多种数值约分算法的效率比较研究

摘 要

本项目源自于对分子分母为超大整数的分数进行约分的方法探索。我们将独立研究过程中寻找到的一种倒数分解约分算法与历史已有的另外几种常用约分算法进行了计算效能差异性的理论分析，并通过计算机编程进行了多种约分算法效率的工程测试验证与比较。

本项目研究中，我们根据四种不同约分算法的数学原理，编写了相对应的 Python 语言计算程序，还进一步设计和构造了从测试用例生成器、测试分析器到数据绘制器的一系列研究工具。基于对各算法性质的分析，我们初步推测对于不同类型的待约分数，各种约分算法的计算效率存在差异性，因此有目的的设计了不同类型的测试用例数据集合，并通过生成器程序制作了对应的测试数据包。通过测试器分析器程序对待约分数进行约分运算，记录了不同算法处理下的约分完成时间，从而得到了各种测试条件下算法执行效率的实测数据。最后通过数据绘制器程序将测试结果进行有序的绘制，生成了直观可视化的数据图表，从而对多种数值约分算法的计算效率差异进行了较为系统的的对比研究。

通过这次研究，我们发现几种约分算法在处理不同数据集合时，效率的表现模式存在不同，并初步分析了产生差异的一些原因。本研究的结果有助于针对不同的计算对象选择并发挥不同约分算法的优势，提高超大整数约分计算程序的性能。

关键词：**约分法，倒数分解法，连分数，算法编程，时间复杂度，测量干扰抑制，数据可视化**

第一章 综述

1. 关于约分法

约分法是一种非常基础的数学算法，在数值计算和计算机编程领域有广泛应用。把一个分数化成最简分数（或称既约分数）的过程就叫约分，具体的过程为：根据分数的性质，分子分母同除以公因数，分数的值不变，通过这一方式对分数进行化简，直到最后分子分母两数互质而得到最简的等值分数。在数值计算的过程中，如果需要对有理数进行精确计算，必须使用分数的形式，而约分是整个计算过程中必不可少的算法。由于现在的数值计算已经越来越多地借助电子计算机进行处理，如果要进行快速与高精度的计算，设计高效与精确的约分算法程序具有重要的现实意义。

通常我们对于约分的认知最初是来自小学课本中简单约分的例子，例如$\frac{168}{240}$，我们可以快速的通过公因数的尝试和多次约分计算，从而得到结果$\frac{7}{10}$；但是当需要计算处理超大整数之间约分的时候，例如分数$\frac{19132241174083603}{267163316146024169}$，通常这已经不是通过直接的观察就可以下手计算的了，不借助一定的工具和特定方法，想计算得到结果就相当困难。

事实上，$\frac{19132241174083603}{267163316146024169}=\frac{69821\times954379\times287117}{974983\times954379\times287117}=\frac{69821}{974983}$，分子分母分别是三个五至六位质数的乘积，这个约分计算基本上不太可能通过手工的公因数筛选来完成了。通过编程，利用辗转相除法我们在普通的笔记本电脑上只需要用 15 毫秒就可以得到它的结果；不过如果直接用人们习惯的枚举公因数的方法，即使由计算机编程实现也要接近 1 秒，两者耗时相

差 60 多倍，而且随着数值的位数增加，不同的算法之间效率差异可能还会急剧扩大。约分算法不仅是一类应用广泛的算法，而且和超大整数的因数分解等应用具有密切的联系，而基于后者的秘钥算法是当今整个互联网安全机制的核心。因此对这个课题进行研究，分析不同算法的效率表现是一个有着实际应用意义的工作。

目前已经存在多种的约分算法，通过对这些算法比较，我们选取了几种有代表性的算法进行了本次的研究。以下为本文所涉及的一些约分算法。

2. 若干约分算法的列举

2.1. 辗转相除法

辗转相除法是一种很容易实现的约分算法，由古希腊数学家欧几里得发明，它可以求得两个自然数的最大公因数（Greatest Common Divisor, GCD）。辗转相除法的基本迭代公式是 $gcd(a,b)=\begin{cases}gcd(b,a\ mod\ b) & (b>0)\\ a & (b=0)\end{cases}$。

以 $\frac{105}{126}$ 举例，迭代过程为 $(105,126)\to(126,105\ mod\ 126)\to(126,105)\to(105,126\ mod\ 105)\to(105,21)\to(21,105\ mod\ 21)\to(21,0)$，所以这两个数的最大公因数为 21，用它约分可得 $\frac{5}{6}$。

2.2. 更相减损术

更相减损术是另一个古老的算法，它出自中国古代第一部数学专著《九章算术》。其方法描述为，先用 2 将两数约分，然后求出两数的最大公因数：比较分数的分子与分母，总是用其中的大数减去小数，直到减数与差相等为止，此时用差来约分可以得到正确结果。用数学语言表示其中求最大公因数的过程为：

$$gcd(a,b)=\begin{cases}gcd(|a-b|,min(a,b)) & (a\neq b)\\ a & (a=b)\end{cases}$$

以 $\frac{42}{60}$ 举例，先用 2 将其约分为 $\frac{21}{30}$，其后迭代过程为 $(21,30)\to(21,9)\to(12,9)\to(9,3)\to(6,3)\to(3,3)$，此时用 3 约 $\frac{21}{30}$，得 $\frac{7}{10}$。

2.3. 因数筛选法

此外还有一种很容易想到的直观的算法：**因数筛选法**。这是我们从小学课本中就熟知的算法，其基本思路就是从 2 开始枚举正整数，逐个判断此数是否为分子和分母的公约数，如果是则用这个数及它的正整数次幂约分。当这个正整数大于分子或分母其中的任意一个数时完成约分。

以 $\frac{252}{684}$ 举例，迭代过程为 $(252,684)\xrightarrow{用\ 2\ 约分}(126,342)\xrightarrow{用\ 2\ 约分}(63,171)\xrightarrow{用\ 3\ 约分}(21,57)\xrightarrow{用\ 3\ 约分}(7,19)$，即最终结果为 $\frac{7}{19}$。

2.4. 倒数分解法

倒数分解法基于我们对其他数学问题的思考并根据从中发现的规律而总结的算法，在第三章中将会详细介绍。

第二章 研究目的和方法

1. 研究目的

本项目主要的研究目的，是从理论分析与工程测试的角度，分析多种约分算法在处理不同数据集合时的时间效率差异，以此掌握科学的分析方法，提升对算法优劣的辨别能力以及编写高效算法的能力，同时学习数据可视化分析的基本方法。

2. 使用的工具

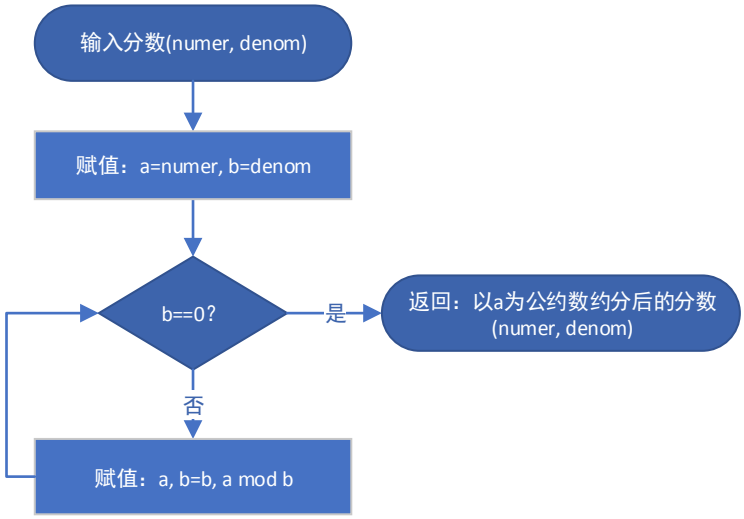
本项目使用 Python 程序语言实现算法和数据分析系统。Python 是 Guido van Rossum 首先创立的面向对象的编程语言，开发效率很高，且目前拥有许多功能强大的程序库。本项目的数据分析使用了 Matplotlib 这个第三方库，便于数据的处理和图表的渲染。

以下是研究采用的四种约分算法的编程实现，利用 Python 语言我们可以写出非常简洁的核心算法。

3. 算法核心代码

以下为本研究中所使用的四种算法的流程图与核心代码文本。这是我们根据各个算法的数学描述，编写成计算机可执行的程序代码来实现的。

3.1. 辗转相除法

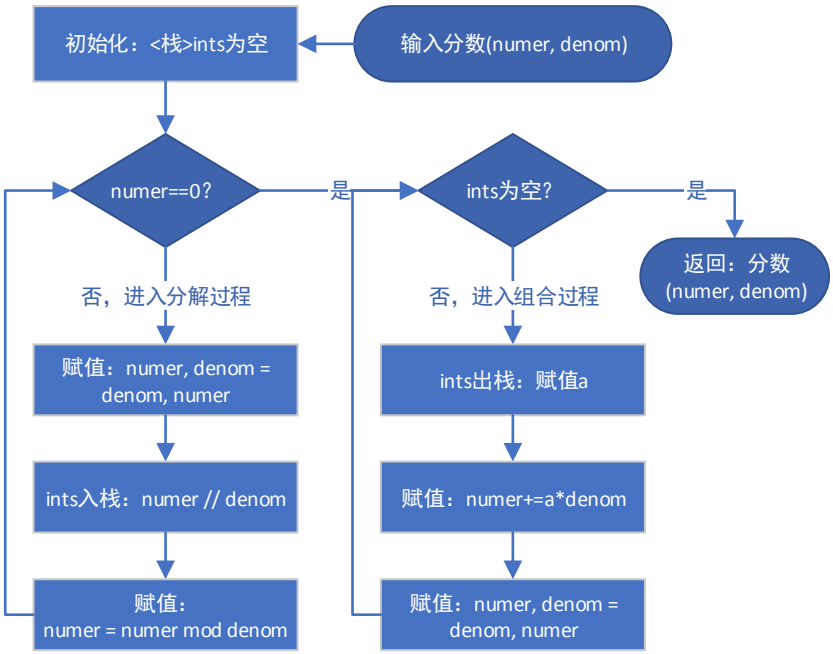


辗转相除法算法流程图

```
def Euclidean(numer:int, denom:int)->tuple:
    """
    Euclidean Algorithm
    辗转相除法
    """
    a, b = numer, denom
    while b:
        a, b = b, a % b
    return (numer // a, denom // a)
```

辗转相除法的核心代码

3.2. 倒数分解法

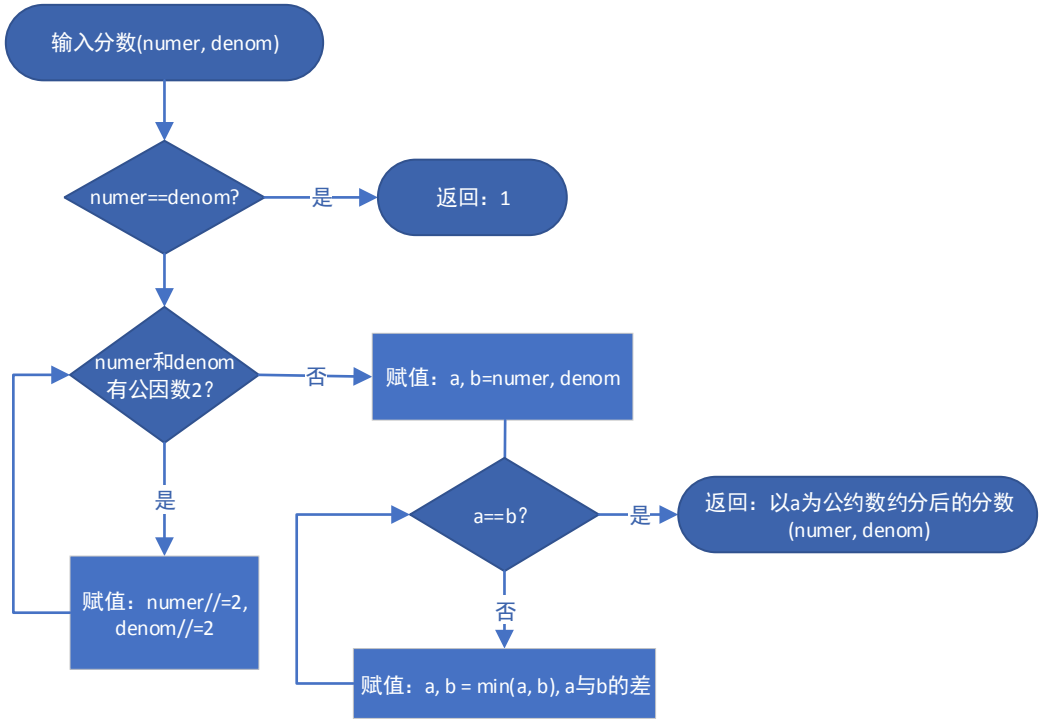


倒数分解法算法流程图

```
def RD(numer:int, denom:int) -> tuple:
    """
    Reciprocal decomposition algorithm
    倒数分解法
    """
    ints = []
    while numer:
        denom, numer = numer, denom
        ints.append(numer // denom)
        numer %= denom
    numer, denom = 1, ints.pop()
    while ints:
        numer += ints.pop() * denom
        numer, denom = denom, numer
    return (numer, denom)
```

倒数分解法的核心代码

3.3. 更相减损术

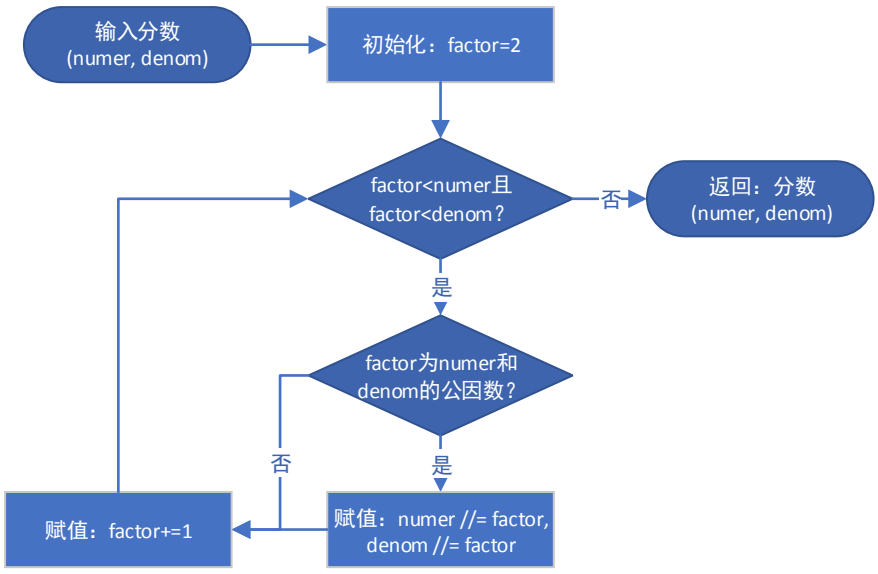


更相减损术算法流程图

```
def Subtract(numer:int, denom:int)->tuple:
    """
    更相减损术
    """
    while not (numer % 2 or denom % 2):
        numer >>= 1
        denom >>= 1
    a, b = numer, denom
    while True:
        if a == b:
            return (numer // a, denom // a)
        a, b = min(a, b), abs(a - b)
```

更相减损术的核心代码

3.4. 因数筛选法



因数筛选法流程图

```
def FS(numer:int, denom:int)->tuple:
    """
    Factor screening algorithm
    因数筛选法
    """
    factor = 2
    while factor < min(numer, denom):
        while (numer % factor == 0) and (denom % factor == 0):
            numer //= factor
            denom //= factor
        factor += 1
    return numer, denom
```

因数筛选法的核心代码

4. 理论分析方法

理论分析采用计算机科学中用于算法效率评估的**算法复杂度**的概念（即根据数学理论分析不同的算法所需的时间和内存资源），根据近似的计算得到各种算法的效率。在接下来的所有理论分析中都会使用大 O 表示法示意算法渐进的时间复杂度（算法的计算工作量）或空间复杂度。

大 0 表示法简介。简单来说，大 O 表示法就是将算法的耗时变化的通项取其对整体的影响最大的项，并去除系数，以此表示算法的渐进复杂度。例如，如果算法 f 处理规模为 n 的数据时耗时为 3n + 7，那么将系数去处并保留对整体的影响最大的项，我们就得到算法 f 的时间复杂度为 O(n)。类似地，还有 O(1)、O(n^2)、O(log n)等常见的时间复杂度。

本项目的约分算法由于分数有分子、分母 2 个变量，因此在使用大 O 表示法时会用”numer” ”denom”分别表示分子、分母的数据规模给出几种算法近似的耗时变化，便于理论分析。

5. 效率评估准则

在这个项目中，我们以对每个约分算法的计算执行时间的实际测量作为其性能的评价准则，即依据确定的算法程序逻辑，一个待约分数从输入到算法模块开始，到完成全部约分计算的逻辑过程得到最简分数结果，对这个过程中所使用的时间进行测量。

从工程的角度来看，算法执行的不同步骤例如加减法、乘除法、取模、堆栈存取、指令逻辑调度等等各个计算环节，计算机做出处理的时间花费各不相同，如果只是以其中一部分过程的复杂度来评价的话，比如说只看循环迭代的次数，对于效率的比较则是不完整的，因此我们使用前后计时的方式来做整体上的衡量。

