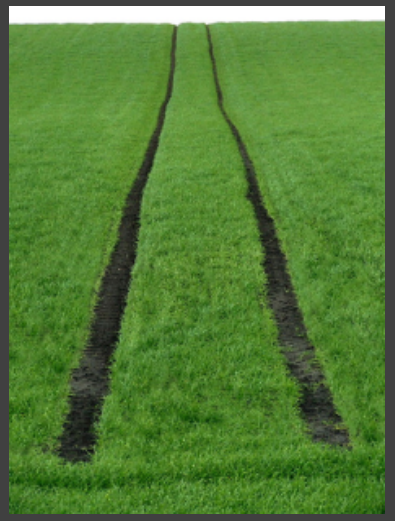
Efficient Decomposition of Dense Matrices over GF(2)

Martin R. Albrecht[1] and Clément Pernet[2]

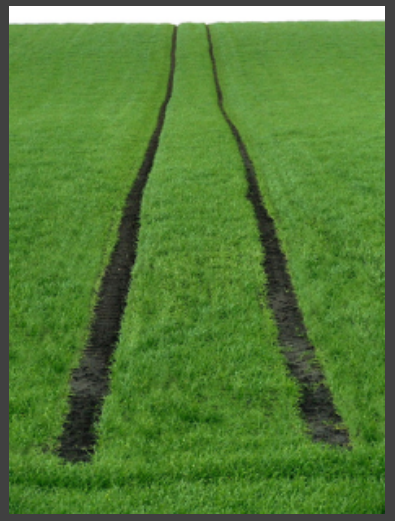
1 Information Security Group, Royal Holloway, University of London
2 INRIA-MOAIS LIG, Grenoble Univ. ENSIMAG

Tools for Cryptanalysis 2010, Egham, UK

Outline



Outline



Introduction

- Matrix decomposition is an essential building block for solving dense systems of linear and non-linear equations.
- Much research has been devoted to improve the asymptotic complexity of such algorithms.
- Various matrix decompositions can be reduced to matrix multiplication ($\mathcal{O}(n^\omega)\,,\ \omega \leq \log_2 7$).

In this work we focus on matrix decomposition in the special case of $\mathbb{F}_2$ and discuss an implementation of both well-known and improved algorithms in the M4RI library [AB09].

Does Dense Matter?

Time SSE2 can perform 128 $\mathbb{F}_2$ additions in one instruction. If time is the main concern then this provides a rough estimate how sparse the matrix should be to switch to sparse methods.

Space For non-linear system solving, we usually need echelon forms which suffer from fill-in effect.

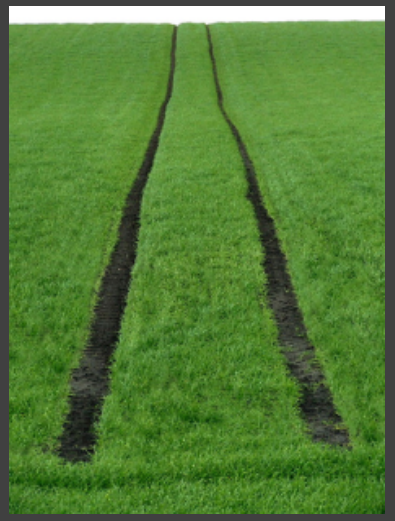
Sometimes matrices are just dense.

The M4RI Library

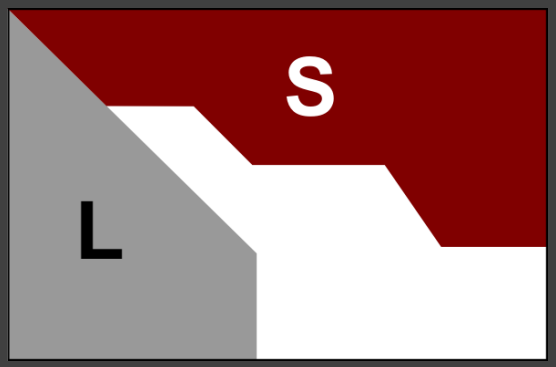
- available under the GPL Version 2 or later (GPLv2+).
- provides basic arithmetic (addition, equality testing, stacking, augmenting, sub-matrices, randomisation, etc.).
- implements asymptotically fast multiplication [ABH09].
- implements asymptotically fast decomposition (this talk).
- Linux, Mac OS X (x86 and PPC), OpenSolaris (Sun Studio Express) and Windows (Visual Studio and Cygwin).

`http://m4ri.sagemath.org`

Outline



PLS Decomposition I



Any $m \times n$ matrix A with rank r, can be written $A = PLS$ where P is a permutation matrix of dimension $m \times m$, L is $m \times r$ unit lower triangular and S is an $r \times n$ matrix which is upper triangular except that its columns are permuted, that is $S = UQ$ for U $r \times n$ upper triangular and Q is a $n \times n$ permutation matrix.

PLS decomposition can be in-place, that is L and S are stored in A and P is stored as an m-vector.

PLS Decomposition II

From the PLS decomposition we can

- read the rank r,
- read the row rank profile (pivots),
- compute the null space,
- solve $y = Ax$ for x and
- compute the (reduced) row echelon form.

The PLS decomposition computed in this work is computes S such that it is in row echelon form.

Asymptotically Fast PLS Decomposition

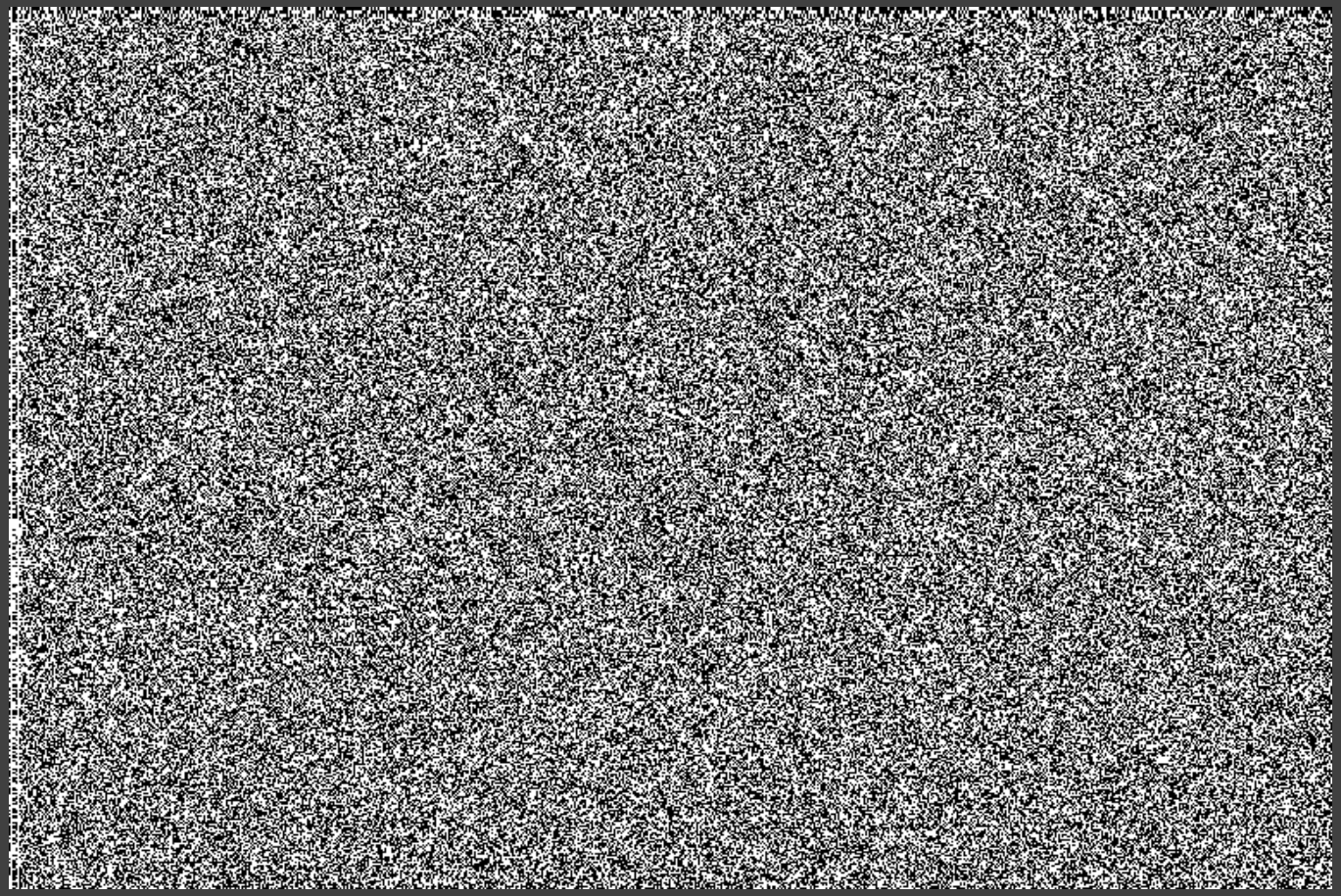
Write

$$A = \begin{pmatrix} A_W & A_E \end{pmatrix} = \begin{pmatrix} A_{NW} & A_{NE} \\ A_{SW} & A_{SE} \end{pmatrix}$$

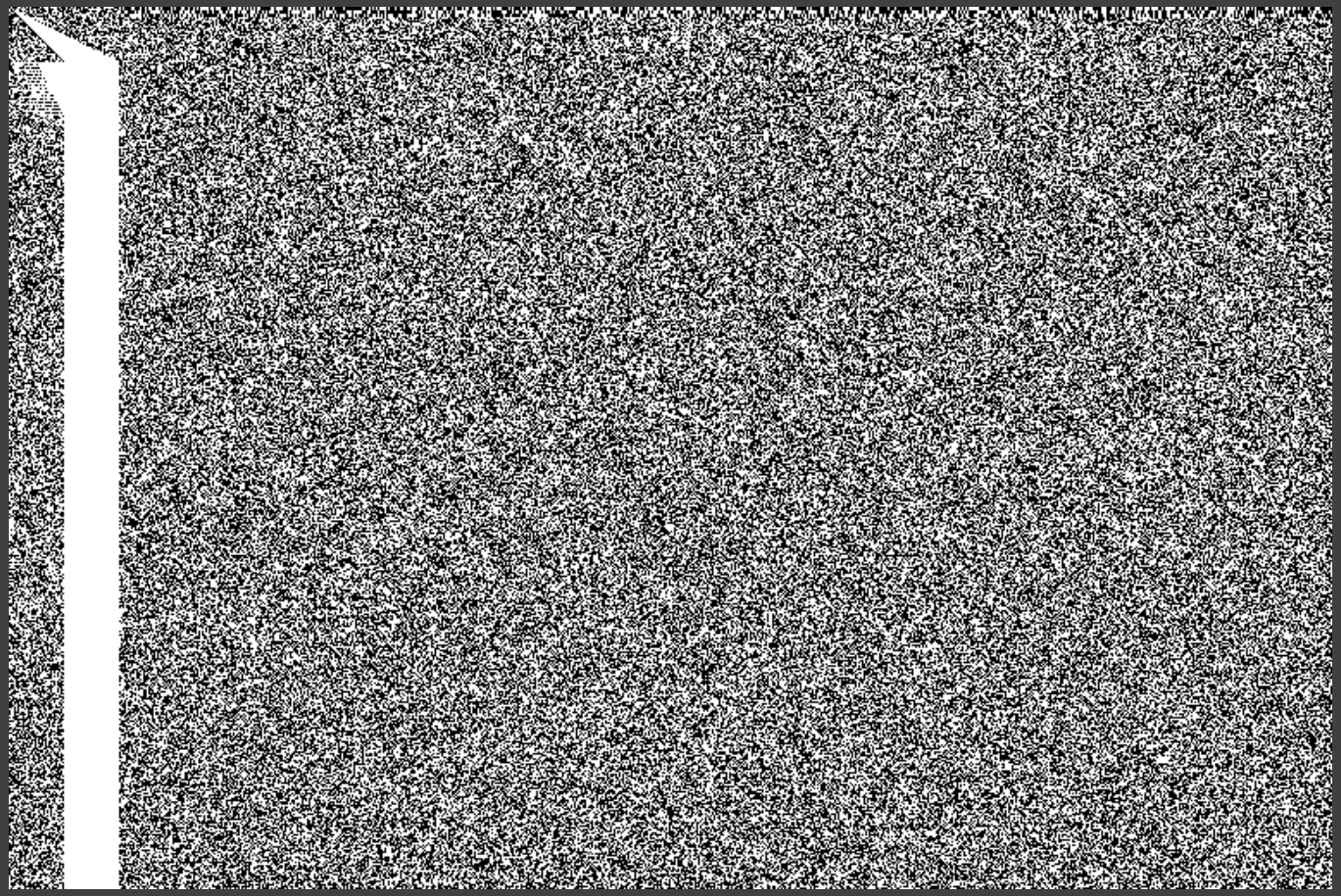
Main steps:

1. Call PLS on A_W
2. Apply row permutation to A_E
3. $L_{NW} \leftarrow$ the lower left triangular matrix in A_{NW}
4. $A_{NE} \leftarrow L_{NW}^{-1} \times A_{NE}$
5. $A_{SE} \leftarrow A_{SE} + A_{SW} \times A_{NE}$
6. Call PLS on A_{SE}
7. Apply row permutation to A_{SW}
8. Compress L

