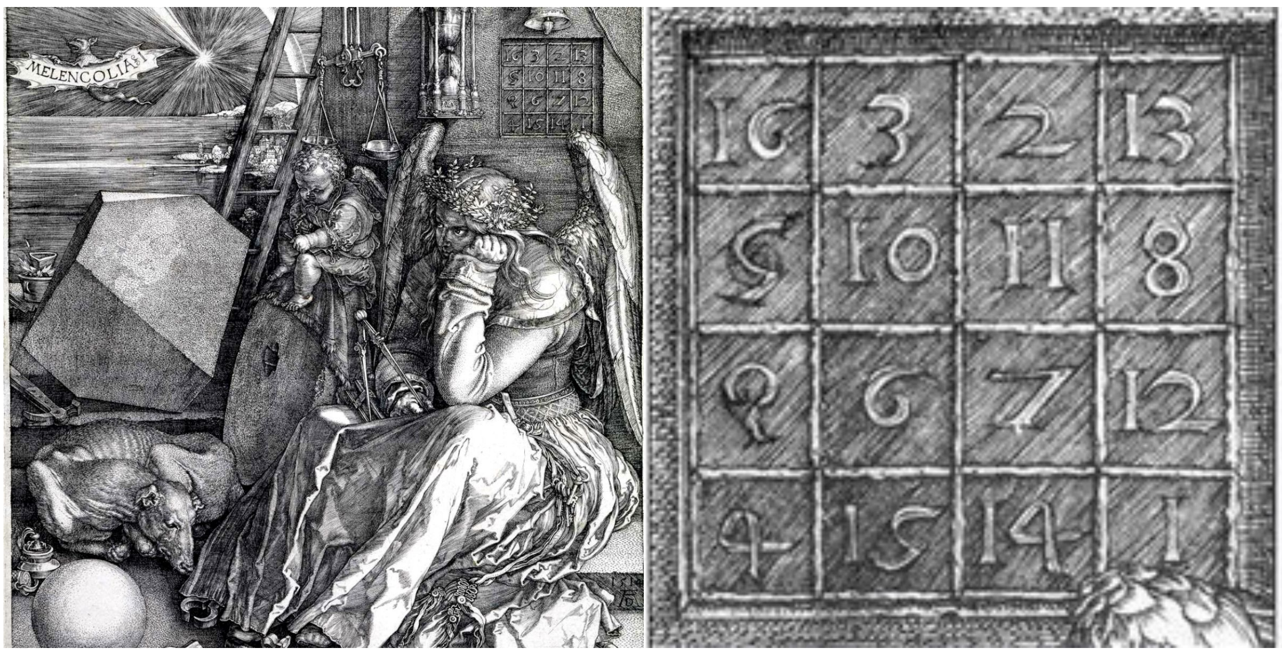
Making Magic
Mathematical Programming with Python

MATH 2604: Advanced Scientific Computing 4
Spring 2025
Monday/Wednesday/Friday, 1:00-1:50pm

https://people.sc.fsu.edu/~jburkardt/classes/python_2025/magic/magic.pdf

---



Albrecht Dürer's Melencolia II includes a magic square in the upper right corner,
He fiddled with the numbers so that the date 1514 appeared in the bottom row.

A magic square is a classic math object

- *Magic squares have intrigued recreational mathematicians for centuries;*
- *Algorithms have been found for creating squares of odd or even order;*
- *An odd algorithm is defined by Christian Hill in his textbook;*
- *To set up a square requires that we be able to do certain simple tasks;*
- *We will create a magic square using* `numpy()` *arrays;*

1 Magic squares

A magic square of order n is an $n \times n$ array of numbers such that all rows, all columns, and all diagonals have the same sum. (In a "semi-magic square", this is only true for the rows and columns.) It is not obvious at first how to construct a magic square, and hence they are often considered to be lucky charms with magic properties.

A magic square is a simple example of a `matrix`. When programmers need a “random” matrix to illustrate some procedure, they frequently use a magic matrix, since the entries are distinct integers, and the matrix has some interesting properties. MATLAB, for example, has a built-in `magic(n)` function to create a magic matrix of any order.

