Homework 4

Homework Submission Workflow

When you submit your work, follow the instructions on the submission workflow page (https://www.cs.duke.edu/courses/fall18/compsci371d/homework/workflow.html) ***scrupulously*** for full credit.

Important: Failure to do any of the following will result in lost points:

- Submit **one** PDF file and **one** notebook per group
- Enter **all the group members** through the Gradescope GUI when you submit your PDF files. It is **not** enough to list group members in your documents
- Match each **answer** (not question!) with the appropriate page in Gradescope
- Avoid large blank spaces in your PDF file

Part 1: Exam-Style Questions

As usual, these questions will be graded like all the others. Please resist the temptation to solve questions by software (other than when explicitly suggested), as you would then miss the training necessary to do well in the exams.

Except when explicitly asked, there is no need to show your work unless you are unsure of your answer and you aim for partial credit.

Problem 1.1

Suppose that the histogram of orientations $\theta \in [0, 180)$ has 18 equal bins, numbered 0 to 17, and with centers $5, 15, \ldots, 175$. Bilinear voting is used, as described in the class notes on Histograms of Oriented Gradients. Give the fraction of vote that each of the bins receives when a measurement $\theta = 42$ is observed. Fractions are values in $[0, 1]$.

Problem 1.2

The x and y components of ∇I at a particular pixel in an image are $10\sqrt{3}$ and 10, respectively. What are the magnitude and orientation of the gradient there? Express orientation in degrees relative to the x axis.

Problem 1.3

A HOG feature is computed from a 128×64 window divided into cells with 8×8 pixels each. The size of a block is 2×2 cells. Central differences are used to compute gradients. Each cell histogram has nine bins of equal size, and bin 0 is for orientations in $[0, 20)$ degrees.

Rather than using a constant ϵ in the denominator during normalization of a histogram $\mathbf{v}$, the following procedure is used at all stages to normalize a histogram $\mathbf{v}$ to $\mathbf{v}'$:

$$\mathbf{v}' = \begin{cases} \frac{\mathbf{v}}{\|\mathbf{v}\|} & \text{if } \|\mathbf{v}\| \neq 0 \\ \mathbf{v} & \text{otherwise.} \end{cases}$$

The threshold τ used to saturate the feature entries during contrast normalization is set to infinity (so that it has no effect).

For a given window, the pixel at position $(5, 5)$ has value 50, and all other pixels are zero.

Give an expression for the entries of the HOG feature $\mathbf{h}$ for that window.

Problem 1.4

The magnitude of a vector $\mathbf{x} = [x, y]^T \in \mathbb{R}^2$ is

$$f(x, y) = \sqrt{x^2 + y^2}\ .$$

Write algebraic expressions and exact decimal numerical values of the gradient ∇f and Hessian H_f of f at $x = 3$ and $y = 4$.

[Hint: $1/5^3 = 1/125 = 8/1000$.]

Problem 1.5

The second stage of line search repeatedly narrows a bracketing triple (a, b, c) established in the first stage. Suppose that at some point during the second stage the bracketing triple is:

$$a = 0 \quad , \quad b = 2 \quad , \quad c = 6$$

with function values

$$h(a) = 6 \quad , \quad h(b) = 3 \quad , \quad h(c) = 4$$

along the search line. Assume further that

$$h(1) = 2 \quad \text{and} \quad h(4) = 5 \; .$$

What is the bracketing triple after that? Standard line search is applied, that is, no golden ratio is used. No need to explain.

[Hint: Draw a sketch.]

Problem 1.6

The gradient and Hessian at $\mathbf{z}_0 = [4, 1]^T$ for some function $f : \mathbb{R}^2 \to \mathbb{R}$ are as follows:

$$\nabla f(\mathbf{z}_0) = \begin{bmatrix} 2 \\ 1 \end{bmatrix} \quad \text{and} \quad H_f(\mathbf{z}_0) = \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix} \; .$$

Find the point $\mathbf{z}_1$ that results by taking a single Newton step from $\mathbf{z}_0$. Show your reasoning briefly.

