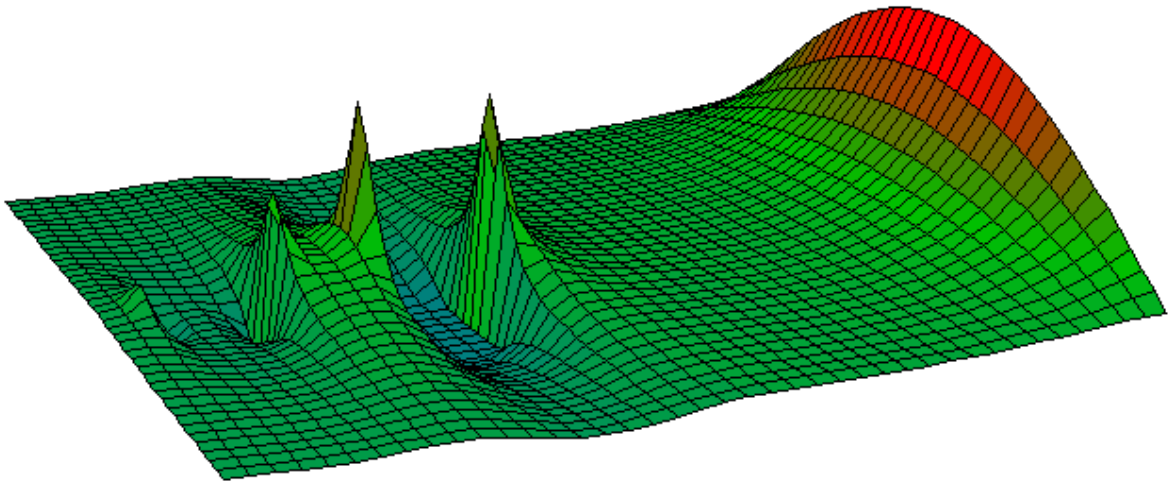
Journal of J

An interdisciplinary journal on J programming language and applications in science and technology.

New web site www.journalofj.com



Please Circulate

Journal of J.

MPM press

An open access Journal

ISSN: 2174-9280

Vol.2, No.1 July 2013

JoJ (J2-team)

J, a language for the science, technology and more…

Aims and scope

Journal of J is a **not for profit** journal , involving a large research and users community in J programming language.

Journal of J is an interdisciplinary **journal** devoted to J and science and technology.

Journal of J aims to provide fast access to papers about science, technology and J.

Journal of J embodies the following principle:

Open Access: Knowledge is a public good. All readers have open access to reading and downloading papers. The simple and free access ensures maximum readership and high citation records for published papers.

Contact: journalofj at hotmail dot com (CONTRIBUTIONS)

info at journalofj dot com (GENERAL)

mikelpater at hotmail dot es (EDITOR)

Style and Contents

Journal of J aims to cover all the main areas of science. Inevitably, articles in different areas are addressed at different audiences. Many of the articles submitted to the journal are standard technical pieces, addressed to a purely academic audience

To attract this variety of contributions *Journal of J* will contain the following areas:

- Mathematics: number theory, logic, calculus, algebra, arithmetic, algorithms and others…
- Physics: dynamical systems, chaos, fractals, disorder, statistical physics and others…
- Computer science, Visualization, Engineering, Computer Art, …
- **others about J and J applications**

Visit: www.journalofj.com **Edited by Mikel Paternain**

Editorial

Comenzamos este año estrenando nueva web donde podemos encontrar todos los trabajos publicados en la revista en un formato más profesional y agradable. No obstante, mantendremos, por motivos históricos, activo el antiguo google site.

Una vez más debemos agradecer a los colaboradores sus interesantes contribuciones que ponen de manifiesto la elegancia y eficacia del lenguaje J en los ámbitos más variados de la ciencia y la técnica.

Confiamos en que el proyecto siga ganando adeptos y podamos continuar este 2013 con más números de la revista.

We started this year launched a new website where you can find all the papers published in the magazine in a more professional and pleasant format.

However, we keep active for historical reasons the old google site.

Once again we must thank to the collaborators and their interesting contributions highlight the elegance and efficiency of J language in the most varied fields of science and technology.

We believe that the project continues to gain followers and we can continue this 2013 with more issues of the magazine.

mikelpater at Hotmail dot es

Prime Factorization in the Gaussian integers

Cliff Reiter

Lafayette College, Department of Mathematics, Easton PA, 18042 U.S.A.

The Gaussian integers are complex numbers $a+bi$ where *a* and *b* are integers and $i=\sqrt{-1}$. It is well known that the Gaussian integers have a multiplicative structure very similar to the integers [3]. In particular, non-zero Gaussian integers factor into a product of primes in a fashion analogous to the factorization of non-zero integers into a product of primes. During the summer of 2012 research students and I were trying to take advantage factorizations into Gaussian integers and I wanted to do it in J. While implementing that some interesting computational problems arose and while they could be solved, I thought there must be a better way. We ended up using an unrelated approach for our research [8]. However, this note is an excuse to revisit the problem of factoring Gaussian integers into Gaussian primes and then we will use the resulting verb to create an interesting picture of the positions of the Gaussian primes in the complex plane.

1. A Brief Introduction to Arithmetic in the Gaussian Integers

We note that J has built-in complex arithmetic so that we are freed from worrying about most of the details. Addition, multiplication and division can be done treating *i* as a formal symbol such that $i^2=-1$. The complex number $3+4i$ is denoted 3j4 in J. Its conjugate is $3-4i$ which can be computed by +3j4 and its norm is the product of the number with its conjugate which the same as the sum of the squares of its real and imaginary parts. A J script containing all the J execution expressions in this note may be found at [5] while a script containing only the functions may be found at [6].

```
   + 3j4
3j_4

   3j4*+3j4
25

   N=: *+

   N 3j4
25

   3j4 + 2j3
5j7

   0j1 * 0j1
_1
```

```
   0j1 * 1j2
_2j1

   3j4 * 1j2
_5j10

   3j4 % 1j2
2.2j_0.4
```

Notice above we see that the quotient of Gaussian integers may require fractions. Expressing a Gaussian integer quotient as a Gaussian integer plus a remainder is more complicated than for ordinary integers, but can be done. However, we do not need to worry about those details since we are really interested in greatest common divisors and J's `+.` works fine for Gaussian integers.

```
   */1j1 3j_4 5j12
47j79

   */1j1 1j_1
2

   2 +. 47j79
1j1
```

Gaussian integers with norm 1 are called units and these are ± 1 and $\pm i$.

```
   N 1 0j1 _1 0j_1
1 1 1 1

   3j4 * 1 0j1 _1 0j_1
3j4 _4j3 _3j_4 4j_3
```

Notice that multiplying a Gaussian integer by the units results in signs and interchanges of real and imaginary parts. It is traditional to take a Gaussian prime with nonnegative real and imaginary parts as a representative. Notice the conjugate `3j_4` is not in the list. Hence, it is not the same as `3j4` up to multiplication by a unit; however, it is the same as `4j3` up to a unit.

The norm of a complex number, $N(a+bi) = (a+bi)(a-bi) = a^2 + b^2$ has the useful property that it is multiplicative. That is, for complex numbers α and β, $N(\alpha\beta) = N(\alpha)N(\beta)$. When α and β are Gaussian integers, the norms are integers.

A Gaussian prime is a Gaussian integer with only trivial factorizations into Gaussian integers; that is, if a Gaussian prime γ factors in Gaussian integers as $\gamma = \alpha\beta$ then one of α or β must be a unit. When we get to the point of factoring a Gaussian integer into Gaussian primes we will first factor the norm into integer primes and then look for Gaussian integers with those prime norms or their squares, as appropriate.

