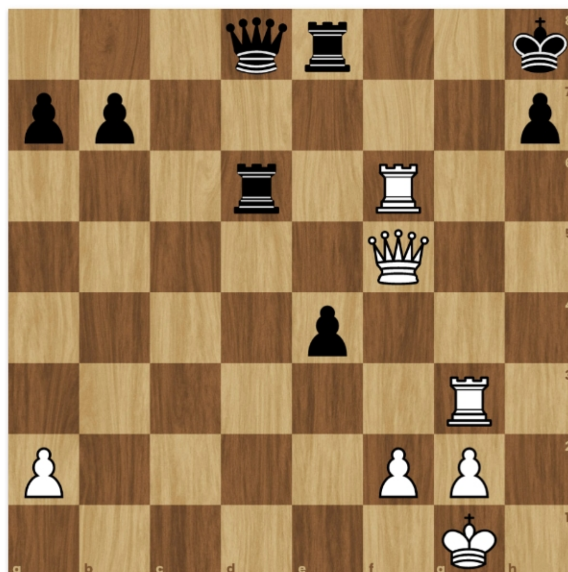
Reduced Row Echelon Form for Matrices Mathematical Programming with Python

MATH 2604: Advanced Scientific Computing 4
Spring 2025
Monday/Wednesday/Friday, 1:00-1:50pm
Room A202 Langley Hall

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

---



This chess layout is almost in row echelon form!

---

1 The Reduced Row Echelon Form

There are some important features of a matrix that are worth knowing. Such features include singularity, the determinant, the inverse, the rank, and the solution of related linear systems. It is usually not possible to determine these quantities simply by inspection of the matrix entries. Instead, we usually try to gently transform the matrix to an equivalent form, in which these questions are more easily answered.

One technique is to determine the *reduced row echelon form* or **RREF** of the matrix. The RREF has a simple structure that makes it easy to answer our questions. As a programming exercise, the RREF can be computed interactively, and requires the programmer to be familiar with the rules for array indexing and modification.

Although there are other techniques for analyzing a matrix, we will examine the RREF because if is both useful, and moderately challenging, and involves many of the operations necessary for any matrix analysis.

2 Definition of the RREF

Define the **pivot** of each matrix row as the first nonzero entry, if any.

Then an $m \times n$ matrix is in row echelon form if

1. zero rows, if any, occur after all other rows;
2. The pivot of each row occurs in a later column than the pivots of all preceding rows.
3. Every pivot has the value 1.
4. Every pivot is the only nonzero entry in its column.

The identity matrix and the zero matrix are extreme examples of RREF. Here is a more general example:

$$A = \begin{pmatrix} 1 & 3 & 0 & 9 & 4 \\ 0 & 0 & 1 & 7 & 8 \\ 0 & 0 & 0 & 1 & -3 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

3 Identify any RREF matrix

Our challenge is to recognize whether a matrix A is in RREF form. We need to write a function `is_rref(A` which returns `True` if A is an RREF matrix. How do we do this?

Perhaps it makes sense to organize the inspection one row at a time, using a loop `for r in range ( 0, m ):`. When checking row r, we need to know that the first nonzero value occurs later than in the previous row. Use the variable c for this purpose. Generally, c will be the column where we find the first nonzero in a row, and then before we examine the next row, we increase c by 1. Before we start, set $c = -1$. Initialize the return value of the function to `True`, that is, we think this is an RREF matrix. Now we check row by row, to see if this is true:

1. If row r is entirely zero, set $c = n$, so that all subsequent rows must be entirely zero too.
2. If there is a nonzero, and it occurs before column c, the matrix is not in RREF, and return `False`;
3. If there is a nonzero, and it occurs on or after column c, we update c to that column index.
4. If the nonzero entry is not 1, the matrix is not RREF and return `False`;
5. If the pivot is not the only nonzero in column c, return `False`;