Problem 1.7

A classification problem asks to find a classifier $h \in \mathcal{H}$ whose domain X and co-domain Y of h are defined as follows:

$$X = \{0, 2, 4, 6, 8, 10\} \quad \text{and} \quad Y = \{0, 1\}$$

and the hypothesis space $\mathcal{H}$ is the set of the following five functions

$$h_k(x) = \begin{cases} 0 & \text{if } x < k \\ 1 & \text{otherwise} \end{cases} \quad \text{for} \quad k = 1, 3, 5, 7, 9 \ .$$

The following tiny training set is given:

$$T = \{(0, 0), (2, 0), (4, 1), (6, 1)\} \ .$$

Make a table of the training risk $L_T(h_k)$ for $k = 1, 3, 5, 7, 9$. Express values in percent. Then use the table to determine the optimal classifier $\hat{h}$ for this training set (thus, we only care about empirical risk here).

Problem 1.8

Assume that the probability model $p(x, y)$ for the data in Problem 1.7 is as specified in the table below. The table contains the joint probability $p(x, y)$ with values expressed in percent, rather than as numbers in $[0, 1]$. It has one column per value of x and one row per value of y.

	0	2	4	6	8	10
0	12	11	10	7	6	4
1	2	4	6	10	12	16

Under the same circumstances as in Problem 1.7, and with the probability model above, make a table of the statistical risk $L_p(h)$ and determine the classifier that generalizes best. Express values in percent.

Notes:

- In an exam, this question would be simplified, so that it would take less time to solve. In this assignment, using code may reduce the likelihood of mistakes. However, do **not** submit your code.
- For any realistic machine learning problem it is impossible to know or estimate the model $p(x, y)$ that generates the data. This is not the case for a problem as tiny as the one given here.

Part 2: Projections

Given a vector space $\mathbb{R}^m$ and a nonzero vector $\mathbf{u}$ in it, the space $\mathcal{O}(\mathbf{u})$ orthogonal to $\mathbf{u}$ is the set of all vectors in $\mathbb{R}^m$ that are orthogonal to $\mathbf{u}$. For instance, if $m = 3$ and the coordinates for $\mathbb{R}^3$ are called x, y, z, then $\mathcal{O}([1, 0, 0]^T)$ is the (y, z) plane.

Suppose now that you are given a vector $\mathbf{u} \in \mathbb{R}^m$. A complicated way to project some new vector $\mathbf{b}$ onto $\mathcal{O}(\mathbf{u})$ would be to find an orthonormal basis for $\mathcal{O}(\mathbf{u})$, and put the basis vectors into the columns of a matrix U, which is then orthogonal: $U^T U = I$. The projection $\mathbf{p}$ of $\mathbf{b}$ onto $\mathcal{O}(\mathbf{u})$ can then be computed by using the formula we saw in class for projecting a vector onto a subspace:

$$\mathbf{p} = P\mathbf{b} \quad \text{where} \quad P = UU^T \ .$$

We will explore both this way and a simpler way to compute $\mathbf{p}$ in this part of the assignment.

Note that a "projection" is still a vector in $\mathbb{R}^m$, and therefore has m entries, even if the space we project onto has fewer dimensions. Study the class notes if this is unclear.

Problem 2.1

What is the shape (number of rows and columns) of U? Explain briefly.

Problem 2.2

Explain clearly how, given $\mathbf{u}$, you can use the Singular Value Decomposition (SVD) to compute a matrix whose columns form an orthonormal basis for $\mathcal{O}(\mathbf{u})$. Also explain why.

Then show a basis for $\mathcal{O}([1, 1, 0]^T)$ using your technique.

Give the projection matrix P for this space, and the projection of the vector $\mathbf{b} = [0, 2, -4]^T$ onto $\mathcal{O}([1, 1, 0]^T)$.

Use whatever code you'd like, but do **not** submit your code. Give your numerical results either exactly (if you can guess what they are) or with four decimal digits after the period.

Do **not** use any other techniques, such as Gram-Schmidt.

Problem 2.3

For each of the two matrices U and P and the vector $\mathbf{p}$ you found in the previous problem, state whether the object is unique, and explain briefly why or why not.