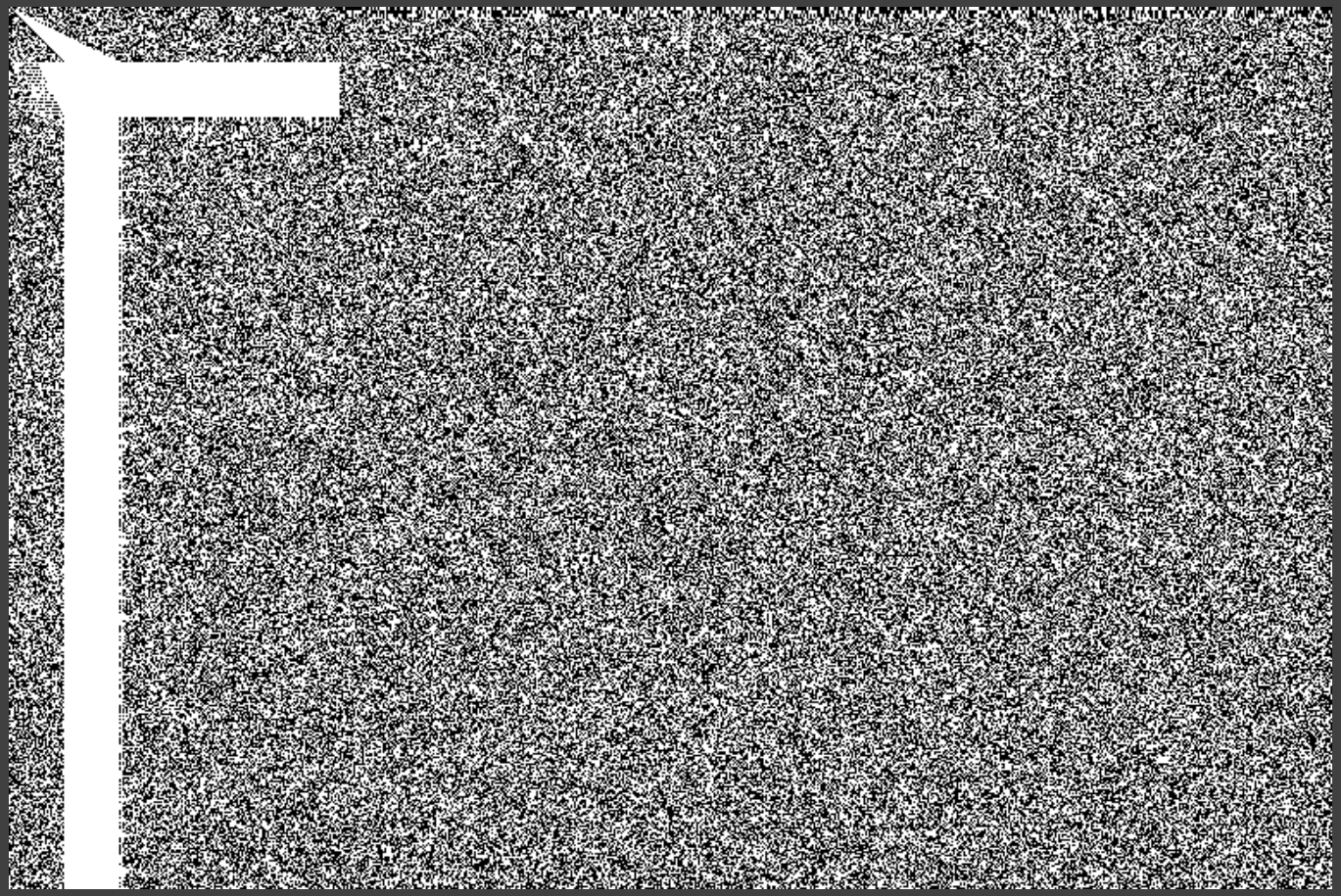
Visualisation



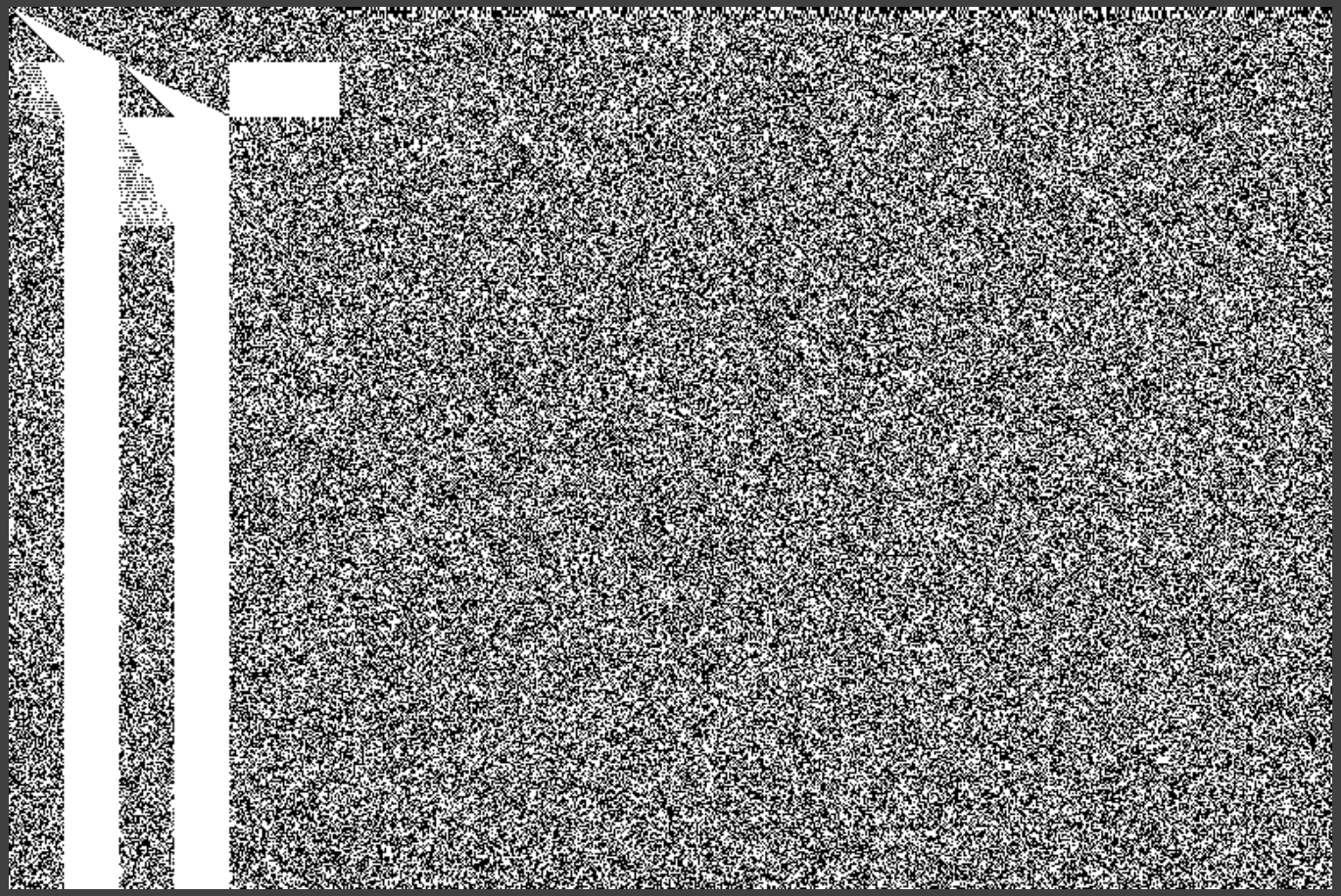
Visualisation



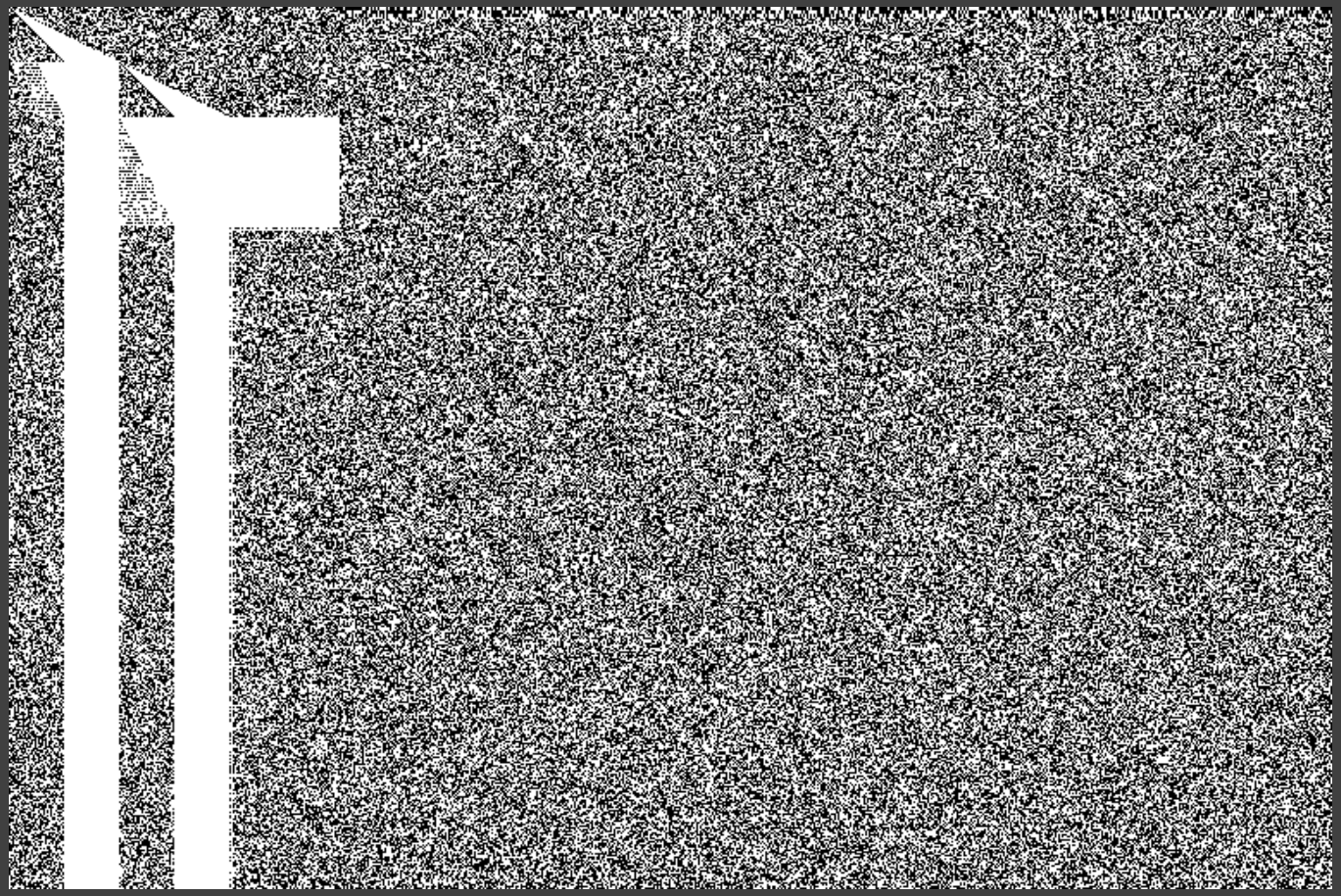
Visualisation



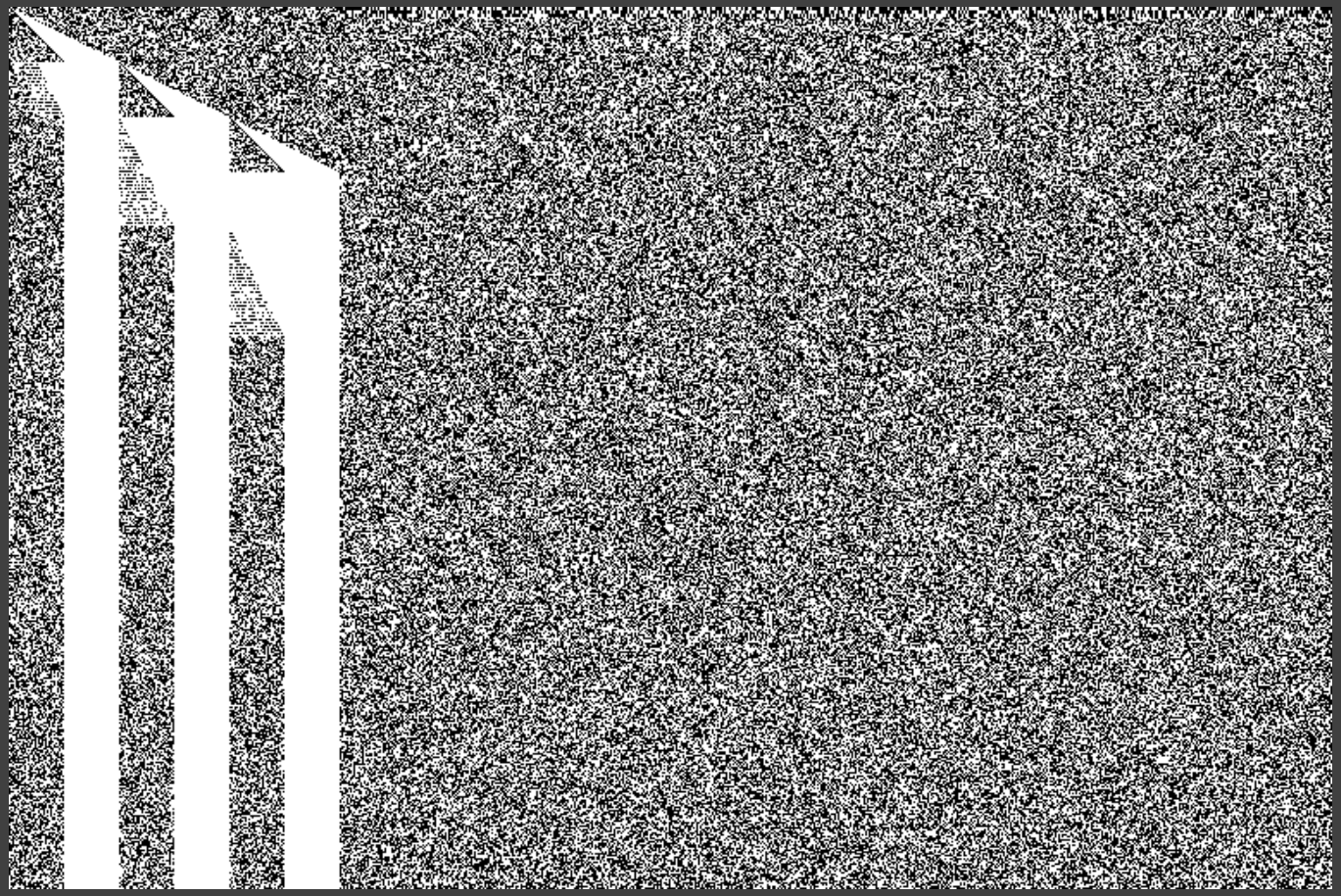
Visualisation



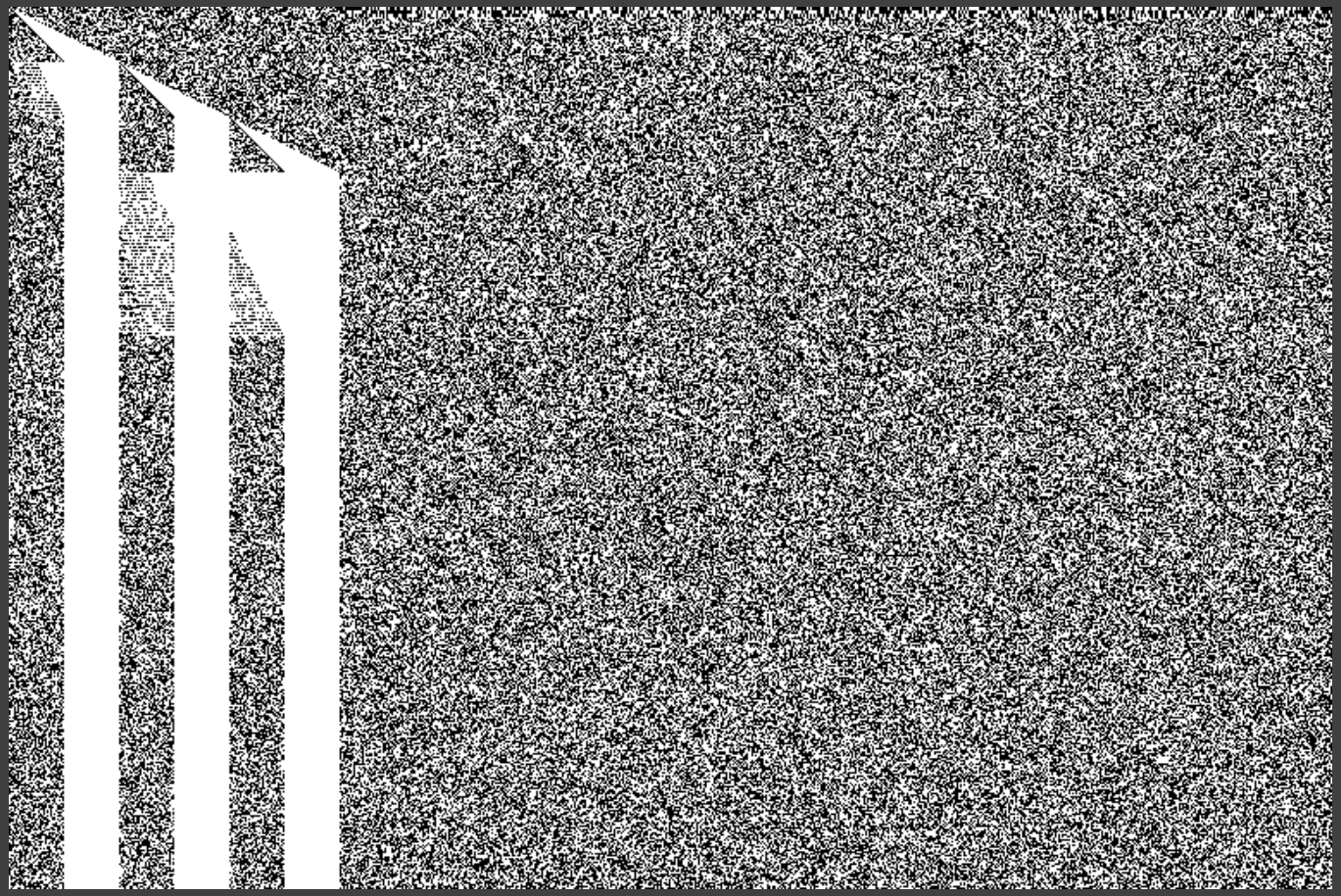
Visualisation



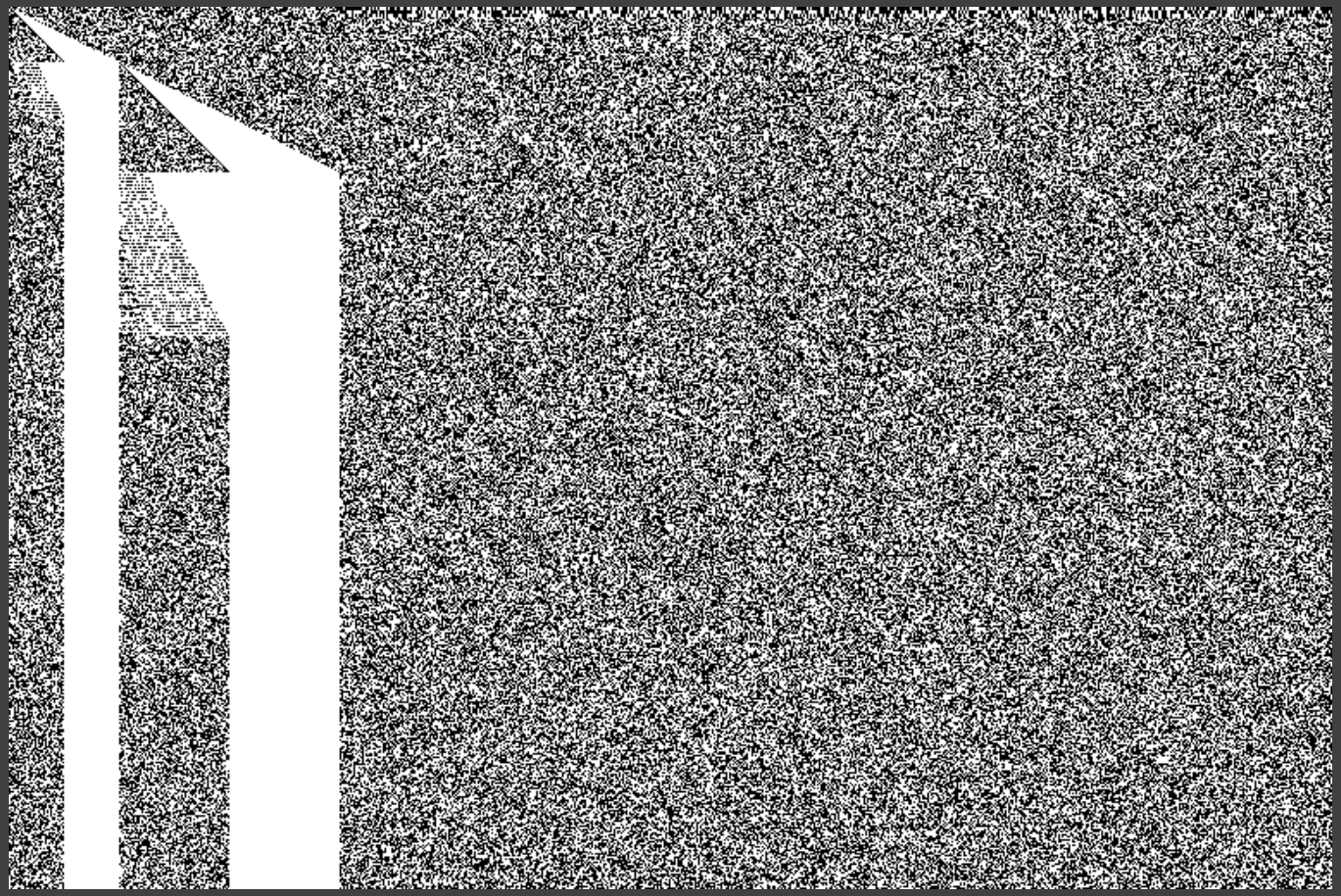
Visualisation



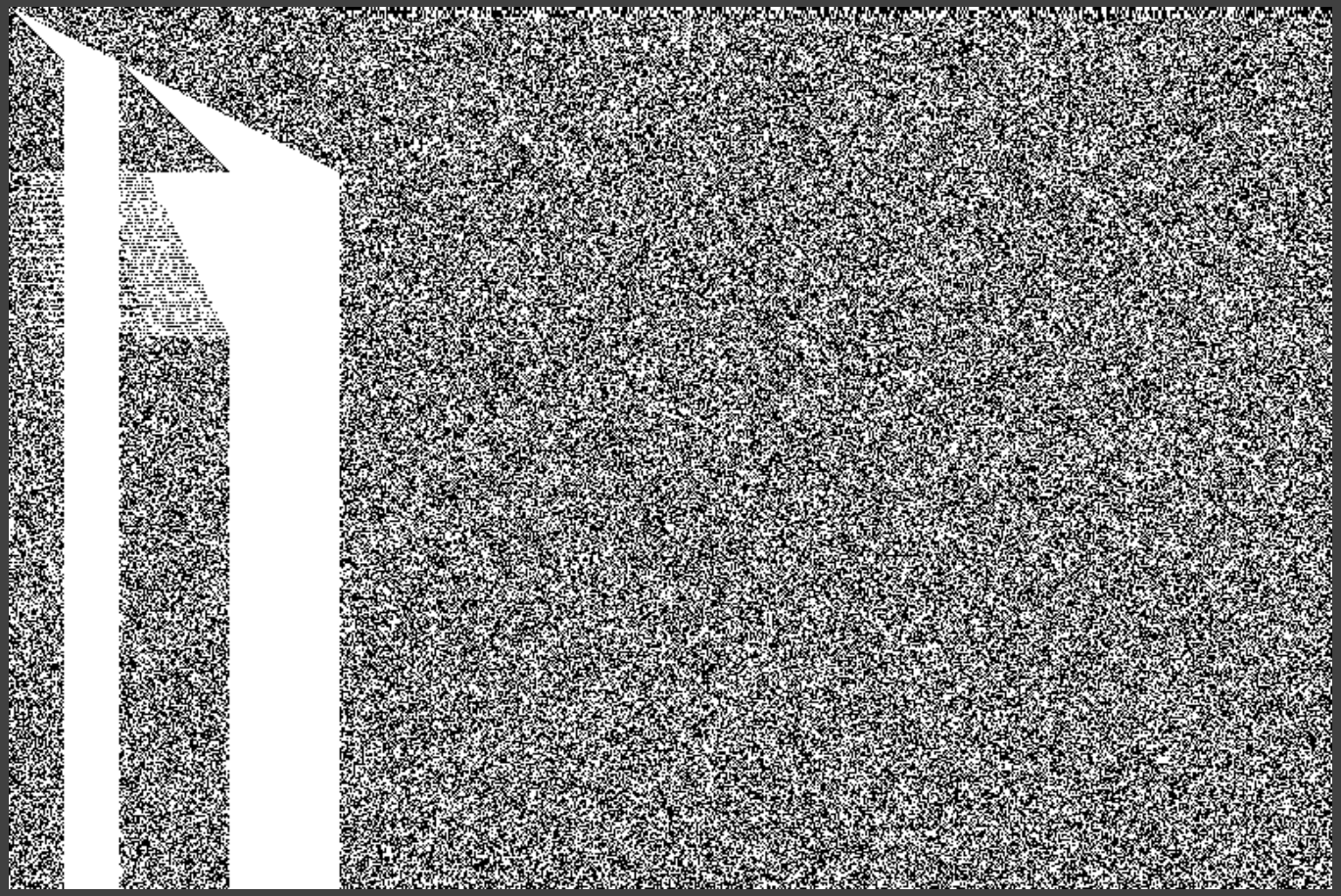
Visualisation



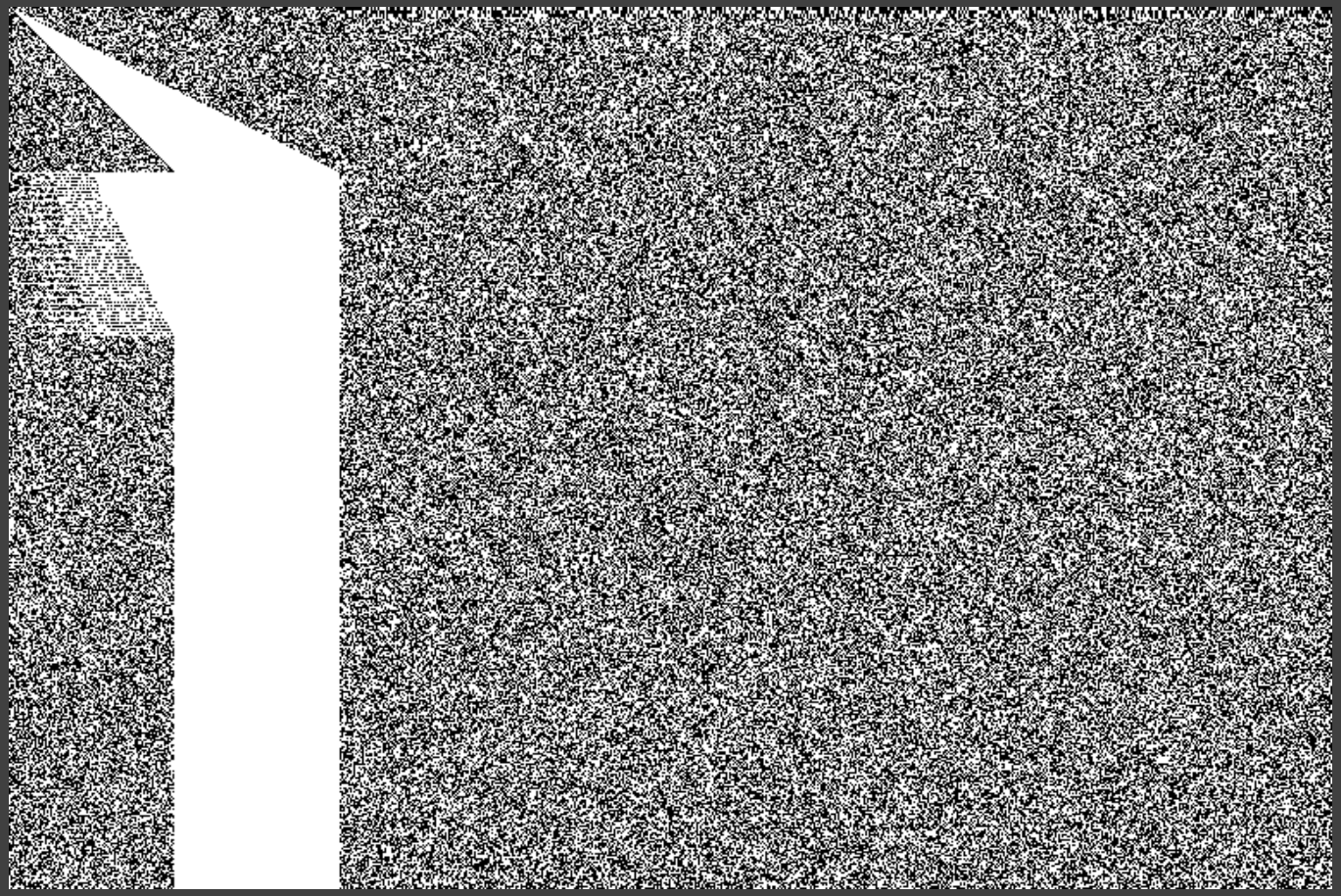
Visualisation



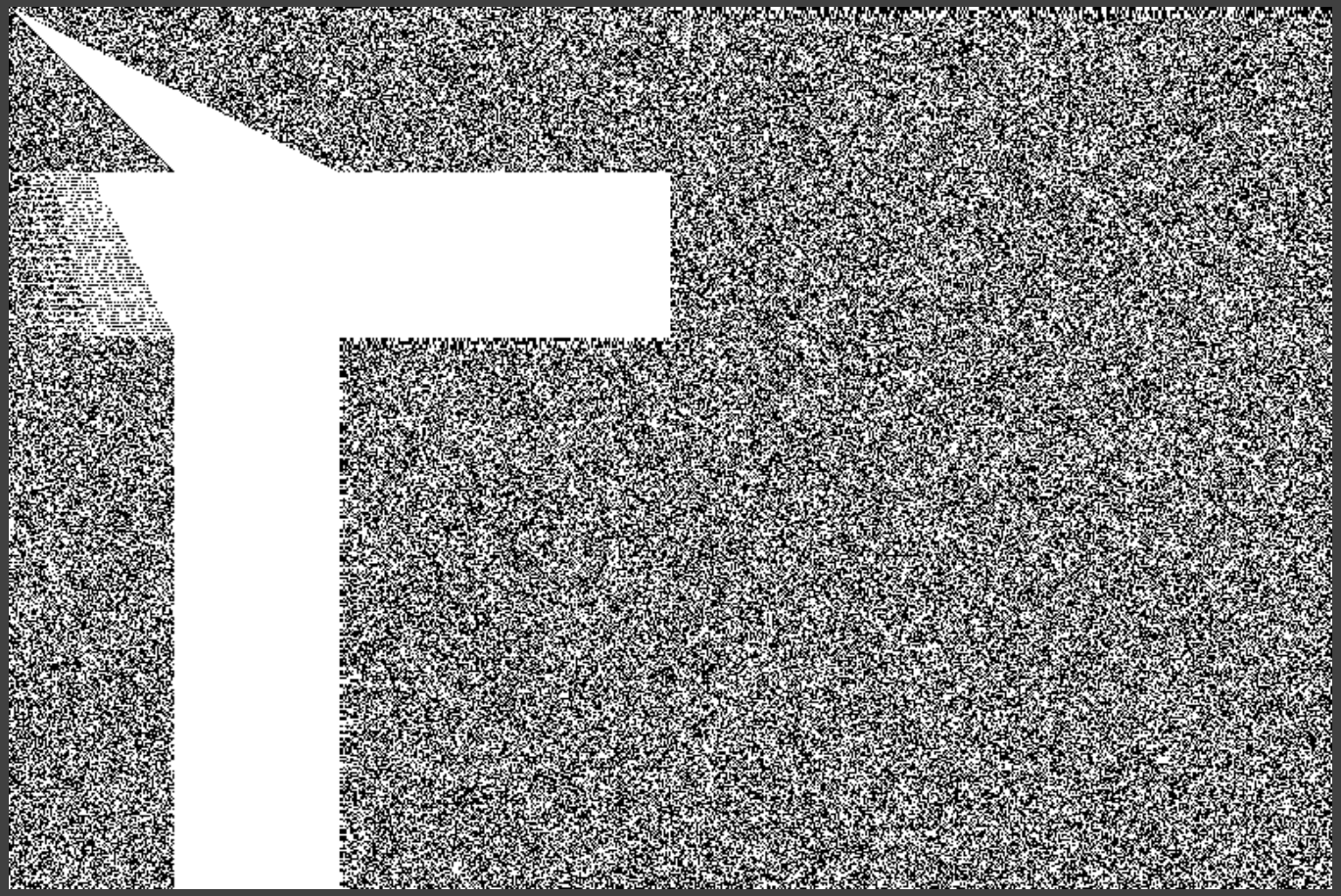
Visualisation



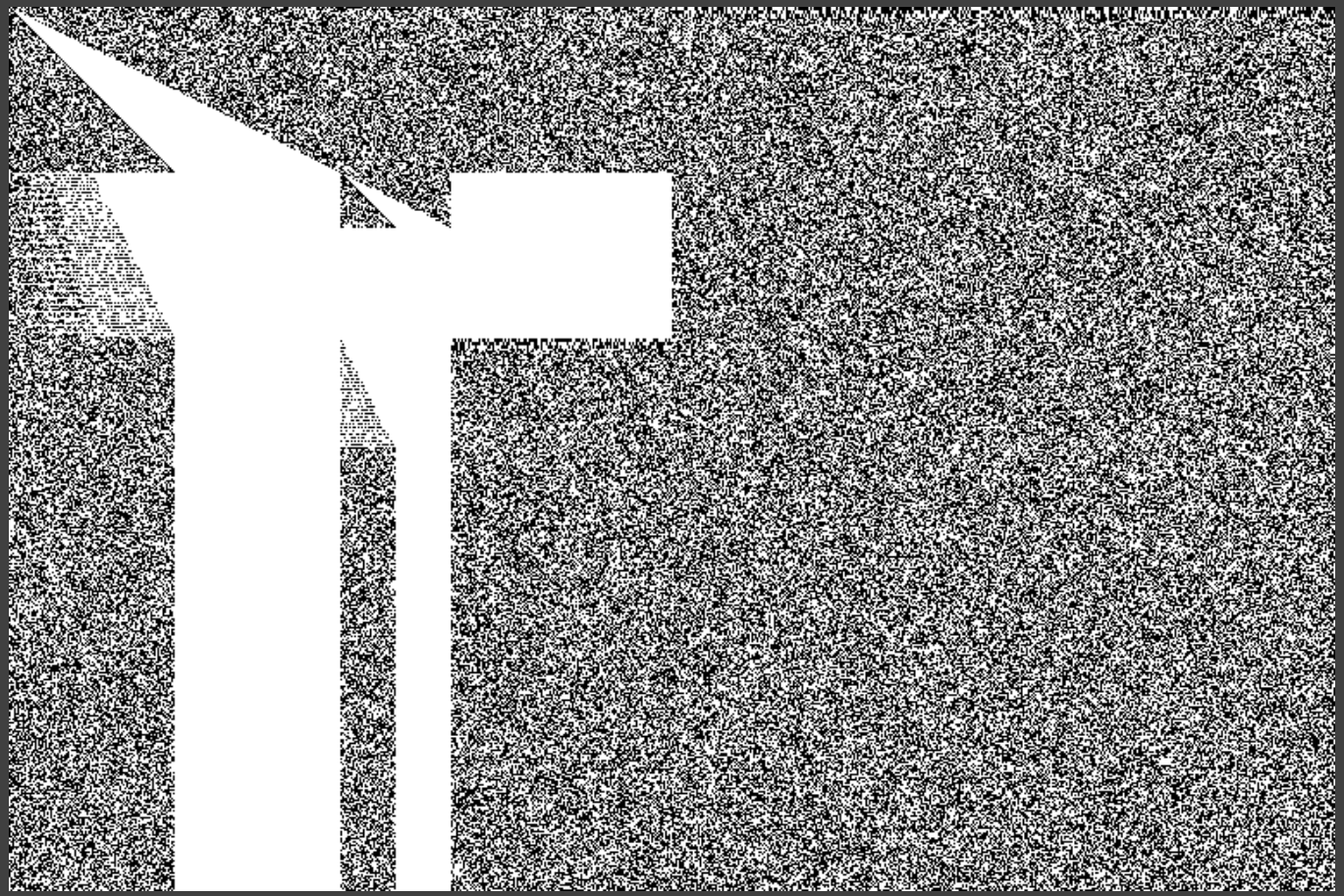
Visualisation



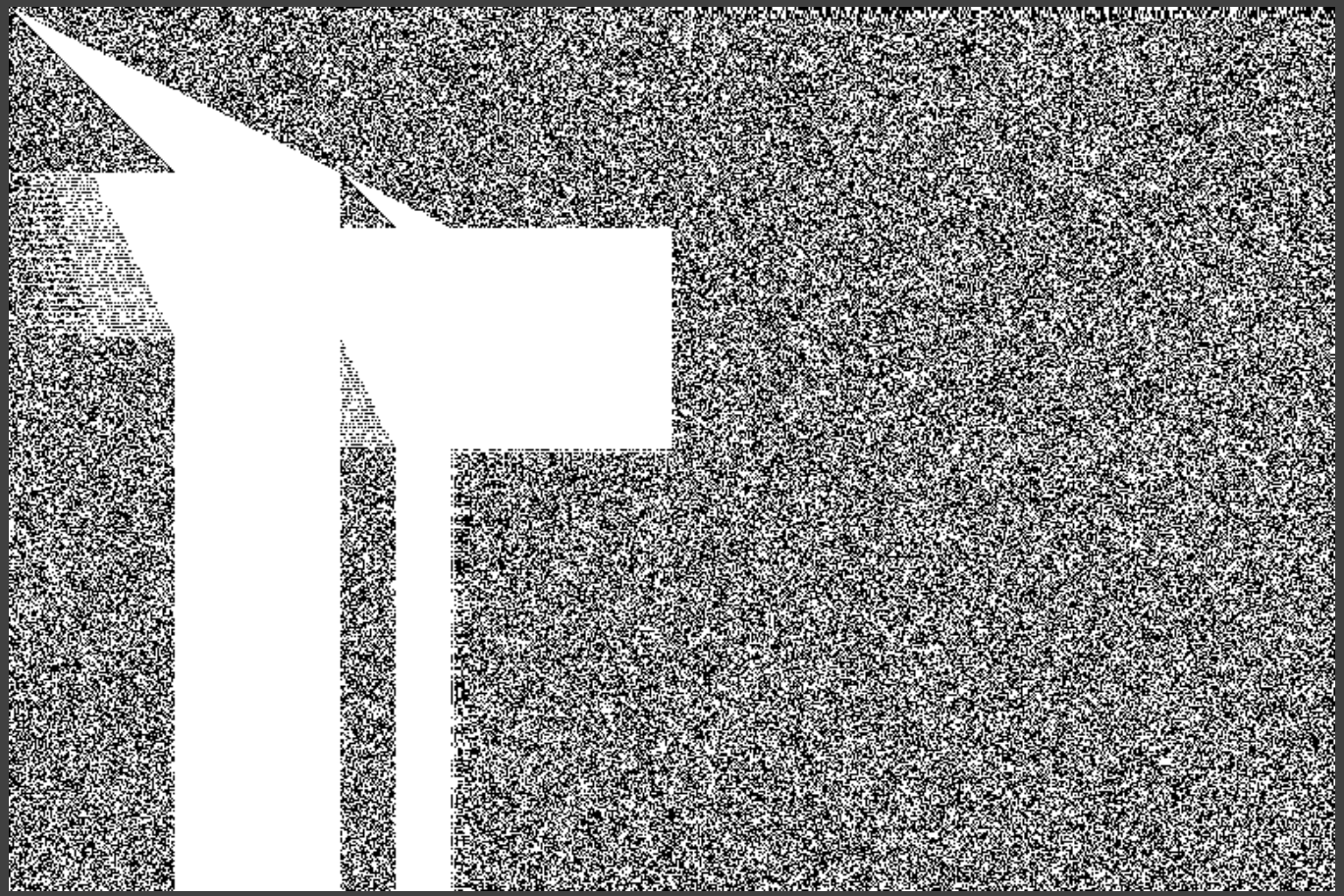
Visualisation



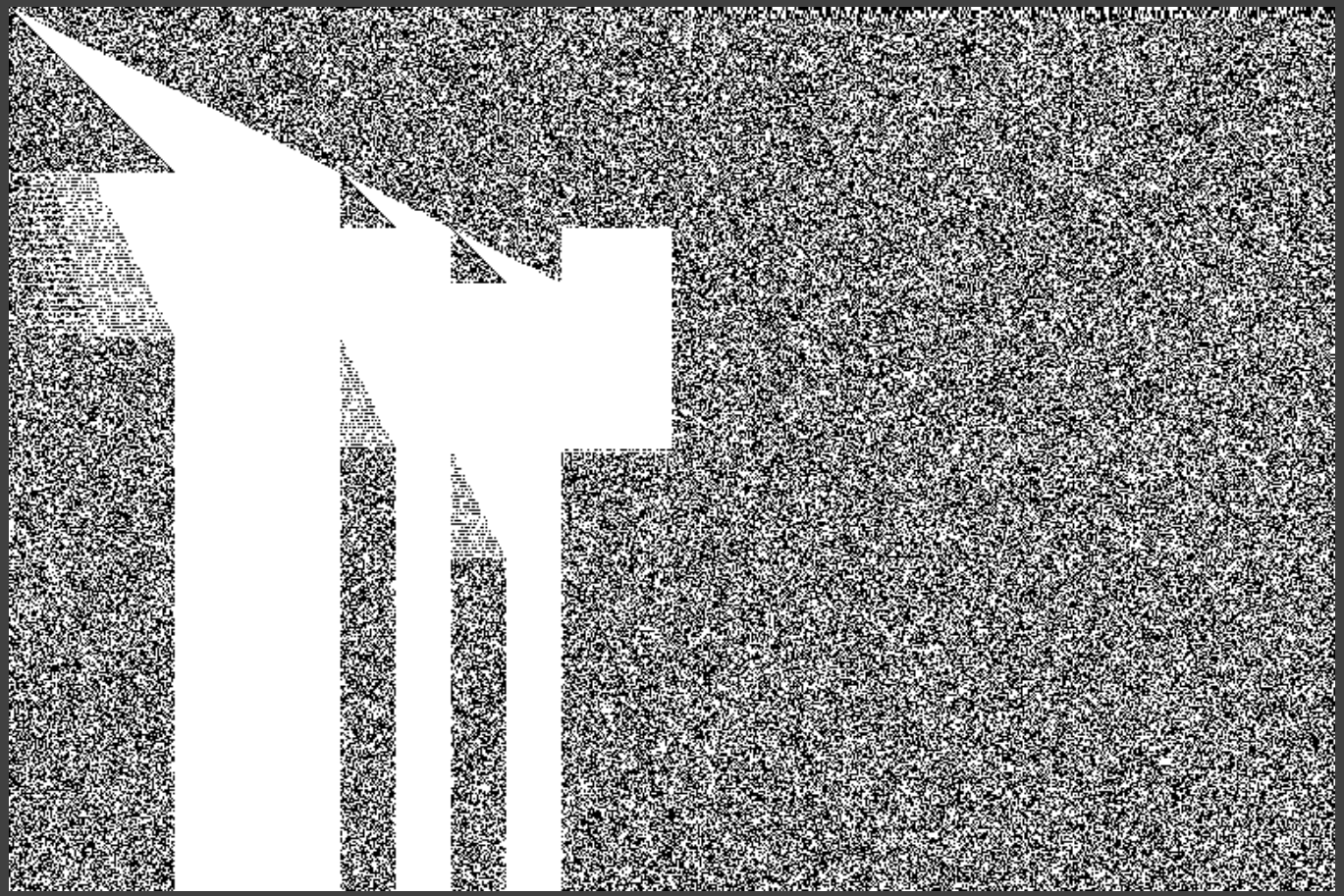
Visualisation



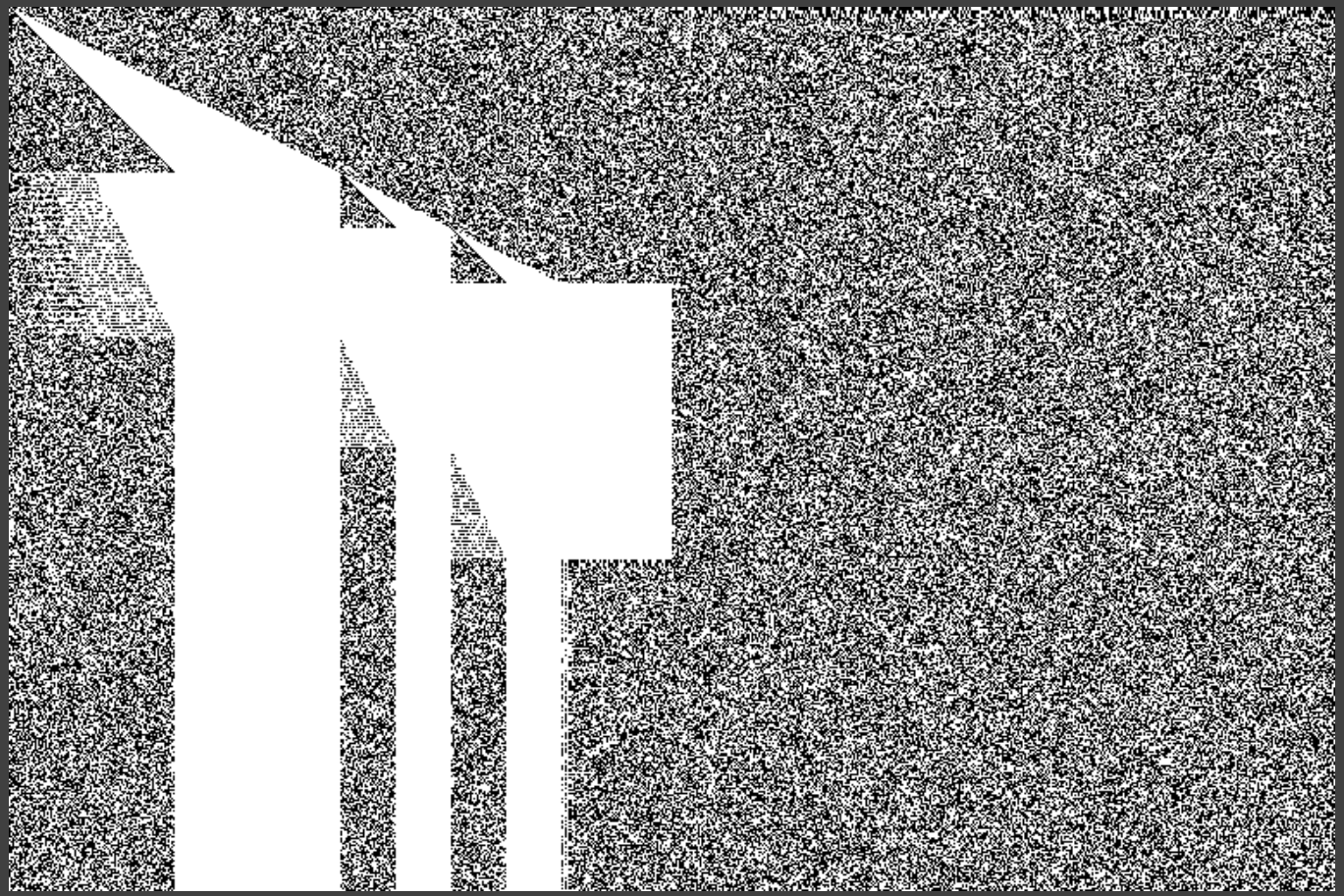
Visualisation



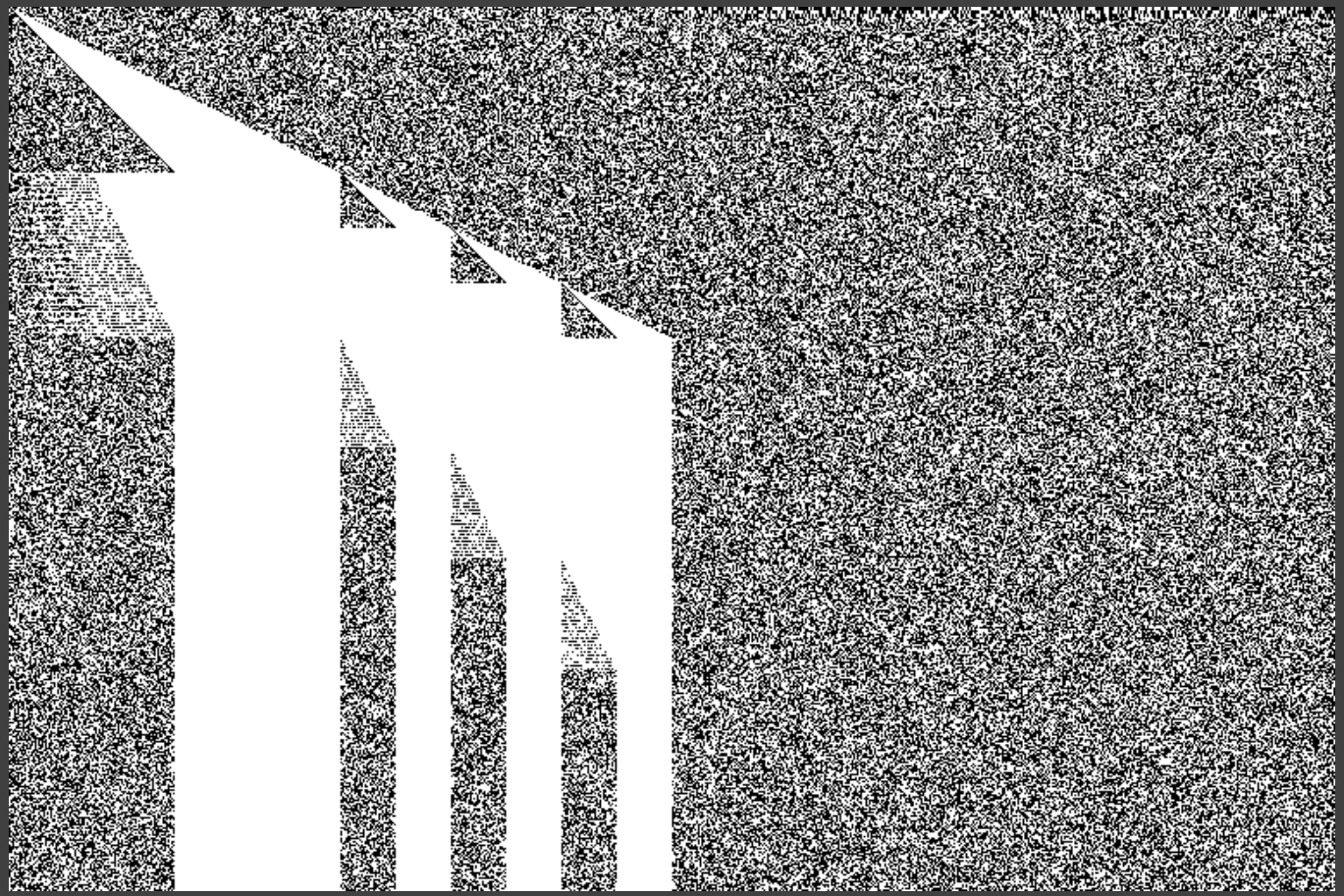
Visualisation



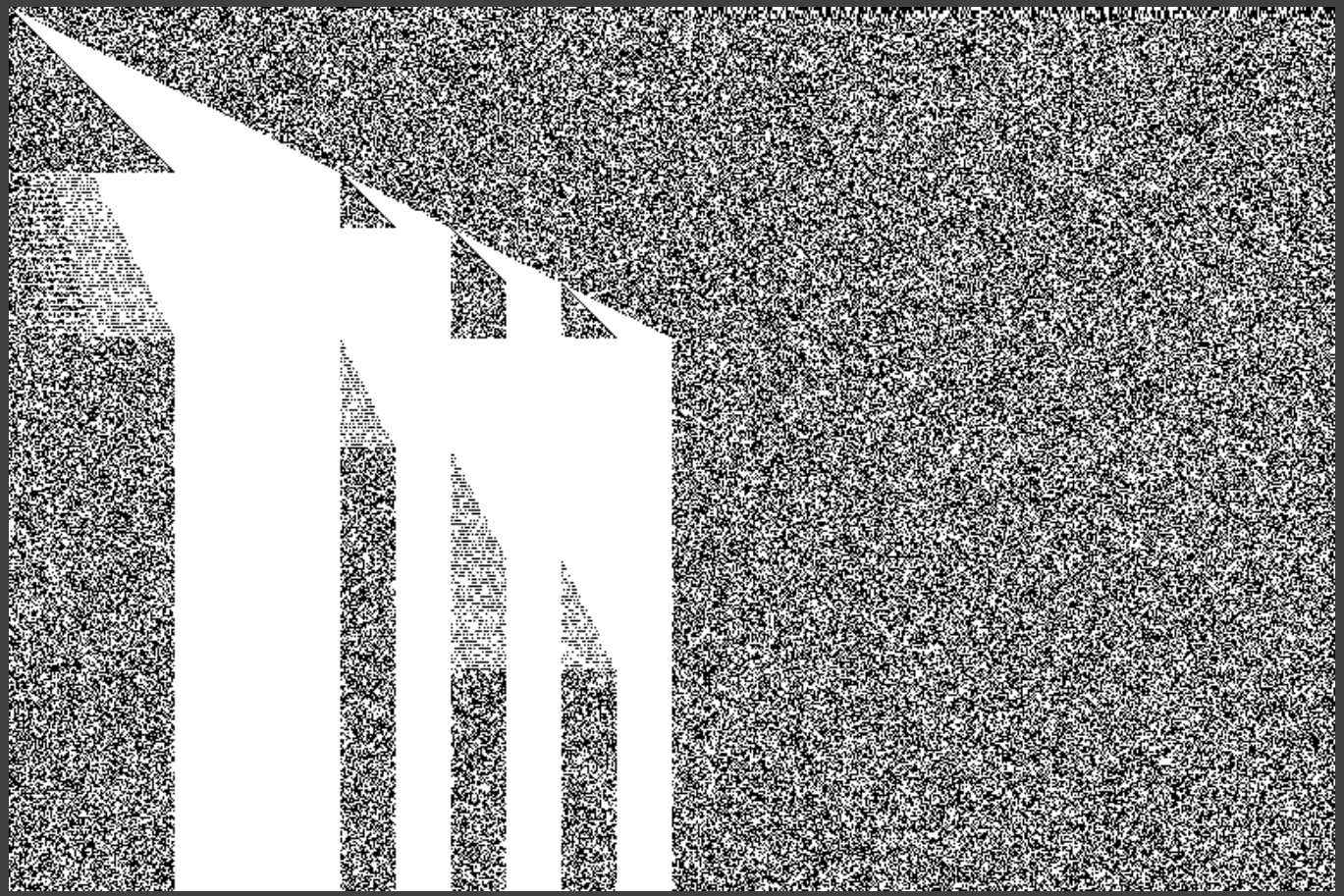
Visualisation



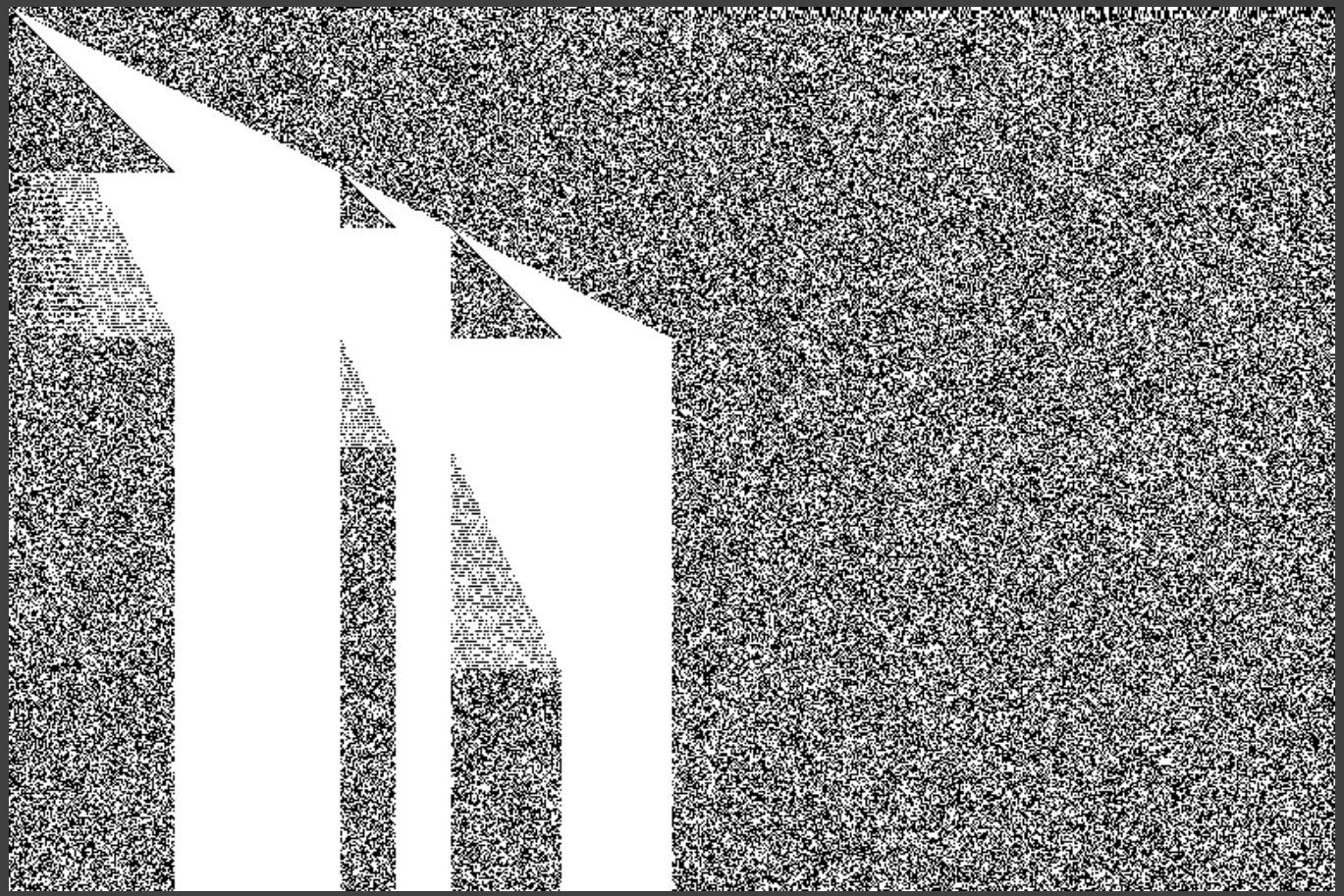
Visualisation



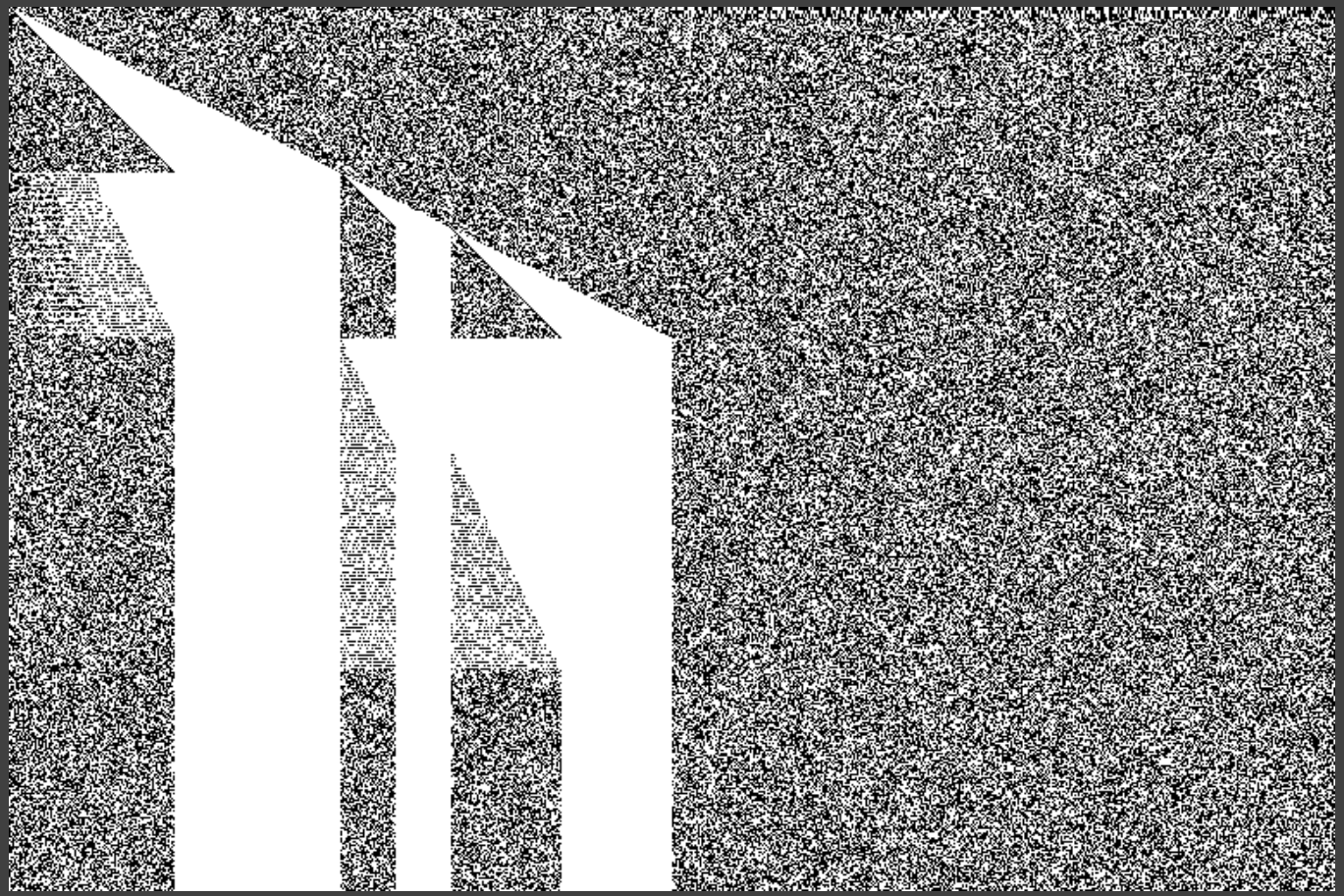
Visualisation



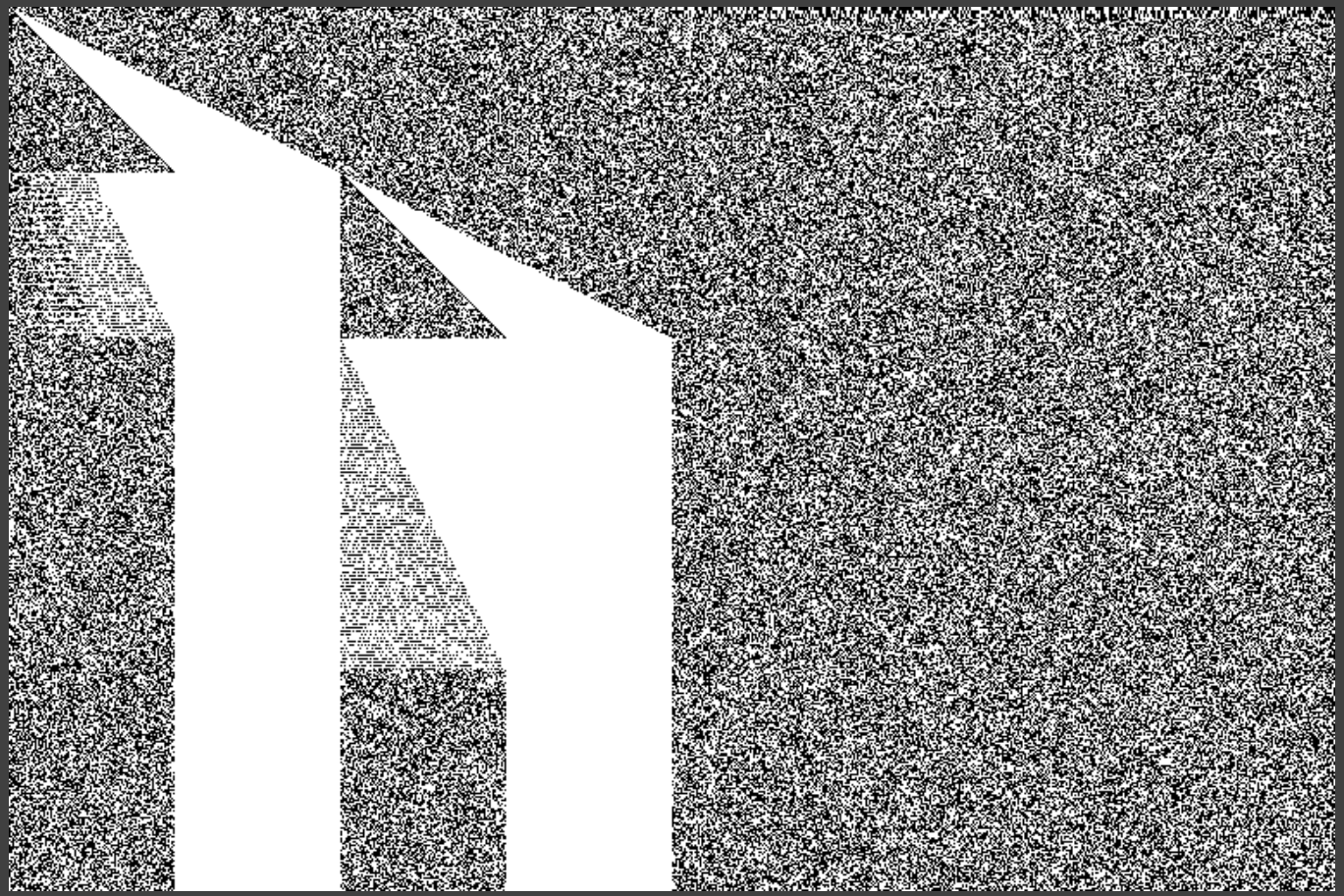
Visualisation



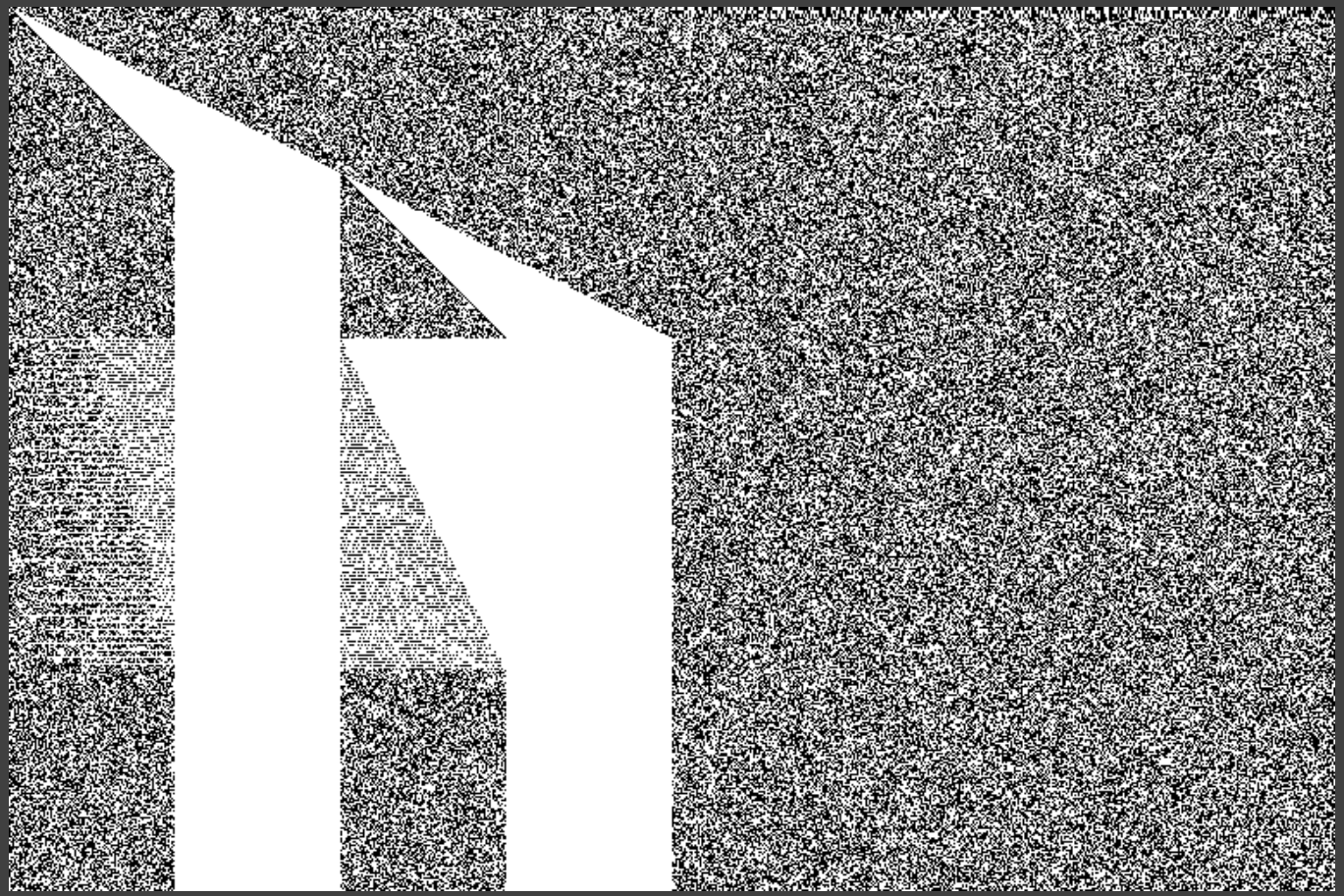
Visualisation



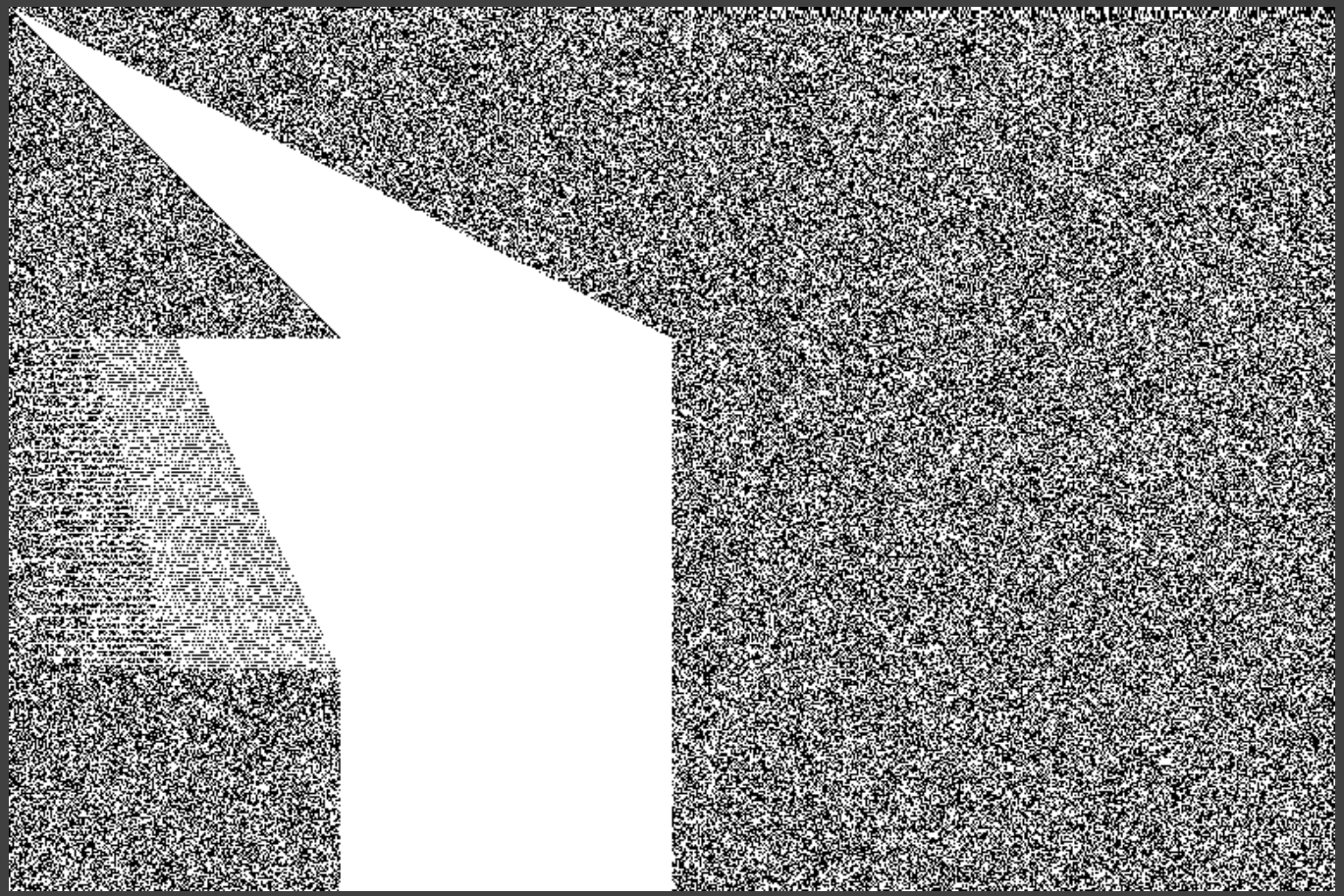
Visualisation



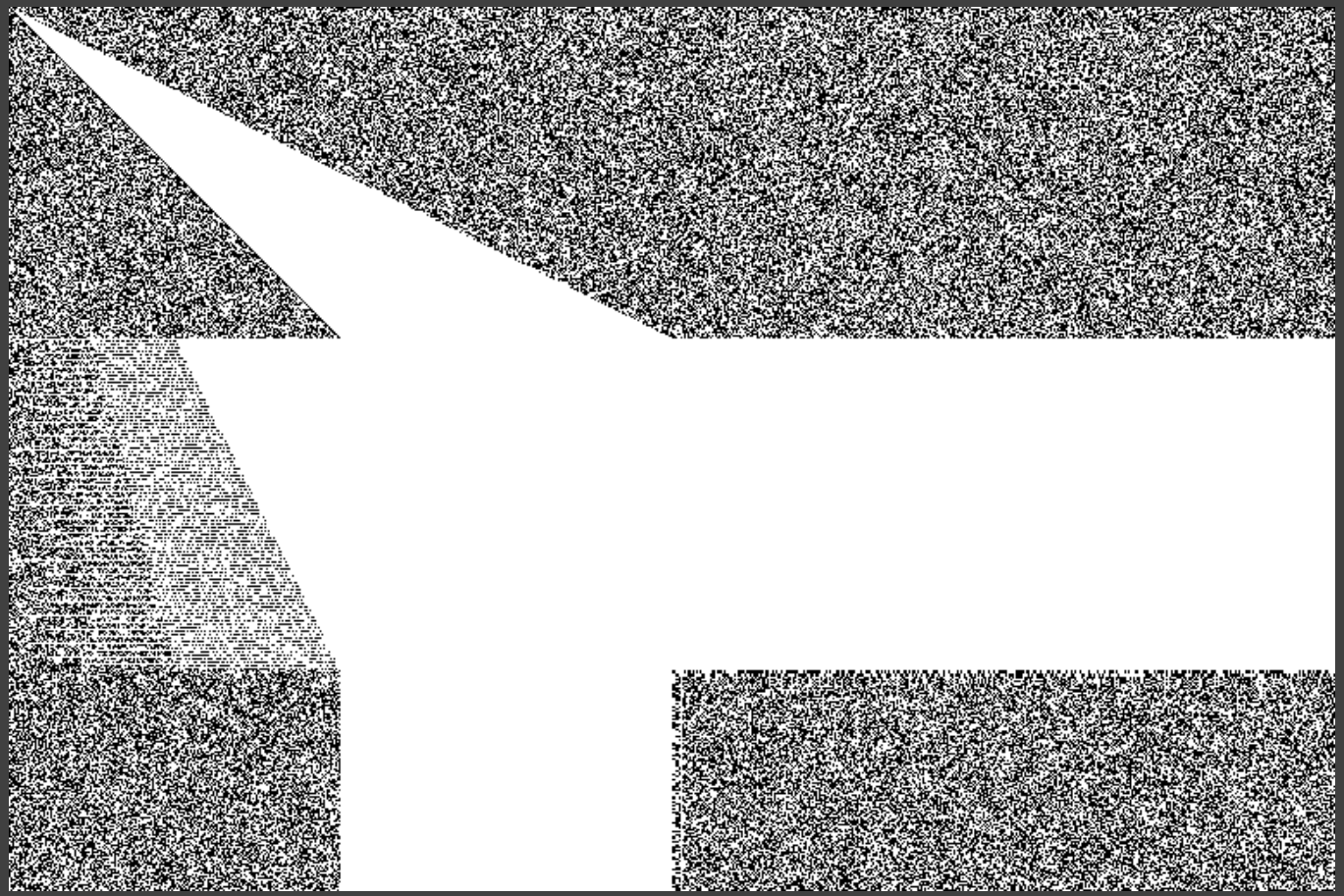
Visualisation



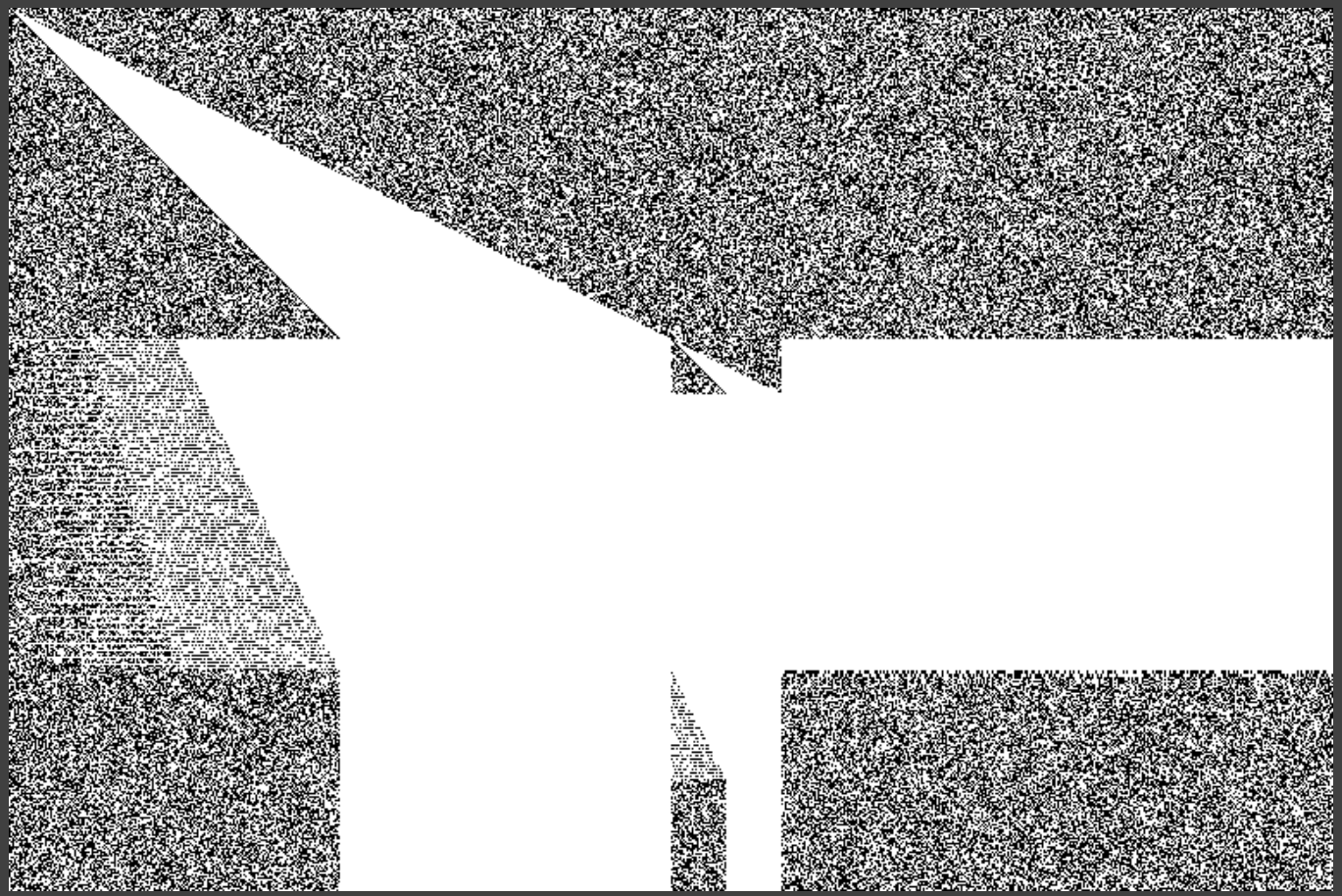
Visualisation



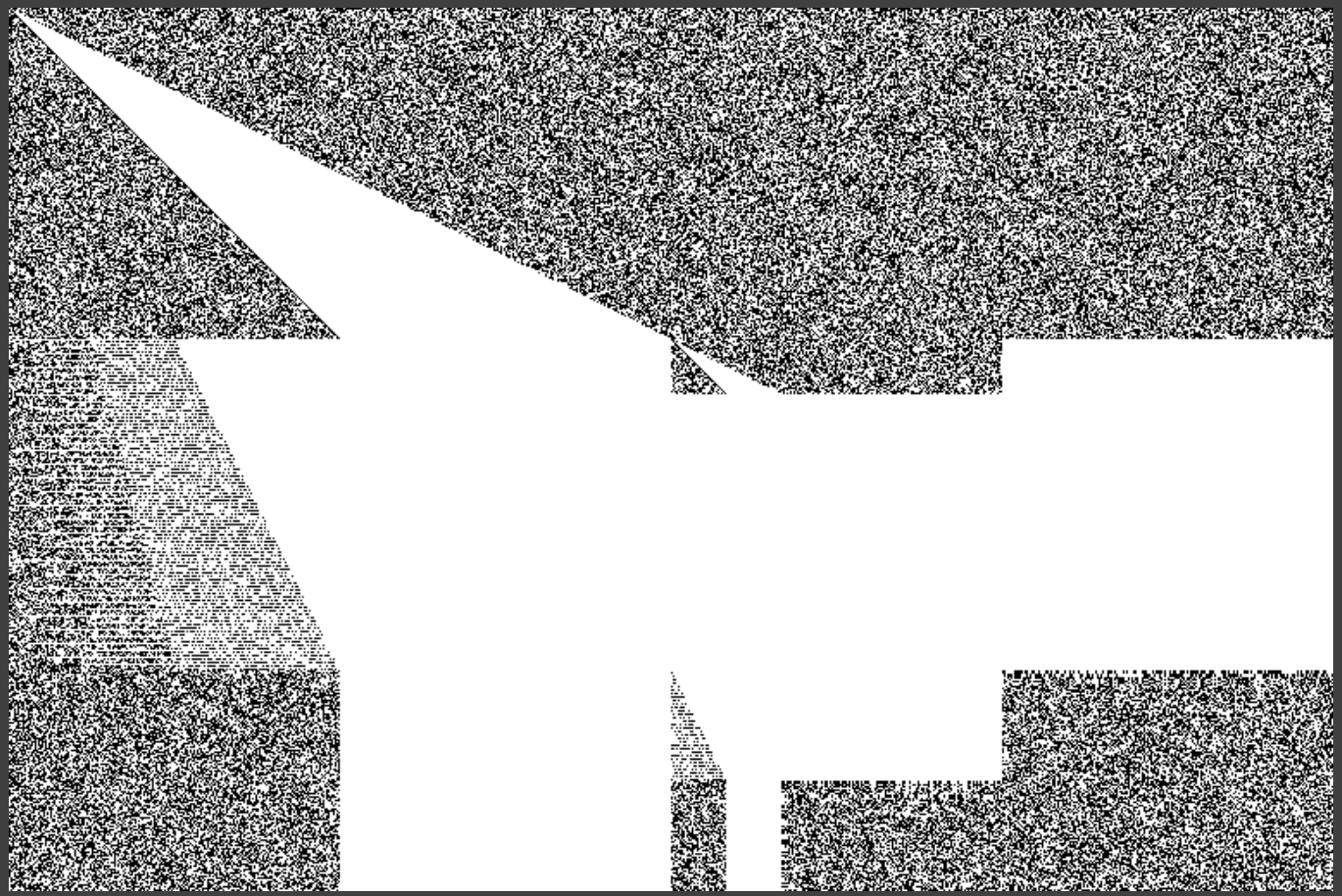
Visualisation



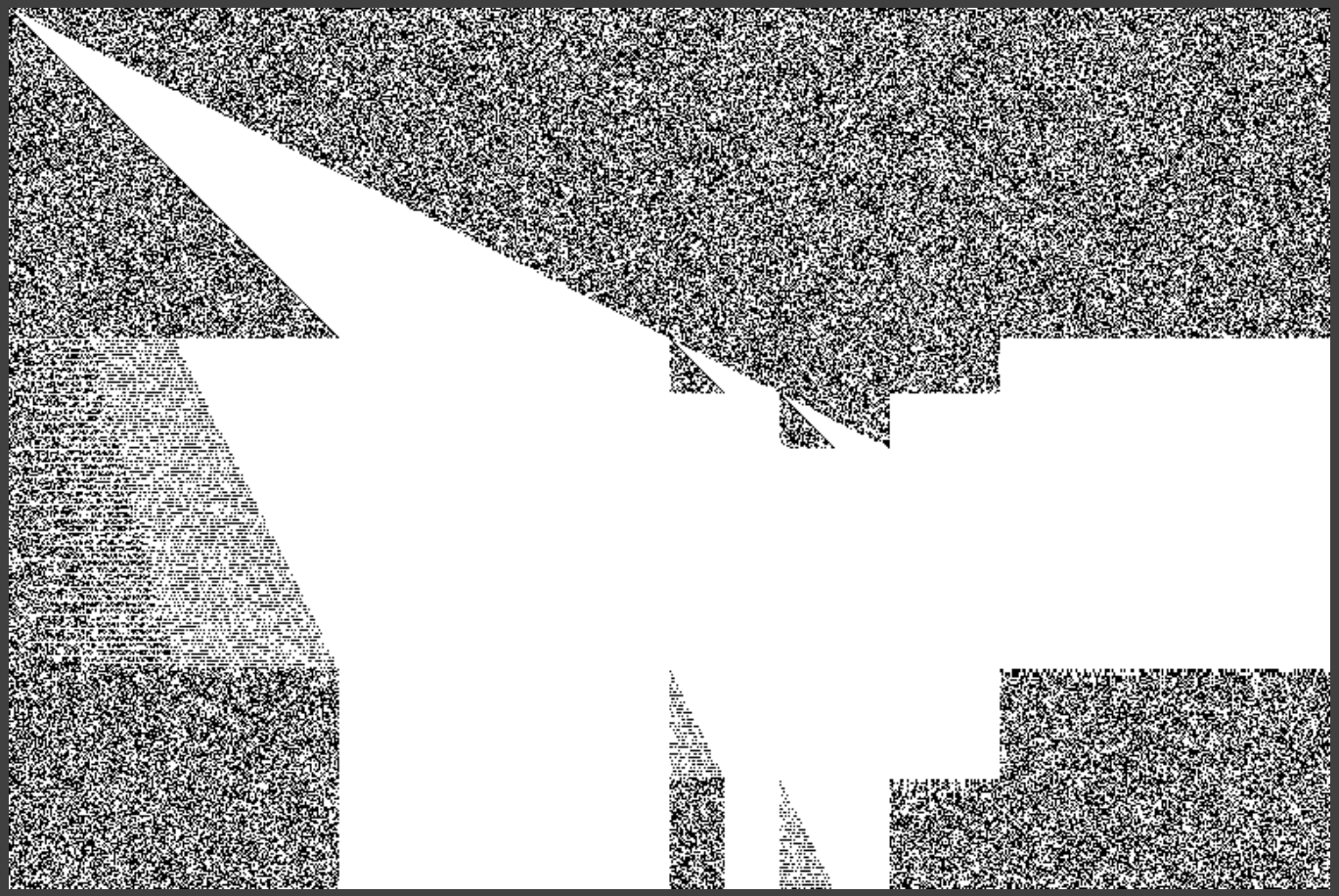
Visualisation



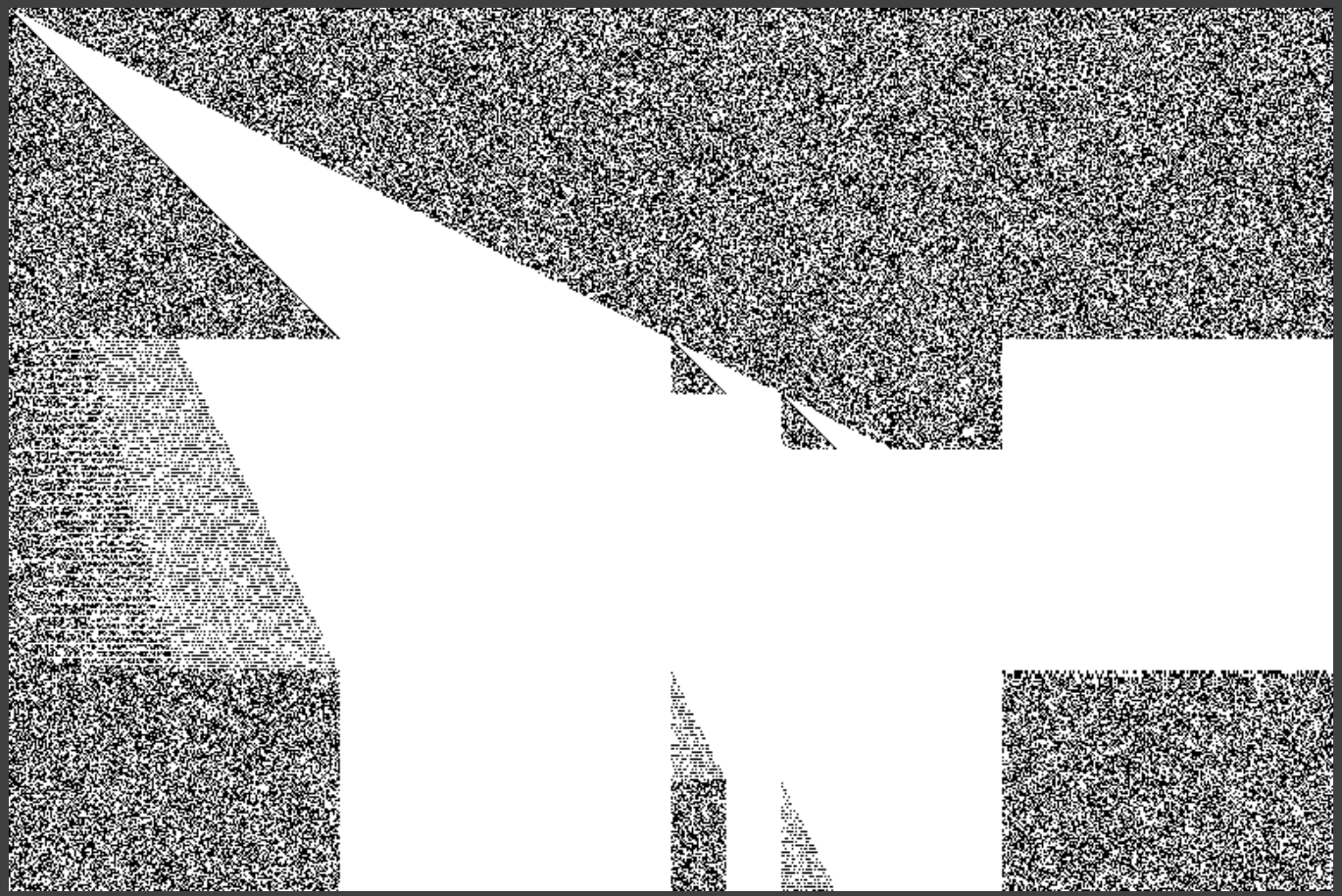
Visualisation



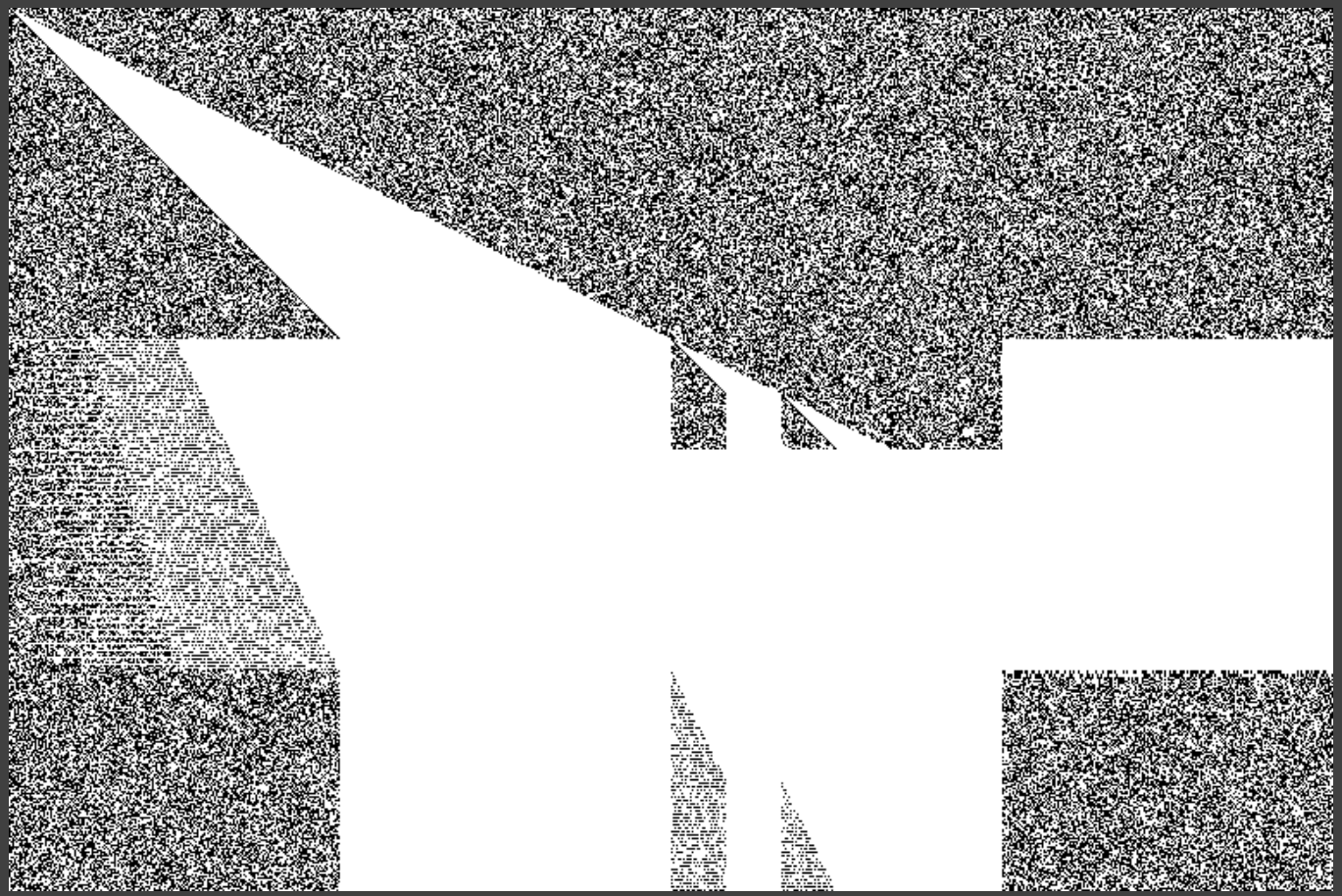
Visualisation



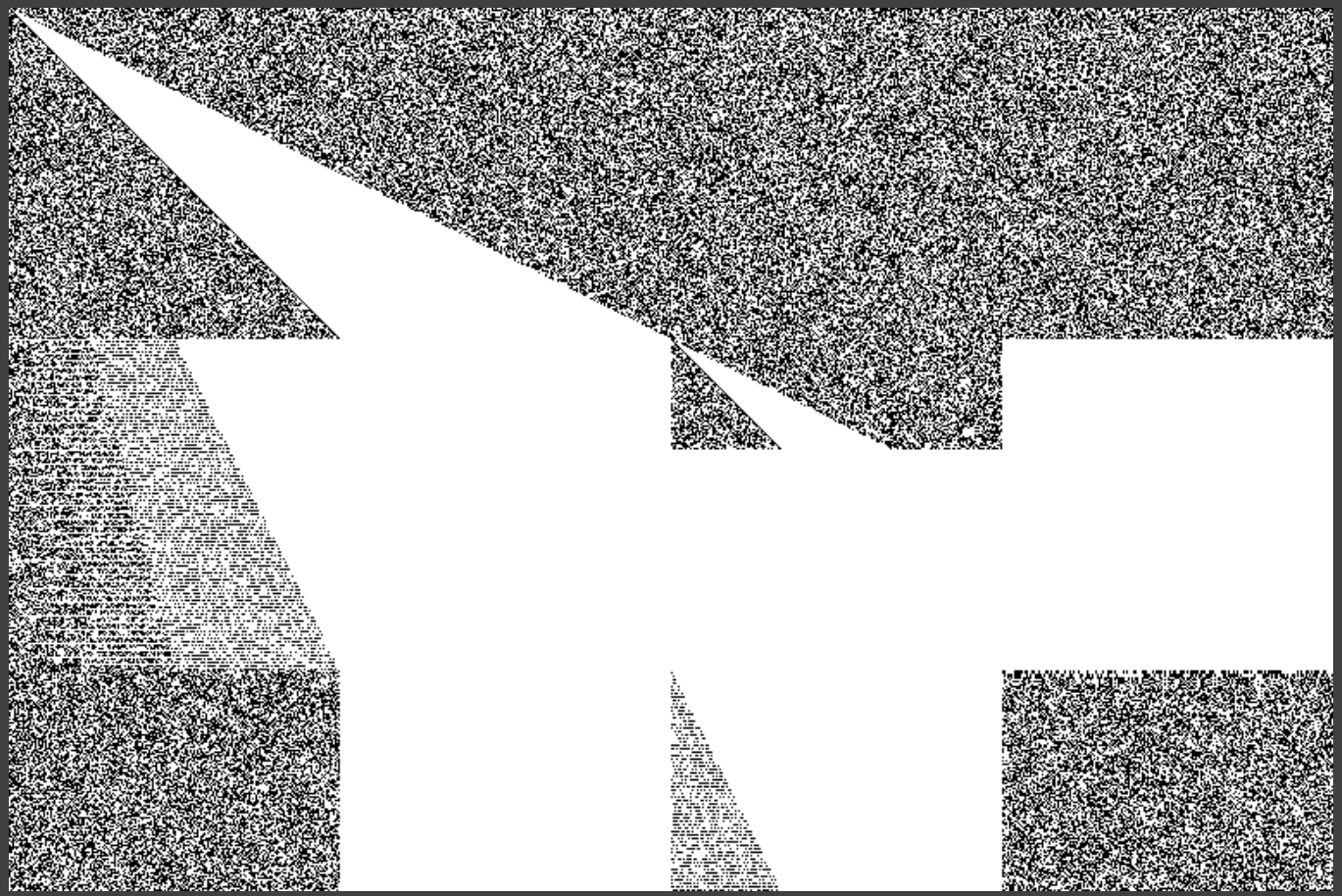
Visualisation



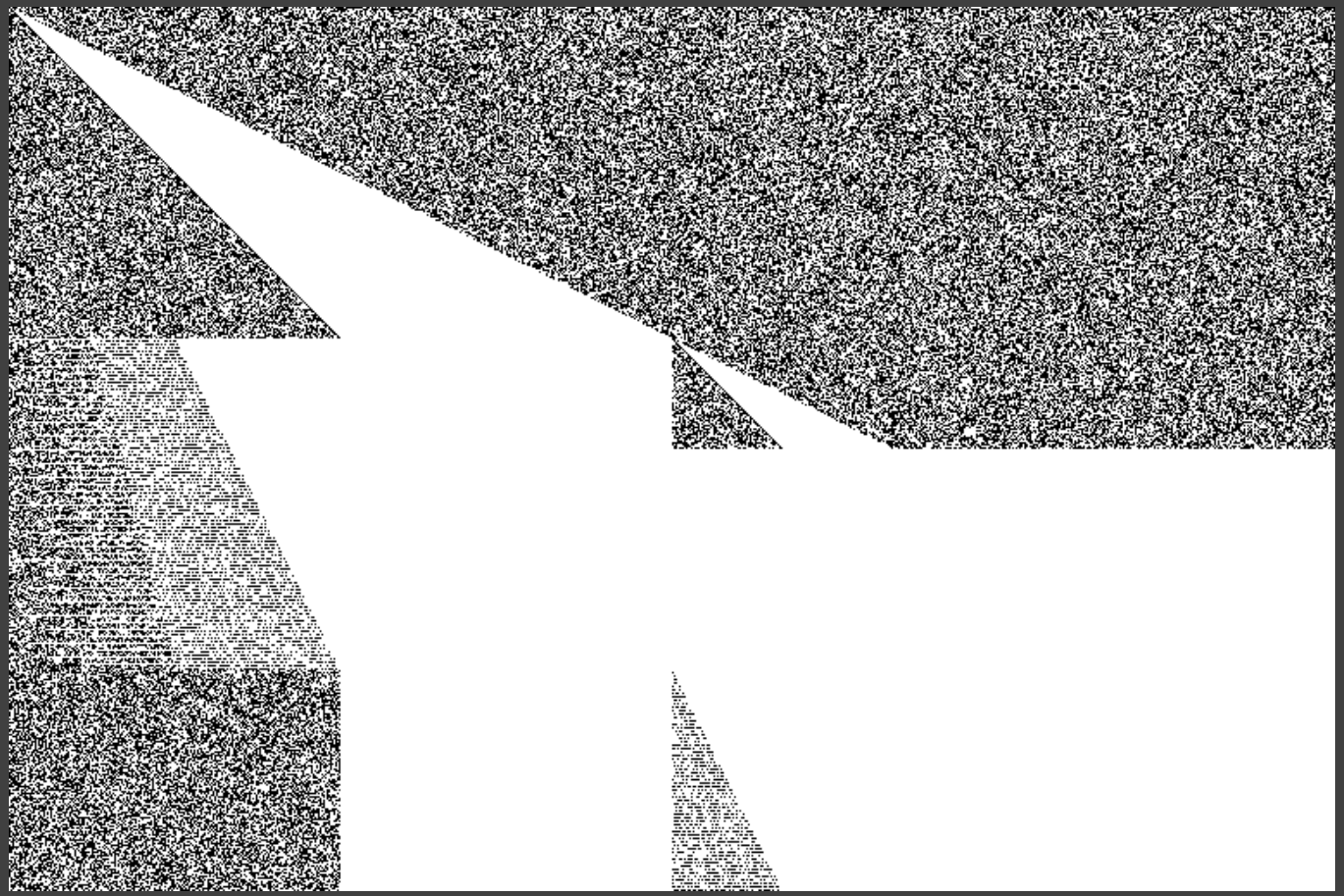
Visualisation



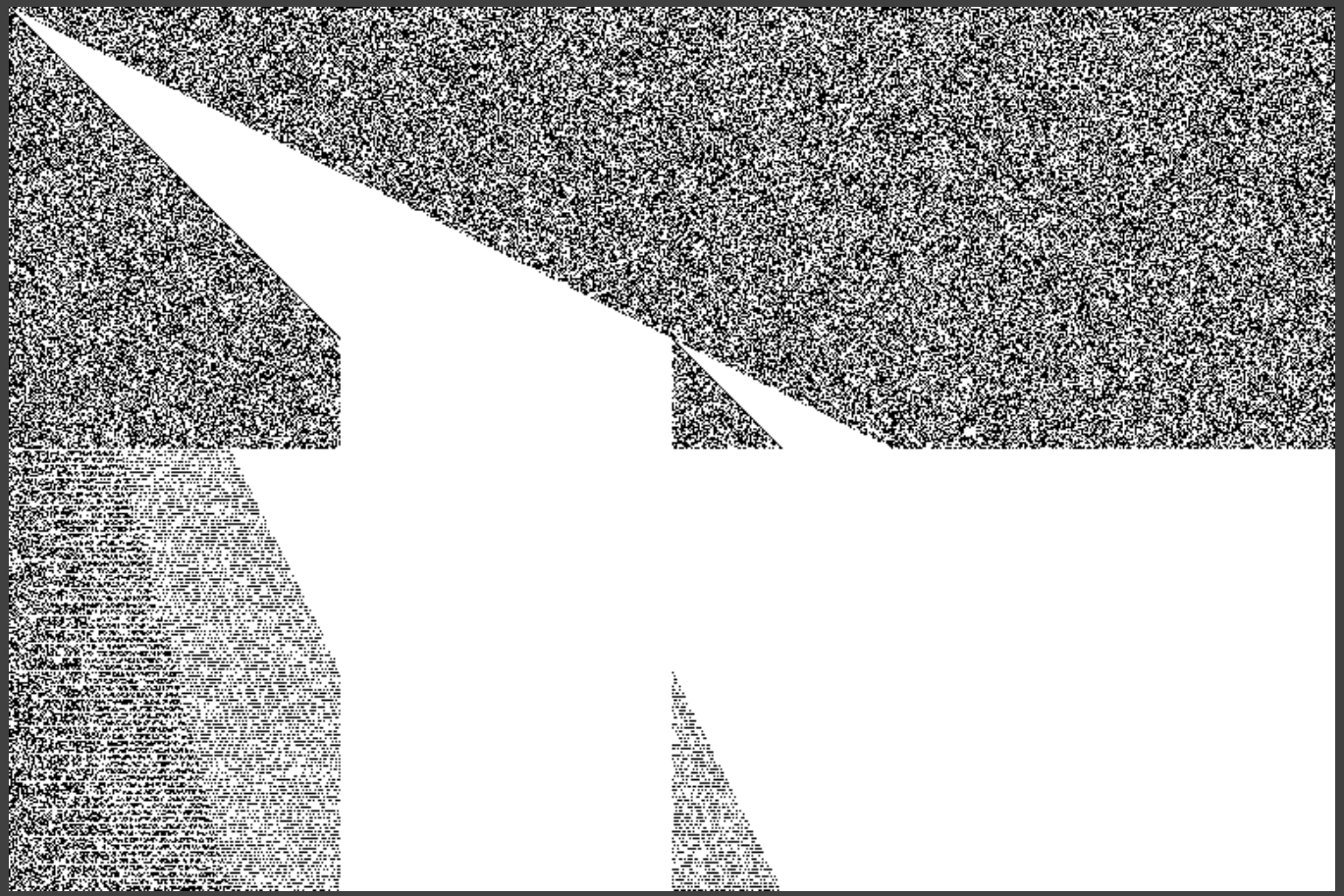
Visualisation



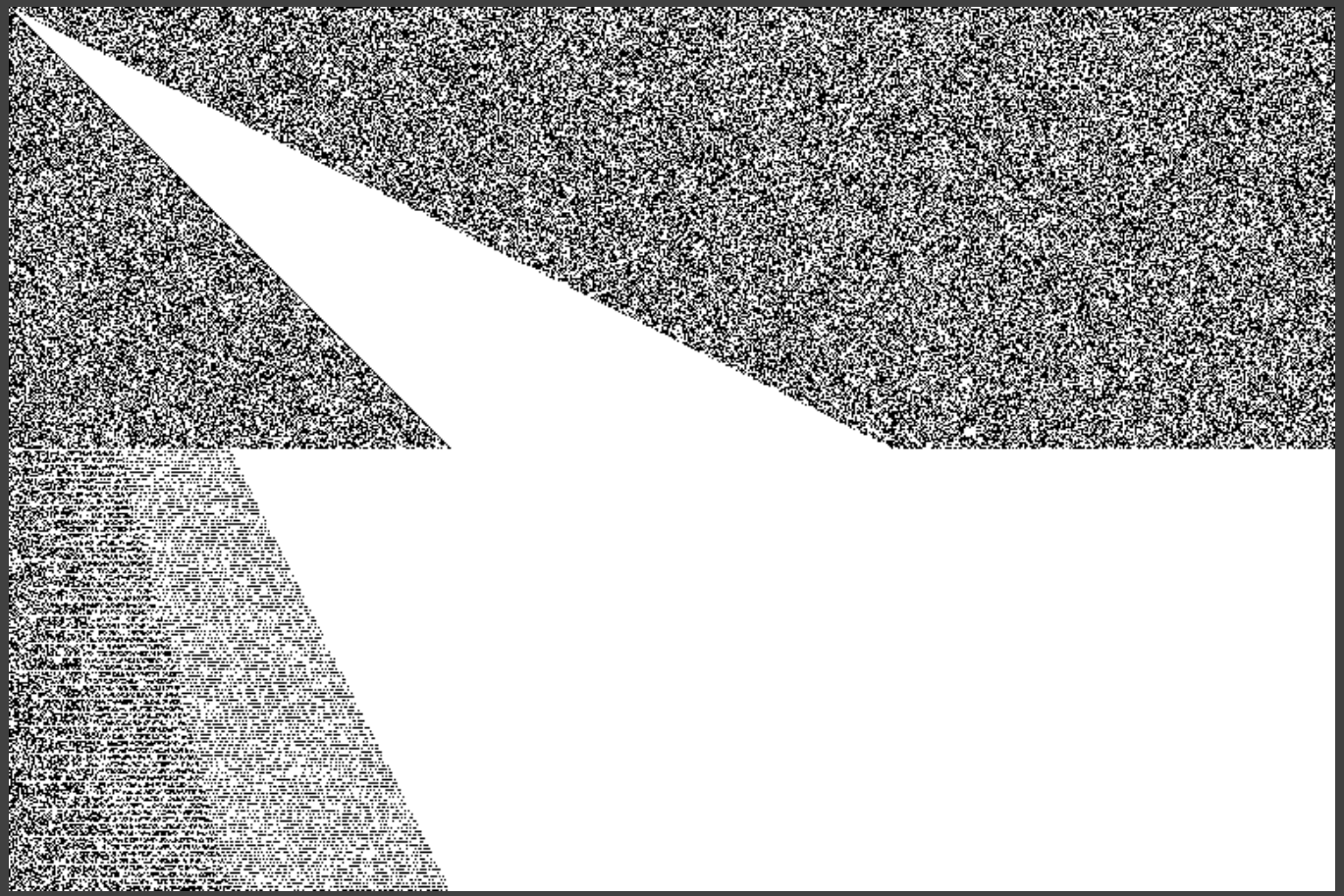
Visualisation



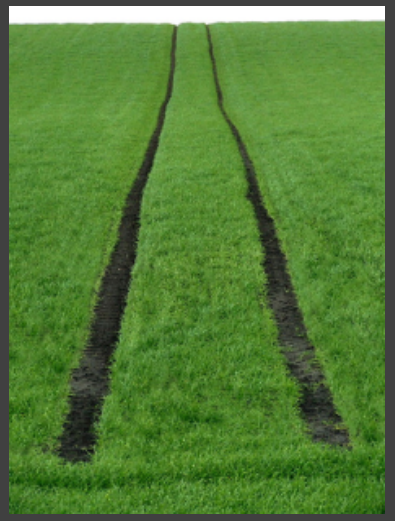
Visualisation



Visualisation



Outline



Why?

Asymptotically fast PLS decomposition is only efficient in practice if we switch over to a non-recursive algorithm at some dimension.

Thus, we need an efficient base-case implementation.

M4RI I

$$A = \left(\begin{array}{ccc|cccc}
1 & 0 & 0 & 1 & 0 & 1 & \dots \\
0 & 1 & 0 & 1 & 1 & 1 & \dots \\
0 & 0 & 1 & 0 & 0 & 1 & \dots \\
\dots & & & & & & \\
0 & 0 & 0 & 1 & 1 & 0 & \dots \\
\mathbf{1} & \mathbf{1} & \mathbf{0} & 0 & 1 & 0 & \dots \\
0 & 1 & 0 & 0 & 1 & 0 & \dots \\
\dots & & & & & & \\
