REPORT ON

ZHÌ-WĚI SŪN'S 1-3-5 CONJECTURE AND SOME OF ITS REFINEMENTS

ANTÓNIO MACHIAVELO, ROGÉRIO REIS, AND NIKOLAOS TSOPANIDIS

ABSTRACT. We report here on the computational verification of a refinement of Zhì-Wěi Sūn's "1-3-5 conjecture" for all natural numbers up to 105 103 560 126. This, together with a result of two of the authors, completes the proof of that conjecture.

1. INTRODUCTION

In a paper on refinements of Lagrange's four squares theorem, Zhì-Wěi Sūn (孙智伟) made the conjecture that any $m \in \mathbb{N}$ can be written as a sum of four squares, $x^2 + y^2 + z^2 + t^2$, with $x, y, z, t \in \mathbb{N}_0$, in such a way that x + 3y + 5z is a perfect square. This is Conjecture 4.3(i) in [Sūn17], and Zhì-Wěi Sūn called it the "1-3-5 conjecture". Qìng-Hǔ Hóu (侯庆虎) verified it up to 10^{10} : see https://oeis.org/A271518. We report here on our computational verification of this conjecture for all $m \leq 105 \, 103 \, 560 \, 126$.

This last number arose from the work [MT20], in which it was proved that the 1-3-5 conjecture is true for all numbers

$$m > \left(\frac{10}{\sqrt[4]{35} - \sqrt[4]{34}}\right)^4 \simeq 105103560126.80255537.$$

Hence, the computation we are here reporting on, together with the main result of [MT20], completes the proof that the 1-3-5 conjecture holds for all natural numbers.

Along the way, we have used another conjecture of $S\bar{u}n$, at his own suggestion, Conjecture 4.9(ii) of $[S\bar{u}n19]$, the part whose content is as follows.

Conjecture 1 (Zhì-Wěi Sūn). Any positive integer can be written as $x^2 + y^2 + z^2 + t^2$ with $x, y, z, t \in \mathbb{N}_0$ such that x + 3y + 5z is a square, and either x is three times a square, or y is a square, or z is a square.

Date: 15/05/2020.

²⁰¹⁰ Mathematics Subject Classification. Primary 11E25; Secondary 11D85, 11E20. Key words and phrases. 1-3-5 Conjecture.

As we will see below, we checked this conjecture for all natural numbers up to 105 103 560 126, which implies the 1-3-5 conjecture for the same range. We have noticed that, actually, for most numbers, one may drop the possibility that z is a square. Moreover, it now seems that after some point on, all numbers have a representation as in the statement of Conjecture 1, but with $x \in \{0, 3\}$ or $y \in \{0, 1\}$. This is the content of Conjecture 2 below.

2. The Modus Operandi

We will say that a quadruple $(x, y, z, t) \in \mathbb{N}_0^4$ is a 1-3-5 representation of m if $x^2 + y^2 + z^2 + t^2 = m$ and x + 3y + 5z is a perfect square. Since it is clear that if a number m has the 1-3-5 representation (x, y, z, t), then (4x, 4y, 4z, 4t) is a 1-3-5 representation of 16m, in order to verify the 1-3-5 conjecture, we may disregard multiples of 16.

Early on, during the first computations we made, it was found out that, apparently, only the following 15 numbers:

31, 43, 111, 151, 168, 200, 248, 263, 319, 456, 479, 871, 1752, 1864, 3544,

and their multiples by powers of 16, do not have a 1-3-5 representation (x, y, z, t) where either x is three times a square or y a square. We used this, together with Conjecture 1, to speed up the search.

Furthermore, while testing the program's speed, and while running it for values up to 10^7 , and then up 10^8 , it was noted that only 123 numbers have a special 1-3-5 representation requiring x and y bigger than 4, and that the last one of these was 779832. This observation motivated the introduction of a "tolerance" input on the program, make it to exit whenever the list of numbers to be checked was smaller than a certain size, and returning the list of those numbers, which can then be checked individually in a faster way.

