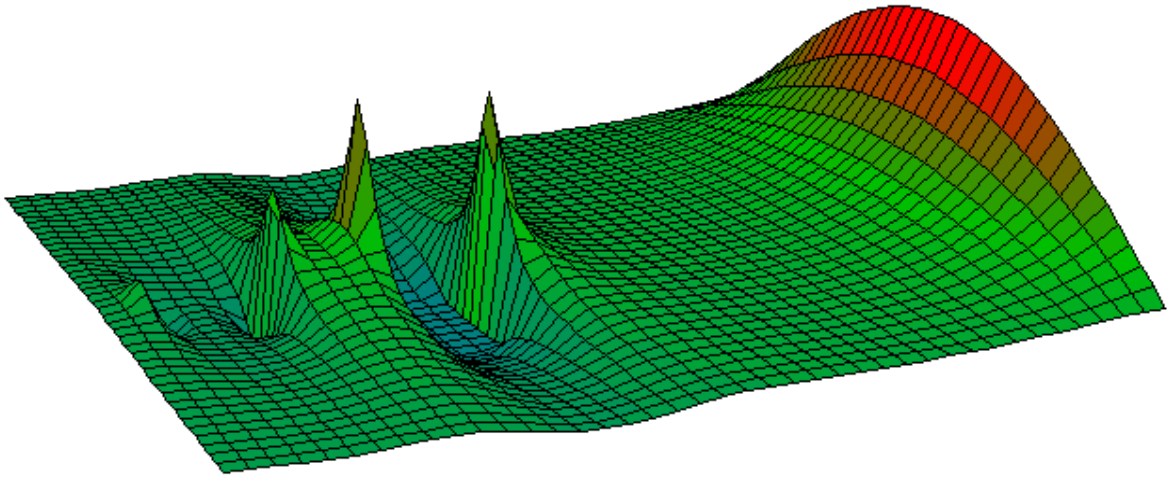
Journal of J

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

New web site www.journalofj.com



Please Circulate

MPM press

An open access Journal

ISSN: 2174-9280

Vol.3, No.1 May 2014

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

2014 J Conference

I'm really excited about the annual J conference in July 2014.

For the last three years we have been wanting to extend the knowledge of J. This year is our turn to do all we can to spread the knowledge of J.

This conference is very important for all of us because we have the chance of publishing J conference proceedings.

So, we want to encourage J-ters to send their contributions (letters, papers, thoughts and more) to make this project grow.

Thanks in advance. We look forward to hearing from you.

Mikel Paternain

JoJ team

In this issue

A Rule-based Sodoku Solver in J

Zhe Hu

huzhe@sigenics.com

Introduction

There are many implementations of computer Sudoku solvers, most of which are based on brute force trial-and-error or backtracking. One can browse through these at Rosetta code website (http://rosettacode.org/wiki/Sudoku).

The trick of a faster solver is a better way to "guess" or "search", as pointed out by Peter Norvig (http://norvig.com/sudoku.html). That is to try to first fill an empty cell that has the least number of candidates. For example, pick an empty cell that can be filled with numbers 6 or 7, trying either of these possibilities essentially halves the search space, whereas another cell with possibilities of 1 to 9 may require a lot more trial-and-error.

To reduce the size of candidates, the solver program repeatedly applies some "forced move" rules (Peter Norvig called it "constraint propagation") after the "guess" step.

Here are the two strongest "forced move" rules:

1. if a cell (also called square) can only be assigned one number, then assign it that number
2. if a number can only belong to one empty cell in a row, column or 3x3 box (also called region or unit), then assign that number to that one empty cell

Interestingly, Roger Hui's J solver (http://www.jsoftware.com/jwiki/Essays/Sudoku) has all these nice properties, which makes it a very fast solver.

```
sudoku =: guess @: (ok # ]) @: assign ^:_ @ ,
```

As we can see, the three steps of Hui's J solver :

1. apply 2 "forced move" rules (ac and ar in verb assign) to fill the grid
2. select the resulting grids that do not have contradictions (verb ok)
3. if the grid is not filled, then guess based on least-number-candidate and go back to step 1

As mentioned in David Eppstein's paper (http://arxiv.org/abs/1202.5074), there are many other computer problem-solving techniques being applied to the Sudoku puzzle. These are not the focus of this article.

We are interested in constructing a computer Sudoku solver that mimics a human puzzle solver. As any Sudoku player knows that a fun Sudoku puzzle for humans must be solvable by a sequence of logic deductions, not mindless trial-and-error or backtracking. It's simply because humans are good at

recognizing visual patterns and using local information for logic reasoning. Our memory system isn't built with a deep stack for backtracking.

To borrow Eppstein's term, we'd like to code a rule-based Sudoku solver that simulates human solvers without backtracking. In this article, we are going to implement one extra rule, among many of which are detailed in Eppstein's Python code (http://www.ics.uci.edu/~eppstein/PADS/) in J.

Obviously, not every possible Sudoku puzzle can be solved using this rule-based approach. But that's exactly the purpose of such a program. It differentiates between fun-for-human and fun-for-computer Sudoku puzzles.

Read and Display the Puzzle

The following J verb reads a Sudoku puzzle from a zero or dot separated string of numbers. The key verb here is ;. which splits a string via certain "cutter" template. The resulting p1 and p2 are 81-element list of numbers. Empty cells are filled with number 0, the rest are filled with numbers 1 to 9. They can be displayed in a grid with verb see. (Verbs not shown explicitly in this article come from Hui's Sudoku solver. Please load his code first before trying the following code.)

```
   readSudoku =: (1 #~ #) (0&". ;. 1) ]

   p1 =: readSudoku
'..............3.85..1.2.......5.7.....4...1...9.......5......73..2.1..
......4...9'

   p2 =: readSudoku
'00000001204005000000000900007060040000010000000000050000087500601000 3
00200000000'

   see p1,:p2
+-------------+-------------+
|+---+---+---+|+---+---+---+|
||...|...|...|||...|...|.12||
||...|..3|.85|||.4.|.5.|...||
||..1|.2.|...|||...|..9|...||
|+---+---+---+|+---+---+---+|
||...|5.7|...|||.7.|6..|4..||
||..4|...|1..|||...|1..|...||
||.9.|...|...|||...|...|.5.||
|+---+---+---+|+---+---+---+|
||5..|...|.73|||...|.87|5..||
||..2|.1.|...|||6.1|...|3..||
||...|.4.|..9|||2..|...|...||
|+---+---+---+|+---+---+---+|
+-------------+-------------+
```

Additional Sudoku puzzle strings can be found at
https://github.com/attractivechaos/plb/blob/master/sudoku/sudoku.txt, or
http://school.maths.uwa.edu.au/~gordon/sudoku17

Existing Rules for Solving the Puzzle

Two "forced move" rules mentioned earlier are coded as ac and ar in Hui's J solver. Both of these verbs examine the 81x9 candidate list, which is generated by verb free. It produces a list of candidates (number 1 to 9) for each cell (81 total) in a puzzle.

```
   $ free p2
81 9
   8 { free p2
0 0 0 0 0 0 0 0 0
   0 { free p2
0 0 1 0 1 0 1 1 1
```

For example, in the puzzle grid `p2`, the top-right cell `(index 8{)` is already filled with number 2, so there is no possible candidate for it. On the other hand, the top-left cell is empty, and it can't take 1, 2, 4, 6, because these numbers already share the row, column or box with it. So only numbers 3, 5, 7, 8 and 9 are possible candidates, indicated by '1's in the 9-element list.

The `ac` rule is easy to understand. For example,

```
   ac 0 0 0 0 1 0 0 0 0
5
   ac 0 0 0 0 1 0 0 0 1
0
```

In the first case, the 9-element candidate list has only a single '1'. Verb `ac` returns its corresponding number 5. In the second case, the 9-element list has 2 '1's. Verb `ac` returns 0, meaning indeterminate.

The `ar` rule examines the candidate list for all the cells in a region. For example, let's pick two regions out of grid `p2`, the top-right `(index 2{)` and the top-left `(index 0{)` boxes. Their 9x9 candidate lists, rows representing cells, columns representing numbers 1 to 9:

```
   0{ R{ free p2
0 0 1 0 1 0 1 1 1
0 0 1 0 1 1 0 1 1
0 0 1 0 1 1 1 1 1
1 0 1 0 0 0 1 1 1
0 0 0 0 0 0 0 0 0
0 1 1 0 0 1 1 1 1
1 0 1 0 1 0 1 1 0
1 1 1 0 1 1 0 1 0
0 1 1 0 1 1 1 1 0
   2{ R{ free p2
0 0 0 0 0 1 1 1 1
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 1 1 1
0 0 1 0 0 1 1 1 1
0 0 1 0 0 1 1 1 1
0 0 0 0 0 1 1 1 0
```

```
0 0 1 1 0 1 1 1 0
0 0 1 1 1 1 1 1 0
```

Next we sum `+/"2`, resulting a 9-element candidate list for all cells in the region. An element 1 indicates that a number can only belong to one of the cells. In the top-left box of `p2`, no such case is found. But in the top-right box, number 5 stands out as the only possible candidate for an empty cell.

```
   1 = +/ (0{ R{ free p2)
0 0 0 0 0 0 0 0 0
   1 = +/ (2{ R{ free p2)
0 0 0 0 1 0 0 0 0
```

Can you see which empty cell number 5 belongs to? (Hint: look at the last row in `2{R{free p2)`

So far we've dissected Hui's J solver, especially experimenting with his two forced move rules: verbs ac and ar. Incidentally, repeatedly applying these two rules can solve both puzzles `p1` and `p2`.

```
   see p1,: assign p1
+-------------+-------------+
|+---+---+---+|+---+---+---+|
||...|...|...|||987|654|321||
||...|..3|.85|||246|173|985||
||..1|.2.|...|||351|928|746||
|+---+---+---+|+---+---+---+|
||...|5.7|...|||128|537|694||
||..4|...|1..|||634|892|157||
||.9.|...|...|||795|461|832||
|+---+---+---+|+---+---+---+|
||5..|...|.73|||519|286|473||
||..2|.1.|...|||472|319|568||
||...|.4.|..9|||863|745|219||
|+---+---+---+|+---+---+---+|
+-------------+-------------+
   see p2,: assign p2
+-------------+-------------+
|+---+---+---+|+---+---+---+|
||...|...|.12|||598|463|712||
||.4.|.5.|...|||742|851|639||
||...|..9|...|||316|729|845||
|+---+---+---+|+---+---+---+|
||.7.|6..|4..|||175|632|498||
||...|1..|...|||869|145|273||
||...|...|.5.|||423|978|156||
|+---+---+---+|+---+---+---+|
||...|.87|5..|||934|287|561||
||6.1|...|3..|||681|594|327||
||2..|...|...|||257|316|984||
|+---+---+---+|+---+---+---+|
```

But there are many puzzles that cannot be solved using only these 2 rules. Here is an example

```
   p3 =: readSudoku
'........3..1..56...9..4..7......9.5.7.......8.5.4.2....8..2..9...35..1
..6........'
   see p3,:assign p3
+-------------+-------------+
|+---+---+---+|+---+---+---+|
||...|...|..3|||...|...|..3||
||..1|..5|6..|||..1|..5|6..||
||.9.|.4.|.7.|||.9.|.4.|.7.||
|+---+---+---+|+---+---+---+|
||...|..9|.5.|||...|..9|.5.||
||7..|...|..8|||7..|.5.|..8||
||.5.|4.2|...|||.5.|4.2|...||
|+---+---+---+|+---+---+---+|
||.8.|.2.|.9.|||.8.|.2.|.9.||
||..3|5..|1..|||..3|5..|1..||
||6..|...|...|||6..|...|...||
|+---+---+---+|+---+---+---+|
+-------------+-------------+
```

Only a number 5 is filled in the middle box. `p3` is certainly a valid Sudoku puzzle, which is solvable with Hui's solver. But that involves verb guess at work.