\mathbf{1} & \mathbf{1} & \mathbf{0} & 1 & 1 & 1 & \dots
\end{array}\right) \quad T = \left[\begin{array}{ccc|cccc}
0 & 0 & 0 & 0 & 0 & 0 & \dots \\
0 & 0 & 1 & 0 & 0 & 1 & \dots \\
0 & 1 & 0 & 1 & 1 & 1 & \dots \\
0 & 1 & 1 & 1 & 1 & 0 & \dots \\
1 & 0 & 0 & 1 & 0 & 1 & \dots \\
1 & 0 & 1 & 1 & 0 & 0 & \dots \\
\mathbf{1} & \mathbf{1} & \mathbf{0} & 0 & 1 & 0 & \dots \\
1 & 1 & 1 & 0 & 1 & 1 & \dots
\end{array}\right]$$

M4RI II

Input: A – a $m \times n$ matrix
Input: k – an integer $k > 0$
begin
  $r, c \longleftarrow 0, 0;$
  while $c < n$ **do**
    $\overline{k} \longleftarrow \textsc{GaussSubmatrix}(A, r, c, k, m);$ `// cubic!`
    $T, L \longleftarrow \textsc{MakeTable}(A, r, c, \overline{k});$
    $\textsc{AddRowsFromTable}(A, 0, r, c, \overline{k}, T, L);$
    $\textsc{AddRowsFromTable}(A, r + \overline{k}, m, c, \overline{k}, T, L);$
    $r, c \longleftarrow r + \overline{k}, c + \overline{k};$
    if $k \neq \overline{k}$ **then** $c \leftarrow c + 1;$
  end
end

M4RI III

- The average case complexity of M4RI is $\mathcal{O}(n^3/\log_2 n)$.
- The worst case complexity of M4RI as presented in this work is $\mathcal{O}(n^3)$.
- We use ideas from [ABH09] such as multiple pre-computation tables in our implementation of M4RI.

Our implementation of the M4RI algorithm is competitive with MAGMA's asymptotically fast implementation up to $32,000 \times 32,000$ on a Xeon and up to $16,384 \times 16,384$ on a Opteron.

MMPF

The Method of Many People Factorisation (MMPF) is our adaption of the M4RI algorithm to perform PLS decomposition.

MMPF: Column Swaps

Column swaps to compress L only have to happen at the very end of the algorithm and thus we can modify the M4RI algorithm in the obvious way to introduce them.

MMPF: U vs. I

Recall, that the function GaussSubmatrix generates small $\overline{k} \times \overline{k}$ identity matrices. Thus, we would fail to produce U.

The reason the original specification [Bar06] of the M4RI requires $\overline{k} \times \overline{k}$ identity matrices is to have a *a priori* knowledge of which bit combination corresponds with which index in AddRowsFromTable.

On the other hand, the rows of any $\overline{k} \times n$ upper triangular matrix also form a basis for the $\overline{k}$-dimensional vector space $\mathrm{span}(r, \ldots, r + \overline{k} - 1)$.

