COMPULSORY EXERCISE 1

JOHAN ÅMDAL ELIASSEN

[Mat-inf4130 2015]

1. Givens rotations to solve $A\mathbf{x} = \mathbf{b}$

General idea, using roughly the same notation as in the book:

$$\left(\begin{array}{cccc|c} x & x & x & x & b_1 \\ x & x & x & x & b_2 \\ x & x & x & x & b_3 \\ x & x & x & x & b_4 \end{array}\right) \to \left(\begin{array}{cccc|c} \mathbf{r} & x^* & x^* & x^* & b_1 \\ x & x & x & x & b_2 \\ x & x & x & x & b_3^* \\ \mathbf{0} & x^* & x^* & x^* & b_4^* \end{array}\right) \to \left(\begin{array}{cccc|c} \mathbf{r}^* & x^{**} & x^{**} & x^{**} & b_1 \\ x & x & x & x & b_2^{**} \\ \mathbf{0} & x^* & x^* & x^* & b_3^* \\ 0 & x^* & x^* & x^* & b_4^* \end{array}\right)$$

$$\to \left(\begin{array}{cccc|c} r_{1,1} & r_{1,2} & r_{1,3} & r_{1,4} & c_1 \\ \mathbf{0} & x^{**} & x^{**} & x^{**} & b_2^{**} \\ 0 & x^{**} & x^{**} & x^{**} & b_3^{**} \\ 0 & x^* & x^* & x^* & b_4^* \end{array}\right) \to \cdots$$

with the asterix superscripts denoting the number of times that element has been changed via matrix multiplication.
In Python, this becomes:

```python
def rotsolve(A, b):
    """
    Solve the system Ax=b where A any N x N matrix, again using
    Givens rotations. Modified from rothesstri(A, b).

    A: A matrix. Nonsingular ones only.
    b: The right hand side.
    """
    n = shape(A)[0]
    A = hstack([A, b])
    for k in xrange(n-1):    # columns
        for j in xrange(n - 2, k -1, -1):    # rows
            r = linalg.norm([ A[j , k] , A[j + 1, k] ])
            if r>0:
                c=A[j, k]/r; s=A[j + 1, k]/r
                A[[j, j + 1],(k + 1):(n + 1)] = \
                    [[c, s],[-s, c]]*A[[j, j + 1],(k + 1):(n +
    1)]
            A[j, k] = r; A[j+1,k] = 0
    z = A[:, n].copy()
    rbacksolve(A[:, :n], z, n)
    return z
```

.

The program itself is enclosed, obviously.

An aside of arguable interest. Comparing errors (problem 3) with a friend gave very different results; there are obviously many ways to implement this solver. The following is a perhaps more intuitive approach:

$$\left(\begin{array}{cccc|c} x & x & x & x & b_1 \\ x & x & x & x & b_2 \\ x & x & x & x & b_3 \\ x & x & x & x & b_4 \end{array}\right) \to \left(\begin{array}{cccc|c} \mathbf{r} & x^* & x^* & x^* & b_1^* \\ x & x & x & x & b_2 \\ x & x & x & x & b_3 \\ \mathbf{0} & x^* & x^* & x^* & b_4^* \end{array}\right) \to \left(\begin{array}{cccc|c} \mathbf{r} & x^* & x^* & x^* & b_1^{**} \\ x & x & x & x & b_2 \\ \mathbf{0} & x^* & x^* & x^* & b_3^* \\ 0 & x^* & x^* & x^* & b_4^* \end{array}\right)$$

$$\to \left(\begin{array}{cccc|c} r_{1,1} & r_{1,2} & r_{1,3} & r_{1,4} & c_1 \\ \mathbf{0} & x^* & x^* & x^* & b_2^{**} \\ 0 & x^* & x^* & x^* & b_3^{**} \\ 0 & x^* & x^* & x^* & b_4^* \end{array}\right) \to \cdots.$$