2. Primes in the Gaussian Integers

The primes in the Gaussian integers are well known [2,3,4,9]. The Gaussian primes of norm 2 are $\pm i \pm 1$ but as per the previous remarks, that means up to a unit factor, the only prime of norm 2 that we will use is $1+i$. It is a classic theorem of number theory that an odd prime is a sum of two

squares if and only if it is congruent to 1 modulo 4. Thus, integer primes of the form $p = 4k + 3$ are also Gaussian primes with norm $N(p) = p^2$. If $p = 4k + 1$, then there exist integers *a* and *b* such that $p = a^2 + b^2$. The Hermite-Serret algorithm for determining such *a* and *b* proceeds by first finding a solution to $x^2 \equiv -1 \bmod p$. Note that $N(x \pm i) = x^2 + 1 = (x + i)(x - i) \equiv 0 \bmod p$. Since *p* divides the product $(x + i)(x - i)$ but neither factor, the $\gcd(p, x + i)$ will be a nontrivial Gaussian integer with norm *p*. Thus, either it or its conjugate (or both) will divide the Gaussian integer we are trying to factor.

In order to solve $x^2 \equiv -1 \bmod p$ we select a random nonzero *n* mod *p* and compute $y = n^{\left(\frac{p-1}{2}\right)} \bmod p$ and $x = n^{\left(\frac{p-1}{4}\right)} \bmod p$ (this can be done very efficiently). It is known that *y* must be congruent to 1 or -1 with each occurring half the time. If $y \equiv -1$, then *x* is as desired. Otherwise we recompute these for another *n* until we obtain the desired *x*. Comparisons of various methods for finding the solutions to $p = a^2 + b^2$ that include the Hermite-Serret algorithm may be found at [1, 7]. I am not aware of when the probabilistic exponential aspect of this algorithm first appeared, but it was included in the analysis in [7].

3. Implementation of Gaussian Integer Prime Factorizations

We begin the implementation by defining `qrw` (quadratic residue witness) that finds a solution to $x^2 \equiv -1 \bmod p$ for $p \equiv 1 \bmod 4$.

```
   qrw=: 3 : 0
p=.x: y
e=.(p-1)%2 4
y=.0
while. y~:p-1 do.
  n=.x: ?&.<:p
  'y x'=.p&|@:(n&^)e
end.
x
)
   qrw 101
91
```

Next we compute $\gcd(p, x+i)$ with the verb `splitm1` and see that we get both elements of norm p with positive real and imaginary parts as below. Note that `splitm1` is based on the probabilistic verb `qrw` so that different evaluations may give the results in reverse order.

```
   splitm1=: (,0j1*+)@(+. qrw j. 1:)"0
      splitm1 13
3j2 2j3
```

Now we work a partial example by beginning to factor $-13090-8806i$. Below we see that the prime factors of the norm are 2, 7, 13 and 17. We organize according to whether they are congruent to 1, 2 or 3 modulo 4. Each 2 corresponds to one factor of $1+i$. Each (equal) pair of primes congruent to 3 corresponds to one prime factor. We see the possible Gaussian prime factors of the primes congruent to 1 (up to units) are listed in `sp1`. All that remains is to trial divide as far as possible with each of the Gaussian primes in `sp1`.

```
   y=. _13090j_8806

   ]pn=.q: N y
2 2 2 7 7 13 13 13 17 17

   'p1 p2 p3'=.}.&.>(4&| </.])(1 2 3),pn

   p1
13 13 13 17 17

   p2
2 2 2

   p3
7 7
```

```
   ]pf=.(#p2)#1j1
1j1 1j1 1j1

   ]pf=.pf,;<@(-:@#{.])/.~p3
1j1 1j1 1j1 7

   ]sp1=.,splitm1 ~.p1
3j2 2j3 1j4 4j1

   ]y=.y%*/pf
153j782
```

The function `gifactor` below assembles those steps and does the trial division. It orders the prime factors by magnitude and prepends the unit required so that the product of the result is the input. Before we give its definition we define verbs that test whether a number is an integer, test whether a complex number is a Gaussian integer, and clean the results of complex numbers that are zero up to round-off error in real or imaginary parts.

```
   iq=: =<.
   iq 3 3.1
1 0

   giq=:*./@:iq@:(**|)@:+."0
   giq 3j4 3.1j4
1 0
   giclean=:(**|)&.+.
   giclean 3j1e_16
3
  gifactor=:3 : 0
pn=.q: N y
'p1 p2 p3'=.}.&.>(4&| </. ])1 2 3,pn
```

```
pf=.(#p2)#1j1
pf=.pf,;<@(-:@#{.])/.~p3
sp1=.,splitm1 ~.p1
y=.giclean y%*/pf
for_sp. sp1 do.
   while. giq q=.y%sp do.
   y=.giclean q
   pf=.pf,sp
   end.
end.
y,(/:|)pf
)
 gifactor _13090j_8806
_1 1j1 1j1 1j1 2j3 2j3 2j3 4j1 1j4 7
```

We should note that since complex numbers in J have real and imaginary parts represented via floating point numbers, and norms are proportional to the square, we should be wary of applying this implementation when the real or imaginary parts have more than about 7 digits.

4. Gaussian Prime Query

When we apply `gifactor` to a Gaussian prime we get a unit and the prime in standard form. Thus, a Gaussian integer is prime exactly when there are two elements in the result of `gifactor`. The verb `giprimeq` defines a Boolean test based on that idea. In order to draw a plot of the positions of the Gaussian primes we define a verb `gipq` that gives 0 for input 0 and the result of `giprimeq` otherwise.

```
   giprimeq=:2 = #@gifactor
   gipq=:0:`giprimeq@.(0&~:)"0
   gipq 0 1 2 3 4 5 1j4
0 0 0 1 0 0 1
```

Next we look at a matrix of complex numbers around the origin and then a matrix showing the positions of the Gaussian primes. We then view the patterns that arise on larger matrices.

```
   j./~i:5
_5j_5 _5j_4 _5j_3 _5j_2 _5j_1 _5 _5j1 _5j2 _5j3 _5j4 _5j5
_4j_5 _4j_4 _4j_3 _4j_2 _4j_1 _4 _4j1 _4j2 _4j3 _4j4 _4j5
_3j_5 _3j_4 _3j_3 _3j_2 _3j_1 _3 _3j1 _3j2 _3j3 _3j4 _3j5
_2j_5 _2j_4 _2j_3 _2j_2 _2j_1 _2 _2j1 _2j2 _2j3 _2j4 _2j5
_1j_5 _1j_4 _1j_3 _1j_2 _1j_1 _1 _1j1 _1j2 _1j3 _1j4 _1j5
 0j_5  0j_4  0j_3  0j_2  0j_1  0  0j1  0j2  0j3  0j4  0j5
 1j_5  1j_4  1j_3  1j_2  1j_1  1  1j1  1j2  1j3  1j4  1j5
 2j_5  2j_4  2j_3  2j_2  2j_1  2  2j1  2j2  2j3  2j4  2j5
 3j_5  3j_4  3j_3  3j_2  3j_1  3  3j1  3j2  3j3  3j4  3j5
 4j_5  4j_4  4j_3  4j_2  4j_1  4  4j1  4j2  4j3  4j4  4j5
 5j_5  5j_4  5j_3  5j_2  5j_1  5  5j1  5j2  5j3  5j4  5j5
```

```
   gipq j./~i:5
0 1 0 1 0 0 0 1 0 1 0
1 0 0 0 1 0 1 0 0 0 1
0 0 0 1 0 1 0 1 0 0 0
1 0 1 0 1 0 1 0 1 0 1
0 1 0 1 1 0 1 1 0 1 0
0 0 1 0 0 0 0 0 1 0 0
0 1 0 1 1 0 1 1 0 1 0
1 0 1 0 1 0 1 0 1 0 1
0 0 0 1 0 1 0 1 0 0 0
1 0 0 0 1 0 1 0 0 0 1
0 1 0 1 0 0 0 1 0 1 0
```

```
load 'viewmat'

viewmat -.gipq j./~i:10

viewmat -.gipq j./~i:40
```

Figure 1 shows the Gaussian primes in black where position corresponds to all Gaussian integers with real and imaginary parts less than 10. Figure 2 shows the result when the parts are less than 40. Notice that the primes seem to form interesting patterns.