Thus, we can adapt GaussSubmatrix to compute the upper triangular matrix instead of the identity.

MMPF: Preserving L

For MMPF we need to fix the table T to update the transformation matrix correctly.

For example, assume $\overline{k} = 3$ and that the first row of the $\overline{k} \times n$ submatrix generated by $\textsc{GaussSubmatrix}$ has the first $\overline{k}$ bits equal to `[1 0 1]`. Assume further that we want to clear $\overline{k}$ bits of a a row which also starts with `[1 0 1]`.

Then – in order to generate L – we need to encode that this row is cleared by adding the first row only, i.e. we want the first $\overline{k} = 3$ bits to be `[1 0 0]`. Thus, we want the first row to read `[0 0 1]` $=$ `[1 0 1]` $\oplus$ `[1 0 0]`.

MMPF: Other Bookkeeping

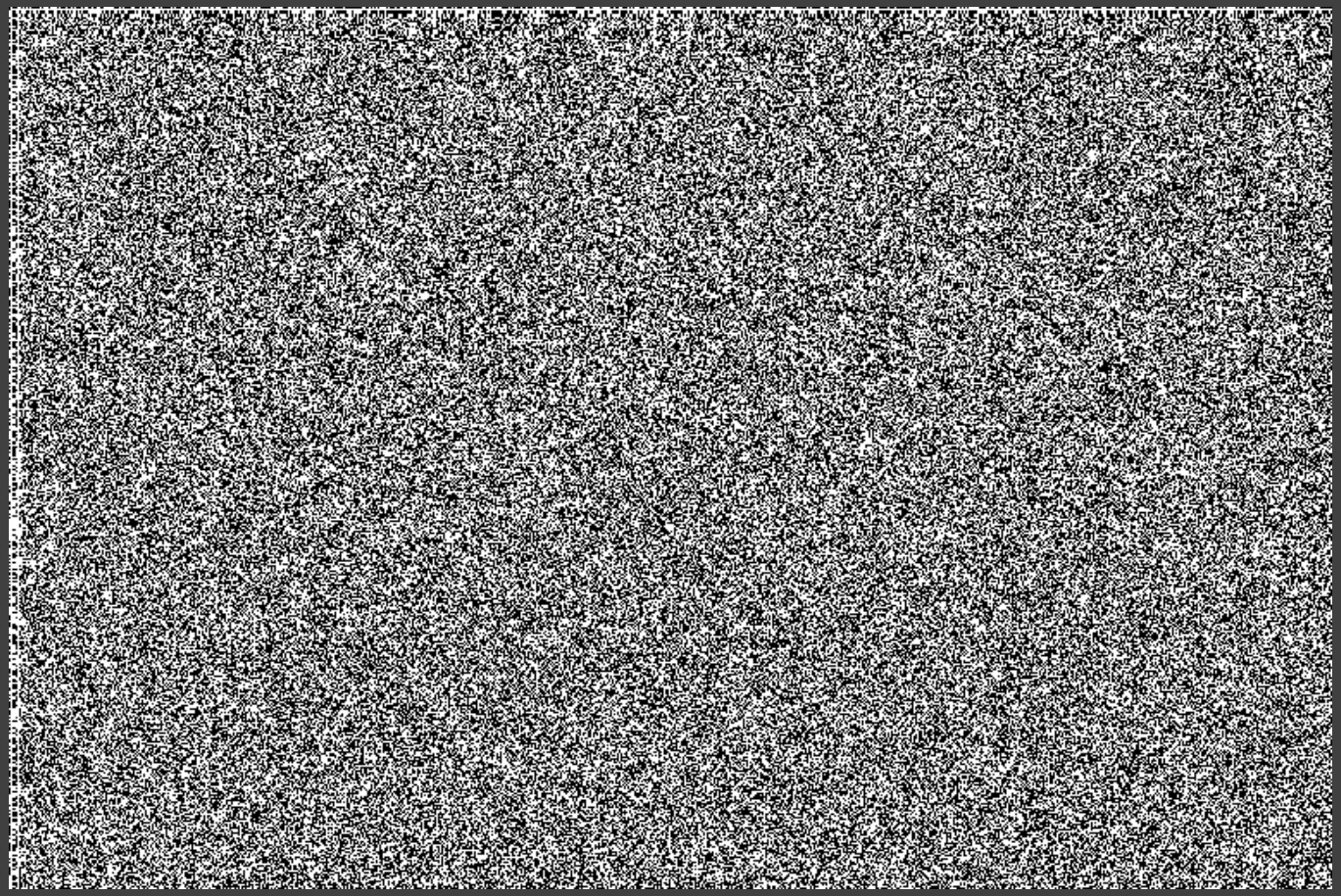
Recall that GaussSubmatrix's interaction with the M4RI algorithm uses the fact that processed columns of a row are zeroed out to encode whether a row is "done" or not. This is not true if we compute the PLS decomposition since we store L below the main diagonal.

We explicitly encode up to which row a given column is "done" in PlsSubmatrix which replaces GaussSubmatrix in MMPF.

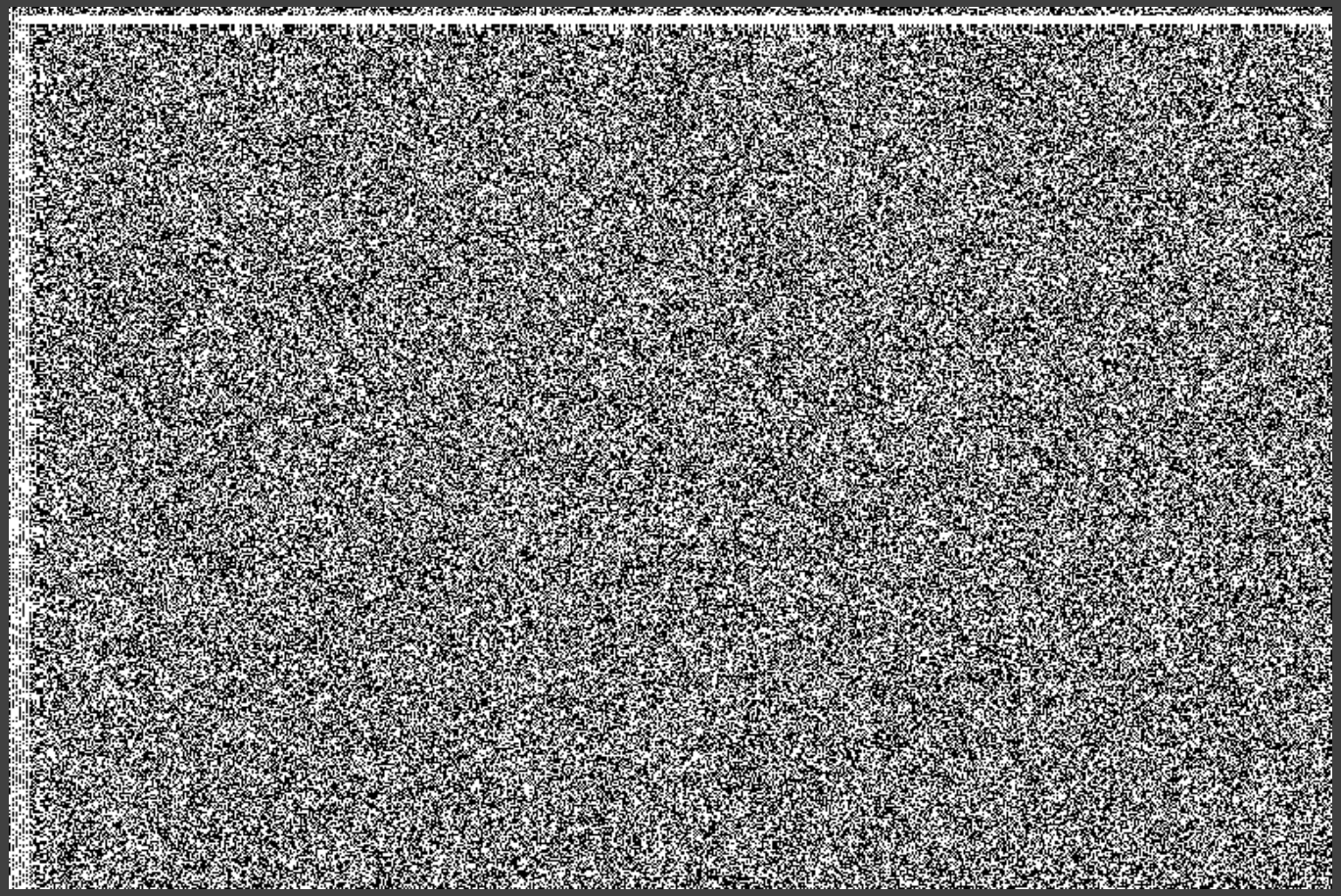
MMPF: Main Steps

```
begin
    while r < m and c < n do
        k̄, d_r ⟵ PlsSubmatrix(A, r, c, k, P, Q);
        U ⟵ the upper triangular submatrix starting at (r, 0);
        if k̄ > 0 then
            T, L ⟵ MakeTable1(U, 0, c, k̄);
            AddRowsFromTable(A, d_r + 1, m, c, k̄, T, L);
            r, c ← r + k̄, c + k̄;
        else
            c ← c + 1; // skip zero column
        end
    end
    // Now compress L
end
```

