使用旧的 C 生成器，而是使用新的 `random` one; 章节 24.7 (教材).

尝试找到一个解决方案，以便在所有需要随机数的地方使用同一个生成器。提示：将生成器设为 `static` 在你的类中。

9.5 探索

与其让船只随机移动，能否让它们优先朝最近的垃圾带方向航行？这是否能改善海龟种群的健康状况？

能否通过让船只占据网格中的 2×2 个区块来体现船只与海龟的相对大小差异？

目前你让垃圾保持静止。若存在洋流呢？能否让垃圾具有 ' 粘性 '，使得接触的垃圾颗粒开始以斑块形式移动？

海龟以沙丁鱼为食。（其实并非如此。）若海龟灭绝，沙丁鱼种群会发生什么变化？你能设定出对应稳定生态系统或失衡生态系统的参数值吗？

9.5.1 代码效率

研究你在**枚举**章节 9.4.3 中的实现是否影响计时。解析 24.10 章节（教材）的细则。

你可能会注意到，在几乎空旷的海洋上进行范围遍历效率相当低下。可以考虑维护一个 ‘ 活动列表 ’ 来记录海龟等生物的位置，并仅对该列表进行循环遍历。你将如何实现这一点？预计会看到时间上的差异吗？实际效果如何？

选择向量嵌套向量的实现方式对海洋模拟的运行时间有何影响；参见章节 9.4.2。

9. 大太平洋垃圾带

第 10 章

高性能线性代数

线性代数是计算科学的基础。涉及偏微分方程（ PDEs ）的应用最终归结为求解大型线性方程组；固态物理学则涉及大规模特征值系统。但即便在工程应用之外，线性代数同样至关重要：深度学习（ DL ）网络的主要计算部分就包含矩阵与矩阵的乘法运算。

诸如矩阵乘法之类的线性代数操作，用简单的方式编码很容易实现。然而，这并不能带来高性能。在这些练习中，你将探索实现高性能的基本策略。

10.1 数学基础

矩阵乘法 $C \leftarrow A \cdot B$ 的定义如下

$$\forall_{ij} : c_{ij} \leftarrow \sum_k a_{ik} b_{kj}.$$

实现这一功能的直接代码示例如下：

```
for (i=0; i<a.m;i++) for (j=0; j<b.n; j
++)   s = 0; for (k=0; k<a.n; k++)
s += a[i,k]* b[k,j]; c[i,j] =s;
```

然而，这并不是编码该操作的唯一方式。通过调整循环顺序，总共可得到六种不同的实现方案。

练习 10.1. 编写其中一种调整循环顺序的算法并测试其正确性。若将上述参考算法称为 ‘ 基于内积 ’ 的实现，您会如何描述您实现的变体?

另一种实现基于分块策略。设 A, B, C 按 2×2 分块形式划分：

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \qquad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \qquad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Then

$$\begin{aligned} C_{11}&=A_{11}B_{11}+A_{12}B_{21},\\ C_{12}&=A_{11}B_{12}+A_{12}B_{22},\\ C_{21}&=A_{21}B_{11}+A_{22}B_{21},\\ C_{22}&=A_{21}B_{12}+A_{22}B_{22} \end{aligned} \tag{10.1}$$

请确认这实际上计算的是相同的乘积 $C = A{\cdot}B$。关于分块算法的更多内容，请参阅高性能计算书籍 [6]，章节 5.3.6。

练习 10.2. 编写一个带有乘法例程的矩阵类：

矩阵 *Matrix*：：乘法（ 矩阵 *other* ）；

首先实现传统的矩阵乘法，然后将其改写为递归形式。对于递归算法，你需要实现子矩阵处理：提取子矩阵，并将子矩阵写回外围矩阵中。

10.2 矩阵存储

存储 $M \times N$ 矩阵最简单的方式是使用一个长度为 MN 的数组。在这个数组中，我们可以选择将行首尾相接存储，或是存储列。虽然这一选择对库的实现具有明显的实际重要性，但从性能角度来看并无差异。

备注 5 历史上，线性代数软件如基础线性代数子程序 *(BLAS)* 采用列优先存储方式，这意味着元素 (i, j) 的位置被计算为 $i + j \cdot M$*(* 本项目将始终使用从零开始的索引，无论是代码还是数学表达式 *)*。这一选择的根源在于 *BLAS* 起源于 *Fortran* 语言，该语言采用列主序排列数组元素。另一方面，*C/C++* 语言中的静态数组 *(* 如 `x[5][6][7]` *)* 采用行主序排列，其中元素 (i, j) 存储在位置 $j + i \cdot N$。

前文提到的分块算法思想需要提取子矩阵。出于效率考虑，我们不希望将元素复制到新数组中，因此需要让子矩阵对应于子数组。
现在我们面临一个问题：只有由连续列组成的子矩阵才是内存连续的。当矩阵作为更大矩阵的子块时，元素 (i, j) 定位公式 $i + j \cdot M$ 将不再正确。

因此，线性代数软件通过三个参数 M, N, LDA 来描述子矩阵，其中 'LDA' 表示 ' A 的前导维度 '（参见 BLAS [10], 和 Lapack [1]）。如图 53.1 (教材) 所示。

Exercise 10.3. In terms of M, N, LDA, what is the location of the (i, j) element?