Problem 2.4

A simpler way to compute the projection $\mathbf{p}$ of a vector $\mathbf{b}$ onto $\mathcal{O}(\mathbf{u})$ is based on the observation that the vector $\mathbf{b} - \mathbf{p}$ is parallel to $\mathbf{u}$, because it is orthogonal to $\mathcal{O}(\mathbf{u})$, as we saw in class and in the class notes.

Therefore, one can compute $\mathbf{p}$ by subtracting from $\mathbf{b}$ the projection of $\mathbf{b}$ along $\mathbf{p}$. If

$$\mathbf{n} = \frac{\mathbf{u}}{\|\mathbf{u}\|}$$

is the unit vector that points in the same direction as $\mathbf{u}$, then

$$\mathbf{p} = \mathbf{b} - \mathbf{n}\mathbf{n}^T\mathbf{b} = P\mathbf{b} \quad \text{where} \quad P = I - \mathbf{n}\mathbf{n}^T \ .$$

Compute the matrix

$$Q = I - \mathbf{n}\mathbf{n}^T$$

for the example given in Problem 2.2.

Show Q if it is different from the matrix P you obtained through SVD. Otherwise, just state that the two matrices are the same.

Remark

Computing the matrix $Q = I - \mathbf{n}\mathbf{n}^T$ and then multiplying Q by $\mathbf{b}$ is wasteful, because the product requires m^2 multiplications and almost as many sums.

A more efficient way is to compute the product as follows:

$$\mathbf{p} \;=\; \mathbf{b} \;-\; \mathbf{n}\,(\mathbf{n}^T\mathbf{b})\;.$$

This expression only uses vector-vector operations, and involves no matrices. It requires $2m$ multiplications and almost as many sums.

Part 3: Optimization and the SVD

Steepest descent is an algorithm that can be used to find a local minimum of an unconstrained function $f(\mathbf{z}) \;:\; \mathbb{R}^m \to \mathbb{R}$, starting from an initial point $\mathbf{z}_0$.

This part describes a variant of steepest descent that is modified in two ways:

- It prevents the algorithm from stopping at a local maximum or saddle point. At these points the gradient of f is zero, and the search direction is consequently undefined. There are various ways in which the algorithm can encounter a stationary point other than a minimum. The most obvious is that $\mathbf{z}_0$ itself is either a maximum or a saddle.
- It minimizes f *on the unit sphere*, that is, under the constraint that $\|\mathbf{z}\| = 1$. This constraint arises in many mathematical problems, and we will see an example later on that relates to the SVD.

The description of the function `sphere_step` is given so you understand what is going on when you are asked to *use* this function to solve a constrained minimization problem.

The function `sphere_step` below performs a single step of steepest descent, taking care not to stop at maxima or saddle points. It takes a function f, its gradient g, and a current point $\mathbf{z}_0$. It returns the next point $\mathbf{z}_1$, under ther constraint that both these points are unit-norm vectors. More specifically, the domain of f is the unit sphere in $\mathbb{R}^m$ with $m > 1$.

This function takes the following arguments:

- A function `f` and `g` with signatures

```
f(z, *args)
```

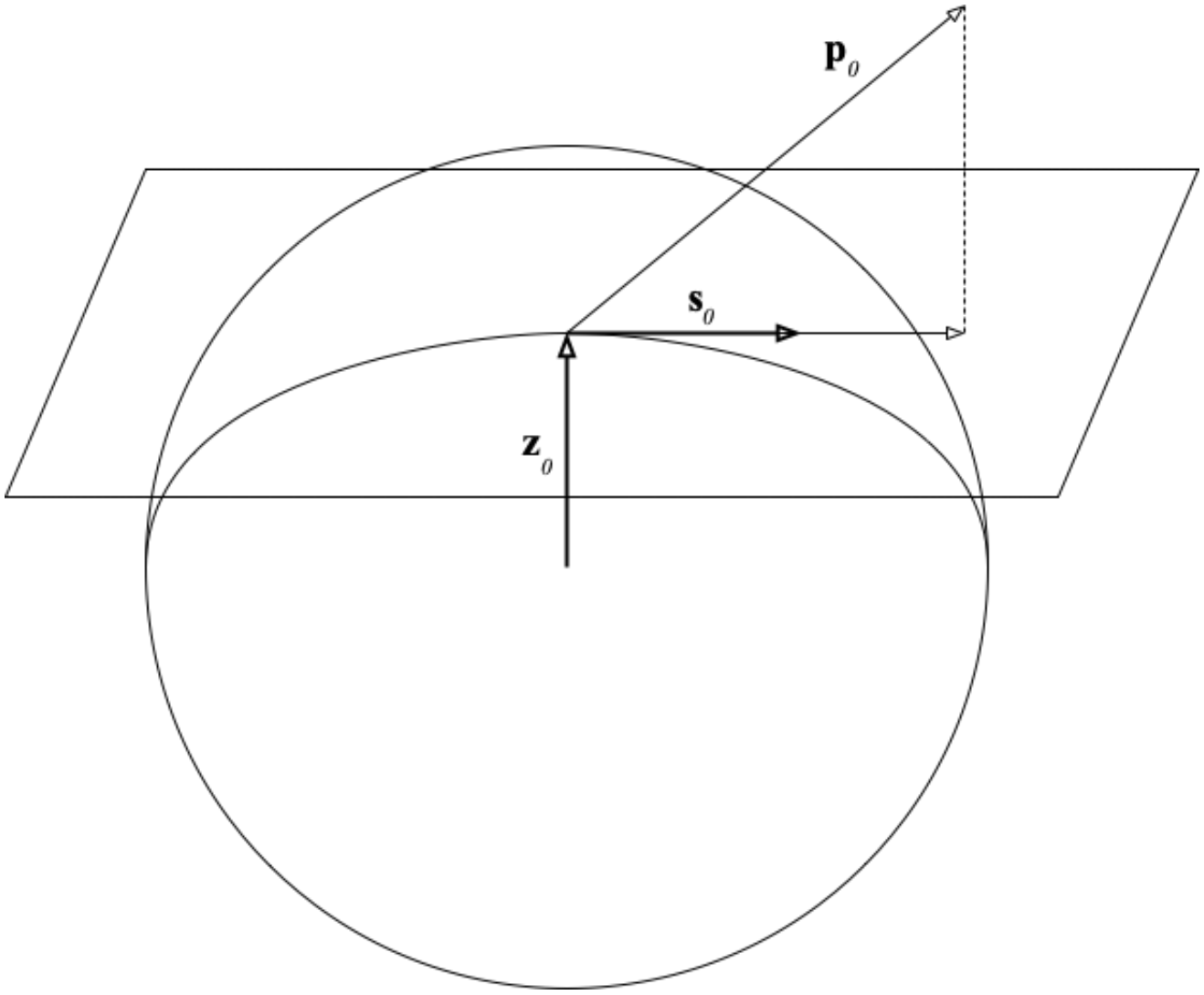
```
g(z, *args)
```

where `z` is a one-dimensional `numpy` array and arguments in `args` are optional. The function `f` is the real-valued function to be minimized, and `g` is its gradient. If the tuple `args` passed to `sphere_step` is not empty, its elements are passed to `f` and `g` when these functions are called.
- An initial point `z0`, which is a one-dimensional `numpy` array of the same size as the first argument ot `f` and `g`.
- An optional floating-point value `epsilon` between 0 and 1 excluded. Quantities below this value are to be regarded as zero in the presence of numerical inaccuracies.
- An optional tuple `args` of parameters to be passed to `f` and `g`.

Constraining the Function to the Unit Sphere

To enforce the unit-sphere constraint, the function first normalizes the starting point $\mathbf{z}_0$ to have unit norm. This normalization is merely precautionary, as $\mathbf{z}_0$ is *supposed* to be on the unit sphere.

After this, the function computes the negative gradient $\mathbf{p}_0$ of f at $\mathbf{z}_0$, and projects $\mathbf{p}_0$ onto the tangent plane of the unit sphere at $\mathbf{z}_0$. This plane is the space orthogonal to $\mathbf{z}_0$, so that the projection can be found by the method developed in Problem 2.4. Let $\mathbf{s}_0$ be the projection, normalized to have unit length, as shown in the figure below.