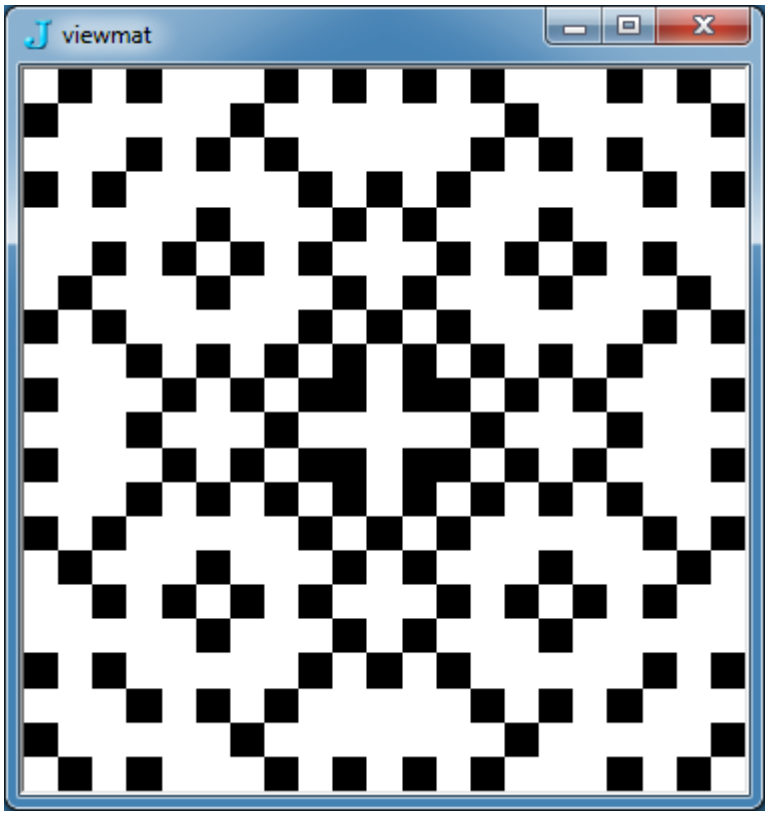


Figure 1. Gaussian primes (black) with real and imaginary parts less than 10.

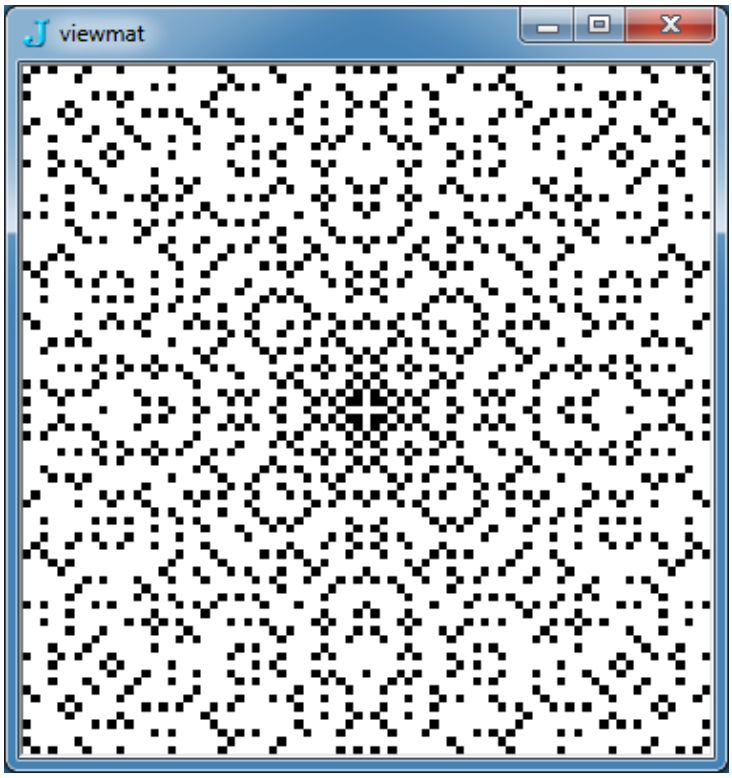


Figure 2. Gaussian primes (black) with real and imaginary parts less than 40.

References

[1] J. Brillhart, Note on Representing a Prime as a Sum of Two Squares, *Mathematics of Computation*, v26, 1972, pp 1011-1013.

[2] H. Davenport, *The Higher Arithmetic, An Introduction to the Theory of Numbers,* Harper 1960, Dover reprint, 1983.

[3] G. H. Hardy & E. M. Wright, *An Introduction to the Theory of Numbers*, 6th edition, Oxford University Press, 2008. Revised by D. R. Heath-Brown and J. H. Silverman.

[4] H. Pollard and H. G. Diamond, *The Theory of Algebraic Numbers*, Dover reprint, 1998.

[5] C. Reiter, Prime Factorization in the Gaussian integers, J expressions, https://webbox.lafayette.edu/~reiterc/j/JoJ/gi_factor_notes.html, June 2013.

[6] C. Reiter, Prime Factorization in the Gaussian integers, J functions, https://webbox.lafayette.edu/~reiterc/j/JoJ/gi_factor.html, June 2013.

[7] D. Shanks, review of: A table of Gaussian Primes, by L. G. Diehl & J. H. Jordan, *Mathematics of Computation*, v21, 1967, pp 260-262.

[8] B. D. Sokolowsky, A. G. VanHooft, R. M. Volkert & C. A. Reiter, An Infinite Family of Perfect Parallelpipeds, *Mathematics of Computation*, to appear; preprint: https://webbox.lafayette.edu/~reiterc/nt/ppinf/ppinf2_preprint.pdf.

[9] S. Wagon, *Mathematica in Action*, W.H. Freeman and Company, 1991.

BREAKING SIMPLE SUBSTITUTION CIPHERS

John Dixon

jdixon AT math DOT carleton DOT ca

1. Introduction

A very classical form of sending a secret message is to use a simple substitution cipher. The sender and receiver together choose a permutation p of the alphabet $Alpha = ABC...Z$ and the sender enciphers the message by replacing each letter by the corresponding letter of the permutation. The receiver, knowing p, can then decipher the message. For example, if p corresponds simply to a reversal of the alphabet then *HELP* is enciphered as *SVOK*. The problem which we consider here is how we can "break" a cipher of this kind and read the message when we do not know the permutation which has been chosen. Since there are 26!=403291461126605635584000000 choices for p obviously we cannot try every one.

The basis of cryptanalysis in general is use of redundancy, regularity in the underlying language. In the present article we assume that that we are working in English but the technique works equally well in any alphabetic language and the program in Sec. 3 can be easily modified to work with other languages. One regularity is the relative frequency of letters. In English the letters ordered in decreasing frequency in a typical piece of text are *ETAOINHSRDLUMWFCGYPBKVXZJQ*. Already *E* occurs about 12% of the time and over 50% of the time the letters are from the list *ETAION*. However, although the frequency distribution of single letters is helpful in breaking a simple substitution cipher, it is more effective to work at a second level: the frequency of digraphs (consecutive pairs).

Remark We shall assume that we do not know where the word breaks are in the cipher text; the analysis is a little easier if we do know word breaks. In the latter case *Alpha* should be replaced by *Alpha,''* and our frequency tables of digraphs will be 27×27.

The idea of the method is this (we follow the paper [1]). To obtain the statistics of frequencies of digraphs in English we downloaded a large text file *T* (in our case we chose Raymond Chandler's *Big Sleep*), deleted all spaces, numbers and other punctuation, changed all alphabetic characters to upper case, and then counted the frequencies of digraphs. This gave us a 26×26 matrix, say *F*, whose *(i,j)*th entry is equal to the number of times that the *i*th letter is followed by the *j*th letter of the standard alphabet. For example, the *(3,0)*th entry of *F* is the number of times that the digraph *DA* appeared in *T*. The entries of *F* are roughly proportional to the relative frequencies in which corresponding digraphs appear in the English language.