在实现层面我们也遇到问题。若使用 `std::vector` 存储，由于 C++ 强制要求向量拥有独立存储空间，无法获取子数组。解决方案是使用 `span`，详见章节 10.9.5(教材)。

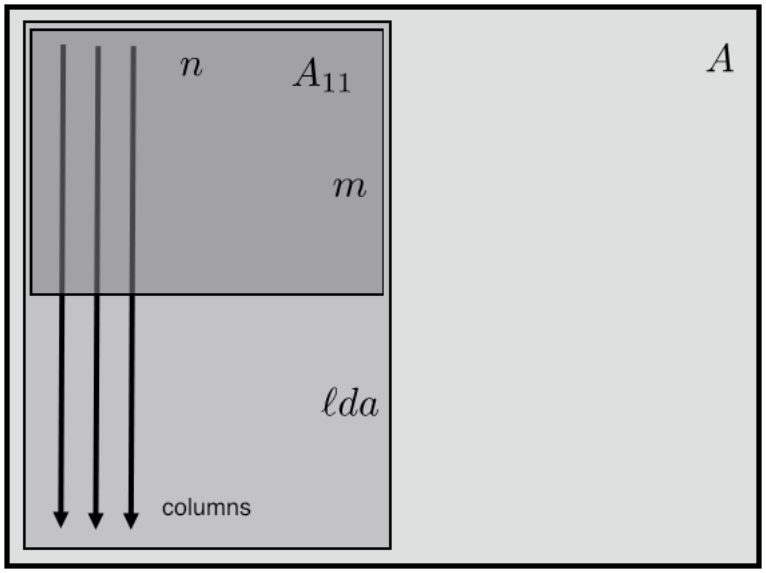


图 10.1：矩阵中的子矩阵，标明了 M 、 N 和子矩阵的 LDA

我们可能有两种类型的矩阵：存储向量 <double> 的顶层矩阵，以及存储跨度 <double> 的子矩阵，但这会带来很多复杂性。可以通过 std::variant （参见教材第 24.6.4 节）实现，但我们暂不采用。

相反，我们采用以下惯用法：在顶层创建一个向量，然后从其内存中创建矩阵。

```
// /lapack_alloc.cpp // M,LDA,N 的示例值 M = 2;
LDA = M+2; N = 3; // 创建存储数据的向量 vector<
double> one_data(LDA*N,1.); // 使用向量数据创建矩
阵 Matrix one(M,LDA,N,one_data.data());
```

（如果你之前没有用 C 编程过，需要适应 double* 机制。参见教材第 10.10 节。）

Exercise 10.4. Start implementing the *Matrix* class with a constructor

```
矩阵 :: 矩阵（整型行数，整型主维，整型列数，双精度 * 数据）
```

以及私有数据成员：

```
// /lapack_alloc.cpp
私有:
  int m,n,lda;
  span<double> data;
```

编写方法

```
double& Matrix::at(int i,int j);
```

可作为安全访问元素的方式使用。

让我们从简单操作开始。

练习 10.5. 编写一个矩阵相加的方法。在具有相同 M, N 但不同 LDA 的矩阵上测试它。

使用 `at` 方法非常适合调试，但效率不高。通过预处理器（教材第 21 章）引入替代方案：

```
#ifdef DEBUG
  c.at(i,j) += a.at(i,k) * b.at(k,j)
#else
  cdata[ /* expression with i,j */ ] += adata[ ... ] * bdata[ ... ]
#endif
```

where you access the data directly with

```
// /lapack_alloc.cpp
  auto get_double_data() {
    double *adata;
    adata = data.data();
    return adata;
  };
```

练习 10.6. 实现此功能。使用 cpp `#define` 宏来优化索引表达式。（参见第 21.2.2 节（教材）。）

10.2.1 子矩阵

接下来我们需要支持构造实际的子矩阵。由于我们主要针对 2×2 块形式进行分解，因此只需编写四种方法：

```
矩阵左侧 (int j); 矩阵右侧
(int j); 矩阵顶部 (int
i); 矩阵底部 (int i);
```

例如，`Left(5)` 给出包含 $j < 5$ 的列。

练习 10.7. 实现这些方法并对其进行测试。

10.3 乘法

You can 现在编写第一个乘法例程，例如使用 a prototype

```
void Matrix::MatMult( Matrix& other,Matrix& out );
```

或者，你也可以编写

```
矩阵乘法 :: 矩阵相乘 ( 矩阵 & 其他矩阵 );
```

但我们希望将对象的创建 / 销毁次数控制在最低限度。

10.3.1 单层分块

接下来，编写

```
void Matrix::BlockedMatMult( Matrix& other,Matrix& out );
```

使用上述 2×2 表单。

10.3.2 递归分块

最后一步是使分块递归化。

练习 10.8. 编写一个方法

```
void 递归矩阵乘法（矩阵 & 其他，矩阵 & 输出）;
```

其中

- 执行 2×2 块乘积，并再次使用 *RecursiveMatMult* 处理子块。
- 当块足够小时，改用常规的 *MatMult* 乘积。

10.4 性能问题

如果你稍微实验一下常规与递归矩阵乘法之间的分界点，会发现可以获得显著的性能提升因子。这是为什么呢?