The question is whether we can solve the puzzle by using only logic deduction rules without going into guess.

```
   see p3, sudoku p3
+-------------+-------------+
|+---+---+---+|+---+---+---+|
||...|...|..3|||562|987|413||
||..1|..5|6..|||471|235|689||
||.9.|.4.|.7.|||398|146|275||
|+---+---+---+|+---+---+---+|
||...|..9|.5.|||236|819|754||
||7..|...|..8|||714|653|928||
||.5.|4.2|...|||859|472|361||
|+---+---+---+|+---+---+---+|
||.8.|.2.|.9.|||187|324|596||
||..3|5..|1..|||923|568|147||
||6..|...|...|||645|791|832||
|+---+---+---+|+---+---+---+|
+-------------+-------------+
```

Extra Rules

For beginning Sudoku solvers, there are other easy to practice logic deduction rules. One of them is called "pair rule".

pair rule

If in a region, be it a row, column or box, 2 numbers only share a pair of cells, then these 2 numbers belong exclusively to that pair. It means that we can remove all the other numbers from the candidate lists of that pair of cells.

One can certainly generalize this way of thinking to 3 cells sharing 3 numbers exclusively, so on and so forth. But it becomes more difficult for humans to keep track as the group becomes larger.

This rule can be used to trim down the candidate list. Let's implement it in J.

```
 pair_in_region =: 4 : 0 NB. input y: candidate list, x: region index
  clist =. > y
  rg =. > x

  t =. rg{ clist  NB. 9x9: cell x number 1--9
  m =. 2 = +/"2 t   NB. numbers belong to 2 cells
  d =. I. m    NB. numbers
  if. 2 <: #d do. NB. a least 1 pair
    c =. m # |: t NB. pick cells
    j =.  ((2 = +/"1) # ]) = c NB. find true pairs
    if. 0 < #j do.
        v =. 3 : '2 9$(1 1) (;/y)}9$0'
        nv =. ,/ v"1 j # d NB. new vector cleaning out other numbers
        i =. ,(({."1 j#d){ |: t) # rg
        < nv i } clist NB. return updated clist
        return.
    end.
  end.
  y  NB. nothing to change

)
  pair =: 3 : 0 NB. input "free p3" 81x9: cell x 9 candidates
    NB. J right fold
    >pair_in_region / (;/"_ R),<y
    NB. return updated candidate list
  )
```

For example, test is a 9x9 candidate list for a region. Verb `pair_in_region` zeros out other number candidates in the second row.

```
    [test =. (0 1 0 0 1 1 0 0 1,:0 1 0 0 0 0 0 0 1) (<<1 2)} 9 9$0
0 0 0 0 0 0 0 0 0
0 1 0 0 1 1 0 0 1
0 1 0 0 0 0 0 0 1
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 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
```

```
    (i.9) pair_in_region test
+-----------------+
|0 0 0 0 0 0 0 0 0|
|0 1 0 0 0 0 0 0 1|
|0 1 0 0 0 0 0 0 1|
|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 0 0 0 0 0|
|0 0 0 0 0 0 0 0 0|
|0 0 0 0 0 0 0 0 0|
+-----------------+
```

align rule and others

Here is another rule that is also quite intuitive. It states that if the cells in a box which can contain a number, all lies in a row or column, then we can "zero" the candidate lists of that number in other cells, which are in the same row or column. Similarly, if the cells in a row (or column) which can contain a number, all lies in a box, then we can "zero" the candidate lists of that number in other cells, which are in the same box.

It is also a rule to trim down the candidate list. Its implementation is left to the reader for exercise.

More advanced player may apply chain rules, such as bi-value and bi-location, as well as semi-trial-and-error methods, such as nishio. These can also be added to a rule-based Sudoku solver. (see Eppstein's Python code for details)

Solvable Puzzles

Now we can add our pair rule to Hui's existing `ac` and `ar` rules. We simply redefines verb `assign` to be

```
assign2 =: (+ (ac >. ar)@pair@free)^:_"1
```

We load the puzzle strings from http://school.maths.uwa.edu.au/~gordon/sudoku17. It contains 49151 pieces of puzzle.

```
   puzzles =:  readSudoku (;._2) 1!:1 < jpath '~temp/sudoku17.txt'
```

```
   $puzzles
49151 81
   +/ (-.@ (0 e. ])@ assign) puzzles
21905
   +/ (-.@ (0 e. ])@ assign2) puzzles
30449
```

Rules Applied	Percent of Puzzles Solved
ac + ar	45%
ac + ar + pair	62%

Assuming these puzzles are some random samples, as the above table shown, using only simple logic deduction rules, one can already solve more than half the number of Sudoku puzzles. And the pair rule certainly helps the human solver.

Discussion

The logic puzzles such as Sudoku bear a lot of similarities to real world engineering problems, such as analyzing an electrical circuit.

In circuit analysis, the local rules are Ohm's law and certain voltage-current relationship for a specific device. Using these local rules, one can deduce from node voltage to the branch current, or vice versa. However, a real circuit may have many interconnected nodes, so the local rules themselves are not sufficient. One has to use nodal or mesh analysis (KCL or KVL) based on the complete circuit topology (i.e. global information). It's difficult to reach an analytical solution once the circuit topology gets complicated, if one has to solve matrix inverse symbolically.

On the other hand, it's easy for a computer circuit simulator (EAD program) to fill the matix and solve the linear equations for a numerical answer.

Since the computer already does a fast enough job in both cases,

1. Why bother with a human-like Sudoku solver?
2. Why bother with analyzing the circuit manually?

The first question is easy to answer, because human likes to have fun with puzzles too. Now let's try to answer the second question. There have been successful efforts called D-OA (design oriented analysis) by late Dr. David Middlebrook. He tried to reduce/simplify a complicated circuit through a sequence of logic deductions, so that eventually local rules can be applied to an equivalent circuit that has analytical solutions (his term: avoiding paralysis by algebra). Because designing a circuit is the inverse problem of analyzing it. A good circuit designer usually has a good "intuitive" understanding of the circuit. That doesn't come from running a simulator with different parameters many times (trial-and-error), rather that understanding comes from the propagation of local information, just like a human Sudoku solver. (additional discussion, see Dr. Gerald Jay Sussman's lecture *minute 25* at http://www.infoq.com/presentations/We-Really-Dont-Know-How-To-Compute) So it's important to be able to analyze a circuit manually, even approximately.

Interestingly a new school of thinking (NKS) believes that the human logic reasoning may not cover the whole complexity of the computational universe. A well directed search by a computer may find a

circuit, for example, that fulfills a specification much better than what a human circuit designer can come up with. Such a program is certainly out there to be discovered.

Conclusion: By augmenting the "forced move" rules of Roger Hui's J solver, we made a rule-based Sudoku solver in J. It doesn't have all the possible rules used by a human player, especially advanced techniques used by experienced players. But it can already play beginner level Sudoku puzzles using the same sequence of logic deductions as human players do. So it can be used to differentiate between fun-for-human and fun-for-computer Sudoku puzzles.

Visual Dynamics of the Complex 3*x*+1 Function Revisited

Cliff Reiter

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

The Collatz or 3*x*+1 function is often formulated by $t(x)=\begin{cases}\dfrac{x}{2} & \text{if } x \text{ is even}\\ \dfrac{3x+1}{2} & \text{if } x \text{ is odd}\end{cases}$ where x is a positive integer. For example, if $t(x)$ is iterated on 17, one obtains the sequence: 17, 26, 13, 20, 10, 5, 8, 4, 2, 1, 2, 1. It is conjectured, but not known, whether that function always reaches the cycle (1,2) when it is iterated upon a positive integer. The function may be extended to the complex domain by considering $T(x)=\dfrac{3^{\mathrm{mod}_2(x)}x+\mathrm{mod}_2(x)}{2}$ where $\mathrm{mod}_2(x)$ is a suitable oscillating (complex) function such as $\mathrm{mod}_2(x)=\sin^2\left(\dfrac{\pi x}{2}\right)$ or $\mathrm{mod}_2(x)=\dfrac{1}{2}\left(1-e^{i\pi x}\right)$. Images of the complex dynamics of those functions was explored in [1,2]. A recent Jprogramming forum post Bo Jacoby [3] noted the function $t(x)$ can also be formulated via $r(x)=\dfrac{1}{2}\left(x+(1+2x)\mathrm{mod}_2(x)\right)$ which likewise generalizes to the complex domain. We have translated his function `collatz=:-:&(+2&|*>:&+:)` to common math notation with some liberties. Many ideas regarding the **Collatz** function were discussed in that thread, but [3] resonated with me. This note is motivated by the idea of revisiting the visual dynamics of the $3x+1$ function using the formulation given by $r(x)$. In reviewing [1,2] we discover that the

function $C(x)$ discussed there is essentially the same formulation as $r(x)$ but only $\mathrm{mod}_2(x) = \sin^2\left(\frac{\pi x}{2}\right)$ was considered. Thus, this note revisits the dynamics of $r(x) = \frac{1}{2}\left(x + (1+2x)\,\mathrm{mod}_2(x)\right)$ in the unexplored complex exponential case where $\mathrm{mod}_2(x) = \frac{1}{2}\left(1 - e^{i\pi x}\right)$.