We now take our cipher text *C* which we assume has been constructed from an (unknown) message by enciphering with an (unknown) permutation p, and compute a second 26×26 matrix *M* whose

entries are the frequencies of the digraphs which appear in *C*. Given any permutation *q* of the integers 0...25 we want to measure how plausible it is that $q = q\{Alpha$ is the deciphering permutation (the inverse of *p*). If *C=c[0]c[1]...c[n]* it turns out via a Bayesian argument (see [1]) that this plausibility may be measured by

$$\text{Plaus}(q) = \prod \text{Freq}(q[c[i]],q[c[i+1]])$$

where the product is over $i = 0,...,n-1$ and Freq(*X,Y*) is the relative frequency that the digraph *XY* occurs in the English language. Let M_q be the matrix obtained from *M* by simultaneously permuting the rows and columns according to *q*. If we replace Freq(*X,Y*) by the number of times which the digraph *XY* occurs in T (the latter should be roughly proportional to Freq(*X,Y*)) and take logarithms, we obtain the *log plausibility* measure

$$LP(q) = \sum M_q[i,j]\log F[i,j]$$

where the sum is over all $i,j = 0,...,25$. A high value of *LP(q)* signals a permutation *q* which is "plausible". Indeed it is obvious that *LP(q)* will be large when the large entries of M_q are matched to the large entries of *F,* and it possible to justify the argument that a *q* which makes *LP(q)* largest is a good guess for deciphering the message correctly. (The alert reader will have noticed that we have taken the raw frequencies in M_q and *F* rather than relative frequencies, but this will not affect our objective which is to find *q* to maximize *LP(q)*.)

We now turn to the problem of finding a permutation *q* to maximize *LP(q)*. This is a hard problem, so we use a heuristic algorithm, known as the Metropolis algorithm, to search through the set of 26! permutations for an approximate solution. The idea is as follows. Imagine that you are walking through the mountains and are trying to find the highest peak. If you find a point nearby which is higher than your current point then you move to that point. On the other hand all nearby points may be lower (you could be at a local maximum but not the absolute maximum), so you must be prepared sometimes to descend.

In terms of permutations, if our present position is the permutation *q* then we randomly generate a nearby permutation *q'*. If *LP(q) < LP(q')* then we move to *q'*. On the other hand if *LP(q') < LP(q)* then we may move to *q'* with a probability which depends on how large *LP(q)* is compared with *LP(q')*; otherwise we stay at *q* and try another nearby permutation.

More precisely, assuming that our current guess is *q*, then the general step is as follows ($\varkappa$ is a parameter > 0):

- compute a candidate permutation *q'* from *q* by multiplying *q* by a random 2-cycle *(i,j)* (that is, interchange the *i*th and *j*th entries of *q* to obtain *q'*)
- if *LP(q') > LP(q)* then accept the candidate *q'* as the new current guess

- if $LP(q') \leq LP(q)$ then with probability $exp((LP(q')-LP(q))/\varkappa)$ accept q' as the new current guess, otherwise keep q as our current guess.

We shall keep repeating this step until we are successful (find that we can read the message), or until we observe no further progress. We may wish to vary $\varkappa$ in the range $1 \leq \varkappa < 10$, say. When the values of $\varkappa$ are larger, the search is more likely to be spread wider (useful if the search is stuck near a permutation which is not near the solution); as the values of $\varkappa$ decrease the search becomes more concentrated.

Because of the power of J and its interactive possibilities, J is an excellent language for this kind of textual analysis. We present an implementation of the Monte Carlo algorithm in J in Sec. 3, but first give some examples to show how surprisingly well it works.

Remarks The classic book [2] gives an authorative and interesting account of the state of cryptology and cryptanalysis up to the 1960s. The book [3] is an excellent description written at a level suitable for upper high school and college students of the use of elementary mathematics in constucting and analysing ciphers; during World War II the author headed the US Communications Intelligence Organization whose job was to intercept and decode Japanese messages in the Southwest Pacific theater.

The field of cryptology has completely changed since the 1960s largely due to the invention of asymmetric systems. The latter include trap-door functions which encrypt your password so that the recipient can authenicate you without knowing or being able to use your password, and public-key cryptosystems where an enciphering key can be made publically available but a message enciphered by this key can only be read by a person who knows the private deciphering key. A web search for: trapdoor function, public-key cryptosystem or RSA system will give more information. A description of the RSA-system is given in the 2nd edition of [3].

2. **Examples**

In constructing examples, it is worth noting that the program *MonteCarlo* given in Sec. 3 begins by applying a random permutation *perm=: 26?26* to the original ciphertext before any analysis is applied. This means that to test the program we can read in a plain text as our cipher text: the plain text is first automatically enciphered with a random permutation before the program starts trying to break the cipher. Because the program in Sec. 3 uses ? (*roll*), we obtain different outputs if we run it with the same input more than once.

1. For our first example we loaded a text string which we have called *text1* (about 600 characters from the introduction of *J for C Programmers*) and called
   perm =: perm MonteCarlo text1;1;"
   the output read
   THIS POOZ WIDD TEDD BOU ENOUGH APOUT K COR BOU TO USE IT AS A DANGUAGE

COR YEVEDOFING SERIOUS AFFDIL
THIS BOOZ YIDD TEDD POU ENOUGH ABOUT K COR POU TO USE IT AS A DANGUAGE
COR MEVEDOFING SERIOUS AFFDIW
THIS BOOJ WILL TELL YOU ENOUGH ABOUT K POR YOU TO USE IT AS A LANGUAGE
POR DEVELOFING SERIOUS AFFLIC
THIS BOOZ WILL TELL YOU ENOUGH ABOUT K FOR YOU TO USE IT AS A LANGUAGE
FOR DEVELOPING SERIOUS APPLIC
THIS BOOJ WILL TELL YOU ENOUGH ABOUT K POR YOU TO USE IT AS A LANGUAGE
POR DEVELOFING SERIOUS AFFLIC

which gives an almost complete deciphering of the text.

The current value of *perm* is that used to provide the last line of output. It can be used with the function *Decode* in Sec. 3 to decipher the whole of *text1*.

2. For out second example, we loaded the string *text2* (a selection of about 800 characters from an earlier issue of the *Journal of J*) and called

   perm =: perm MonteCarlo text2;1;''

   several times (*perm* is the deciphering permutation which changes and must be retained from one stage to the next). Each call of MonteCarlo gives five lines of output but this time we only list the final line of output. From the first two calls to this function we obtained the output

   NDE PTHTOTS WELL IHTYOR I FTSON SO I RSAWHENE WTLLEWNSTO TG FTSONA SA NDE
   NDE PTHTSTA RELL IHTYSW I UTASN AS I WAORHENE RTLLERNATS TM UTASNO AO NDE

   Since we were not making any obvious progress we decided to increase the parameter *c* from 1 to 4 to move away from what is apparently an unproductive local maximum, and called

   perm =: perm MonteCarlo text2;4;''

   This seemed to make no progress either, so we decided to choose a new starting point by redefining

   perm =: 26?26

   before our next call. We immediately obtained as final line of our output
   THE VORONOI CELL AROUND A POINT IN A DISCRETE COLLECTION OF POINTS IS THE

3. For the third example we loaded a string *text3* (a passage of about 600 characters from *Wikipedia*) and called
   perm =: perm MonteCarlo text3;1;''
   The last line of output for the first two calls gave
   ON THE MOLW BINF OU FOURATDHEF TI GARAN AMTES AWESODAN ANF UIVOET WANNEF URADEDSAMT FOUARREAS WYUTE
   ON THE FOLM BIND OU DOUSATWHED TI VASAN AFTER AMEROWAN AND UIPOET MANNED USAWEWRAFT DOUASSEAR MYUTE
   Looking at this output we guess that perhaps the characters '*LMNTHEAFRD*' have been correctly deciphered, so we fix these characters by calling
   perm =: perm MonteCarlo text3;1;'LMNTHEAFRD'
   several times which soon yields
   IN THE FILM WOND IS DISPATCHED TO XAPAN AFTER AMERICAN AND SOBIET MANNED SPACECRAFT DISAPPEAR MYSTE
   We now fix more characters and call
   perm =: perm MonteCarlo text3;1;'LMNTHEAFRDIPCY'

and obtain
IN THE FILM BOND IS DISPATCHED TO KAPAN AFTER AMERICAN AND SOVIET MANNED SPACECRAFT DISAPPEAR MYSTE

4. As a challenge you may wish to solve Poe's classic Gold Bug cipher. The cipher text is given in the appendix and will need to be translated into a text of alphabetic characters before you try to solve it.