6. 项目协助分工

整个项目分为算法的实现和效率分析两个部分。项目小组由两人构成，其中一人主要负责搜集算法资料并实现要用于计算比较的算法代码，并进行理论分析；一人编写数据分析系统和测试及绘制工具，为不同的算法进行时间效率测试，并汇总不同数据集下的时间耗用值信息。

第三章 倒数分解算法的思考起点与本项目的源起

1. 产生历程

关于优化约分算法的思考一开始是从化简带分数引起的。当一个分数是带分数时，我们可以将其转化为整数、真分数两部分，对于其中的真分数部分$\frac{a}{b}$，$a<b$，此时如果取倒数，就可以形成另一个带分数。我们发现，通过反复的迭代，能够使每次产生的真分数分子分母数值迅速减小，而经过多次的计算，最后必然会出现真分数的分母能够整除分子从而消除分数，此时如果进行一个过程相反的逆运算，就可以重新组合得到一个分数，而这个分数正是原来的分数进行完全约分后的结果。

我们发现与熟知的通过筛选公因数约分法相比，对于分子分母比较大的分数，这个新的做法可以使约分的过程大幅减化，数值计算的复杂性可以明显降低。

2. 计算举例

化简$\frac{1938}{2584}$:

$\frac{1938}{2584}=\left(1+\frac{646}{1938}\right)^{-1}=\left(1+\left(3+\frac{0}{646}\right)^{-1}\right)^{-1}=\left(1+\frac{1}{3}\right)^{-1}=\frac{3}{4}$。

3. 数学理论定义

倒数分解法运用以下事实：$\frac{a}{b}=\left\lfloor\frac{a}{b}\right\rfloor+\frac{a\ mod\ b}{b}=\left\lfloor\frac{a}{b}\right\rfloor+\left(\frac{b}{a\ mod\ b}\right)^{-1}\ (a,b\in\mathbb{Z}^{+})$。设$\frac{b}{a\ mod\ b}=\frac{a_1}{b_1}$，则可继续迭代，当$b_n|a_n$时重新组合分解出的一层层整数部分得到结果。因此

$$\mathrm{RD(a,b)}=\begin{cases}\left\lfloor\frac{a}{b}\right\rfloor+RD(b,a\ mod\ b)^{-1} & (b\neq 1)\\ \frac{a}{b} & (b=1)\end{cases}$$，其中RD(a,b)表示分数$\frac{a}{b}$的化简结果。

4. 计算机实现的思考

有了以上的分析，我们开始思考这个思路是不是可以用来做一个可以给超大整数间进行约分的工具，于是就尝试开始编写程序来代替笔算实现这个过程。与浮点计算不同，为了保证精度，约分计算过程始终使用两个整数代表当前的分数，并且需要维护一个**栈**，用于存储每一层的整数部分。栈数据结构是一种后进先出（Last-in-first-out,LIFO）的序列。算法的主要的步骤可以直接按照数学定义执行，同时对栈进行**压入**或**弹出**操作，通过有限的循环过程就可以完成这个约分的计算和正确的数学表达输出了。

在这个过程实现完成后，我们很想知道这是不是约分计算的最优方法，于是就开始了对数值约分计算这个方向的探索，并继续尝试将所发现的各种约分法都以最高效的算法程序实现。为了精确的比较不同方法之间的差异，我们开始思考如何进行算法运行的时间测量、如何确定分析和比较的原则、如何将测量数据进行简便和直观的分析，随着这一系列问题的深入，也就形成了《多种数值约分算法的效率比较研究》这个项目的开端。

5. 与前人知识的对照

我们在研究做出这个算法之前，当时只熟悉最基本的因数筛选算法，还没有对辗转相除法等其他约分算法进行很深入的理解。在项目进入到后期时，我们在查阅更多资料后发现，我们自己独立探索的这个算法，实际上是分数在转换成有限简单连分数后的逆转换过程，和辗转相除法也存在很深的联系。

前面$\frac{1938}{2584}$的例子计算为连分数的形式就是[0,1,3]，转换回到普通分数形式时为以下过程：

$\frac{1938}{2584} = [0,1,3] = \left[0,1+\frac{1}{3}\right] = \left[0,\frac{4}{3}\right] = \left[0+\frac{3}{4}\right] = \frac{3}{4}$。

连分数使用一串特定的整数数列可以用最为简洁优美的方式来表示任何一个有理数，如果我们把约分计算看做一个分数在表达形式上的求真求美过程，那么连分数的转换就是通向这种数学之美的一座赏心悦目的桥梁。

第四章 约分算法效率的理论分析

在这个部分，我们将从理论方面对综述部分所提到的四种约分算法进行效率的分析与比较。为了方便分析和表示各种算法的效率，此处使用的大 O 表示法统一把四则运算、取模运算看作时间复杂度为 O(1)的基础运算，并且把数组的每个元素的空间复杂度看作 O(1)。

1. 更相减损术

更相减损术采用求差的方法进行迭代，因此执行次数和分子、分母的差距有关。它的平均时间复杂度是$O(max(numer, denom) - min(numer, denom))$。据此分析：

(i) 当分子、分母数量级差距极大时，时间复杂度趋近于$O(max(numer, denom))$。

(ii) 当差距极小时，按照算法逻辑，如果两数都比较大，再次迭代后分子、分母数量级差巨大，会和情况(i)类似，时间复杂度为$O(min(numer, denom))$。

(iii) 分子、分母其一接近另外一数的一半时迭代次数较少。并且如果分子、分母足够大，迭代的前几次时间复杂度接近$O(log(max(numer, denom)))$（因为每次的迭代大致会使较大数减半），而后变为情况(i)。

由上可总结，更相减损术的最坏情况下时间复杂度为$O(max(numer, denom))$。

2. 辗转相除法

由于当$a > b$时，$a \bmod b < \frac{a}{2}$，根据辗转相除法的定义，每次迭代（除去第一次，因为首次不能保证a > b）至少会使 a 和 b 减半。因此它在最坏情况下时间复杂度为$O(log(max(numer, denom)))$。如果分子、分母之间有较大差距，或差距很小时，每次迭代会使结果减少不止一半；而如果差距较小，导致每次的取模运算效率退化成减法，例如取分子、分母为斐波那契数列的相邻两项时，会使效率下降。

3. 倒数分解法

事实上我们最初思考的这个方法和辗转相除法很接近。它们的迭代计算的部分（即**状态转移方程**）都为$(a, b) \to (b, a \bmod b)$，因此它们的时间复杂度相同，但是由于倒数分解法还有一个重组整数部分的过程，它的计算量会大一倍；并且为了存储整数部分还消耗一定的内存。由于整数部分和迭代次数增长率相同，所以空间复杂度与时间复杂度一致，最坏情况下也为$O(log(max(numer, denom)))$。

4. 因数筛选法

因数筛选法由于要枚举从 2 开始到分子、分母中较小数范围内的所有的自然数，表面上可以看出最坏时间复杂度为$O(min(numer, denom))$。但是事实上，对于任意公因数 m，在算法步骤执行时会同时消除 m 的任意次幂，这会使枚举的范围变得更小。因此如果分子、分母有$\begin{cases} numer = a^l \cdot b^m \cdot \cdots \\ denom = a^p \cdot b^q \cdot \ldots \end{cases}$的形式且公因数的指数大时，因数筛选法效率会很高；但是如果分子、分母互质，则它们越大，该算法的计算次数越多。

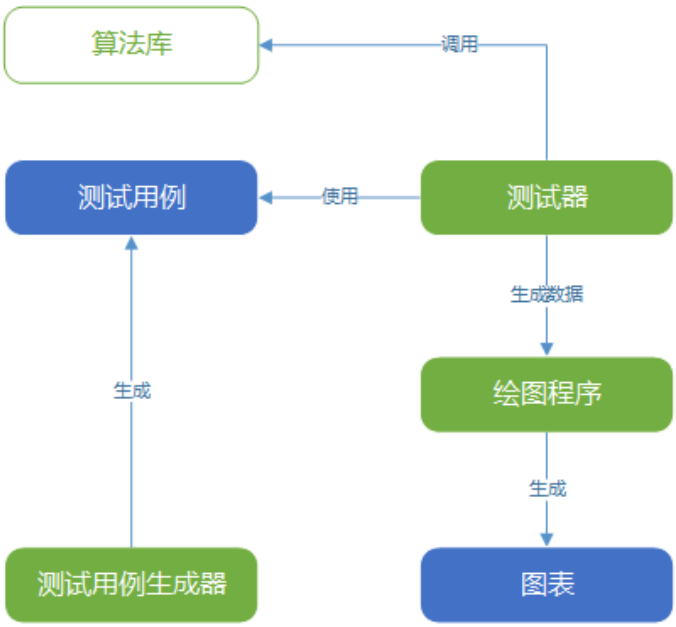
从以上的分析中，我们可以看出，对于综合的数据集，辗转相除法和倒数分解法的效率是较高的，比较稳定，然而后者在内存空间使用上仍会有少量开销；更相减损术在分子、分母比例不同的情况下计算次数有明显的优劣对比；因数筛选法在处理公因数重复较多的两数时，效率是很高的，反之在两数很大时则很低。

第五章 数据测试的方法与工具设计

我们为这个项目的数据测试专门编写了相关的测试器程序，并嵌入了前面所述的四种核心算法代码，通过这一工具，我们能够以足够高效的进行数据测试与效率分析评估，从而从实证的角度来验证我们对于各种算法其相对效率的理论预测。以下是关于测试器的整体结构和测试原理的说明。

1. 测试器主要组成部分

算法测试器大致可分为两部分：测试用例生成部分和测试结果处理部分。在最开始的时候，测试结果处理部分的程序没有非常清晰的划分，在测试完成之后立即绘制出对应的图表。后面的 0.8 版本对此进行了改进，将测试结果处理部分的模块一分为二，一部分只负责测试和生成测试结果，另外一部分则只负责导入结果并画出图表。这个改进使得测试器变得更灵活了。



这个框架图表示了我们测试工具的模块结构以及数据调用的方式。下面将会详细介绍**用例生成器**、**算法测试器**和**绘图分析器**。

2. 用例生成器

用例生成器调用我们根据用例集的特定规则制作好的生成函数，用以生成不同的测试用例数据文件。不同的测试用例集合是根据不同算法的运算方式特点选取的，这是为了更好地揭示不同算法之间的差别。

2.1. 测试用例数据的设计

根据多个算法的特性，我们设计并制作出了几个能够明显体现算法之间差异的用例生成函数，如下是它们的名称和数据范围描述（详细生成操作过程见后文）：

生成函数名	数据范围描述
fib(N)	相邻的斐波那契数列 N 项作为分子和分母
prime(N)	相邻的质数数列 N 项作为分子和分母
magdif(N)	取 N 对分子和分母有10^{4}~10^{6}的倍数差距
mcomfact(N)	取 4~5 个质数乘积为 base，令 N 项分子分母均有$base^{2}$的倍数差距
nnatural(N)	相邻的自然数列 N 项作为分子和分母
anatural(N)	取分子、分母均为 1~N 自然数，交织成二维网格
afib(N)	取分子、分母均为 1~N 项斐波那契数列，交织成二维网格
aprime(N)	取分子、分母均为 1~N 项质数数列，交织成二维网格

这里选取的测试用例不仅仅是为算法定制的，也是为了适应绘图分析器不同的数据渲染和一些开发预留所准备的。以下为其中两个比较主要的生成函数的代码。

```
def fib(N):  # 分母、分子为 fib 数列相邻数
    cache = [1]*(N + 1)
    for i in range(2, N + 1):
        cache[i] = cache[i - 1] + cache[i - 2]
    return [cache[:-1], cache[1:]]
```

相邻的斐波那契数对生成函数代码

```
PRIME_INF = 10000000
def prime(N):  # 分母、分子为质数数列相邻数
    l = []
    primes = []
    count = 0  # 用以质数计量，适时停止
    for apri in range(2, PRIME_INF):
        if count == N+1:
            break
        for b in range(2, int(math.sqrt(apri) + 1)):
            l.append(apri % b)
        if 0 not in l:
            primes.append(apri)
            count += 1
        l = []
return [primes[:-1], primes[1:]]
```

相邻质数数对生成函数代码

2.2. 用例生成器的结构

即便是用例生成器这样较为简单的程序，也经过几次修整。在 0.9 版本的用例生成器中，定义了专门方便输出用例的主生成器接口，让该程序的结构更整齐，也更加便于让开发者使用。用例生成器是通过应用设定的规则，自动生成一系列的测试用分子/分母组合，并把准确的用例以表格的形式写入到一个数据文件中，让后续的测试器可以逐个读取并加载算法进行计算测试。

主生成器函数 console()的代码如下：

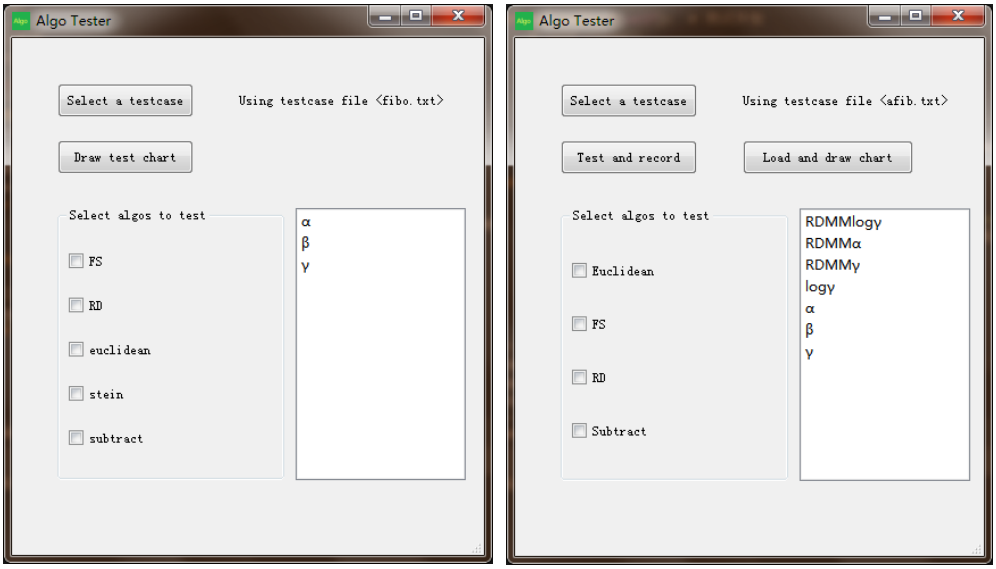
```
def console():
    while True:
        cmd = input(">>> ")
        funcname = cmd.split('(')[0]
        try:
            if funcname == "_all":
                eval(cmd)
                print("successfully wrote testcase");   continue
            result = eval(cmd)
            _write(funcname, result)
            print("successfully wrote testcase")
        except Exception as e:
            print(e)
```

3. 算法测试器

算法测试器可以算是整个数据测试程序最关键的模块。在这个模块中，我们将用例生成器生成好的**测试用例**以及算法库中等待测试的**算法**一并导入；这些导入的算法经过了时间测量函数的调用和干扰抑制算法的结果处理，最终才被测出**相对准确**的运行时间。

3.1. 算法测试器的交互界面

为了让操作更加简便，算法测试器的用户交互界面采用了 GUI（Graphical User Interface，图形化用户界面），本测试器的 GUI 使用 PyQt5 编写。下面分别是 0.6 版本和 0.9 版本的测试器 GUI 外观：



主程序 main.py 的 GUI 控制面板集成了算法测试器和绘图分析器的功能。

3.2. 算法测试器的后端结构

0.9 版本中，算法测试器的模块是 `record.py`。这个模块只负责读入算法和数据进行速度测试，然后按照一定的格式输出结果到`./Results` 路径之下的结果库。

算法测试器的后端结构中其主要函数有两个：`MainTester` 和 `test`。`MainTester` 负责输入用例并调用 `test` 函数，`test` 函数执行具体的计时算法。`MainTester` 的代码如下：

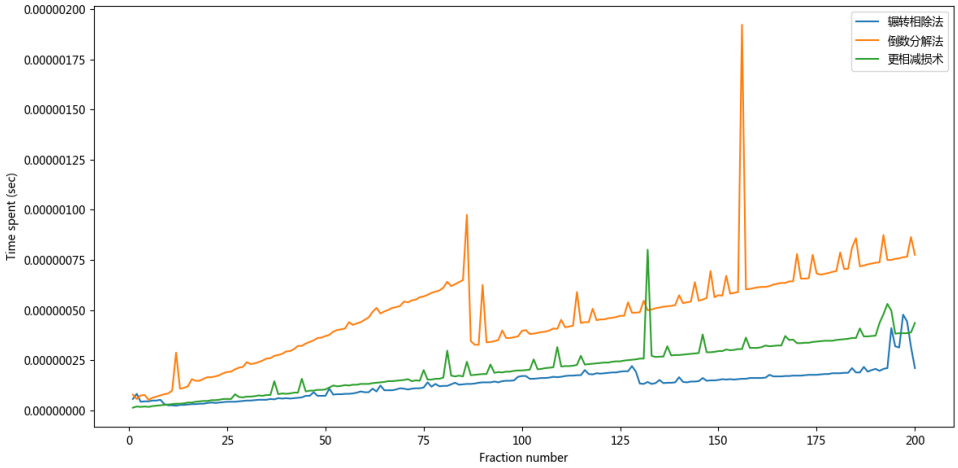
```
def MainTester(testlist:list, filename:str) -> None:
# （准备测试的算法，测试用例路径名）
    print("Starting tests in record.py...")
    casename = filename.split('/')[-1]
    print("Using testcase %s..." % casename)
    with open(filename, 'r') as File:
        case = makeCase(File)
    test_result = test(testlist, case)  # 测试结果
    outputResult(testlist, case, test_result, casename.split('.')[0])
    print("Successfully ended tests in record.py.")
```