矩阵乘法是科学计算中的基础操作，人们投入了大量精力对其进行优化。一个有趣的事实是，它几乎是求和运算下最可优化的操作。简而言之，原因在于它涉及 $O(N^3)$ 次操作作用于 $O(N^2)$ 数据量。这意味着理论上每个被取出的元素会被多次使用，从而克服了内存瓶颈。

要理解与硬件相关的性能问题，你需要阅读相关资料。HPC 书籍 [6]，章节 1.3.5 阐述了关键概念 —— 缓存。

习题 10.9。 论证朴素矩阵乘法实现实际上不太可能复用数据。

解释为何递归策略确实能实现数据复用。

前文已设定从递归乘法切换至常规乘法的分界点。

练习 10.10. 论证一旦乘积被包含在缓存中，继续递归不会有太大益处。你的处理器缓存大小是多少?

Do expe尝试不同的截断点。你能将此与 缓存 he sizes?

10.4.1 并行性（可选）

方程 53.1 (教材) 的四个子句针对 C 矩阵中的独立区域，因此它们可以在任何至少拥有四个核心的处理器上并行执行。

探索 OpenMP 库以并行化 *BlockedMatMult* .

10.4.2 比较（可选）

最终的问题是：你的速度离最佳可能速度有多接近？遗憾的是，你仍有差距。你可以通过以下方式探索这一点。

你的计算机可能有一个优化实现，可通过以下方式访问：

```
#include <cblas.h>cblas_dgemm ( CblasColMajor,
CblasNoTrans, CblasNoTrans, m,other.n,n,
alpha,adata,lda, bdata,other.lda, beta,cdata,out.lda);
```

用于计算 $C \leftarrow \alpha A \cdot B + \beta C$。

练习 10.11. 使用另一种 C++ 条件语句通过调用 `cblas_dgemm` 来实现 `MatMult` 。现在获得的性能如何？

您会发现递归实现比朴素实现快，但远不及 CBlas 实现。这是因为

- CBlas 实现可能基于完全不同的策略 [8]，并且
- 它可能涉及一定量的汇编编码。

第 11 章

图算法

在本项目中，您将探索一些常见的图算法及其多种可能的实现方式。核心主题在于：教科书上常见的算法表述方式在计算实现层面未必是最优方案。

作为本项目的基础知识，建议阅读 HPC 书籍 [6]，第 10 章；关于图的入门教程，可参阅 HPC 书籍 [6]，第 25 章。

11.1 传统算法

我们首先实现两种单源最短路径（*SSSP*）算法的「教科书式」表述：先处理无权图，再处理加权图。下一节我们将探讨基于线性代数的表述形式。

为了开发这些实现，我们从一些必要的预备知识开始，

11.1.1 代码预备知识

11.1.1.1 邻接图

我们需要一个类 `Dag` 来表示有向无环图（DAG）：

```
    // /dijkstra1.cpp 类 Dag{private:vector<
vector<int> > dag;public:// 构造含 'n' 个节点的
Dag 对象，初始无边。Dag( int n ):dag( vector<
vector<int> >(n) ) {};
```

It's probably a good idea to have a function

```
// /dijkstra1.cpp
  const auto& neighbors( int i ) const { return dag.at(i); };
```

给定一个节点，返回该节点邻居列表的函数。

练习 11.1. 完成 `Dag` 类。特别地，添加一个生成示例图的方法：

- 测试时 ‘环形’ 图通常很有用：连接边

$$0 \to 1 \to \cdots \to N-1 \to 0.$$

- 拥有一个带随机边的图可能也是个好主意。

编写一个显示该图的方法。

11.1.1.2 节点集

经典的单源最短路径（SSSP）算法表述，例如 ***Dijkstra*** 最短路径算法（参见 HPC 书籍 [6]，章节 10.1.3）使用逐步构建或消耗的节点集合。

You could implement that as a vector:

```
vector< int> set_of_nodes(nnodes);for
(int inode=0; inode<nnodes; inode++)// 标记
inode 为距离未知: set_of_nodes.at(inode) =
inf;
```

其中您可以使用某种约定（例如负距离）来指示节点已从集合中移除。

然而，C++ 拥有实际的 `set` 容器，其包含添加元素、查找元素及移除元素的成员方法；参见教材 24.3.2 节。这使得我们能够更直接地表达算法逻辑。在本例中，我们需要一个 int/int 或 int/float 对的集合（具体取决于图算法类型）。（也可使用 `map`，以 int 作为查找键，int 或 float 作为值。）

对于无权图，我们仅需要一个已处理节点集合，并将节点 0 作为起点插入：

```
// /queuelevel.cpp
  using node_info = std::pair<unsigned,unsigned>;
  std::set< node_info > distances;
  distances.insert( {0,0} );
```

对于 Dijkstra 算法，我们既需要一个已处理节点集合，也需要一个待处理节点集合。我们再次将起始节点设为 0，并将所有未处理节点的距离初始化为无穷大：

```
//queuedijkstra.cpp  const unsigned inf = = std::numeric_limits<
<unsigned>>::max(); using node_info= = std::pair<<unsigned, unsigned>
>; std::set<<node_info> > distances, to_be_done; to_be_
_done.insert({0, 0}); for (unsigned n=1 = 0; n< < graph_size; ++++n)  to_
_be_done.insert({n, inf});
```

(为什么在加权图情况下需要第二个集合，而无权图却不需要？)

练习 11.2. 编写一个代码片段，测试节点是否在 `distances` 集合中 t.