4 RREF Test Cases:

Test the following matrices for RREF:

$$A0 = \begin{pmatrix} 1 & 0 & 0 & 9 & 4 \\ 0 & 0 & 1 & 0 & 8 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

$$A1 = \begin{pmatrix} 1 & 0 & 0 & 9 & ,4 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 8 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$A2 = \begin{pmatrix} 1 & 0 & 0 & 9 & 4 \\ 0 & 1 & 0 & 2 & 8 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$A3 = \begin{pmatrix} 1 & 0 & 3 & 9 & 4 \\ 0 & 1 & 0 & 2 & 8 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$A4 = \begin{pmatrix} 1 & 0 & 3 & 0 & 4 \\ 0 & 1 & 2 & 0 & 8 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

5 EROS: Elementary Row Operations

Any matrix can be transformed to RREF format by a series of elementary row operations or **ERO**'s that gradually reveal the desired RREF. There are three of these operations:

- ERO #1: $R_i \leftrightarrow R_j$ interchanges rows i and j;
- ERO #2: $R_i \rightarrow R_i/s$ divides row i by s;
- ERO #3: $R_i \rightarrow R_i + s * R_j$ adds a multiple of row j to row i;

Along with these operations, we will also need to be able to

- find the first nonzero in row i;
- find the largest entry in a partial column j (rows i through $m-1$);

Our strategy will be as follows:

```
Initialize r = 0
For c = 0 to n - 1: consider column c
  Find k, the index of the largest entry in column c, from row r to m-1.
  If largest entry is zero
    continue (move on to next value of c)
  Else
    ERO #1 swap rows r and k
    ERO #2 rescale row r so that A[r,c] = 1
    ERO #3 add multiples of row r to all other rows, zeroing out
  the other entries in column c.
    increase r = r + 1;
```

The algorithm isn't difficult to describe; however, when we try to implement it with a Python program, we will find difficulties, surprises and some disasters on almost every step!

6 Choose the pivot row

We are seeking a pivot for column `c`, and we are searching from row `r` downwards. We are looking for a nonzero value, and it is best if we look for the entry of largest absolute value.

The numbers we are examining can be described as `A[r:,c]`

The absolute values are computed as `np.abs ( A[r:,c] )`

The maximum would be `np.max ( np.abs ( A[r:,c] ) )`

We don't want this value, we want its location. `p = np.argmax ( np.abs ( A[r:,c] ) )`

The value of `p` counts starting at `r`, so to get the location of this entry in the full matrix we have to add `r`. `p = p + r`

Now we have found that row `p` of the matrix is our next pivot row.

But what happens if all the values we examined are zero (or even just extremely small)? Then the search for a pivot for column `c` has failed, and we move on.

```
  row = 0
#
#  Seek a pivot for each column.
#
  for col in range ( cols ):
#
#  Exit if we have run out of rows to examine.
#
    if ( rows <= row ):
      break
#
#  Find the pivot row.
#
    pivot_row = np.argmax ( np.abs ( A[row:rows, col] ) ) + row
#
#  Skip this column if all values are below the tolerance.
#
    if ( np.abs ( A[pivot_row, col] ) <= tol ):
      continue

    (...more to come)
```

7 Swap current row and pivot row

Unless `p` and `r` are the same, we need to interchange these two rows of the matrix. The straightforward way is to use a `for()` loop:

```
for j in range ( 0, cols ):
  t = A[r,j]
  A[r,j] = A[p,j]
  A[p,j] = t
```

This is a little fussy. A neater way would see to be as follows:

```
t           = A[r,0:cols]   # Warning.  This is not right!
A[r,0:cols] = A[p,0:cols]
A[p,0:cols] = t
```

This perfectly reasonable-looking set of commands actually does the wrong thing. To see what is happening, consider this example:

```
>>> A = np.array ( [[ 1, 2, 3],[4,5,6],[7,8,9]])
>>> t = A[0,:]
>>> A[0,:] = A[2,:]
>>> A[2,:] = t
>>> print ( A )
```

```
[[7 8 9]
 [4 5 6]
 [7 8 9]]
```

The variable `t` is not a copy of `A[0,:]`, but rather a pointer to it. So now whenever we mention `t`, we need to go to row 0 of the matrix `A`. So row 0 gets a copy of row 2, and then row 2 gets a copy of ... whatever `t` is pointing to, namely, whatever is in row 0, that is, a copy of row 2.