To tackle the verification of the 1-3-5 conjecture up to the required number, 105 103 560 126, a program in the C programming language was written that takes into account the above remarks, and runs as follows. Firstly, it allocates the necessary memory for the range one is checking, ignoring the multiples of 16. Then, it looks for all triplets (x, y, z) such that x + 3y + 5zis a square, and either x is three times a square or y is a square. Each time such a triplet is found, one removes from the appropriate memory the numbers $m = x^2 + y^2 + z^2 + s$, for all squares s with m in the desired range. The program ends when the list of the remaining numbers has size less than a prescribed number, which is part of the input. Since what is wanted is to check, for every integer below a given bound, if there is an additive decomposition in four squares, the naive implementation of such check would have a $\mathcal{O}(n^3)$ complexity¹. But the simple observation that we can check the existence of the fourth square summand by subtracting the summation of the first three to the integer that is tested, and check that this result is a square, lowers this complexity to $\mathcal{O}(n^{\frac{5}{2}})$. This is still a complexity that makes the algorithm intractable for the desired bound.

Since we could not find a canonical ordering for the possible summands that would allow to efficiently prune the search tree, the solution relied in the classic space/time tradeoff. Thus, we represented the whole integer search space as a bitmap, and with a $\mathcal{O}(n^{\frac{3}{2}})$ search could sweep this space and verify the conjecture. As a matter of fact, and because, as already mentioned, we can exhaust the whole search space with just a few instances of the variable of the outer cycle, in practice the algorithm finishes in a $\mathcal{O}(n)$ time.

The problem with this approach is that, because the size of the search space is quite considerable, the memory space necessary to store the bitmap representing the referred set was larger than the one available in our laptops. Thus, the program does not try to cover the whole space of considered integers in just one run, but splits this space in various slices, that are searched independently. This has the advantage of a "parallelism for poor people", running the program on different slices in different laptops, but has the drawback that the program for the higher slices, by the nature of the additive decomposition, needs the same time to conclude as the same program would need to run on an unsplitted space of integers. With laptops of 16GB of RAM, we splitted the search space in 11 slices: the *i*-th slice covering the range $[i \times 10^{10}, (i + 1) \times 10^{10}]$, for $0 \le i \le 9$, and the 11-th slice covering the remaing numbers up to 105 103 560 126. In the process, the range previously checked by Qing-Hǔ Hóu was rechecked.

The code of the C program we used is given in Appendix A. The input consists of three numbers: the range over which one is checking; the "tolerance", which is the size of the list of the numbers that were not checked yet; and, finally, the interval one is checking.

¹All complexity considerations made here suppose that the cost of arithmetic operations for integers in the range considered has complexity $\mathcal{O}(1)$.

A. MACHIAVELO, R. REIS, AND N. TSOPANIDIS

3. The results

The different slices were distributed by several machines, and each slice took between 2 to 3.5 days (depending on the machine, and on the extra use that its owner was making of it). While working on slice 1, it was noticed that only four numbers required the outter "for" cycle to go beyond 1, and checking these four numbers was taking a huge amount of time. Thus, the program was interruped, and restarted with a tolerance of 10, and here is the output:

```
135Siever version 1.3
Sieving 135 for 105103560126, with tolerance 10,
in the interval [1000000000,2000000000]
0 9375000000
1 1562500004
4 4
```

```
Done!! Lasting numbers: 4
10234584952,11035927288,11051651704,14485001848
```

This output means that there are 9375000000 numbers to be checked (recall that one is ignoring the multiples of 16), that after the first run of the outter cycle (which looks for 1-3-5 representations (x, y, z, t) where $x = 3k^2$ or $y = k^2$, with k = 0, 1, ...), there remained only 1562500004 numbers, and that after the second run (k = 1) only 4 numbers are left. The program then stops, and outputs those numbers.

These four 11-digits-long positive integers were then checked using the PARI/GP functions presented in appendix B, which uses an algorithm to write a prime congruent to one modulo 4 as a sum of two squares that is described by John Brillhart in [Bri72]. Using those functions, one very quickly gets, for example (it is a random algorithm), the following 1-3-5 representations:

(8524, 9502, 33094, 94744)	for	10234584952,
(13438, 32472, 12774, 98172)	for	11035927288,
(84720, 34818, 28982, 42684)	for	11051651704,
(32742, 93858, 36824, 56988)	for	14485001848.