1. Escape Time Images

We use J6.02 in order to be able to access some addons available there. It is straight forward to implement our function for the modulus.

```
   mod2=: -:@-.@^@(0j1p1&*)
   mod2 0 1 2 3 4
0 1j_6.12323e_17 0j1.22465e_16 1j_1.83697e_16 0j2.44929e_16
```

However we see there is distracting round-off error, so we will use a utility `cclean` to clean the real and imaginary parts.

```
   cclean=:(**|)&.+.
   cclean mod2 0 1 2 3 4
0 1 0 1 0
```

Likewise, it is straightforward to implement the $r(x)$ version of the 3*x*+1 function.

```
   r=: [: -: ] + 1 2&p. * mod2
   cclean r 1 2 3 4 5 6 7
2 1 5 2 8 3 11
   cclean@r^:(i.12)17
17 26 13 20 10 5 8 4 2 1 2 1
```

We turn to utilities defined in Section 7.4 of [4] to create an array of complex starting points and a conjunction that allows us to compute escape for our starting points. The verb `zl_clur` takes the

number of horizontal steps as its left argument and the center left and upper right points desired as its right argument.

```
  zl_clur=: 4 : 0
w=.-~/9 o.y
h=.-~/11 o.y
xs=.({.y)+w*(i.%<:)1+x
ys=.h*(i:%j.)<.0.5+x*h%w
ys +/ xs
)
   6 zl_clur _2 2j1
   _2j1    _1.33333j1    _0.666667j1    0j1    0.666667j1    1.33333j1    2j1
 _2j0.5  _1.33333j0.5  _0.666667j0.5  0j0.5  0.666667j0.5  1.33333j0.5  2j0.5
     _2      _1.33333      _0.666667      0      0.666667      1.33333      2
_2j_0.5 _1.33333j_0.5 _0.666667j_0.5 0j_0.5 0.666667j_0.5 1.33333j_0.5 2j_0.5
  _2j_1   _1.33333j_1   _0.666667j_1   0j_1   0.666667j_1   1.33333j_1   2j_1
```

We bolded the center left and upper right entries in the output for emphasis. When we iterate `r` we may get errors.

```
   r^:(1 2) 1j_1
18.6055j_12.5703 _4.00933e17j_1.56731e18
   r^:(3) 1j_1
|limit error: mod2
|       r^:(3)1j_1
|[-2]
```

Thus, we will use r with adverse, `r :: _:` as the function argument to the complex escape time conjunction `escapetc` which is also from Section 7.4 of [4].

```
  escapetc=: 2 : 0
#@((,u@{:)^:(<&({:n)@# *. (<&({.n)@|@{:))^:_) f."0
)

   r :: _: escapetc (1e4 253) 6 zl_clur _2 2j1
253 253 253 253 253 253 253
253 253 253  10 253 253  15
253 253 253 253 253   9 253
 10 253   4 253   3 253 253
253 253   3 253   3 253 253
```

The right conjunction argument to `escapetc` is the bound for what has escaped and the maximum number of iterations to attempt. The right argument to the derived verb is the array of complex starting points. In the small example above, we see most points remained bounded for 253 iterations but a few isolated points escaped relatively quickly. Following [4] Section 7.6 we create an image size version as follows.

```
   b=:r :: _: escapetc (1e4 255) 800 zl_clur _5 15j2.5
   $pal3x=:255,0,~254{.<.,/( >:-:(i.%-)22) */(Hue (i.%])12)
256 3
   load '~addons/media/image3/view_m.ijs'
       view_image pal3x;b
801 201
```

Figure 1 shows a slightly higher resolution version of the image. Figure 2 shows the corresponding image when using T to create the escape times, as below.

```
   T=:[: -: mod2 + ] * 3 ^ mod2

   $b=:T :: _: escapetc (1e4 255) 800 zl_clur _5 15j2.5
201 801

   view_image pal3x;b
801 201
```

Note that both have fractal fingers rising, but the pattern in Figure 1 appears more regular.

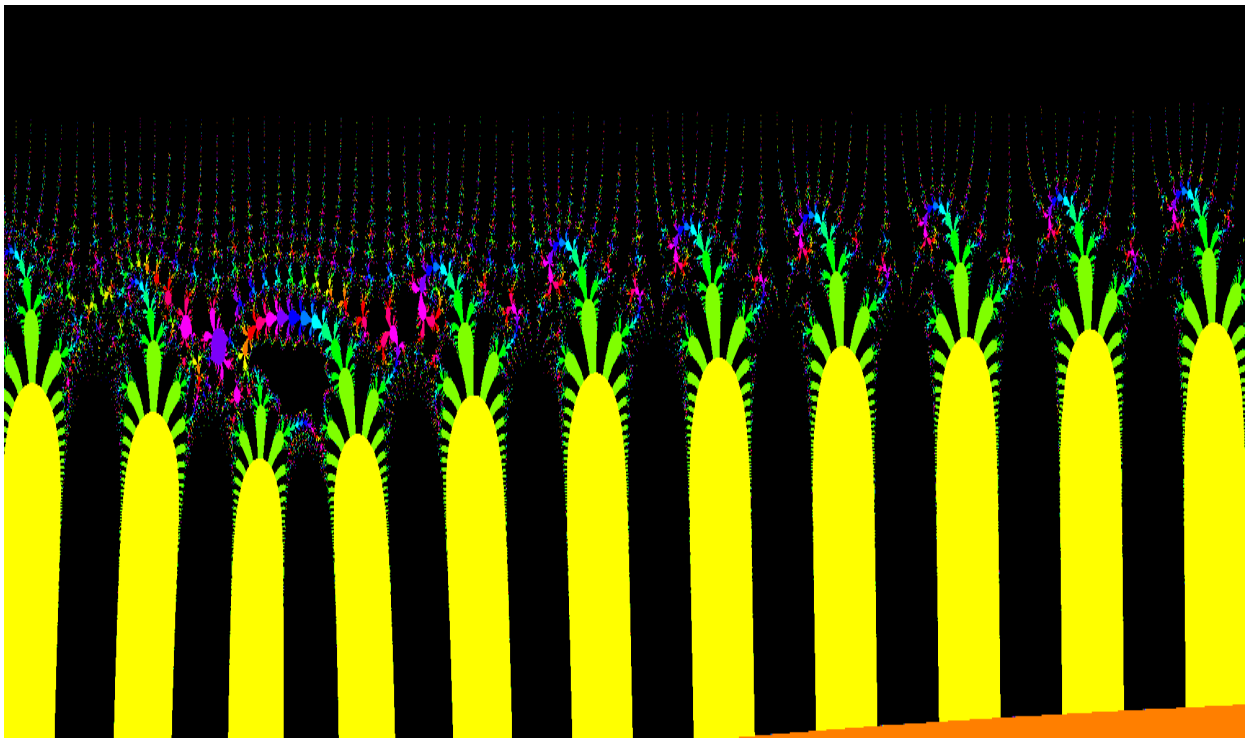


Figure 1. Escape time of $r(x)$ along the axis for $-5 \le \mathrm{Re}(z) \le 15$.

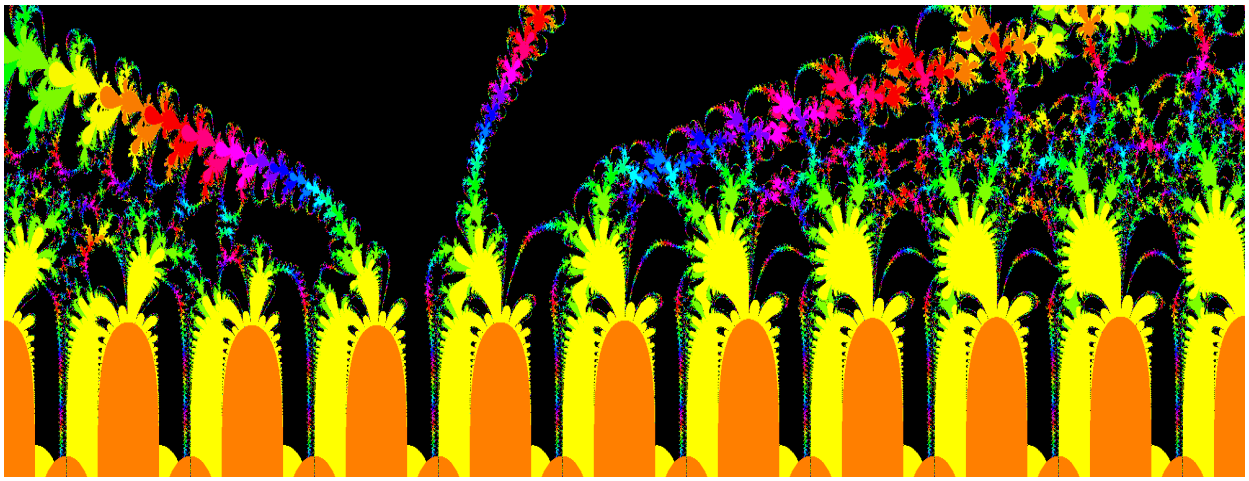


Figure 2. Escape time of $T(x)$ using the same complex exponential $\mathrm{mod}_2(x)$ along the axis for $-5 \le \mathrm{Re}(z) \le 15$.

2. Zooms

We saw that 17 eventually reaches the cycle (1,2).

```
   cclean@r^:(i.12)17
17 26 13 20 10 5 8 4 2 1 2 1
```

Thus, when we create an image containing 17, it will be a black pixel since it never escaped. We turn to considering zoomed regions around 17. Below are expressions that would yield an image around 17 with radius 1. Changing n in the expression would yield other radii. Figure 3 shows higher resolution versions for radii 1, 0.1 and 0.01. Figures 4 to 7 show further zooms up to 0.1^{13}.

```
   b=:r :: _: escapetc (1e4 255) 400 zl_clur 17+_1 1j1*0.1^n=:0
   view_image pal3x;b
401 401
```

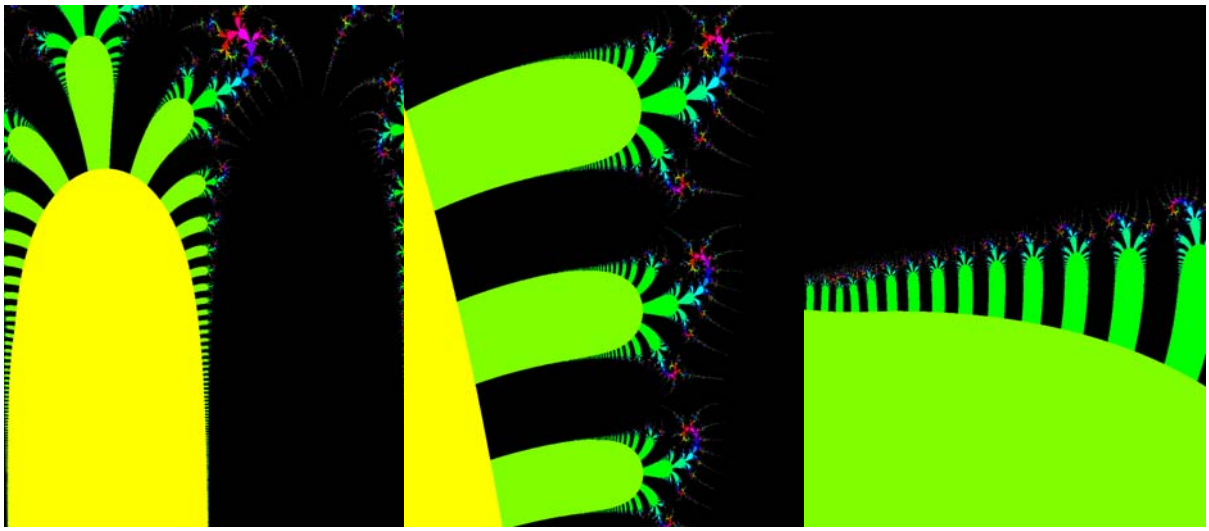


Figure 3. Zoom of escape time around 17 of radii 0.1^0 1 2.