- 当然可以为此编写循环。需要注意的是，遍历集合时会获取键 / 值对。使用结构化绑定；参见教材 24.5 节。

- 但最好使用 ‘算法’，即技术意义上的 ‘标准库内置算法’。本例中，`find`。
- … 区别在于 `find` 需要精确匹配键值对，而此处需查询：‘该节点是否存在于 `distances` 集合中且值任意’。使用 `find_if` 算法；参见 24.3.2 节（教材）。

11.1.2 水平集算法

我们从简单算法开始：无权图中的 SSSP 算法；详见 HPC 教材 [6]，第 10.1.1 节。等效地，我们在图中寻找水平集。

对于无权图，距离算法相当简单。归纳步骤如下：

- 假设我们有一组最多在 n 步内可达的节点，
- 那么它们的邻居（尚未包含在该集合中的节点）可以在 $n+1$ 步内被到达。

算法概要如下

```
// /queuelevel.cpp for (;;) { if (distances.size()==graph_size)break; /* * 遍历所有
已完成节点 */ for ( auto [node,level] : distances ) { /* * 将该节点的邻居距离设为
‘level + 1’ */ const auto& nbors= graph.neighbors(node); for (auto n :
nbors ) { /* * 检查 ‘n’ 是否已有已知距离，* 若无，则以 level+1 将其加入 ‘
distances’ */ /*... */ { cout << "node "<< n << " level " << level+1 << ’ \n’ ;
distances.insert( {n,level+1} ); }
```

Exercise 11.3. Finish the program that computes the SSSP algorithm and test it.

这段代码存在明显的低效问题：对于每个层级，我们都会遍历所有已完成节点，即使它们的邻居可能已被全部处理完毕。

练习 11.4. 维护一组 ‘当前层级’ 节点，并仅在这些节点中探查以确定下一层级。在若干大型图上对两种变体进行计时比较。

11.1.3 Dijkstra 算法

在 Dijkstra 算法中，我们既维护已确定最短距离的节点集合，也维护仍在确定最短距离的节点集合。注意：对于某个节点的暂定最短距离

节点可能会被多次更新，因为可能存在多条路径到达该节点。就权重而言的 “ 最短 ” 路径，在遍历的边数上未必是最短的！

主循环现在看起来类似于：

```
// /queuedijkstra.cpp for (;;) { if (to_be_done.size()==0) break; /* * 找到距离最小的节点 */ /* ... */ cout << "min: " << nclose << " @ " << dclose<< ' \n' ; /* * 将该节点移至已完成集合 */ to_be_done.erase(closest_node); distances.insert( *closest_node ); /* * 设置 nclose 邻居的距离为当前距离 + 1 */ const auto& nbors = graph.neighbors(nclose); for ( auto n: nbors ) { // 在 distances 中查找 ‘n’ /* ... */ { /* * 如果 ‘n’ 的距离未知， * 在 ‘to_be_done’ 中找到并更新其位置 */ /* ... */ to_be_done.erase(cfind ); to_be_done.insert( {n,dclose+1} ); /* ... */ }
```

(注意，我们会在 *to_be_done* 集合中删除一条记录，然后重新插入相同键名的新值。如果使用 *map* 而非 *set*，本可直接更新数据。)

在已完成 / 未完成集合中定位节点的具体实现方式由您决定。可以使用简单循环，或通过 *find_if* 查找匹配节点编号的元素。

练习 11.5. 完善上述框架细节以实现 Dijkstra 算法。

11.2 线性代数表述

在本项目环节中，您将探索如何让图算法更贴近线性代数的表现形式。

11.2.1 代码预备工作

11.2.1.1 数据结构

你需要一个矩阵和一个向量。向量很简单：

```
// /graphmvpdijkstra.cpp 类Vector {私有: 向量<向量值> 值; 公开 : 向量 ( 整型 n): 值 ( 向量<向量值>(n, 无限 ) )
{};
```

对于矩阵，初始时使用密集矩阵：

```
//graphmvpdijkstra.cpp 类 AdjacencyMatrix { private: vector<vector<matrixvalue> > adjacency; public: AdjacencyMatrix(int n) : adjacency( vector<vector<matrixvalue>>( n,vector<matrixvalue>(n,empty) ) ) { };
```

但后续我们会对此进行优化。

备注 6 通常来说，将矩阵存储为向量套向量的形式并非良策，但在此案例中，我们需要能够返回矩阵的行，因此这种方式较为便利。

11.2.1.2 Matrix vector multiply

让我们编写一个例程

```
向量邻接矩阵 :: 左乘 ( 常量 向量 & 左操作数 );
```

这是最简单的解决方案，但不一定是最有效的，因为它为每次矩阵 - 向量乘法都创建了一个新的向量对象。

正如理论背景中所解释的，图算法可以表述为采用非常规加法 / 乘法运算的矩阵 - 向量乘法。因此，乘法例程的核心可能如下所示

```
// graphmvp.cpp for (int row=0; row<n; ++row) { for (int col=0; col<n; ++col) { result[col] = add(result[col],mult(left[row],adjacency[row][col])); } }
```

