Polygonal Cannonball Numbers

Izaak van Dongen

April 10, 2019

Contents

1 Introduction

Being a huge fan of Matt and of Numperphile, I recently watched the video `https://www.youtube.com/watch?v=q6L06pyt9CA`, featuring Matt Parker. Despite Matt's infallibility, I decided to have my own crack at the problem, in the spirit of mathematical enquiry and whatnot.

I reasoned that checking if a number is polygonal should be a roughly $\mathcal{O}(1)$ operation as we can find the nth term of the base-s polygonal numbers $P(s, n)$, which will be quadratic in n, and solve it for n with the quadratic formula, so to check if some cannonball numbers $C(s, n_c)$ is polygonal we just see if the corresponding n_p is an integer. Now 10^9 is a fairly small number. Seeing as my CPU's clockspeed is in the range of gigahertz, and we're just checking a tiny fraction of those numbers as we're just computing the cannonball numbers under this limit, it seems reasonable that this should be doable fairly fast.

I've thought about the problem of higher-dimensional stacks of cannonballs (ie the ones formed by adding up the cannonball numbers), but I've not done anything about it.

While I'm here I'd also like to plug square triangular numbers: `https://en.wikipedia.org/wiki/Square_triangular_number`. I conjecture that these are one of the least talked about, but coolest things in maths. For some inexplicable reason ("""Pell's Equation"""), if you take a convergent b/c of $\sqrt{2}$, then b^2c^2 will be a square triangular number. (Matt Parker voice) How cool is that?!

2 State of the Union

Below is presented a table of depths of s to which I've searched for various upper bounds on C. I did this mostly with one monumental overnight computation on a fairly standard issue desktop computer (I can't bear to hear my laptop's fan). Unfortunately I forgot to store the upper bound for each computation, so they're just the largest number that's been found at that depth. A couple of these may be an underestimate by about an order of magnitude.

$C(s,n) <$	$s <$
1^{7}	322
1^{8}	322
1^{9}	2378
1^{10}	2378
1^{11}	31265
1^{12}	31265
1^{13}	31265
1^{18}	223613
1^{19}	223613
1^{20}	659
1^{22}	83135
1^{24}	5549
1^{26}	26

I've also separately run some computation on numbers congruent to 2 module 3, as I'll discuss more later. With these, I've checked polygons up the the gargantuan 103436-gon .

3 The Maths

Indeed, this approach does seem to work. Almost by definition we have the recurrence in polygonal numbers

$$P(s,n) = P(s,n-1) + n(s-2) - (s-3)$$

so we can use

$$\begin{aligned}P(s,n) &= \sum_{r=1}^{n} P(s,r) - P(s,r-1)\\ &= \sum_{r=1}^{n}(n(s-2)-(s-3))\\ &= \frac{1}{2}n(n+1)(s-2) - n(s-3)\\ &= \frac{n^2(s-2) - n(s-4)}{2}\end{aligned}$$

Fortunately this seems to agree with what Wikipedia thinks. Now, we have

$$\begin{aligned}0 &= (s-2)n^2 - (s-4)n - 2P(s,n)\\ \implies n &= \frac{s-4+\sqrt{(s-4)^2+8(s-2)P(s,n)}}{2s-4}\end{aligned}$$

Wikipedia still seems to think we're on track.

Another result that I don't really use is that

$$C(s,n) = \sum_{r=1}^{n} P(s,n)$$

$$
\begin{aligned}
&= \frac{1}{2}\sum_{r=1}^{n}(n^2(s-2) - n(s-4)) \\
&= \frac{1}{2}\left(\frac{n(n+1)(2n+1)(s-2)}{6} - \frac{n(n+1)(s-4)}{2}\right) \\
&= \frac{1}{12}n(n+1)[(2n+1)(s-2) - 3(s-4)]
\end{aligned}
$$

In fact I've only used this in verification of the results.

Regardless, now we need only work our way up the $C(s,n)$s using the recurrence $C(s,n) = P(s,n) + C(s,n-1)$, and check for each if the quadratic formula gives an integer result. This is most easily done by checking if the discriminant is a perfect square and then checking that the denominator divides the numerator.

4 The Programming

For speeeeeeed I implemented this in C (although there is a long abandoned parallel Python implementation). I used 128-bit integers to be on the safe side, as 10^{19} is a little small for my liking. This meant I had to do a lot of messing around to get things to actually display in base 10. This program is shown in Listing 1.

Of course, an isolated source code listing is both not executable and not necessarily helpful, but fret not as my intact source tree is in `../src`.

I did briefly consider either implementing or importing some kind of arbitrary precision integer arithmetic functionality, but then I decided I wasn't going to run it on anything fast enough to have to worry about that, and I have better things to do.

There's also a slick little progress update that gets printed to STDERR. I've written a number of interacting zsh scripts and Python scripts here and there to manage the actual programs, forming a sort of terribly managed little pipeline, with files that don't make any sense all over the place. Much of it is probably also dependent on software that happens to be on my computer, like some Linux coreutils. Obviously the C code will probably require GCC to work out of the box.

I also have a program that verifies results, removes duplicates and formats them into a LaTeX table (spoilers for table 2), shown in listing 2.