ZHÌ-WĚI SŪN'S 1-3-5 CONJECTURE

As a further example, we give here the output of the last slice:

```
135Siever version 1.3
Sieving 135 for 105103560126, with tolerance 10,
in the interval [10000000000,105103560126]
0 4784587619
1 797431269
4 0
Done!! Lasting numbers: 0
132334.83 user 0.20 system 36:45:2 6elapsed 100%CPU
(0avgtext+0avgdata 623532maxresident)k
0inputs+8outputs (0major+155918minor)pagefaults 0swaps
4. A NEW CONJECTURE
```

As a consequence of the computational results displayed above, we now make the following conjecture.

Conjecture 2. Any $m \in \mathbb{N}$, that is not a multiple of 16, with the exception of 31, 43, 111, 151, 168, 200, 248, 263, 319, 456, 479, 871, 1752, 1864, 3544, can be represented as a sum of four squares, $x^2 + y^2 + z^2 + t^2$, with $x, y, z, t \in \mathbb{N}_0$ such that x + 3y + 5z is a square, and either x is three times a square, or y is a square. Moreover, for $m > 14\,485\,001\,848$, one has a representation with $x \in \{0,3\}$ or $y \in \{0,1\}$, and (disregarding multiples of 16) exactly $\frac{5}{6}$ of the numbers have a representation with x = 0 or y = 0, while the remainder $\frac{1}{6}$ have a representation with x = 3 or y = 1.

Appendix A. T	he C program
---------------	--------------

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 #define VERSION "1.4"
6
7 #define MAX 1000000L
8 #define LIM 0
```

6

```
9
   #define MAXS 10000L
10
    #define MINS OL
    #define FULLSET (B64)0xfffefffefffefffe
11
12
    typedef unsigned long B64;
13
    typedef unsigned long Long;
14
15
    typedef struct {
16
17
        B64 *map;
        {\rm Long\ min}\,,\ {\rm max}\,,\ {\rm nelements}\;;
18
19
    } BitMap;
20
21
    BitMap map, squares;
    B64 *masks;
22
23
    Long max = MAX, mins=MINS, maxs=MAX;
24
    B64* buildMasks(void){
25
        B64 * masks, *pt, val=(B64)1;
26
        int i;
27
28
29
        masks = (B64*) malloc(64* sizeof(B64));
30
         pt = masks;
         for (i=0; i<64; i++){
31
             *(pt++) = val;
32
33
             val = val \ll 1;
34
        }
35
        return masks;
36
    }
37
    void newBMapFull(BitMap *bmap, Long min, Long max){
38
        B64 *pt;
39
        Long nbytes, i, size;
40
41
         size = \max - \min + 1;
42
         nbytes = (size /(sizeof(B64)*8)) +1;
43
        bmap \rightarrow map = (B64 *) malloc(sizeof(B64) * nbytes);
44
        bmap \rightarrow max = max;
45
46
        bmap \rightarrow min = min;
        bmap->nelements = max-(max/16) - (min-(min/16));
47
48
        pt = bmap \rightarrow map;
         for (i=0; i < nbytes; i++) (*pt++) = FULLSET;
49
50
    }
51
    int memberP(BitMap *bmap, Long n){
52
        B64 byte, foo;
53
        Long rnumber;
54
55
```

```
56
          if (n > bmap \rightarrow max || n < bmap \rightarrow min) return 0;
57
          rnumber = n - bmap \rightarrow min;
          byte = rnumber/64;
58
          foo = rnumber - byte * 64;
 59
          if (*(masks+foo) & *(bmap->map+byte)) return 1;
 60
          return 0;
61
     }
62
 63
     void removeM(BitMap *bmap, Long n){
64
65
          Long byte, foo, rnumber;
 66
67
          if (n > bmap \rightarrow max || n < bmap \rightarrow min) return;
 68
          rnumber = n - bmap \rightarrow min;
          byte = rnumber / 64;
 69
 70
          foo = rnumber - byte * 64;
          if (*(masks+foo) & *(bmap->map+byte)) {
 71
 72
               (bmap->nelements)--;
               *(bmap->map + byte) = *(bmap->map+byte) & ~*(masks+foo);
 73
 74
          }
 75
     }
76
 77
     void addM(BitMap *bmap, Long n){
78
          B64 byte, foo;
79
          Long rnumber;
 80
          if (n > bmap \rightarrow max || n < bmap \rightarrow min) return;
81
 82
          rnumber = n - bmap \rightarrow min;
 83
          byte = rnumber /(64);
          foo = rnumber - byte * 64;
84
          if (!(*(masks+foo) & *(bmap->map+byte))) {
 85
 86
               (bmap->nelements)++;
               *(bmap->map + byte) = *(bmap->map+byte) | *(masks+foo);
 87
          }
 88
     }
89
 90
91
     void printM(BitMap *bmap){
          Long i, size;
92
93
          int j;
94
95
          size = bmap \rightarrow max - bmap \rightarrow min +1;
          for (i=0; i \le (bmap->max)/64; i++)
96
               if(*(bmap \rightarrow map+i))
97
 98
                    for (j=0; j<64; j++)
                        if(i*64+j+(bmap->min) > bmap->max)
99
                             printf(" \setminus n");
100
101
                             return;
102
                        }
```