with code:

```
def rotsolveAlt(A, b):
    """
    Solve the system Ax=b where A any N x N matrix, again using
    Givens
    rotations. Modified from rothesstri(A, b).

    A: A matrix. Nonsingular ones preferred, obviously.
    b: The right hand side.
    """
    n = shape(A)[0]
    A = hstack([A, b])
    for k in xrange(n-1):    # columns
        for j in xrange(k + 1, n):    # rows
            r = linalg.norm([ A[k , k] , A[j, k] ], 2)
            if r>0:
                c=A[k, k]/r; s=A[j, k]/r
                A[[k, j],(k + 1):(n+1)] = \
                    mat([[c, s],[-s, c]])* \
                    A[[k, j],(k + 1):(n+1)]
            A[k, k] = r; A[j,k] = 0
    z = A[:, n].copy()
    rbacksolve(A[:, :n], z, n)
    return z
```

.

2. Complexity of the algorithm

Taking, for simplicity, the computation of r to be four operations (squaring, squaring, adding, taking the square root), and having that computing c and s each takes one operation, we have besides this each inner step in the algorithm (i.e. for each j) containing a multiplication of a (2×2) matrix with a $(2 \times (n-k+1))$ matrix, for a total of $6(n-k+1)+6 = 6(n-k+2)$ operations.

There are $n-k$ j-steps, making for a total of $6(n-k+2)(n-k)$ arithmetic operations per k-step. Thus, the total number of operations are

$$\sum_{k=1}^{n-1} 6(n-k+2)(n-k) = 6\sum_{l=1}^{n-1} l(l+2) \tag{2.1}$$

$$\approx 6\int_0^{n-1} l^2 + 2l\,\mathrm{d}l = 2(n-1)^3 + 2(n-1) = 2n^3 - 6n^2 + 8n - 1 \approx \underline{2n^3\,.} \tag{2.2}$$

as n becomes large.

Finally, we end up with a system $U\mathbf{x} = \mathbf{c}$, where U is upper triangular, at which point the backsolver can be applied, but that algorithm has takes n^2 operations, thus not changing the result. $\square$

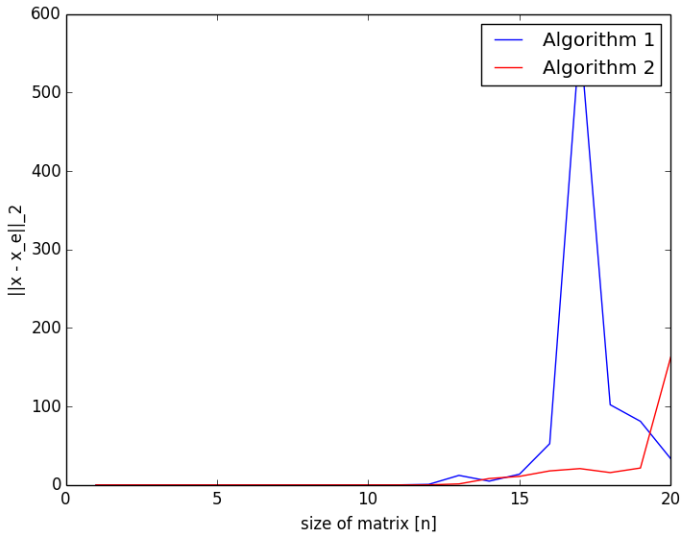
3. Also

The code is enclosed, but:

```
if __name__ == '__main__':

    N = 20

    H_ = hilbert(N)
    xe_ = mat(ones(N)).T
    err = empty(N)
    errAlt = empty(N)
    iterations = arange(N)
    for n in iterations:
#        H = H_[:n+1, :n+1]; xe = xe_[:n+1]
        H = hilbert(n+1); xe = mat(ones(n+1)).T
        b = H*xe
        x = rotsolve(H, b)
        err[n] = linalg.norm(x - xe, 2)
        y = rotsolveAlt(H, b)
        errAlt[n] = linalg.norm(y - xe, 2)

    plt.plot(1+iterations, err, 'b')
    plt.plot(1 +iterations, errAlt, 'r')
    plt.legend(['Algorithm 1','Algorithm 2'])
    plt.xlabel('size of matrix [n]')
    plt.ylabel('||x - x_e||_2')
    plt.show()
```

.

As noted, there were two implementations of `rotsolve`; I have plotted the er-



ror of both.