After having used these programs to obtain some data, and plot it and so on and so forth as discussed in the next section, I noticed the glaring pattern with the cannonball numbers derived from a side congruent to 2 modulo 3. By assuming that this pattern continues, in that you can move 3 along and a little up to get to a new cannonball polygonal number, it is easy to generate these kinds of numbers at a preposterous rate. I wrote a little C program (listing 3 which took maybe ten minutes to hit the upper bounds of 128-bit integer arithmetic, so I for now I've written a Python program to bear the torch, and painstakingly squeeze out every last member of the congruence class at my leisure.

```c
// Finding cannonball numbers that are equal to a polygonal number of the same
// base. See https://www.youtube.com/watch?v=q6L06pyt9CA

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
```

```c

// Macro to calculate the n-th polygonal number of side s. It's a macro so I
// don't have to keep typing it but it stays efficient.
// There also also some other macros with the nth term of a cannonball number
#define POLYGONAL(s, n) ((n * n * (s - 2) - n * (s - 4)) >> 1)
#define CANNON(s, n) n * (n + 1) * ((s - 2) * (2 * n + 1) - 3 * (s - 4)) / 12
// Symbolic constants for the default values of the parameters
#define MAX_CHECK_DEFAULT ipow(10, 11)
#define MAX_BASE_DEFAULT 31265
// How many numbers to check before giving an update
#define UPDATE_CYCLES ipow(10, 6) * 5

// integer type being used to represent cannonball numbers
typedef __int128_t cannonball_int;
// maximum possible amount of memory needing to be allocated to represent a
// cannonball_int in base 10 (in an ASCII-encoded string)
#define CANNON_INT_STR_LEN (int)(sizeof(cannonball_int) * log10(0xff) + 2)

// custom function to format a cannonball int into a base 10 string, as printf
// doesn't know how.
void fmt_c(cannonball_int n, char *target) {
    ssize_t i = 0;
    ssize_t size;
    cannonball_int tmp;
    while (n != 0) {
        target[i++] = '0' + (n % 10);
        n = n / 10;
    }
    size = i;
    target[size--] = '\0';
    // reverse it because we built the string back to front
    for (i--; i > size - i; i--) {
        tmp = target[i];
        target[i] = target[size - i];
        target[size - i] = tmp;
    }
}

// Integer exponentiation by squaring - basically just so I can write integers
// in standard form.
cannonball_int ipow(cannonball_int base, cannonball_int exp) {
    cannonball_int result = 1;
    while (exp) {
        if (exp & 1)
            result *= base;
        exp >>= 1;
        base *= base;
    }
    return result;
}

```

```c
// Find the integer square root, with the bit-shifting algorithm. This is used
// when applying the quadratic formula to see if there are rational solutions.
cannonball_int isqrt(cannonball_int n) {
    cannonball_int small, large;
    if (n < 2) {
        return n;
    } else {
        small = isqrt(n >> 2) << 1;
        large = small + 1;
        if (large * large > n) {
            return small;
        } else {
            return large;
        }
    }
}

// Routine to check all cannonball numbers of side `base` up to `max` to see if
// they are also a polyhedral number of side `base`.
void check_base(cannonball_int base, cannonball_int max_check,
                cannonball_int max_base) {
    char *c_1 = malloc(CANNON_INT_STR_LEN),
         *c_2 = malloc(CANNON_INT_STR_LEN),
         *c_3 = malloc(CANNON_INT_STR_LEN),
         *c_4 = malloc(CANNON_INT_STR_LEN);
    cannonball_int i, cannonballs;
    cannonball_int discriminant, discriminant_sqrt, numerator, denominator;
    denominator = 2 * base - 4;
    for (  i = 2, cannonballs = 1 + POLYGONAL(base, 2);
           cannonballs <= max_check;
           i++, cannonballs += POLYGONAL(base, i)) {
        if (i % UPDATE_CYCLES == 0 || (i == 2 && base % UPDATE_CYCLES == 0)) {
            fmt_c(base, c_1);
            fprintf(stderr, "\r%3.0f%% %3.0f%% %s",
                    100.0 * base / max_base,
                    // As cannonballs grows roughly cubically, take a cube root
                    // to linearise the progress
                    100.0 * pow(1.0 * cannonballs / max_check, 1.0 / 3),
                    c_1);
            fflush(stderr);
        }
        discriminant = (base - 4) * (base - 4) + 8 * (base - 2) * cannonballs;
        discriminant_sqrt = isqrt(discriminant);
        if (discriminant_sqrt * discriminant_sqrt == discriminant) {
            numerator = base - 4 + discriminant_sqrt;
            if (numerator % denominator == 0) {
                // not using %n$ syntax but just passing the same argument twice
                // because of something something ISO C
                fmt_c(cannonballs, c_1);
                fmt_c(base, c_2);
                fmt_c(numerator / denominator, c_3);
                fmt_c(i, c_4);
                fprintf(stderr, "\r");
                printf(">%s == P(%s, %s) == C(%s, %s)\n",
                       c_1, c_2, c_3, c_2, c_4);
            }
        }
    }
    free(c_1); free(c_2); free(c_3); free(c_4);
}

int main(int argc, char **argv) {
    cannonball_int base,
                   max_check = MAX_CHECK_DEFAULT,
                   max_base = MAX_BASE_DEFAULT;
    char *c_1 = malloc(CANNON_INT_STR_LEN),
         *c_2 = malloc(CANNON_INT_STR_LEN);
    if (argc >= 2) {
        max_check = (cannonball_int)strtold(argv[1], NULL);
    }
    if (argc >= 3) {
        max_base = (cannonball_int)strtold(argv[2], NULL);
    }
    fmt_c(max_check, c_1);
    fmt_c(max_base, c_2);
    printf("Finding polygonal cannonball numbers <= %s, with base <= %s\n",
           c_1, c_2);
    printf("Using integers of width %zu bytes, which go up to about %.5e\n",
           sizeof(cannonball_int), exp(log(0xff) * sizeof(cannonball_int)));
    for (base = 3; base <= max_base && base <= max_check; base++) {
        check_base(base, max_check, max_base);
    }
    free(c_1); free(c_2);
    return 0;
}
```