```
8
                        A. MACHIAVELO, R. REIS, AND N. TSOPANIDIS
103
                       if(*(bmap->map+i) & *(masks+j)) printf("%lu ",i*64+j+bmap->min);
104
                  }
              }
105
         }
106
107
     }
108
     void saveM(BitMap *bmap, int ord){
109
         FILE * file;
110
111
         char fname[100];
112
         Long i;
         int j;
113
114
         sprintf(fname, "c135-%d.csv",ord);
115
116
          file = fopen(fname,"w");
          for (i=0; i \le (bmap - max)/64; i++)
117
118
              if(*(bmap \rightarrow map+i))
119
                   for (j=0; j<64; j++){
                       if(i*64+j > bmap \rightarrow max)
120
                            printf("\setminus n");
121
122
                            return;
123
                       }
                       if (*(bmap->map+i) & *(masks+j)) fprintf(file,"%lu, ",i*64+j+bmap->min);
124
125
                  }
126
              }
127
         }
128
          fclose(file);
129
     }
130
131
     int squarep(Long n){
         Long i;
132
         i = (int)(sqrt(n)+0.5);
133
         return i * i == n;
134
135
     }
136
     void dealWTriple(BitMap* map, Long i, Long j,Long k){
137
         Long foo, n=0, n2=0;
138
139
          if(squarep(i+3*j+5*k)) \{
140
              foo = i * i + j * j + k * k;
141
              while (1) {
142
                   if (foo + n2 > max) break;
                  removeM(map, foo+n2);
143
                  n2 += 2*(n++)+1;
144
145
              }
146
         }
147
     }
148
149
     int main(int argc, const char * argv[]) {
```

9

```
150
         Long i2=0, i=0, j, k, i4=0, lim=LIM;
151
          printf("135 Siever version \%s\n", VERSION);
152
153
         masks = buildMasks();
          if (argc == 4)
154
              \lim = \operatorname{atol}(\operatorname{argv}[3]);
155
              mins = atol(argv[1]);
156
              \max = \operatorname{atol}(\operatorname{argv}[2]);
157
              \max = \max;
158
              printf("Sieving 135, with tolerance %lu, in the interval [%lu,%lu]", lim, mins, maxs);
159
160
          } else {
161
              printf("Usage c135 lim min maxn");
162
              exit(-1);
163
         }
164
         newBMapFull(&map, mins, maxs);
165
          while (i4 \ll max)
               printf("%lu\t%lu\n",i2 ,map.nelements);
166
               if (map.nelements <= lim) {
167
                   printf("\n");
168
169
                   printf("Done!! Lasting numbers: %lu\n",map.nelements);
170
                   printM(&map);
                   printf(" \setminus n");
171
                   exit(0);
172
173
              }
174
              for (j=0; j*j \le \max(i+1))
                   for (k=j; k*k \le (maxs-i4-j*j); k++)
175
176
                        if(k*k+j*j+i2*i2 > maxs) break;
                        dealWTriple(&map, 3*i2,j,k);
177
178
                        dealWTriple(&map, 3*i2,k,j);
                        dealWTriple(&map, j,i2,k);
179
180
                        dealWTriple(&map, k,i2,j);
181
                   }
              }
182
              i4 += 2*i2+1;
183
              i2 + 2*(i++)+1;
184
185
          }
          \operatorname{printf}("\setminus n");
186
187
         printM(&map);
          printf("\n Done! Lasting numbers: %lu\n",map.nelements);
188
189
          return 0;
190
     }
```

APPENDIX B. THE PARI/GP FUNCTIONS

```
1 /* Representation of a quaternion as a 4 x 4 matrix */
```

 $\mathbf{2}$

A. MACHIAVELO, R. REIS, AND N. TSOPANIDIS