此外，针对某些算法执行测试的时间较久的情况，我们让 `MainTester` 和 `test` 函数在控制台输出进度提示，虽然这并不是最好的进度提示方式，但是也能够保证测试期间算法执行的正确性。

3.3. 测量中的噪音影响与干扰抑制算法的思考

在算法测试器的开发过程中，我们遇到了不少因为计算机测试平台自身能力所限引发的问题，但是这些问题最终都化解了。其中特别重要的是**算法测速时的干扰和误差问题**。

在 0.2 版本中，我们正式开始编写测试器的代码，此时对算法库中的算法均执行单次测速。测试器使用 Python 内置模块 **`timeit`** 作为测速工具，该模块的精准性较高。但是，单次测速虽然测试得很快，却也面临着一个问题——由于 Windows 平台是一个多任务多线程的操作系统，计算机 CPU 的时间分配因为后台程序的影响存在一定程度的波动，因此测试的速度数值也跟着波动，最终绘制出的图表于是就十分不精准，显示不出算法执行真实的情况。这该如何处理呢？



直接测量记录每次约分计算时间时的数据结果
（以斐波拉契相邻数为基准测试数据集合）

如上图所示，无论是哪种算法，在进行约分计算的过程中，所记录到的时间数据都存在大量的杂峰，而且多次运行时产生的杂峰情况还不一样，也没有规律性，除了杂峰，还对曲线的斜率趋势产生了不确定的影响。在这种情况下，是完全没有办法进行精确的量化评估的。

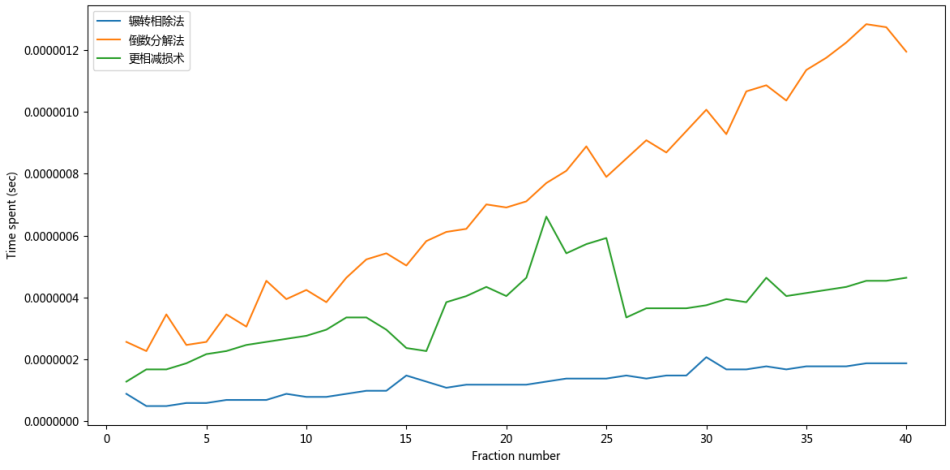
经过仔细的思考，我们采取一种方法来对这种干扰信号进行抑制。我们取了三个常量——NUMBER、REPEAT 和 BALANCE，在测试算法的一些关键位置加上必要的重复测量循环，并对于测得的数据进行评估，将绝大部分被噪音干扰的数据予以抛弃。下面是具体的设计：

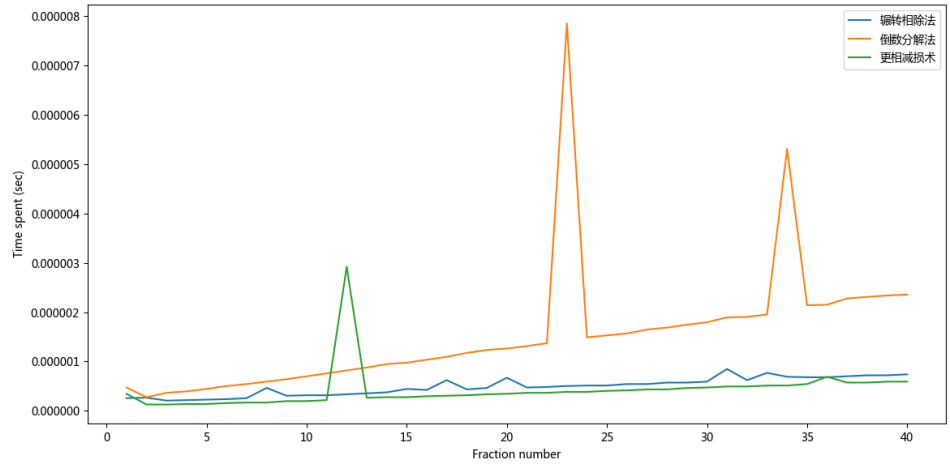
- 将一算法执行 NUMBER 次并取其执行总速度。
- 执行以上步骤 REPEAT 次，并且只保留 REPEAT 个结果中的最小值。
- 执行以上步骤 BALANCE 次并得到结果，将这 BALANCE 组结果用一个 average 函数取得平均值，再将这些结果除以 NUMBER 得到单次算法的执行速度。

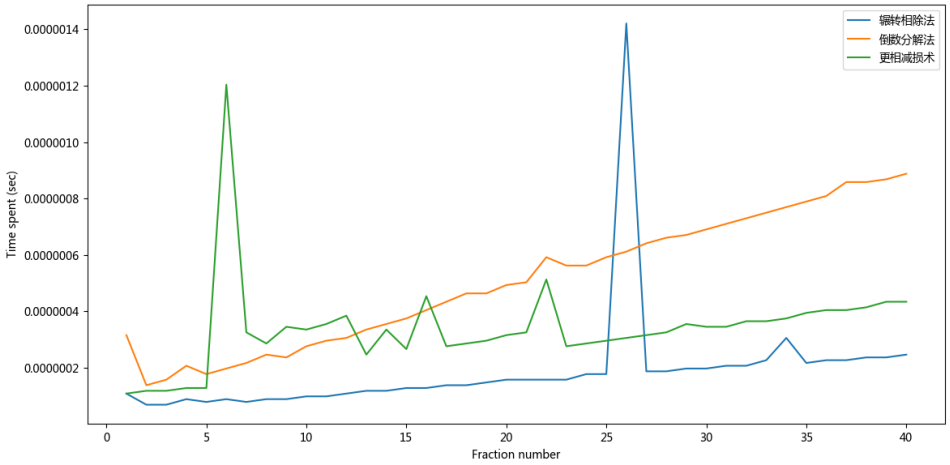
实验证明，这种方法能够非常好地减少因为 CPU 分时波动带来的执行速度误差，从而使图表曲线更加稳定、平滑。当然，这样做需要付出时间上的代价，其耗时为单次测速的 NUMBER*REPEAT*BALANCE 倍。对于一般的算法，取 NUMBER=50，REPEAT=10，BALANCE=10 是较为合适的。

下面是检验此矫正方法的几组数据，它们用的都是同一组测试用例和同一组被测算法。其中涉及到上述三个常量的取值，我们对于相应的测试结果加上对应 “NUMBER-REPEAT-BALANCE”的标记。

以下为作为对照的 1-1-1 测试模式（即单次运行，不做干扰抑制处理）的数据结果，数据图的横坐标为从小到大的测试数据项的编号，纵坐标是每次测量的算法完成时间：

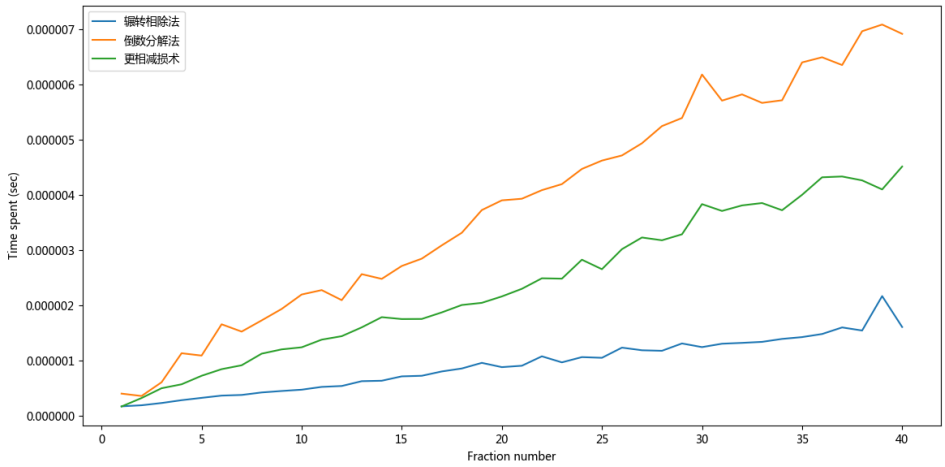


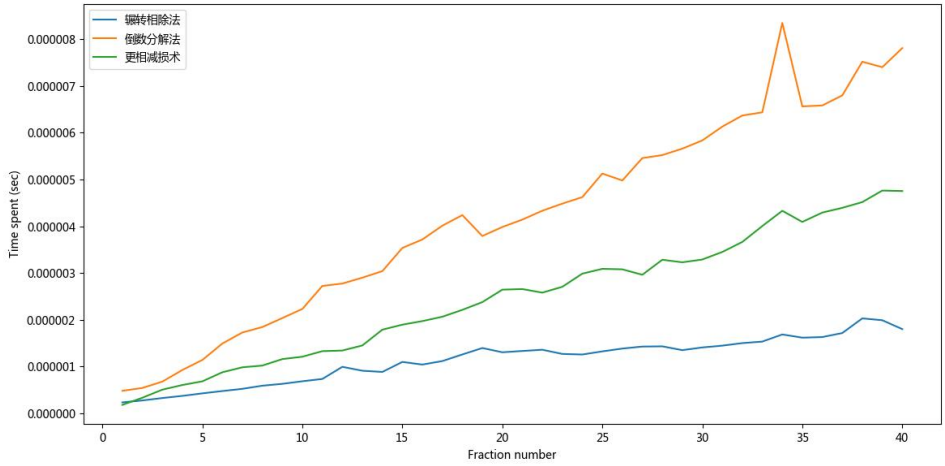


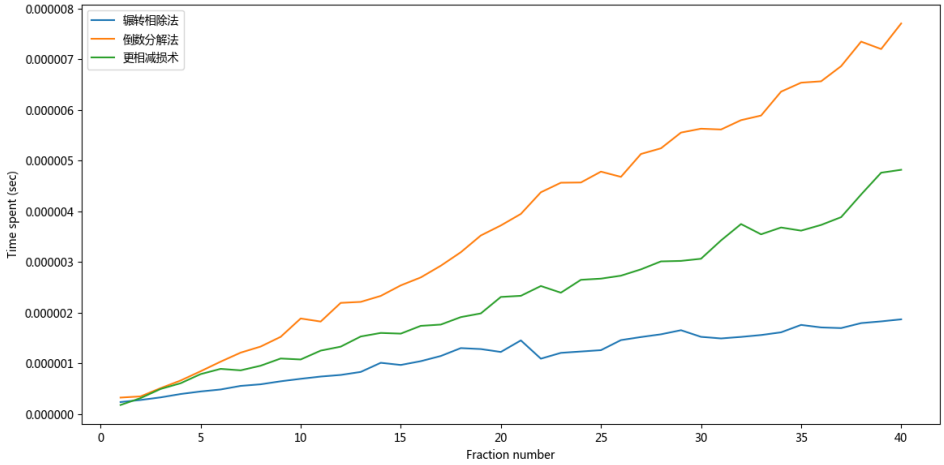


我们可以看到，未进行干扰抑制的这三组测试结果中，曲线非常不平滑，出现了大量的尖峰，每个尖峰都意味着了测试程序因为操作系统后台的并行线程启动而发生了 CPU 分时损失，从而产生了计算周期的滞后，因此导致延迟尖峰的出现。而且因为操作系统的波动是随机发生的，我们在三次测量取样中观察到的尖峰位置和峰高都各不相同。

50-1-10 测试模式：

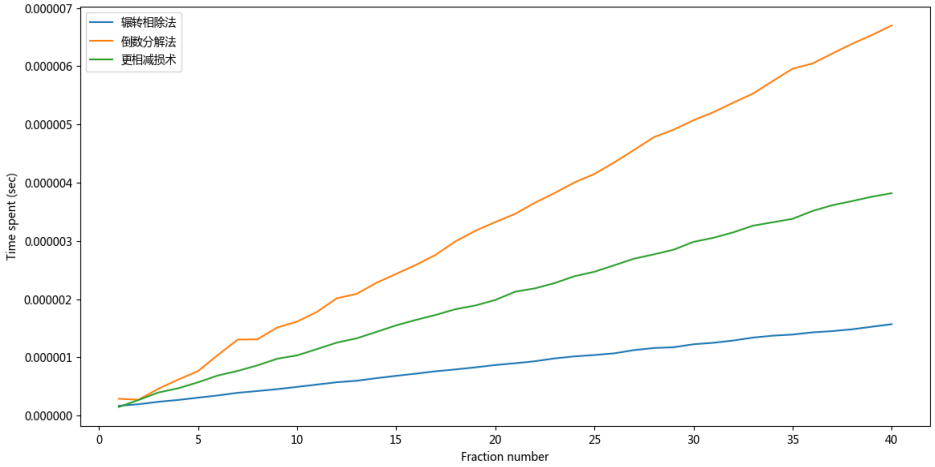


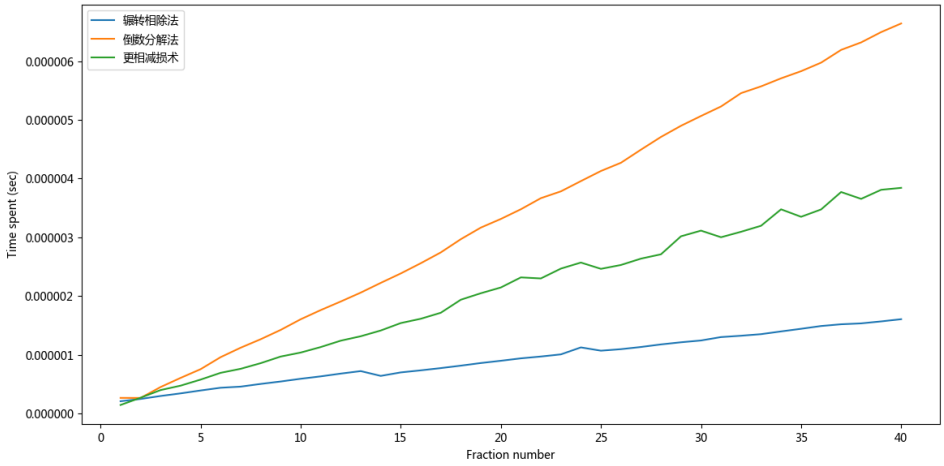


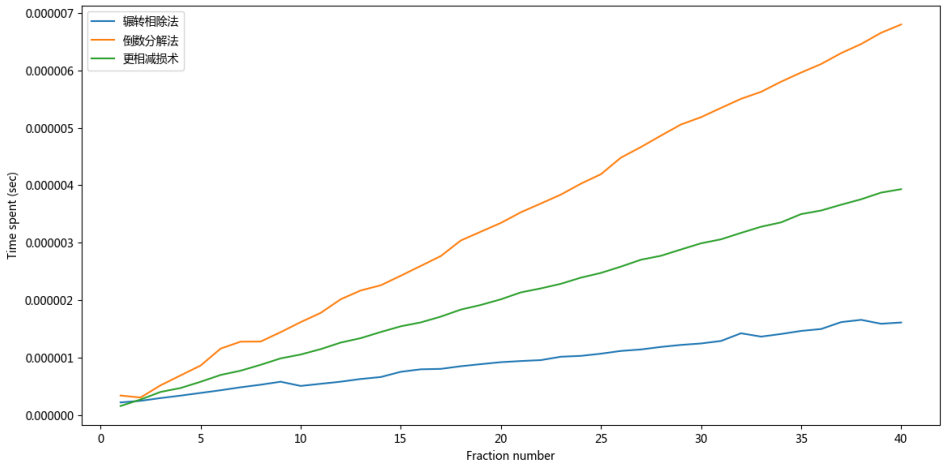


在 50-1-10 测试的这三张数据图上，我们可以观察到曲线的波动性有着明显的降低，曲线比上一个测试要平滑很多。

50-10-10 测试模式：







在 50-10-10 的数据图上我们看到，随着重复次数的增加，算法速度的波动也越来越小了。通过这种方法，我们找到了一个可以消除操作系统波动性影响，并以较高精度保障测量准确性和重现性的良好方法。精度的提高是以测量效率的损失为代价的，为了在精确度和测试效率间保持平衡，我们最后选择 50-10-10 的参数作为后面所有测试的基准模式。