Knowing the values and their placement in advance, it’s easy to define a `numpy()` array for our 3×3 example:

```
A = np.array ( [
   [ 8, 1, 6 ],
   [ 3, 5, 7 ],
   [ 4, 9, 2 ] ] )
```

and we can test our matrix with the commands

```
  np.sum ( A[:,0] )
  np.sum ( A[:,1] )
  np.sum ( A[:,2] )
  np.sum ( A[0,:] )
  np.sum ( A[1,:] )
  np.sum ( A[2,:] )
  A[0,0] + A[1,1] + A[2,2]
  A[2,0] + A[1,1] + A[0,2]    # Surely there is a neater way?
```

But our question today is, how can we use Python to create a new magic square, by following the steps of the magic square algorithm? Assuming the algorithm is clear, we would hope that we know enough Python to get a suitable program fairly quickly and easily.

Creating a magic matrix is really a test example for us. We will soon be wanting to create and use numerical vectors and matrices, and perform all sorts of linear algebra operations with them. We really need to be comfortable with the Python tools we will use to do this.

2 An algorithm for magic matrices of odd order

I remember learning this algorithm in third grade!

To create an $n \times n$ magic square, draw a grid of empty boxes to hold your values. Number the rows from top to bottom, and the columns from left to right.

1. Start in the middle of the top row, and let $k = 1$.
2. Write k in the current grid position;
3. If $k = n^2$, the grid is complete, so stop;
4. Else, set $k = k + 1$;
5. Plan to move diagonally up and right. But if this move leaves the grid, wrap to the first column or last row.
6. If this cell is already filled, move vertically down one space instead;
7. Return to step 2.

While the algorithm seems to make sense, let’s work through some examples to see what is being talked about! Create a suitable empty grid of cells, and then recreate the magic matrices of order 3, 5, and 7.

The 3×3 example:

$$A_3 = \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix}$$

The 5×5 example:

$$A_5 = \begin{bmatrix} 17 & 24 & 1 & 8 & 15 \\ 23 & 5 & 7 & 14 & 16 \\ 4 & 6 & 13 & 20 & 22 \\ 10 & 12 & 19 & 21 & 3 \\ 11 & 18 & 25 & 2 & 9 \end{bmatrix}$$

The 7×7 example:

$$A_7 = \begin{bmatrix} 30 & 39 & 48 & 1 & 10 & 19 & 28 \\ 38 & 47 & 7 & 9 & 18 & 27 & 29 \\ 46 & 6 & 8 & 17 & 26 & 35 & 37 \\ 5 & 14 & 16 & 25 & 34 & 36 & 45 \\ 13 & 15 & 24 & 33 & 42 & 44 & 4 \\ 21 & 23 & 32 & 41 & 43 & 3 & 12 \\ 22 & 31 & 40 & 49 & 2 & 11 & 20 \end{bmatrix}$$

3 Implementing the algorithm

There are several tasks we need to do computationally, in order to carry out this algorithm in Python.

We **need to**:

- create, in advance, a place to store the entries, that is an array of n rows, each row specifying n values.
- locate any entry using $[i, j]$ indexing.
- tell when a planned move would takes us out of the grid.

We would also **like to**:

- print the resulting magic matrix.
- compute the row, column and diagonal sums.
- check the row and column sums.

This very last item can best be illustrated by using MATLAB. If A_n is a magic matrix of order n, and v is a vector of 1's of length n, then $A * x$ is the vector of row sums, and $A^T * x$ is the vector of column sums. There are other commands that allow us to also check the diagonal and antidiagonal sums.

```
%  Warning: This is MATLAB code, not Python!
n = 7
Asum = ( n * ( n^2 + 1 ) ) / 2
A = magic ( n )
x = ones ( n, 1 )
A*x
A'*x              % Transpose of A times x
sum ( diag(A) )
sum ( diag ( flipud(A) ))  % Flip A upside down to get antidiagonal
```

Although we could presumably check magic squares in other ways, we will really wish to be able to use more sophisticated tools for later numerical calculations.