```
quat(a, b, c, d) = [a, b, c, d; -b, a, -d, c; -c, d, a, -b; -d, -c, b, a];
 3
 4
5
    /* The Hurwitz units */
 6
    unid = [quat(1,0,0,0), quat(-1,0,0,0), quat(0,1,0,0), quat(0,-1,0,0),
 7
    quat(0,0,1,0), quat(0,0,-1,0), quat(0,0,0,1), quat(0,0,0,-1),
 8
    \operatorname{quat}\left(1/2\,,1/2\,,1/2\,,1/2\,\right),\ \operatorname{quat}\left(-1/2\,,-1/2\,,-1/2\,,-1/2\,\right),
9
    quat (1/2, 1/2, 1/2, -1/2), quat (-1/2, -1/2, -1/2, 1/2),
10
    quat(1/2, 1/2, -1/2, 1/2), quat(-1/2, -1/2, 1/2, -1/2),
11
    \operatorname{quat}\left(1/2\,,-1/2\,,1/2\,,1/2\right),\ \operatorname{quat}\left(-1/2\,,1/2\,,-1/2\,,-1/2\right),
12
    quat(1/2, 1/2, -1/2, -1/2), quat(-1/2, -1/2, 1/2, 1/2),
13
14
    quat (1/2, -1/2, 1/2, -1/2), quat (-1/2, 1/2, -1/2, 1/2),
    quat(1/2, -1/2, -1/2, 1/2), quat(-1/2, 1/2, 1/2, -1/2),
15
    quat(1/2, -1/2, -1/2, -1/2), quat(-1/2, 1/2, 1/2, 1/2)];
16
17
18
    /* Fast modular exponentiation */
19
20
    expmod(a, e, m) = \{
21
       local(x, y, s, d);
       x\!\!=\!\!a\,;\ y\!=\!1;\ s\!=\!e\,;
22
23
       while (s, d=s\%2;
24
         s=(s-d)/2; if (d, y=(y*x)\%m); x=(x*x)\%m);
25
       return(y);
       }
26
27
    /* Imodp computes the solution of x^2 = -1 \pmod{p} with 0 < x < p/2, for p=1
28
    (mod 4) */
29
30
    Imodp(p) = \{
31
32
       local(g,x);
33
       if (p%4<>1, return ("not a valid prime!"));
       while (1,g=random(p);
34
          if (expmod(g, (p-1)/2, p) = p-1, x = expmod(g, (p-1)/4, p);
35
            if(x>p/2, x=p-x); return(x)));
36
37
       }
38
    /* Euclid algorithm to compute gcd(a,b) but stopping at the first
39
40
    remainder that is < sqrt(a) */
41
42
    EuclSp(a, b) = \{
       local(r, x, z);
43
44
       r=a\%b;
45
       if (r = 1, return([b, 1]));
46
       x=b;
       while (r > sqrt(a), z = x\%r; x = r; r = z);
47
       return ([r, x%r]);
48
49
       }
```

10

```
50
    /* Writes p == 1 \pmod{4} as a sum of two squares, using the algorithm
51
    described in [1] */
52
53
    PrimeSS(p) = EuclSp(p, Imodp(p));
54
55
    /* A method to decompose an odd number as a sum of of either four
56
    integer squares, or half integer squares, in a random way */
57
58
59
    sum4sqF(a) = \{
60
      local(b, c, sq, x, y, z, tt, uu, vv, ct, s);
61
      if (a==1, return(quat(1,0,0,0)));
62
      while (1,
        b=random(2*sqrtint(a)-1)+1; if (b\%2==0,b=b-1);
63
        c = 4 a - b^{2};
64
65
        sq=floor((sqrtint(c)+1)/2);
        j=random(sq-1)+1;
66
        z=(c-(2*j-1)^2)/2;
67
         if (isprime(z), v=PrimeSS(z); x=v[1]; y=v[2];
68
69
           uu=x+y; vv=x-y; tt=2*j-1;
70
             s=matsolve([1,1,1,1;1,1,-1,-1;1,-1,1,-1;1,-1,-1,1],[b;uu;vv;tt]);
             return(quat(s[1,1],s[2,1],s[3,1],s[4,1]))
71
72
           ));
73
      }
74
    /* Expanding the sum4sqF function to all natural numbers with results
75
    only in the integers */
76
77
78
    v2(n) = \{
      local(oddp);
79
80
      e=0;oddp=n;
      while (\operatorname{oddp} \%2 = = 0, \operatorname{oddp} = \operatorname{oddp} / 2; e = e + 1);
81
82
      return(e);
83
      };
84
85
    sum4sqFall(a) = \{
86
      local(r,e);
87
      e = v2(a);
      if (e==0, r=sum4sqF(a), a=a/2^{e}; r=(quat(1,1,0,0)^{e})*sum4sqF(a));
88
89
      while (floor(r[1,1]) < r[1,1]),
              r=r*unid[random(length(unid))+1];
90
91
             );
92
      return(r);
93
      }
94
    /* All possible permutations of the numbers 0, 1, 3, 5 as 4D-vectors */
95
96
```

12

