The Ulam numbers form an increasing sequence beginning 1, 2 such that each subsequent number can be uniquely represented as the sum of two smaller Ulam numbers. An algorithm is described and implemented in Java to compute the first billion Ulam numbers.

Introduction

At first sight the Ulam numbers appear to be pseudo-random. If this were the case their asymptotic density would be expected to fade towards zero at high numbers as the number of possibilities for forming sums from previous numbers increases. In actuality the density decreases at first but settles around a distribution where about one in 13.5 numbers are in the sequence.

A closer inspection shows that the numbers fall into dense clumps occurring about every 22 integers with sparser breaks between. Steinerberger performed a Fourier analysis on the sequence using the first 10 million numbers and found a clear signal with an angular frequency given by \( \alpha = 2.5714474995 \) [1]. This corresponds to a wavelength of \( \lambda = \frac{2\pi}{\alpha} \approx 2.443443 \ldots \) which is approximately 22/9 so the clumping apparently repeats about every nine wavelengths.

When the frequency of Ulam numbers are plotted against their residue modulo \( \lambda \) they are found to have a non-uniform distribution concentrated in two peaks in the middle third of the wavelength. Figure 1 shows the distribution plotted from the first billion Ulam numbers counted in 1200 bins and normalised so that the vertical axis shows the probability for a large positive integer with a given residue to be Ulam. Ulam numbers whose residue falls outside this middle range are outliers. The outliers are relatively rare. There are only 1828 outliers in the first billion Ulam numbers and empirically the number of outliers less than a given Ulam number is less than its cube root for sufficiently large numbers. However they are important since almost all Ulam numbers are formed from a sum including one outlier and they control the shape of the distribution [2].
Computing the Ulam Numbers

The most straightforward way to compute the Ulam numbers $a_n$ is to build up the sequence from the start testing each subsequent positive integer $t$ to see if it is the sum of two previous Ulam numbers. This can be done by simply taking previous Ulam numbers $a_n < \frac{1}{2} t$ and checking to see if $t - a_n$ is in the list of Ulam numbers so far constructed. The search for each number $t$ can be stopped once two sums have been found but if $t$ is an Ulam number the search will have to continue until all possible sums have been checked. If we assume that the density of Ulam numbers has a constant positive density then the computation time for the first $n$ numbers using this method is $O(n^2)$.

To compute the first billion Ulam numbers in a reasonable timeframe a more efficient method is required. An alternative for testing the number $t$ is to sort the smaller numbers $a_n$ according to their residue $r_n$ modulo $\lambda$. If $t$ itself has a residue $r$ modulo $\lambda$ and if $a_n + a_m = t$ then $r_n + r_m = r$ or $r + \lambda$. This means
that one of the residues $r_n$ or $r_m$ must be in one of the ranges $0 < r_k < \frac{1}{2} r$ or
$\frac{1}{2} (r + \lambda) < r_k < \lambda$. Therefore it is only necessary to test smaller Ulam number
$a_k$ whose residue $r_k$ lies in these ranges to see if it forms a sum.

This search method works for any value of $\lambda$ but if the $\lambda$ we use is (close to) the
recently discovered natural wavelength of the Ulam sequence then the
number of Ulam numbers with residues in these ranges will be much less than
half the number of previous Ulam numbers. If $r$ lies in the central third of the
range as is the case for most Ulam numbers then we only need to test the
subset of the outliers which lie in these ranges to determine if $t$ is Ulam. If $r$
lies outside the central third there will be a section from the denser portion of
the distribution in the ranges, but in this case we usually find two sums very
quickly and can rule out the possibility that $t$ is Ulam. In practice we have
found that only about 5 tests are required on average to determine whether a
number is Ulam using this method.

The efficiency of the algorithm therefore depends on the ability to maintain a
list of the previous Ulam numbers sorted by their residues that can be rapidly
traversed from either end. When each new Ulam number is found it must be
inserted in the list. To find the correct place to insert it quickly we can use a
binary search or maintain an index and once we have found the correct place
to insert we need to form a linked list structure for rapid insertion avoiding the
need to shift up all the subsequent entries in the list.

This can be done by using built in data structures such as Treemaps in Java but
for simplicity and transparency we have used custom structures based on
ordinary arrays. The Java code used is shown in the Annex below.

With this implementation the running time to compute the first billion Ulam
numbers is less than one hour on an ordinary PC. The limiting factor with the
current code which makes is hard to go to higher numbers is memory space
rather than computation speed. With some space optimisations the program
ran on a machine with 16 Gigabytes of RAM and this would need to be
extended in proportion to the number to be calculated. However, memory
optimisations are possible and it should be possible to take the computation to
significantly larger numbers.
**Results**

The value for the wavelength is computed to be $\lambda = 2.44344296778474$ with the corresponding frequency $\alpha = 2.57144749847630$. The largest gap in the first billion Ulam numbers was found to be $966291200 - 966290117 = 1083$.