11.2.2 无权图

练习 11.6. 实现 `add`/`mult` 例程，使无权图上的 SSSP 算法能够运行。

11.2.3 Dijkstra 算法

例如，考虑以下邻接矩阵：

```
. 1 . . 5
. . 1 . .
. . . 1 .
. . . . 1
1 . . . .
```

最短距离 $0 \to 4$ 为 4，但在第一步中发现了一个更大的距离 5。您的算法应针对已知最短距离的连续更新显示类似以下的输出：

输入 : 0 . . . . 步骤 0: 0

1 . . 5 步骤 1: 0 1 2 . 5

步骤 2: 0 1 2 3 5 步骤

3: 0 1 2 3 4

练习 11.7. 实现新版本的 `add` / `mult` 例程，使矩阵 - 向量乘法对应于加权图上单源最短路径（ SSSP ）的 Dijkstra 算法。

11.2.4 稀疏矩阵

通过仅存储非零元素，可以使上述矩阵数据结构更加紧凑。请实现这一点。

11.2.5 进一步探索

通过运算符重载，你能将代码优化到多优雅的程度?

你能编写所有节点对最短路径算法吗?

你能扩展 SSSP 算法以同时生成实际路径吗?

11.3 测试与报告

你现在拥有两种完全不同的图算法实现。生成一些大型矩阵并对算法进行计时。

讨论你的发现，重点关注执行的工作量和所需的内存容量。

第 12 章

气候变化

> 气候一直在变化，且始终处于变化之中。
>
> 拉杰 · 沙阿，白宫首席副新闻秘书

“ 气候始终在变化 ” 这一论断远非严谨的科学主张。若将其理解为对气候统计行为的描述（此处以全球平均温度为衡量指标），我们可赋予其一定意义。本项目将使用真实温度数据进行简单分析。（灵感来源于 [11]。）

理想情况下，我们会采用全球各地监测站的多组数据集。Fortran 因其数组操作特性成为理想语言（参见第 39 章（教材））：单行代码即可处理所有独立测量值。为简化操作，此处采用 1880-2018 年间逐月数据的单一文件，将各月份视为 “ 模拟 ” 独立测量值。

12.1 读取数据

在代码仓库中，您可以找到两个文本文件

GLB.Ts+dSST.txt GLB.Ts.txt

这些文件包含 1951–1980 年平均温度的偏差数据。偏差数据给出了 1880–2018 年间每年的每个月。这些数据文件及更多内容可在 https://data.giss.nasa.gov/gistemp/ 找到。

练习 12.1。首先列出可用的年份，并创建一个大小为 $12 \times$ nyears 的数组 monthly_deviation，其中 nyears 是文件中完整年份的数量。使用格式和数组表示法。

文本文件中包含一些与您无关的行。您是在程序中过滤掉它们，还是使用一个 shell 脚本？提示：巧妙地使用 grep 将使 Fortran 代码更加容易。

12.2 统计假设

我们假设 Shah 先生实际上是在说气候具有 ‘ 平稳分布 ’，即高低值具有与时间无关的概率分布。这意味着在 n 个数据点中，

每个点有 $1/n$ 的概率成为新高记录。由于超过 $n+1$ 年中每年都有 $1/(n+1)$ 的概率，第 $n+1$ 年有 $1/(n+1)$ 的概率成为新高。

我们得出结论，作为 n 的函数，新高（或新低，但这里我们专注于新高）出现的概率随 $1/n$ 递减，且连续新高之间的间隔大致是年份 [1] 的线性函数。

这是我们可以验证的。

练习 12.2. 创建一个与 monthly_deviation 形状相同的 previous_record 数组。该数组记录（针对每个月，请记住我们将其视为独立测量）该年是否为记录年，如果不是，则记录前一次记录出现的时间：

$$\mathrm{PrevRec}(m,y) = \begin{cases} y & \text{if } \mathrm{MonDev}(m,y) = \max_{m'}\big(\mathrm{MonDev}(m',y)\big) \\ y' & \text{if } \mathrm{MonDev}(m,y) < \mathrm{MonDev}(m,y') \\ & \text{and } \mathrm{MonDev}(m,y') = \max_{m''<m'}\big(\mathrm{MonDev}(m'',y)\big) \end{cases}$$

再次强调，使用数组表示法。这也是使用 Where 子句的绝佳场景。

Exercise12.3. 现在针对每个月，找出记录之间的间隔。这将得到两个数组：gapyears 记录间隔开始的年份，gapsizes 记录该间隔的长度。

此函数由于是单独应用于每个月，因此不使用数组符号 on.

假设现在认为 gapsizes 是年份的线性函数，例如以起始年份为基准的距离。当然它们并非严格的线性函数，但或许可以通过线性回归拟合出一条线性函数。

Exercise 12.4. 复制代码来自

http://www.aip.de/groups/soe/local/numres/bookfpdf/f15-2.pdf并加以调整以适应我们的需求：找到描述记录间隔的线性函数的最佳斜率和截距拟合。

你会发现这些间隔明显不是线性增长的。那么这个负面结果是否意味着故事的终结，还是我们可以进一步探索?

Exercise12.5. 你能将这个练习转化为全球变暖的测试吗？能否将偏差解释为温度逐年上升加上一个平稳分布的总和，而非单独的平稳分布?