```
perm=List([[0, 1, 3, 5], [0, 1, 5, 3], [0, 3, 1, 5], [0, 3, 5, 1],
97
    [0, 5, 1, 3], [0, 5, 3, 1], [1, 0, 3, 5], [1, 0, 5, 3], [1, 3, 0, 5],
98
    [1, 3, 5, 0], [1, 5, 0, 3], [1, 5, 3, 0], [3, 0, 1, 5], [3, 0, 5, 1],
99
    [3, 1, 0, 5], [3, 1, 5, 0], [3, 5, 0, 1], [3, 5, 1, 0], [5, 0, 1, 3],
100
    [5, 0, 3, 1], [5, 1, 0, 3], [5, 1, 3, 0], [5, 3, 0, 1], [5, 3, 1, 0]]);
101
102
    /* rep135 gives a solution of the system 1-3-5, returning the
103
    Lipschitz integer whose norm is the imput number, and the permutation
104
    of 0135 with whom its inner product is a square */
105
106
107
    rep135(a)={
108
      local(s,t,f,c);
109
      if (issquare(a), return([[sqrtint(a),0,0,0],[1,3,5,0]]));
      while (1, s=sum4sqFall(a);
110
111
        t = [abs(s[1,1]), abs(s[1,2]), abs(s[1,3]), abs(s[1,4])];
112
      for (i=1, length (perm), f=perm[i]*t~;
         if(issquare(f), return([t,perm[i]])));
113
114
      }
```

Acknowledgments

The authors would like to thank Professor Zhì-Wěi Sūn (孙智伟) and Professor Bō Hé (何波) for their kind feedback, helpful comments, and suggestions, and Graça Brites and Vasco Machiavelo for letting their personal computers be used for slices 3 and 4, respectively.

The authors would also like to acknowledge the financial support by FCT — Fundação para a Ciência e a Tecnologia, I.P.—, through the grants for Nikolaos Tsopanidis with references:

PD/BI/143152/2019, PD/BI/135365/2017, PD/BI/113680/2015, and by CMUP — Centro de Matemática da Universidade do Porto —, which is financed by national funds through FCT under the project with reference UID/MAT/00144/2020.

References

- [Bri72] John Brillhart. Note on Representing a Prime as a Sum of Two Squares. *Mathematics of Computation*, 26(120):1011–1013, 1972.
- [MT20] António Machiavelo and Nikolaos Tsopanidis. Zhì-Wěi Sūn's 1-3-5 Conjecture and Variations. arXiv:2003.02592, March 2020.
- [Sūn17] Zhì-Wěi Sūn. Refining Lagrange's Four-Square Theorem. Journal of Number Theory, 175:167–190, 2017.
- [Sūn19] Zhì-Wěi Sūn. Restricted Sums of Four-Squares. International Journal of Number Theory, 15(9):1863–1893, 2019.

Centro de Matemática da Universidade do Porto and Mathematics Department of Faculdade de Ciências do Porto, Rua do Campo Alegre 687, 4169-007 Porto, Portugal

E-mail address: ajmachia@fc.up.pt

Centro de Matemática da Universidade do Porto and Computer Science Department of Faculdade de Ciências do Porto, Rua do Campo Alegre 1021/1055, 4169-007 Porto, Portugal

E-mail address: rvreis@fc.up.pt

Centro de Matemática da Universidade do Porto, Rua do Campo Alegre 687, 4169-007 Porto, Portugal

E-mail address: tsopanidisnikos@gmail.com