4. 绘图分析器

绘图分析器中有不同方式、不同类型的绘图分析函数，它们将同样的.csv 逗号分隔赋值类型的结果库文件解析成不同的含义。绘图分析是最后一个环节，也是让数据可视化，进而分析计算速度规律的重要环节。

4.1. 基本分析函数

基本分析函数是指诸多分析函数中的原型，目前我们有 α 和 β 这两个基本分析函数，γ 分析函数是 β 的变种，但是仍是其他变种函数的原型，因此 α 、 β 和 γ 分析函数都属于基本分析函数。目前开发出的这三个基本分析函数都是二维分析图表。下面是它们的详细信息和适用范围：

分析函数名	X 坐标含义	Y 坐标含义	其他维度	图表类型	适用范围
α	测试用例序号	测试速度	无	折线图	多个算法速度比较
β	分母数值	分子数值	散点颜色蓝→红表示速度快→慢	散点图	单个算法相对速度分析
γ	分母数值	分子数值	同 β，同时散点小→大表示速度快→慢	散点图	单个算法相对速度分析

4.2. 变种分析函数

变种分析函数都是为某一特定测试用例集合或某一算法的特点定制的，我们用拉丁字母和希腊字母拼接表示变种分析函数的名称，以标记它们的用途。

目前，我们有这些变种分析函数：

分析函数名	源于函数	变种用途	变动
logγ	γ	方便显示 fib 等指数级测试用例的测试结果散点图	将 X、Y 坐标改为对数坐标形式
RDMMα	α	为 RDMM 内存监视器定制的 α	修改 Y 坐标意义为 RDMM 耗用内存长度
RDMMγ	γ	为 RDMM 内存监视器定制的 γ	修改散点颜色和大小意义为 RDMM 相对耗用内存长度
RDMMlogγ	γ	为 RDMM 内存监视器定制的 log γ	RDMM γ 与 log γ 的变动

以上的这些分析绘制函数是为了用来尝试并优化不同的绘图方式，其中效果比较好的就留作后面分析的主要绘制手段了，有一些是为以后的其他测试而预留的。通过测量和分析工具的准备，我们终于可以进入到正式的算法效率对比阶段。

第六章 测试用例集合与数据分析

数据分析将会使用多种测试用例，并用绘图工具渲染直观的图表。以下数据测试和分析以用例数据集分类，比较多种算法之间的效率差异。

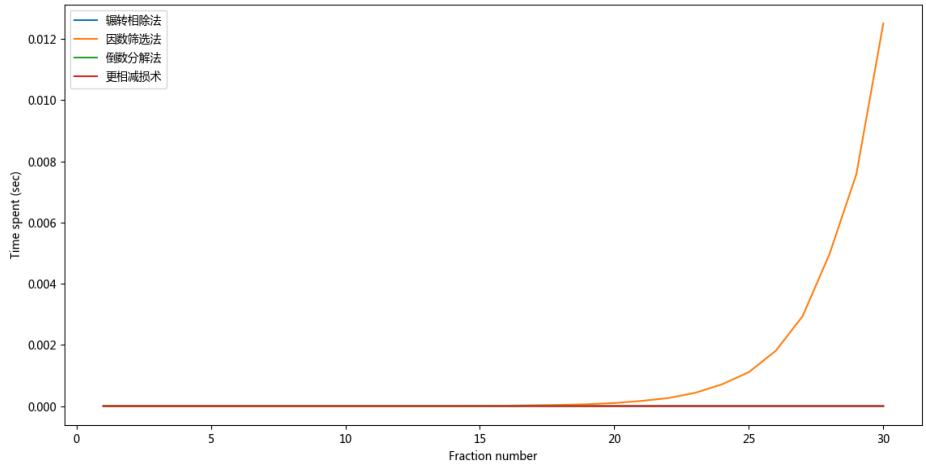
1. 用例集合一：以斐波那契数列的相邻两项作为分子和分母

1.1. 前 30 对斐波那契相邻数

我们生成了一串斐波那契数列，并且用相邻数两两组合的方式产生了 30 对斐波那契相邻数作为测试用例的分子和分母。

用例分子	用例分母
1	1
1	2
…	…
2584	4181
4181	6765
…	…
514229	832040
832040	1346269

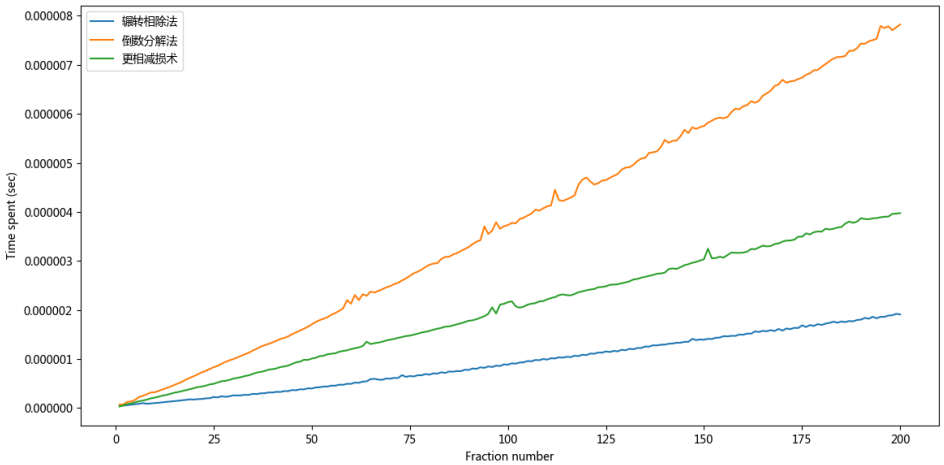
四种约分算法的耗时变化表现如下：



从图中我们可以看出，因数筛选法的耗时随着斐波那契数的增加快速增长，与其他三个算法出现巨大差异，这是斐波那契数增长速度导致的。因数筛选的时间耗用随着数列的数值增加而呈现出耗时接近与指数增长的情况，很快就超过了笔记本的计算支持能力，因此接下来的更大范围测试中我们移除了因数筛选算法，对剩下的三种算法继续进行比较。

1.2. 前 200 对斐波那契相邻数

以下是对于 200 对斐波那契相邻数（从 1, 1 开始），其余三种算法的耗时增长图：从图中，我们可以看出，它们的耗时都大致是线性增长；辗转相除法和倒数分解法并未表现



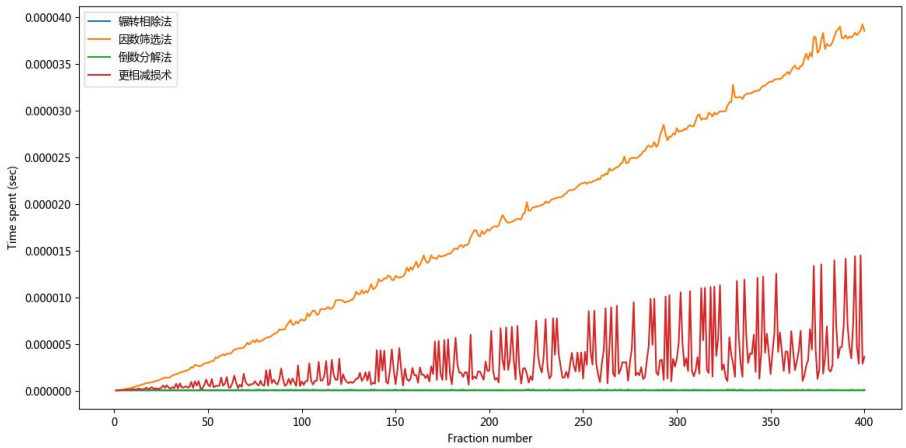
出优势，这是因为对于斐波那契数列，这两种算法的取模运算的效率不能最大化，因此出现了效率瓶颈；而更相减损术采用求差运算，此时相对而言性能更高。

2. 用例集合二：以质数数列中相邻两项作为分子和分母

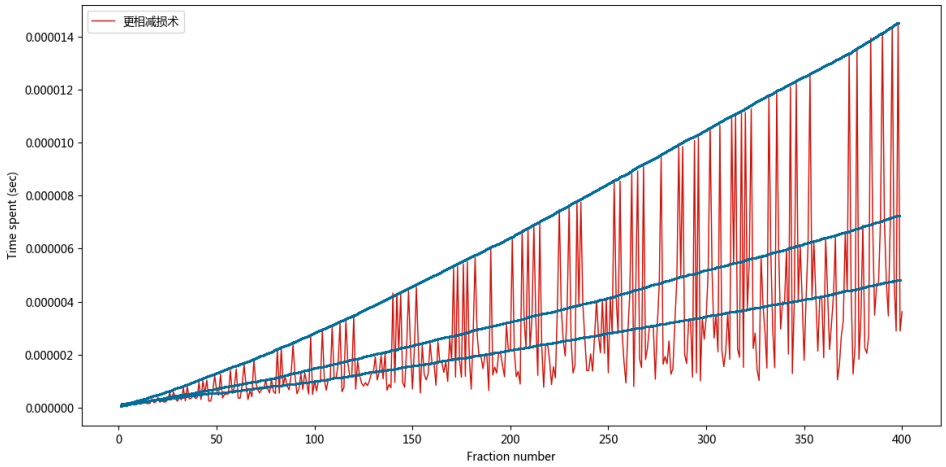
前 400 对相邻质数的用例表：

用例分子	用例分母
2	3
3	5
…	…
1223	1229
1229	1231
…	…
2731	2741
2741	2749

四种算法的耗时增长图如下，我们发现因数筛选法的耗时呈线性增长，这是因为质数对组成的分数无法化简，因此算法必定要枚举从 2 到较小质数中的所有数，并试除。



我们对这个数据进行更进一步的分析，发现更相减损术的耗时出现震荡的情况，我们把这个算法的数据单独放大来看，发现震荡的范围有一些分布在几条类似线性函数分割开的区间中，如下图所示。



更相减损术震荡范围的规律——震荡区间的分割情况

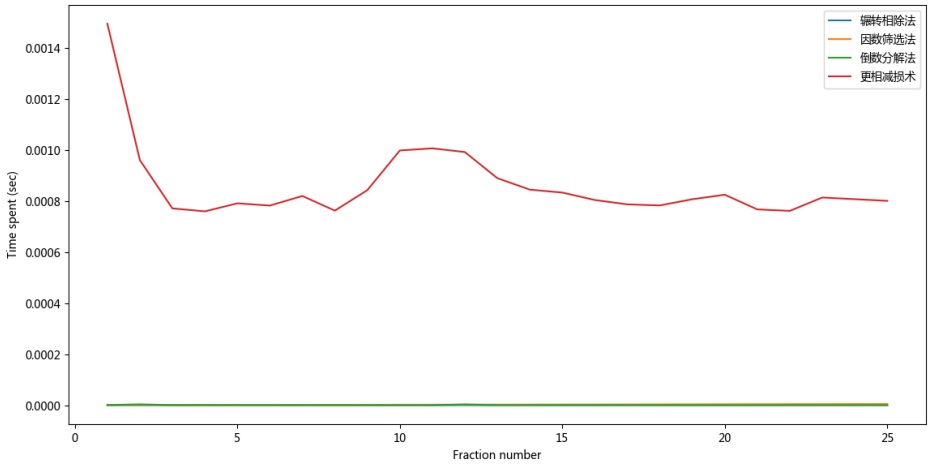
我们推测，随着用例的增加，这样的震荡区间会变得越来越多和更加细分，震荡区间的产生原因跟质数列的空间分布情况有关，越相邻的质数约分计算耗时越长，而距离较远的质数对则算法完成耗时较短。我们查看了图中的部分数据证实了这一判断，最上面一条渐近线上的测试数据全部都是孪生质数对，第二条则是相差为 4 的相邻质数对，第三条为相差 6 的相邻质数对。在图的最下方，辗转相除法和倒数分解法始终没有太大的效率变化，这是因为它们不采用枚举因数的方法，对于相邻质数，它们的性能由于取模计算后的优势而变得较强。

3. 用例集合三：取分子和分母，使其公因数幂次较大

公因数的幂次较大的数据集的生成法则如下：取质数集里的n个最小质数，把它们相乘得到$base$值。生成多组用例：case1：取分子为$base^3$，分母为$base^5$。case2：取分子为$base^4$，分母为$base^6$。以此类推。

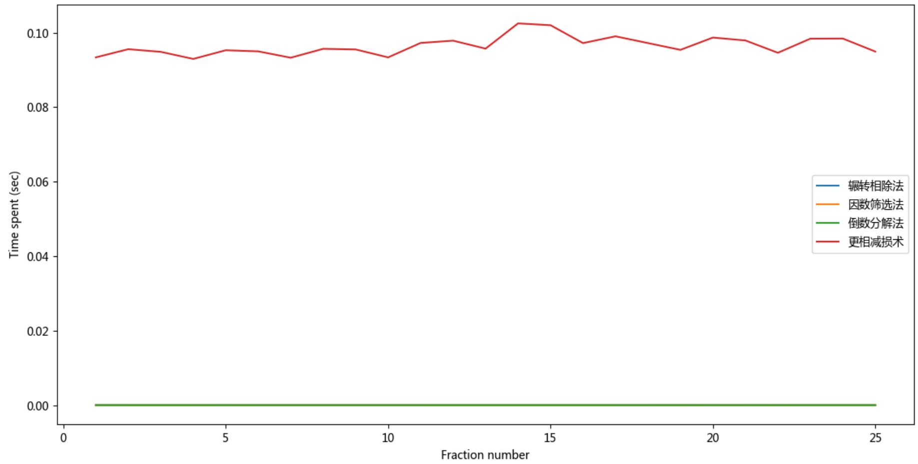
3.1. 取n = 4，生成 25 组测试用例时四种算法的耗时变化图

用例分子	用例分母
9261000	408410100000
1944810000	85766121000000
...	...
3004194249508169189474100000000000000000	1324849664033102612558078100000000000000000000
6308807923967155297895610000000000000000000	2782184294469515486371964010000000000000000000
...	...
2386171548437720960155517162893052100000000000000000000	1052301652861034943428583068835835976100000000000000000000
5010960251719214016326586042075409410000000000000000000000	2209833471008173381200024444555255549810000000000000000000000



3.2. 取n = 5，生成 25 组测试用例时四种算法的耗时变化图

用例分子	用例分母
12326391000	65774855015100000
28473963210000	151939915084881000000
...	...
1518461056448570054194301417893336895831100000000000000000	8102660043315214666186211796020635009844332710000000000000000000
3507645040396196825188836275333608229369841000000000000000000	1871714470005814587889014924880766687274040856010000000000000000000
...	...
2843881376358265139181805051547225514380537065511945010489848100000000000000000000000000000000	1517523541238533860918802993556115006728598383527828977047487844641000000000000000000000000000000000
6569365979387592471509969669074090938219040621332592974231549111000000000000000000000000000000000	3505479380261013218722434915114625665543062265949284936979696921120710000000000000000000000000000000000

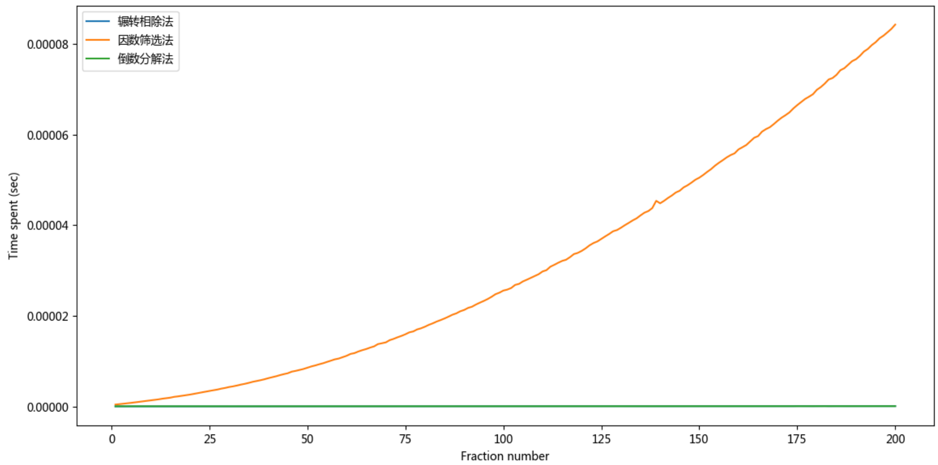


此用例取的分母是分子的$base^2$倍，由理论分析和上图数据可得，更相减损术对于相差悬殊的分子、分母效率低下。由于随着用例范围的扩大，更相减损术算法超出了我们的笔记本算力，我们去除了这一算法，使用其余 3 种算法继续进行测试分析。