1. 从技术上讲，我们处理的是温度的均匀分布，这使得最大值和最小值服从贝塔分布。

第 13 章

桌面计算器解释器

在本系列练习中，你将编写一个 ‘桌面计算器’：一个结合数值与符号计算的小型交互式计算器。

这些练习主要涉及教材第 36 章（教科书）、 41 章（教科书）、 35 章（教科书）的内容。

13.1 命名变量

我们从处理 ‘命名变量’ 开始：*namedvar* 类型将字符串与变量关联起来：

```
// /varhandling.F90
  type namedvar
     character(len=20) :: expression = ""
     integer :: value
  end type namedvar
```

命名变量包含一个值和生成该变量的表达式字符串字段。创建变量时，表达式可以是任意内容。

```
// / 变量处理 .F90 类型 ( 命名变
量 ) ::x,y,z,ax = 命名变量 ("
x",1 )y = 命名变量 ("yvar",2 )
```

接下来我们将用这些类型对象进行计算。例如，将两个对象相加

- 会累加它们的值，并且
- 会拼接它们的 *expression* 字段，生成与总和值对应的表达式。

你的第一个任务是编写 *varadd* 和 *varmult* 函数，使以下程序能按照指定输出运行。这涉及教材第 35.3 节（教科书）和 41.5 节（教科书）中的字符串操作。

练习 13.1. 以下主程序应产生对应输出：

Code:

```
1// /varhandling.F90
2  打印 *,x
3  print *,y
4  z = 变量相加 (x,y)
5  print *,z
6  a = varmult(x,z)
7  print *,a
```

输出 [t t s ruc f] varhandling:

```
x                          1
yvar                       2
(x)+(yvar)                 3
(x)*((x)+(yvar))           3
```

(需要明确的是：这两个例程需要同时处理数值和字符串的 ‘加法’ 与 ‘乘法’ 运算。)

你可以基于代码仓库中名为 *namedvar.cpp* 的文件进行开发

13.2 首次模块化

让我们通过引入模块来组织目前的代码；详见第 37 章（教材）。

练习 13.2。 创建一个模块（建议名称：`VarHandling`），并将 `namedvar` 类型定义及例程 `varadd`、`varmult` 移至其中。

练习 13.3。 同时创建另一个模块（建议名称：`InputHandling`），包含第 35 章（教材）字符练习中的例程 `islower`、`isdigit`。你还需要一个 `isop` 例程来识别算术运算符。

13.3 事件循环与栈

在编写解释器的过程中，我们将实现一个 ‘事件循环’：持续接收单个字符输入并处理的循环。输入 `"0"` 表示终止该进程。

练习 13.4. 编写一个循环，接收字符输入，并仅输出遇到的字符类型：小写字母、数字或表示算术运算的字符 `+-*/`。

代码:

```
1 // /interchar.F90
2  do
3     read *,input
4     if (input .eq. '0') then
5        exit
6     else if ( isdigit(input) ) then
```

输出 [结构体 f] 字符间:

```
Inputs: 4 x 3 + 0
 4 is a digit
 x is a lowercase
 3 is a digit
 + is an operator
```

使用上文介绍的 `InputHandling` 模块。

13.3.1 栈

接下来，我们将把值存储在 `namedvar` 类型的栈上。栈 是一种数据结构，新元素被置于顶部，因此需要通过栈指针 来指示顶部元素。同理，栈指针也表明了当前已有元素的数量：

```
// / 解释。F90 类型 ( 命名变量 ), 维度 (10) ::
堆栈整型 :: 堆栈指针 =0
```

由于我们正在使用模块，建议将栈从主程序中分离出来，放入相应的模块中。

练习 13.5. 将栈 变量及栈指针添加到 *VarHandling* 模块中。

Since Fortran uses 1-based indexing, a starting value of zero is correct. For C/C++ it would have been −1.

接下来我们将开始实现栈操作，例如将 *namedvar* 对象压入栈中。

13.3.2 栈操作

我们在原有仅识别输入字符的事件循环基础上进行了扩展，实际整合了操作行为。即，我们循环执行以下步骤：

1. 从输入中读取一个字符；2. 0 会导致事件循环退出；否则：3. 如果是数字，则在栈顶创建一个新的命名变量条目，其数值同时作为值字段的数字形式和表达式字段的字符串形式。您可能会想在主程序中编写如下代码：

```
// /interpret.F90
     if ( isdigit(input) ) then
        stackpointer = stackpointer + 1
        read( input,'( i1 )' ) stack(stackpointer)%value
        stack(stackpointer)%expression = trim(input)
```

（您已在练习 *isdigit* 中编写过 35.1 （教材）。）但更清晰的设计是调用 *VarHandling* 模块中的方法：

```
// /internum.F90
     else if ( isdigit(input) )
    then
        call stack_push(input)
```

```
// /interpretm.F90
子程序 stack_push(input)
  implicit none
  character,intent(in) :: input
```

注意 *stack_push* 例程并未将栈或栈指针作为参数：由于它们同属一个模块，可作为全局变量访问。最后，

4. 如果是表示操作 +、 −、 ×, / 的字母，则：(a) 从栈顶取出两个条目，降低栈指针；(b) 对操作数应用该操作；(c) 将结果压入栈中。

辅助函数 *stack_display* 的实现有些技巧性，因此在此提供。这里使用了字符串格式化（参见教材第 41.3 节）以及隐式 do 循环（教材第 32.3 节）：另外需注意，*stack* 数组与 *stackpointer* 的行为类似于全局变量。

```
// /internum.F90
subroutine stack_display()
  implicit none
  ! local variables
  integer :: istck

  if (stackpointer.eq.0) return
  print '( 10( a,a, a,i0,"; ") )', ( &
       " expr=",trim(stack(istck)%expression), &
       " val=",stack(istck)%value, &
       istck=1,stackpointer )

end subroutine stack_display
```