Listing 1: The main C source code

```python
#!/usr/bin/env python3

"""
Program to verify polygonal cannonball numbers and then do a little
post-processing.

It's probably quite slow, but in the big O sense, this program is basically
constant time compared to some of the other computation that's happening.
"""

import argparse

```

```python
from cannonball import polygonal
from re import findall
from itertools import chain
from math import log10, inf

def cannonball(s, n):
    """
    Derived cubic nth term of cannonball numbers.
    """
    return n * (n + 1) * ((2 * n + 1) * (s - 2) - 3 * (s - 4)) // 12

def check_line(line):
    """
    Parse and check one line, just by extracting all present integers with some
    regex.
    """
    C, s, n_P, _, n_C = map(int, findall(r"\d+", line))
    if not (C == cannonball(s, n_C) == polygonal(s, n_P)):
        raise ValueError("line {!r} incorrect".format(line))
    return s, C, n_P, n_C

def check_files(files, args):
    """
    Parse and check all the solutions in each file
    """
    solutions = set()
    for line in chain.from_iterable(files):
        if line.startswith(">"):
            solutions.add(check_line(line))
    output_solutions(solutions, args)

def is_boring(sol):
    """
    The idea here is to not display the dull ones
    """
    s, C, n_P, n_C = sol
    return (s > 100 and
            s % 3 == 2 and
            log10(C) > -3 + 7 * log10(s) and
            log10(C) < -2.5 + 7.5 * log10(s))

def solutions_key(sol):
    """
    Key to push boring solutions to the end
    """
    if is_boring(sol):
        return (inf, *sol)
    return sol

def output_solutions(solutions_, args):
    """
    Write solutions to a LaTeX table
    """
    solutions = list(sorted(solutions_, key=solutions_key))
    # write the output as LaTeX. We're not here to be pretty, so might as well
    # play a few rounds of code golf.
    for solution in solutions:
        write_files = [args.write_all]
        if is_boring(solution):
            write_files.append(args.write_boring)
        else:
            write_files.append(args.write_interesting)
        for wfile in write_files:
            print((" {} ".join("&" * 5)[2:-1] + r"\\").format(*solution),
                  file=wfile)

def get_args():
    """
    Get arguments from command line
    """
    parser = argparse.ArgumentParser(description=__doc__)
    parser.add_argument("--files", type=argparse.FileType("r"), required=True,
                        nargs="+", help="list of files to read")
    parser.add_argument("--write-interesting", type=argparse.FileType("w"),
                        required=True,
                        help="File to write table of interesting data to")
    parser.add_argument("--write-boring", type=argparse.FileType("w"),
                        required=True,
                        help="File to write boring data to")
    parser.add_argument("--write-all", type=argparse.FileType("w"),
                        required=True,
                        help="File to write all data to")
    return parser.parse_args()

if __name__ == "__main__":
    args = get_args()
    check_files(args.files, args)
```

