The following numbers are used:

NC – number of chambers;

NA – number of assays;

MA – assay multiplicity;

Ci – chamber *i*;

Aj – assay *j*;

Assay allocation can be presented as a table, where column is a chamber and row is an assay.

Here is an example of allocation for NC=8, NA=5, and MA=3:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | C0 | C1 | C2 | C3 | C4 | C5 | C6 | C7 |
| A0 |  |  |  |  |  |  |  |  |
| A1 |  |  |  |  |  |  |  |  |
| A2 |  |  |  |  |  |  |  |  |
| A3 |  |  |  |  |  |  |  |  |
| A4 |  |  |  |  |  |  |  |  |

Question: For given number of chambers NC, and given assay multiplicity MA, how many unique permutations are possible (i.e. how many assay can we place)?

NAmax = NC! / (MA! \* (NC – MA)!), which is a binomial coefficient[[1]](#footnote-1).

In our example for NC=8 and MA=3 there are 56 possible combinations[[2]](#footnote-2).

We can represent the allocation table as a binary matrix:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | C0 | C1 | C2 | C3 | C4 | C5 | C6 | C7 |
| A0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| A1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| A2 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 |
| A3 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| A4 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |

Table 1: Example of assay allocation table

In this representation each assay is encoded as a vector:

A0 = {1, 0, 1, 1, 0, 0, 0, 0}

A1 = {0, 1, 0, 1, 1, 0, 0, 0}

A2 = {0, 0, 1, 0, 1, 1, 0, 0}

A3 = {0, 0, 0, 1, 0, 1, 1, 0}

A4 = {0, 0, 0, 0, 1, 0, 1, 1}

From these we can calculate “assays correlation” or cross-talk by performing a dot-multiplications of the vectors:

CorrAij = Ai \* Aj

The correlation number tells how many chambers two assays share. We can calculate the entire correlation matrix:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | A0 | A1 | A2 | A3 | A4 |
| A0 | 3 | 1 | 1 | 1 | 0 |
| A1 | 1 | 3 | 1 | 1 | 1 |
| A2 | 1 | 1 | 3 | 1 | 1 |
| A3 | 1 | 1 | 1 | 3 | 1 |
| A4 | 0 | 1 | 1 | 1 | 3 |

Table 2: Assays correlation matrix

Here assays A0 and A4 do not correlate, because they do not share any chambers.

For given multiplicity number MA, how can we optimise the assays placement? We can, for instance, consider chambers utilisation. In example ( Table) if we sum all the numbers per column we’ll get the number of assays allocated per chamber:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | C0 | C1 | C2 | C3 | C4 | C5 | C6 | C7 |
| Utilisation | 1 | 1 | 2 | 3 | 3 | 2 | 2 | 1 |

Table 3: Chambers utilisation

We can even up the chambers utilisation, for example, in a following way:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | C0 | C1 | C2 | C3 | C4 | C5 | C6 | C7 |
| A0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| A1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| A2 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| A3 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| A4 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| Utilisation | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 |

Table 4: Optimised allocation table and chambers utilisation

Corresponding assays correlation matrix will be:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | A0 | A1 | A2 | A3 | A4 |
| A0 | 3 | 1 | 1 | 1 | 0 |
| A1 | 1 | 3 | 1 | 0 | 1 |
| A2 | 1 | 1 | 3 | 1 | 0 |
| A3 | 1 | 0 | 1 | 3 | 1 |
| A4 | 0 | 1 | 0 | 1 | 3 |

Table 5: Assays correlation matrix (corresponds to allocation Table)

If we have a matrix representation of the allocation table, then we can write the “call” as multiplication of the allocation matrix by the signal vector. Signal vector’s length is the number of chambers, and each element of this vector is either 1 (chamber fired) or 0 (no signal detected in this chamber)[[3]](#footnote-3).

A = [M] \* (S), where [M] is the assay allocation matrix, and (S) is the detection signal.

For example, let’s (S) = {0, 1, 0, 0, 1, 1, 0, 1}, then the assay call will be:

In this particular example we do not detect assay 0 (A0 = 0), we detect assays 1, 2, and 4 with 2/3 probability, and we detect assay 3 with 1/3 probability[[4]](#footnote-4).

Considerations:

If we want to increase the detection probability we may increase MA for a given assay, but this may also increase the assays crosstalk (correlation).