3.3. 取$n = 4$，生成 200 组测试用例时四种算法的耗时变化图

用例分子	用例分母
9261000	408410100000
1944810000	85766121000000
...	...
3500650617232308454311541426906609711251331 4024407971300958674856604793610476885199293 6943572809650339989192789470204652553334672 2202100000000000000000000000000000000000000 0000000000000000000000000000000000000000000 00000000000000000000	1543786922199448028351389769265814882661837 1484763915343722775611762713982220306372888 5192115609055799935234020156360251776020590 4491126100000000000000000000000000000000000 0000000000000000000000000000000000000000000 0000000000000000000000000
7351366296187847754054236996503880393627795 9451256739732013217198870066582001458918516 7581502900265713977304857887429770362002811 6624410000000000000000000000000000000000000 0000000000000000000000000000000000000000000 0000000000000000000000	3241952536618840859537918515458211253589858 0118004222221817828784701699362662643383065 8903442779017179863991442328356528729643239 9431364810000000000000000000000000000000000 0000000000000000000000000000000000000000000 000000000000000000000000000
...	...
5835502259013781979473330871708564240343164 4293024173368231045756480006136194698131623 1745732940406604037637532619357520019705771 2663062370616738537879553015188992327791990 7366591483096081636011392074685233822708732 2663554325925832919061344853500382451844001 3844402100000000000000000000000000000000000 0000000000000000000000000000000000000000000 0000000000000000000000000000000000000000000	2573456496225077852947738914423476829991335 5133223660455389891178607682706061861876045 8199868226719312380598151885136666328690245 1284410505441981695204882879698345616556267 9148666844045372001481023904936188115814550 9294627457733292317306053080393668661263204 6105381326100000000000000000000000000000000 0000000000000000000000000000000000000000000 0000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000 0000000000000000000000000000000000000	0000000000000000000000000000000000000000000 000000000000000000000000000000000000000000
5404258642072663491190251720289301342981804 5779769686956318771475076133682729909939696 2219723276110555999256118958786999290249514 7697262061428161559930254047366525794768162 6212200372495281203110150200365995043210556 9518717661239913866342711468826704188652729 6821300784810000000000000000000000000000000 0000000000000000000000000000000000000000000 0000000000000000000000000000000000000000000 0000000000000000000000000000000000000000000 0000000000000000000000000000000000000000000 0	5404258642072663491190251720289301342981804 5779769686956318771475076133682729909939696 2219723276110555999256118958786999290249514 7697262061428161559930254047366525794768162 6212200372495281203110150200365995043210556 9518717661239913866342711468826704188652729 6821300784810000000000000000000000000000000 0000000000000000000000000000000000000000000 0000000000000000000000000000000000000000000 0000000000000000000000000000000000000000000 0000000000000000000000000000000000000000000 0

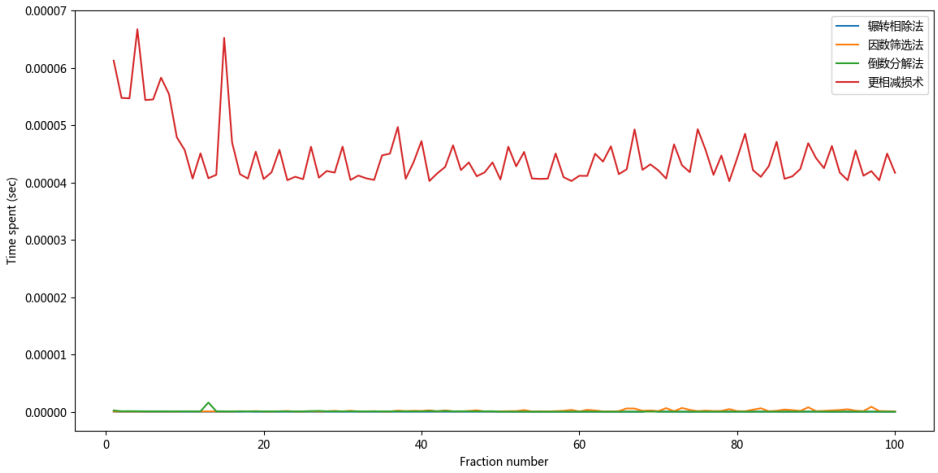


从图中可知，因数筛选法虽然对于重复的因数可以提前去除，缩小因数的枚举范围，但是它的效率仍没有辗转相除法和因数筛选法的效率高。后两者的耗时几乎看不出增长，是因为它们采用取模的方法，对于分子和分母呈一定的比例的情况，能够使效率控制得很高。

4. 用例集合四：取分子和分母，使其数量级差距大

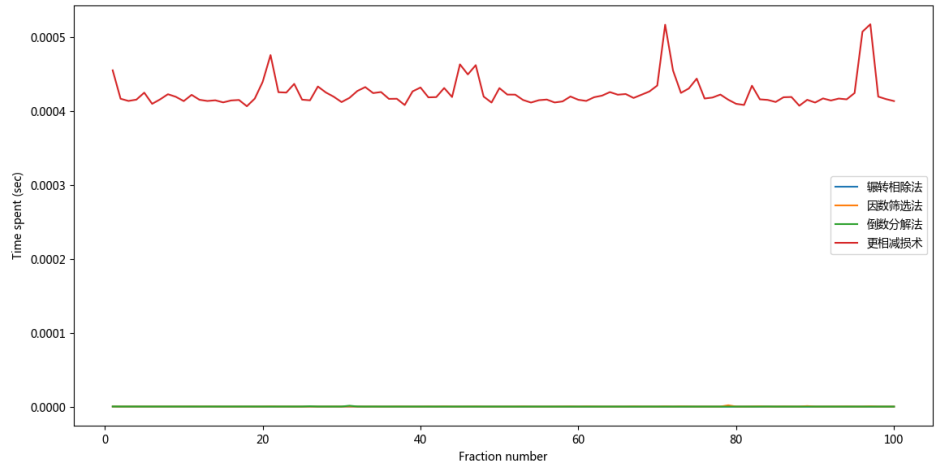
4.1. 当分母为分子的10^4倍时，四个算法的耗时增长图如下：

用例分子	用例分母
10	10000
20	20000
…	…
490	490000
500	500000
…	…
990	990000
1000	1000000



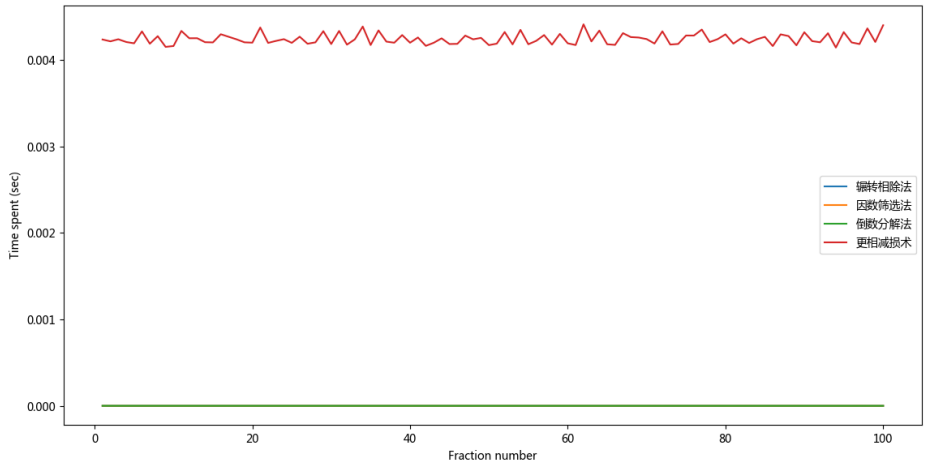
当分母为分子的10^5倍时，四个算法的耗时增长图如下：

用例分子	用例分母
10	100000
20	200000
...	...
490	4900000
500	5000000
...	...
990	9900000
1000	10000000



4.2. 当分母为分子的10^6倍时，四个算法的耗时增长图如下：

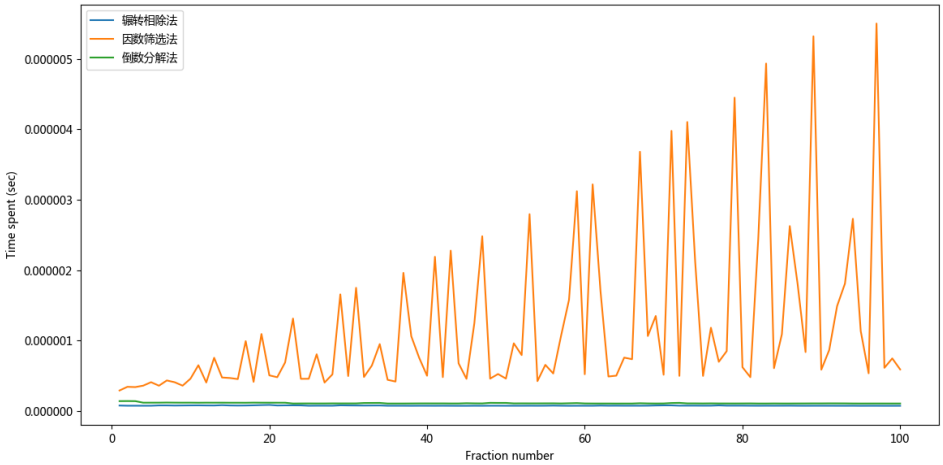
用例分子	用例分母
10	10000000
20	20000000
...	...
490	490000000
500	500000000
...	...
990	990000000
1000	1000000000



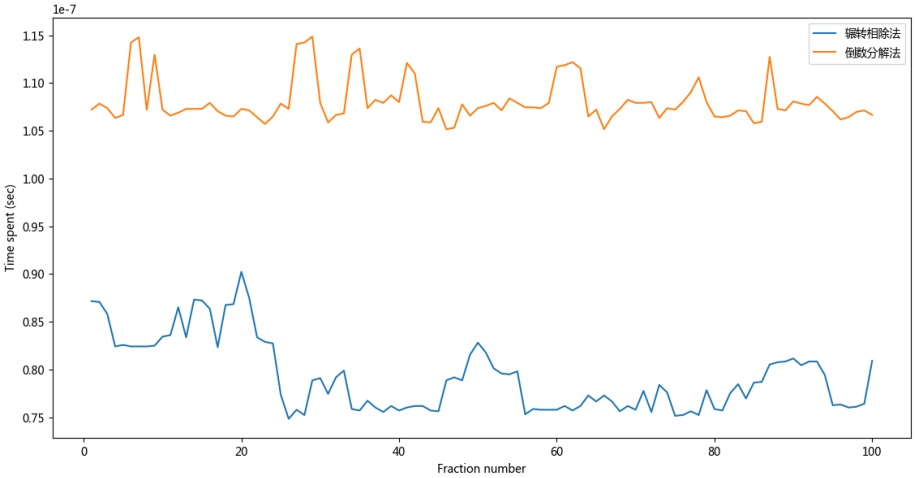
我们所取的用例为分子分母相差比较远，有 6 个数量级的差距。可以看出，更相减损术耗时持续上升，这是因为对于这类测试数据，它所要做的减法次数基本上是一样的并且相差之大使其计算次数变得非常多。

除了更相减损术算法外，其余 3 个算法曲线基本重叠，我们对于同样的数据集，去除更相减损术算法后，余下 3 种算法曲线放大后得到了更清晰的信号曲线。

最明显地，因数筛选法的耗时出现大范围震荡，并且范围逐渐变宽，这是因为测试用例的分子、分母中较小数的增长导致，并且由于因数的个数是不定的，因此导致枚举的范围时大时小。



以下是在相同的数据测试配置下，只测试辗转相除法和倒数分解法所得到的计算耗时曲线。



这两个算法是对于该测试用例最快的两个算法，因为测试的精度有限，所以两个算法的图像在纵坐标比例尺放大到一定程度后，可以看到在细微尺度下有明显的波动存在，但是我们仍然可以看出，它们的耗时在这几个有限的用例中仍然稳定在一定的范围内，此外倒数分解法的耗时大致是辗转相除法的耗时的 1-2 倍。原因在前面理论分析章节已有提及。

5. 结论

通过数据分析，在我们测试所覆盖的数据集合范围内，我们可以得到以下结论：

倒数分解法和辗转相除法在处理不同的测试数据集合时，其耗时情况接近且变化趋势一致，因为倒数分解法在正向分解到最后消除分数部分后，还需要多一个逆向倒数计算组合回来的过程，所以倒数分解法比辗转相除法效率稍低一些。

在处理斐波那契数列相邻项的数据集时，辗转相除法、更相减损术和倒数分解法的效率均远高于因数筛选法，因为因数筛选法需要一直枚举到分子分母中最小的那个数，计算的次数最多。

在处理相邻质数测试数据集合的时候，因数筛选法因为需要逐个枚举的缘故依然表现最慢，但是更相减损术因为测试用例本身的特点，在分子分母差值不同的测试数之间出现了辗转减法次数的震荡变化。

在处理公因数幂次较大测试数据集时，更相减损术因为要反复进行更多次的辗转减法而效率最低，而因数筛选法只需要枚举到较小数时即可终止，效率反而较前者高一些。

在处理分子分母数量级差距较大的数据集时，更相减损术也因为辗转减法次数较多的原因而效率最低，而因数分解法因为用例数据的公因数个数存在较大范围的变化而出现效率震荡情况。

我们数据分析的结果和之前的理论预测基本一致，同时也有一些新的情况。例如，对于单个公因数幂次较大的数据集，我们推测更相减损术的表现不会太差，但是在实际测试时发现它的实际耗时过长。这是由于公因数幂次较大的数据集在一定程度上也具备分子、分母数量级差距大的特征，不同的测试数据集既有相区别的一方面，也会在某些特征上存在相似的地方。

以下是最终得到的结果，对于每个测试数据用例集合，在各自有限的测试范围之内我们观察到每种算法耗时变化有以下比较特征。

	辗转相除法	更相减损术	倒数分解法	因数筛选法
分子、分母为斐波那契数列相邻数对	接近线性	接近线性	接近线性	耗时近似于斐波那契数列项的增长
分子、分母为相邻质数	无明显耗时增长	在一定的区间范围内震荡	无明显耗时增长	接近线性增长
分子、分母的公因数幂次较大	无明显耗时增长	耗时持续为较高的值	无明显耗时增长	耗时小，但是增长快，是高次曲线
分子、分母之间有 10^n 倍的数量级差异	无明显耗时增长	耗时持续为较高的值	无明显耗时增长	耗时在逐渐变宽的范围震荡

第七章 约分算法测试的网格扩展与三维数据可视化绘制

前面的研究所使用的分数，是将某一个顺序排列的数据集合中的元素按照前后相关的方式进行的组合，所生成的测试数据其分子分母具有比较固定的配对规则，这样能够设计出来的测试用例存在一定的局限性，所能够做的测试和分析覆盖面还不够完善，我们思考后尝试用另一种测试模式来扩展测试的范围，将之前顺序排列的元素按照纵横网格的形式进行彼此枚举配对，构造了一个更完全的测试用例数据集合，以此来进行更全面的分析。

1. 新版本分析器工具的设计

在我们升级到 1.x 的新版本二维网格分析程序中，首先定义了更加便于测试的.2dtc 测试用例与.2drm 结果清单数据格式。随后我们又添加了用于兼容二维网格分析的 interface 模块，并在其他几个主要模块中进行桥接，成功地实现了 1.x 双模式兼容的程序模型。紧接着，我们创建了新的 2D 生成器模块，并用它创建了第一批.2dtc 测试用例。

测试出了对应的结果清单之后，就到了关键处——选用合适的分析图表。在 1.x 版本中，我们共构造出三个新型绘图分析器：三维柱状图分析器——δ，三维曲面图分析器——ε，以及二维热力图分析器——ζ。其中，ε 和 ζ 的实现更加成功，通过数据的格式化处理和色谱染色，我们可以从色谱、高低等因素中直观地看出每一个测试结果数据中的速度变化特点。

2. 二维网格待测数据的生成情况

我们根据需要通过新的测试用例生成器工具制作了以下待约分的数据，因为组合之后待测数据量大为上升，我们后面根据计算机的算力情况进行了适当的算法测试组合，对以下数据集合进行了测试分析。

斐波那契数列10×10网格

序号	分子	0	1	…	5	6	…	8	9
分母	数值	1	1	…	8	13	…	34	55
0	1	$\frac{1}{1}$	$\frac{1}{1}$	…	$\frac{8}{1}$	$\frac{13}{1}$	…	$\frac{34}{1}$	$\frac{55}{1}$
1	1	$\frac{1}{1}$	$\frac{1}{1}$	…	$\frac{8}{1}$	$\frac{13}{1}$	…	$\frac{34}{1}$	$\frac{55}{1}$
…	…	…	…	…	…	…	…	…	…
5	8	$\frac{1}{8}$	$\frac{1}{8}$	…	$\frac{8}{8}$	$\frac{13}{8}$	…	$\frac{34}{8}$	$\frac{55}{8}$
6	13	$\frac{1}{13}$	$\frac{1}{13}$	…	$\frac{8}{13}$	$\frac{13}{13}$	…	$\frac{34}{13}$	$\frac{55}{13}$
…	…	…	…	…	…	…	…	…	…
8	34	$\frac{1}{34}$	$\frac{1}{34}$	…	$\frac{8}{34}$	$\frac{13}{34}$	…	$\frac{34}{34}$	$\frac{55}{34}$
9	55	$\frac{1}{55}$	$\frac{1}{55}$	…	$\frac{8}{55}$	$\frac{13}{55}$	…	$\frac{34}{55}$	$\frac{55}{55}$

