Row Reduction

Different algorithms to calculate the LU decomposition of a matrix.

First of all let us have constructors for two types of matrices. The first is the Hilbert matrix which is well-known to be a "difficult" matrix to work with:

```
def hilbmat(k):
      return matrix(k,k,lambda i,j: 1.0/(1.0+i+j))
h = hilbmat(5)
```

Secondly, we want a matrix with random entries:

```
def randmat(k):
      return matrix(k,k,lambda i,j: random())
m = randmat(5)
```

Row Reduction without Pivoting

We will calculate the row-reduction of a matrix without using pivoting.

For each column, we have to apply row reduction by using the entries below the diagonal. This gives a matrix (called `elem` below) which is strictly lower triangular and has square 0. Note how we keep track of the LU decomposition incrementally.

```
def no_pivot(m):
      cols = len(m.columns())      # We only work with square matrices
      li   = matrix.identity(cols) # To "store" L^(-1)
      ll   = matrix.identity(cols) # To "store" L
      u    = m                     # To "store" U
      for l in range(cols-1):
          assert u[l,l] != 0,\
```

```
            'diagonal entries should not become zero for no_pivot to work'
        elem = matrix(cols,cols,
            lambda i,j: 0.0 if j!=l else (
                    0.0 if i<=l else u[i,l]/u[l,l]
                    )
            )
        u  = u - elem*u
        li = li - elem*li
        ll = ll + ll*elem
    return (ll,li,u)
```

Apply this to our test matrix `m`.

```
(ll,li,u) = no_pivot(m)
```

Check that the first two are inverses of each other.

```
ll*li
```

$$\begin{pmatrix} 1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 1.0 \end{pmatrix}$$

Check that we have an LU decomposition of `m`.

```
ll*u-m
```

$$\begin{pmatrix} 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & -2.22044604925 \times 10^{-16} & 0.0 & 1.11022302463 \times 10^{-16} \\ 0.0 & 0.0 & 1.11022302463 \times 10^{-16} & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.11022302463 \times 10^{-16} & -1.11022302463 \times 10^{-16} & 0.0 \end{pmatrix}$$

We see that small errors of the order of $10^{-15} \simeq 2^{-53}$ have crept in. Note that the entries of the matrix are (with high probability) of the same order of magnitude as 1.0. Hence a number of such small magnitude should be considered to be effectively the same as 0 (with high probability)!

Partial Pivoting

In this method, we look for the largest entry in each column leaving out those that belong to rows that already have a pivot in them. To simplify the function, let us first write a helper function that generates the elementary matrix required to:

- perform row reduction using a particular entry in a column as a pivot
- excludes some rows from the search for a pivot

```
def row_reduce_mat(mat,col,piv,excl):
      """Create the row-reduction matrix corresponsing the col-th column
      of the matrix mat using the piv-th entry in the column.
      We leave out the entries listed in excl (which is assumed to include piv)."""
      cols = len(mat.columns())
      return matrix(cols,cols,
                lambda i,j: 0.0 if j!=piv else (
                       0.0 if (i in excl) else mat[i,col]/mat[piv,col]
                       )
                )
```

We also define a function that will find the pivot in a given column.

```
def find_pivot(mat,col,excl):
      """Find the index of the largest entry of matrix mat in the col-th
      column while leaving out the entries from excl."""
      cols = len(mat.columns())
      piv  = 0
      while piv in excl:
          piv += 1
      assert piv < cols, "cannot exclude all entries!"
      for k in range(piv+1,cols):
          if not (k in excl) and mat[k,col] > mat[piv,col]:
             piv = k
      return piv
```

This function first skips over the initial excluded pivots and then looks for the pivot

```
def part_pivot(m):
      cols = len(m.columns())      # We only work with square matrices
      li   = matrix.identity(cols) # To "store" L^(-1)
      ll   = matrix.identity(cols) # To "store" L
      pivs = []                    # To "store" list of pivots
      u    = m                     # To "store" U
      for l in range(cols-1):
          # First find the pivot
          piv = find_pivot(u,l,pivs)
          pivs.append(piv)
          # Now we can do row-reduction with the pivot
          elem = row_reduce_mat(u,l,piv,pivs)
          u  = u - elem*u
          li = li - elem*li
          ll = ll + ll*elem
      return (pivs,ll,li,u)
```

Apply this to our test matrix `m`.

```
(pivs,ll,li,u) = part_pivot(m)
```

Check that the next two are inverses of each other.

```
ll*li
```

$$\begin{pmatrix} 1.0 & 0.0 & -5.55111512313 \times 10^{-17} & 0.0 & 0.0 \\ -1.11022302463 \times 10^{-16} & 1.0 & 8.32667268469 \times 10^{-17} & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 6.93889390391 \times 10^{-17} & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 1.0 \end{pmatrix}$$

Check that we have an LU decomposition of `m`.

```
ll*u-m
```

$$\begin{pmatrix} 0.0 & 0.0 & -1.11022302463 \times 10^{-16} & -5.55111512313 \times 10^{-17} & 0.0 \\ 0.0 & 0.0 & 0.0 & -5.55111512313 \times 10^{-17} & 1.11022302463 \times 10^{-16} \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & -1.11022302463 \times 10^{-16} & 0.0 & 0.0 & 0.0 \end{pmatrix}$$