3. J Program

```
NB. Cryptography: for a monoalphabetic substitution cipher.   Uses a Monte
Carlo method
NB. described by Persi Diaconis in Bull. Amer. Math. Soc. 46 (2009) 179-205
NB. Assumes that the enciphered text is written in the standard alphabet (case
      independent) and that
NB. the plain text is in English. It is *not* assumed that the breaks are
      natural word breaks.


      ReadFile =: 1!:1 @: <
      LowAlpha =: (97+i.26){a.     NB. 'ab...z'
      UppAlpha  =: (65+i.26){a.    NB. 'AB...Z'
      Alpha2Num =: LowAlpha&i. <. UppAlpha&i.
            NB. Replaces letters by numbers 0...25 and other symbols by 26
      CP =: ((< @: ,)"0)/    NB. Cartesian product of lists x and y


NB. The frequency table for pairs of letters is stored in file_1
=:'c:\...\freqcount.txt'

NB. It was computed using the functions Alpha2Num and FreqTable below.
NB. The table is a count of the digraphs in the book "The Big Sleep" by Raymond
      Chandler
NB.  (about 250,000 alphabetic characters) which is available through the
      Gutenberg Project

NB. scaled by a factor of 10.  A copy of  freqcount will be found in the
      Appendix (Sec. 5).


      StandardLogFreq =: (26 26) $ ^. (0.05 + ". ReadFile file_1)
            NB. 0.05 fudge to ensure that log is finite
      Interchange =: <@] C. [

      NB.  list Interchange i,j   interchanges ith and jth items
      Decode =: 4 : '(x i. Alpha2Num y) {27{. UppAlpha'
      NB. Decodes y (text string) using permutation x
      FreqTable =: 3 : '(26 26)$ +/"1 (,CP ~i.26) =/ 2 <\ (y < 26)#y'
      NB. Frequency table of consecutive pairs in list of integers from 0...26
      NB. after deleting all occurences of 26
      perm =: 26 ? 26
      NB. random permutation
```

```
MonteCarlo =: 4 : 0
NB. Applies 2500 steps of MonteCarlo process to find deciphering
      permutation.

NB. Prints out tentative text every 500 steps.
NB. Usage:     perm =: perm MonteCarlo text;c;fix
NB. text  is the cipher text
NB. c  is parameter for convergence (usually between 1 and 10)

NB. fix  is a list of letters which appear correctly deciphered

NB. (and will remain unchanged in this call of MonteCarlo).
perm =. x
'txt c fix' =. y
ntext =. , Alpha2Num txt
left =. (i.26) -. UppAlpha i. ~.fix   NB. the letters not yet fixed
ft =. FreqTable ntext
P =: +/ , (perm{perm{"1 ft)*StandardLogFreq
for. i.5 do.
   for. i.500 do.
      p =. perm Interchange (2?#left){left
      t =. +/ , (p{p{"1 ft)*StandardLogFreq
      if. (?0)<^(t-P)%c do.
         perm =. p
         P =. t
      end.
   end.
   smoutput perm Decode 100 {. txt
end.
val =. perm
)
```

4. References

[1] Persi Diaconis, "The Markov chain Monte Carlo revolution", *Bull. Amer. Math. Soc.* **46** (2009) 179-205.

[2] David Kahn, "The Codebreakers: The Story of Secret Writing", Macmillan, New York, 1967.

[3] Abraham Sinkov, "Elementary Cryptanalysis", Math. Assoc. Amer., 1967 (2nd ed revised and updated by Todd Fiel, Cambridge Univ. Press, 2009).

5. Appendix

A list of frequency counts (scaled by 10) of 26^2 digraphs in *The Big Sleep* used in the program in Sec. 3:

freqcount =: 1 38 83 80 0 17 43 11 95 1 22 105 43 297 1 31 1 153 149 169 12 34 26 1 48 2 35 5 0 0 70 0 0 1 10 0 0 35 0 0 48 0 0 30 1 1 32 0 1 0 9 0 65 0 2 0 55 0 0 59 14 0 68 18 0 0 55 0 0 19 1 13 11 0 0 0 2 0 99 29 17 35 89 18 18 52 107 4 5 30 25 33 109 15 2 35 60 89 14 3 31 0 34 0 162 52 60 293 81 44 46 64 77 6 9 98 66 162 54 46 2 287 202 159 7 25 88 9 53 1 37 3 5 3 22 27 3 15 39 1 0 18 7 2 48 3 0 31 8 56 12 0 5 0 4 0 63 6 3 3 72 5 8 77 27 1 0 20 6 7 42 3 0 22 13 24 23 1 8 0 3 0 200 5 3 3 557 3 3 10 168 0 0 8 5 2 78 1 0 17 9 63 17 0 7 0 7 0 10 6 60 112 37 30 74 13 1 0 24 67 55 308 19 17 1 40 134 178 2 23 18 3 0 1 6 0 0 0 4 0 0 0 1 0 0 0 0 0 7 0 0 0 0 0 13 0 0 0 0 0 17 4 2 2 97 2 1 7 40 0 0 8 3 22 7 1 0 1 14 11 1 0 5 0 4 0 73 6 3 55 129 12 4 7 107 1 10 106 9 2 102 5 0 6 24 19 11 5 7 0 57 0 77 12 2 2 125 2 2 6 36 1 1 4 5 5 49 15 0 6 10 13 11 0 6 0 34 0 61 10 26 234 108 10 149 28 62 3 22 14 13 14 90 6 1 4 43 204 8 5 18 0 25 0 19 20 19 50 9 115 13 21 27 1 49 45 69 156 110 36 0 117 44 106 237 29 94 2 12 2 38 2 1 1 60 1 0 8 20 0 0 25 2 1 32 24 0 18 13 16 21 0 3 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 12 0 0 0 0 0 97 14 18 32 199 13 11 24 77 1 15 24 23 26 121 9 0 21 59 63 19 5 18 0 31 0 118 18 25 10 92 14 10 119 82 3 16 30 31 19 73 24 2 8 70 176 24 3 35 0 9 0 108 25 20 16 125 14 11 554 114 2 6 42 33 12 206 9 1 36 64 119 22 2 53 0 28 0 9 7 19 12 10 5 34 8 14 0 4 48 15 59 3 31 0 58 54 104 0 1 7 0 6 1 7 0 0 0 106 0 0 0 14 0 0 0 0 0 9 0 0 0 0 0 0 0 0 0 2 0 124 2 1 3 70 1 1 63 73 0 0 7 2 27 46 1 0 5 9 6 1 0 5 0 3 0 2 0 1 0 2 0 0 1 1 0 0 0 0 0 0 2 0 0 0 3 0 0 0 0 1 0 32 14 11 8 29 12 5 17 26 4 2 9 8 6 92 8 1 5 29 31 2 1 19 0 5 0 0 0 0 0 4 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 2