让我们将各种选项添加到事件循环中。

练习 13.6 Make your event loop accept digits, creating a new entry:

代码:

```
1// /internum.F90
2       否则如果 ( isdigit(input)) 那么
3          调用 栈_压入 ( 输入 )
```

输出
[结构体] 内部编号:

```
输入:4560
 表达式=4 值=4;
 表达式=4 值=4;  表达式=5 值=5;
 表达式=4 值=4;  表达式=5 值=5;  expr=6
```

接下来我们整合操作：如果输入字符对应算术运算符，我们就调用堆栈_操作符处理该字符。该例程会根据字符类型调用相应的运算。

Exercise 13.7. Add a clause to your event loop to handle characters that stand for arithmetic operations:

代码:

```
1 // /internum.F90
2       else if ( isop(input) ) then
3          call stack_op(input)
```

Output
[structf] internumop:

```
Inputs: 4 5 6 + + 0
 expr=4 val=4;
 expr=4 val=4;  expr=5 val=5;
 expr=4 val=4;  expr=5 val=5;  expr=6
 expr=4 val=4;  expr=(5)+(6) val=11;
 expr=(4)+((5)+(6)) val=15;
```

13.3.3 项目复制

最后，我们可能希望多次使用同一个栈条目，因此需要实现复制栈条目的功能。

为此我们需要能够引用栈条目，因此添加一个单字符标签字段：`namedvar` 类型现在存储

1. 一个单字符标识符，
2. 一个整数值，以及

3. 其生成表达式作为字符串。// /

```
vartype.F90type namedvarcharacter ::
idcharacter(len=20)::
expressioninteger ::valueend
typenamedvar
```

练习 13.8. 向 namedvar 添加 id 字段，并确保您的程序仍能编译和运行。

事件循环现在扩展了一个额外步骤。如果输入字符是小写字母，则将其用作 namedvar 的 id，如下所示。

- If there is already a stack entry with that id, it is duplicated on top of the stack;
- otherwise, the id of the stack top entry is set to this character.

以下是新 stack_print 函数的相关部分 n:

```
//interprets.F90 打印 '( 10( a,a1, a,a, a,i0,"; ") )', ( & "id:",
stack(istck)%id, & " expr=",trim(stack(istck)%expression), & "
val=",stack(istck)%value, & istck=1,top )
```

练习 13.9. 编写事件循环中缺失的函数及其子句：

代码：

```
1 // /interpret.F90
2          stacksearch =
       在 _ 栈上查找 _( 栈 , 栈指针 , 输入 )
3          如果 ( stacksearch>=1 ) 那么
4             堆栈指针 =  堆栈指针 +1i
5             stack(stackpo nter) =  stack(stacksearch)
```

输出

[结构体 f] 栈查找：

```
Inputs: 1 x 2 y x y + z 0
id:. expr=1val=1;
id:x expr=1val=1;id
;xd=xp id=x expr=1val=1; id:y exp x r=2 val=2;
                                  r=2 val=2;
id:x 表达式 =1 值 =1; id:y 表达式 id 1  r=2 val=2;
l1 :id expr=  va = ;   :y expr=2 val=2;
    值 =2;1
id:x expr=  val=1; id:y expr=2 val=2;
id:x 表达式 =1  值 =1;id:y 表达式 r=2 val=2;
```

（这个条件语句的 else 部分包含什么内容？）

13.4 模块化

利用目前开发的模块和函数，您已拥有一个非常简洁的主程序：

```
// /intermod.F90
  do
     call stack_display()
     read *,input
     if (input .eq. '0') exit
     if ( isdigit(input) ) then
```

```
    调用 堆栈_压入（输入）否则如果（ 是
运算符（输入））则调用 堆栈_运算符（输
入）否则如果（ 是小写字母（输入））则调
用 堆栈_名称（输入）结束条件 结束循环
```

可以看到，通过将堆栈移入模块后，主程序中既看不到堆栈变量也看不到堆栈指针了。

但这种设计存在一个重要限制：整个系统只有一个堆栈，它被声明为某种全局变量，通过模块进行访问。
使用全局数据是否属于良好实践是另一个问题。在当前场景下这是合理的：计算器应用中有且仅有一个堆栈。

13.5 面向对象

但有时我们确实需要不止一个栈。让我们将栈数组和栈指针打包成一个新类型：

```
// / 类间.F90 类型 栈结构类型（命名变量），维度（10）:: 数据 整型 ::
顶部 =0 包含 过程，公开 :: 显示， 查找_标识， 名称， 操作， 压入结束
类型栈结构
```

练习 13.10。修改事件循环，使其调用 *stackstruct* 类型的方法，而非以该栈为输入参数的函数。

For instance, the *push* function is called as:

```
// / 类间.F90
     if （ isdigit(input) ） then
        调用堆栈 % 压入（输入）
```

13.5.1 运算符重载

varadd 及类似的算术例程通过函数调用来实现我们本希望以算术运算符形式编写的操作。

练习 13.11. 在 *varop* 函数中使用运算符重载：

```
// /interpretov.F90
  if (op=="+") then
     varop = op1 + op2
```