For the purposes of comparison we provide a table of example Ulam numbers

<table>
<thead>
<tr>
<th>$n$</th>
<th>$a_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>100,000</td>
<td>1,351,223</td>
</tr>
<tr>
<td>1,000,000</td>
<td>13,509,072</td>
</tr>
<tr>
<td>10,000,000</td>
<td>135,160,791</td>
</tr>
<tr>
<td>100,000,000</td>
<td>1,351,856,726</td>
</tr>
<tr>
<td>200,000,000</td>
<td>2,703,579,147</td>
</tr>
<tr>
<td>500,000,000</td>
<td>6,758,780,604</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>13,517,664,323</td>
</tr>
</tbody>
</table>

I would like to thank Jud McCranie for writing an independent version of the algorithm to check these numbers and in the process helping to correct the original code and numbers.

**References**


public class Ulam {

    static int maxn = 1000000000;
    static int a[] = new int[maxn+1]; // list of ulam numbers
    static int nx[] = new int[maxn+1]; // next when ordered by residue
    static int pv[] = new int[maxn+1]; // previous when ordered by residue
    static int k[] = new int[maxn/2]; // true for ulam numbers (packed bits)

    static int nindex = maxn/100;
    static int index[] = new int[nindex];

    static int nbin = 12000; // bin Ulam numbers by residue
    static int bins[] = new int[nbin]; // should be multiple of 3 to separate outliers

    static long kk1=0;
    static long kk2=0;
    static long kk3=0;
    static long kk4=0;
    static long kk5=0;

    static double lamda = 2.44344296778474;
    static double step  = 13.517831473;

    public static void main(String[] args) {
        double alpha = 2.0*Math.PI/lamda;
double lamdarun = lamda;

System.out.println("lamda = "+lamda);
System.out.println("alpha = "+alpha);

initUlam();

// initialise index

for(int i=0; i<nindex; i++) {
    index[i] = 0;
}

pv[0] = 0; // index to number with largest residue
nx[0] = 0; // index to number with smallest residue

setUlam(0,0); // not really an ulam number
setUlam(1,1);
setlinks(1);

setUlam(2,2);
setlinks(2);

int n = 2;
intron = 1;
intron = 0;
long bestgap = 0;

for(long a0 = 3; n < maxn; a0++) {

    // search for a sum in residue order from both ends

double rd0 = mod(a0,lamda)/lamda;

boolean more = true;

int kount2 = 0;

long a1x = 0;

long a1x = 0;

boolean ulam = false;

if(rd0 < 0.24 || rd0 > 0.80) {  // to mind the gap use the brute search

    int j = n;                   // better to start from larger end
    long aj = getUlam(j);
    while(more && 2*aj > a0) {
        kount2++;
        long a1 = aj;
        long a2 = a0-a1;
        kk3++;
        if(isUlam(a2)) {
            if(ulam) {  // found more than one sum

ulam = false;
more = false;
} else {
    ulam = true;
    a1x = a2;
}
}

j--;
aj = getUlam(j);
}
more = false;
}

int kount0 = 0;

int i = nx[0];  // start with smallest residue
long ai = getUlam(i);
double rdi = mod(ai,lamda)/lamda;
while(2*rdi <= rd0+0.0002 && more && i != 0) {
    kount0++;
    long a2 = a0-ai;
    kk1++;
    if(isUlam(a2) && ai != a2 && a2 != a1x) {    // pair adds up
        if(ulam) {              // already had a sum
            more = false;        // found two so can stop
            ulam = false;
        } else {                // otherwise note first sum
            } else {
        } else {
    } else {
}
ulam = true;
alx = ai;            // note this to check against double counting
}

i = nx[i];            // jump to next smallest residue
ai = getUlam(i);
rdi = mod(ai, lamda)/lamda;
}

int kount1 = 0;
i = pv[0];     // now work back from the largest residue
ai = getUlam(i);
rdi = mod(ai, lamda)/lamda;
while(2*(1.0-rdi) <= (1.0 - rd0)+0.0002 && more && i != 0) {
    kount1++;
    long a2 = a0-ai;
    kk2++;
    if(isUlam(a2) && ai != a2 && a2 != a1x) {   // pair adds up
        if(ulam) {
            more = false;    // already had a sum
            ulam = false;
        } else {
            ulam = true;     // otherwise note first sum
            alx = ai;        // note this to check against double counting
        }
    }
}
i = pv[i];            // jump to next largest residue
ai = getUlam(i);
rdi = mod(ai, lamda)/lamda;
}

if(ulam) {
n++;
setUlam(a0, n);

double z = mod(a0, lamda)/lamda;
setlinks(n);

long d = (long)(a0/lamda);
double p = 0.0;
if(z < 1.0/3.0) {
nor++;
p = a0/(d+1.0/3.0);
}