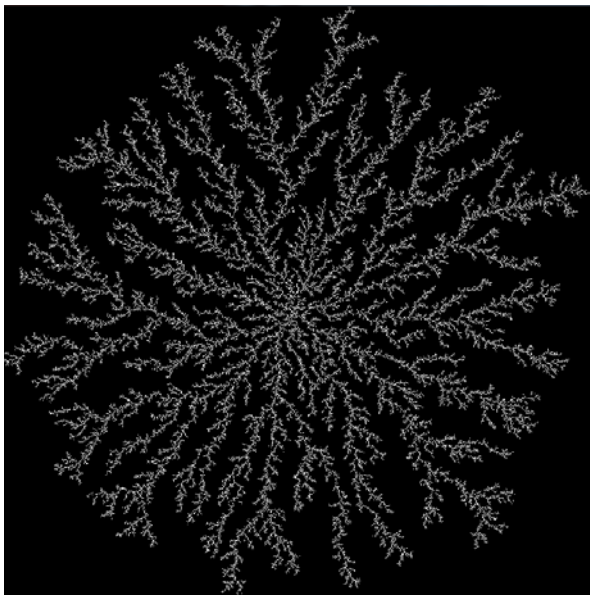
The cipher text below is from Edgar Alan Poe's short story *The Gold Bug* (1843). It is written using characters 5,3,),*,., etc. (we have replaced several nonASCII characters by *a,b,c,*.. so that they are not lost in printing). SPOILER ALERT: the solution to this cipher is given in Wikipedia and several other books.

```
gold_bug_ciphertext =:
'53aab305))6*;4826)4a.)4a);806*;48b8c60))85;1a(;:a*8b83(88)5*

b;46(;88*96*?;8)*a(;485);5*b2:*a(;4956*2(5*---4)8c8*;4069285
;)6b8)4aa;1(b

9;48081;8:8a1;48b85;4)485b528806*81(a9;48;(88;4(a?34;48)4a;161;:18
8;a?;'
```

Extending Diffusion-Limited Aggregation to Multiple Dimensions

Devon McCormick

"Diffusion-limited aggregation" is a simple process whereby an initial cluster of points, typically in two-dimensions, grows by having randomly-released particles stick to this central cluster. This gives us random patterns like the one to the right here. The feathery branches mimic some types of growth in the natural world.



We'll look at some J code I developed a while ago to build diffusion-limited aggregates. The code was written as an exercise in re-vamping some non-J-like code into a more natural form for the language. The changes also generalized the code to, among other things, extend it to build aggregates in an arbitrary number of dimensions.

First we'll look at the two-dimensional case for which the code was originally developed, then consider a problem with generalizing this to three dimensions: it's harder to examine a picture of the three-dimensional result. This difficulty motivates us to develop some browser-based code to help us visualize our three dimensional results.

Initializing a Point-set

The initialization code is a logical place to start, after we load the script and put its namespace in our environment:

```
   load '~Code/DiffLimAgg.ijs'

   coinsert 'dla'
NB. Path in the namespace for
easier typing...
```

The code is defined in the namespace "dla", so we insert this into our class object path after loading the script so we don't have to suffix all the names with "_dla_" in the following exposition. This means we can simply enter the name "init" of the initialization function instead of "init_dla_", as shown here, to see its definition.

```
   init

3 : 0

   D=: y            NB. D-
dimensions

   RNI=: D dimnbr 1 NB. Relative
neighbor indexes: 1 cell around
center.

   neigh2=: [ -.~ [: ~. [: ,/ RNI
+"1/~ ]   NB. Empty neighbors of
these

   PTS=: ,:y$0      NB. Start
point list w/Origin.

NB.EG  init 2       NB.
Initialize 2-dimensional space.

)
```

We see that running *init* with an argument of "2" initializes a two-dimensional list of points, starting at the origin, in the global *PTS*.

```
   init 2

0 0

   $PTS

1 2
```

Next we'll look at the cover function created during the development of the code to aggregate points to the list according to a random walk. This cover function returns a lot of timing information as this is an important consideration in the development of this code. The original version of this process used a fixed-size Boolean matrix to represent the aggregation. This had the advantage of simplicity and is easy to handle in non-J languages as well as in J.

However, this data structure introduces hard limits at the outset to how large our result can be. On the other hand, a fixed matrix helps the run-time efficiency as it also limits how far from the central aggregate we can release a random point and the distance from a possible "sticking" point is the primary determinant of how long it will take a particle to add the structure.

By changing the data structure to represent the points explicitly, we allow more than two dimensions but we also open the release space to be infinitely large. So, we need a way to limit the release space to be near the existing cluster. We've worked out a way to do this by building a perimeter around the existing cluster which we'll illustrate shortly. First, we'll continue the logical progression of our code exposition by looking at the next verb we'd run in the sequence.

```
   do1

3 : 'tm,(#PTS),(sv-
~#PTS),(tm%~sv-~#PTS),(usus
nn),(#nn)%~nn+/ . =>./nn [ tm=.
6!:2 ''nn=. growMany y''[sv=.
#PTS'
```

This rather long line buries the core work done by the *growMany* verb in a mess of statistics about the run-time performance. It's easier to show before we explain, so we'll do so now by giving an example of running this cover function.

```
   do1 25;20;5 2

0.0298599 5 4 133.959 7 25 22.25
6.02517 0.8
```

The arguments to *do1* are number of steps (25, here) per random walk (before abandoning an attempt), number of points (20) to release, then a pair of numbers denoting how far out from the cluster to we draw the release perimeter (5 cells) and how wide to make this perimeter (2 cells). Note that the number of steps should be greater than or equal to the perimeter distance squared (5^2); "squared" because it's a two-dimensional space and raising to this power reflects how many steps a random walk should take, on average, to travel that distance.

The results of this run are: number of seconds to complete (about 0.03), total number of resulting points (5), number of points added this time (4), number of points added per second (about 134); the five numbers following these first four give statistics on the number of random steps: the minimum (7), maximum (25), mean (22.25), standard deviation (6), and portion of walks reaching the maximum (80%).

Running this again with the same arguments results in about the same efficiency.

```
   do1 25;20;5 2
```

```
0.00197011 10 5 2537.93 5 25
22.55 5.2463 0.75
```

The time spent is much shorter but the little importance should be attached to differences between such small numbers. The more important numbers in these initial accretions are the statistics of the number of random steps which are essentially the same as for the earlier invocation. Before we look at the resulting set of points, we'll run this one more time, doubling the number of points we try (to 40) for a slight increase in efficiency.

```
   do1 25;40;5 2
```

```
0.00315529 22 12 3803.14 2 25
20.975 7.48498 0.7
```

We can use the standard *viewmat* verb to visualize the set of points – shown here transposed for efficiency of display - we have so far:

```
   |:PTS
```

```
0 _1 _1 _2 _3 _2  0 _4 _5 _5 _6
_1 _6 1 0 _6 1 _7 0 1 _8  0
```

```
0 _1 _2 _2 _1 _3 _3 _2 _1 _3  0
1 _1 0 2 _4 3 _2 4 4 _2 _4
```

```
   viewmat 1 bordFill PTS
```

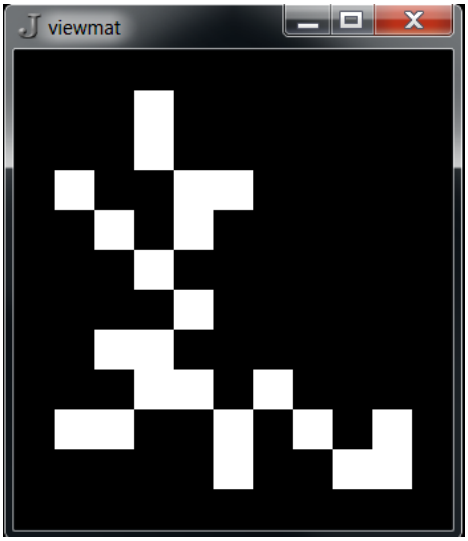
We view the result of the utility verb *bordFill* which converts the explicit list of points into a simple Boolean matrix for display purposes.

```
   bordFill
4 : '(1)(x adjPts
y)}0$~D$(>:+:x)+(>./y)-<./y'
```

The left argument is the number of cells to put as a border around the cluster. The *adjPts* utility shifts the point values to be all non-negative by subtracting the smallest value in each column from each of the rows.

```
   adjPts
[: <"1 ] -"1 [ -~ [: <./ ]
```

It also encloses the values in each row so we can use them as indexes to set the cell values in the simple matrix representation of the cluster.