Correlation between two assays can be computed really fast if the assays placement is represented as a binary number, where each bit corresponds to a chamber. For Table we will get:

A0 = 0x58

A1 = 0xC8

A2 = 0x64

A3 = 0x16

A4 = 0x0B

Then vector dot multiplication is equivalent to a bit-wise AND operation followed by counting the number of 1 bits. An effective algorithm to count the number of bits is described in Hacker’s Delight by Henry S. Warren Jr.[[5]](#footnote-5):

### Assay allocation matrix permutations.

We can start with a default assays placement where none of the assays correlate[[6]](#footnote-6):

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | C0 | C1 | C2 | C3 | C4 | C5 | C6 | C7 |
| A0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| A1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| A2 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| A3 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| A4 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |

Table 6: Default initial placement matrix [M].

From this placement matrix [M] we can obtain all possible placements by multiplying it by a permutation matrix[[7]](#footnote-7). A permutation matrix is a square matrix that has NC rows and NC columns. Initially it is constructed as an identity matrix [I] (diagonal elements are 1, and all other elements are 0). Further the matrix is transformed by permutating its rows:

Illustration 1: Example of obtaining a random permutation matrix from an identity matrix.

Now we can obtain a variation of a allocation matrix by multiplying the original matrix [M] ( Table):

[M]’ = [M]\*[P]

Permutation matrices can be easily computed if considering each row as a binary number (integer). For 32 chambers we would use 32-bit integer for example. In our case we have 8 chambers so we can use 8-bit numbers:

// Identity matrix

**const** std::vector<uint8\_t> Identity = {

0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80

};

// Computing all permutations

**auto** permutation = Identity;

**do** {

// Apply permutation on default assay allocation matrix

// to obtain an alternative placement

} **while** (std::next\_permutation(permutation.begin(), permutation.end()));

Unfortunately this operation simply rearranges the chambers, which is actually pointless considering that all the chambers are identical.

What we actually want to do is iterate over all permutations of a single assay placement. For NC=8, MA=3 there are 56 possibilities. And we need choose NA=5 from those 56. Which gives us 3819816 possible combinations. Some of them however will be redundant, considering that swapping chambers does not result in a new unique allocation.

The problem is that permutation obtained from an identity matrix does not affect the assays correlation. Geometrical analogy: permutation matrix is equivalent to rotation matrix where rotation angle is multiple of π/2 (90º, 180º, 270º, etc.) so that coordinate system change is just swapping the axes.

### Generating allocation for optimal chambers utilisation

Let’s assume we want to generate an optimal allocation, in the sense that it evenly utilises the chambers (i.e. ~same number of assays per chamber). Again:

NA – number of assays to allocate

NC – number of chambers

MA – assays multiplicity (number of assays copies).

Since we do not really care which chambers an assay goes (we can “rename” the chambers or the assays afterwards) we are going to generate an allocation of a single assay, and all the following assays can use the same allocation but circularly shifter by one position.

For example we want to obtain an optimal allocation for NC=8, NA=5, and MA=3. We start from an empty allocation matrix:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | C0 | C1 | C2 | C3 | C4 | C5 | C6 | C7 |
| A0 |  |  |  |  |  |  |  |  |
| A1 |  |  |  |  |  |  |  |  |
| A2 |  |  |  |  |  |  |  |  |
| A3 |  |  |  |  |  |  |  |  |
| A4 |  |  |  |  |  |  |  |  |

And we start placing the assays in separate chambers following the diagonal of the matrix:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | C0 | C1 | C2 | C3 | C4 | C5 | C6 | C7 |
| A0 | \* |  |  |  |  |  |  |  |
| A1 |  | \* |  |  |  |  |  |  |
| A2 |  |  | \* |  |  |  |  |  |
| A3 |  |  |  | \* |  |  |  |  |
| A4 |  |  |  |  | \* |  |  |  |

Once we reach the margin of the matrix, we obtain a placement of multiplicity 1. We continue the placement by wrapping the matrix along the edge we’ve just reached:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | C0 | C1 | C2 | C3 | C4 | C5 | C6 | C7 |
| A0 | \* |  |  |  |  | \* |  |  |
| A1 |  | \* |  |  |  |  | \* |  |
| A2 |  |  | \* |  |  |  |  | \* |
| A3 |  |  |  | \* |  |  |  |  |
| A4 |  |  |  |  | \* |  |  |  |

Now we have to mirror it over the vertical edge of the matrix:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | C0 | C1 | C2 | C3 | C4 | C5 | C6 | C7 |
| A0 | \* |  |  |  |  | \* |  |  |
| A1 |  | \* |  |  |  |  | \* |  |
| A2 |  |  | \* |  |  |  |  | \* |
| A3 | \* |  |  | \* |  |  |  |  |
| A4 |  | \* |  |  | \* |  |  |  |

Here we’ve just obtained the multiplicity of 2.