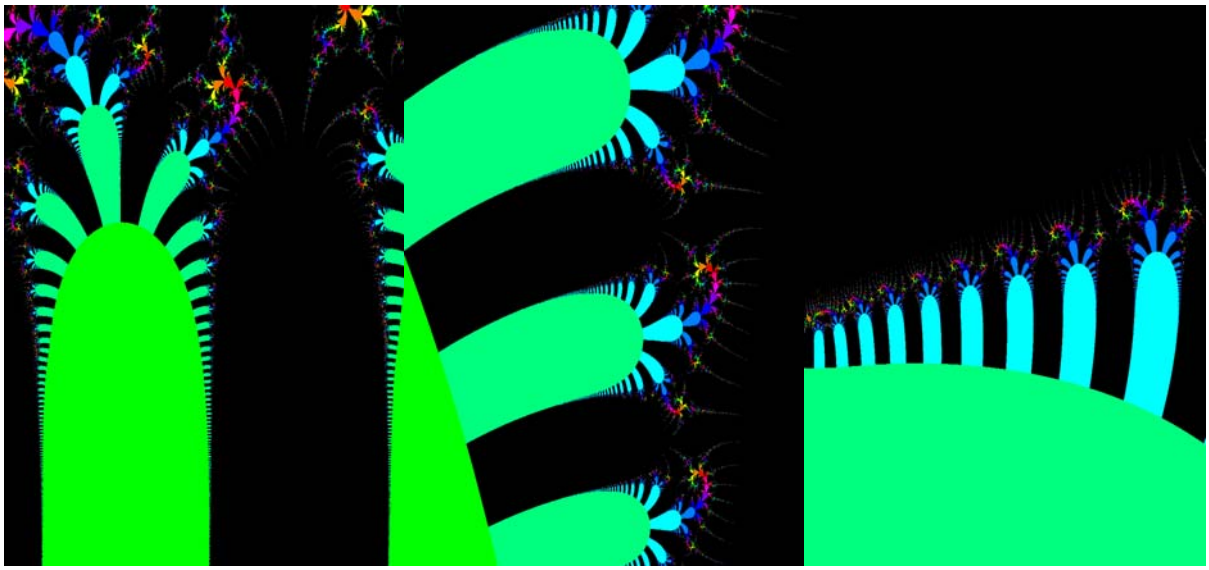


Figure 4. Zoom of escape time around 17 of radii 0.1^3 4 5.

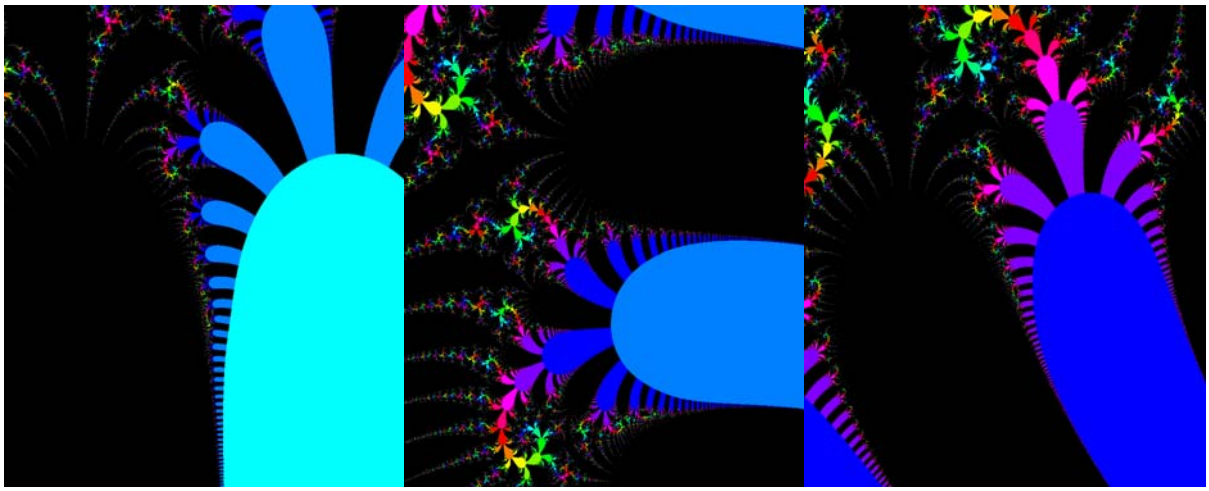


Figure 5. Zoom of escape time around 17 of radii 0.1^6 7 8.

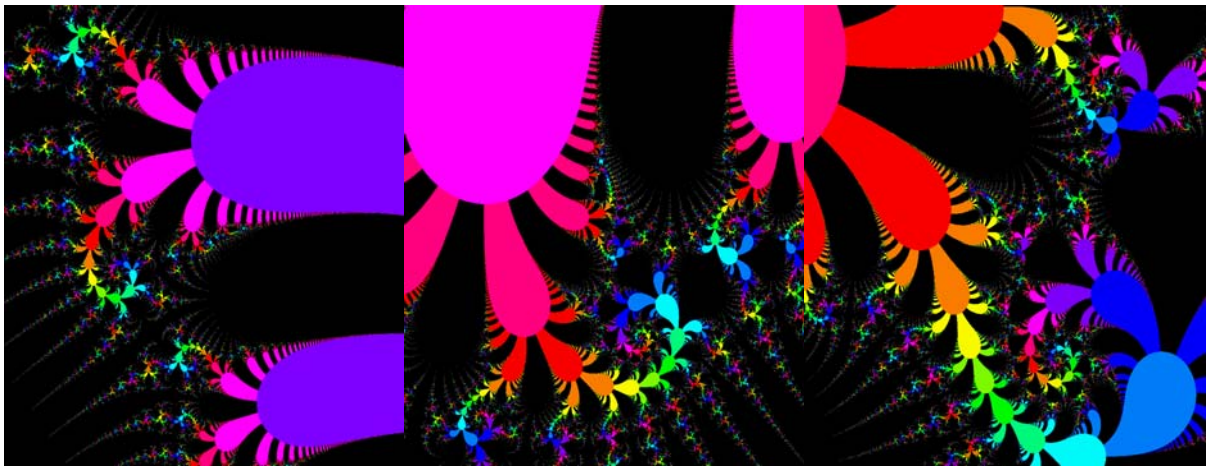


Figure 6. Zoom of escape time around 17 of radii 0.1^9 10 11.

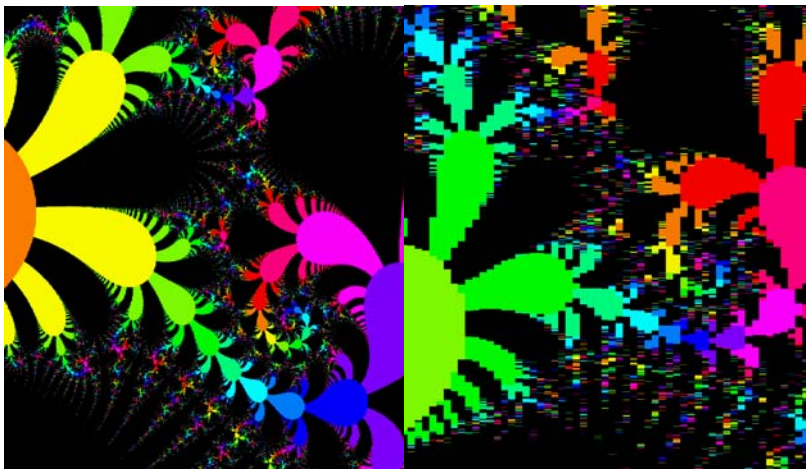


Figure 7. Zoom of escape time around 17 of radii 0.1^12 13.

Each image in Figures 3 to 7 has 17 as the central black pixel. In the last several images spirals are visible and 17 is intertwined with fractal structures. The last image has reached the point of pixilation since the resolution of the floating point arithmetic has been reached. The images in this note and a movie zooming from radius 10^2 to 0.1^{13} are available at [5].

References

[1] Jeffrey P. Dumont and Clifford A. Reiter, Visualizing Generalized 3*x*+1 Function Dynamics, *Computers & Graphics*, 25 5 (2001) 883-898.

[2] J. Dumont and C. Reiter, Complex dynamics of some generalizations of the 3*x*+1 function, *http://webbox.lafayette.edu/~reiterc/3x+1/index.html*.

[3] Bo Jacoby, Jprogramming forum, Simple Number Theory, March 3, 2014.

[4] Clifford A. Reiter, *Fractals, Visualization and J, 3rd Edition*, Published by Lulu, (2007).

[5] Cliff Reiter, Auxiliary materials for Visual Dynamics of the Complex 3*x*+1 Function Revisited, *http://webbox.lafayette.edu/~reiterc/j/JoJ/v3x+1_rv/index.html*, 2014.

****

****

**

*

Matrix Products

A collaborative work in J forums

Moderate by Mikel Paternain

1. Introduction

Not is the first time that *JoJ* propose a question to solve in J. Here we study some generalizations on matrix products are as defined in [1]. The question was posted in J forums and here there principal codes and answers.