Setting the "Release" Perimeter

This small cluster of points we've generated will better help us illustrate how we determine the cluster perimeter into which we release potential new points on their random walks. As mentioned before, the ***do1*** verb is an informative wrapper for the core ***growMany*** verb, shown here.

```
   growMany
```

```
3 : 0
```

```
   'maxwalk numpts pp'=. y
```

```
   pts=. numpts (]{~ ([<.[:#])
?[:#]) calcPerim pp
```

```
   ctr=. _1 [ rr=. i.0
```

```
   while. maxwalk>ctr=. >:ctr *.
(0<#pts)
```

```
   do. wcc=. PTS chkCollision
pts=. walk pts
```

```
       if. 1 e. wcc do. addPt
wcc#pts
```

```
           pts=. pts#~-.wcc [
rr=. rr,ctr$~+/wcc
```

```
      end.

   end.

   rr,maxwalk$~#pts

)
```

The three arguments to ***growMany*** are exactly those to ***do1***; they are broken into named groups on the first line of this verb. The second line, where the release perimeter is calculated, is again best illustrated by first showing the sub-function ***calcPerim*** in action by itself. Its definition is thus:

```
   calcPerim

3 : 0

   NP=: PTS neigh2^:(0{y)]PTS

   edges=. ~.PTS-.~NP -.~
,/NP+"1/RNI

   edges=. ~.PTS
neigh2^:(1{y)]edges

NB.EG PPTS_dla_=: calcPerim 5 2

)
```

We see that it directly references the global *PTS* noun in the first line, along with the first of its two inputs (0{y), as inputs to the ***neigh2*** verb. We have already seen this verb as it was defined in the *init* code because it incorporates a global that depends on the dimensionality of our points. It was necessary to define this in the ***init*** function because it is an explicit verb, meaning, among other things, it resolves the global at the point of definition.

Here we see the value of ***neigh2***, as well as its definition, along with the value of the global *RNI* that it incorporates.

```
   neigh2=: [ -.~ [: ~. [: ,/ RNI
+"1/~ ]

   neigh2
[ -.~ [: ~. [: ,/ (8 2$_1 _1 _1 0
_1 1 0 _1 0 1 1 _1 1 0 1 1) +"1/~
]

   |:RNI

_1 _1 _1  0 0  1 1 1

_1  0  1 _1 1 _1 0 1
```

This noun RNI, (which stands for *Relative Neighbor Indexes*) comprises the relative indexes of all the neighbors of a point in two-space, illustrated graphically here:

```
   3 3$1 2 3 4 0 5 6 7
8{(<''),<"1 RNI

+-----+----+----+

|_1 _1|_1 0|_1 1|

+-----+----+----+

|0 _1 |    |0 1 |

+-----+----+----+

|1 _1 |1 0 |1 1 |

+-----+----+----+
```

We can run the first line of ***calcPerim*** in isolation to see what it does.

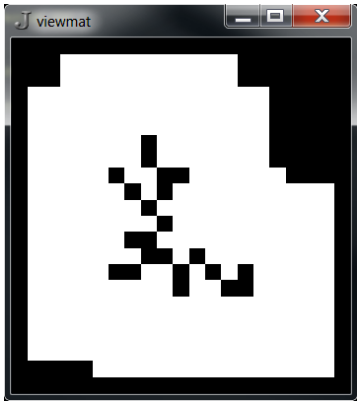
```
   y=. 5 2
NB. What y would have been…

   $NP=: PTS neigh2^:(0{y)]PTS
NB. Array of 2D points

315 2

   viewmat 1 bordFill NP
NB. See what it looks like.
```

We can see that the current set of points matches the empty cells in the middle: *NP* is the set of neighboring points around this set.

Running the next line of *calcPerim* gives us this:

```
   $edges=. ~.PTS-.~NP -.~
,/NP+"1/RNI
82 2
```

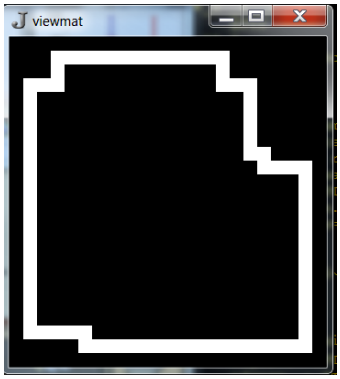
```
   viewmat   1   bordFill   edges
```



We see that this gives us only the outermost edge of this surrounding region.

The last line of ***calcPerim*** is nearly the same as the first one but it is applied to these edges and uses the second element of *y* as the argument to the power conjunction.

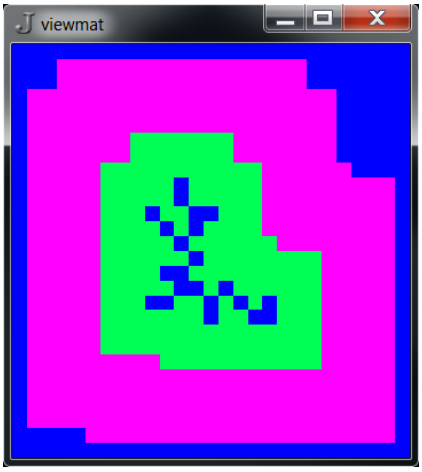
```
$e2=. ~.PTS neigh2^:(1{y)]edges
410 2
   viewmat 1 bordFill edges
```




```
   $1 bordFill edges,e2,NP
NB. Determine which set of points
28 27
   $1 bordFill edges,e2
NB. encompasses the largest
bounding box
28 27
   $1 bordFill edges
NB. and sum the different sets to
get all
24 23
   $1 bordFill e2
NB. in one picture with
distinctive colors.
28 27
   viewmat (1 bordFill e2)+(1
bordFill edges,e2)+1 bordFill
edges,e2,NP
```

So we can see the point set outlined in the center in blue and the result from ***calcPerim*** as the magenta border: this is the perimeter set of points from which we'll choose to release our random walkers. This helps us understand that the second line of ***growMany*** selects ***numpts*** points from this perimeter.

```
pts=. numpts (]{~ ([<.[:#])
?[:#]) calcPerim pp
```



Looking at the next few lines after this,

```
   ctr=. _1 [ rr=. i.0
   while. maxwalk>ctr=. >:ctr *.
(0<#pts)
   do. wcc=. PTS chkCollision
pts=. walk pts
```

we see a counter *ctr* and result set ***rr*** being initialized, the start of a *while* loop, and the check for a collision of any of the points after they've been walked one random step. The last two lines within the while loop take care of adding any collided points to the global *PTS* and removing them from the set being randomly walked. The result set *rr* keeps track of how many steps were required to add any new points by repeating the current value of the counter by the number of newly-added points at any step.

```
       if. 1 e. wcc do. addPt
wcc#pts
           pts=. pts#~-.wcc [
rr=. rr,ctr$~+/wcc
```

The final line of *growMany* returns the resulting new points to be added to the set as well as a count of how many released points reached the random walk limit without being added to the set.

```
   rr,maxwalk$~#pts
```

The two sub-functions referenced above are simply this:

```
   addPt
3 : 'PTS=: PTS,y'
   chkCollision
4 : '+./"1 (y+"1/RNI) e. x'"_
```

Looking again at the wrapper *do1* for *growMany*, we can now better appreciate how it works:

```
  do1
3 : 'tm,(#PTS),(sv-
~#PTS),(tm%~sv-~#PTS),(usus
nn),(#nn)%~nn+/ . =>./nn [ tm=.
6!:2 ''nn=. growMany y''[sv=.
#PTS'
```

We see that we start, going from right to left, by saving the number of starting points, then we time how long *growMany* takes to give us the result *nn* which is the vector of random walk lengths. The major part of the line uses these results to produce statistics to give us an idea of how efficiently we're adding points to the cluster.

