Instructions

- There are two pages of questions.
- This examination is open book.
- **You have 2 hours and 20 minutes (including uploading). People with special needs (according to the official information of the Educational support center) have 2 hours and 40 minutes in total.**
- The grade will be computed as the number of obtained points, plus 1.
- Do not communicate with other students during the examination. The pledge you signed is valid for this instance too, so you do not need to sign and upload it again.
- Stay in the online room during the exam. Information delivered there is official and part of the instructions of the examination.
- The exam must be written by hand **in a tidy and legible way**, scanned and uploaded as PDF to Nestor.
- **Upload the PDF in vertical orientation, such that it requires no rotation to be readable.**
- You are supposed to write algorithms using Python language, but syntactic errors (e.g. indentation) are allowed as far as the written algorithm is understood. Please note that having wrong indexing in an array, for instance, it is an algorithmic error, not a syntax error.
- Keep the names of the variables as stated in the question.
- **If you do not follow these instructions you will receive the minimal grade.**
- For all complexity calculations, define your elementary operations as addition, division and multiplication.

1 Determinants (2.5 points)

Consider the invertible matrix $A \in \mathbb{R}^{n \times n}$. Recall that the determinant of such matrix can be computed recursively as:

$$\det(A) = \sum_{j=0}^{n-1} (-1)^j \det(B_{0,j}) A[0,j]$$

with $B_{i,j}$ the matrix built from A by removing the i-th row and j-th column.

1. **(0.5 pt)** Write a function that, receiving a 2×2 matrix as an input, returns the determinant of the matrix.

Solution:

```
def determinant_size2(A):
    det = A[0, 0] * A[1, 1] - A[1, 0] * A[0, 1]
    return det
```

2. **(2 pts)** Write a function that, by receiving a $n \times n$ matrix as an input, returns the determinant of the matrix by calling itself recursively. You need to implement also the construction of the $B_{0,j}$.

Solution:

```python
def determinant(A):
    n = A.shape[0]

    if n == 1:
        # base case: stop the recursion
        det = A
    else:
        # recursion
        det = 0
        for i in range(n):
            B = np.block([A[1:, 0:i], A[1:, i+1:]])
            det += (-1)**i * A[0, i] * determinant(B)

    return det
```

2 Low-rank updates (6.5 points)

Consider an invertible matrix $A \in \mathbb{R}^{n \times n}$ and a vector $\vec{b} = [\beta_1, \ldots, \beta_n]^\intercal$. Consider then the matrix $B = A + \vec{b}\vec{b}^\intercal$.

1. **(2 pts)** Write a function that receives A and $\vec{b}$ and a vector $\vec{c}$ and returns the result of $B\vec{c}$. The algorithm should have a computational cost as close as possible to performing $A\vec{c}$. You cannot use vector or matrix operations, you have to iterate over the components and use scalar operations. Hint: building B and then computing $B\vec{c}$ is not the answer.

Solution:

```python
def mat_vec(A, b, vec):

    # Initialize
    n = A.shape[0]
    result = np.zeros(n)
    dotprod = 0

    # compute dot product b . vec
    for i in range(n):
        dotprod += b[i] * vec[i]

    for i in range(n):
        result[i] = b[i] * dotprod
        for j in range(n):
            result[i] += A[i, j] * vec[j]

    return result
```

2. **(1 pt)** Compute the total number of operations of performing $B\vec{c}$ in terms of n using the algorithm you created. Specify the constants for each of power of n.

Solution:

- **(0.2 pts)** Dot product between $\vec{b}$ and $\vec{c}$: $2n-1$ ops (n multiplications and $n-1$ additions).

- **(0.2 pts)** Scaling of $\vec{b}$ with $(\vec{b} \cdot \vec{c})$: n ops (n multiplications).
- Dot product between one row of A and $\vec{c}$: $2n-1$ ops (n multiplications and $n-1$ additions) **(0.2 pts)** , and for n rows: $2n^2 - n$ **(0.2 pts)**
- **(0.2 pts)** Adding all operations: $2n^2 + 2n - 1$

It can be shown that:

$$B^{-1} = A^{-1} - A^{-1}\vec{b}(1 + \vec{b}^\intercal A^{-1}\vec{b})^{-1}\vec{b}^\intercal A^{-1} \tag{1}$$

3. **(1.5 pt)** Write a function that, by receiving the LU-factors of A, $\vec{b}$ and a vector $\vec{\gamma}$ as input, returns the solution $\vec{x}$ to the system $B\vec{x} = \vec{\gamma}$ using the identity (1). Therefore, the function should not compute the matrices A^{-1}, B, B^{-1} explicitly. Assume that you already have available a function that returns the solution to a linear system by receiving the LU-factors of a matrix and a right-hand-side. You can use vector (not matrix) operations sum, subtraction, dot product, and multiplication/division by a scalar. Hint: if you have a matrix $S = C + D@E$, then you have the indentity $S@g = C@g + D@(E@g)$

Solution:

```python
def lin_sys(L, U, b, rhs):
    # Solve linear system with A and b
    invAb = lin_sys_A(L, U, b)

    # Solve linear system with A and rhs
    invArhs = lin_sys_A(L, U, rhs)

    # Scalar factor
    scalar = np.dot(b, invArhs) / (1 + np.dot(b, invAb))

    # Compute result
    result = invArhs - invAb * scalar
    return result
```

4. **(1 pts)** Compute the number of operations of the algorithm solving the linear system $B\vec{x} = \vec{\gamma}$. Assume that solving the linear systems associated with L and U costs n^2 elementary operations each.

Solution:

- **(0.2 pts)** Solve linear system for L, U and $\vec{b}$: $2n^2$ ops.
- **(0.2 pts)** Solve linear system for L, U and $\vec{\gamma}$: $2n^2$ ops.
- **(0.2 pts)** Compute the scalar factor: two dot products $2n$ each, n sums by one, n divisions. So $4n$ in total.
- **(0.2 pts)** Compute the result: multiplication of vector by scalar n, n substractions. So $2n$ in total.
- **(0.2 pts)** Adding all operations: $4n^2 + 6n$

5. **(1 pts)** If the cost of doing an LU factorization of a matrix in $\mathbb{R}^{n\times n}$ is $(2/3)n^3$, which strategy would lead to a reduced number of operations to solve the linear system $B\vec{x}$ when the LU factors

of A and B still need to be computed? Strategy (i): to compute the full matrix B then compute LU and then solve, or Strategy (ii): the one you developed using identity (1)? What about if A has a sparse structure?

Solution: Computing the full matrix B can be performed in n^2 multiplications of the rank 1 term and then n^2 additions, hence $2n^2$ **(0.5 pts)**. Then the total cost of solving $B\vec{x} = \vec{\gamma}$ would be $2n^2 + 2/3n^3 + 2n^2 = 4n^2 + 2/3n^3$. For the strategy above one obtain $4n^2 + 6n + 2/3n^3$. All cubic and quadratic terms are the same, so both strategies lead to the same results asymptotically **(0.25 pts)**. If A is sparse, then the LU decomposition of A may take much less operations than the one of B and therefore strategy (ii) may be more efficient **(0.25 pts)**.