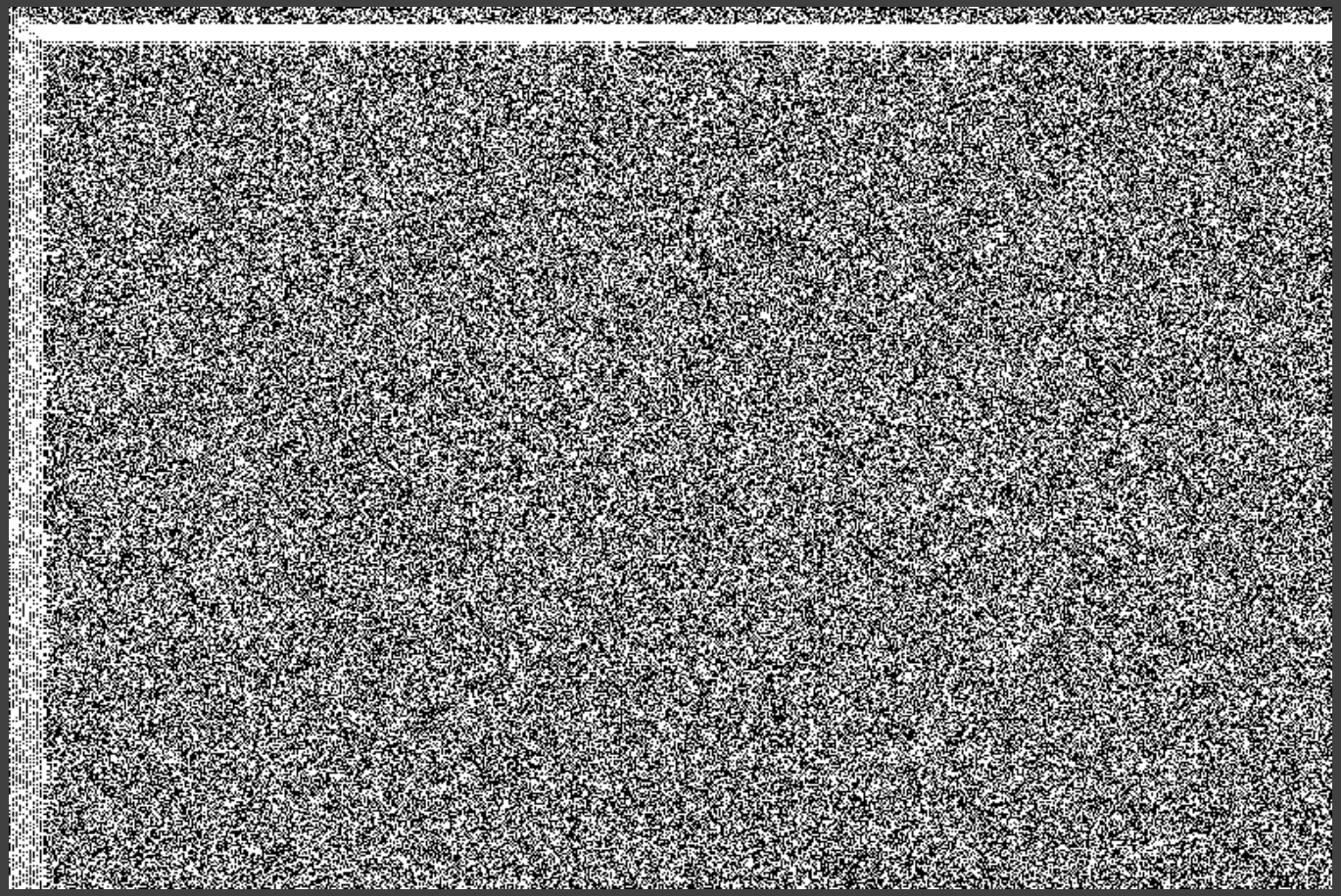
Visualisation



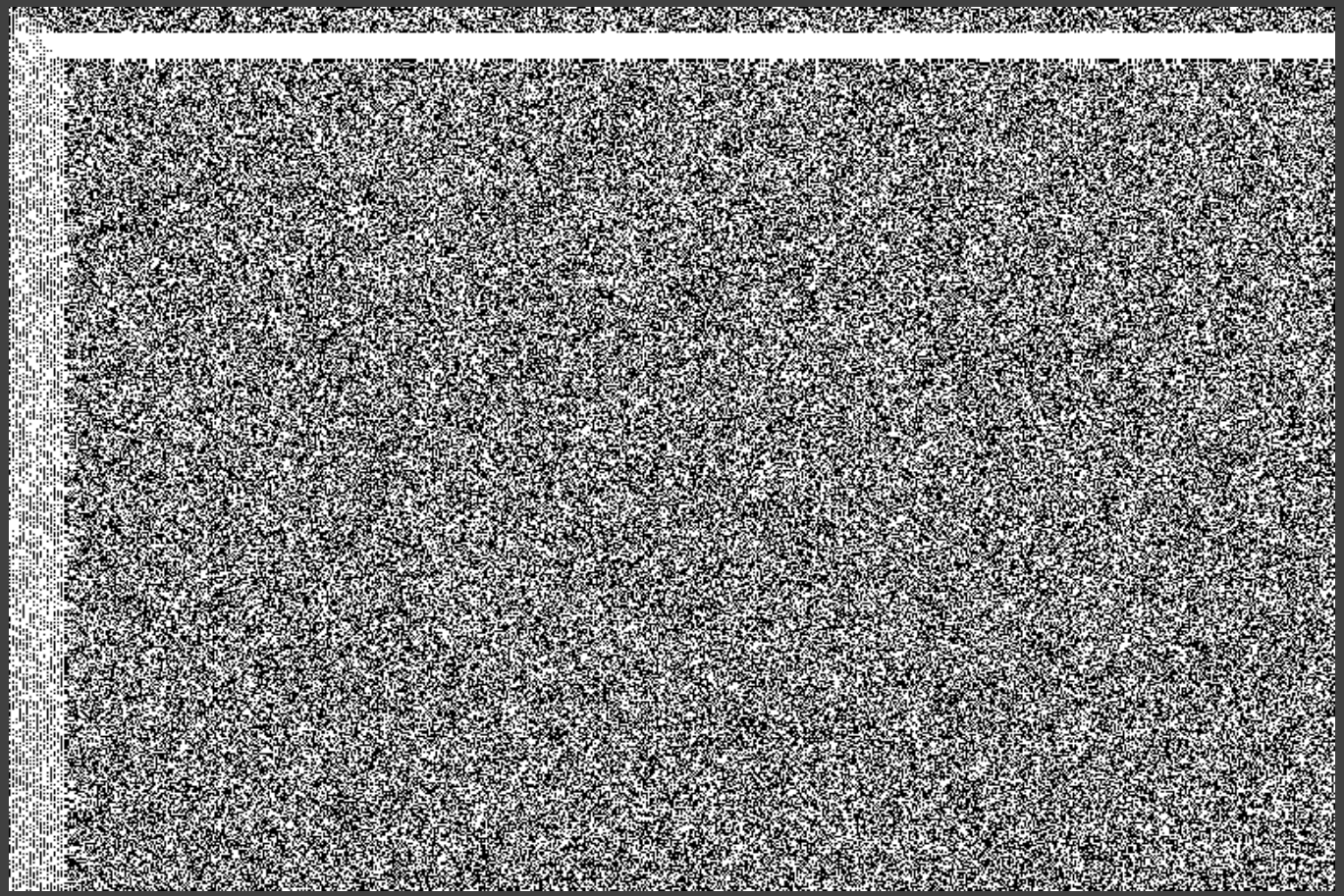
Visualisation



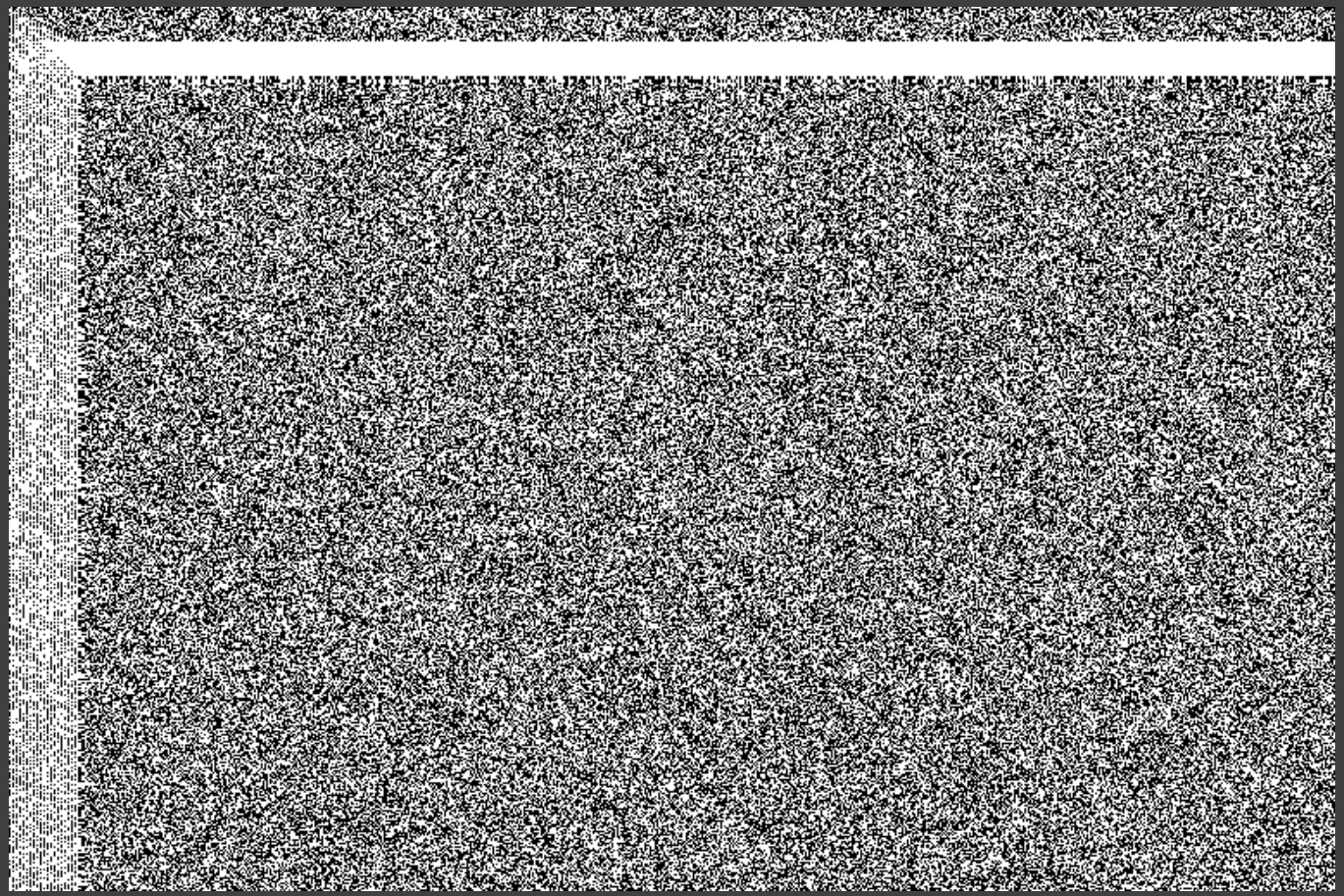
Visualisation



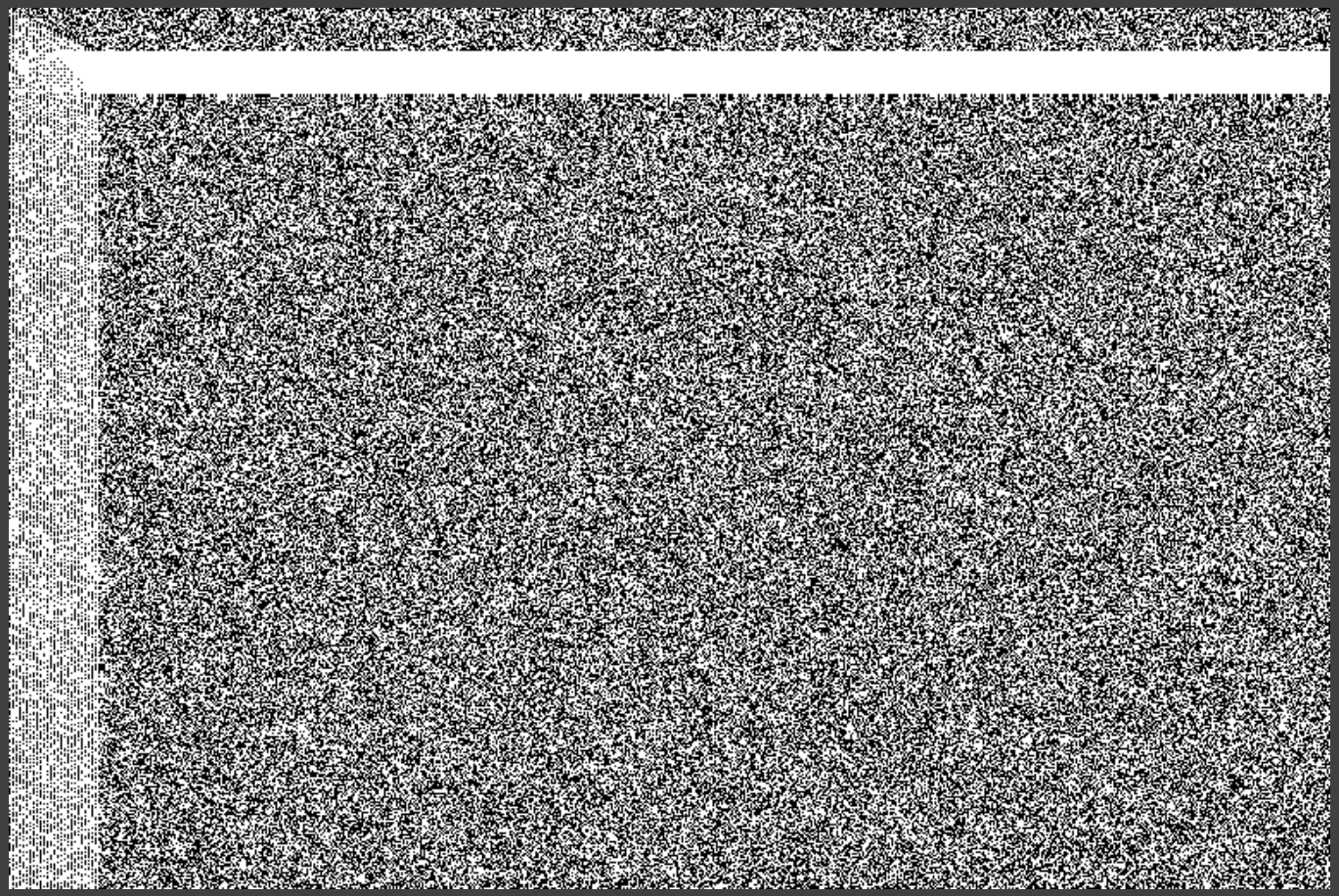
Visualisation



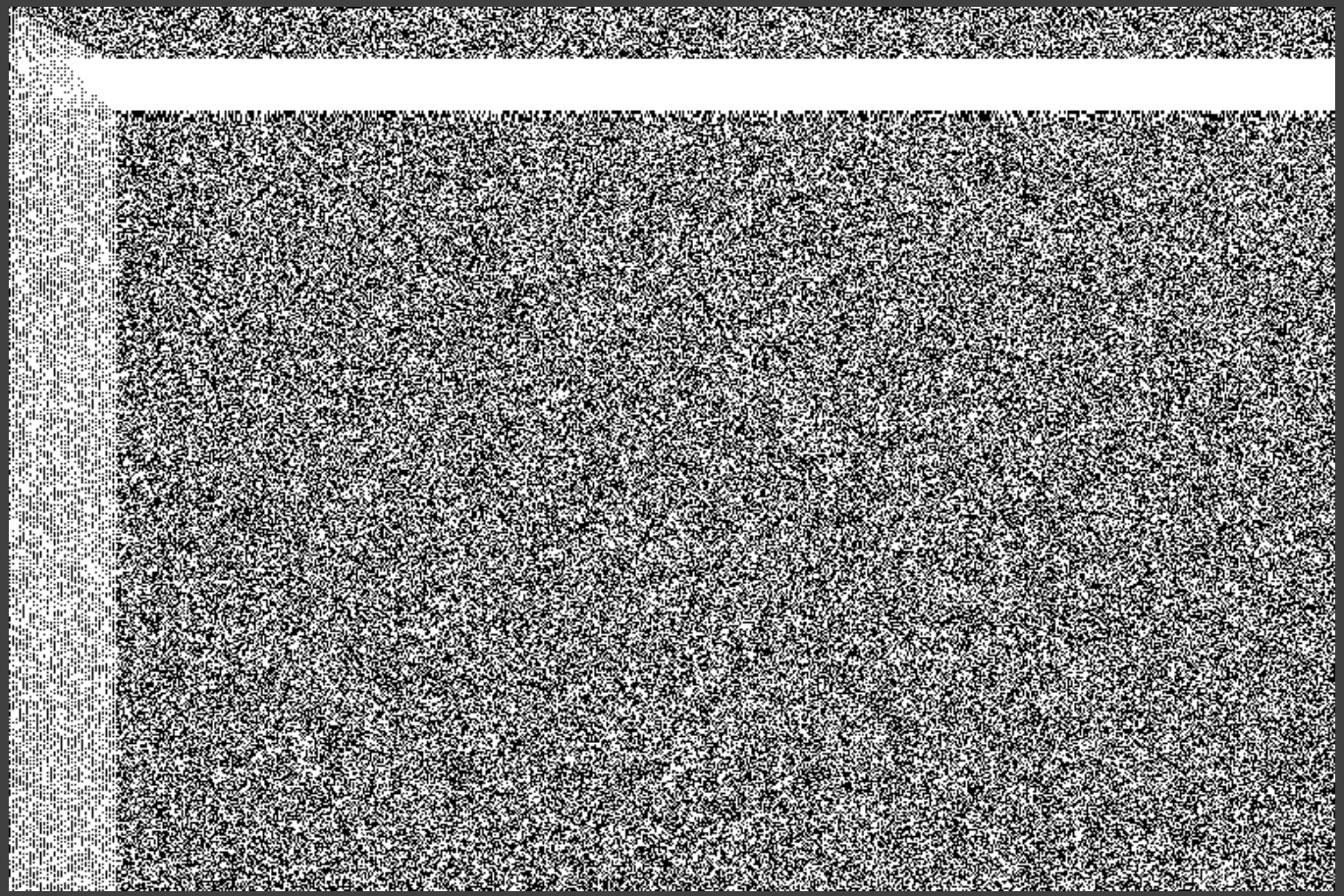
Visualisation



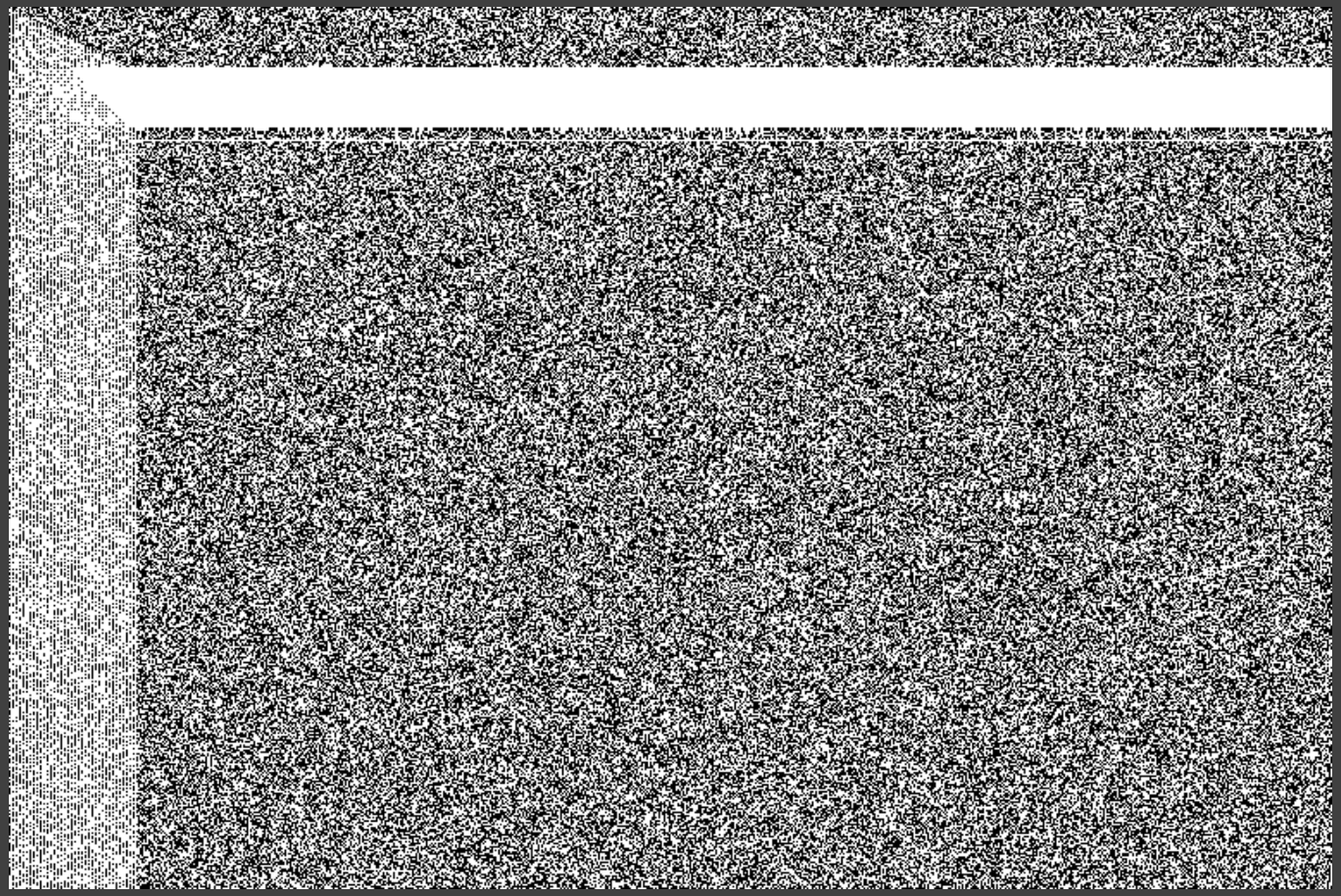
Visualisation



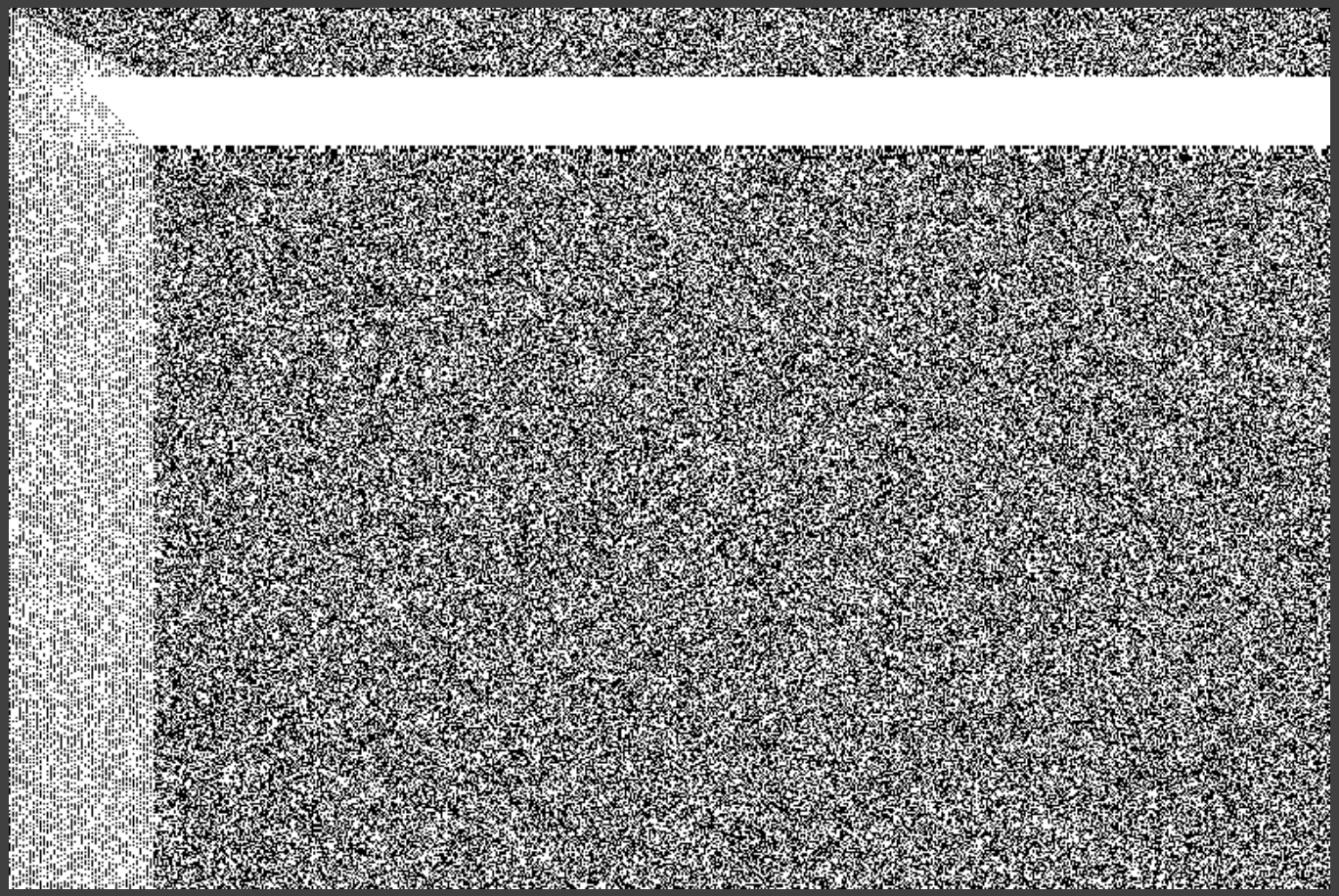
Visualisation



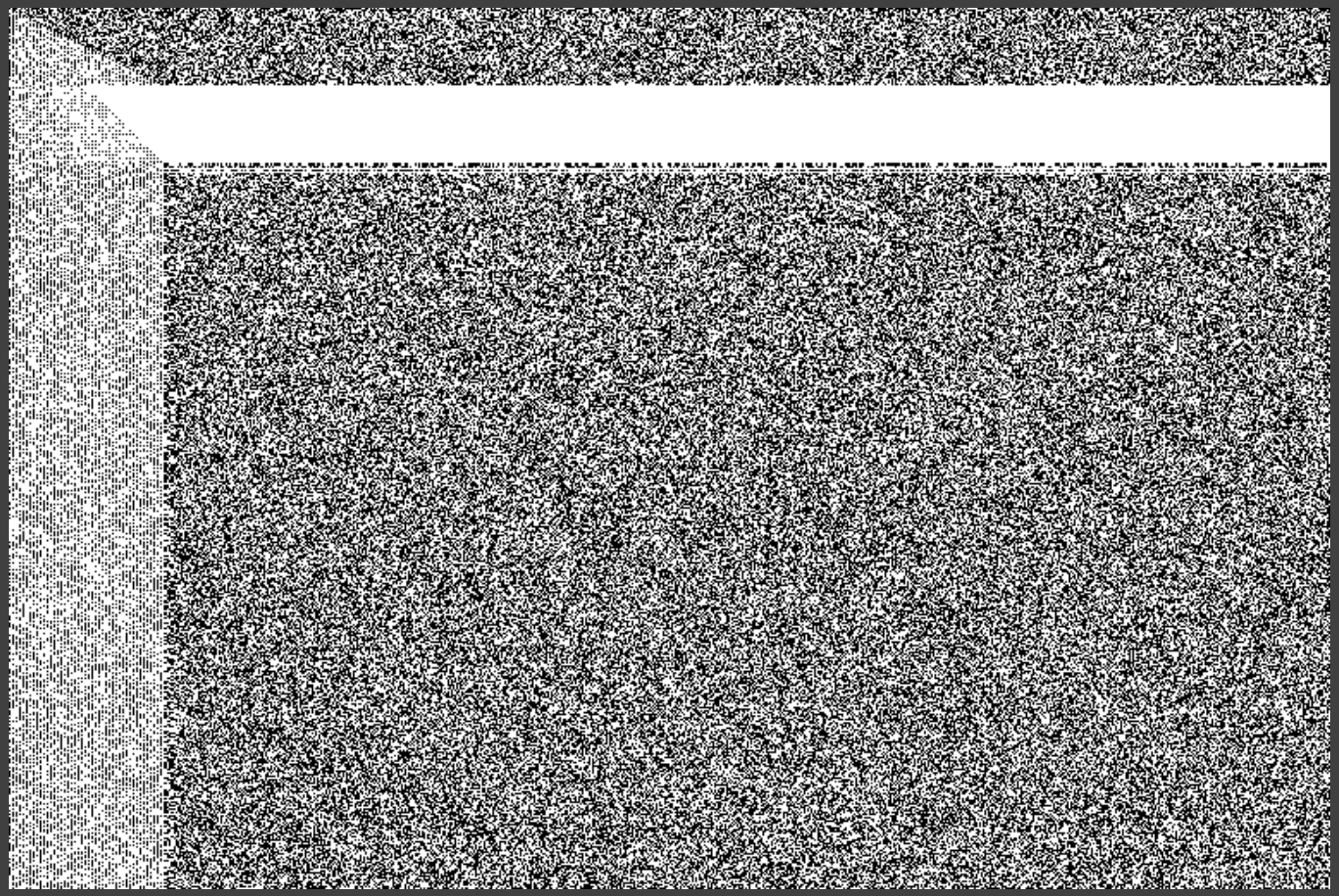
Visualisation



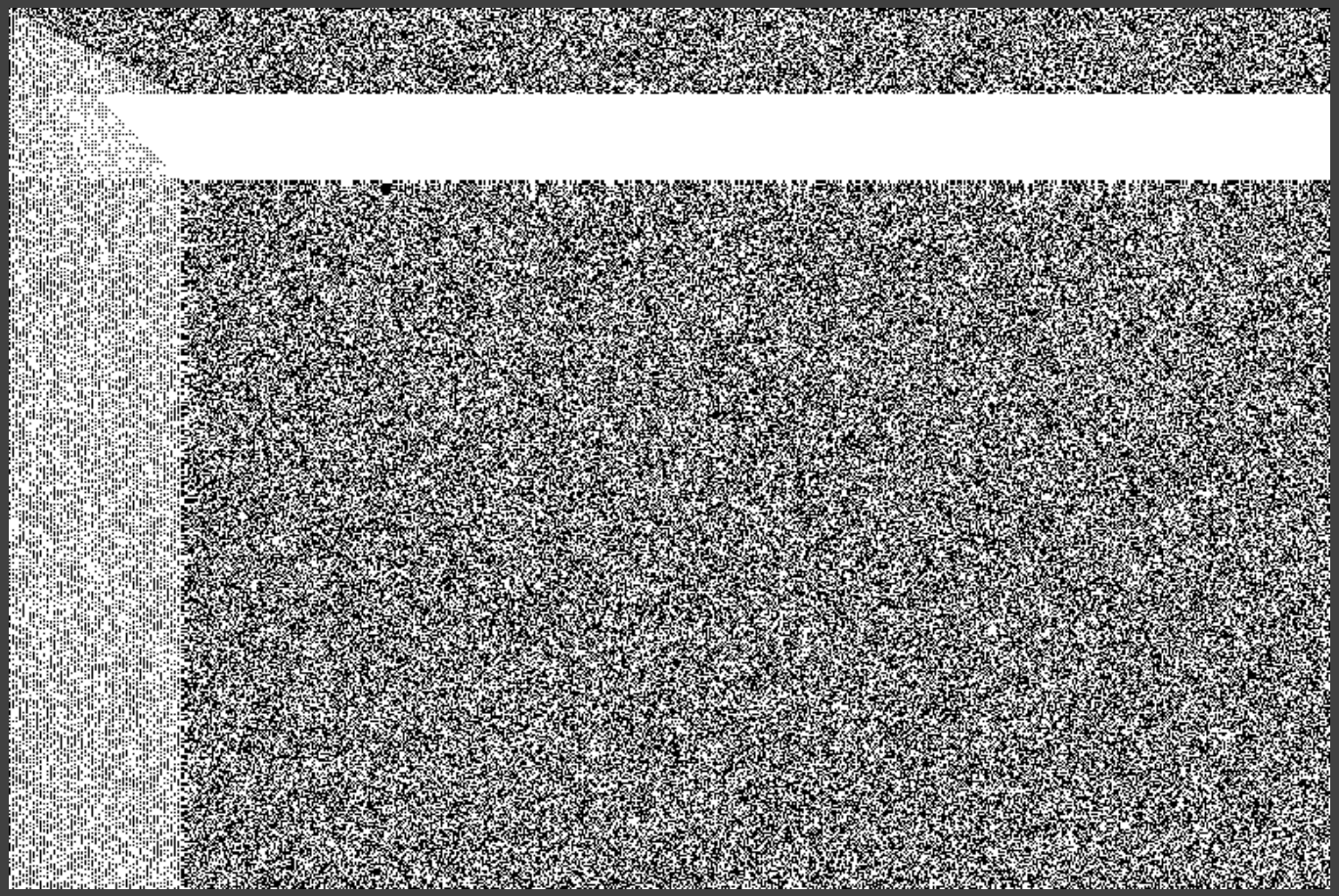
Visualisation



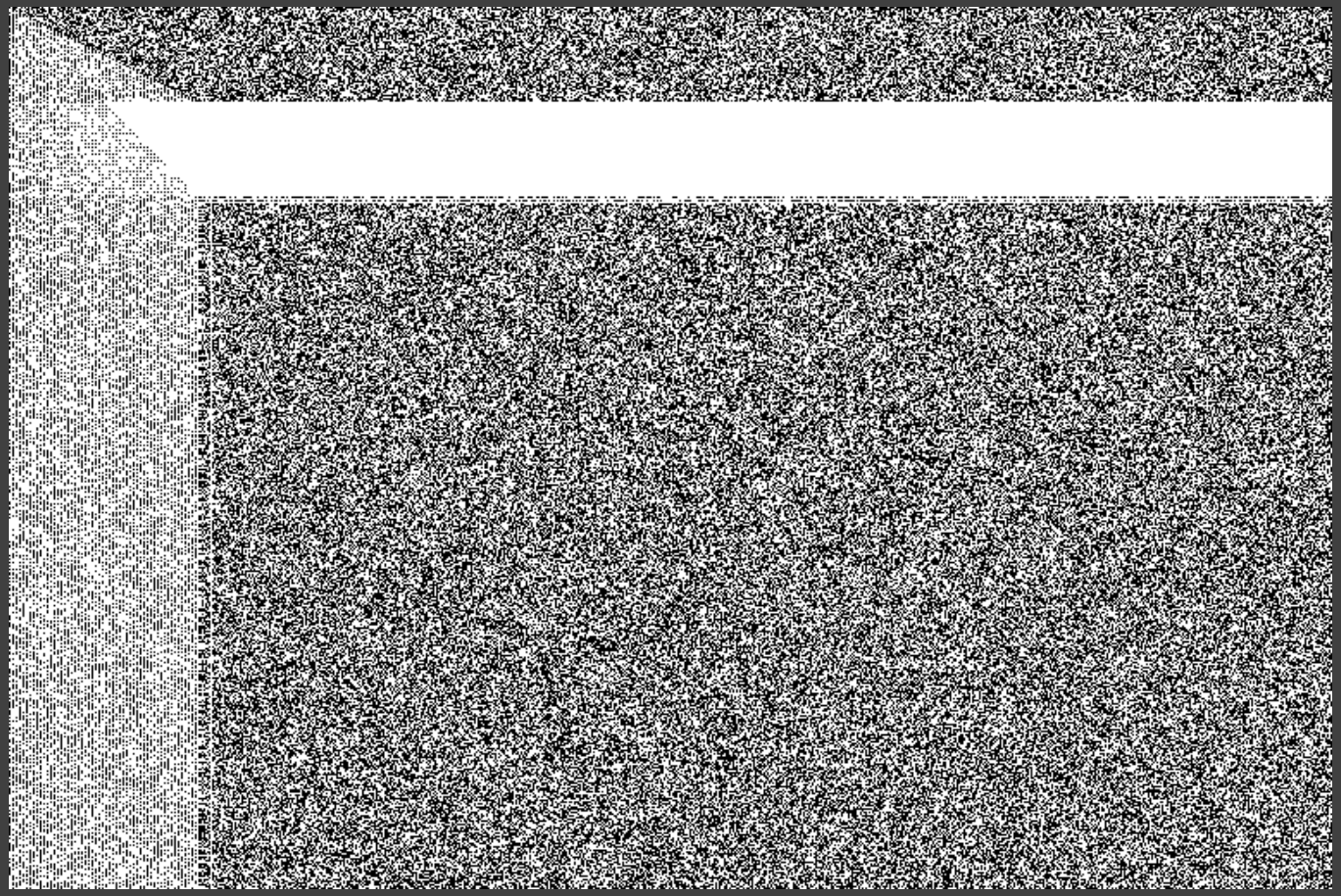
Visualisation



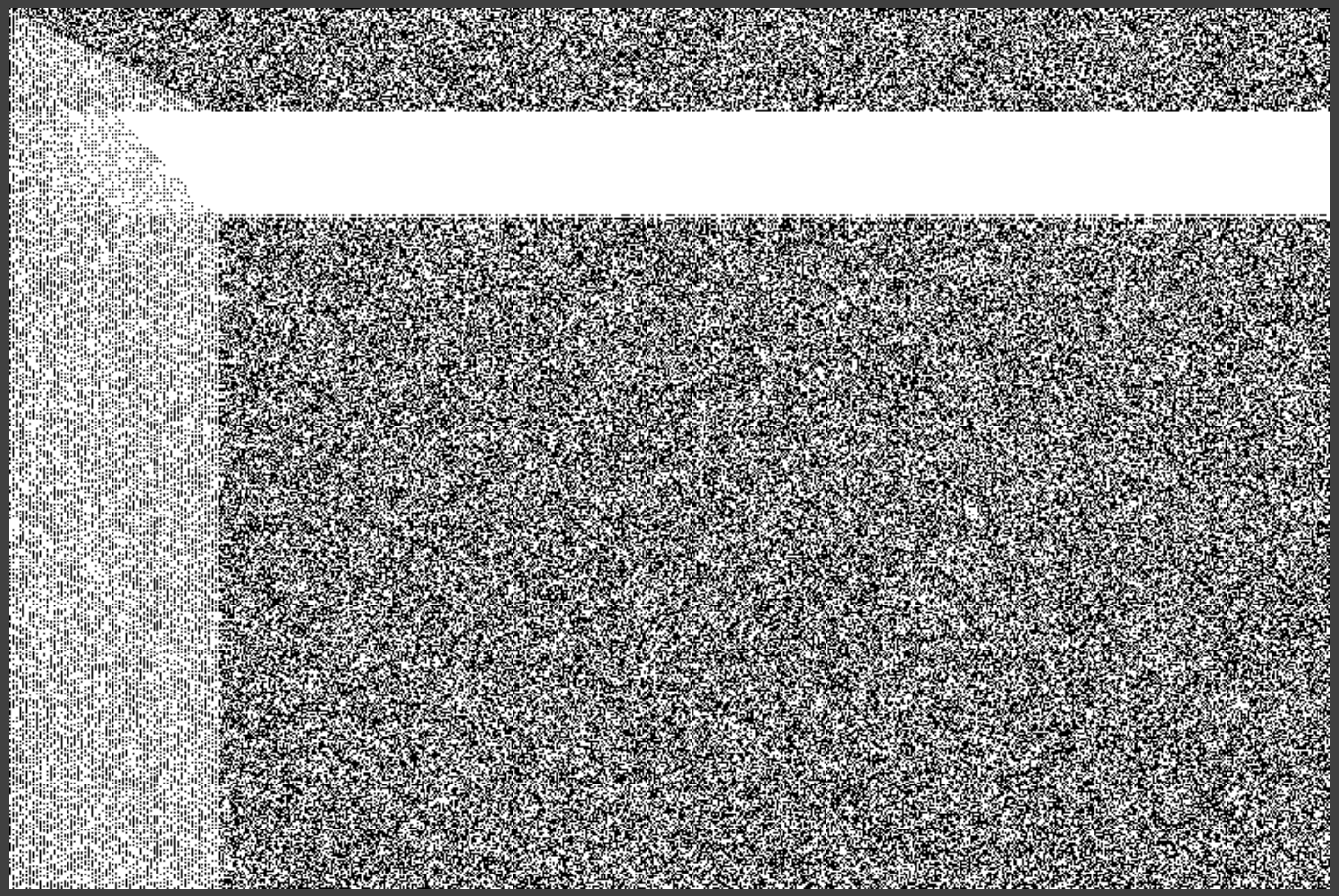
Visualisation



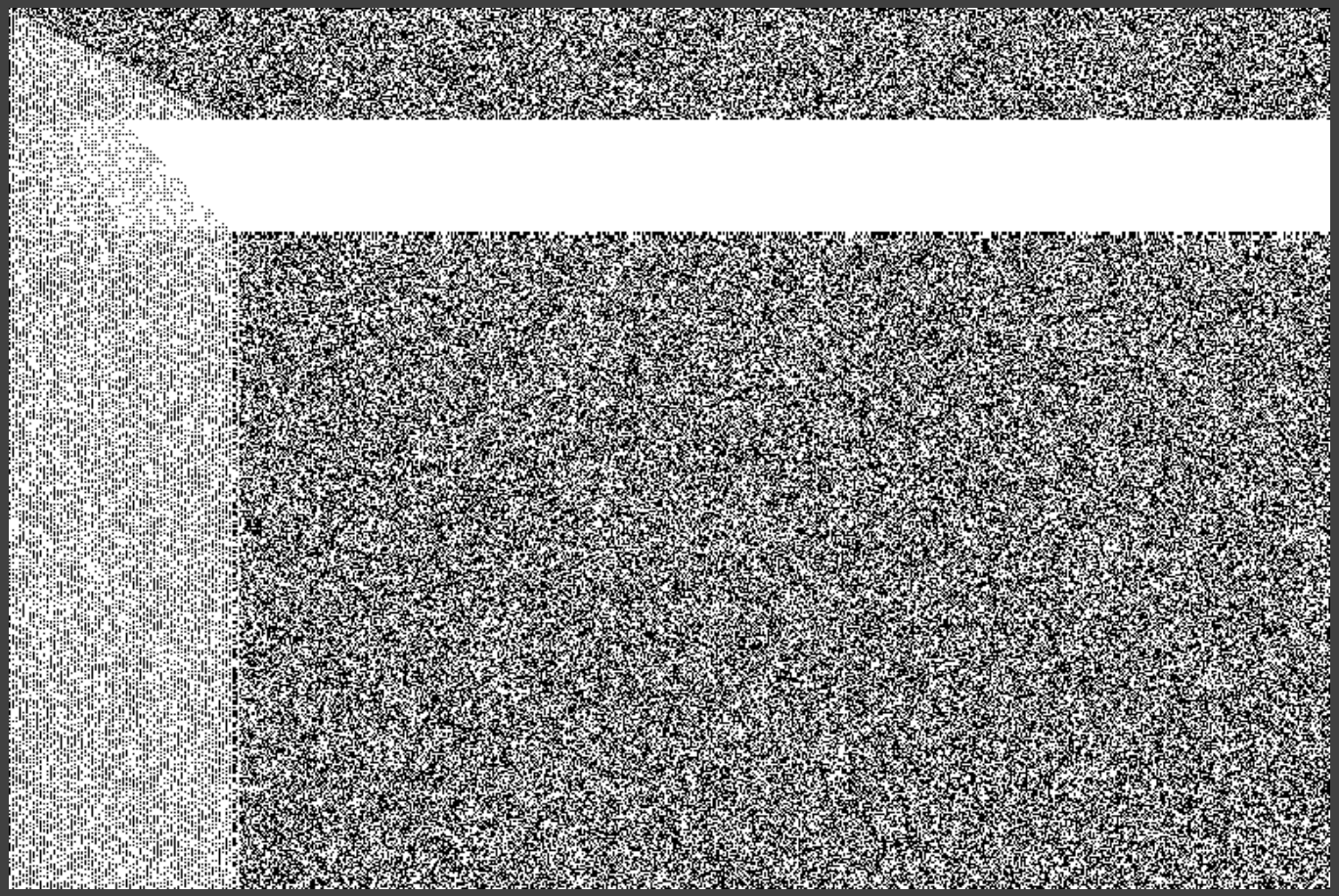
Visualisation



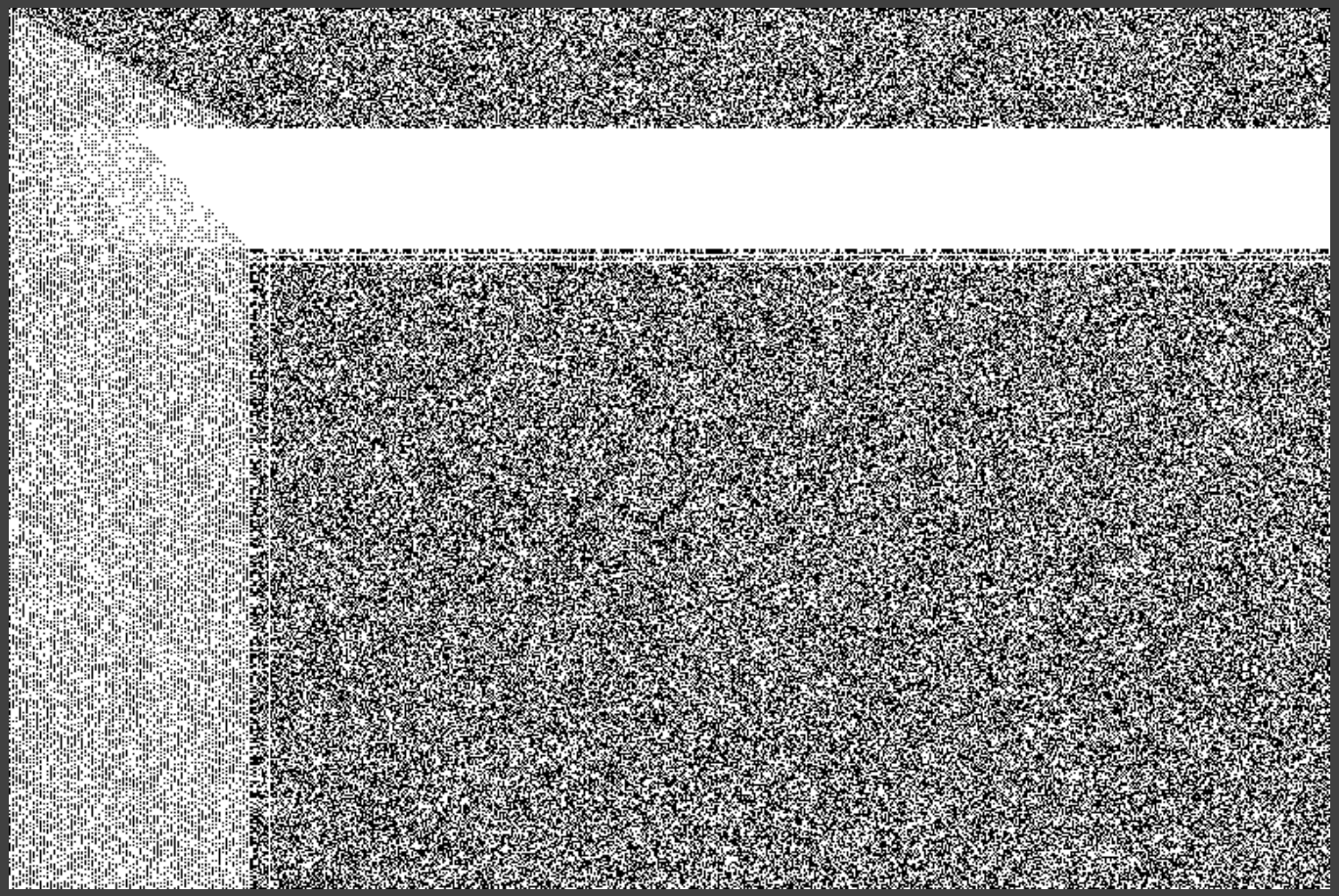
Visualisation



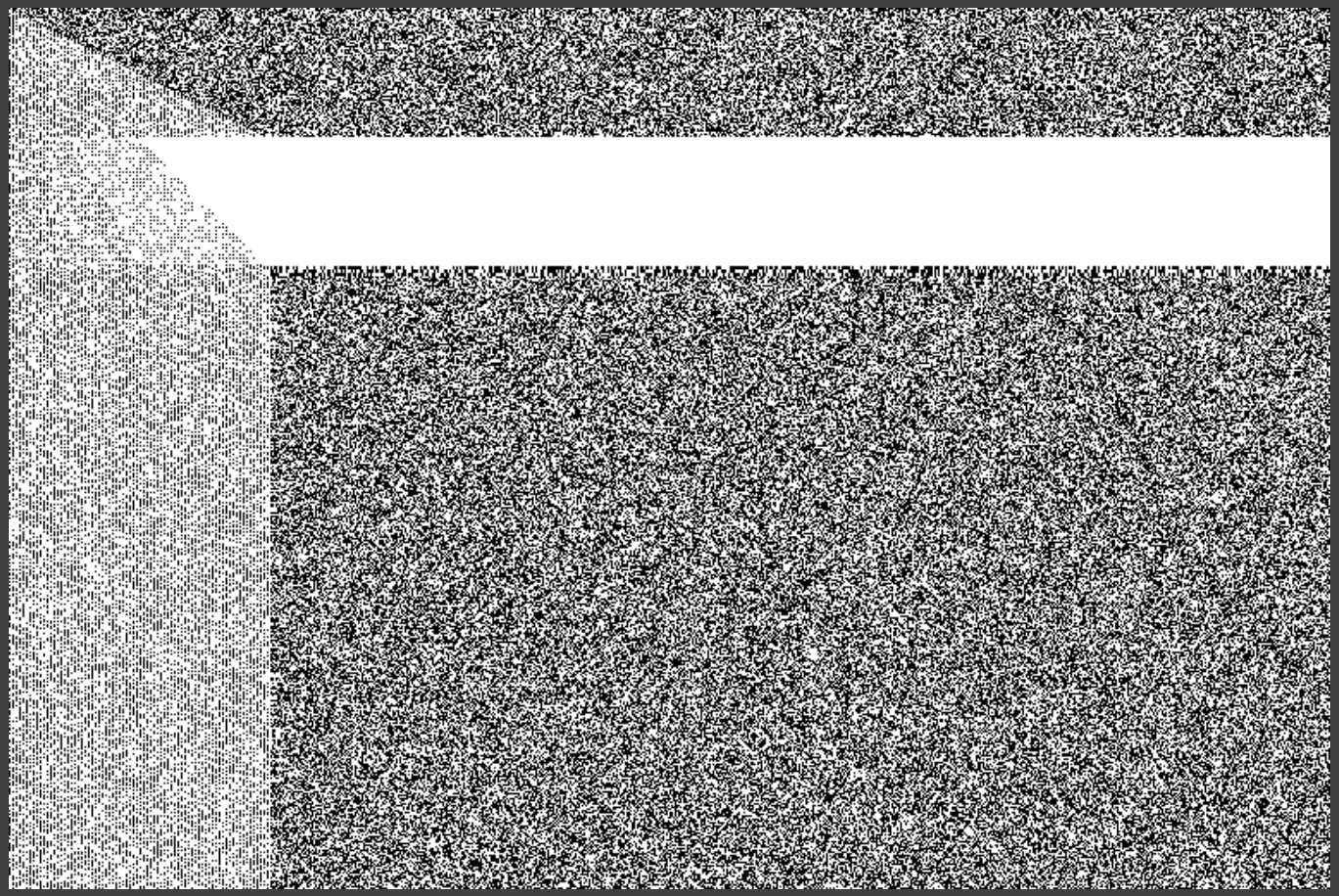
Visualisation



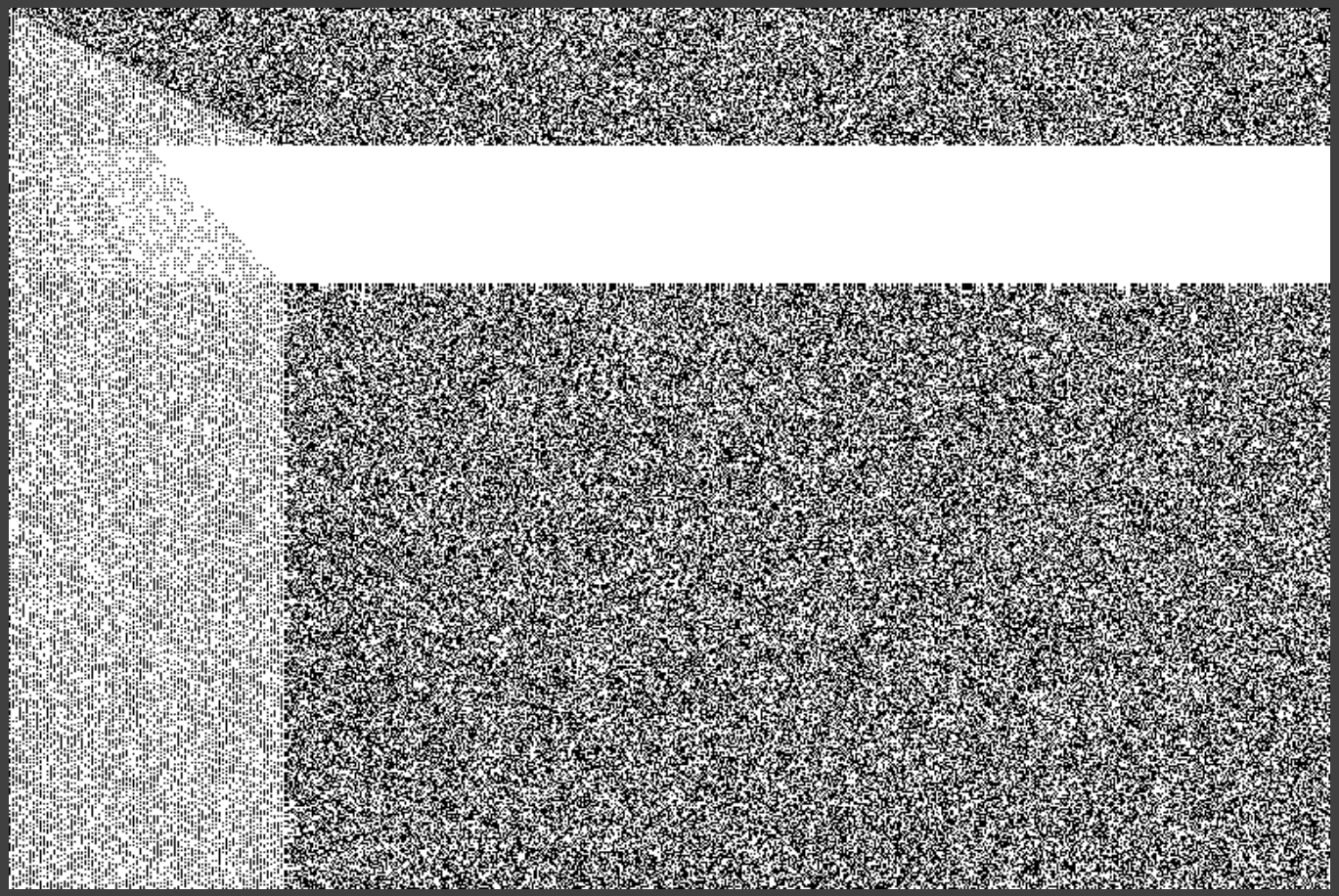
Visualisation



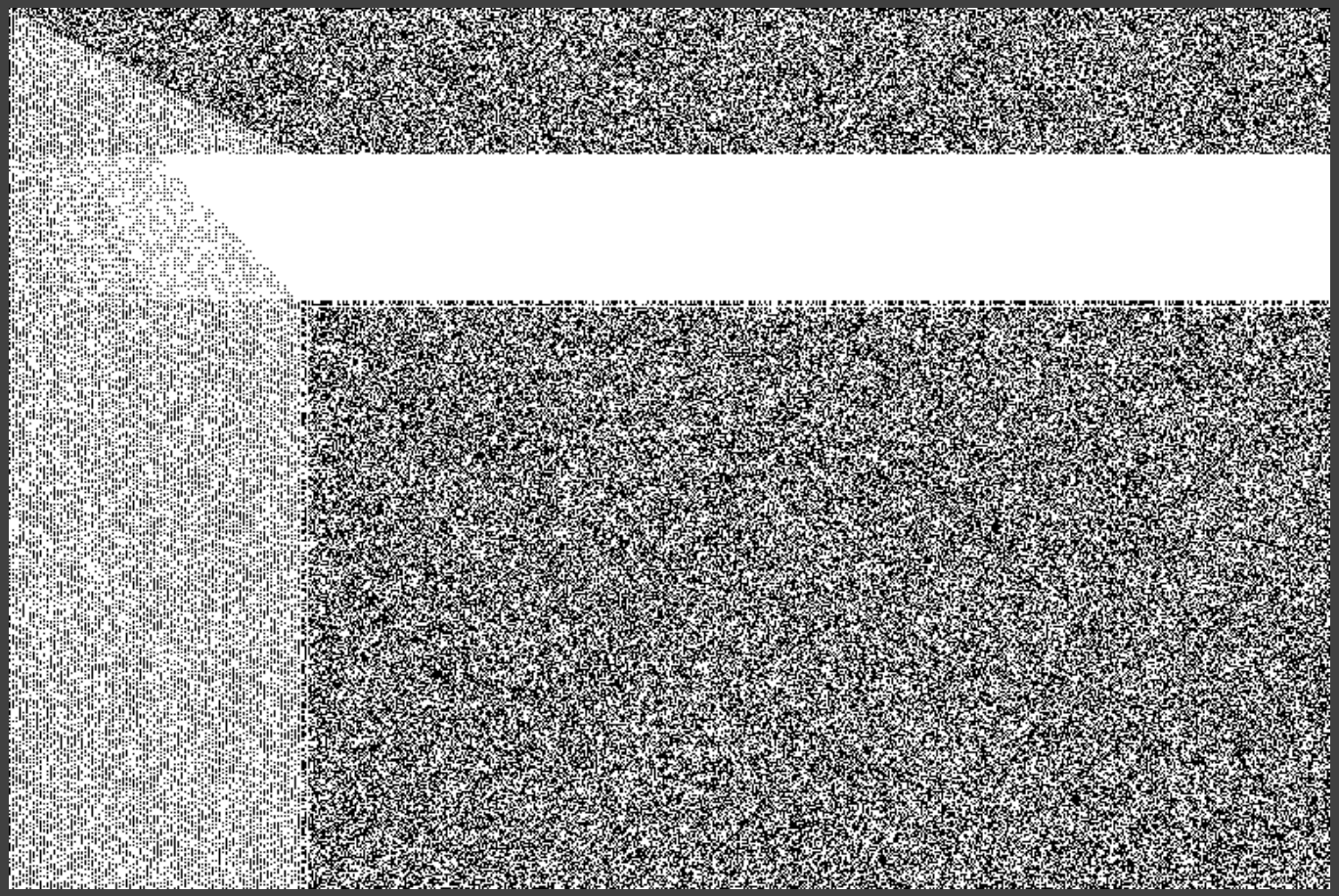
Visualisation



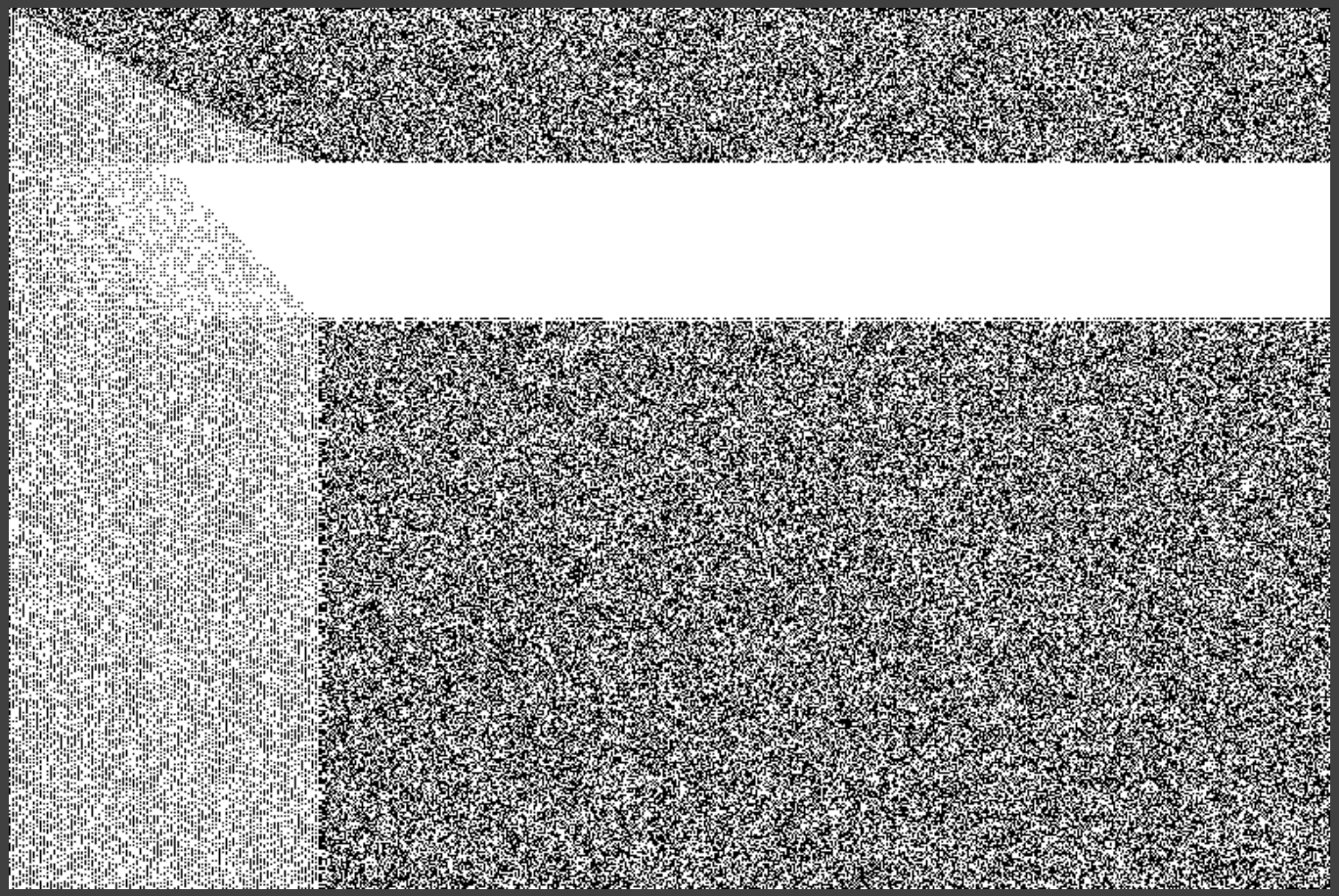
Visualisation



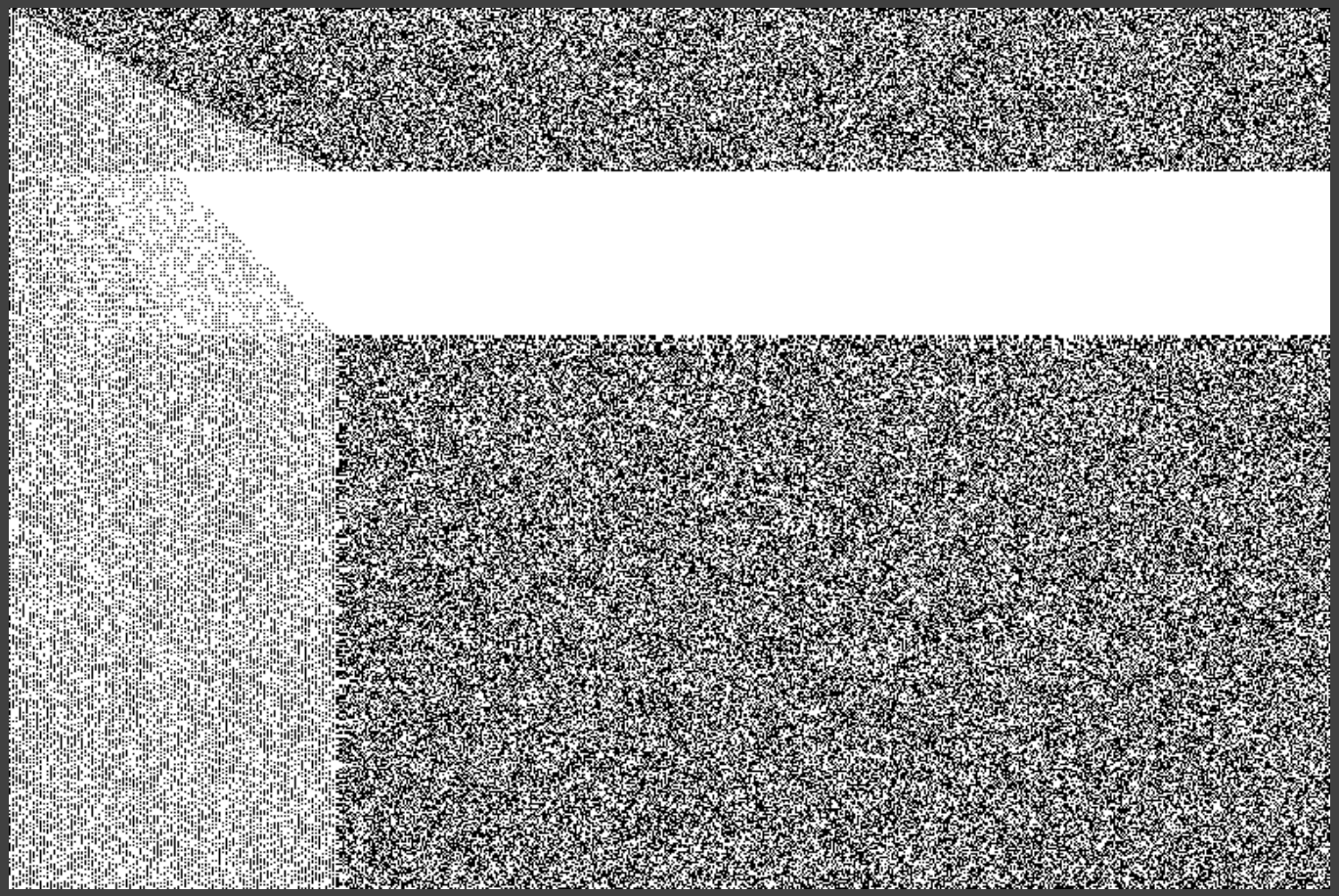
Visualisation



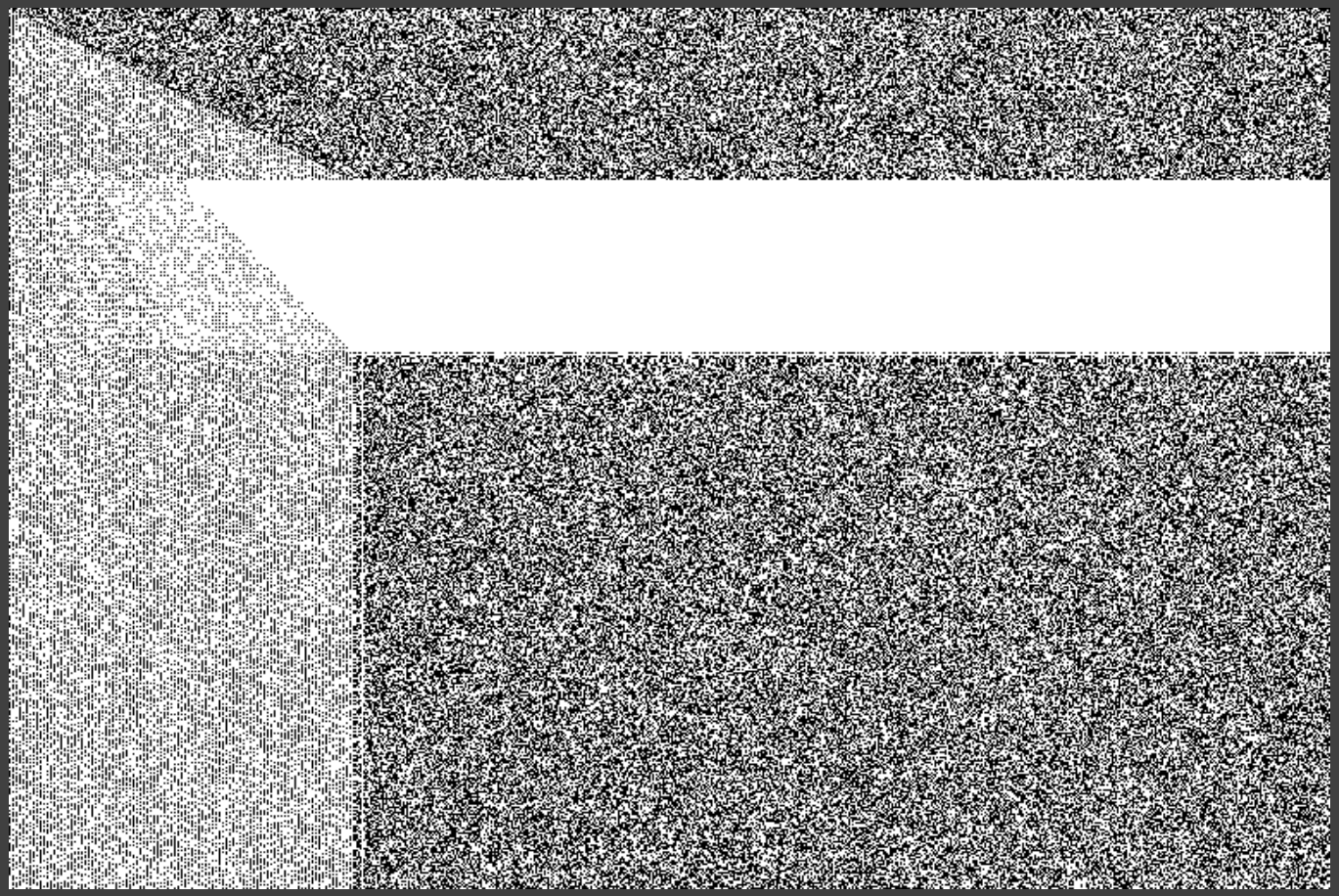
Visualisation



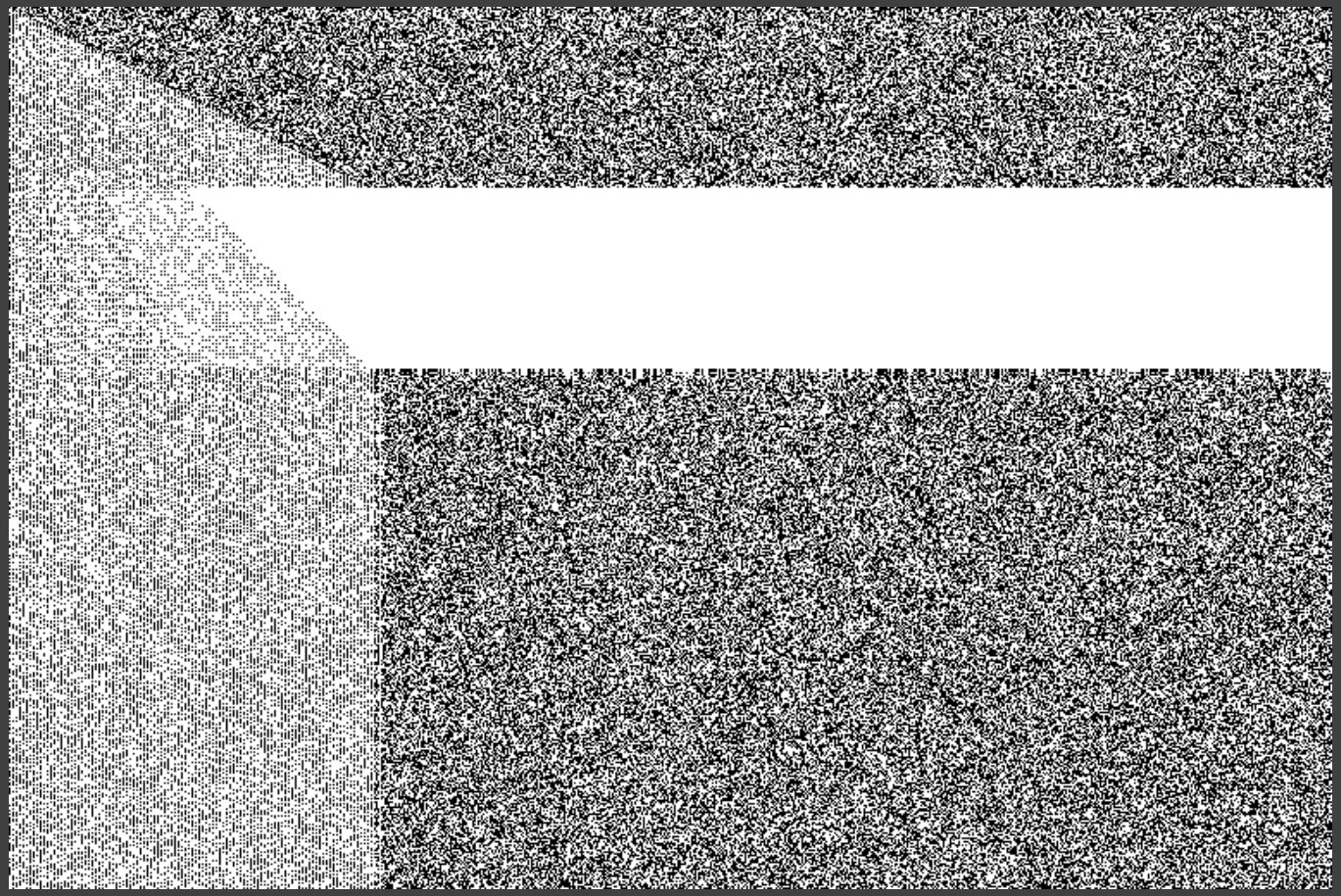
Visualisation



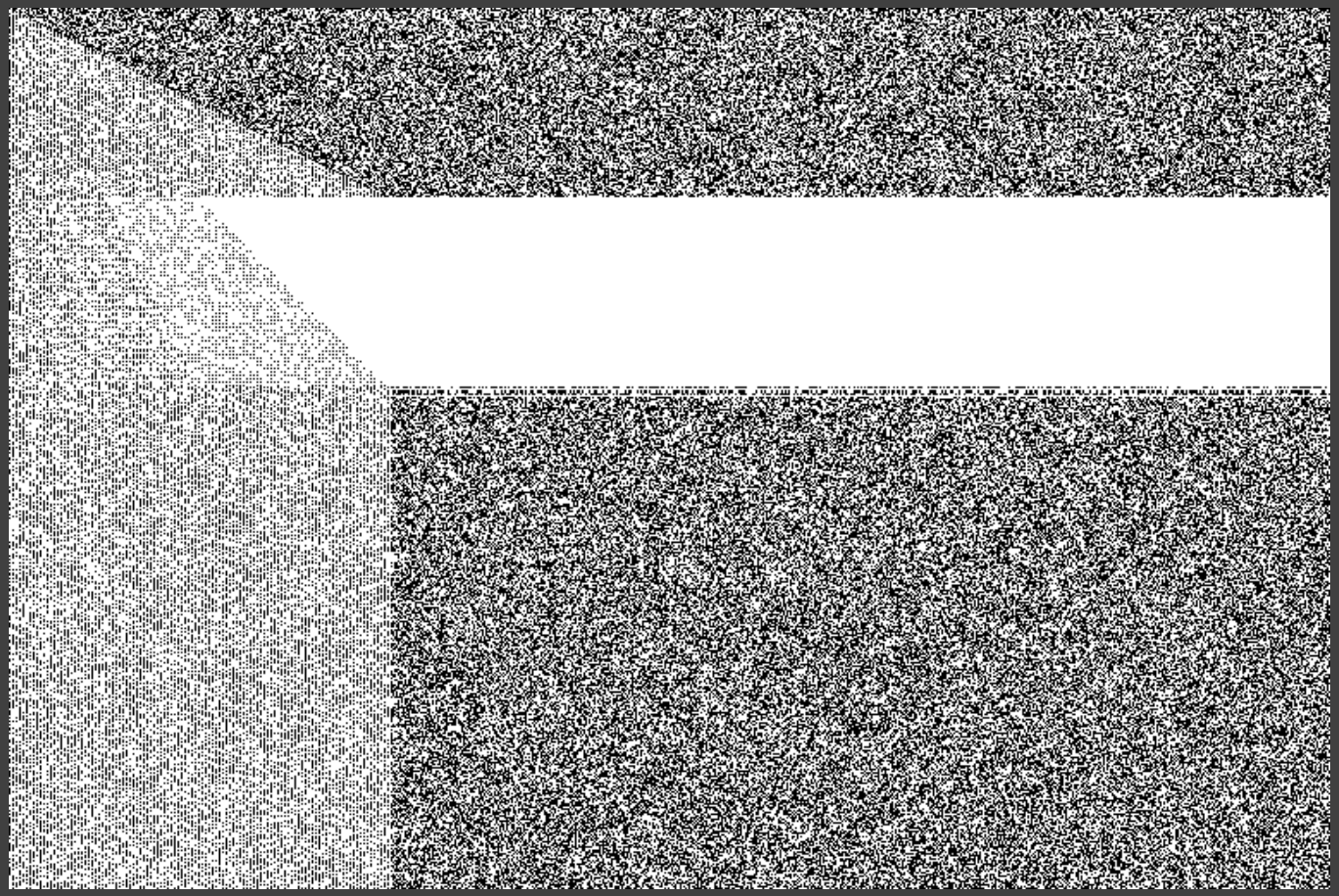
Visualisation



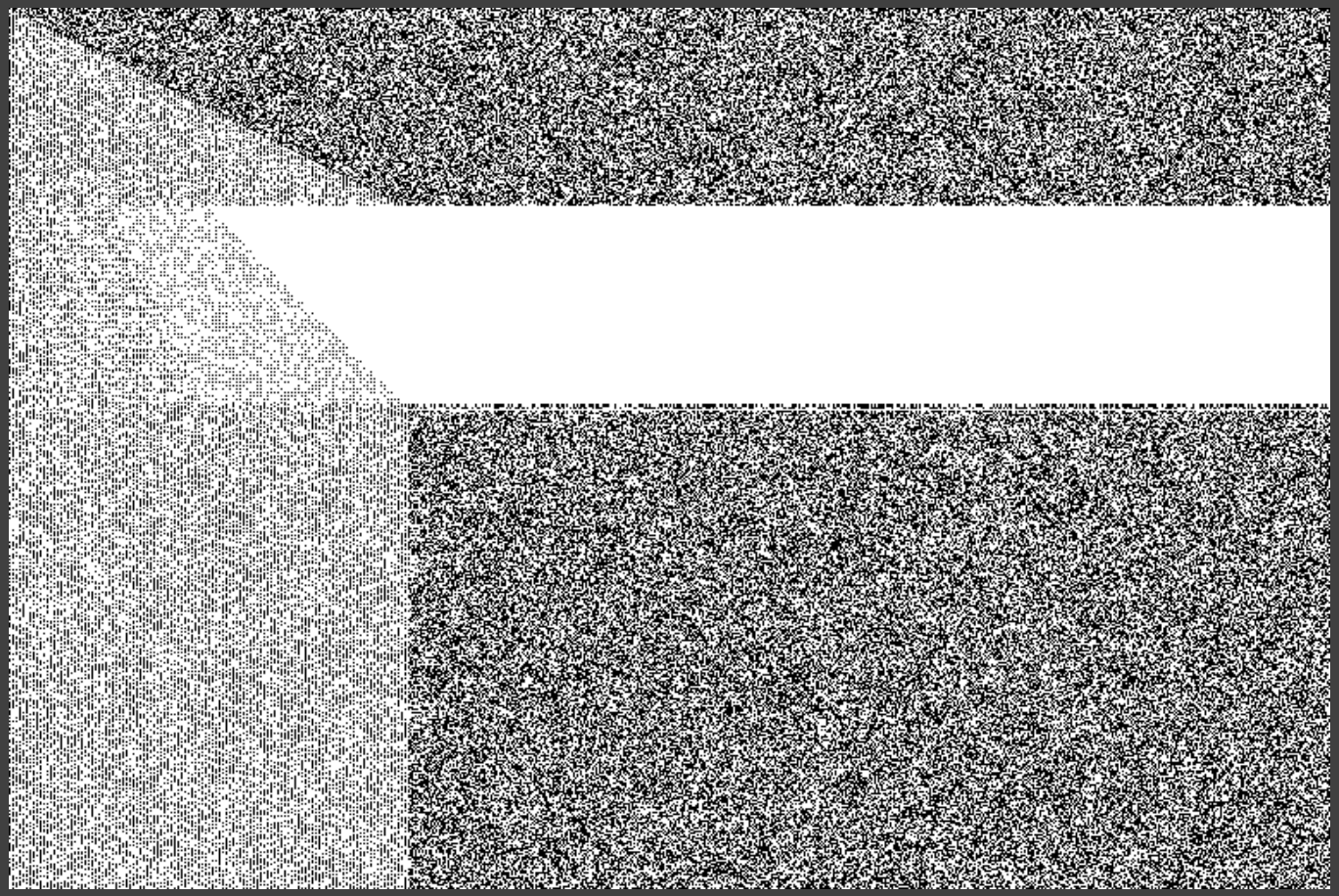
Visualisation



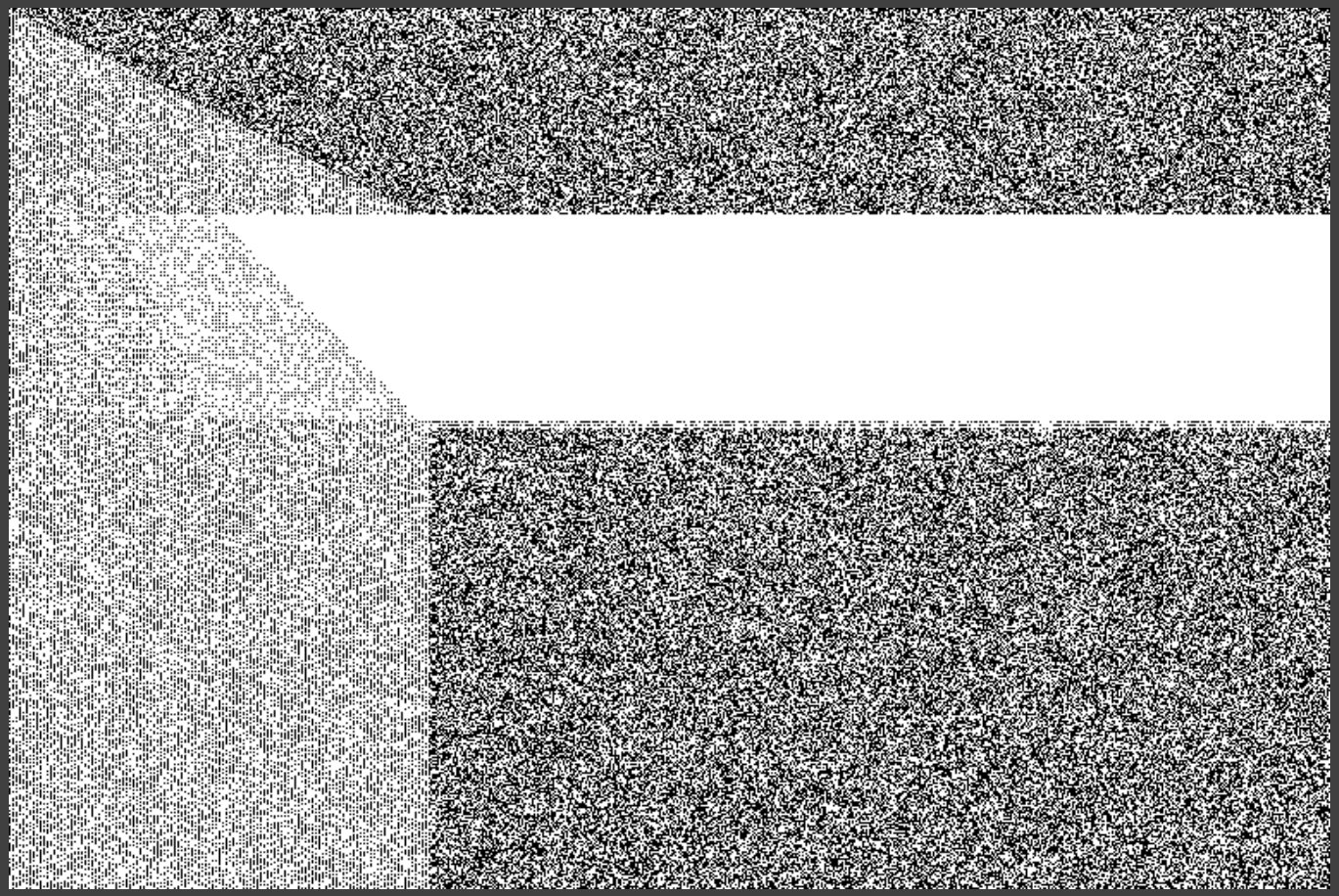
Visualisation



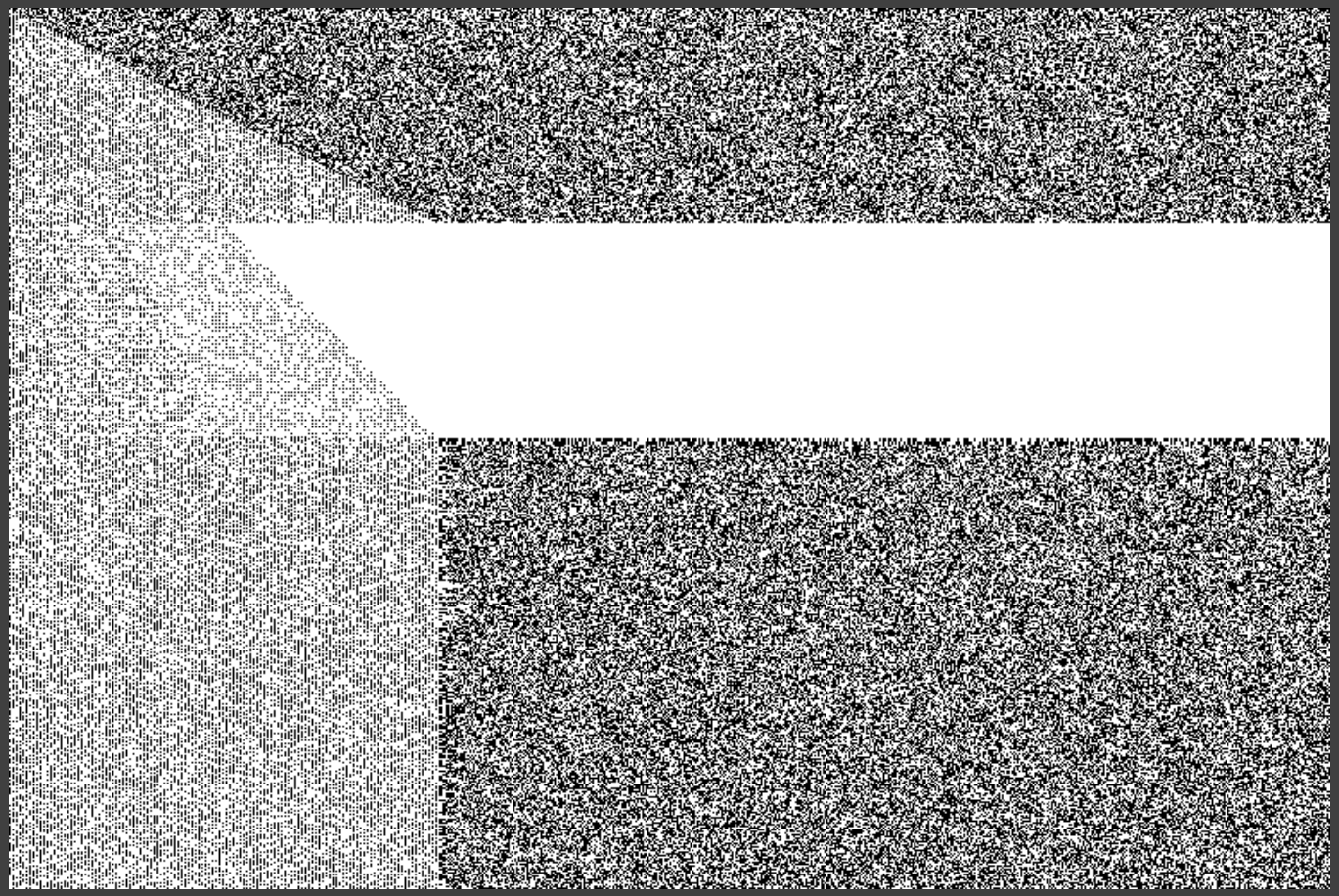
Visualisation



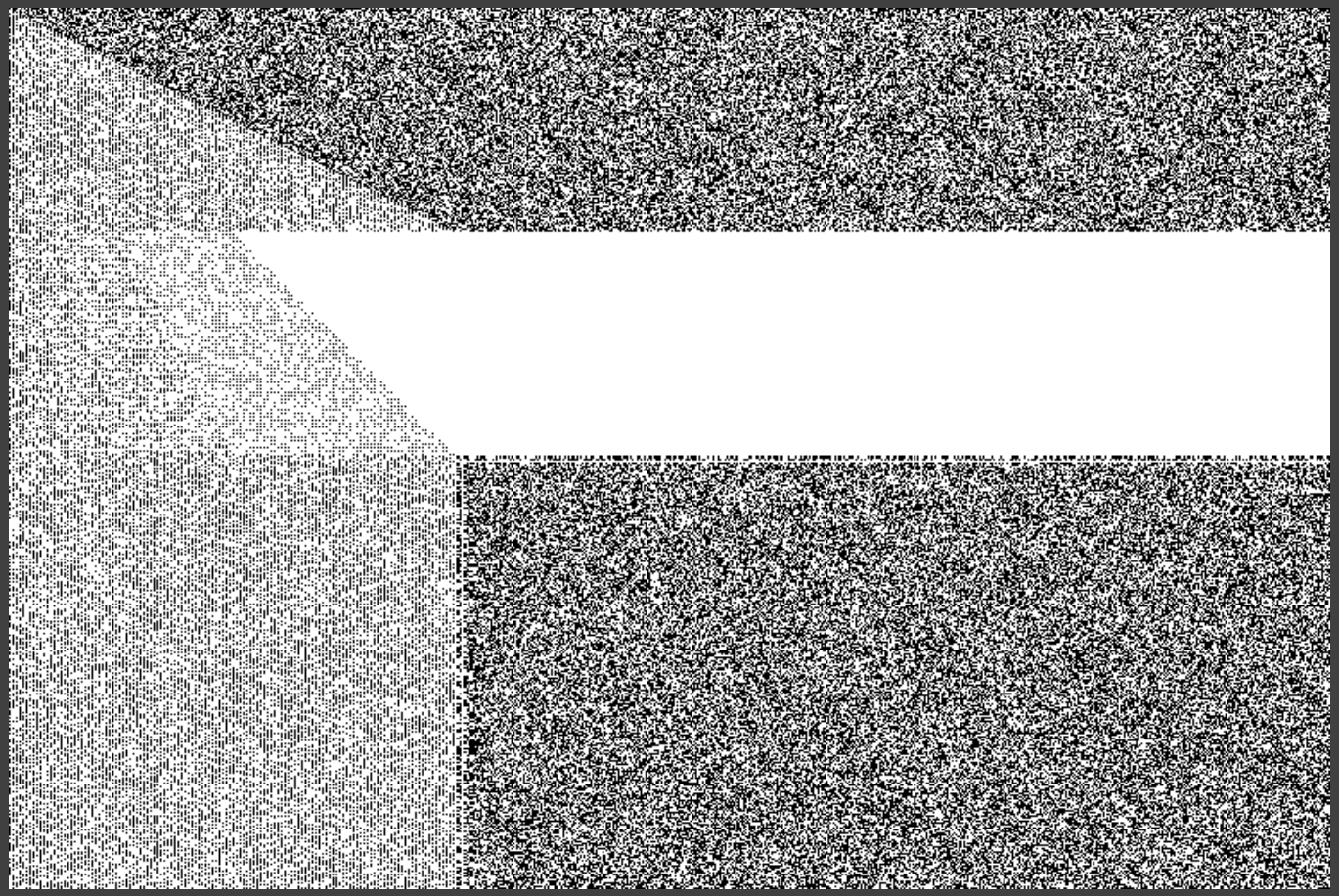
Visualisation



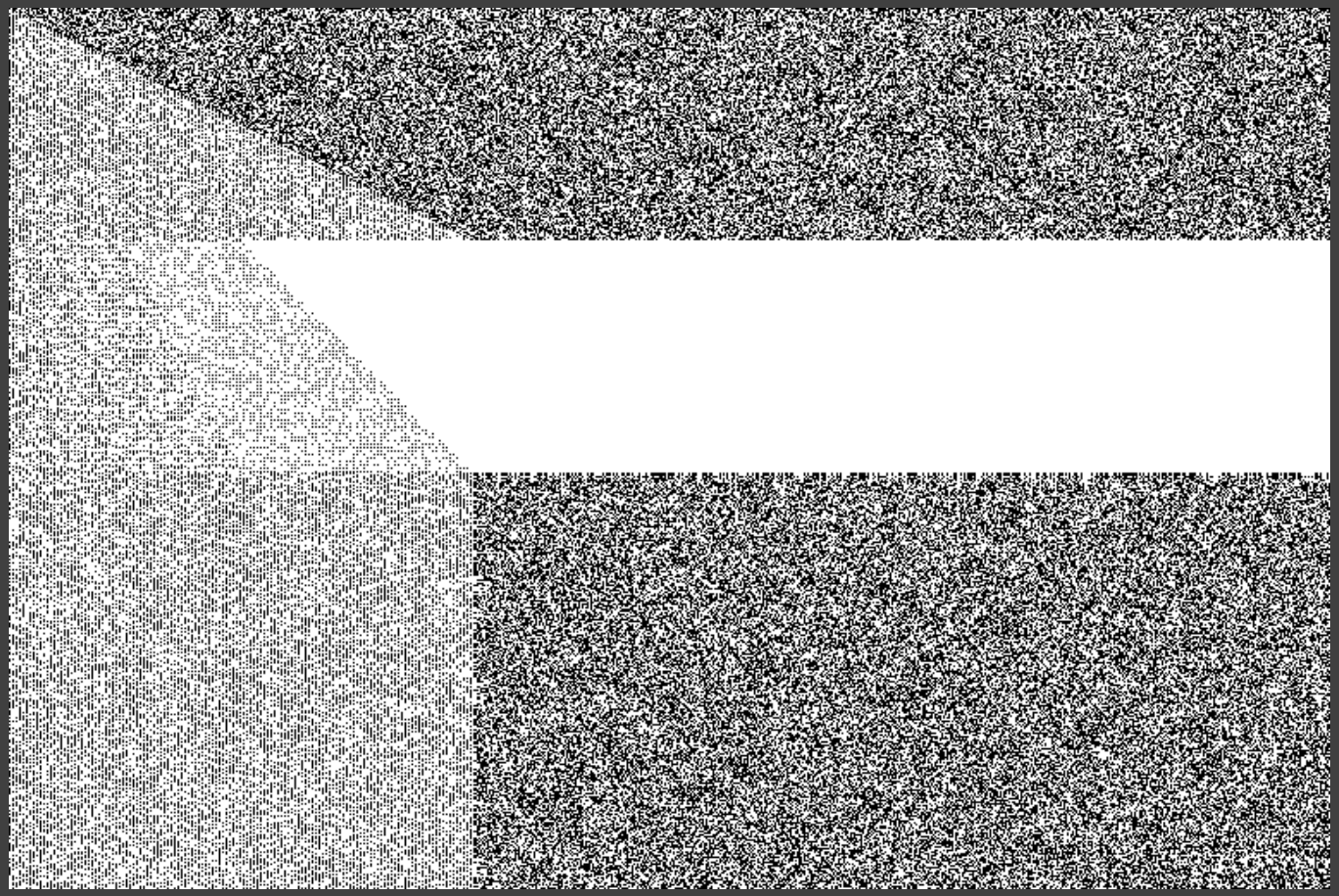
Visualisation



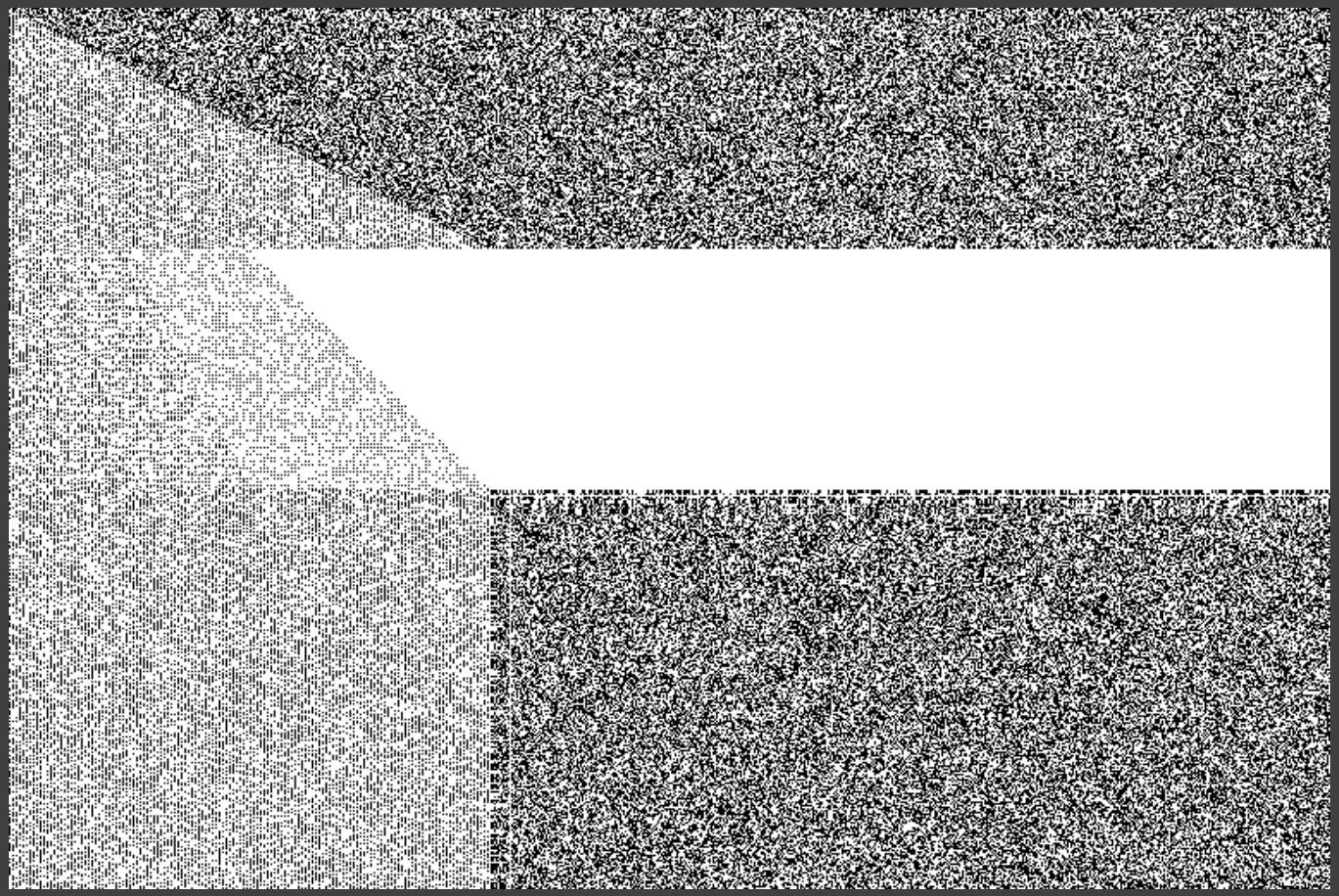