Please repeat this set of commands, with one change:

```
t = A[0.:].copy()
```

Try to understand that with the `copy()` command, we are creating a pointer `t` which points to a new set of data that was copied out of the matrix `A`. This is a confusing idea. It means that a Python variable name is not always what you are used to thinking it to be.

For those who like to be on the cutting edge, there is actually a one line command to do the row swapping:

```
A[[r,p]] = A[[p,r]]
```

This is elegant, but dangerous. You might forget to use two sets of square brackets, and `A[r,p]=A[p,r]` does not do what you want! Also, it only works for swapping rows, not columns. But if you understand what it is doing, it is a clean one-line solution to our task.

8 Scale the pivot row

We are going to use the pivot row to zero out the column `c` entries in all other rows. Before doing this, it is convenient to use an elementary row operation that divides the pivot so that `A[r,c]=1`. We might try using a `for()` loop, but let's prepare to make a horrible mistake:

```
for j in range ( 0, cols ):
  A[r,j] = A[r,j] / A[r,c]
```

Try this code on an example matrix and see the mysterious result!

This seems to work better:

```
t = A[r,c]
for j in range ( 0, cols ):
  A[r,j] = A[r,j] / t
```

Again, it is tempting to replace several commands by one. Again, we are lucky that in Python, it is easy to address an entire row using a single row index:

```
A[r] = A[r] / A[r,c]
```

Unlike the problem we had using a step-by-step `for()` loop, here we don't have to worry about the fact that our divisor `A[r,c]` is going to be changed during the execution of the command.

However, here another crazy thing can happen. If Python thinks the matrix `A` is of integer type, then it will before integer division here, so, for example the result of `13/5` is 2, not 2.6. To avoid this problem, you can specify your data using decimal points, or the RREF code can specify that, no matter what the matrix looks like, it should be treated like real data:

```
  A = A.astype ( float )
```

9 Use pivot to zero out lower entries

So our pivot row `r` has the value `A[r,c]=1`. We want to use this row to eliminate all other nonzero entries in column `c` of the matrix. If we use a `for()` loop, we might think to write

```
for i in range ( 0, rows ):
  A[i] = A[i] - A[r] * A[i,c]
```

Remember, `A[i]` indicates an entire row of the matrix, while `A[i,c]` indicates a particular entry.

Alas, if we execute this statement, every entry of column `c` is zero, including the value of `A[r,c]`, which should have remained 1. This is a case of overkill. We wanted to zap all the rows except for row `r`. We need to make a small, obvious, adjustment to our loop to correct our over-enthusiasm. Surely you can tell me what that is!

Finally, having completed the computation of the pivot for column `c` using row `r`, we increment `r` by 1 before starting the next iteration of the loop.

10 The code

```
def rref ( A ):

  import numpy as np

  A = A.astype ( float )

  rows, cols = A.shape

  tol = np.sqrt ( np.finfo(float).eps )

  row = 0

  for col in range ( cols ):

    if ( rows <= row ):
      break

    pivot_row = np.argmax ( np.abs ( A[row:rows, col] ) ) + row

    if ( np.abs ( A[pivot_row, col] ) <= tol ):
      continue

    A[[row, pivot_row]] = A[[pivot_row, row]]

    A[row] = A[row] / A[row, col]

    for i in range ( rows ):
      if ( i != row ):
        A[i] = A[i] - A[i, col] * A[row]

    row = row + 1

  return A
```

11 Learning from disasters

The reduced row echelon form of a matrix is an important tool, and so it's good to understand how it is computed. It's easy to do an example by hand (except for the arithmetic). It's easy to imagine the steps

mathematically. But as we have struggled to immplement this algorithm as a Python computation, we have been surprised by several pitfalls. Even though your commands may look correct mathematically, you have to be aware that sometimes there are hidden effects that you need to watch out for!