4 Allocating an $n \times n$ numpy() array

Assuming we have chosen `n=5`, the size of our magic matrix, we need to create a numpy() array and initialize it to zeros:

```
n = 5
A = np.zeros ( [ n, n ] )
```

Our first task is to place a value `k=1` in the middle of the top row.

```
k = 1
i = 0
j = n // 2     # We want integer division here, so 5//2 = 2
A[i,j] = k
```

To check that we got this right, we can print our updated matrix:

```
print ( A )
```

5 Implementing the algorithm

Assuming we have set up space for the array, the remainder of our Python implementation is:

```
k = 1
i = 0
j = n // 2

while ( k <= n**2 ):

  A[i,j] = k

  k = k + 1
#
#  Try to go up one row, and over one column.
#
  new_i = ( i - 1 ) % n
  new_j = ( j + 1 ) % n

  if ( A[new_i,new_j] ):     # True if A[new_i,new_j] is already set
    i = i + 1
  else:
    i = new_i
    j = new_j

return A
```

What does "`if ( A[new_i,new_j] ):`" mean? Remember that, to Python, 0 plays the role of `False` and 1, or in fact, any nonzero value, is understood as `True`. So all we are saying here is `if ( A[new_i,new_j] has been set):`.

But wait, why is `new_i` computed by subtracting 1? Don't we want to go up? Yes, but in a matrix, going up a row means decreasing the row number!

6 Verify the Magic

Let's try to verify that our matrix is magic. Instead of using many individual `sum()` commands, or multiplying by a vector of 1's, we can use the fact that the `sum()` command can return the sum of rows or columns of a matrix if we specify the "axis":

```
np.sum ( A )            # sums all the entries
np.sum ( A, axis = 0 )  # sums the columns
np.sum ( A, axis = 1 )  # sums the rows
```

It turns out that we can compute the diagonal sum because there is a linear algebra function called `trace()` that sums the diagonal elements of a matrix, and `numpy()` implements it:

```
np.trace ( A )
```

How do we get the antidiagonal sum? There are a number of `numpy()` functions for flipping a matrix. In particular, `flipud()` flips up and down, with the result that what was the antidiagonal of the matrix is now the diagonal.

```
np.trace ( np.flipud ( A ) )
```

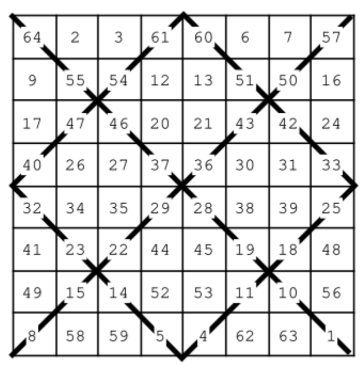
As an alternative, we could take the trace of the transpose of A:

```
np.trace ( np.transpose ( A ) )
```

In any case, if we have done our work correctly, all these operations should return the same magic value.

7 What's so odd about an even magic matrix?

The rule for creating a magic matrix of odd order is pretty simple to describe. But in the even case, the story is more complicated. First, there is no magic matrix of order $n = 2$. Secondly, the even case splits into the *doubly even case*, where n is a multiple of 4, and the `singly even case`, where $n = 4 * k + 2$ for some k. For both these cases, there are algorithms for creating magic matrices. In the doubly even case, the algorithm is actually simpler than the case for odd n.



The doubly even magic matrix algorithm draws X's over the grid.

For the doubly even case, we begin by placing the numbers 1 through n^2 consecutively in the grids of the cells. Then we imagine drawing an X through each 4x4 subblock. The cells that are crossed by the lines of the X's are then "reflected", so that if the cell was original numbered k, it should now be renumbered as $n^2 + 1 - k$. That's it.

Writing a computer program to do this is not difficult, once you think about how to determine whether a cell is X's or not. You should see that arithmetic modulo 4 is involved, and the sum or difference of the cell indices i and j is needed. The lines slanting up and to the right look like $i - j = c$ formulas, while the down and to the right lines look like $i + j = c$ formulas. Figure out the equations for these lines, and you have pretty much worked out the algorithm.