if(z > 2.0/3.0) {
nol++;
p = a0/(d+2.0/3.0);
}

if(z > 2.0/3.0 || z < 1.0/3.0) {
lamdarun = (lamdarun*9.0+p)/10.0;
//System.out.println(nor+" "+nol+" "+n+" "+a0+" "+z+" "+p+" "+lamdarun);
System.err.println(n+" "+a0+" kk: "+(kk1/a0)+" "+(kk2/a0)+" "+
(kk3/a0)+" "+(kk4/a0)+" "+(kk5/a0));
if (n%10000000 == 0) {
    System.out.println("a["+n+"] = "+a0+" = "+a1x+"+(a0-a1x));
    System.err.println("a["+n+"] = "+a0+" = "+a1x+"+(a0-a1x));
}

long a1 = getUlam(n-1);
long gap = a0-a1;

if (gap > bestgap) {
    bestgap = gap;
    System.out.println(n+" "+a0+" - "+a1+" = "+gap+" is bigger gap");
}

// build distribution by residue in bins

int ibin = (int)(z*nbin);
bins[ibin]++;

}

System.out.println("a["+n+"] = "+getUlam(n));
System.out.println("biggest gap was "+bestgap);

double density = ((double) n)/((double) getUlam(n));
System.out.println("density = "+density);
System.out.println("step = "+(1.0/density));

System.out.println("\n");
System.out.println("bin frequencies:");
for(int ibin=0; ibin<nbin; ibin++) {
    System.out.println(ibin+","+bins[ibin]);
}

cchecklinks(n);
}

public static double mod(long x, double m) {
    double dx = x;
    double z = dx/m;
    long iz = (long) z;
    z -= iz;
    z *= m;

    return z;
}

public static void setlinks(int n) {
    // set the next and previous links in ordering by residue

    // use an index to find a starting point with a lower residue
double rdn = mod(getUlam(n), lamda)/lamda;
int j = (int)(nindex*rdn);
int pvi = index[j];

boolean more = true;
while (more) {
    kk4++;
    int i = nx[pvi];
    long ai = getUlam(i);
    double rdi = mod(ai, lamda)/lamda;
    if (i == 0) {
        more = false;
    } else if (rdi < rdn) {
        pvi = i;
    } else {
        more = false;
    }
}

int nxi = nx[pvi];

pv[n] = pvi;
nx[pvi] = n;

nx[n] = nxi;
pv[nxi] = n;

// update index
j++; 
double rdi = 0;
if(j < nindex) rdi = mod(getUlam(index[j]),lamda)/lamda;
while( j < nindex && rdi < rdn ) {
    kk5++;
    index[j] = n;
    j++;
    if(j < nindex) rdi = mod(getUlam(index[j]),lamda)/lamda;
}

static void checklinks(int n) {
    int pvi = 0;
    int m = 0;
    while(nx[pvi] != 0) {
        if(pv[nx[pvi]] != pvi) System.err.println("links are inconsistent at "+
pvi+" -> "+nx[pvi]+" <- "+pv[nx[pvi]]);
        if(nx[pv[pvi]] != pvi) System.err.println("links are inconsistent at "+
pvi+" <- "+pv[pvi]+" -> "+nx[pv[pvi]]);
        double rdi = mod(getUlam(pvi),lamda)/lamda;
        double rdn = mod(getUlam(nx[pvi]),lamda)/lamda;
        if(rdi > rdn) System.err.println("links are not ordered at "+
pvi+","+nx[pvi]+" "+rdi+" > "+rdn);
        m++;
        pvi = nx[pvi];
if(m != n) System.err.println("link list is wrong length "+m+" != "+n);
System.err.println("links check complete");

// booleans flagging the ulam numbers are packed to save space

static int pow2[] = {1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768,
                     1<<16, 1<<17, 1<<18, 1<<19, 1<<20, 1<<21, 1<<22,
                     1<<23, 1<<24, 1<<25, 1<<26, 1<<27, 1<<28, 1<<29};

public static boolean isUlam(long a0) {
  long i30 = 30;
  int m30 = (int) (a0%i30);
  int d30 = (int) (a0/i30);
  boolean ulam = ((k[d30] & pow2[m30]) > 0);
  return ulam;
}

public static void setUlam(long a0, int n) {
  long i30 = 30;
  int m30 = (int) (a0%i30);
  int d30 = (int) (a0/i30);
  boolean ulam = ((k[d30] & pow2[m30]) > 0);
  return ulam;
}
a[n] = an;
}

public static long getUlam(int n) {
    double dn = (double) n;
    long ground = (long) (dn*step);
    long a0 = ground+a[n];
    return a0;
}

public static void initUlam() {
    for(int i=0; i<k.length; i++) k[i] = 0;
    for(int i=0; i<bins.length; i++) bins[i] = 0;
}

}