Visualisation



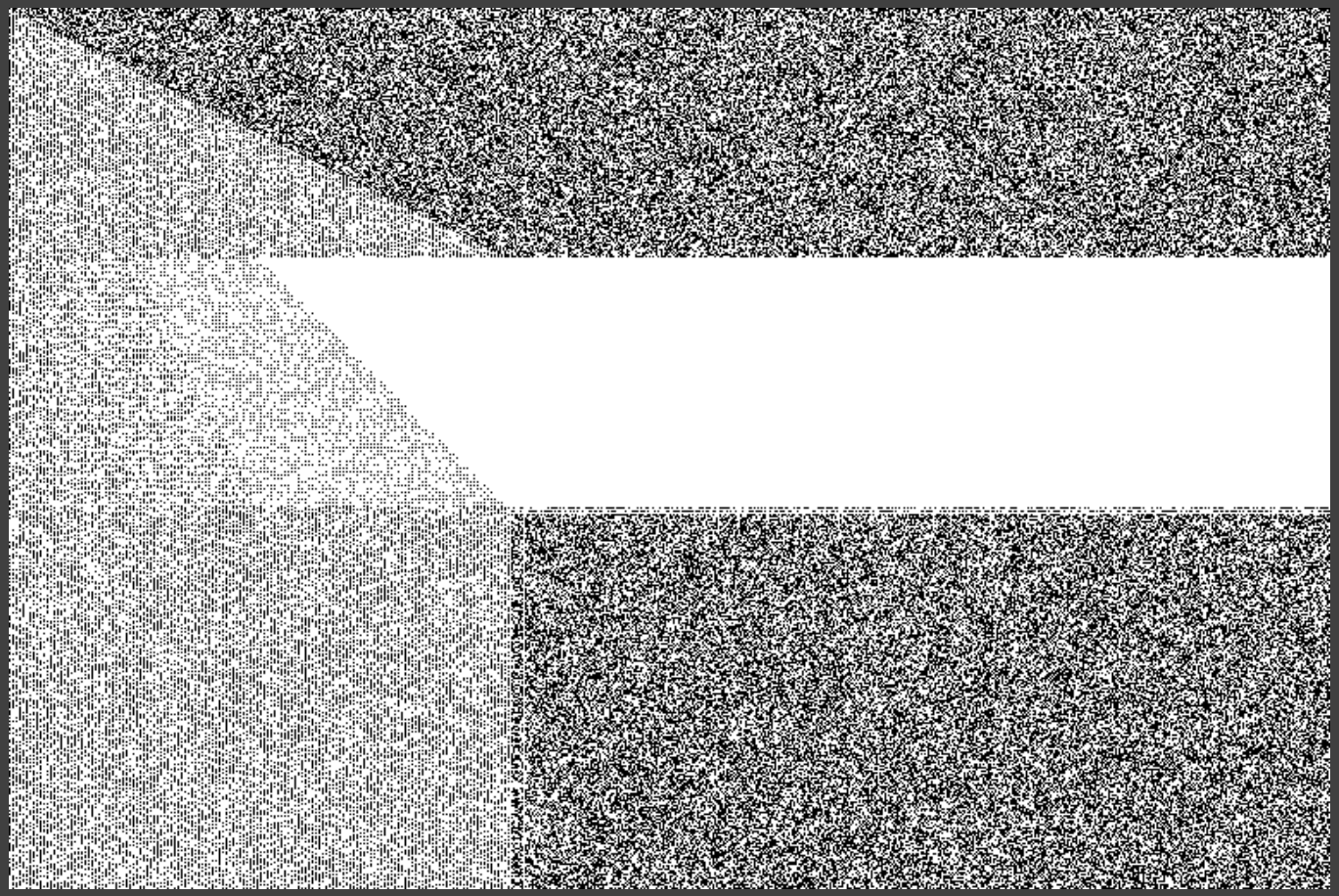
Visualisation



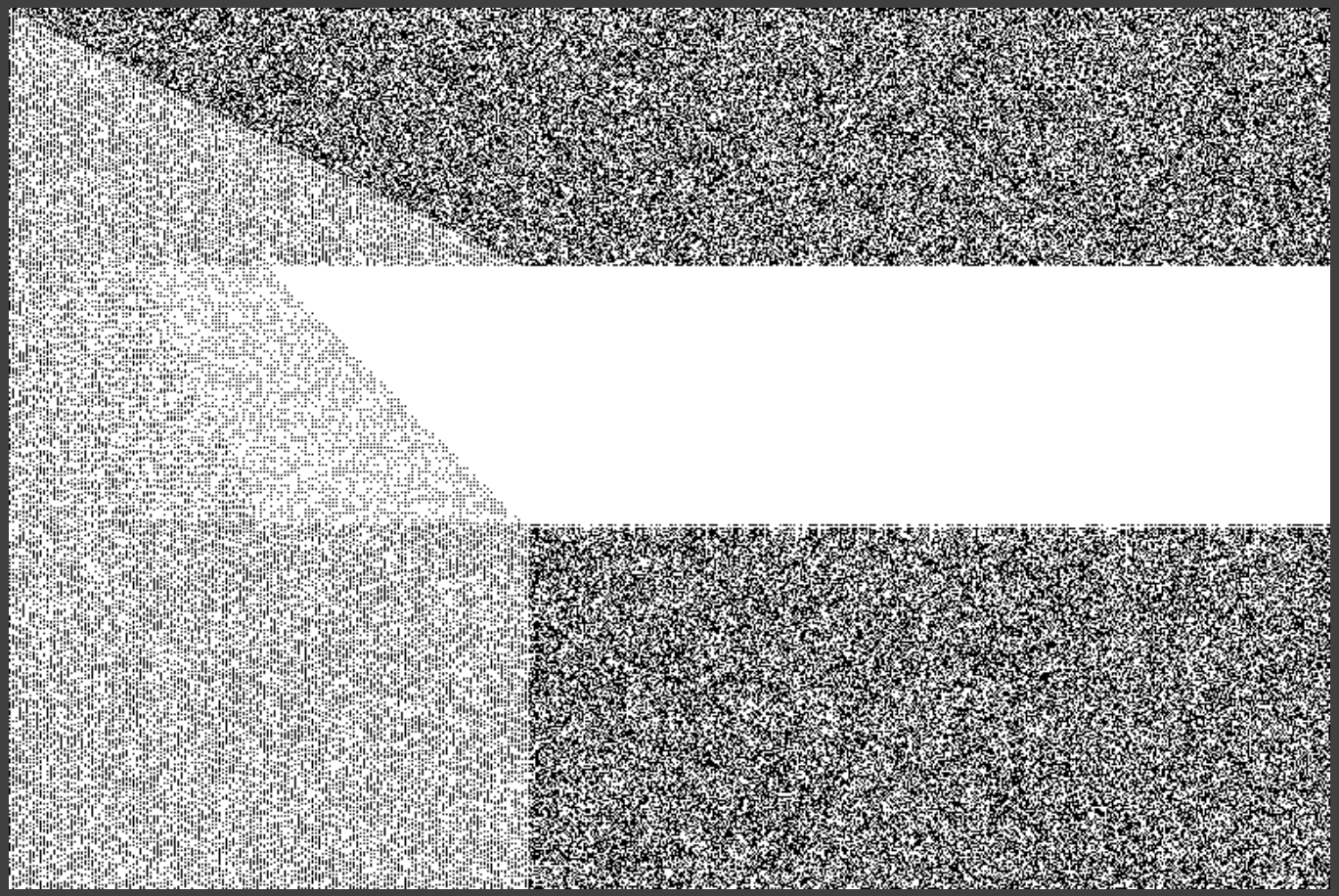
Visualisation



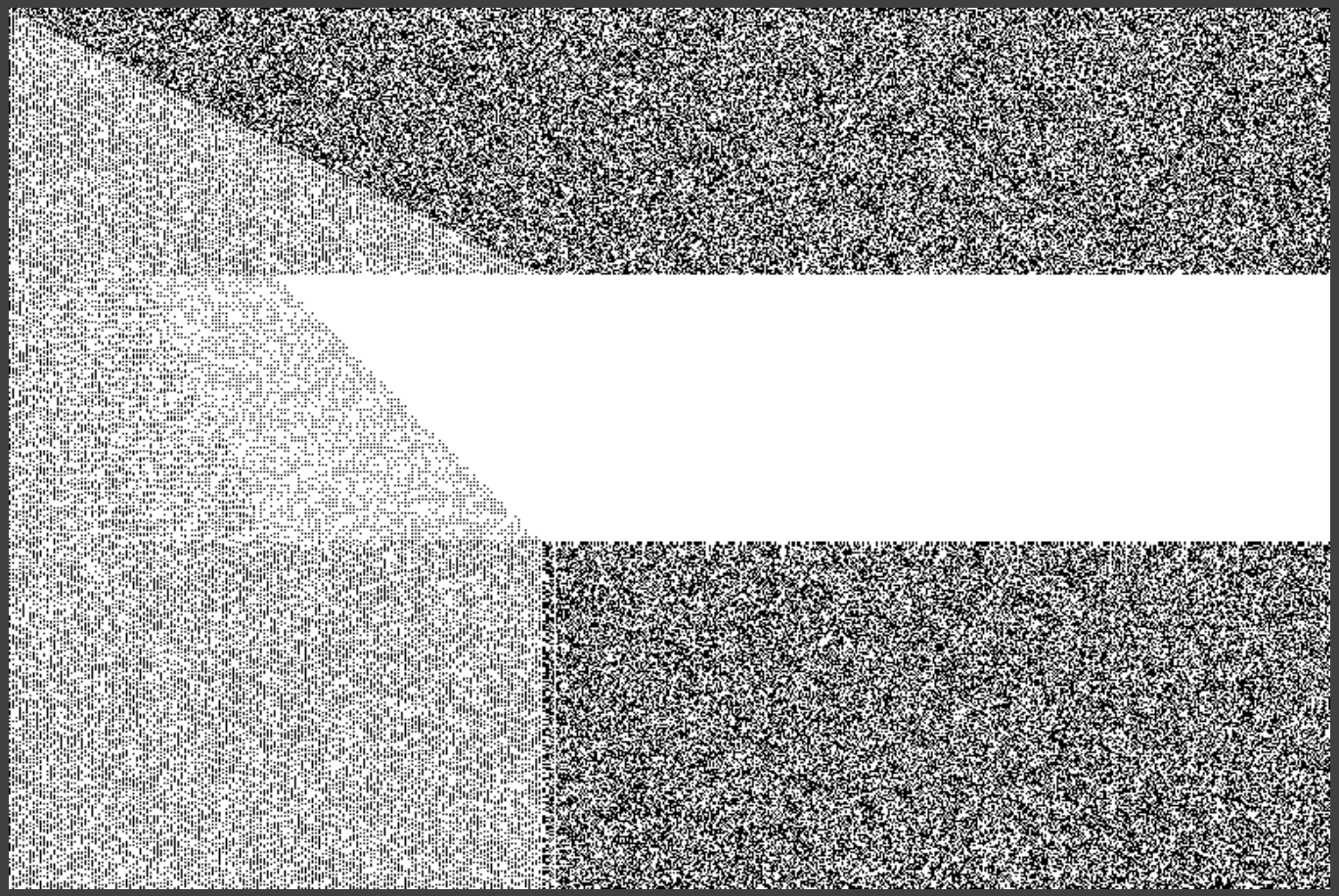
Visualisation



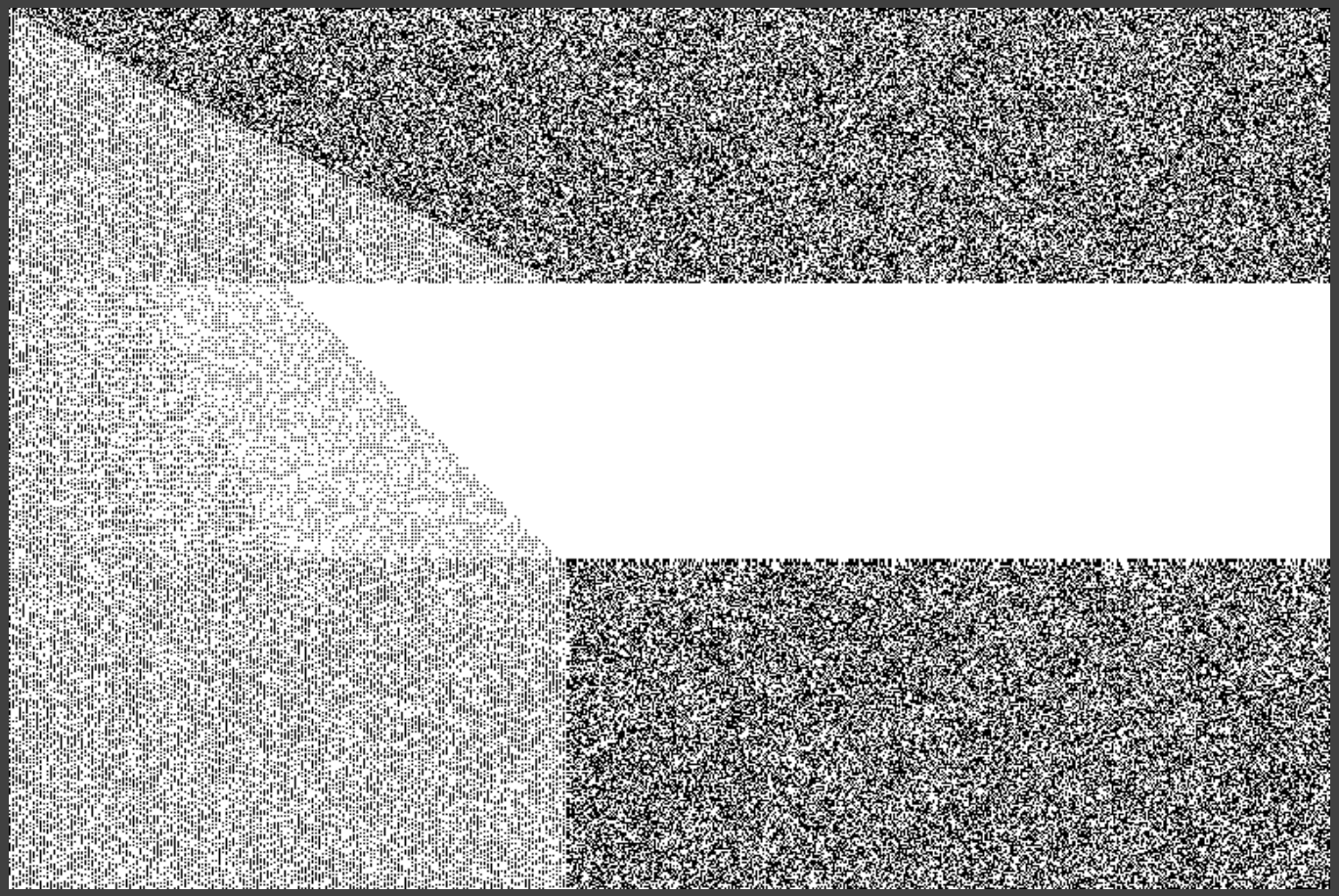
Visualisation



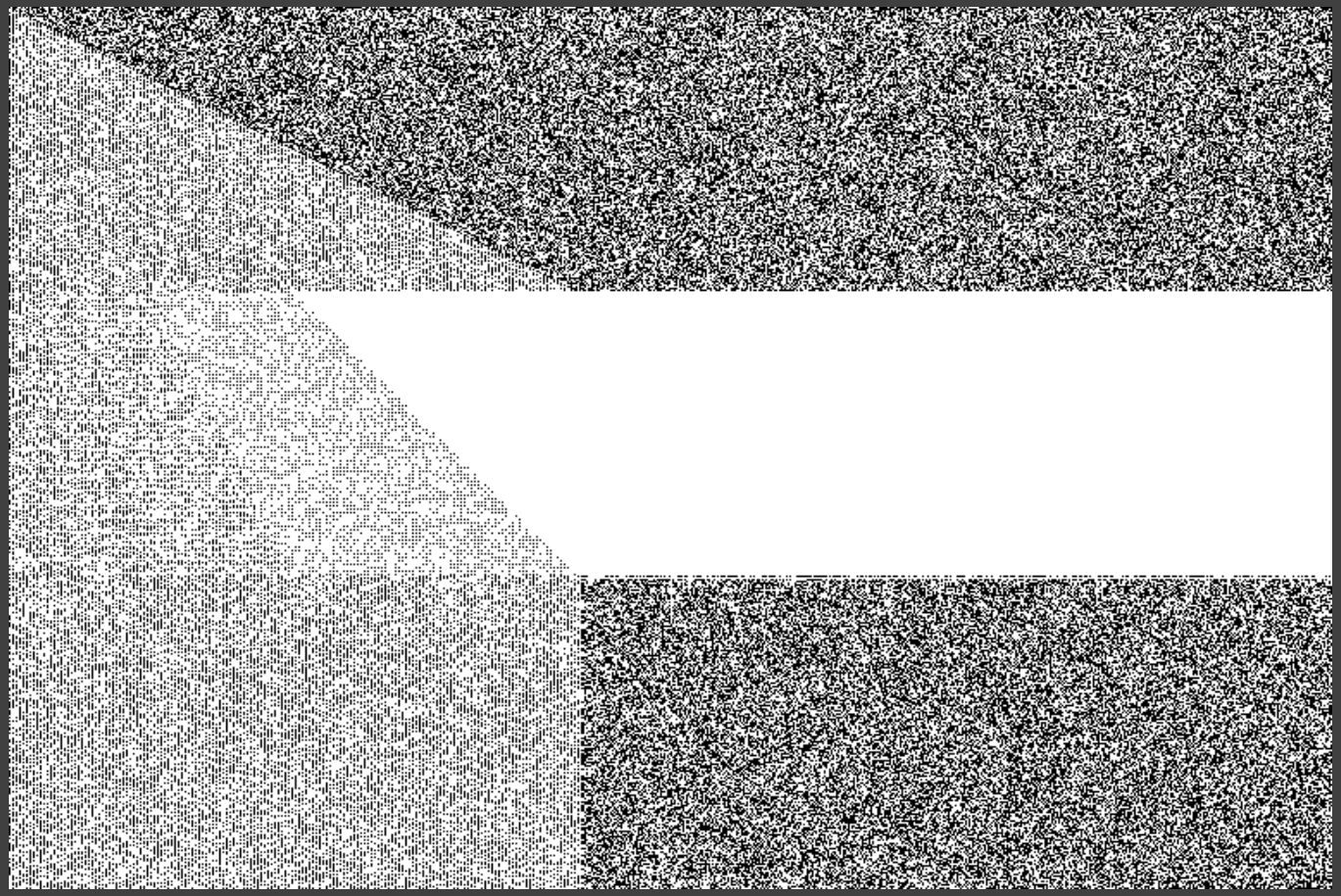
Visualisation



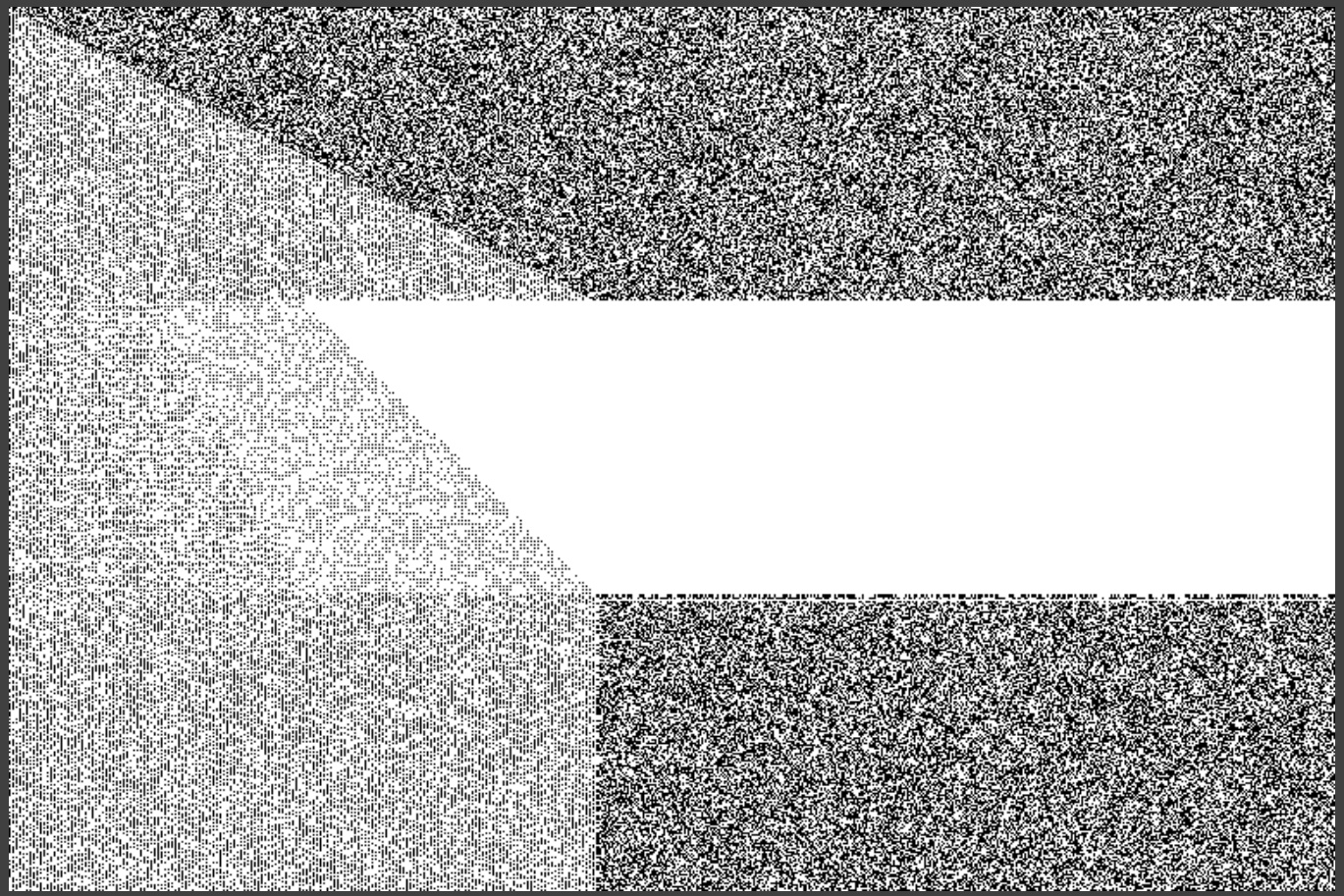
Visualisation



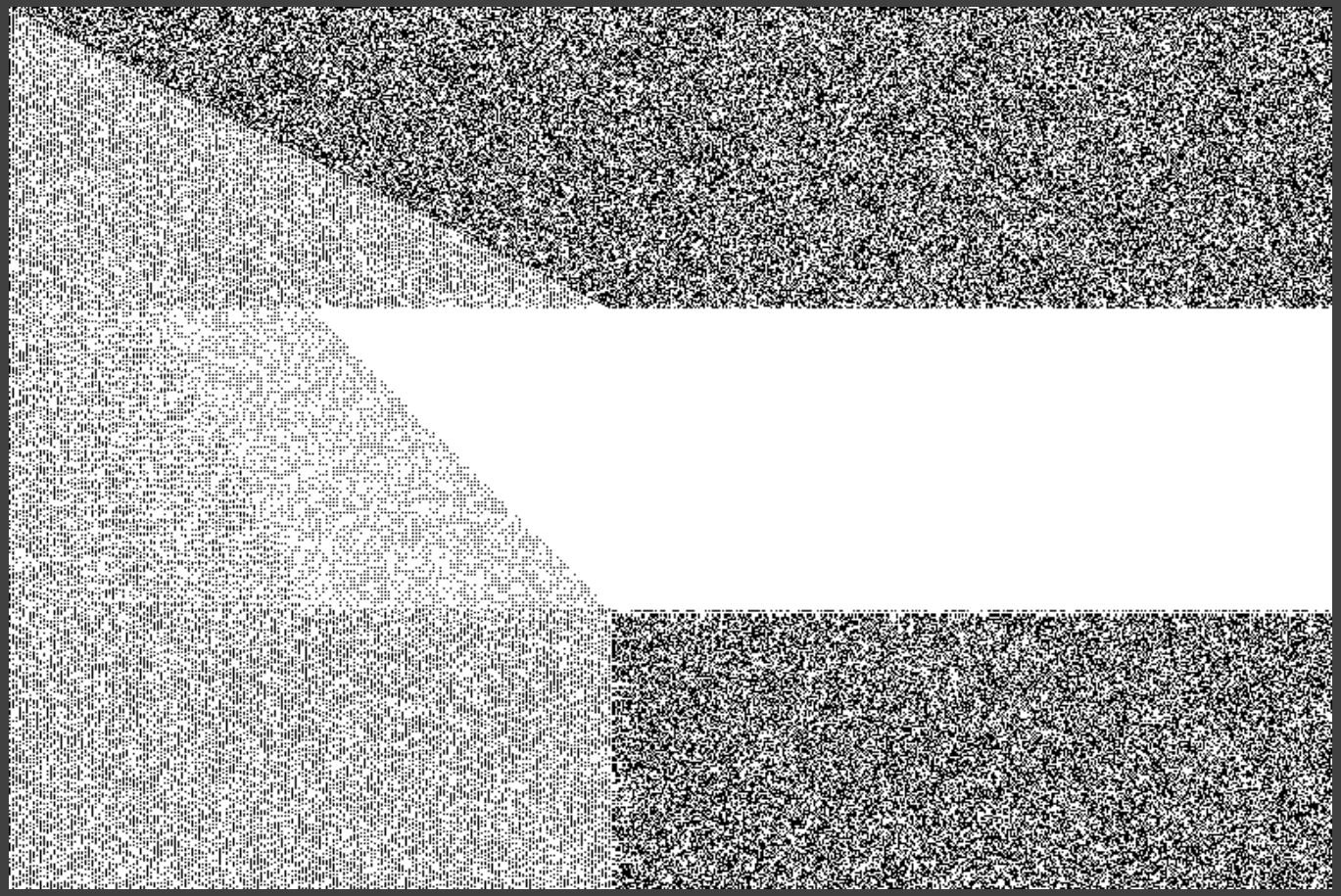
Visualisation



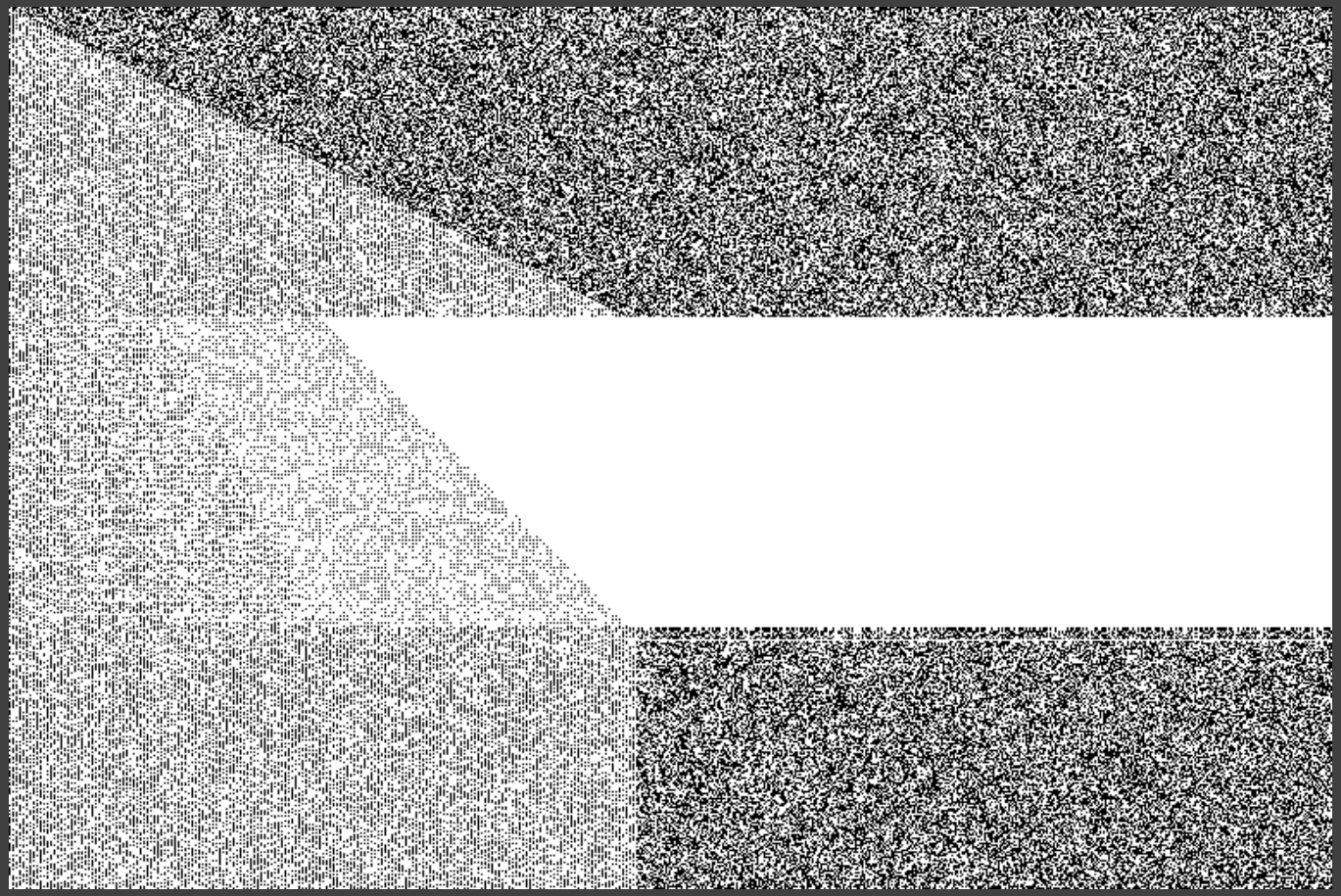
Visualisation



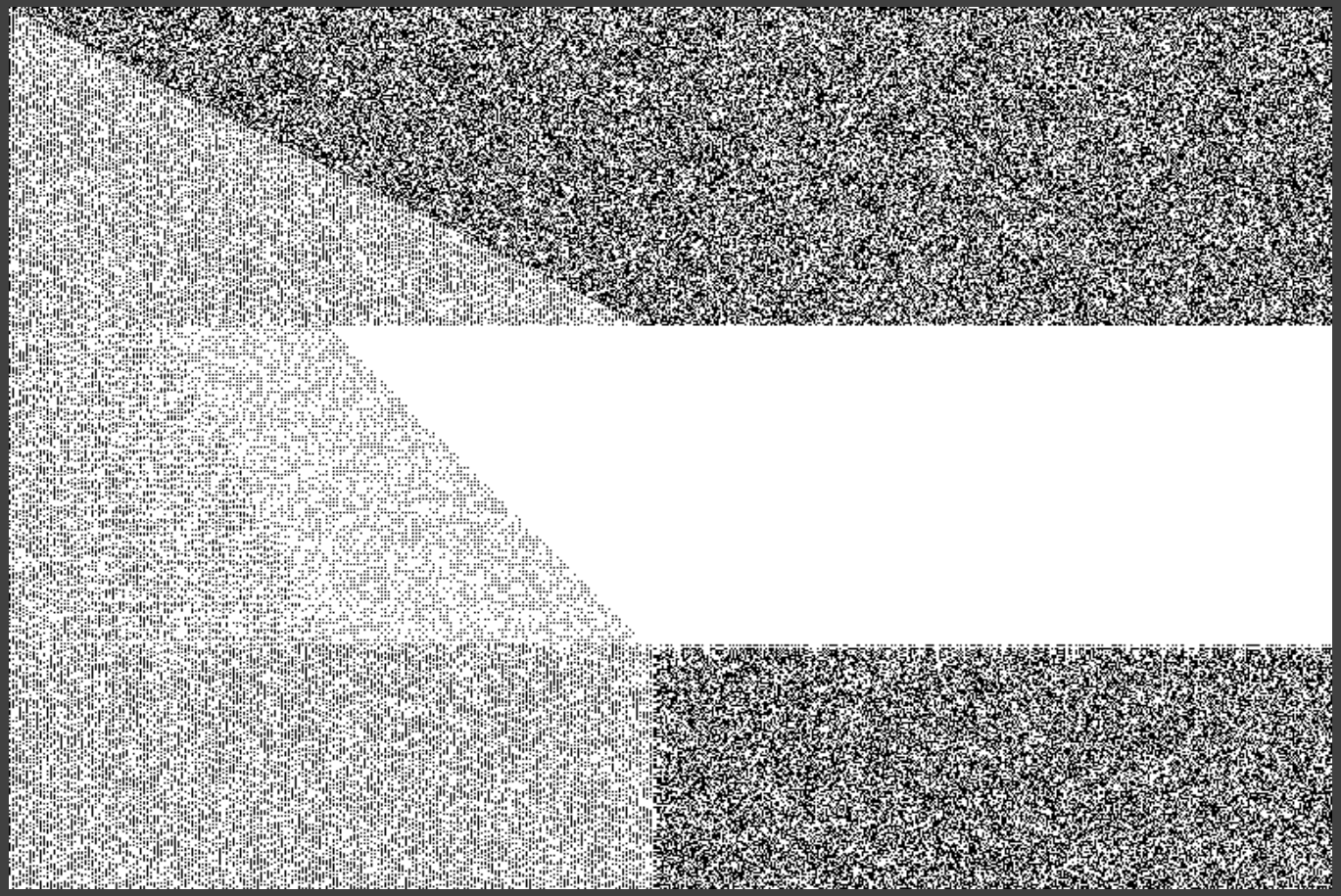
Visualisation



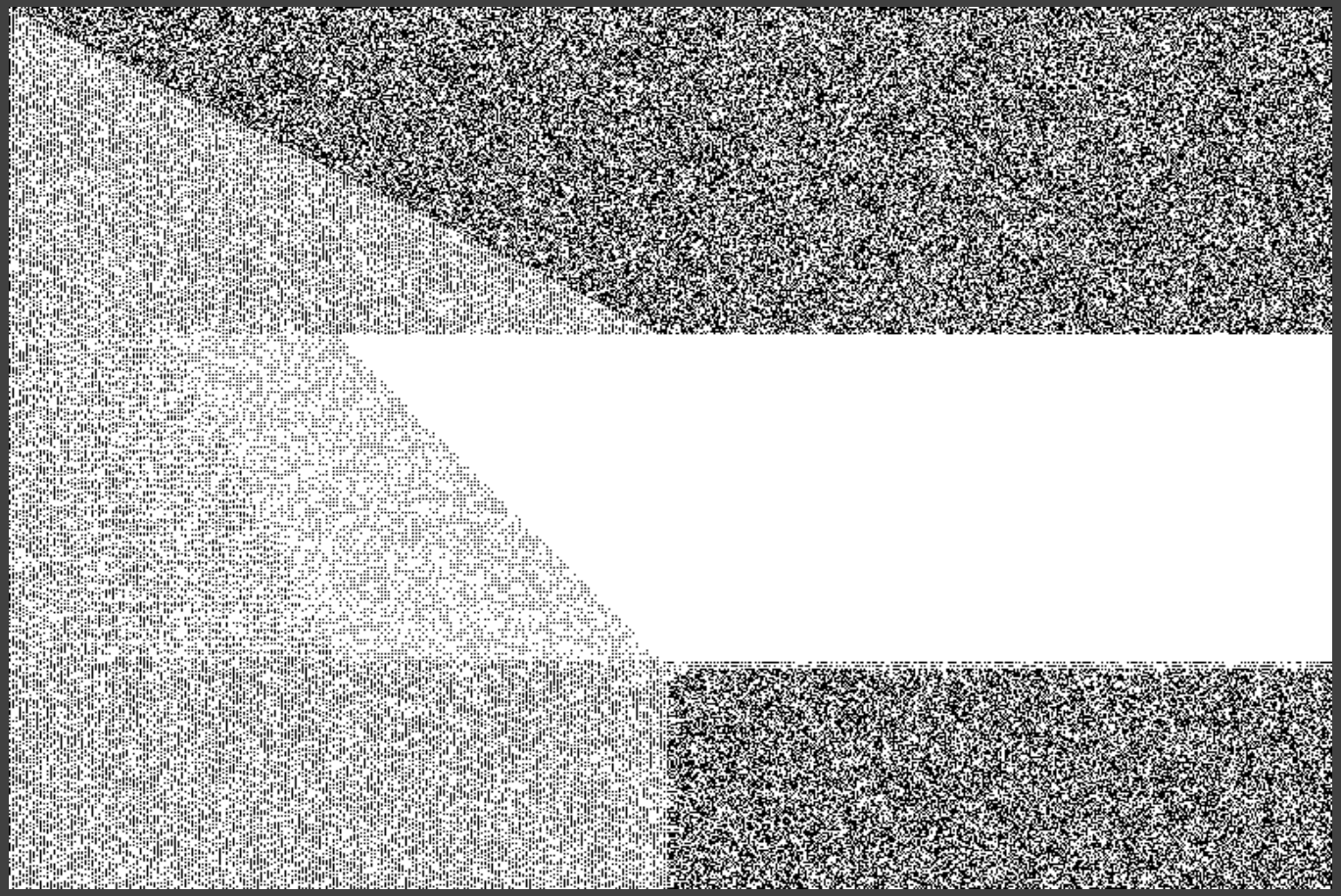
Visualisation



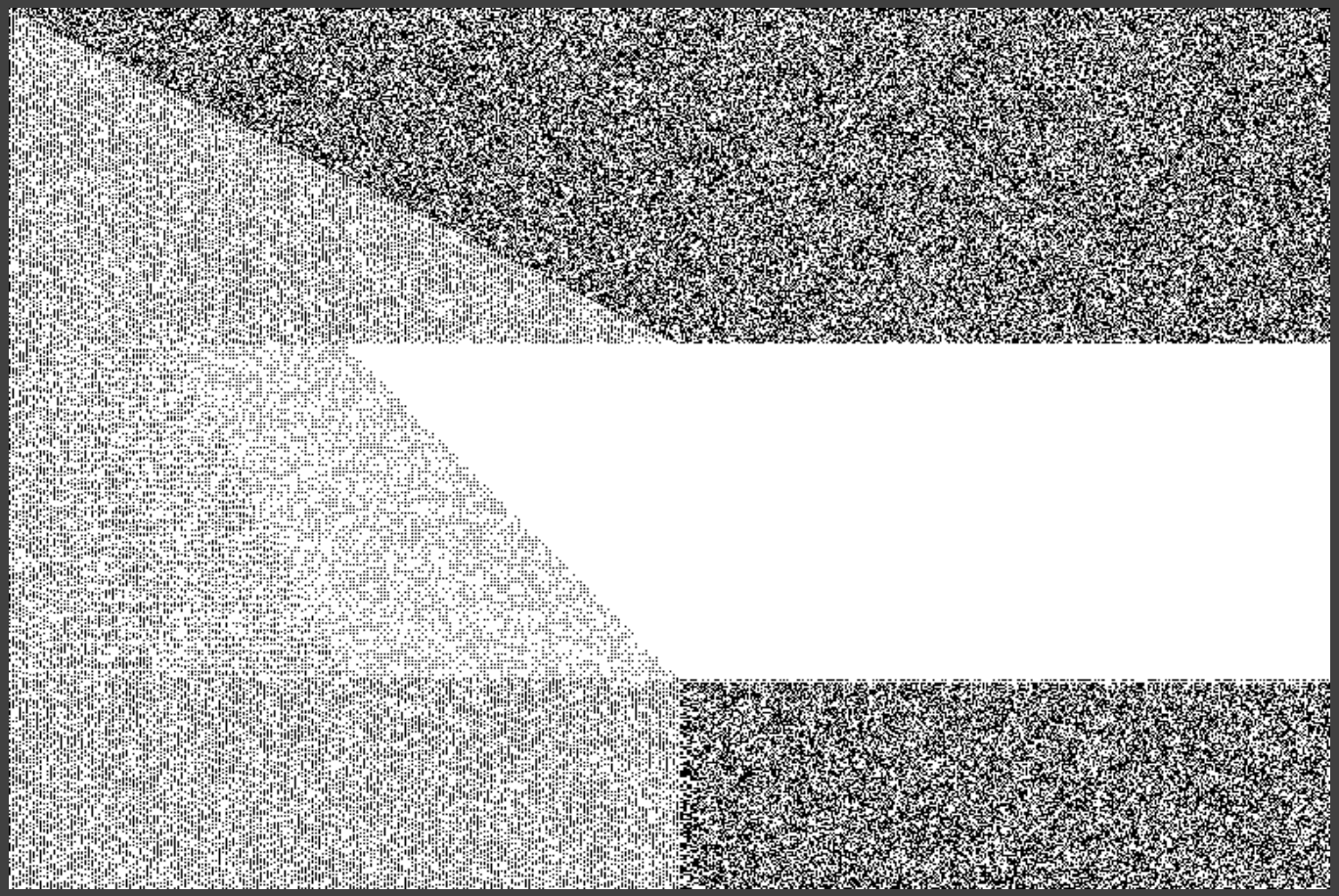
Visualisation



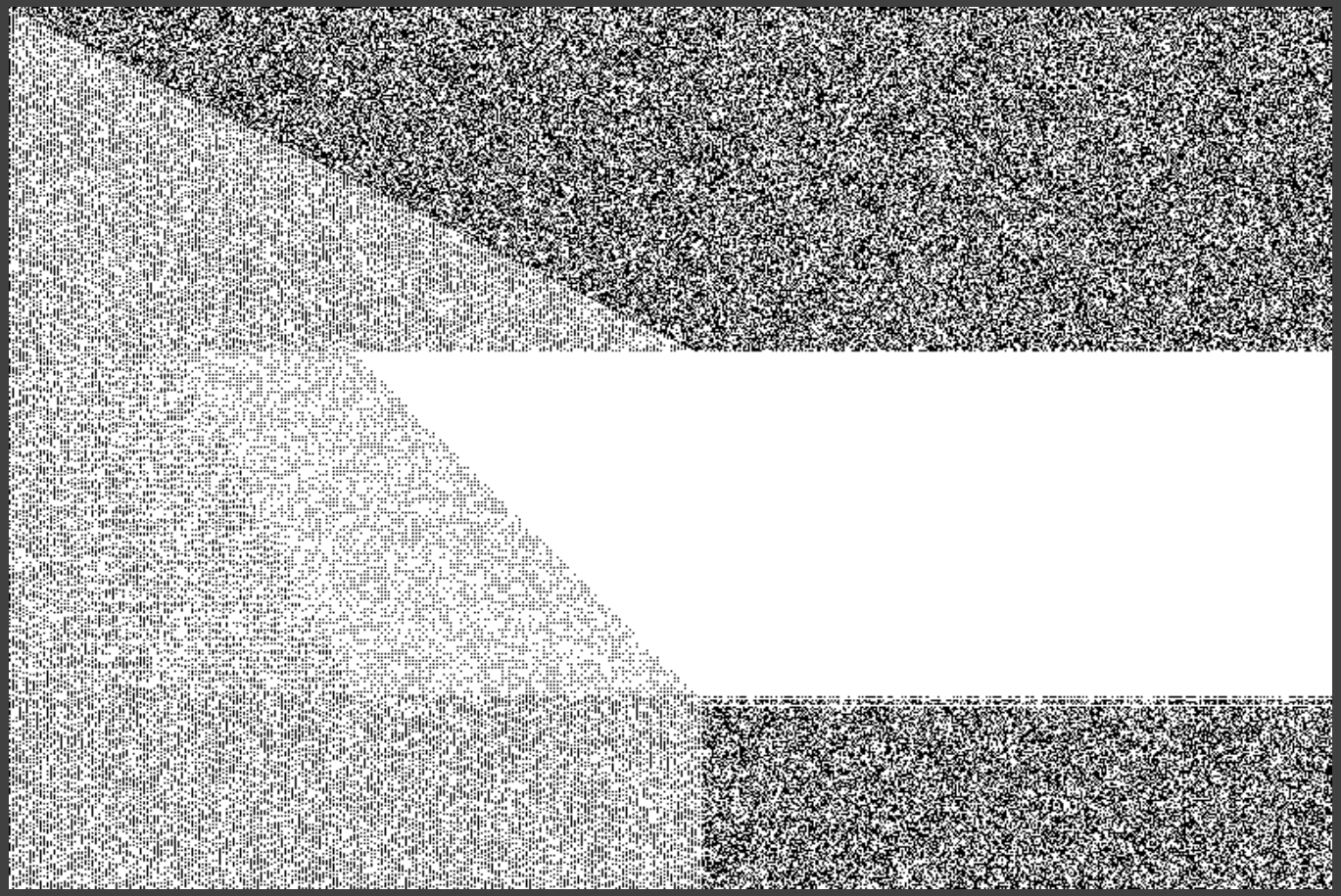
Visualisation



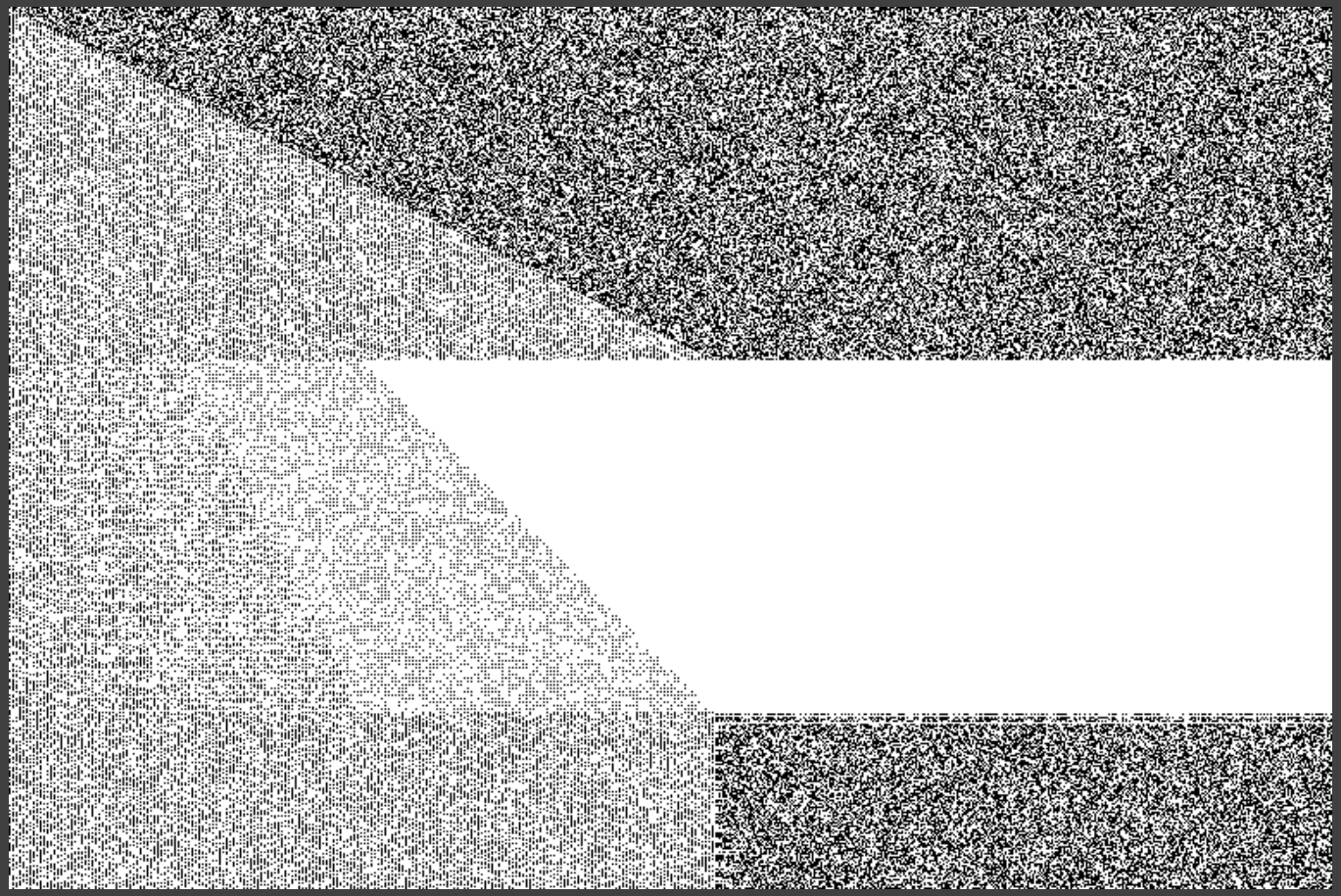
Visualisation



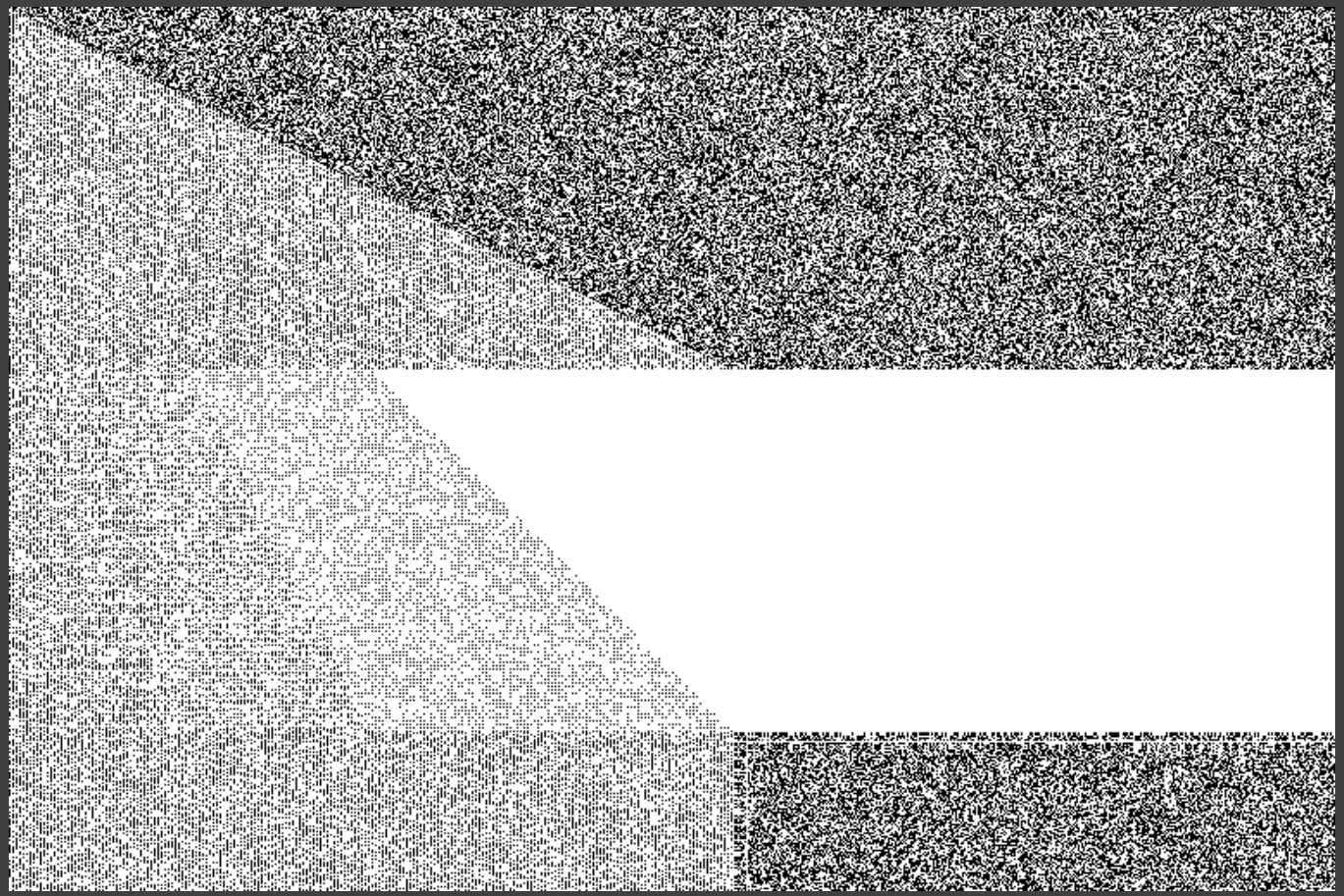
Visualisation



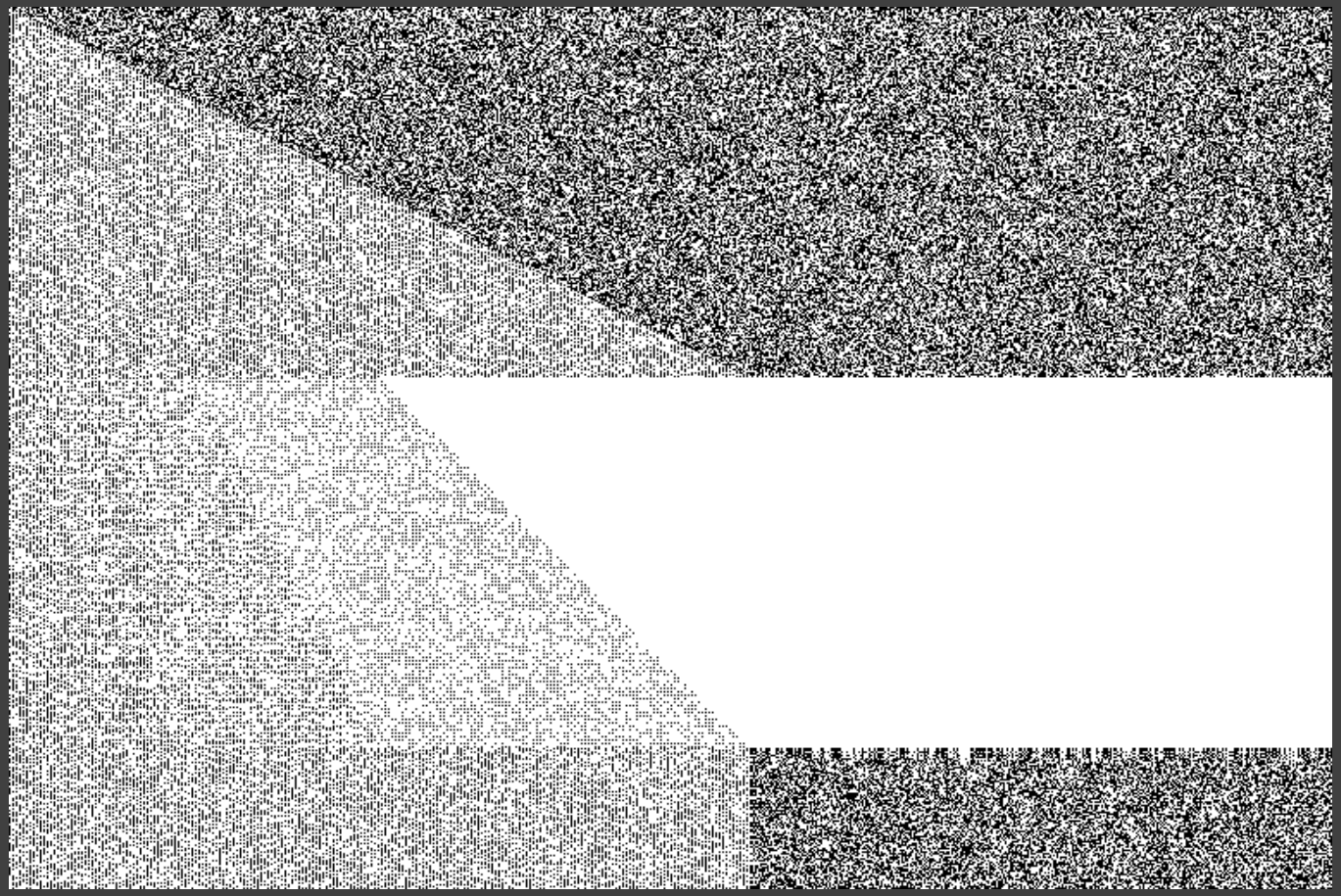
Visualisation



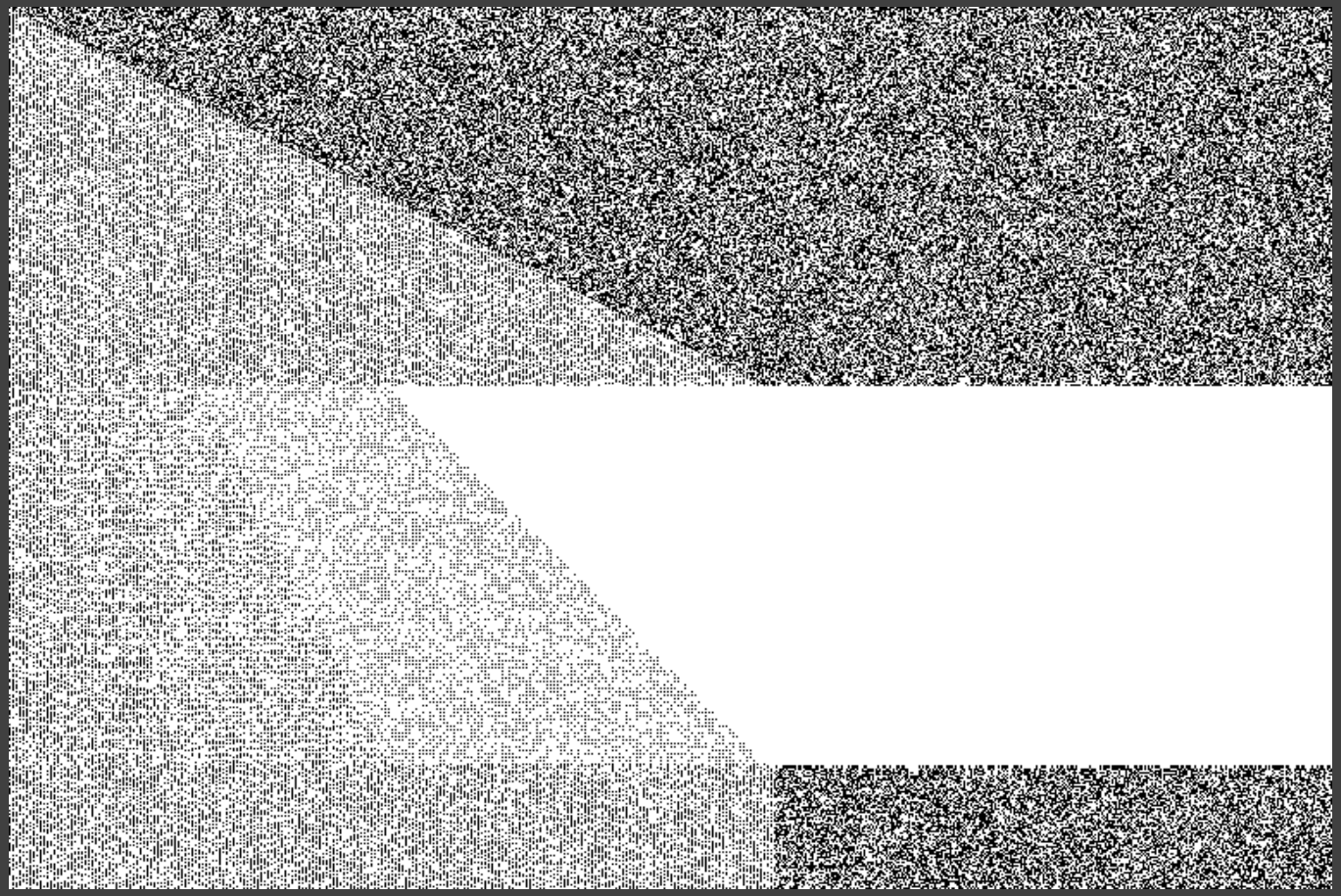
Visualisation



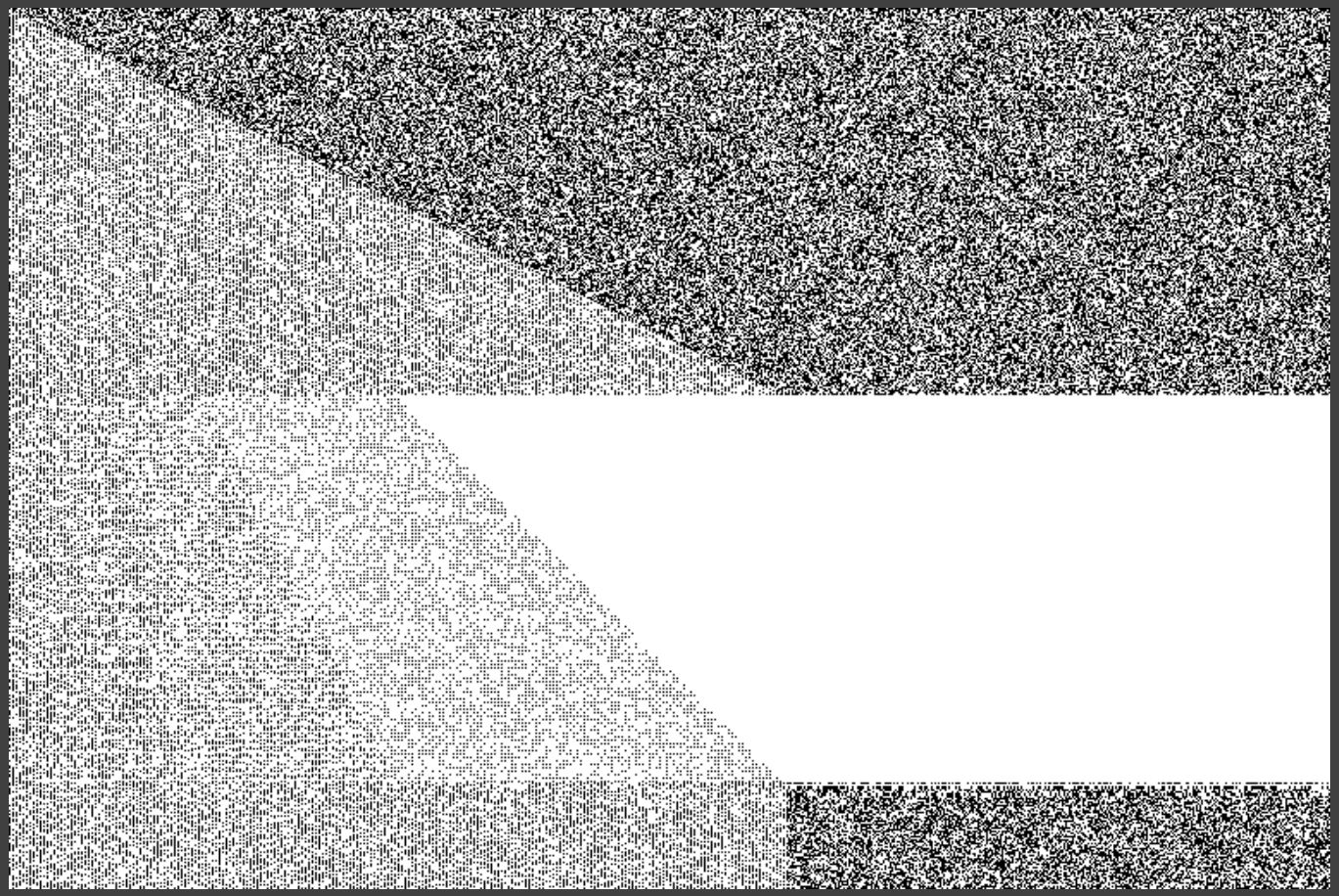
Visualisation



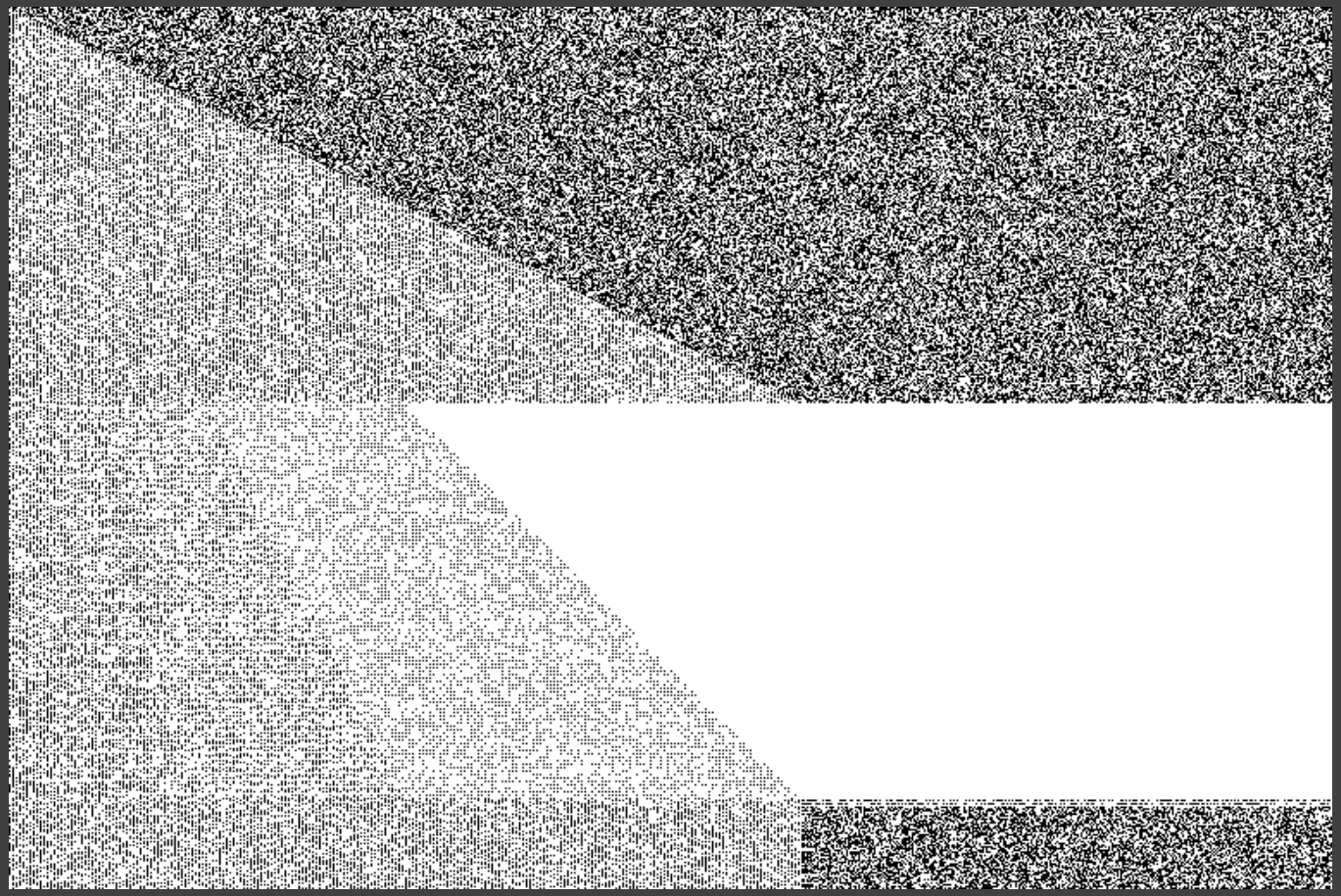
Visualisation



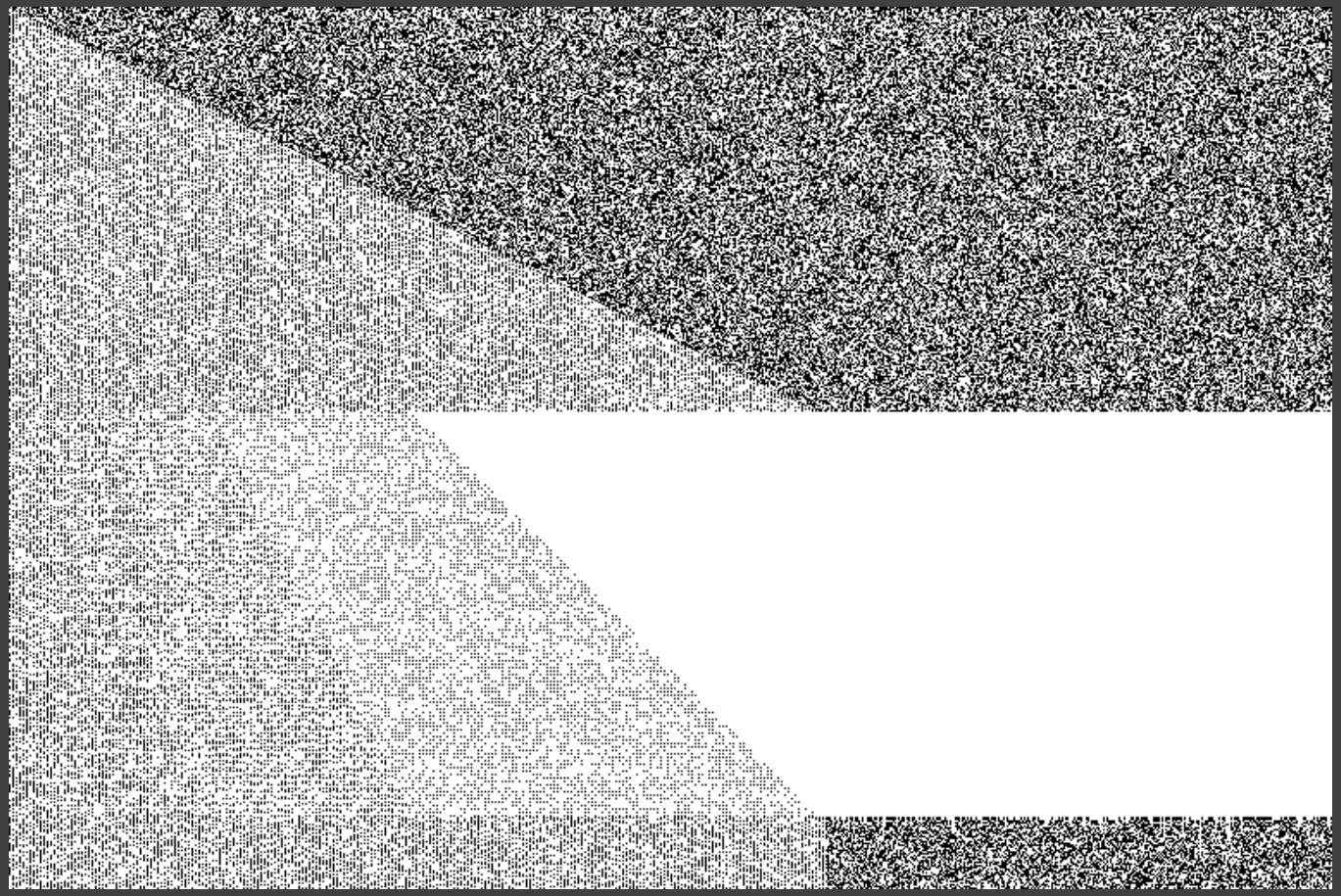
Visualisation



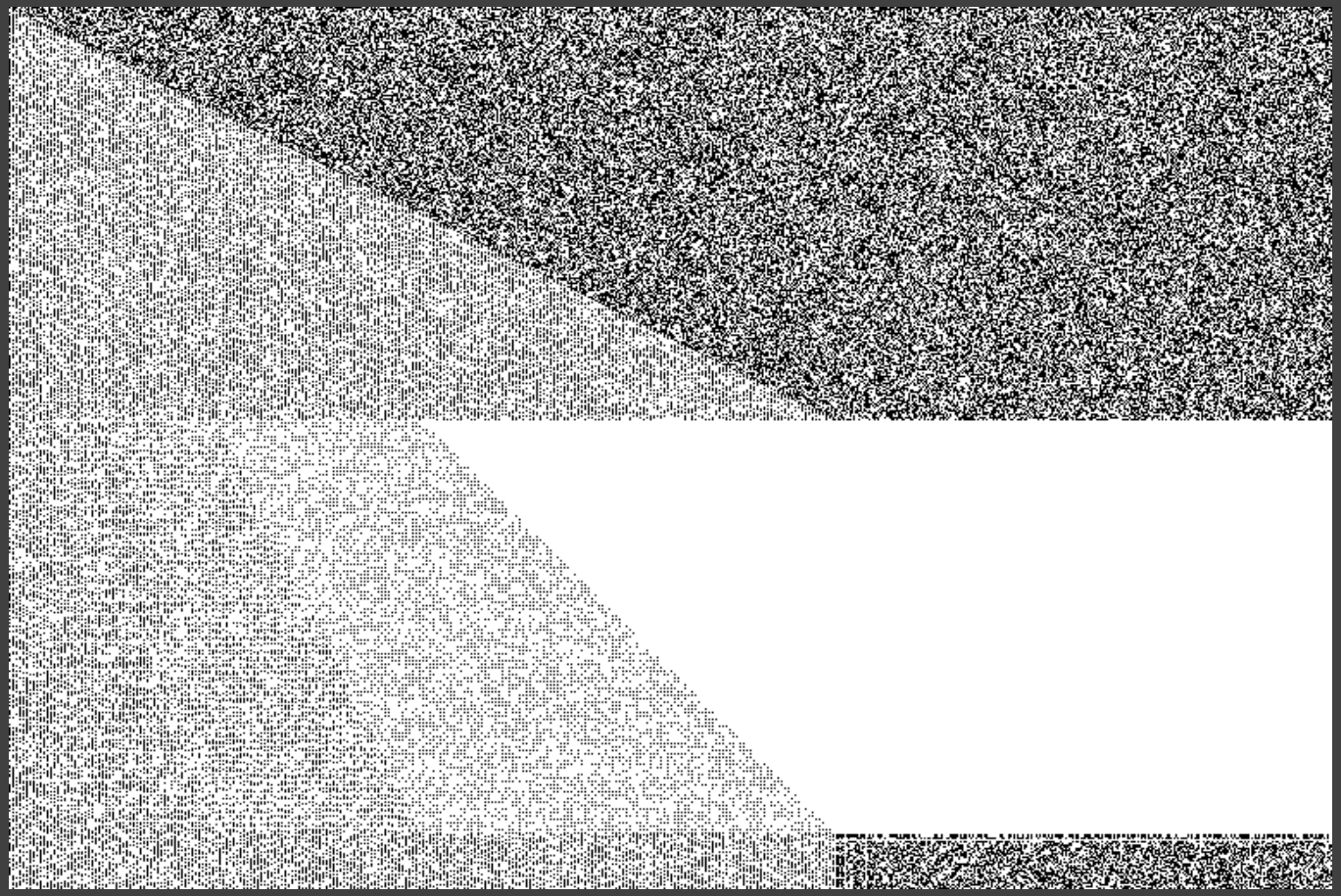
Visualisation



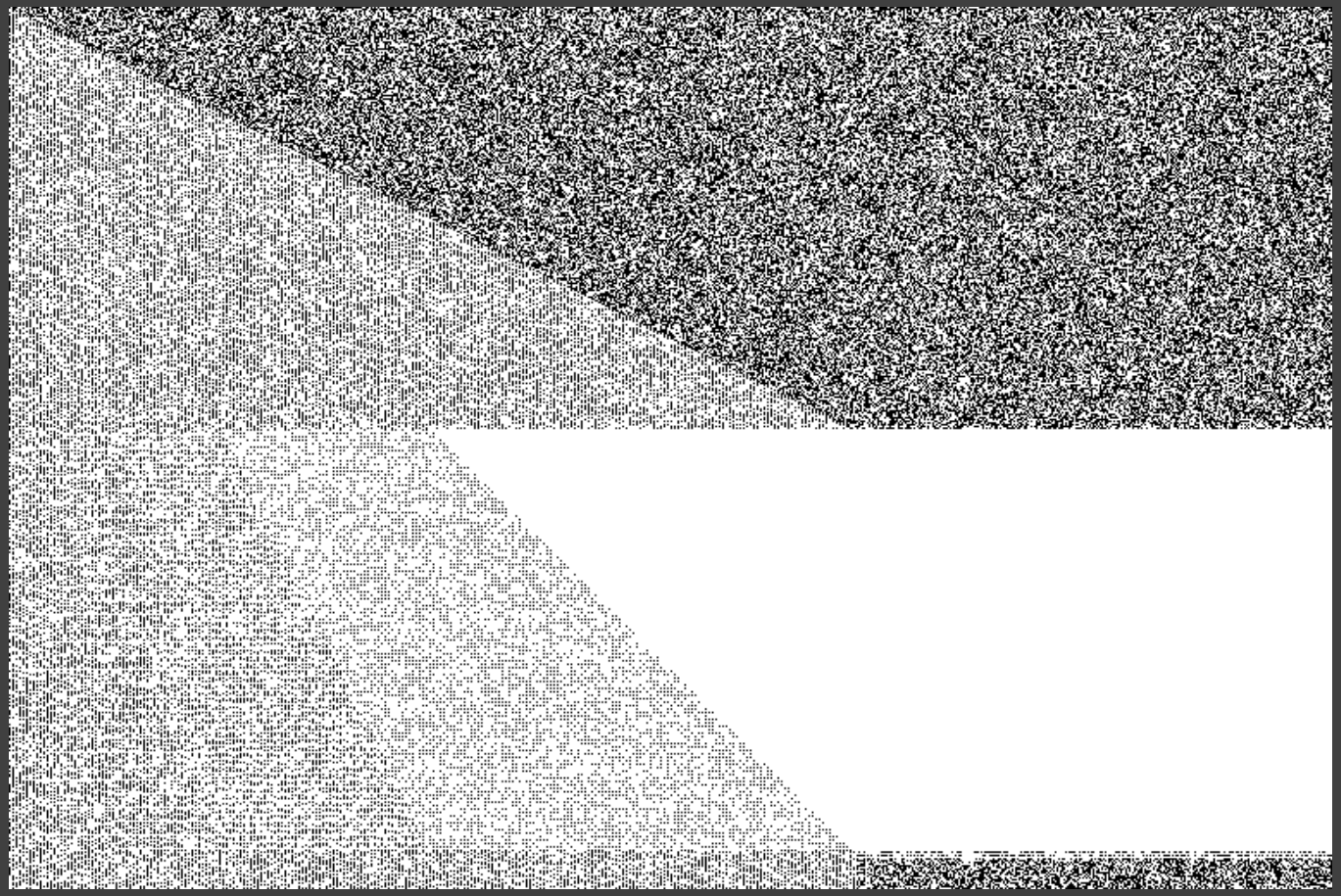
Visualisation



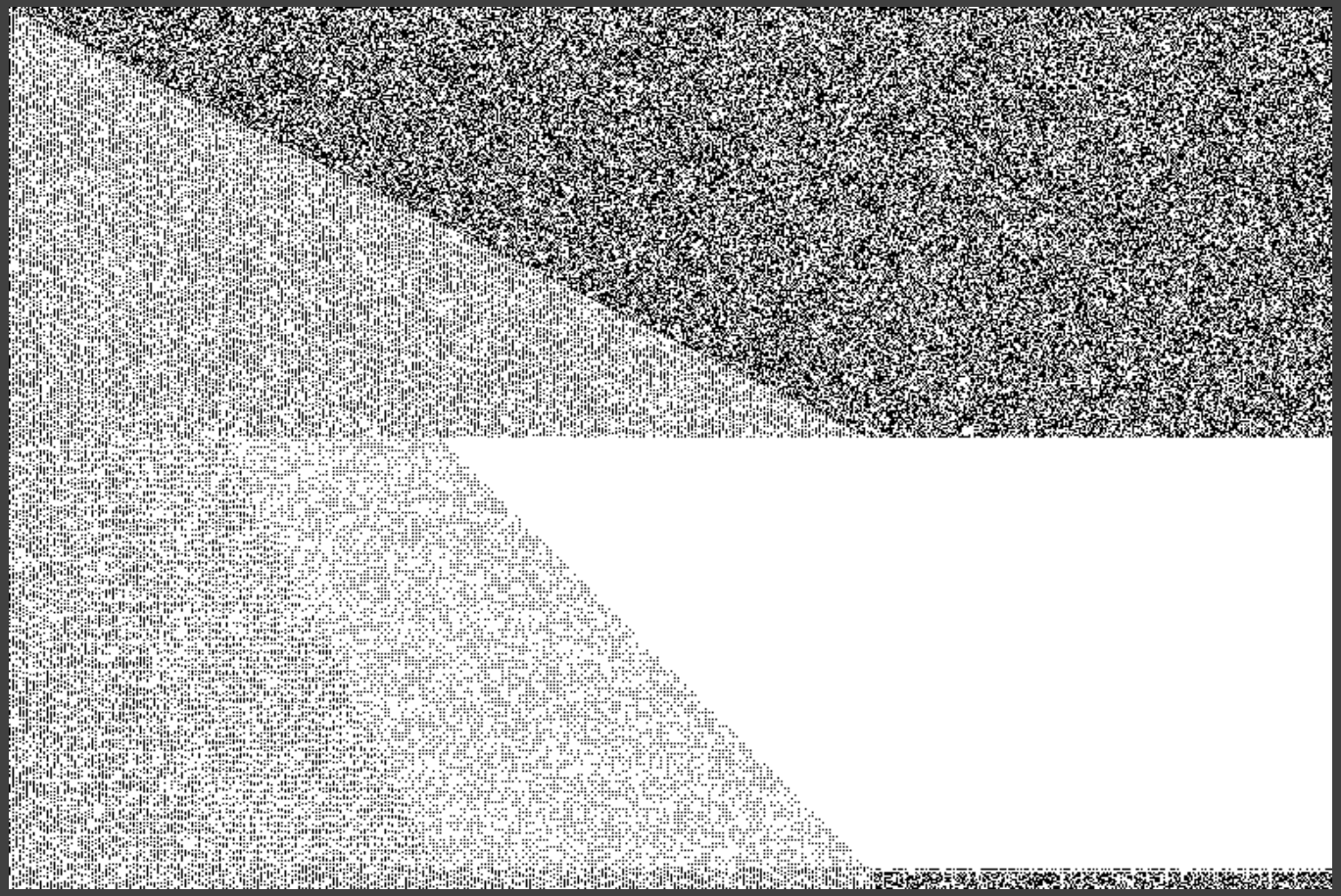
Visualisation



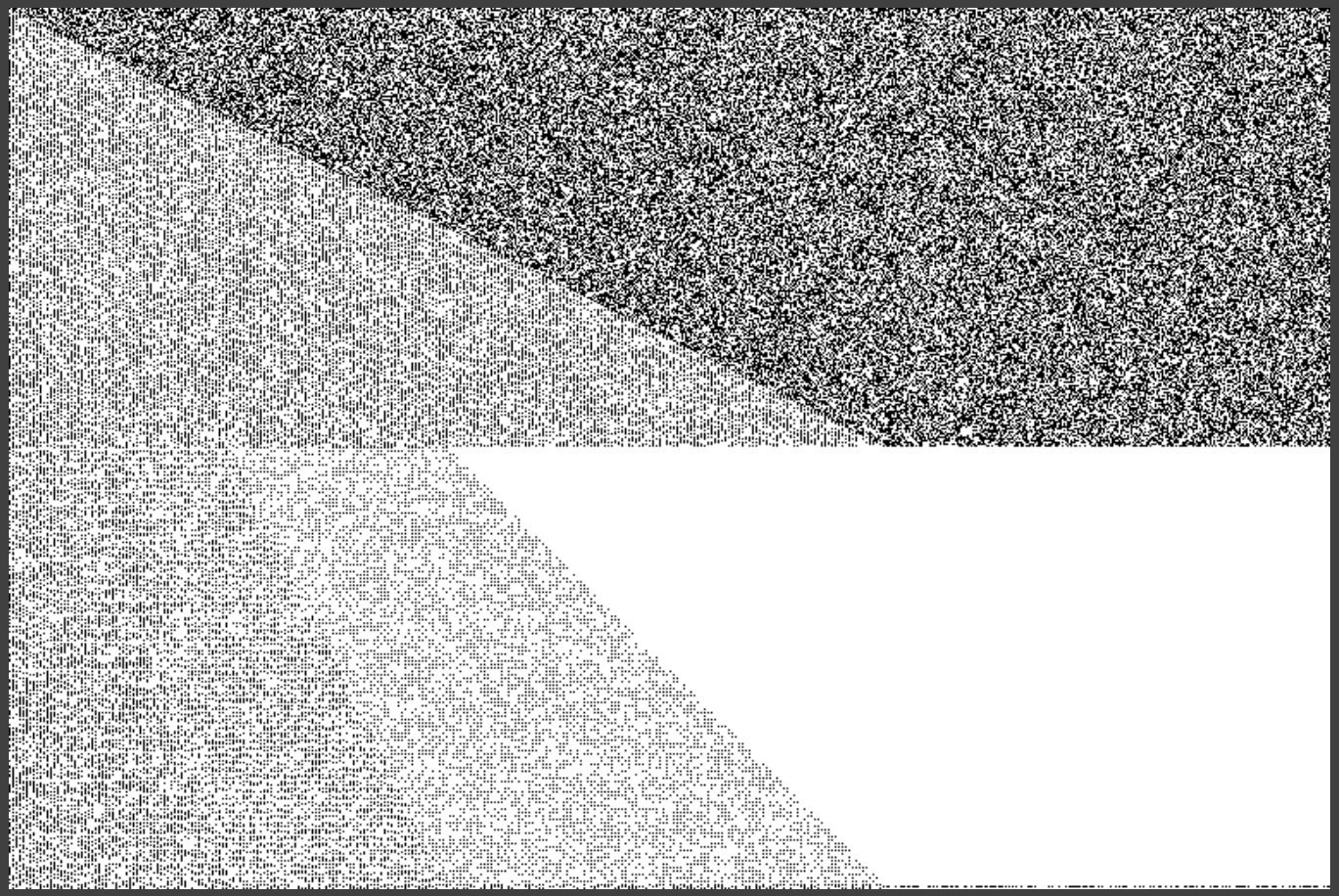
Visualisation



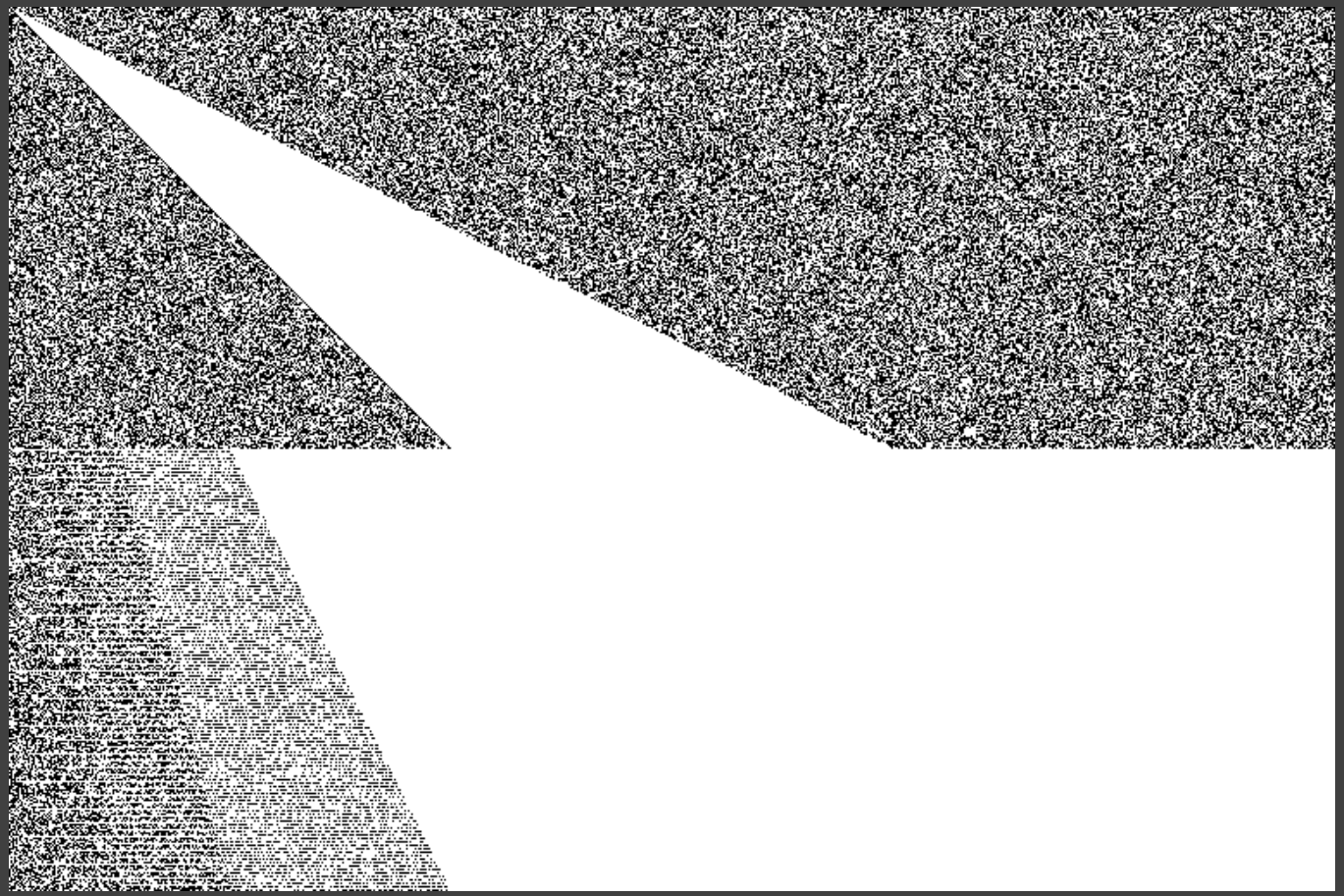
Visualisation



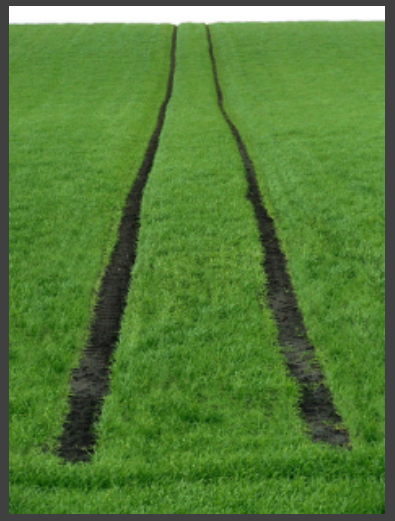
Visualisation



Visualisation



Outline



Implementation Issues I

Column swaps are used