Listing 2: Python verification program

```c
// Finding cannonball numbers for polygons with s sides, where s = 2 mod 3.
// Makes the technically unfounded assumption that for each s >= 8 there is such
// a number and it follows the rough upward trend seen in the graph, but, I
// mean, really, have you seen the graph??

#include <stdio.h>
#include <math.h>
#include <stdlib.h>

// Macro to calculate the n-th polygonal number of side s. It's a macro so I
// don't have to keep typing it but it stays efficient.
// There also also some other macros with the nth term of a cannonball number
```

```c
#define POLYGONAL(s, n) ((n * n * (s - 2) - n * (s - 4)) >> 1)
#define CANNON(s, n) n * (n + 1) * ((s - 2) * (2 * n + 1) - 3 * (s - 4)) / 12
// How many numbers to check before giving an update
#define UPDATE_CYCLES ipow(10, 6) * 5

// integer type being used to represent cannonball numbers
typedef __int128_t cannonball_int;
// maximum possible amount of memory needing to be allocated to represent a
// cannonball_int in base 10 (in an ASCII-encoded string)
#define CANNON_INT_STR_LEN (int)(sizeof(cannonball_int) * log10(0xff) + 2)

// custom function to format a cannonball int into a base 10 string, as printf
// doesn't know how.
void fmt_c(cannonball_int n, char *target) {
    ssize_t i = 0;
    ssize_t size;
    cannonball_int tmp;
    while (n != 0) {
        target[i++] = '0' + (n % 10);
        n = n / 10;
    }
    size = i;
    target[size--] = '\0';
    // reverse it because we built the string back to front
    for (i--; i > size - i; i--) {
        tmp = target[i];
        target[i] = target[size - i];
        target[size - i] = tmp;
    }
}

// Integer exponentiation by squaring - basically just so I can write integers
// in standard form.
cannonball_int ipow(cannonball_int base, cannonball_int exp) {
    cannonball_int result = 1;
    while (exp) {
        if (exp & 1)
            result *= base;
        exp >>= 1;
        base *= base;
    }
    return result;
}

// Find the integer square root, with the bit-shifting algorithm. This is used
// when applying the quadratic formula to see if there are rational solutions.
cannonball_int isqrt(cannonball_int n) {
    cannonball_int small, large;
    if (n < 2) {
        return n;
    } else {
        small = isqrt(n >> 2) << 1;
        large = small + 1;
        if (large * large > n) {
            return small;
        } else {
            return large;
        }
    }
}

// find the first polygonal number and break, going up from the previous stack
// height.
cannonball_int run_base(cannonball_int base, cannonball_int n_c) {
    char *c_1 = malloc(CANNON_INT_STR_LEN),
         *c_2 = malloc(CANNON_INT_STR_LEN),
         *c_3 = malloc(CANNON_INT_STR_LEN),
         *c_4 = malloc(CANNON_INT_STR_LEN);
    cannonball_int cannonballs;
    cannonball_int discriminant, discriminant_sqrt, numerator, denominator;
    denominator = 2 * base - 4;
    for (  cannonballs = CANNON(base, n_c);;
           n_c++, cannonballs += POLYGONAL(base, n_c)) {
        discriminant = (base - 4) * (base - 4) + 8 * (base - 2) * cannonballs;
        discriminant_sqrt = isqrt(discriminant);
        if (discriminant_sqrt * discriminant_sqrt == discriminant) {
            numerator = base - 4 + discriminant_sqrt;
            if (numerator % denominator == 0) {
                // not using %n$ syntax but just passing the same argument twice
                // because of something something ISO C
                fmt_c(cannonballs, c_1);
                fmt_c(base, c_2);
                fmt_c(numerator / denominator, c_3);
                fmt_c(n_c, c_4);
                fprintf(stderr, "\r");
                printf(">%s == P(%s, %s) == C(%s, %s)\n",
                       c_1, c_2, c_3, c_2, c_4);
                break;
            }
        }
    }
    free(c_1); free(c_2); free(c_3); free(c_4);
    return n_c;
}

int main(void) {
    cannonball_int base, n_c;
    printf("Finding cannonball numbers where s = 2 mod 3\n");
    n_c = 2;
    for (base = 8; ; base += 3) {
        n_c = run_base(base, n_c);
    }
    return 0;
}
```

Listing 3: C program to find cannonball polygons for side congruent to 2 mod 3

5 The Ugly

I have plotted both the data in its entirety on a double logarithmic scale 1.

The obvious pattern that jumps out is the big line of points for all the sides congruent to 2 (mod 3). Particularly because it looks like such a straight line on the log-log plot, we would expect it to be modelled well as a constant multiple of some power of s. I drew two lines that seemed to roughly bound it, and used those to extract the points on the line and then do some linear regression on that (figure 3). I obtained the formula

$$C = 0.006008708 \cdot s^{7.002464} \qquad \text{Average percentage error of } 0.1387323\ \%$$

I have also plotted these points on a linear scale, demonstrating their relationship 2.

Lastly, I plotted all points other than the points along this line in figure 4.

The R code I used to achieve all this is in Listing 4.

Table 2 lists some solutions that I've found, so far. The TEX source of the table is in `../graph/interesting.tsv` which is derived from `../src/c/solutions/*`. I have deliberately omitted the "boring" solutions along the dense line, favouring the more flavourful, stylish and individualistic solutions.