We continue the same way to get the desired MA=3:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | C0 | C1 | C2 | C3 | C4 | C5 | C6 | C7 |
| A0 | \* |  | \* |  |  | \* |  |  |
| A1 |  | \* |  | \* |  |  | \* |  |
| A2 |  |  | \* |  | \* |  |  | \* |
| A3 | \* |  |  | \* |  | \* |  |  |
| A4 |  | \* |  |  | \* |  | \* |  |

Now we have multiplicity of MA=3.

Chambers utilisation:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | C0 | C1 | C2 | C3 | C4 | C5 | C6 | C7 |
| Utilisation | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 |

Assays correlation matrix:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | A0 | A1 | A2 | A3 | A4 |
| A0 | 3 | 0 | 1 | 2 | 0 |
| A1 | 0 | 3 | 0 | 1 | 2 |
| A2 | 1 | 0 | 3 | 0 | 1 |
| A3 | 2 | 1 | 0 | 3 | 0 |
| A4 | 0 | 2 | 1 | 0 | 3 |

Assay vectors as binary numbers:

A0=10100100

A1=01010010

A2=00101001

A3=10010100

A4=01001010

Which is basically the same number but circularly shifted by one.

There is a way to compute the first number easily without actually following the placement procedure described above.

We initialise the position counter to zero and place ‘1’ to this position of C0. In order to obtain the next position for ‘1’ we add NA to the counter and perform modular division by NC:

(0 + 5) mod 8 → 5.

So, the next allocation goes to C5. Now we’ve just placed 2 bits (at position 0 and 5), but we need 3 for MA=3. We continue the same way:

(5 + 5) mod 8 → 2.

So, the next allocation does to C2. This way, assay A0 should be placed to chambers 0, 2, and 5, and all other assays are just shifted by one position.

It may happen though that when performing (counter + NA) mod NC we obtain zero. This mean we’ve created a loop. In order to break it (if we have not yet reached the desired multiplicity) we will have to reset the counter to any zero position in current allocation mask.

Say we want to allocate NA=2 assays in NC=4 chambers with multiplicity of MA=3. After a second chamber allocation we will reach the loop (because (2 + 2) mod 4 = 0), by that moment we would have reached multiplicity of 2 only. So we have to reset to position 1 and continue the allocation until we’ve reached MA=3:

Multiplicity of 1:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | C0 | C1 | C2 | C3 |
| A0 | \* |  |  |  |
| A1 |  | \* |  |  |

Multiplicity of 2 (we created a loop):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | C0 | C1 | C2 | C3 |
| A0 | \* |  | \* |  |
| A1 |  | \* |  | \* |

Continue from the next empty chamber to get to MA=3:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | C0 | C1 | C2 | C3 |
| A0 | \* | \* | \* |  |
| A1 |  | \* | \* | \* |

Pseudo-code (C++)[[8]](#footnote-8):

std::vector<**bool**> get\_first\_allocation(**int** nChambers, **int** nAssays, **int** multiplicity)

{

std::vector<**bool**> allocation(nChambers);

**int** bits\_allocated = 0;

**int** placement = 0;

**do** {

allocation[placement] = **true**;

++bits\_allocated;

placement = (placement + nAssays) % nChambers;

// Work around a placement loop

**while** (allocation[placement]) {

placement = (placement + 1) % nChambers;

}

} **while** (bits\_allocated < multiplicity);

**return** allocation;

}

Now we have another problem: this approach will work only for NA ≤ NC. In case of NA > NC the cyclic shift operation will start producing the same placements. One possibility to work around this problem is to permutate the initial placement sequence for every NC assays being allocated.

1. https://en.wikipedia.org/wiki/Binomial\_coefficient [↑](#footnote-ref-1)
2. http://www.statisticshowto.com/calculators/permutation-calculator-and-combination-calculator/ [↑](#footnote-ref-2)
3. We can extrapolate this idea beyond binary numbers and use analogue signal intensity, like 0.85 for insstance. [↑](#footnote-ref-3)
4. Probability is Ai/MA, in this example assay multiplicity MA=3. [↑](#footnote-ref-4)
5. https://www.amazon.co.uk/Hackers-Delight-Henry-S-Warren/dp/0201914654 [↑](#footnote-ref-5)
6. This means that Ai\*Aj < MA for any *i* and *j*. The easiest way to satisfy this condition is to offset the next vector by single position. [↑](#footnote-ref-6)
7. http://mathworld.wolfram.com/PermutationMatrix.html [↑](#footnote-ref-7)
8. This code does not check error conditions when placement is not possible. [↑](#footnote-ref-8)