The Goldbach Conjecture

Sections

Summing Primes

Here we verify the conjecture for small numbers.

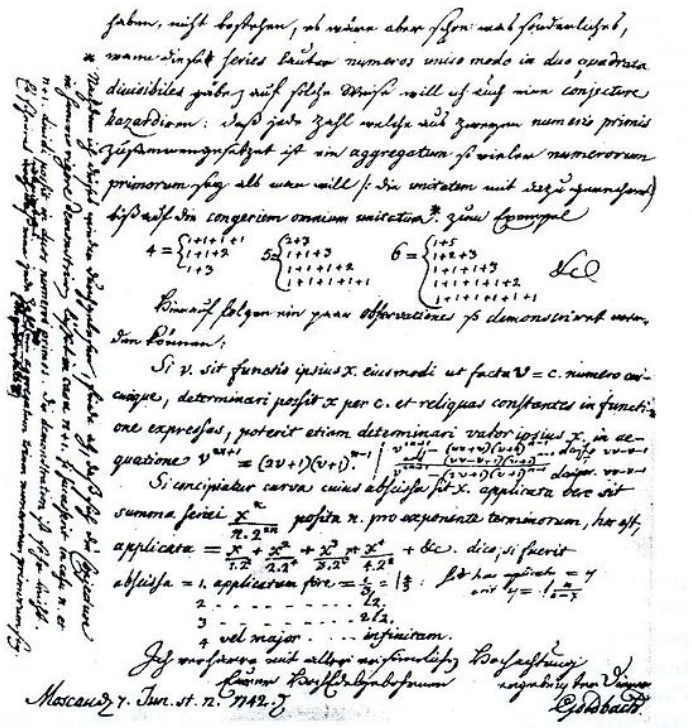
The Sieve of Eratosthenes

A fairly fast way to determine if small numbers are prime, given storage.

Summing Primes

Here we verify the conjecture for small numbers.

§**1.** On 7 June 1742, Christian Goldbach wrote a letter from Moscow to Leonhard Euler in Berlin making "eine conjecture hazardiren" that every even number greater than 2 can be written as a sum of two primes.[1] Euler did not know if this was true, and nor does anyone else.



Goldbach, a professor at St Petersburg and tutor to Tsar Peter II, wrote in several languages in an elegant cursive script, and was much valued as a letter-writer, though his reputation stands less high today.[2] All the same, the general belief now is that primes are just plentiful enough, and just evenly-enough spread, for Goldbach to be right. It is known that:

(a) every even number is a sum of at most six primes (Ramar, 1995), and

(b) every odd number is a sum of at most five (Tao, 2012).

---

[1] "Greater than 2" is our later proviso: Goldbach needed no such exception because he considered 1 a prime number, as was normal then, and was sometimes said as late as the early twentieth century.

[2] Goldbach, almost exactly a contemporary of Voltaire, was a good citizen of the great age of Enlightenment letter-writing. He and Euler exchanged scholarly letters for over thirty years, not something Euler would have kept up with a duffer. Goldbach was also not, as is sometimes said, a lawyer. See: http://mathshistory.st-andrews.ac.uk/Biographies/Goldbach.html. An edited transcription of the letter is at: http://eulerarchive.maa.org//correspondence/letters/OO0765.pdf

§**2.** Computer verification has been made up to around 10^{18}, but by rather better methods than the one we use here. We will only go up to:

```
define RANGE 100
```

```
#include <stdio.h>
int main(int argc, char *argv[]) {
    for (int i=4; i<RANGE; i=i+2)                    stepping in twos to stay even
        ⟨Solve Goldbach's conjecture for i 2.1⟩ ;
}
```

§**2.1.** This ought to print:

```
$ goldbach/Tangled/goldbach
4 = 2+2
6 = 3+3
8 = 3+5
10 = 3+7 = 5+5
12 = 5+7
14 = 3+11 = 7+7
...
```

We'll print each different pair of primes adding up to i. We only check in the range $2 \le j \le i/2$ to avoid counting pairs twice over (thus $8 = 3 + 5 = 5 + 3$, but that's hardly two different ways).

⟨Solve Goldbach's conjecture for i 2.1⟩ ≡

```
printf("%d", i);
for (int j=2; j<=i/2; j++)
    if ((isprime(j)) && (isprime(i-j)))
        printf(" = %d+%d", j, i-j);
printf("\n");
```

This code is used in §2.

The Sieve of Eratosthenes

A fairly fast way to determine if small numbers are prime, given storage.

S1 Storage; S2 Primality

§**1.** This technique, still essentially the best sieve for finding prime numbers, is attributed to Eratosthenes of Cyrene and dates from the 200s BC. Since composite numbers are exactly those numbers which are multiples of something, the idea is to remove everything which is a multiple: whatever is left, must be prime.

This is very fast (and can be done more quickly than the implementation below), but (a) uses storage to hold the sieve, and (b) has to start right back at 2 - so it can't efficiently test just, say, the eight-digit numbers for primality.

```
int still_in_sieve[RANGE + 1];
int sieve_performed = FALSE;
```

§**2.** We provide this as a function which determines whether a number is prime:

```
define TRUE 1
define FALSE 0

int isprime(int n) {
    if (n <= 1) return FALSE;
    if (n > RANGE) { printf("Out of range!\n"); return FALSE; }
    if (!sieve_performed) ⟨Perform the sieve 2.1⟩ ;
    return still_in_sieve[n];
}
```

§**2.1.** We save a little time by noting that if a number up to `RANGE` is composite then one of its factors must be smaller than the square root of `RANGE`. Thus, in a sieve of size 10000, one only needs to remove multiples of 2 up to 100, for example.

⟨Perform the sieve 2.1⟩ ≡

```
    ⟨Start with all numbers from 2 upwards in the sieve 2.1.1⟩ ;
    for (int n=2; n*n <= RANGE; n++)
        if (still_in_sieve[n])
            ⟨Shake out multiples of n 2.1.2⟩ ;
    sieve_performed = TRUE;
```

This code is used in §2.

§**2.1.1.**

⟨Start with all numbers from 2 upwards in the sieve 2.1.1⟩ ≡

```
    still_in_sieve[1] = FALSE;
    for (int n=2; n <= RANGE; n++) still_in_sieve[n] = TRUE;
```

This code is used in §2.1.

§**2.1.2.**

⟨Shake out multiples of n 2.1.2⟩ ≡

```
    for (int m= n+n; m <= RANGE; m += n) still_in_sieve[m] = FALSE;
```

This code is used in §2.1.