2. Definitions

We consider matrices $\mathbf{A}=(a_{ij})$ and $\mathbf{C}=(c_{ij})$ of order $m\times n$ and $\mathbf{B}=(\mathbf{b}_{kl})$ of order $p\times q$. Let $\mathbf{A}=(\mathbf{A}_{ij})$ be partitioned with $\mathbf{A}_{ij}$ of order $m_i\times n_j$ as the (k,l)th block submatrix

$\left(\sum m_i=m,\ \sum n_i=n,\ \sum p_k=p \text{ and } \sum q_l=q\right)$. The definitions of the matrix products or sums of **A** and **B** are given as follows:

(i) Hadamard product

$$\mathbf{A}\odot\mathbf{C}=(a_{ij}c_{ij})_{ij},$$

where $a_{ij}c_{ij}$ is a scalar and $\mathbf{A}\odot\mathbf{C}$ is of order $m\times n$

(ii) Kronecker product

$$\mathbf{A}\otimes\mathbf{B}=(a_{ij}\mathbf{B})_{ij},$$

where $a_{ij}\mathbf{B}$ is of order $p\times q$ and $\mathbf{A}\otimes\mathbf{B}$ is of order $mp\times nq$

(iii) Khatri-Rao product

$$\mathbf{A}*\mathbf{B}=(\mathbf{A}_{ij}\otimes\mathbf{B}_{ij})_{ij},$$

where $\mathbf{A}_{ij}\otimes\mathbf{B}_{ij}$ is of order $m_ip_i\times n_jq_j$ and $\mathbf{A}*\mathbf{B}$ is of order $\left(\sum m_ip_i\right)\times\left(\sum n_jq_j\right)$.

(iv) Tracy-Singh product

$$\mathbf{A}\bowtie\mathbf{B}=\left(\mathbf{A}_{ij}\bowtie\mathbf{B}\right)_{ij}=\left(\left(\mathbf{A}_{ij}\otimes\mathbf{B}_{kl}\right)_{hl}\right)_{ij}$$

(v) Khatri-Rao sum

$$\mathbf{A}\circledast\mathbf{B}=\mathbf{A}*\mathbf{I}_p+\mathbf{I}_m*\mathbf{B}$$

(vi) Tracy-Singh sum

$$A \boxdot B = A \bowtie I_p + I_m \bowtie B$$

(vii) Vector cross product

$$a \times b = T_a b \ ,$$

where $a = (a_1, a_2, a_3)'$ and $b = (b_1, b_2, b_3)'$ are real vectors, and

$$T_a = \begin{pmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a1 & 0 \end{pmatrix}$$

3. Code

Mario quintana

```
NB. The Hadamard product is just * , for example,
   ( A=. i.2 3 )
0 1 2
3 4 5
   ( B=. >:*: A )
 1  2  5
10 17 26

   A;B
┌─────┬──────┐
│0 1 2│ 1  2  5│
│3 4 5│10 17 26│
└─────┴──────┘

   A * B
 0  2  10
30 68 130
```

Raul Miller

If I understand properly:

Hadamard product is `*`

Kronecker product is `([:,/[:,./"3*/)`

Or,

```
   block=: 1 : '[:,/[:,./"3 u'
   */ block
```

There are of course other ways of defining this.

Khatri-Rao product is `A */block &.> B` and works on a matrix of boxed matrices

Tracy-Sigh product is `(1 0 3 |: */)block&.>/block` and also works on a matrix of boxed matrices

Vector cross product gets interesting because it is typically defined on 3 element vectors and there are at least two very distinct ways of generalizing that to vectors of other lengths.

Mario Quintana

That is also my understanding. A couple of alternatives are:

```
   kp1=: *&$ ($,) 0 2 1 3 |: */
   kp2=. ,/"2@:(,/@:(1 3&|:@:(*/)))

   A=. i.3 4
   B=. >:*: i. 5 2

   A;B
```

0 1 2 3 4 5 6 7 8 9 10 11	1 2 5 10 17 26 37 50 65 82

```
A kp1 B

  0   0   1   2   2   4   3   6
  0   0   5  10  10  20  15  30
  0   0  17  26  34  52  51  78
  0   0  37  50  74 100 111 150
  0   0  65  82 130 164 195 246
  4   8   5  10   6  12   7  14
 20  40  25  50  30  60  35  70
 68 104  85 130 102 156 119 182
148 200 185 250 222 300 259 350
260 328 325 410 390 492 455 574
  8  16   9  18  10  20  11  22
 40  80  45  90  50 100  55 110
136 208 153 234 170 260 187 286
296 400 333 450 370 500 407 550
520 656 585 738 650 820 715 902

   A kp2 B

  0   0   1   2   2   4   3   6
  0   0   5  10  10  20  15  30
```

```
  0   0  17  26  34  52  51  78
  0   0  37  50  74 100 111 150
  0   0  65  82 130 164 195 246
  4   8   5  10   6  12   7  14
 20  40  25  50  30  60  35  70
 68 104  85 130 102 156 119 182
148 200 185 250 222 300 259 350
260 328 325 410 390 492 455 574
  8  16   9  18  10  20  11  22
 40  80  45  90  50 100  55 110
136 208 153 234 170 260 187 286
296 400 333 450 370 500 407 550
520 656 585 738 650 820 715 902
   kp0=. [: ,/ [: ,./"3 */
   A kp0 B

  0   0   1   2   2   4   3   6
  0   0   5  10  10  20  15  30
  0   0  17  26  34  52  51  78
  0   0  37  50  74 100 111 150
  0   0  65  82 130 164 195 246
  4   8   5  10   6  12   7  14
 20  40  25  50  30  60  35  70
 68 104  85 130 102 156 119 182
148 200 185 250 222 300 259 350
260 328 325 410 390 492 455 574
  8  16   9  18  10  20  11  22
 40  80  45  90  50 100  55 110
136 208 153 234 170 260 187 286
296 400 333 450 370 500 407 550
520 656 585 738 650 820 715 902
```

Linda Alvord

Mario Quintana

This is a space-time performance comparison:

```
   block=: 1 : '[:,/[:,./"3 u'
   kp3=. */ block

   st=. (] , <@:(1&({::) * 2&({::)))@:(] ; 7!:2@:] ; 6!:2)

   100 st &> 'A kp0 B' ; 'A kp1 B' ; 'A kp2 B' ; 'A kp3 B'
```

A kp0 B	3456	1.30442109e_5	0.045080793
A kp1 B	3392	1.29415788e_5	0.0438978354
A kp2 B	3392	1.30905609e_5	0.0444031827
A kp3 B	3456	1.2905161e_5	0.0446002363

They very are similar.

Linda Alvord

`Kp3` is defined monadicallly, but it creates the same function as `kp` and it gives the same results when used dyadically.

```
kp3=: 13 :',/,./"3*/y'


   (A kp3 B)-:A kp B
1
```

Raul Miller

I prefer to think of `kp3`'s implementation as ambivalent

```
   13 :',/,./"3*/y'
[: ,/ [: ,./"3 */
```

You could think of this as:

```
   ([: ,/ [: ,./"3 */) : ([: ,/ [: ,./"3 */)
```

The implementation on the left side of the separating : is monadic, the implementation on the right side is dyadic.

There are other ways of expressing this duality also. For example:

```
   (i.3 3) ([: ,/ : (,/) [: ,./"3 :(,./"3) */ :(*/))  i. 2 2
 0  0  0  1  0  2
 0  0  2  3  4  6
 0  3  0  4  0  5
 6  9  8 12 10 15
 0  6  0  7  0  8
12 18 14 21 16 24
```

of, for example:

```
   (i.3 3) ([: , :,/ [: ,. :,./"3 * :*/) i. 2 2
```

Basically, a verb is really a reference to a pair of definitions (monadic verb definition and dyadic verb definition) and that holds true for both primitive and derived verbs. This is a bit of a subtle point, since to reason about it we need to draw a distinction between verbs-as-symbols and verbs-as-their-implementations.

José Mario Quintana

Your definitions reproduce the examples in

http://en.wikipedia.org/wiki/Khatri-Rao_product#Khatri-Rao_product :

```
    ( A=. (;~1 0 1) <;.1 (1+i.3 3) )
```

1 2 4 5	3 6
7 8	9

```
( B=. (;~1 1 0) <;.1 (|: 1+i.3 3) )
```

1	4 7
2 3	5 8 6 9

```
   krp=. */block &.>  NB. Khatri-Rao product
   A krp B
```

1 2 4 5	12 21 24 42
14 16 21 24	45 72 54 81

```
tsp=. (1 0 3 |: */)block&.>/block  NB. Tracy-Singh product

A tsp B
```

1 2 4 5	4 7 8 14 16 28 20 35	3 6	12 21 24 42
2 4 8 10 3 6 12 15	5 8 10 16 20 32 25 40 6 9 12 18 24 36 30 45	6 12 9 18	15 24 30 48 18 27 36 54
7 8	28 49 32 56	9	36 63
14 16 21 24	35 56 40 64 42 63 48 72	18 27	45 72 54 81

They are impressive!

José Mario Quintana

This is yet another way to define the products using an idea from Victor Cerovski (see, http://www.jsoftware.com/jwiki/Essays/Kronecker%20Product):

```
   u Block=. (,./^:2)@:
,./^:2@:u

   ( kp=. */ Block )                          NB. Kronecker
,./^:2@:(*/)
   ( krp=. */Block &.> )                      NB. Khatri-Rao
,./^:2@:(*/)&.>
   ( tsp=. (1 0 3 |: */)Block&.>/Block )      NB. Tracy-Singh
,./^:2@:(,./^:2@:(1 0 3 |: */)&.>/)
```

```
  A krp B
```

1 2 4 5	12 21 24 42
14 16 21 24	45 72 54 81

```
  A tsp B
```

1 2 4 5	4 7 8 14 16 28 20 35	3 6	12 21 24 42
2 4 8 10 3 6 12 15	5 8 10 16 20 32 25 40 6 9 12 18 24 36 30 45	6 12 9 18	15 24 30 48 18 27 36 54
7 8	28 49 32 56	9	36 63
14 16 21 24	35 56 40 64 42 63 48 72	18 27	45 72 54 81

The Khatri-Rao and the Tracy-Singh sums do not seem difficult to implement but I would rather wait to see concrete, inputs and outputs, examples before trying to code them.

I forgot, to remove the boxes one can use:

```
   rb=. >@:(,.&.>/)@:(,&.>/)

   rb A tsp B
 1  2  4  7  8 14  3 12 21
```

```
 4  5 16 28 20 35  6 24 42
 2  4  5  8 10 16  6 15 24
 8 10 20 32 25 40 12 30 48
 3  6  6  9 12 18  9 18 27
12 15 24 36 30 45 18 36 54
 7  8 28 49 32 56  9 36 63
14 16 35 56 40 64 18 45 72
21 24 42 63 48 72 27 54 81
```

References

[1] HADAMARD, KHATRI-RAO, KRONECKER AND OTHER MATRIX PRODUCTS Shuangzhe Liu & Götz Trenkler. International Journal of Infornation ans Systems Sciences Vol. 4, N. 1, 160-177.