斐波那契数列50 × 50网格

序号	分子	0	1	…	24	25	…	48	49
分母	数值	1	1	…	75025	121393	…	7778742049	12586269025
0	1	$\frac{1}{1}$	$\frac{1}{1}$	…	$\frac{75025}{1}$	$\frac{121393}{1}$	…	$\frac{7778742049}{1}$	$\frac{12586269025}{1}$
1	1	$\frac{1}{1}$	$\frac{1}{1}$	…	$\frac{75025}{1}$	$\frac{121393}{1}$	…	$\frac{7778742049}{1}$	$\frac{12586269025}{1}$
…	…	…	…	…	…	…	…	…	…
24	75025	$\frac{1}{75025}$	$\frac{1}{75025}$	…	$\frac{75025}{75025}$	$\frac{121393}{75025}$	…	$\frac{7778742049}{75025}$	$\frac{12586269025}{75025}$
25	121393	$\frac{1}{121393}$	$\frac{1}{121393}$	…	$\frac{75025}{121393}$	$\frac{121393}{121393}$	…	$\frac{7778742049}{121393}$	$\frac{12586269025}{121393}$
…	…	…	…	…	…	…	…	…	…
48	7778742049	$\frac{1}{7778742049}$	$\frac{1}{7778742049}$	…	$\frac{75025}{7778742049}$	$\frac{121393}{7778742049}$	…	$\frac{7778742049}{7778742049}$	$\frac{12586269025}{7778742049}$
49	12586269025	$\frac{1}{12586269025}$	$\frac{1}{12586269025}$	…	$\frac{75025}{12586269025}$	$\frac{121393}{12586269025}$	…	$\frac{7778742049}{12586269025}$	$\frac{12586269025}{12586269025}$

质数数列50 × 50网格

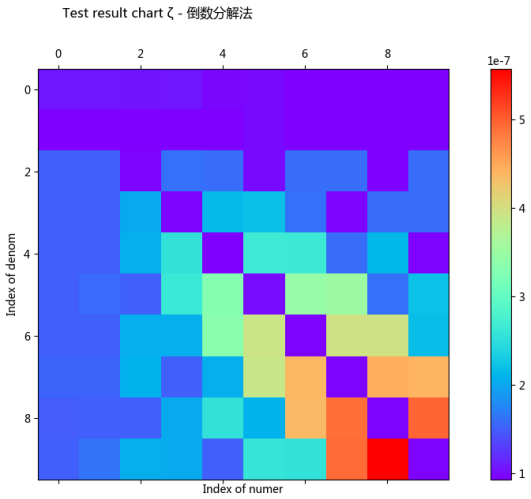
序号	分子	0	1	…	24	25	…	48	49
分母	数值	2	3	…	97	101	…	227	229
0	2	$\frac{2}{2}$	$\frac{3}{2}$	…	$\frac{97}{2}$	$\frac{101}{2}$	…	$\frac{227}{2}$	$\frac{229}{2}$
1	3	$\frac{2}{3}$	$\frac{3}{3}$	…	$\frac{97}{3}$	$\frac{101}{3}$	…	$\frac{227}{3}$	$\frac{229}{3}$
…	…	…	…	…	…	…	…	…	…
24	97	$\frac{2}{97}$	$\frac{3}{97}$	…	$\frac{97}{97}$	$\frac{101}{97}$	…	$\frac{227}{97}$	$\frac{229}{97}$
25	101	$\frac{2}{101}$	$\frac{3}{101}$	…	$\frac{97}{101}$	$\frac{101}{101}$	…	$\frac{227}{101}$	$\frac{229}{101}$
…	…	…	…	…	…	…	…	…	…
48	227	$\frac{2}{227}$	$\frac{3}{227}$	…	$\frac{97}{227}$	$\frac{101}{227}$	…	$\frac{227}{227}$	$\frac{229}{227}$
49	229	$\frac{2}{229}$	$\frac{3}{229}$	…	$\frac{97}{229}$	$\frac{101}{229}$	…	$\frac{227}{229}$	$\frac{229}{229}$

自然数列100×100网格

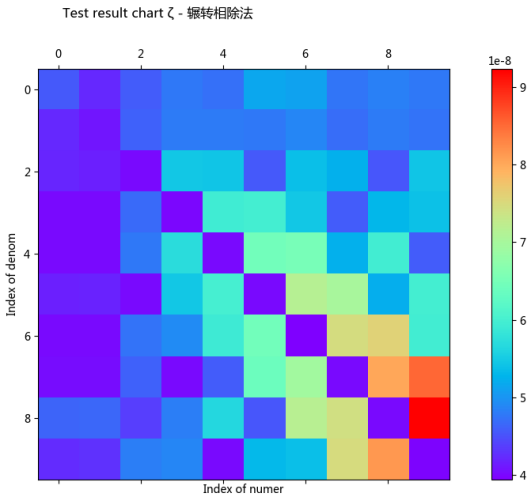
序号	分子	0	1	…	49	50	…	98	99
分母	数值	1	2	…	50	51	…	99	100
0	1	$\frac{1}{1}$	$\frac{2}{1}$	…	$\frac{50}{1}$	$\frac{51}{1}$	…	$\frac{99}{1}$	$\frac{100}{1}$
1	2	$\frac{1}{2}$	$\frac{2}{2}$	…	$\frac{50}{2}$	$\frac{51}{2}$	…	$\frac{99}{2}$	$\frac{100}{2}$
…	…	…	…	…	…	…	…	…	…
49	50	$\frac{1}{50}$	$\frac{2}{50}$	…	$\frac{50}{50}$	$\frac{51}{50}$	…	$\frac{99}{50}$	$\frac{100}{50}$
50	51	$\frac{1}{51}$	$\frac{2}{51}$	…	$\frac{50}{51}$	$\frac{51}{51}$	…	$\frac{99}{51}$	$\frac{100}{51}$
…	…	…	…	…	…	…	…	…	…
98	99	$\frac{1}{99}$	$\frac{2}{99}$	…	$\frac{50}{99}$	$\frac{51}{99}$	…	$\frac{99}{99}$	$\frac{100}{99}$
99	100	$\frac{1}{100}$	$\frac{2}{100}$	…	$\frac{50}{100}$	$\frac{51}{100}$	…	$\frac{99}{100}$	$\frac{100}{100}$

3. 斐波那契数列10×10网格的数据测试分析

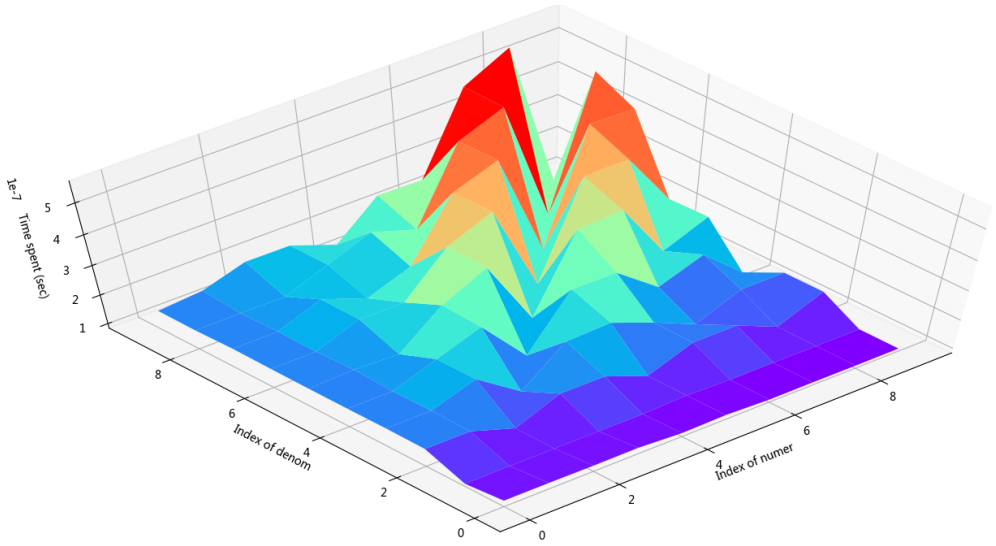
3.1. 倒数分解法与辗转相除法的对比



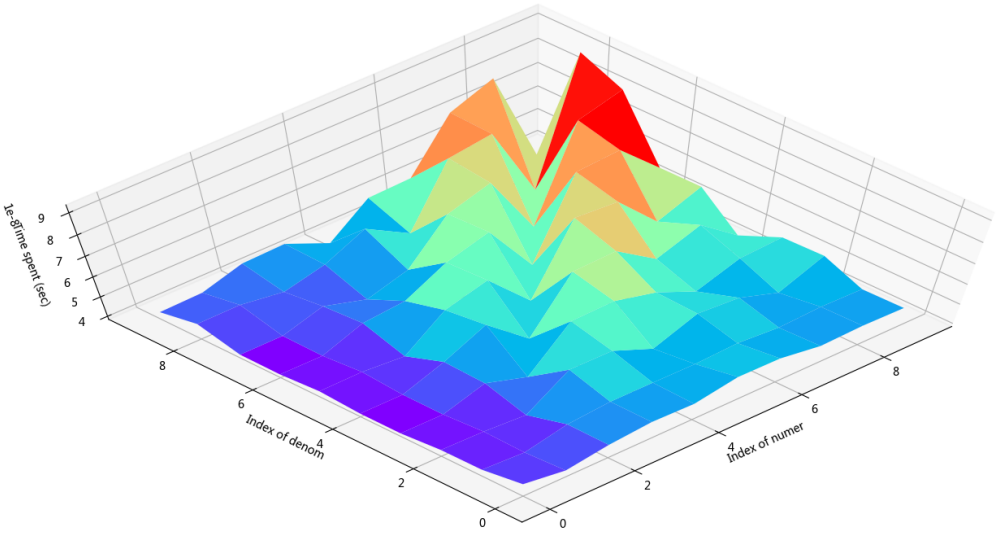
倒数分解法的耗时热力图



辗转相除法的耗时热力图



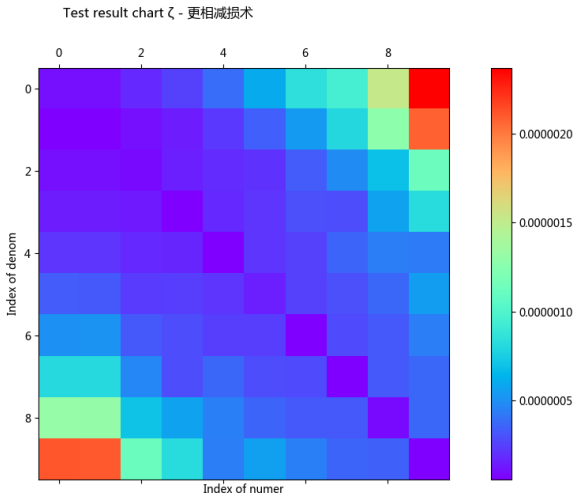
倒数分解法的耗时三维曲面图



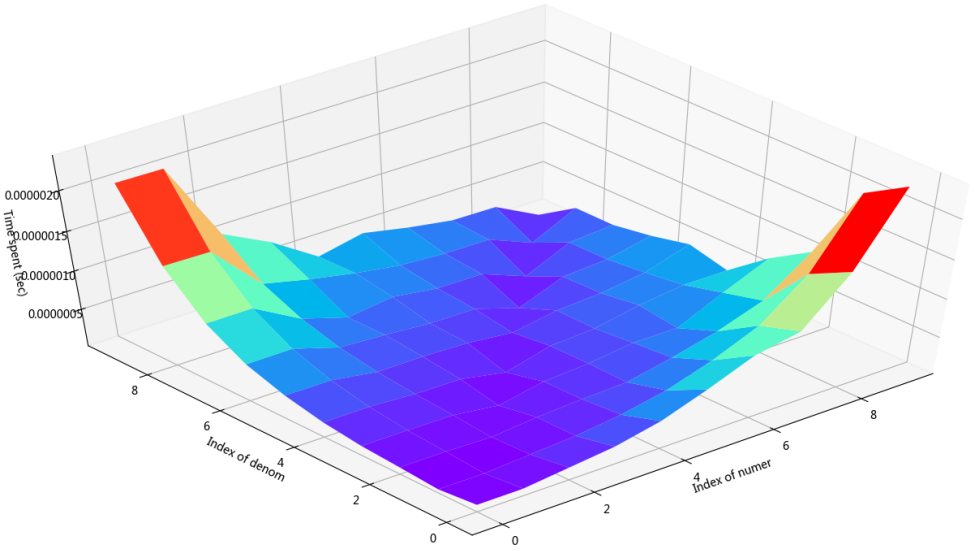
辗转相除法的耗时三维曲面图

从上面几张图可以看出，倒数分解法和辗转相除法的耗时图像非常相似，但两者的效率从有限的数据观察大约相差 3 至 4 倍，辗转相除法更快一些，原因在前面章节有讲述。除分子等于分母的用例外，计算耗时基本都为单调递增，对相差较大的两个斐波那契数计算较快。不过可以发现，它们的图像并不完全对称，以分子等于分母的用例为分界，一侧耗时比另一侧稍多；而且不对称的情况是恰好相反的，这是由于倒数分解法在开始计算的时候先取了一次倒数，分子分母的位置刚好互换了，而且计算步骤也多了一步。

3.2. 更相减损术



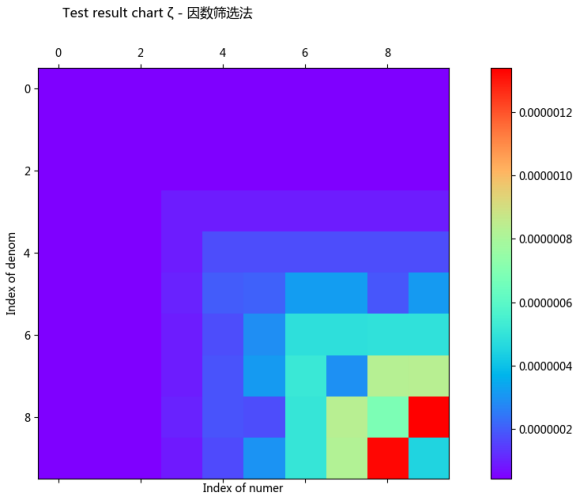
更相减损术的耗时热力图



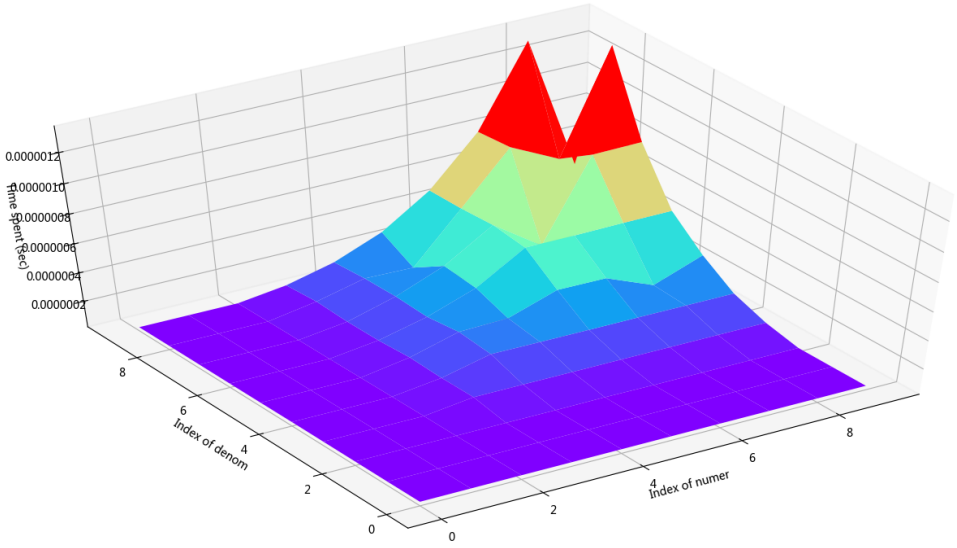
更相减损术的耗时三维曲面图

总体上看，更相减损术的耗时随着分子、分母之差的增大而增大，与前面的理论分析是一致的。与之前发现的变化规律不同的是，当分子、分母相差较小时，更相减损术的耗时并没有增大。这是因为对于斐波那契数列中位置相近的两个数，辗转相减事实上是最优的算法，它的效率比取模运算要高很多，并且由于两数相差不到两倍，其计算步骤数与取模算法的步骤数一样。事实上，更相减损术的运算从模式上来说恰好是斐波那契数生成运算的逆过程。

3.3. 因数筛选法



因数筛选法的耗时热力图

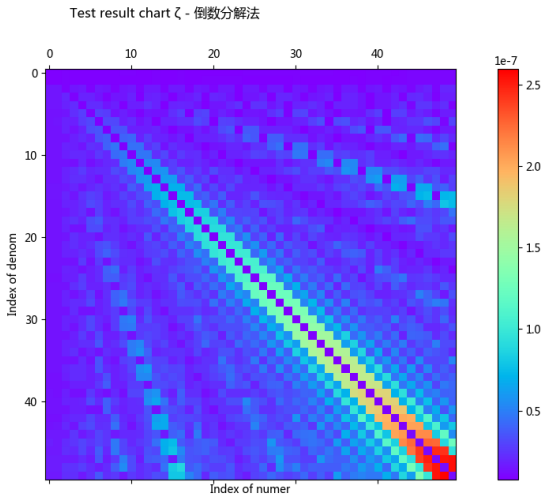


因数筛选法的耗时三维曲面图

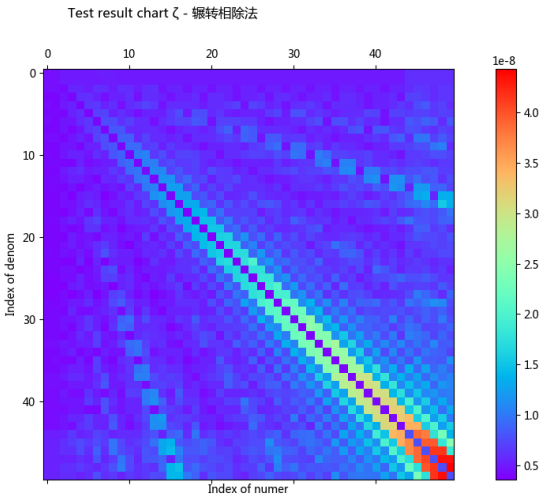
因数筛选法的效率变化规律十分简单，其耗时总体上随着分子、分母中较小数的增大而增大，并且计算有较多公因数的分子、分母所需步骤较少。