The function then uses line search, implemented in Python by the function `scipy.optimize.minimize_scalar`, on the following function:

$$h(\theta) = f(\mathbf{z}_0 \cos\theta + \mathbf{s}_0 \sin\theta) \ .$$

Since both $\mathbf{z}_0$ and $\mathbf{s}_0$ have unit norm, all values of θ yield a point on the unit sphere. The unit vector $\mathbf{s}_0$ is the vector on the tangent plane at $\mathbf{z}_0$ that is closest to the true descent direction $\mathbf{p}_0$, and doing line search on h amounts to doing "circle search" on the plane spanned by $\mathbf{z}_0$ and $\mathbf{p}_0$.

Avoiding Maxima and Saddle Points

To avoid stopping at non-mimimum stationary points (that is, at maxima or saddles), the function first checks whether the norm of the projection of $\mathbf{p}_0$ (the negative gradient of the unconstrained f) onto the unit sphere is below `epsilon`. If it is, then small changes away from $\mathbf{z}_0$ while staying on the unit sphere yield negligible changes in the value of f. In other words, the gradient of f at $\mathbf{z}_0$ is zero when $\mathbf{z}$ is constrained to be on the unit sphere: We are (approximately, to within `epsilon`) at a stationary point of f on the unit sphere.

To tell whether this stationary point is a minimum or not, the function perturbs $\mathbf{z}_0$ by adding a small amount

of random noise, and keeps doing so until the norm of $\mathbf{s}_0$ is greater than `epsilon`.

The function then performs "circle search" as described above. If $\mathbf{z}_0$ is a genuine minimum, this circle search will fall back onto $\mathbf{z}_0$. If not, the search will take a step to a local minimum on the search circle, thereby getting "unstuck" from the maximum or saddle point.

In [1]:
```
import numpy as np
from scipy.optimize import minimize_scalar

def normalize(v):
    n = np.linalg.norm(v)
    if n > 0:
        v = v / n
    return v, n

# sphere_search returns a new point z1, not a search step size
def sphere_step(f, g, z0, epsilon=1.e-6, args=()):
    assert epsilon > 0 and epsilon < 1, 'epsilon must be in (0, 1)'
    assert z0.size > 1, 'sphere must be at least 2-dimensional'

    n0, delta = 0, 0
    while n0 < epsilon:
        z0 += delta * np.sqrt(epsilon) * np.random.random(z0.shape)
        z0, _ = normalize(z0)
        p0 = -g(z0, *args)  # Starting direction
        # Project p0 onto the tangent hyperplane to the sphere at z0 a
nd normalize
        s0, n0 = normalize(p0 - np.dot(z0, p0) * z0)
        delta = 1

    def h(theta, args):
        return f(z0 * np.cos(theta) + s0 * np.sin(theta), args)

    res = minimize_scalar(h, bracket=(0, 1), args=args)
    theta = res.x
    return z0 * np.cos(theta) + s0 * np.sin(theta)
```

Problem 3.1

We saw that the first singular value σ_1 of a nonempty, nonzero matrix A is

$$\sigma_1 = \max_{\|\mathbf{v}\|=1} \|A\mathbf{v}\| .$$

Note that $\mathbf{v} \in \mathbb{R}^m$ is *not* required to be in the rowspace of A. However, when the maximum is found, the value of $\mathbf{v}$ where the maximum is found *is* in the rowspace of A, and is equal to the first right singular vector $\mathbf{v}_1$ of A. The corresponding left singular value is then

$$\mathbf{u}_1 = \frac{A\mathbf{v}_1}{\|A\mathbf{v}_1\|}$$

and the value of σ_1 is equal to the denominator in this expression.

Thus, rather than running the standard SVD algorithm, the first singular value and corresponding singular vectors of A can be found by maximizing a function on the unit sphere or, equivalently, by minimizing the function

$$f(\mathbf{v}) = -\|A\mathbf{v}\|^2$$

over the unit sphere.

Write a function with the following header and `assert` statements:

```
def first(A, epsilon=1.e-6, maxiter=10):
    assert A.size > 0, 'array cannot be empty'
    assert np.max(np.abs(A)) >= epsilon, 'array cannot be zero'
```

that uses `sphere_step` iteratively to return the tuple `u, sigma, v` of the first left singular vector, first singular value, and first right singular vector of `A`.