There are also two tables `boring.tsv` and `all.tsv` containing only the boring solutions and all solutions, respectively, but there are such a truly mind-boggling number of boring solutions that really it's hardly any fun looking at them. The table would literally be three order of magnitude larger if I hadn't. I'm not joking - I made a separate PDF with a full table in `../full_list` and it's about 700 pages long.

```r
library(ggplot2)

interesting_df <- read.table("interesting.tsv")
colnames(interesting_df) <- c("s", "C", "n_P", "n_C")

boring_df <- read.table("boring.tsv")
colnames(boring_df) <- c("s", "C", "n_P", "n_C")

all_df <- read.table("all.tsv")
colnames(all_df) <- c("s", "C", "n_P", "n_C")

model <- lm(log(C) ~ log(s), data=boring_df)
intercept <- coef(summary(model))["(Intercept)", "Estimate"]
grad <- coef(summary(model))["log(s)", "Estimate"]
boring_df$fit <- exp(intercept) * boring_df$s ^ grad
boring_df$err <- abs((boring_df$fit / boring_df$C) - 1)
cat("\\begin{equation*}\n")
cat("C =", exp(intercept), "\\cdot s ^ {", grad, "}\n")
cat("\\qquad \\text{Average percentage error of ",
    100 * mean(boring_df$err), "\\%}")
cat("\\end{equation*}\n")

ggplot(all_df, aes(s, C)) +
    geom_point(shape=16) +
    ggtitle("Log plot of polygonal cannonball numbers") +
    labs(x="s - sides of base polygon", y="C - number of cannonballs") +
    theme(panel.grid.minor = element_line(colour="gray", size=0.4),
          panel.grid.major = element_line(colour="gray", size=1),
          panel.background = element_blank()) +
    scale_x_log10() +
    scale_y_log10() +
    geom_abline(intercept = -3, slope = 7, linetype="dotted") +
    geom_abline(intercept = -2.5, slope = 7.5, linetype="dotted")

ggplot(boring_df, aes(s, C)) +
    geom_point(shape=16) +
    ggtitle("Linear plot of boring bits") +
    labs(x="s - sides of base polygon", y="C - number of cannonballs") +
    theme(panel.grid.minor = element_line(colour="gray", size=0.4),
          panel.grid.major = element_line(colour="gray", size=1),
          panel.background = element_blank())

ggplot(boring_df, aes(s, C)) +
    geom_point(shape=16) +
    ggtitle("Log plot of the subset") +
    labs(x="s - sides of base polygon", y="C - number of cannonballs") +
    theme(panel.grid.minor = element_line(colour="gray", size=0.4),
          panel.grid.major = element_line(colour="gray", size=1),
          panel.background = element_blank()) +
    scale_x_log10() +
    scale_y_log10() +
    geom_smooth(method = "lm", linetype="dashed", color="red")

ggplot(interesting_df, aes(s, C)) +
    geom_point(shape=16) +
    ggtitle("Log plot of interesting bits") +
    labs(x="s - sides of base polygon", y="C - number of cannonballs") +
    theme(panel.grid.minor = element_line(colour="gray", size=0.4),
          panel.grid.major = element_line(colour="gray", size=1),
          panel.background = element_blank()) +
    scale_x_log10() +
    scale_y_log10()
```