4. 斐波那契数列50×50网格的数据测试分析

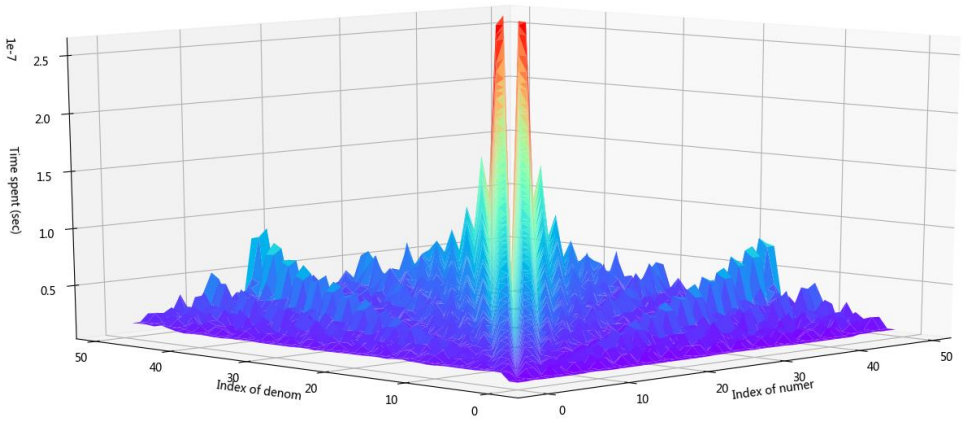
4.1. 倒数分解法与辗转相除法的对比



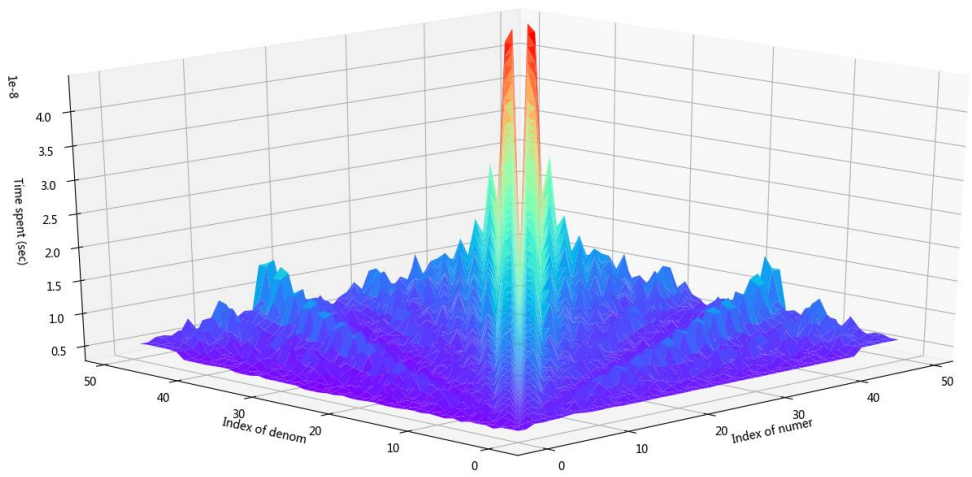
倒数分解法的耗时热力图



辗转相除法的耗时热力图



倒数分解法的耗时三维曲面图

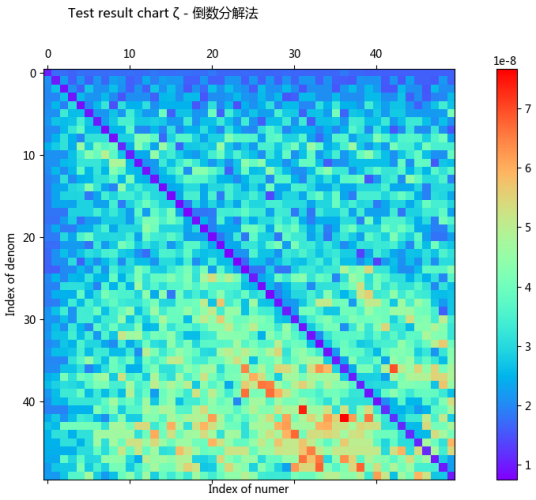


辗转相除法的耗时三维曲面图

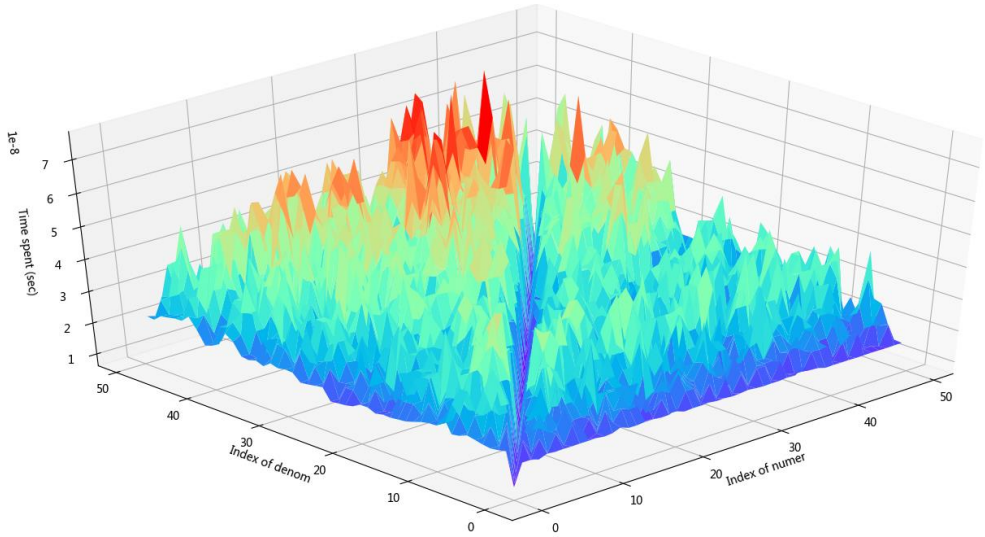
容易发现，辗转相除法和倒数分解法在处理极相近、但不等的两个斐波那契数时，耗时明显增大，这是因为邻近的斐波那契数导致了取模算法的效率退化，如理论分析所述。两者的耗时变化在大范围内与前面的分析一致，不过由于分数的数据规模较大，倒数分解法多计算一步所导致的效率差异已经变得不明显。

5. 质数数列50×50网格的数据测试分析

5.1. 倒数分解法



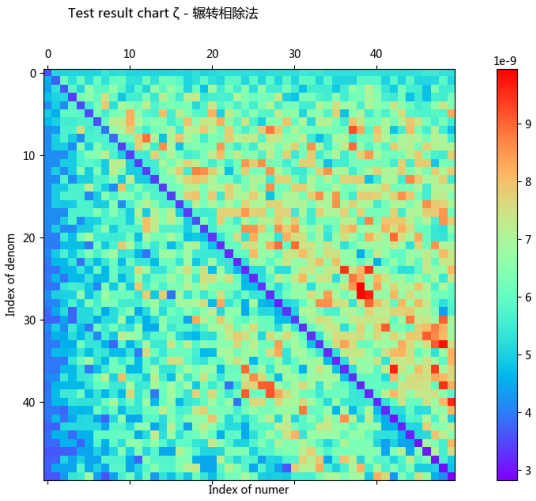
倒数分解法的耗时热力图



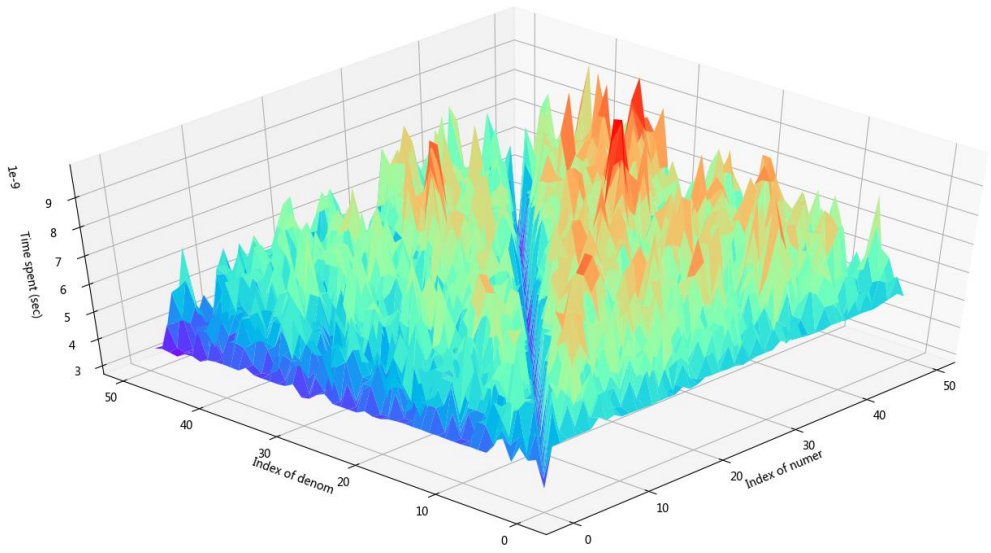
倒数分解法的耗时三维曲面图

可以看出，倒数分解法的耗时除分子等于分母的情况用例以外，并无明显的变化规律，这是质数之比的无规律造成的。

5.2. 辗转相除法



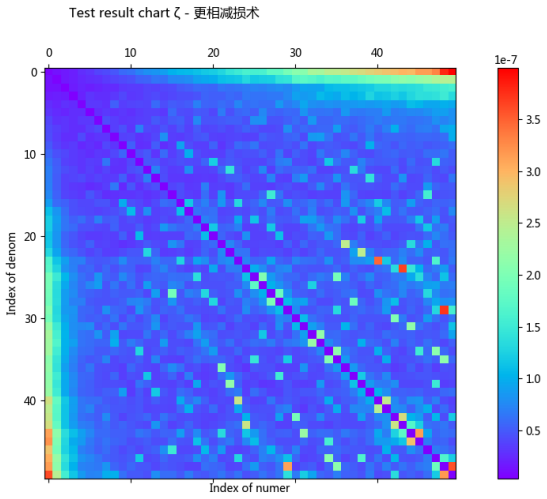
辗转相除法的耗时热力图



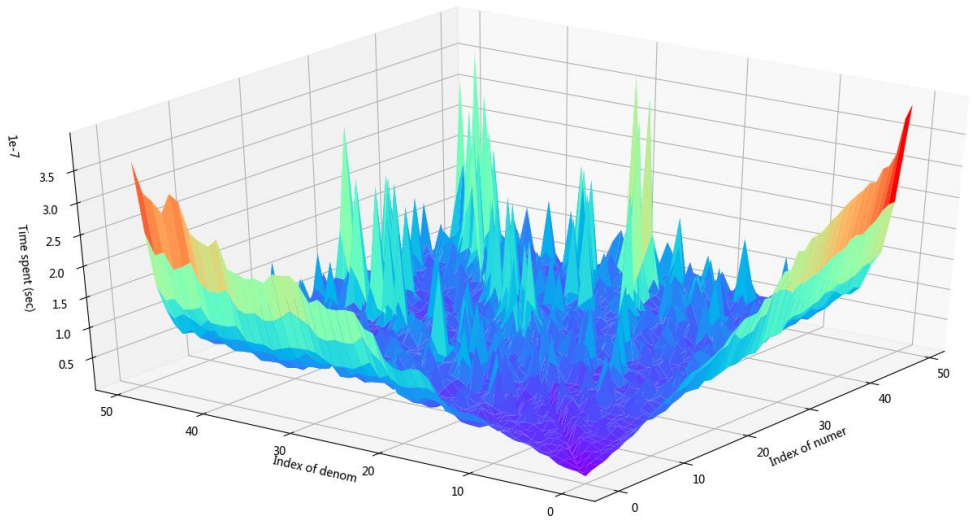
辗转相除法的耗时热力图

辗转相除法的耗时变化特征与倒数分解法一样；和对斐波那契网格的分析一致，仍然存在不对称的情况，且恰好相反。

5.3. 更相减损术



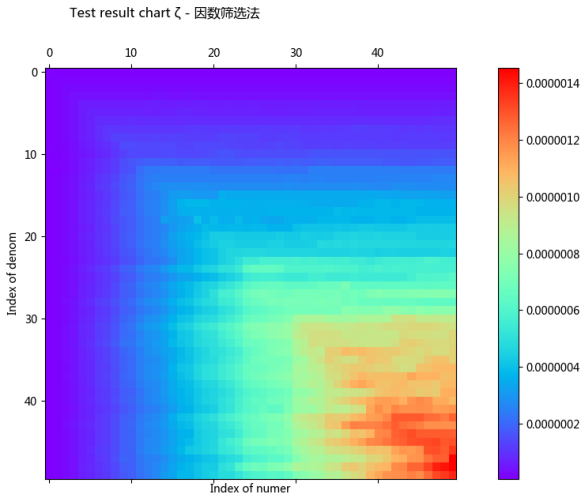
更相减损术的耗时热力图



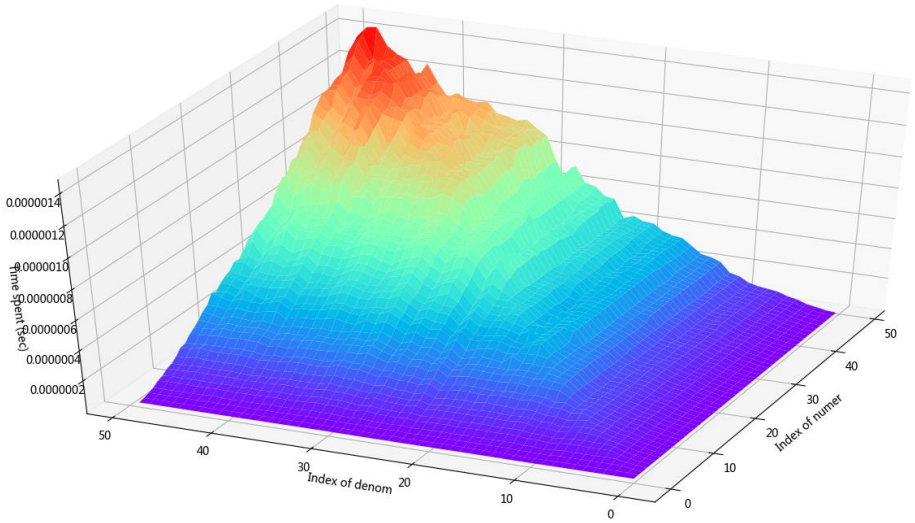
更相减损术的耗时三维曲面图

在大一些的范围内，能够看出更相减损术在处理相差较大的两数时耗时仍有明显的增长的规律。此外，其处理质数网格用例的耗时中，从图中能发现：耗时较高的“亮点”大致在一定的直线侧边分布。在查找了测试用例表后，我们发现在这些直线周围，分子和分母大致为倍数关系，这样的分布规律，在下文对更相减损术处理自然数网格用例时的耗时会深入探究。

5.4. 因数筛选法



因数筛选法的耗时热力图



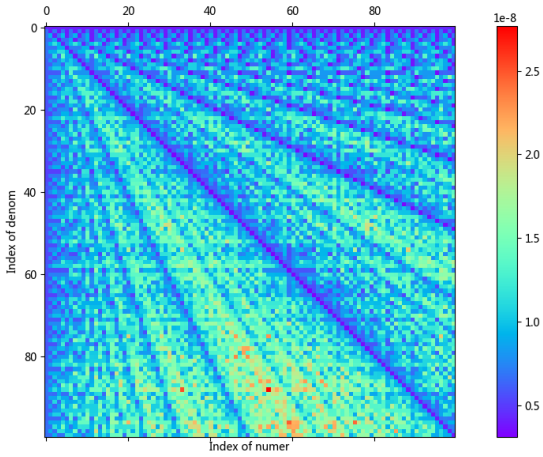
因数筛选法的耗时三维曲面图

因数筛选法的耗时变化规律明显，呈单调递增。这是因为质数互质，因此因数筛选法无法消除公因数以缩小枚举因数的范围，所以决定耗时的唯一因数是质数的大小。

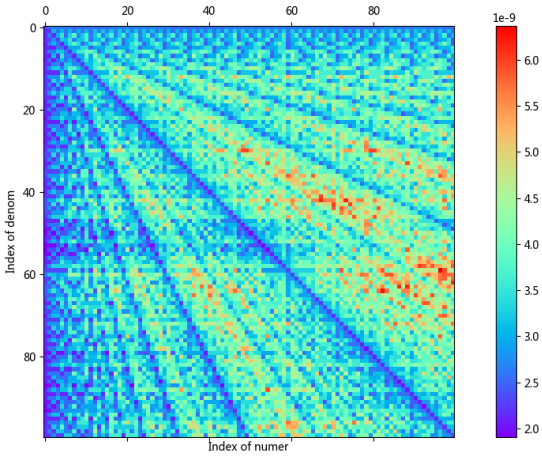
按照耗时的数量级，对四种算法处理质数网格用例的总体效率排序如下（效率从低到高）：因数筛选法 < 更相减损术 < 倒数分解法 < 辗转相除法