等等。

PART III

附录

第 14 章

项目提交风格指南

计算的目的是洞察，而非数字。（理查德 · 汉明）

项目报告与代码同等重要。以下是撰写优质报告的常识性指南，但并非所有部分都适用于您的项目。请运用您的判断力。

14.1 通用方法

作为通用准则，请将编程视为实验科学，而您的报告则是关于已完成测试的总结：阐明您要解决的问题、实施策略及取得的结果。

提交一份由文本处理程序生成的 PDF 格式报告（不接受 Word 或纯文本文档），推荐使用 LaTeX。教程可参考 Tutorials 书籍 [5], 第 15 节。

14.2 风格规范

你的报告应当遵循标准英语的写作规范。

- 正确的拼写和语法使用是基本要求。
- 使用完整的句子。
- 尽量避免使用贬义或不恰当的冗词赘句。*Google* 开发者文档风格指南 [7] 是极佳的参考资源。

14.3 报告撰写结构

请将本次项目报告视为练习撰写科学论文的机会 .

从最基础的要素开始

- 报告应包含标题。不要直接用 ‘项目’ 二字，而应采用类似 ‘时空漏斗模拟’ 这样的具体名称。
- 作者及联系方式。根据具体情况而异。此处需要填写：您的姓名、 EID （电子标识号）、 TACC 用户名以及电子邮箱。
- 概述性引言部分需高度凝练：说明研究问题、采用方法及主要发现。

- 结论：你的发现意味着什么，有哪些局限性，未来扩展的机会。
- 参考文献。

14.3.1 引言

你的文档读者无需熟悉项目描述，甚至无需了解其解决的问题。请说明问题是什么，如适用可提供理论背景，可能的话简述历史背景，并以全局术语描述你如何着手解决问题，以及你的发现的简要陈述。

14.3.2 详细展示

See section 14.5 below.

14.3.3 讨论与总结

将详细阐述与结尾讨论分开：你需要一个简短的最终章节来总结你的工作和发现。你也可以讨论将工作扩展到未涵盖案例的可能性。

14.4 Experiments

你不应期望程序运行一次就能为研究问题提供最终答案。
自问：哪些参数可以调整，然后调整它们！这能让你生成图表或多维绘图。

若调整参数，需考虑使用的粒度。十个数据点是否足够，还是使用 10,000 个能带来更深见解?

最重要的是：计算机速度极快，每秒可执行十亿次操作。因此，不要羞于运行长时间程序。你的程序不是按个键就立即给出答案的计算器：应预期程序运行可能需要几秒，甚至几分钟。

14.5 工作成果的详细展示

工作成果的详细展示应包含代码片段、表格、图表及其相关描述的有机结合。

14.5.1 数值结果的呈现

可通过图表或表格形式呈现结果。选择依据包括数据点数量多少、以及图表中是否存在显而易见的关联性等因素。
图表可通过多种方式生成。若能掌握 LaTeX `tikz` 包的使用值得赞赏，但 Matlab 或 Excel 亦可接受。不过请勿使用截图。
为图表 / 表格编号并在正文中引用编号。为图表添加清晰标签并标注坐标轴名称。

14.5.2 代码

您的报告应从全局角度描述所开发的算法，并包含相关代码片段。如需完整代码清单，请将其移至附录：正文中的代码片段仅应用于说明特别关键的点。

请勿使用代码截图：至少应使用等宽字体（如 verbatim 环境），但强烈推荐使用 listings 包（本书所用）。

14.6 总索引

第 15 章

参考文献

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, 及 D. Sorensen. LAPACK：一种适用于高性能计算机的可移植线性代数库。载于 *Proceedings Supercomputing '90*，第2–11 页。IEEE 计算机学会出版社，加利福尼亚州洛斯阿拉米托斯，1990 年。

[2] Roy M. Anderson 与 Robert M. May. 传染病种群生物学：第一部分。*Nature*，280 卷：361–367 页，1979 年。doi:10.1038/280361a0。

[3] Tolga Bektas. The multiple traveling salesman problem: an overview of formulations and solution procedures. *Omega*, 34(3):209 – 219, 2006.

[4] https://github.com/jarro2783/cxxopts。

[5] Victor Eijkhout. HPC carpentry. https://theartofhpc.com/carpentry.html.

[6] Victor Eijkhout. 计算机科学。 http://theartofhpc.com/istc.html.

[7] Google 开发者文档风格指南 https://developers.google.com/style/。

[8] Kazushige Goto 与 Robert A. van de Geijn。高性能矩阵乘法的剖析。*ACM Trans. Math. Softw.*, 34(3):1–25, 2008.

[9] https://mathworld.wolfram.com/Kermack-McKendrickModel.html.

[10] C. L. Lawson 、 R. J. Hanson 、 D. R. Kincaid 和 F. T. Krogh。Fortran 使用的基本线性代数子程序。*ACM Trans. Math. Softw.*, 5(3):308–323, 1979 年 9 月。

[11] Juan M. Restrepo 与 Michael E. Mann。.. 气候总是这样变化的 ...。*APS Physics,GPS newletter*, 2018 年 2 月。

[12] Lin S. 和 Kernighan B.。旅行商问题的有效启发式算法。*OperationsResearch*, 21:498–516,1973。