Changing Basins of Attraction for Rayleigh Quotient Iteration

Charles Timko, timkoc@lafayette.edu

Student at Lafayette College

INTRODUCTION

Rayleigh Quotient Iteration uses the Rayleigh Quotient algorithm to estimate eigenvalues and eignevectors of a given matrix [2]. It is an iterative method, which means it must be repeated until it converges and an eigenvalue is found or until the maximum iteration count is reached. The images generated by the computer show the basins of attraction of the eigenvalues. Matrix six offered the most interesting image of the eight different matricies so it was chosen as the matrix to be varied [1]. By using the verb*:* f=: 3 :'y(<3 3)}m6', we are able to vary the entry in index three (the third row, third column of the matrix). The pictures show various alterations of this entry ranging from -10 to 1000. The images are also shown in a movie file for simplicity using the last section of code (see link to movie below). From the movie, we can see how varying this one entry affects the computer generated image of the basin of attraction for the sixth matrix. The black section of the images corresponds to points where an eigenvalue was not found before the maximum iteration count occurred.

Link to Movie - http://webbox.lafayette.edu/~reiterc/sp14fp/RayleighQuotientMOVIE.mov

SCRIPT

```
require '~addons\math\lapack\lapack.ijs'
require '~addons\math\lapack\dgeev.ijs'
require '~addons\media\image3\view_m.ijs'

mp=: +/ . *
elen=: (+/&.:*:)@:|
unit=: % elen

m1=: 1|.=i.5
m2=: 1,9 8 4,:5 6 2
m3=: 1,9j1 8 4,:5 6 2
m4=: ".;.(_2) 0 : 0
_6 7 1 _5
7 10 3 _4
1 3 8 _7
_5 _4 _7 8
)
m5=: ".;.(_2) 0 : 0
2 _9j_6 _3j_1 _3j1
_9j6 _4 _9j2 _6j6
_3j1 _9j_2 2 0j_3
_3j_1 _6j_6 0j3 2
```

```
)
m6=: ".;.(_2) 0 : 0
1 2 3 4
2 5 6 7
3 6 8 9
4 7 9 10
)

rqik=: 4 : 0"_ 0
A=. x
k=. 0
x=. x0_rqi
u=. y
while. k<18 do.
  try. x=. unit x %. A - u *I_rqi catch. break. end.
  u=. (+x) mp A mp x
  k=. k+1
end.
({./:|EV_rqi-u),k
)

mk_rqi=: 1 : 0
:
EV_rqi=: (/: |.&.|:@:*.)>1{dgeev_jlapack_ x
I_rqi=: =i.#x
x0_rqi=: (#x)#1
z=. m zl_clur0 y
'b k'=. 0 1|:x rqik z
```

```
p18p;(18~:k)*1+(3|k)+3*b
)

mk_rqi_pic=: 1 : 0
:
view_image x m mk_rqi y
)

zl_clur0=: 4 : 0
w=.-~/9 o.y
h=.-~/11 o.y
(j.h-(+:h)*(i.%<:)x) +/ ({.y)+w*(i.%<:)x
)
$p18p=: 0,,/0 2|:<.1 0.72 0.45 */ Hue 0 2 4 1 3 5{(i.%])6

m6 500 mk_rqi_pic _15 9j12

f=: 3 :'y(<3 3)}m6'
f 11

(f 11) 500 mk_rqi_pic _15 9j12
((f 11) 500 mk_rqi _15 9j12) write_image 'm6(11).png'

path=:'U:\Math 379\j602\bin\ProjectRayleigh\'
load '~addons/graphics/fvj3/automata.ijs'
ProjectRayleigh_fns=:images_in path
1 fseq_to_png_mov ProjectRayleigh_fns;path,'RayleighQuotient.mov'
open_html path,'RayleighQuotient.mov'
```

SCRIPT AND EXAMPLE IMAGES

```
..........Load Previous Script..........
```

```
m6 500 mk_rqi_pic _15 9j12
```

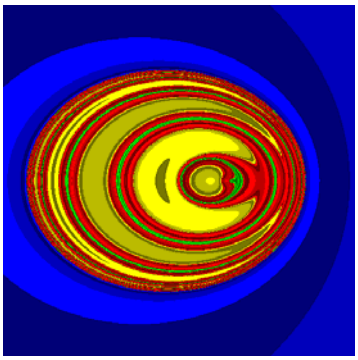


Figure 1. Matrix 6 with index 3 unchanged

```
f=: 3 :'y(<3 3)}m6'
f 11
```

```
(f 11) 500 mk_rqi_pic _15 9j12
((f 11) 500 mk_rqi _15 9j12) write_image 'm6(11).png'
```

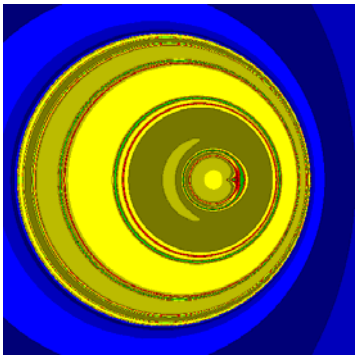


Figure 2. Matrix 6 with index 3 changed to 11

```
f 15
```

```
(f 15) 500 mk_rqi_pic _15 9j12
((f 15) 500 mk_rqi _15 9j12) write_image 'm6(15).png'
```

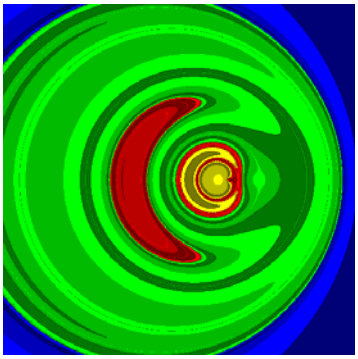


Figure 3. Matrix 6 with index 3 changed to 15

```
f 05
```

```
(f 05) 500 mk_rqi_pic _15 9j12
((f 05) 500 mk_rqi _15 9j12) write_image 'm6(05).png'
```

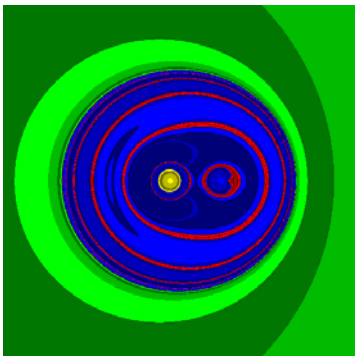


Figure 4. Matrix 6 with index 3 changed to 05

REFERENCES

[1] Clifford A. Reiter, Rayleigh Quotient Iteration Script, http://webbox.lafayette.edu/~reiterc/mvp/rqi/rq_pic_601.html, 2013

[2] Clifford A. Reiter, Rayleigh Quotient Iteration, http://webbox.lafayette.edu/~reiterc/mvp/rqi/index.html, 2013

Fractal Characterization of Turbulent Flow

K. Lawrence Galloway

B.S. Mechanical Engineering, Lafayette College

klgalloway4 AT gmail DOT com

Abstract

Turbulent flow is often thought of as a chaotic process. In this study turbulent flow bubble visualizations were analyzed using J in order to assess this claim. The capacity dimension and Hurst exponent were used to consider how the bulk patterns of the visualizations changed from frame to frame. Several verbs and adverbs were written to calculate these quantities over a large range of frames for a specified threshold. Two flow cases were considered: a laminar case in which the flow was locally 'tripped' to turbulence and a fully turbulent flow. For the laminar case, the capacity dimension was seen to rise and fall with the formation and decay of turbulent flow, with a peak capacity dimension of 1.65 ± 0.05. The fully turbulent case had a capacity dimension of 1.891 ± 0.023 and showed the capacity dimension to be persistent with a Hurst exponent of 0.771. These patterns may be useful for characterizing turbulence and showing transitions. Due to the connection between the capacity dimension and the correlation dimension, which is considered to be connected to chaos, it may be possible to extend these calculations to make a connection to chaos.

Introduction & Methods

A flowing fluid can either be laminar or turbulent. Laminar flow is characterized as being orderly and steady with time; laminar flow corresponds to a high amount of viscosity within the flow

relative to the amount of inertia of the fluid. As the fluid's inertia rises, it will transition to turbulence, which is characterized as being disorderly and unsteady. The Reynolds Number is a dimensionless quantity that is a ratio of the inertial forces to the viscous forces within a fluid. It can be used to characterize how turbulent or laminar a flow is [1].

It is often stated that turbulent flow is the perfect example of a random process. While this seems to be believable upon first viewing turbulent flow visualizations, it becomes apparent after further viewing that there is in fact an underlying pattern. This phenomenon of order arising within a seemingly random time varying system is familiar to anyone that has spent time watching a camp fire. In fact, turbulent flow visualizations can look strikingly similar to fire. For this study, bubble wire flow visualization video was analyzed in an attempt to quantify both the order and chaos within a single instant and the amount of order the flow had from one moment to the next. All of the flow visualization video considered, was taken by Dr. Sabatino of Lafayette College. D. Sabatino (personal communication, May 08, 2014).

The capacity dimension (fractal dimension, D_0) is a measure of how a figure fills space. More precisely, it is the exponent that describes the rate at which the number of boxes to cover a figure grows as the size of the boxes is decreased. This is defined by

$$D_0 = \lim_{\varepsilon \to 0^+} \frac{\ln(N(\varepsilon))}{\ln(1/\varepsilon)} \tag{1}$$

where $N(\varepsilon)$ is the number of boxes required and ε is the length scale of the boxes [2].

In order to determine the capacity dimension of individual frames, a box counting algorithm was implemented to work using Eq. (2). Written in J, the underlying algorithm was borrowed from Dr. Reiter via his book, *Fractals, Visualization, and J* and incorporated into a box counting monadic verb called Boxing (shown below). The box counting algorithm works by counting the number of 'boxes' of a certain size that it takes to cover over a figure defined in a binary two-dimensional array. The algorithm is written such that it detects and counts ones and disregards zeros.