1. to apply matrix-matrix multiplication and triangular system solving and
2. to achieve rank sensitivity.

However, column swaps over $\mathbb{F}_2$ in row major representation are quite expensive. While PLS performs less column swaps than PLUQ, we still must compress L which can be quite expensive.

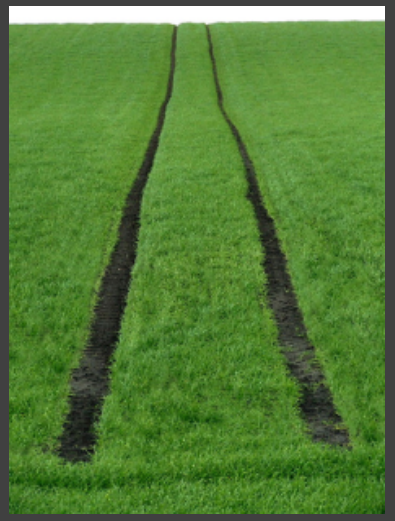
Implementation Issues II

Search for pivots can be expensive since it is cache unfriendly and requires bit-level access.

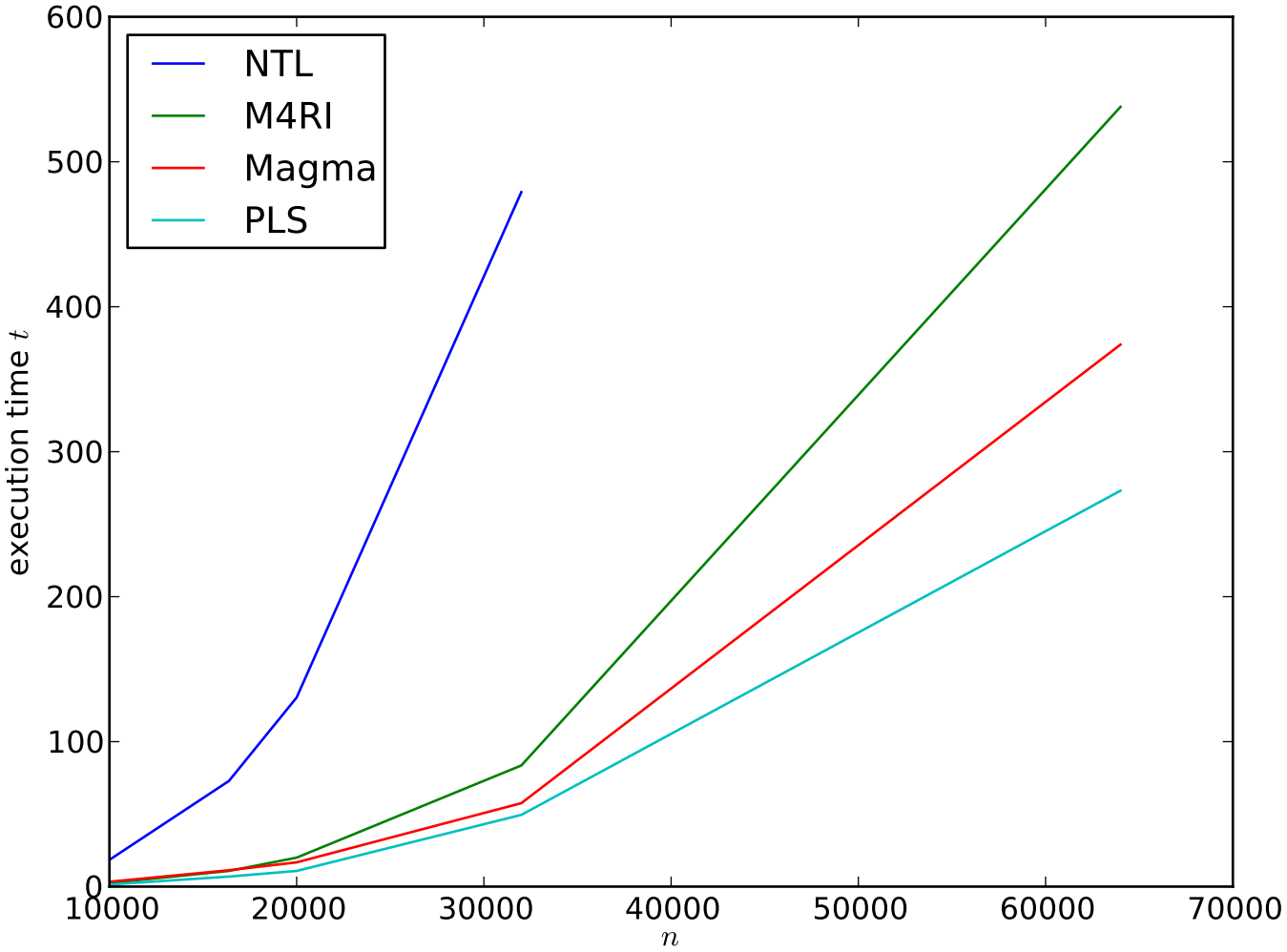
Implementation Issues III

We switch from asymptotically fast PLS to MMPF when the submatrix fits into L2 cache.

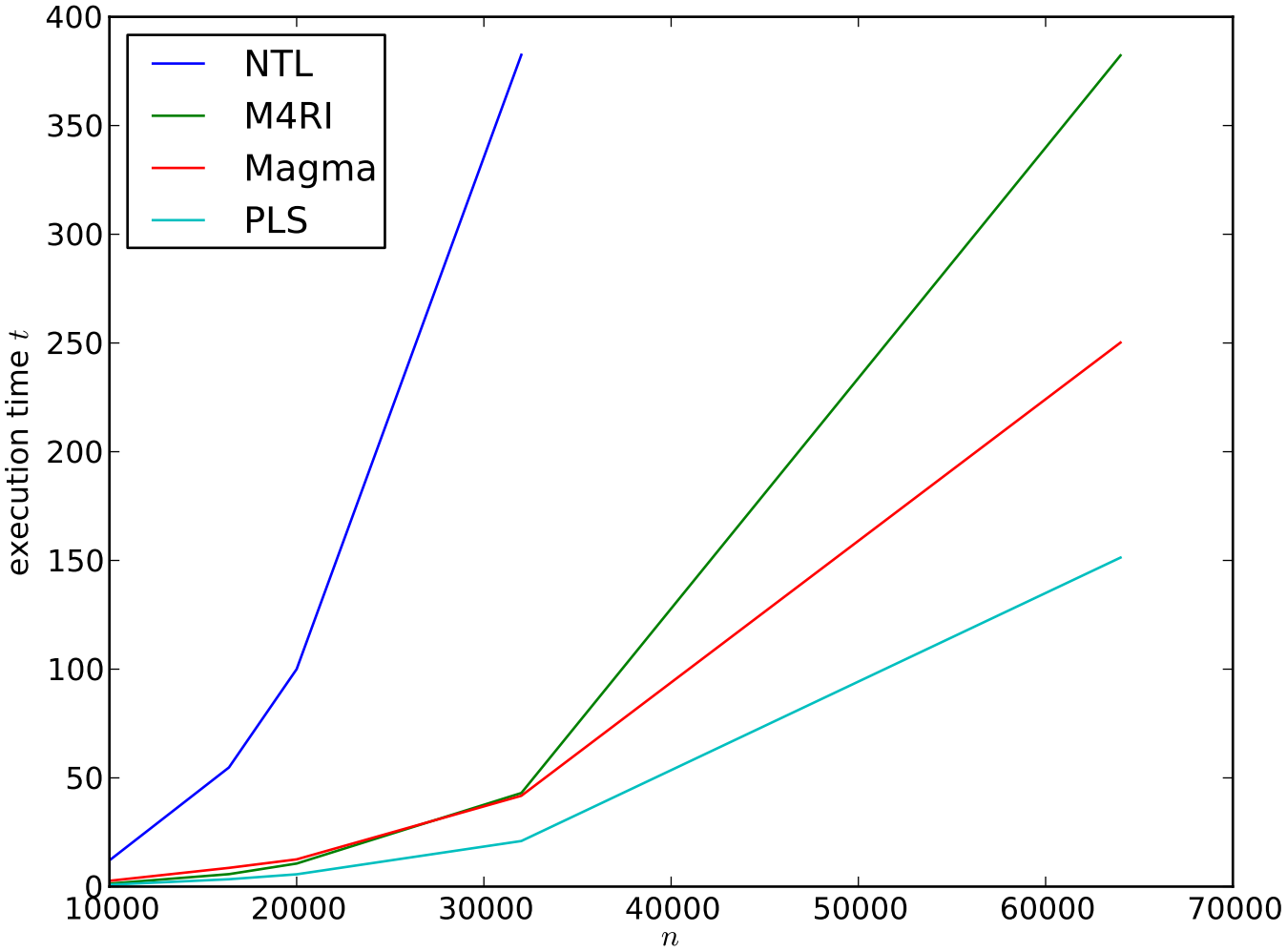
Outline



RREF for Dense Random: 2.6Ghz Opteron



RREF for Dense Random: 2.33Ghz Xeon



RREF for Dense Random: 233Ghz Xeon

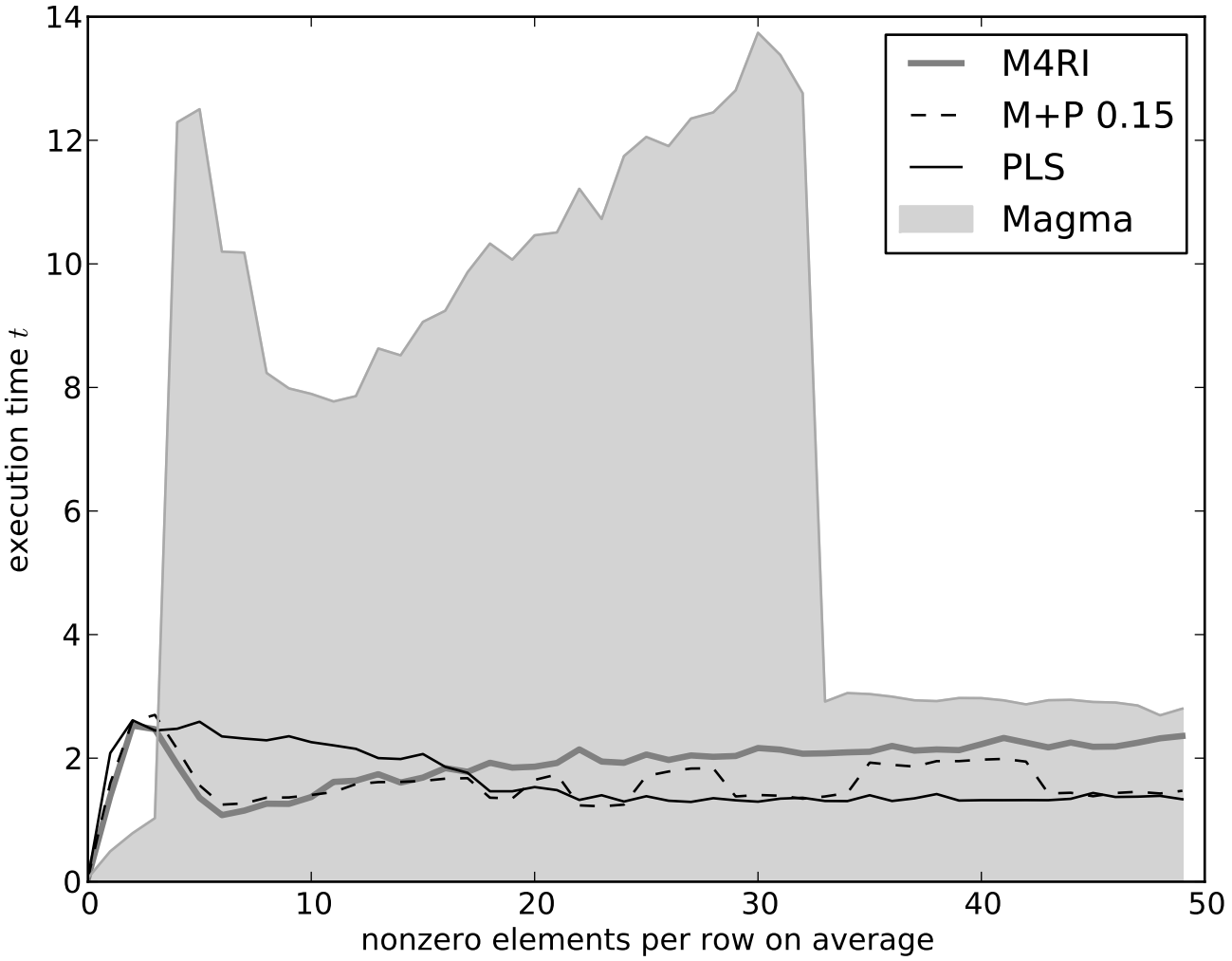
Using one core we can compute the echelon form of a $500,000 \times 500,000$ dense random matrix over $\mathbb{F}_2$ in

$$9711.42 \text{ seconds } = 2.7 \text{ hours.}$$

Using four cores decomposition we can compute the echelon form of a random dense $500,000 \times 500,000$ matrix in

$$3806.28 \text{ seconds } = 1.05 \text{ hours.}$$

Sensitivity to Sparsity



Examples from Practice

			64-bit Fedora Linux, 2.33Ghz Xeon (eno)				
Problem	Matrix Dimension	Density	Magma 2.16-7	M4RI 20100324	PLS 20100324	M+P 0.15 20100429	M+P 0.20 20100429
HFE 25	$12,307 \times 13,508$	0.076	3.68s	**1.94s**	2.09s	2.33s	2.24s
HFE 30	$19,907 \times 29,323$	0.067	23.39s	**11.46s**	13.34s	12.60s	13.00s