A Problem with Three Dimensions

Notice how useful we found the *viewmat* verb in the preceding exposition. This was equally true during the development of this code. Being able to look at a picture of the points being generated helped guide development of these algorithms and provided assurance that we were doing sensible things. However, when run this code to produce a three-dimensional DLA, as shown here, we face the problem of visualizing these new results.

```
   init 3
0 0 0
   do1 125;20;5 2
0.0243368 3 2 82.18 17 125 114.25
33.0882 0.9
   do1 125;20;5 2
0.0298845 5 2 66.9244 60 125
120.85 14.8759 0.9
…
   do1 150;100;5 2
0.14899 120 30 201.355 4 150
117.93 51.3275 0.7
   $PTS
120 3
```

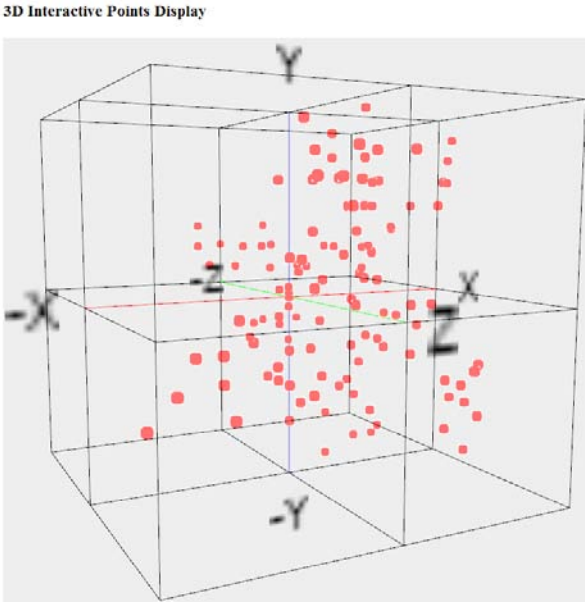
So, it looks like we have 120 three-dimensional points. What do they look like?

Searching the web for a tool with which to visualize this point set yielded little satisfaction. Ideally, we'd like not only to be able to see the points but to move the display in real-time to look at the DLA from different angles. There is some functionality of this type in the R environment, but it appeared to be somewhat clumsy to use and it requires moving the data into that environment as well.

I did find a set of three-dimensional visualization scripts called "Dex" – a "data explorer" – that is based on the popular Javascript graphing package "d3.js". It didn't do exactly what I wanted but appeared to be close. So, in just a couple of hours, I was able to put together something that worked like this for our sample set of points.

```
   buildView
PTS;(0$~#PTS);'sample3D120pts';'S
ample of 120 three-dimensional
points'
```

This shows our points in a browser in a form easily rotated by moving the cursor around it, looking something like this:



The code is currently much cruder than I'd like, but it works fine for this case. In the future, I'd like to better integrate it into the JHS environment and produce the HTML more neatly but it's fine for now.

The J routines look like this:

```
   buildView
3 : 0
   'pts clrs flpfx title'=. 4{.y
   if. 0=#title do. title=. flpfx
end.

   tmplt=. fread
'ScatterPlot3D.tmplt'
   tmplt=. (tmplt rplc
'{title}';title) rplc
'{thisScat3D}';flpfx,'.js'
   (tmplt rplc
'{points}';cvt2d3Array pts)
fwrite flpfx,'.htm'

   set0=. (<cSets) rplc &.>
(<'{i.#pts}');&.>":&.>i.#pts
   set1=. ;set0 rplc&.>
(<'{colorIx}');&.>":&.>clrs
   mystmplt=. fread
'ScatterPlot3D1_files/myScatterPl
ot3DJS.tmplt'
   subfns=. CR-.~mystmplt rplc
'{colorsets}';set1
   subfns fwrite
'ScatterPlot3D1_files/',flpfx,'.j
s'
)
   cvt2d3Array
3 : '}:_1|.;}.&.>;&.><"1
(<''],['']),.~'','',&.>j2n&.>y'
   rplc
stringreplace
   j2n
'S'&J2StdCvtNums
   J2StdCvtNums
3 : 0
NB.* J2StdCvtNums: convert char
rep of num from J to "Standard",
or
NB. vice-versa if left arg is 'J'
or '2J'; optional 2x2 left
argument allows
NB. 2 arbitrary conversions: col
0 is "from", col 1 is "to" char.
NB. Monadic case changes Standard
numeric representation to J
representation.
   (2 2$'-_Ee') J2StdCvtNums y
:
   if. 'S'=x do. if. ' '={.,y do.
y return. end. end.
   if. 0=#y do. '' return. end.
   pw16=. 0j16
NB. Precision width: 16 digits>.
   diffChars=. 2 2$'-_Ee'
NB. Convert '-'->'_' & 'E'->'e'
   toStd=. -.'J'-:''$'2'-.~,x
NB. Flag conversion J->Standard
   if. 2 2-:$x do. diffChars=x
NB. if explicit conversion.
   elseif. toStd do. diffChars=.
|."1 diffChars end.   NB. Convert
other way
   if. 0=1{.0$y do.
NB. Numeric to character
```

```
       fmts=.
(8=>(3!:0)&.>y){0,pw16   NB.
Full-precision floats only
       y=. fmts":y
NB. If this is too slow, go back
   end.

   y=. y-.'+'
NB. EG 1.23e+11 is ill-formed &
the
   wh=. y=0{0{diffChars
NB.  '+' is unnecessary.
   cn=. (wh#1{0{diffChars) (wh#i.
$y)}y    NB. Translate chars that
need it
   wh=. y=0{1{diffChars
NB.  but leave others alone.
   cn=. (wh#1{1{diffChars) (wh#i.
$cn)}cn
   if. -.toStd do.
NB. Special handling -> J nums
       if. '%'e. cn do.
NB. Convert nn% -> 0.nn
          cn=. pw16":0.01*".cn-.
'%'
       end.
       cn=. cn-.','
NB. No ',' in J numbers
   end.
   cn
NB.EG 'S' J2StdCvtNums _3.14
6.02e_23   NB. Convert J numbers
to std rep
```

The main *buildView* routine simply customizes a pair of HTML templates by inserting the points, converted into a form suitable for the Javascript routines, and some title information into the templates. The main template, named *ScatterPlot3D.tmplt*, is assumed to be in the current directory. This is where the data points and the display title are inserted.

The second argument to *buildView* is a vector of colors corresponding to the matrix of points to be shown. These colors, currently 17 of them, are defined in the other template, called *myScatterPlot3DJS.tmplt* and assumed to be in the subdirectory *ScatterPlot3D1_files*. The javascript code to set the colors is more elaborate than desired and is generated by *buildView* to be inserted in this latter template. This second template is named to match the main HTML file being created and written to the sub-directory. The name of these files, rather the file name prefix, is designated by the third element of the argument to *buildView*.

The fourth and final argument to *buildView* is text to title the resulting display.

These two templates are about 101, and 341 lines long, respectively.

Journal of J

An interdisciplinary journal on J programming language and applications in science and technology.

www.journalofj.com

THE J GUIDEBOOK for Programming, Numerics and Graphics.

A international and interdisciplinary challenge !

This is a new and stimulating project of JoJ team and J community. The challenge is to create a companion (or book, or algorithm repository) to the work *The art of computer programming*. Similarly, in this project, we intend to create a work similar to *The Mathematica Guidebook* of Michael Trott. In this context ,the work of Michael is a source of ideas to create verbs, programs and scripts in J.

For more information send a e-mail to info@journalofj.com.

****

****

****

****

*

MPM Press **AN OPEN JOURNAL**

ISSN: 2174-9280

Vol.2, No.1 July 2013