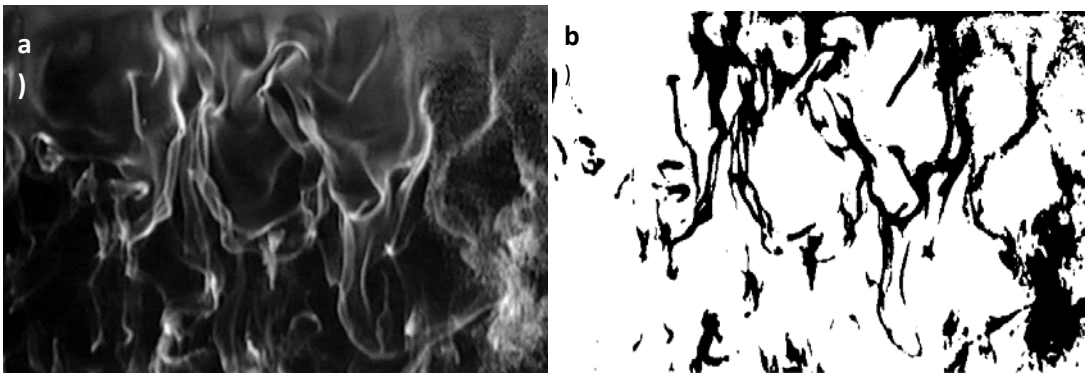


Figure 1. An example threshold taken from the constant Reynolds Number case at a threshold of 75.

In order to create a binary two-dimensional array to pass to Boxing it was necessary to take a threshold. This reduced the video frames to a black and white, figure-ground representation of the turbulent flow. An additional monadic verb called `Thresh` (shown below), was written to do this. Threshold's output was then taken as Boxing's input. Figure 1 shows an example of an input frame and its output threshold. The bubbles are output from Thresh as being black so that Boxing would pick them up instead of the backdrop.

```
Thresh=: 3 : 0
c=: read_image jpath path, y
avg=: +/ % #
clean=: * * |                                   NB. Define a rounding tool
wb_pal=: 255,:0 0 0
wb=: T>avg"1 c                                  NB.Threshold according to a globally
                                                NB.defined Threshold value
wb=: clean 2&o. wb*1r2p1                        NB.Reverse the binary values so that the
)                                               NB.bubbles are black


Boxing=:  3 : 0
avg=: +/ % #
sd=: [: %: ([: +/ [: *: ] - avg) % <:@#
onein=: 1 e. ,                                  NB.A tool that tests for ones
N=: [: +/@, (,:~@,~)@[ onein;._3 ]              NB.Count the numbers occupied
eps=: % #                                       NB.Define epsilon
```

```
fd=: (N %&^. %@eps)"0 2

a4=: 4 fd Thresh y                          NB.fd takes in neighborhood values

a6=: 6 fd Thresh y

a8=: 8 fd Thresh y

a10=: 10 fd Thresh y

a12=: 12 fd Thresh y

AVG=: avg a4,a6,a8,a10,,:a12                NB.Average the fractal dimension over a
                                            NB.range of neighborhood sizes

U=: 1.241*(sd a4,a6,a8,a10,,:a12)           NB.Compute the uncertainty in the box
                                            NB.count due to neighborhood size

Upper=: AVG+U

Lower=: AVG-U

AVG, Lower, Upper

)
```

To choose an appropriate threshold a combination of visual inspection and numerical techniques were used. In other words, the threshold shown in Figure 1b is a good visual representation of the patterns formed by the bubbles shown in Figure 1a. In order to locate the appropriate threshold quickly, plots of capacity dimension against threshold value were created. These plots were used to see what the maximum difference between turbulence and laminar flow was. An example plot is shown in Figure 2. This analysis allowed for reliable threshold values to be chosen. This ensured that the analysis of multiple frames at a time could be made reliably. As Figure 2 hints at, the capacity dimension of the laminar frame is unstable above a certain threshold. This instability is due to the bubbles being entirely filtered out after a certain threshold, leaving nearly white frames. Additionally the capacity dimension is very similar below a certain threshold.

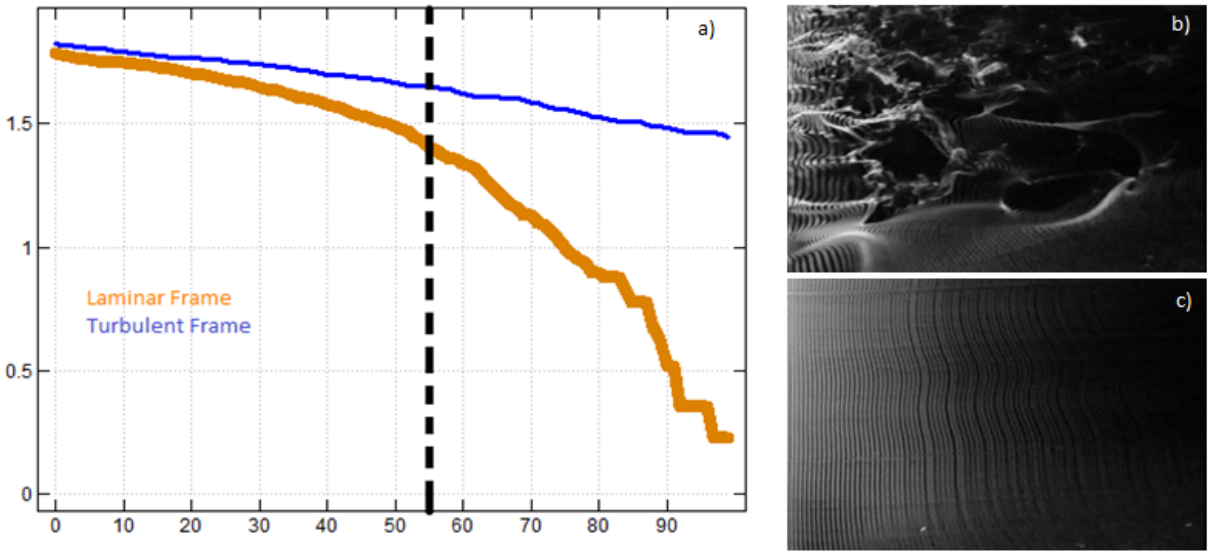


Figure 2. a) Shows the capacity dimension as related to threshold. The threshold ranges from 90 to 190 with the chosen value of 145 marked. b) and c) are the frames considered.

An additional function was written in order to calculate the Hurst exponent of the capacity dimension's change with time. This was done to gain an understanding of how persistent a turbulent flow's pattern is with time. The verb, called Hurst, shown below, is based entirely on the code shown in *Fractals, Visualization, and J* used to analyze sunspot data. When the Hurst exponent is between 0.5 and 1.0, the data tends to be persistent, meaning that a shift up will likely be followed by another shift up and a shift down by yet another shift down [2].

```
Hurst=: 3 : 0
l=: y
min=: <./
max=: >./
ccd=: [: +/\ ] - avg                    NB.Cumulative centered distribution
R=: (max-min)@:ccd                      NB.The range in the data
S=: [: %: ([: +/ [: *: ] - avg) % #     NB.Sample standard deviation
Q=: R % S
RS=: ([: avg ,:~@[ Q;._3 ] )"0 1        NB.Calculate the Hurst Exponent
N=: 2+i.<.-:# l
rs=: N RS l
```

```
}. ((^.rs)%.1,.^.N)
)
```

In addition to the verbs shown above, three verbs were written to plot the capacity dimension as it relates to time, generate a list of file names for the individual frames, and calculate the statistical data (standard deviation and average) for the case of constant Reynolds Number. These functions allow for the analysis of many frames so as to show the changes within the flow over time and the capacity dimension's value as it corresponds to Reynolds Number. The uncertainty in the capacity dimension due to neighborhood size was calculated from neighborhoods of size 6, 8, 10, and 12 as seen in Boxing. All uncertainties are reported to 95% confidence.

The frames of two videos were analyzed. The first case is a spot formation, meaning that the Reynolds Number is spiked locally, tripping the flow to turbulence from a resting laminar flow. The second case is a video of a constant Reynolds Number of 220,000, which is thoroughly turbulent. The second case included video of a pulsing stream of bubbles and a constant stream of bubbles. Only the constant stream video was considered to ensure that the pulsing would not interfere with the accuracy of the Hurst exponent. Threshold and frame information are included in Table 1.

For both visualizations, the bubble wire was placed at non-dimensional distance (y^+) above the channel bottom of about 5 and the frames captured, show 8 inches in eh horizontal direction. The Reynolds number was calculated with respect to the length of channel that the flow had passed through (known as Re_x). It is believed that the local spikes of turbulence do not reach Re_x values that are as high as the constant Re_x case being considered. This means that it can be expected that the Re_x value should be higher for the constant case.

Results

Both cases show trends within the capacity dimension and time graphs. The spot formation case shows that the capacity dimension rises as Reynolds Number rises and the constant Re_x case shows that capacity dimension is persistent with time. Figure 3, shows the change in D_0 as time passes in the spot formation case. As turbulence develops, the capacity dimension also rises. Important trends to note are the sudden rise, the slight stabilization, and then the drop off that is more gradual than the rise. Interestingly, the capacity dimension dips below the initial value and then rises slightly as

laminar flow is restored. It seems almost as if D_0 oscillates about the laminar value as the flow is released from turbulence.

Because of the way that turbulence was incited within the flow, estimates of Re_x during the trip are impossible. However it is believed the Re_x achieved during the trip to turbulence is lower than the constant case. As shown in Table 1, the average D_0 of the constant Re_x case is above the peak value reached during the turbulent trip case. This helps to confirm the trend of increasing D_0 with increasing Re_x. The uncertainty of D_0 for the constant case is small enough that very little variation is observed within the data. Figure 4 confirms this.

Another interesting feature that both cases have is the change in the uncertainty in D_0 according to neighborhood size with the change in frame. As shown by Figure 3, this uncertainty is especially low during the transitions and comparatively high during turbulence. Further investigation of the transition frames to determine a possible cause of this affect would be an interesting pursuit. The uncertainty in the turbulent flow is higher than the uncertainty in the laminar flow by 60%. This is due to the capacity dimension's sensitivity to irregular shapes; as the irregularity of the thresholds grows so too does the sensitivity to neighborhood size.

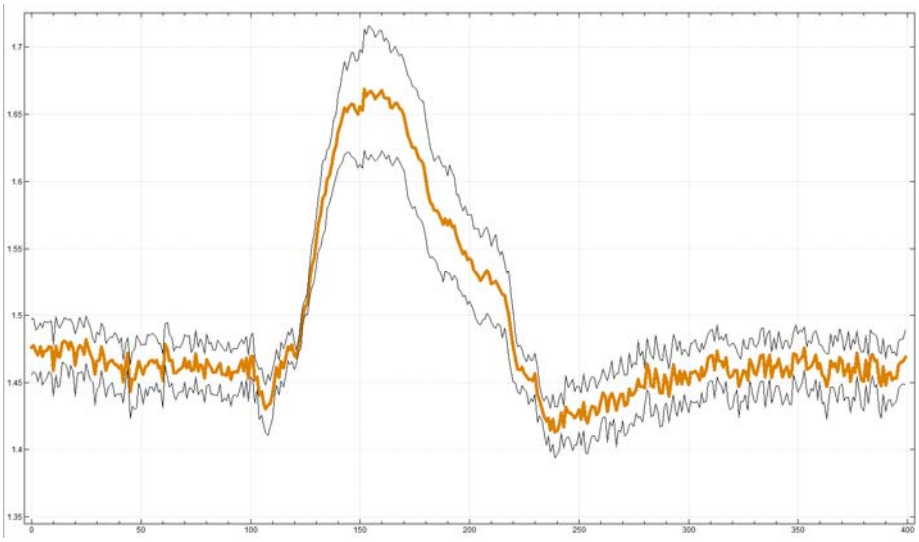


Figure 3. A plot of D_0 with respect to time (by frame) for the turbulent spot case. The rise in D_0 is immediately obvious.