Listing 4: R graphical analysis

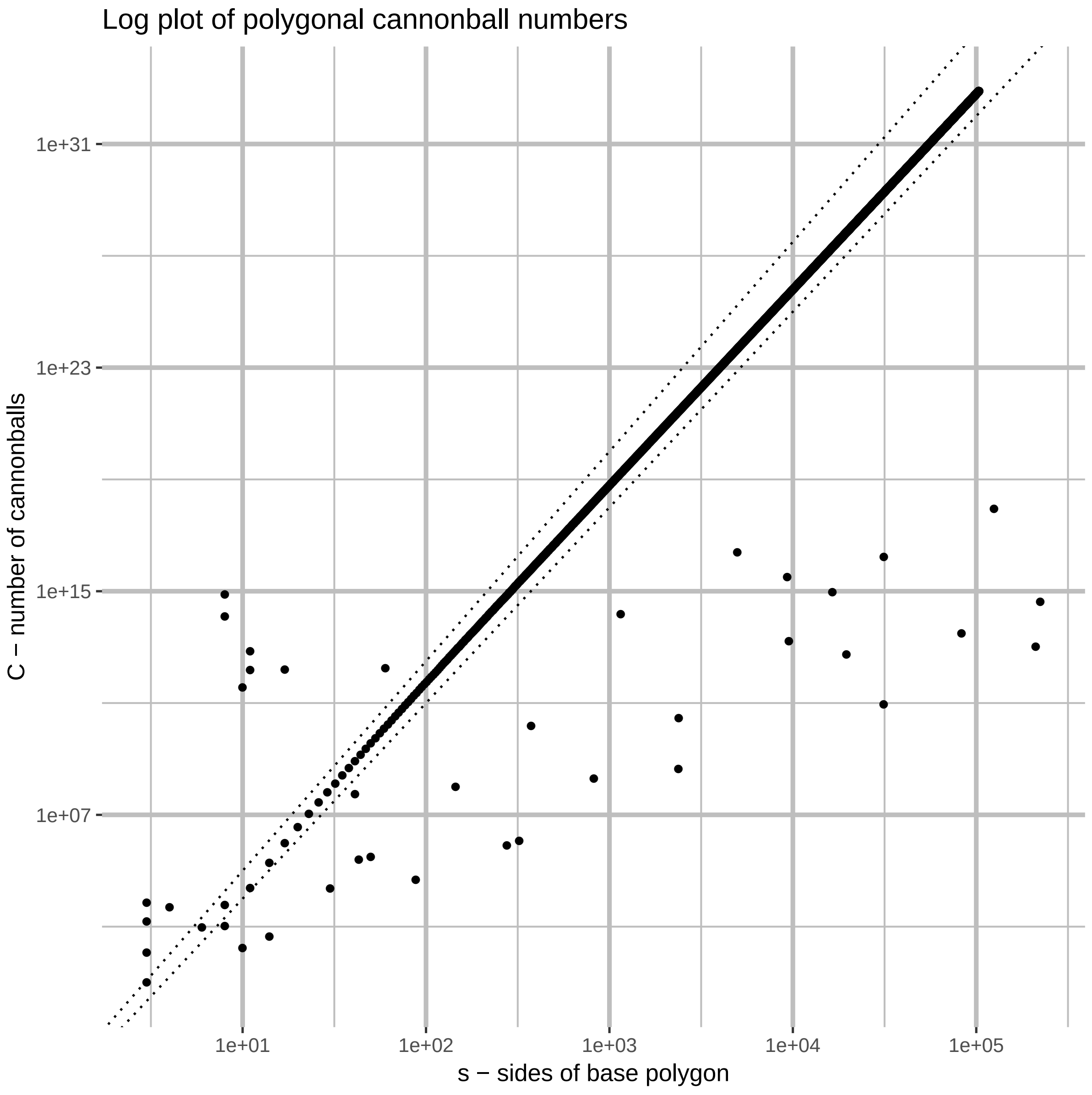

Figure 1: Log plot

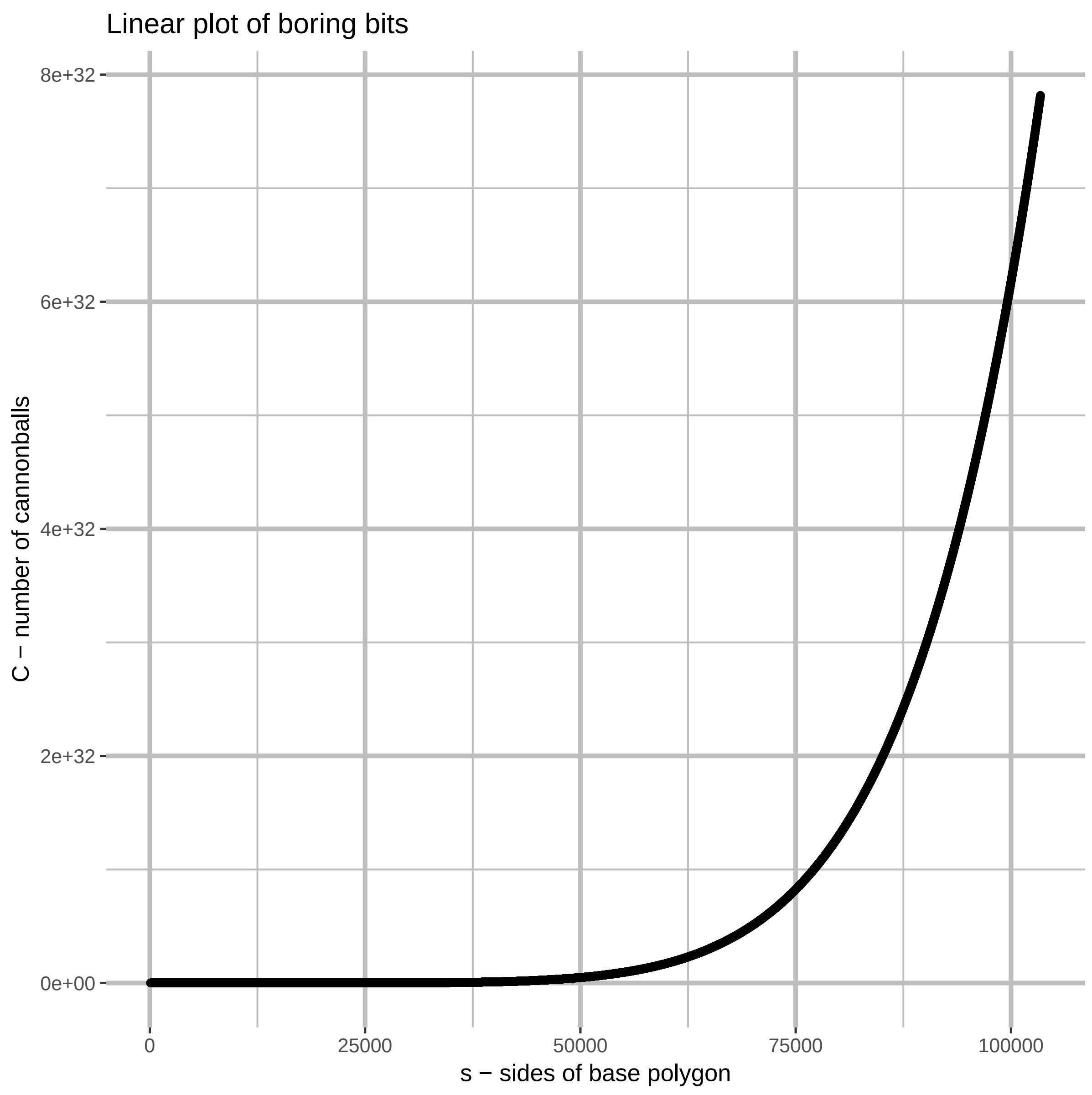

Figure 2: Linear plot of boring points

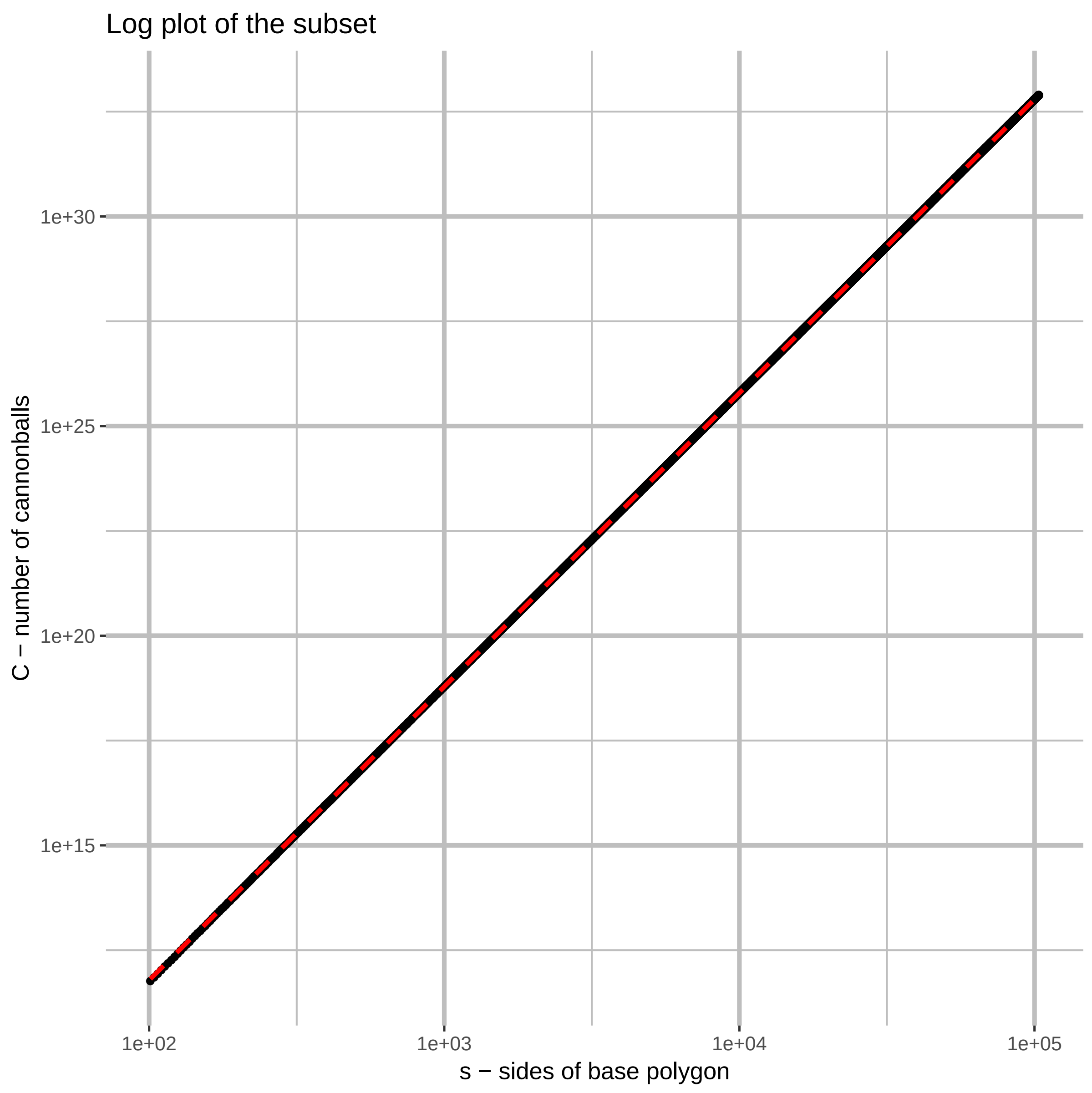

Figure 3: Log plot of the boring points

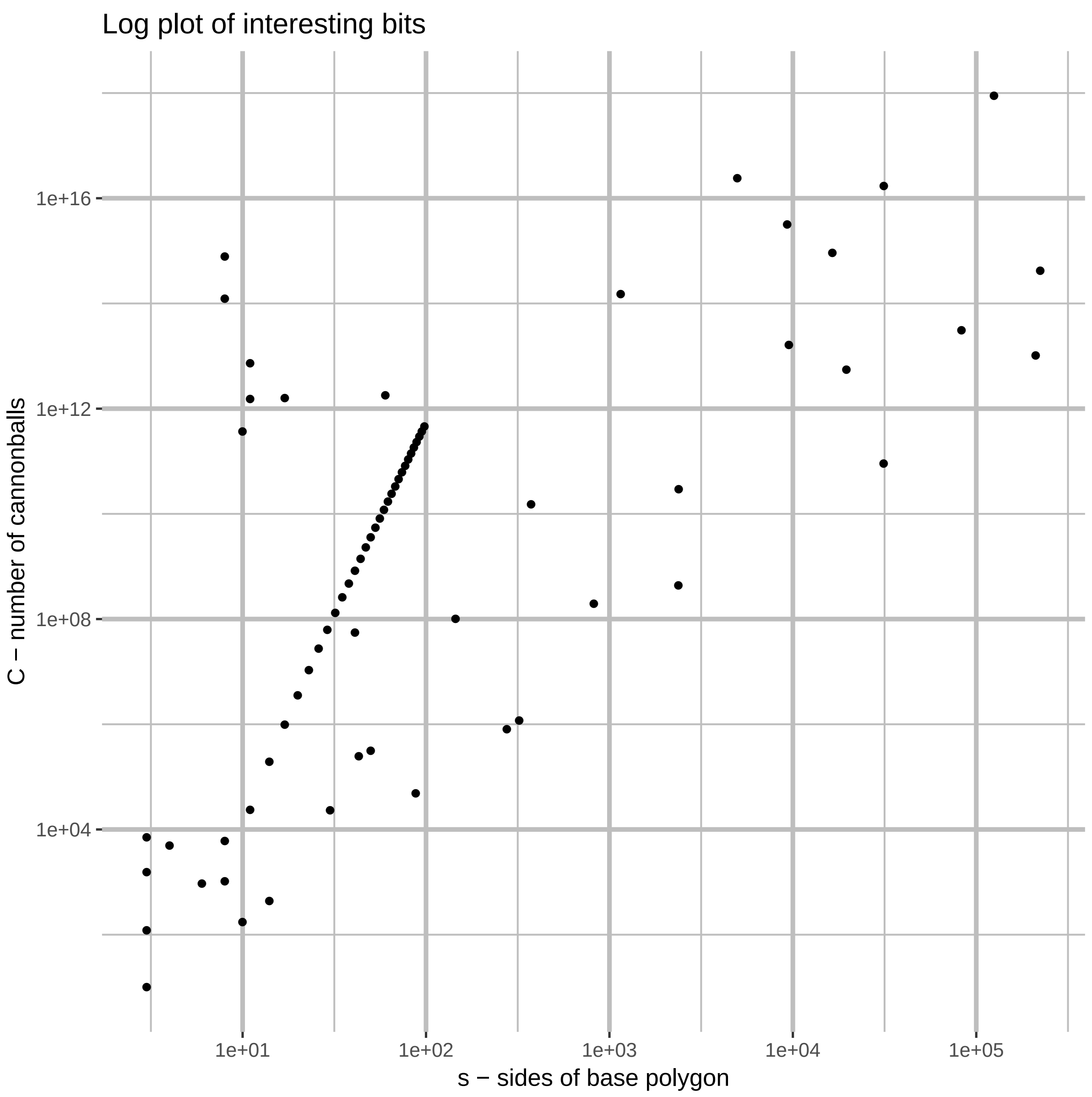

Figure 4: Log plot of the remaining points

s	$C(s, n_c) = P(s, n_p)$	n_p	n_c
3	10	4	3
3	120	15	8
3	1540	55	20
3	7140	119	34
4	4900	70	24
6	946	22	11
8	1045	19	10
8	5985	45	18
8	123395663059845	6413415	49785
8	774611255177760	16068720	91839
10	175	7	5
10	368050005576	303336	6511
11	23725	73	25

s	$C(s, n_c) = P(s, n_p)$	n_p	n_c
11	1519937678700	581175	10044
11	7248070597636	1269127	16906
14	441	9	6
14	195661	181	46