The problem is that L is *not* lower triangular (and U is *not* upper triangular!).

```
ll
```

$$\begin{pmatrix} 1.0 & 0.0 & 0.968807313184 & 0.0 & -0.907712483313 \\ 0.174254699246 & 1.0 & 0.868361004866 & -0.808291556431 & -0.986156973028 \\ 0.0 & 0.0 & 1.0 & 0.0 & 0.0 \\ 0.878792304924 & 0.0 & 0.227859879591 & 1.0 & -0.287313076994 \\ 0.0 & 0.0 & 0.146005956491 & 0.0 & 1.0 \end{pmatrix}$$

So we must use the pivot positions to permute these matrics.

```
def pivs_to_perm(p):
      cols = len(p)+1 # We do not need row-reduction for 1x1 matrix
      t = tuple(p)
      for k in range(cols):
          if not (k in t):
             t = t + (k,)
      return matrix(cols,cols,lambda i,j: 1.0 if i==t[j] else 0.0)
```

Convert our pivot positions to a permutation matrix using this function.

```
perm=pivs_to_perm(pivs)
```

Sanity check!

```
pivs
```

$$[2, 4, 0, 3]$$

```
perm
```

$$\begin{pmatrix} 1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 1.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 0.0 \end{pmatrix}$$

Looks right!

Create a new L and U using this permutation.

```
nl, nli, nu = perm.T*ll*perm, perm.T*li*perm, perm.T*u
```

Check that the new L is lower triangular.

```
nl
```

$$\begin{pmatrix} 1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.896319622126 & 1.0 & 0.0 & 0.0 & 0.0 \\ 1.03219699768 & -5.42974794349 & 1.0 & 0.0 & 0.0 \\ 0.150706909933 & -6.58798009531 & 0.990879698897 & 1.0 & 0.0 \\ 0.235196283606 & 0.427816339412 & -0.192315841427 & -0.513985138113 & 1.0 \end{pmatrix}$$

We should now have a PLU decomposition of our matrix.

```
perm*nl*nu-m
```

$$\begin{pmatrix} 0.0 & 0.0 & 0.0 & -5.55111512313 \times 10^{-17} & 0.0 \\ 0.0 & 0.0 & 0.0 & -5.55111512313 \times 10^{-17} & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & -1.11022302463 \times 10^{-16} & 0.0 & 0.0 \\ 0.0 & -1.11022302463 \times 10^{-16} & 0.0 & 0.0 & 0.0 \end{pmatrix}$$

We can now define a function that performs the PLU decomposition of a matrix.

```
def pplu(m):
      pivs, ll, li, u = part_pivot(m)
      perm = pivs_to_perm(pivs)
      nl, nli, nu = perm.T*ll*perm, perm.T*li*perm,  perm.T*u
      return (perm, nl, nli, nu)
```

This produces the PLU decomposition using partial pivoting.

Scaled pivoting

A way to further improve the choice of pivot is to rank the column entries by the ratio of each entry to the largest entry in the same row. In order to do this we can make a small modifiction to the `part_pivot` function by changing the *way* we find the pivot.

```
def scaled_pivot(m):
      cols = len(m.columns())      # We only work with square matrices
      li   = matrix.identity(cols) # To "store" L^(-1)
      ll   = matrix.identity(cols) # To "store" L
      pivs = []                    # To "store" list of pivots
      u    = m                     # To "store" U
      for l in range(cols-1):
```

```
        # First find the pivot
        piv = find_scaled_pivot(u,l,pivs)
        pivs.append(piv)
        # Now we can do row-reduction with the pivot
        elem = row_reduce_mat(u,l,piv,pivs)
        u  = u - elem*u
        li = li - elem*li
        ll = ll + ll*elem
    return (pivs,ll,li,u)
```

To use this function, we must write a function that finds the scaled pivot.

For this, we need a function that calculates the scaled `i`-th entry of a particular vector.

```
def scaled_entry(vector,i):
    return i/max(vector)
```

```
def find_scaled_pivot(mat,col,excl):
    """Find the index of the scaled pivoting entry of matrix mat in
    the col-th column while leaving out the entries from excl."""
    cols = len(mat.columns())
    piv  = 0
    while piv in excl:
        piv += 1
    assert piv < cols, "cannot exclude all entries!"
    for k in range(piv+1,cols):
        if not (k in excl) and \
          scaled_entry(mat.row(k),col) > scaled_entry(mat.row(piv),col):
           piv = k
    return piv
```

Let us test whether this leads to some change.

```
def splu(m):
      pivs, ll, li, u = scaled_pivot(m)
      perm = pivs_to_perm(pivs)
      nl, nli, nu = perm.T*ll*perm, perm.T*li*perm,  perm.T*u
      return (perm, nl, nli, nu)
```

We calculate:

```
(perm, nl, nli, nu) = splu(m)
```

and check:

```
perm*nl*nu - m
```

$$\begin{pmatrix} 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & -2.22044604925 \times 10^{-16} & 0.0 & 1.11022302463 \times 10^{-16} \\ 0.0 & 0.0 & 1.11022302463 \times 10^{-16} & 0.0 & -1.11022302463 \times 10^{-16} \\ 0.0 & 0.0 & 1.11022302463 \times 10^{-16} & -1.11022302463 \times 10^{-16} & 0.0 \end{pmatrix}$$

```

```