Unsurprisingly, the transition from laminar to turbulent flow is persistent. The Hurst exponent being one (given in Table 1) reflects the fact that the capacity dimension is often steadily growing or falling from one frame to the next over the range considered. The Hurst exponent of D_0 for the constant Re_x

(given in Table 1) case is within the range for persistence. Being in the middle of the range shows that the changes are fairly persistent, but still maintain some chaotic fluctuations. This pattern can be seen in Figure 4. Interestingly, D_0 seems to stabilize for about 25 frames and then suddenly shift, only to stabilize again.

Case	Frames Considered	Threshold [RGB]	Capacity Dimension	Hurst Exponent
Turbulent Spots	400	145	1.65±0.05 (peak)	1.03
Constant Re_x	315	75	1.891±0.023	0.771

Table 1. Table showing the number of frames considered, the threshold, capacity dimension, and the Hurst exponent for both cases.

The Hurst exponent shows that for both cases the capacity dimension changes as expected; when transitioning, the chaotic patterns of fluid flow grow in an orderly way and when the flow is turbulent the patterns change with a certain amount of randomness, but also order. This may confirm the idea that while turbulent flow does have random aspects, there is an overarching measurable pattern.

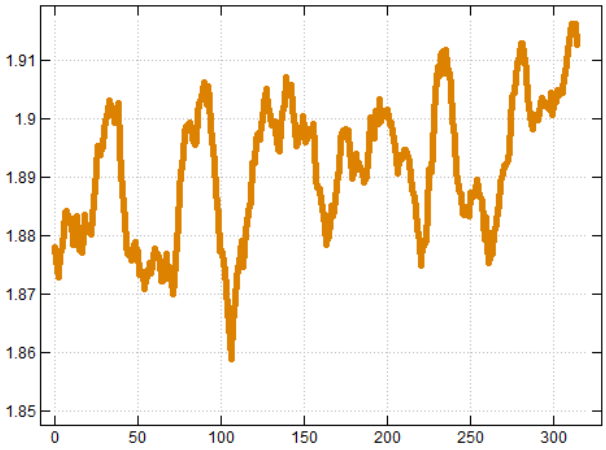


Figure 4. A plot of D_0 with respect to time (by frame) for the constant Re_x case.

Conclusion

Turbulent flow is thought of a random process without pattern, but it has order at a fundamental level. Several verbs were written in J to calculate the capacity dimension as related to frames of flow visualization. Two cases were considered: one of a constant turbulent flow and another of a local turbulent development. Both cases show that turbulence has an underlying pattern that can be quantified numerically. The rise in capacity dimension with Reynolds Number was shown and the Hurst exponent of the constant Reynolds Number case was shown to be persistent. The

capacity dimension corresponding to a Reynolds Number of 220,000 was found to be 1.891±0.023 with a Hurst exponent of 0.771. This technique may be useful in characterizing turbulent flow.

References

1. Munson, B, Young, D, Okiishi, T, & Huebsch, W. (2009). *Fundamentals of Fluid Mechanics*.
2. Reiter, C. (2007). *Fractals, Visualization and J, 3rd edition*. Easton, PA: Lafayette College

The Use of Histories and Predictions in Expanding Image Size

By Brad O'Brien,
B.S. Mechanical Engineering, Lafayette College
(410) 782-5095
bmobrien@princeton.edu
In association with Professor Cliff Reiter
Lafayette College Department of Mathematics (610) 330-5277 226 Pardee Hall
reiterc@lafayette.edu

1. Abstract

This report outlines the usage and purpose of the embedded code, which can be used to double an image size and intelligently guess the intermittent pixels based on a series of pixel histories developed from the initial image. This code is based off of the code provided in section 8.5 of *Fractals, Visualization, and J*, which discusses the correction of distorted images by a similar method1. The program uses a series of histories in order to predict the intermittent spacing's of a larger image. The image result is in grayscale.

Keywords: *histories; pixel; image; grayscale; tessellations; super pixel*

2. Code Summary

The code uses a sample 144 by 144 resolution image of a rubber duck, nicknamed "ducky" for convenience. The image, seen below in Fig. 1, is loaded and converted to grayscale.

```
load '~addons/media/image3/view_m.ijs'
```

```
ducky=: 'U:\Visualization\Ducky_Head_Web_Low-Res.jpg'
view_image ducky
$b=: read_image ducky
avg=: +/ % #
$gray=: <.avg"1 b
view_image BW256;gray
```

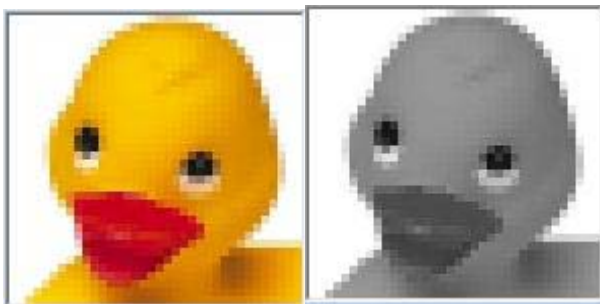


Fig. 1: The tested 144 by 144 pixel image and the grayscale equivalent.

A series of pixel histories are created based on a 1 by 1 offsets of 3 by 3 tessellations of the grayscale image in which the 0, 2, 6, and 8th indices are extracted.

```
$hist=:,/(1 1,:3 3)0 2 6 8&{@,;._3]gray
$pred=:,/ (1 1,:3 3)1 4 5&{@,;._3]gray
```

These are then used to predict the 1, 4, and 5th indices. Together with the 2nd indices from the history, these fulfill the 2 by 2 box in the upper right hand corner of the 3 by 3 tessellation. Using the function for all blocks and then rearranging the data gives the enlarged image.
The prediction function examines the image and makes guesses as to the values of the missing pixels by minimizing the distance between the shape of the current history and the shape of all other histories.

```
predfunc=: 3 : 0
try=: {.@/: hist dist, y
p=: try{pred
2 2 $(0{p),(1{,y),(1 2{p)
)
```

The indices of the resulting matrix are rearranged and combined to form the end result, seen in Fig 2.

```
$newimage=: (1 1,:2 2)predfunc;._3 gray
$newimage2=: 286 286$, 0 2 1 3 |: newimage
```

The image is compared to the super pixel version of the original file (right).

```
view_image BW256;newimage2
spix=: [##"_1
view_image BW256;2 spix gray
```

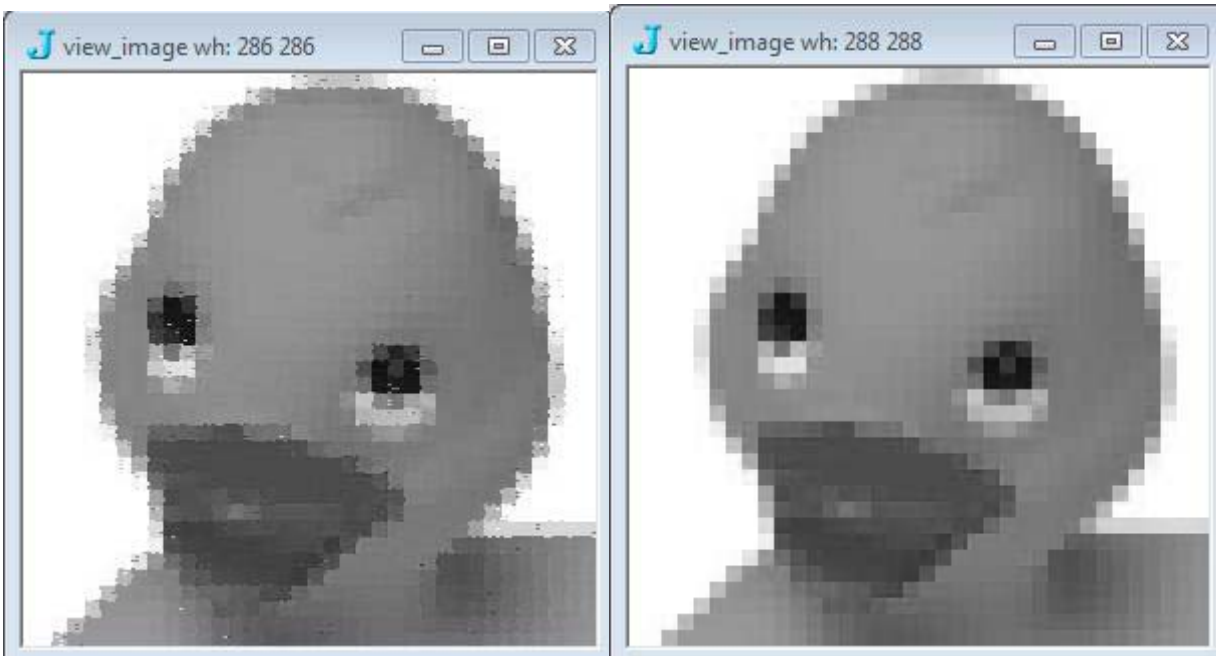


Fig. 2: The code results (left) as well as a super pixel comparison (right).

3. Limitations

There are some similarities and differences between the image created using the code and the super pixel equivalent. First of all, the new image is actually 286 by 286, not exactly double the initial image. This results from one axis being dropped when the axis are combined. Both images still appear to be rather poor quality. However, the new image has much more rounded features, particularly around the outside of the shape. This is expected and desired. There are some flaws, such as several white pixels existing in the ducks eyes where they should not be. This results from the code extracting the first history which meets the minimum distance criteria, a non-ideal result. This issue is much more prevalent in black and white images, but is not critical in grayscale images. This code could be run additional times to further increase the pixel size of the image, but with each subsequent runs, additional errors will begin to surface.

4. References

Reiter, C. A. (2007). Rotation, tilt and barrel distortion. *Fractals visualization and J* (3rd ed., pp. 178) Lulu.

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