17	975061	361	73
17	1580765544996	459096	8583
20	3578401	631	106
23	10680265	1009	145
26	27453385	1513	190
29	63016921	2161	241
30	23001	41	17
32	132361021	2971	298
35	258815701	3961	361
38	477132085	5149	430
41	55202400	1683	204
41	837244045	6553	505
43	245905	110	33
44	1408778281	8191	586
47	2286380881	10081	673
50	314755	115	34
50	3595928401	12241	766
53	5501691505	14689	865
56	8214519205	17443	970
59	12001111741	20521	1081
60	1785508245600	248132	5695
62	17194450141	23941	1198
65	24205450501	27721	1321
68	33535911025	31879	1450
71	45792819865	36433	1585
74	61704091801	41401	1726
77	82135801801	46801	1873
80	108110983501	52651	2026
83	140830060645	58969	2185
86	181692979525	65773	2350
88	48280	34	15
89	232323110461	73081	2521
92	294592986361	80911	2698
95	370651946401	89281	2881
98	462955752865	98209	3070
145	101337426	1191	162
276	801801	77	26
322	1169686	86	28
374	15064335000	9000	624
823	197427385	694	113
1152	149979784926720	510720	9215
2378	432684460	604	103
2386	29437553530	4970	420
4980	24264913354964425	3122317	30810
9325	3176083959788026	825436	12691
9525	16195753597485	58322	2169

s	$C(s, n_c) = P(s, n_p)$	n_p	n_c
16420	913053565546276	333506	6936
19605	5519583702676	23731	1191
31265	90525801730	2407	259
31368	17147031694579605	1045635	14858
83135	31148407558500	27375	1310
125070	890348736143873526	3773306	34956
210903	10290361955160	9879	664
223613	421687634347915	61414	2245

Table 2: Polygonal Cannonball Numbers