6. 自然数列100 × 100网格的数据测试分析

6.1. 倒数分解法与辗转相除法的对比

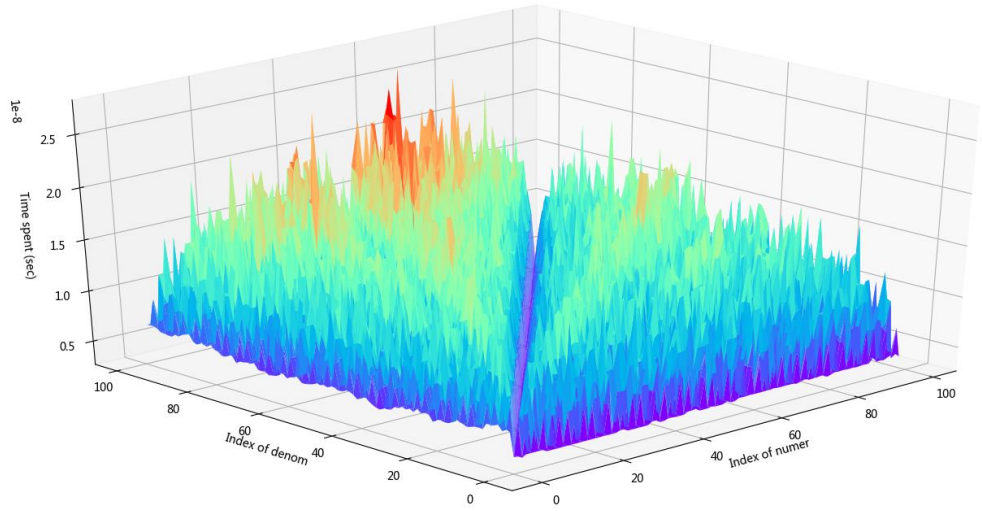


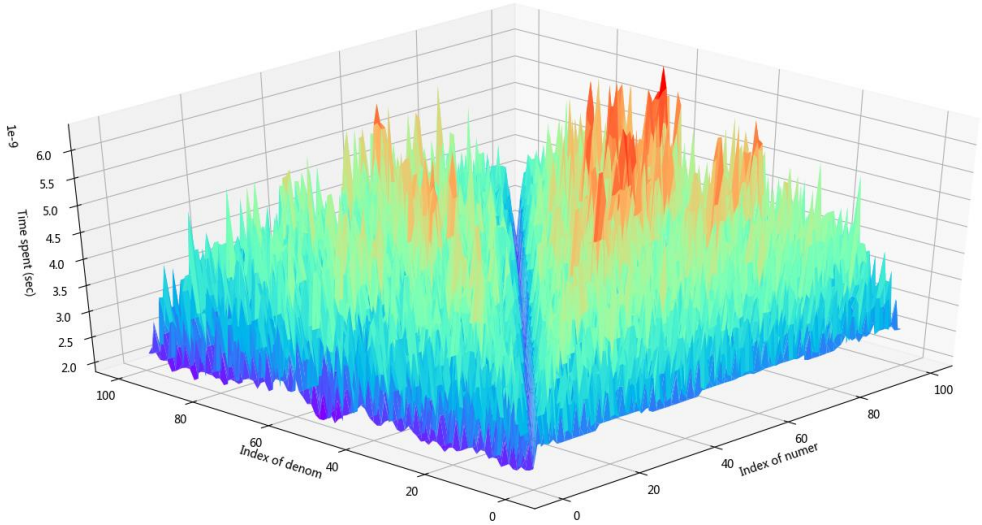
倒数分解法的耗时热力图



辗转相除法的耗时热力图

倒数分解法的耗时三维曲面图



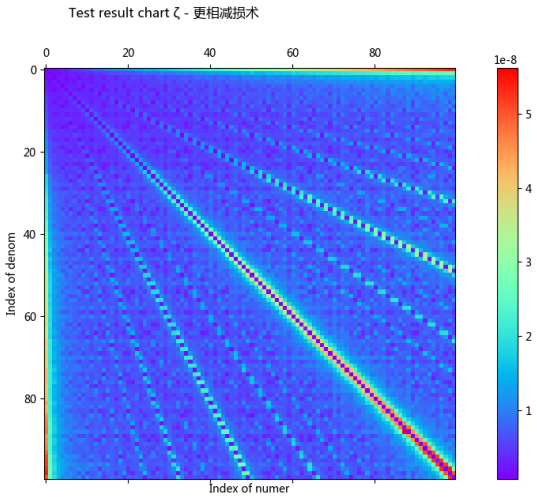


辗转相除法的耗时三维曲面图

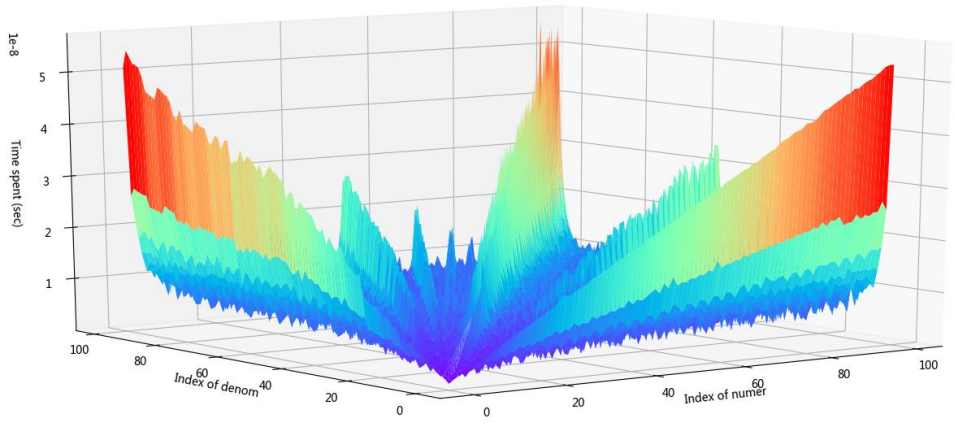
根据用例的值可以发现，辗转相除法和倒数分解法计算效率比较高的那些测试分数

中，分子和分母之比总是为简单分数或非常接近于简单分数的值。大片的耗时较高的区域中，分子与分母之比接近黄金分割数——即近似于斐波那契数列相邻两项比值。其他分散的较高耗时的“红点”对应的用例分数，其分子或分母有等差的规律，这是取模运算的周期性造成的。

6.2. 更相减损术



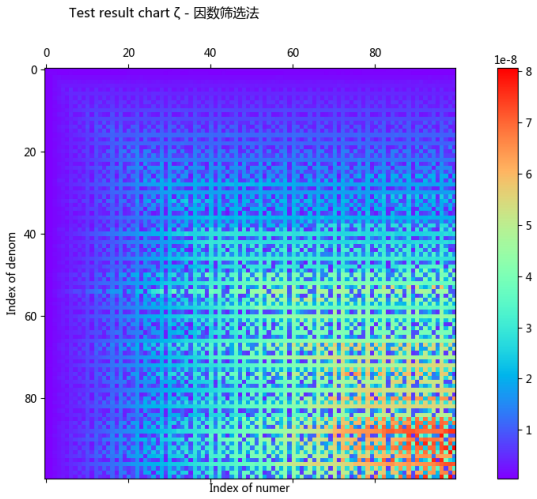
更相减损术的耗时热力图



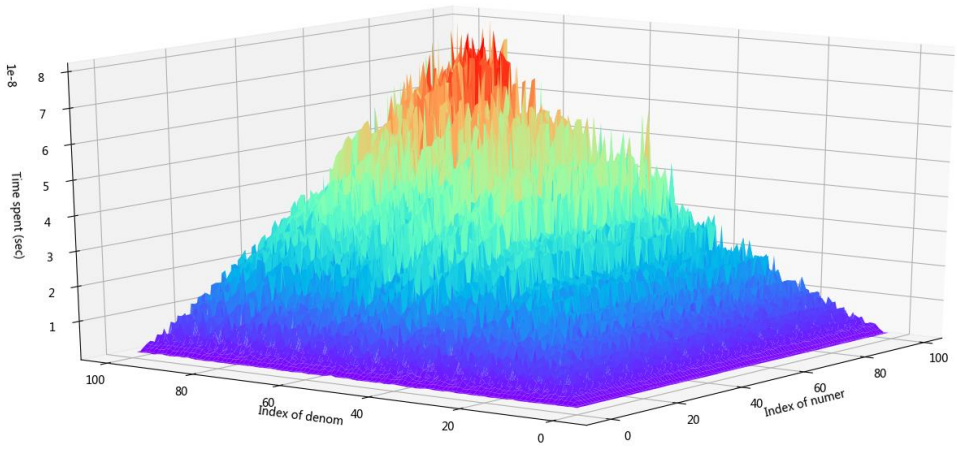
更相减损术的耗时三维曲面图

根据用例的值可以发现，更相减损术在处理分子和分母相差不大、有比较多的公因数（能化简为较简单的分数）时，耗时极小，这是因为计算只需很少几次减法就能完成。但是当非常接近该情况时，耗时却又变得极高，这是因为在进行一次减法之后两数的相差变得极大，导致要进行非常多次的计算，这一点与理论分析的预测结果吻合。在热力图上我们可以看到在分子和分母相等的这条对角线旁边，分子或分母加一或减一形成的两条线上的分数，其计算耗时反而是最多的。

6.3. 因数筛选法



因数筛选法的耗时热力图



因数筛选法的耗时三维曲面图

从因数筛选法的耗时变化图中，我们可以很明显地看出，变化并不是连续而是跳跃性的，且差异呈现成“方框”的形状，方框的骨架所对应的分子分母其公因数数量较少，公因数较多的情况下，计算较快，在曲面图上形成一些陷落下去的蓝色坑点。此外，我们可以发现，耗时的变化以分子、分母相等的对角线为对称分布的，这是因为因数筛选法的效率与分子和分母的大小顺序无关。如果我们把这些公因数很多的坑点都“挤压”掉，最后剩下的就是一些分子分母本身就互质的数值，可以得到和前面质数网格类似的结构分布。

7. 小结

在以上的分析中，我们发现，不同算法的执行耗时虽然数量级变化不大，但是它们的耗时增长趋势是很不一样的。辗转相除法和倒数分解法（均为取模算法）与更相减损术及因数筛选法有着非常不同的最适范围：当分子分母的公因数数量较多时，因数筛选法会处于相对优势；但当在这个基础上分子分母分别偏离一个较小值的时候，辗转相除法和倒数分解法由于有取模运算而占据优势，更相减损术会处于劣势；当分子分母的比值或者计算过程中产生的分数比较接近与黄金分割数的时候，更相减损术会处于相对优势，而辗转相除法和倒数分解法等取模算法会处于劣势。

第八章 总结与展望

1. 研究过程中的感受

本项目的缘起来自于对约分计算问题的深入思考，约分可以说是初等数学中最为基本的一类计算需求，然而在将约分计算的对象扩大到分子分母为超大整数的情况时，我们习以为常的通过观察得出公因数来进行计算的方式就不能奏效了。如何准确且高效的解决大整数之间的约分呢？带着这个问题，我们在偶然间独立寻找到了这个基于连分数的来回变换过程原来可以用来快速产生最简分数的秘密。

经过尝试，这个我们自己命名为倒数分解法的思路，看起来是一个可以通用的约分方法，这时我们又产生了新的问题，是否有更加有效的约分方法？由此我们又开始查找数学史上人们关于这个问题的各种探索。在找到辗转相除法、更相减损术这些经典的方法后，我们想到，是否可以把这些算法全部写成程序，交给计算机来完成这么大量的计算工作呢？

在有了初步的构思，成功的尝试用计算机代码来进行不同算法的约分计算后，我们想到，该如何评价不同约分算法之间的效率差异呢？除了理论上的分析，我们是否可以像物理学的研究一样，通过计算实验来直接对比算法间的差异？

当我们想好了时间测量的方法并且写好代码后，初次运行结果时让我们遇到了这个项目中最麻烦的一个问题，我们该如何消除每次测量中那么多起伏不定的干扰杂峰呢？于是我们继续思考干扰产生的原因，去分析计算机实际运行状态和单纯数学逻辑的不同点。我们发现了由于计算机操作系统和硬件平台本身的特点，随时会存在 CPU 时间分配的波动，在我们测量相对较为微小的算法执行时间的时候，CPU 分时波动会导致所记录的时间值和实验的重复性发生非常大的偏差，这就是在一开始产生如此多噪音信号的原因，认识到了这一点我们就有了解决的办法，最后通过反复试验我们得到了非常有用的平均最小值取样的干扰抑制算法。

在测试器程序逐渐完善的同时，我们也想到了前面一直存在的一个小疑问，不同的算法是否分别有其最适合的计算对象呢？为了对这个问题作出判断，于是我们又做出了一个测试用例生成器来按照我们的猜想方向划分不同的测试数据集合。

当大量的数据不断产生后，我们取得了第一个阶段的成果，为每一种算法对应每一种测试数据类型进行计算，通过大量的测试结果用图表勾画出了它们的模样，这个时候我们开始思考，除了折线图还有没有更好的方法来让数据变得更加直观易于分析？对这个问题的思考让我们突破了原先单一维度的测试思路，把测试集合扩展到了一个二维网格，通过颜色来区分计算速度的快慢，这就是后面的热力图分析了。

到后面我们又想到，能够再直观一些、再好看一些吗？还有别的表现方式吗？我们尝试了柱状图，发现遮挡掉的细节太多，散点图又不易看出相互的关联。最后选择了曲面图，再加上热力值的颜色渲染，当我们打开一张载有一万个数据点的自然数阵列更相减损术约分测试图的时候，我们瞬间被惊艳到了，那张图仿佛就像是一只展翅欲飞的大鹏鸟！

一张一张的图片看过来，我们终于领略到了，什么叫做数学之美……

2. 关于工程学思考的总结

算法的效率问题也是一个工程学问题，因此我们在理论分析之后，为了最客观的对比算法效率，我们决定使用约分运行时间的直接测量来进行最终的速度比较，并在算法代码中嵌入了相关的时间测量记录代码。在项目的设计中我们采取了严格的实证方法来对理论思考进行验证，这个项目的成功构建也是**数学和工程学思维相结合**的结果。

在思考解决 CPU 分时波动带来噪音影响的这个问题时，我们思考了很久，想到了在物理学中学到的知识，结合问题的核心，**把物理量测量方法移植到了数学与计算机项目中**

来，进一步设计了最小测量值作为有效数据认定、算法内自动重复测试和多次启动计算进程求平均值的一系列方法，通过综合使用这些误差消除手段，有效的排除了系统线程对算法执行时间的背景干扰，得到了可信度非常高的结果。

真实的世界是充满复杂性的，这是在单纯的数学题中所不会遇到的情况，这个问题的解决让我们能够用更完整的思维框架来看待科学的研究。

3. 对后续研究的展望和期待

计算速度对于算法而言只是其中的一个指标，从实际运用的角度来说，还需要考虑执行任务所需要的资源，不同的算法在计算机上运行的时候，存储器的占用量可能会存在较大的差别。计算处理的效率和 CPU 资源相关，计算可以处理的规模则和存储器等资源有关，对于算法性能的后续研究，还需要把计算机物理资源的整体耗用情况纳入评价的范围，以比较各算法的资源综合效率。

根据这个研究所得的经验，还可以拓展约分算法的研究范围。例如，与整式的计算法则相结合，我们有可能设计出针对整式约分的高效算法，通过类似的效率分析工具进行建模，构造高效的数学算法，来尝试设计计算机代数分析工具。

通过把约分分析的数据集合扩展到二维网格后，为后续的研究打开了一个新的空间，那就是通过计算机辅助的图形学方法进行数论方面的研究。约分算法本质上也是一种对数字的变换方式，数字之间的内在规律通过各种变换而呈现出来，加上图形学的辅助，有助于我们去发现一些还不为我们所了解的知识。

此外，我们发现形成二维网格排列后不同的数据集合，和对应的算法结果结合在一起，可以形成有特有的空间结构，这令人联想起指纹和密码，如果二维数据集结合不同的秘钥算法，有可能制作出一些破解难度更高的加密系统，这个方向的研究可能也具有比较高的实际用途。

参考文献

[1] 欧几里得 著, 邹忌 编译. 几何原本. 重庆:重庆出版社,2014.1:227-228

[2] 白尚恕. 《九章算术》注释. 北京:科学出版社,1983.12:15-17

[3] Thomas H. Cormen 等. 算法导论（第三版）. 北京：机械工业出版社,2013.1:547-550

[4] Magnus Lie Hetland. Python 基础教程（第二版）. 北京:人民邮电出版社,2014.6

[5] 袁明豪,严培胜,张清芳. 有限简单连分数的几个应用[J]. 黄冈师范学院学报,2003(03):27-30.

[6] 徐德涵.求最大公约数十四法[J].现代特殊教育,2002(03):28-30.