HFE 35	$29,969 \times 55,800$	0.059	–	**49.19s**	68.85s	66.66s	54.42s
MXL	$26,075 \times 26,407$	0.185	55.15	12.25s	**9.22s**	**9.22s**	10.22s

			64-bit Ubuntu Linux, 2.66Ghz Xeon (sage.math)				
Problem	Matrix Dimension	Density		M4RI 20100324	PLS 20100324	M+P 0.15 20100429	M+P 0.20 20100429
HFE 25	$12,307 \times 13,508$	0.076		2.24s	**2.00s**	2.39s	2.35s
HFE 30	$19,907 \times 29,323$	0.067		27.52s	**13.29s**	13.78s	22.9s
HFE 35	$29,969 \times 55,800$	0.059		115.35s	**72.70s**	84.04s	122.65s
MXL	$26,075 \times 26,407$	0.185		26.61s	**8.73s**	8.75s	13.23s

			64-bit Debian/GNU Linux, 2.6Ghz Opteron)				
Problem	Matrix Dimension	Density	Magma 2.15-10	M4RI 20100324	PLS 20100324	M+P 0.15 20100429	M+P 0.20 20100429
HFE 25	$12,307 \times 13,508$	0.076	4.57s	3.28s	3.45s	**3.03s**	3.21s
HFE 30	$19,907 \times 29,323$	0.067	33.21s	**23.72s**	25.42s	23.84s	25.09s
HFE 35	$29,969 \times 55,800$	0.059	278.58s	126.08s	159.72s	154.62s	**119.44s**
MXL	$26,075 \times 26,407$	0.185	76.81s	23.03s	19.04s	**17.91s**	18.00s

Fin

Martin Albrecht and Gregory V. Bard.
The M4RI Library – Version 20091104.
The M4RI Team, 2009.
http://m4ri.sagemath.org.

Martin Albrecht, Gregory V. Bard, and William Hart.
Algorithm 898: Efficient multiplication of dense matrices over GF(2).
ACM Transactions on Mathematical Software, 37(1), 2009.
pre-print available at
http://arxiv.org/abs/0811.1714.

Gregory V. Bard.
Accelerating Cryptanalysis with the Method of Four Russians.
Cryptology ePrint Archive, Report 2006/251, 2006.
Available at http://eprint.iacr.org/2006/251.pdf.