Show your code and the results of running the given tests. You may want to compare your results with those obtained with an off-the-shelf implementation of the SVD. However, do **not** submit your comparison, and do not use SVDs anywhere in your code.

Programming Notes

- The maximum number of iterations is `maxiter`, and `epsilon` is passed to `sphere_step`. This value is also used as a termination condition in `first`: When the Euclidean norm of $\mathbf{v}^{(k)} - \mathbf{v}^{(k-1)}$ (new point minus old point) falls below `epsilon` (or `maxiter` iterations are exceeded), the computation stops and reports the result. For simplicity, we will report the result even if `maxiter` is exceeded. Ten iterations are enough for the small test problems given below.
- Your function `first` must also define a function f (as explained above) and its gradient g. These functions have the following signatures:

  ```
  def f(u, A):

  def g(u, A):
  ```
- For the parameter `epsilon` and the matrix `A` to be passed properly to `f` and `g`, you need to call `sphere_step` with optional arguments

  ```
  epsilon=epsilon, args = (A,)
  ```
- The returned vectors `u` and `v` must be `numpy` arrays with one dimension each (that is, `u.ndim == 1` and `v.ndim == 1` must be `True`), and `sigma` is a floating-point number (not a `numpy` array). Doing so will make it easier to do the next problem in this assignment.
- Since `sphere_step` requires the domain of the function to be minimized to be at least two-dimensional (see the `assert` statements at the beginning of that function), the cases when `np.min(A.shape) < 2` need to be handled separately. Make sure that your outputs have the proper format, as specified in the previous bullet. [Hint: What is the SVD of a row vector or a column vector?]

Problem 3.2

If $m \times n$ matrix A has rank r, then the "tiny" SVD $A = U\Sigma V^T$ has matrices of the following sizes:

- U is $m \times r$
- Σ is $r \times r$
- V is $n \times r$

Thus, only bases for the rowspace and the range of A are given in V and U, and only the nonzero singular values are listed on the diagonal of Σ.

The "tiny" SVD of A can be computed by noticing that

- $\mathbf{u}_1$, σ_1 , $\mathbf{v}_1$ can be found with `first`.
- σ_2 is the maximum of $A\mathbf{v}$ over all vectors $\mathbf{v}$ that are in the space $\mathcal{O}(\mathbf{v}_1)$ orthogonal to $\mathbf{v}_1$. Equivalently, if A is overwritten with $A - A\mathbf{v}_1\,\mathbf{v}_1^T$ (that is, by the matrix formed by the projection of the rows of A on the space orthogonal to $\mathbf{v}_1$), then

$$\sigma_2 = \max_{\|\mathbf{v}\|=1} A\mathbf{v} \ ,$$

and so $\mathbf{u}_2$, σ_2 , $\mathbf{v}_2$ can be computed by running `first` on this new matrix A. This reasoning can be repeated.

- You know that you are done when A is all zeros.

Incidentally, this implementation of the SVD is **not** a good implementation from a numerical standpoint. It is introduced here for pedagogical reasons, and to characterize the SVD as the solution to an optimization problem.

Use your function `first` to write a function with the following header and `assert` statement that computes the "tiny" SVD of a non-empty matrix A, with the given specifications:

```
def SVD(A, epsilon=1.e-3, maxiter=10):
    assert A.size > 0, 'array cannot be empty'
```

The function should return the tuple `U, Sigma, V` of the matrices in the "tiny" SVD of `A`, and should comply with the programming notes.

Show your code and the results of running the given tests. You may want to compare your results with those obtained with an off-the-shelf implementation of the SVD. However, do **not** submit your comparison. Do not use library SVDs anywhere in your code.

Programming Notes

- The output matrices should all have 2 dimensions, that is, `U.ndim == 2`, `Sigma.ndim == 2`, and `V.ndim == 2` should all be `True`, even when one or both of the dimensions are equal to 1.
- Compute the projection $A - ((A\mathbf{v}_1)\mathbf{v}_1^T)$ so as to minimize the number of operations, as explained in the remark at the end of part 2. The parentheses above suggest this order.
- The matrix A is zero numerically when `numpy.max(numpy.abs(A)) < epsilon`