**First Name:** **Signature:**

**Last Name:**

**SCIPER ID:** _____

---

EPFL

# Final exam

## CS328 - Numerical Methods in Visual Computing

## Procedures:

This is a timed examination. You will have a maximum of **2 hours** to complete your answers. Leave your **student ID card** on the table during the entire time so that it can be verified. Write your **first and last name** as well as your **EPFL SCIPER number** on the cover page as well as every additional sheet of paper that you request during the exam. Use only black or blue ink pens and write legibly. Exam questions done in pencil or other colored pens will **NOT** be graded. Write your solutions into the space below answers and indicate if an answer is continued elsewhere.

Multiple-choice items are each worth one point, while incorrect answers deduct one point. The final score of multiple-choice questions is clamped to a non-negative number. Please try to follow Python syntax when answering questions that ask for code. Minor syntax errors are tolerated.

## Regulations:

With the exception of one (1) double-sided A4 page of personal notes, any use of textbooks or other books/printed materials, formula sheets, electronic devices such as calculators, laptops, cell phones or other PDAs, MP3 players, headphones, etc. is strictly **PROHIBITED** during the examination without express permission from EPFL.

A student is deemed to have failed the course if he or she is caught cheating or found to be in violation of the above regulations.

| *Exercise* | *Max. points* | *Earned points* |
|---|---|---|
| 1. Multiple-Choice | 30 | |
| 2. SVD and Eigenvectors | 12 | |
| 3. Numerical Code and Processors | 16 | |
| 4. Linear System Construction | 20 | |
| 5. Automatic Differentiation | 32 | |
| **Total** | **110** | |

---

| | | | |
|---|---|---|---|
| **Course Name:** | Numerical Methods in Visual Computing | **Date:** | 31.01.2024 |
| **Course Number:** | CS 328 | **Time:** | 9:15-11:15 |
| **Lecturer:** | Wenzel Jakob | | |

# 1 Multiple-Choice (30 pt)

Subsections of this section are *separately* graded as *multiple choice* questions (see page 1).

## 1.1 Questions about the IEEE 754 representation (6 pts)

| True | False | |
|---|---|---|
| ☑ | ☐ | IEEE-754 values can only exactly represent numbers that can be written as a fraction with a power-of-two denominator. |
| ☐ | ☑ | Floating point addition and multiplication satisfy $(a+b)\cdot c = (a\cdot c)+(b\cdot c)$. |
| ☑ | ☐ | The following equality always holds: $(a+b)+(c+d) = (d+c)+(b+a)$. |
| ☐ | ☑ | The relative numerical error of a function implemented using double precision is on average half as large as one that uses single precision. |
| ☐ | ☑ | Denormalized numbers should be avoided in calculations, since operations involving them suffer from severe numerical errors. |
| ☑ | ☐ | The floating point multiplication operation of modern processors runs much faster than floating point division. |

## 1.2 Time complexity (16 pts)

Given an invertible matrix $\mathbf{A}$ with $n$ rows and columns, what is the computational complexity of the following operations?

1. Computing the inverse of $\mathbf{A}$.
2. Computing a matrix-vector product $\mathbf{Ax}$.
3. Computing the condition number of $\mathbf{A}$.
4. Solving a linear system of the form $\mathbf{Ax} = \mathbf{b}$ if a LU factorization is available.

   When a LU factorization does not make the problem any easier, give the complexity for solving the linear system without one.

Use $\mathcal{O}(..)$ notation to fill each cell below with the *lowest possible* time complexity for the following classes of matrices: *general* (nothing known a priori), *orthogonal*, *diagonal*, and *permutation* matrices. You can assume that the information in the matrix is encoded in the most efficient manner. It's fine to just write the text within the parentheses of $\mathcal{O}(..)$. You do *not* need to consider advanced (e.g. Strassen multiplication) or galactic algorithms that were briefly mentioned at some point during CS328.

| | General matrix | Orthogonal matrix | Diagonal matrix | Permutation matrix |
|---|---|---|---|---|
| 1. | $\mathcal{O}(n^3)$ | $\mathcal{O}(n^2)$ | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ |
| 2. | $\mathcal{O}(n^2)$ | $\mathcal{O}(n^2)$ | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ |
| 4. | $\mathcal{O}(n^3)$ | $\mathcal{O}(1)$ | $\mathcal{O}(n)$ | $\mathcal{O}(1)$ |
| 3. | $\mathcal{O}(n^2)$ | $\mathcal{O}(n^2)$ | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ |

### 1.3 Polynomials (1pt)

If we try to approximate a *rectangle function*

$$f(x) = \begin{cases} 1, & |x| < 1, \\ 0, & \text{otherwise}, \end{cases}$$

by fitting a low-degree polynomial to only a few evenly spaced points $x_1, \ldots, x_n$, we find that this produces a bad and oscillatory result. This problem can be solved by (circle one):

Choice

- ◯ Increasing the number of points used in the fit.
- ◯ Increasing the number of points and the degree of the polynomial.
- ◯ Increasing the number of points, degree, and using Chebyshev points.
- ⊘ This problem cannot be solved.

### 1.4 Quadrature (1pt)

Approximate the following integral using the basic trapezoid rule (i.e., using only a single trapezoid):

$$\int_{-1}^{1} \frac{3}{2+x}\,\mathrm{d}x$$

Choice: ◯ 0 ◯ 2 ◯ 3 ⊘ 4

### 1.5 Conditioning (3pt)

Which of the following matrices are poorly conditioned or singular? (in the sense that we expect a failure or significant error amplification when using **A** to solve a linear system)

$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} -1^{-1} & 10^{-10} \\ 10^{-10} & 10^{-1} \end{bmatrix} \quad \mathbf{C} = \begin{bmatrix} 1 & 10^{-10} \\ 1 & 1 \end{bmatrix}$$

| True | False | |
|---|---|---|
| ☑ | ☐ | Matrix **A** |
| ☐ | ☑ | Matrix **B** |
| ☐ | ☑ | Matrix **C** |

### 1.6 Derivatives (3pt)

| True | False | |
|---|---|---|
| ☐ | ☑ | Reverse-mode differentiation is only possible when one can run the computer program to be differentiated in reverse. |
| ☑ | ☐ | Analytic integration of a function by applying the rules of calculus can sometimes fail to produce an antiderivative. |
| ☐ | ☑ | When a function has a single input and output, then forward- and reverse-mode differentiation reduce to the same algorithm. |

## 2 SVD and Eigenvectors (12 pts)

### 2.1 Singular Value Decomposition of a 2×2 matrix (6 pts)

Consider the following $2 \times 2$ matrix:

$$A = \begin{bmatrix} 2\cos\theta & -\sin\theta \\ 2\sin\theta & \cos\theta \end{bmatrix}$$

for some unknown value $\theta \in \mathbb{R}$. Specify the *singular value decomposition* (SVD) of **A** below. Note that we never covered the internals of algorithms to compute an SVD in CS328. Instead, you can find the answer by thinking about what the matrix **A** *does* when applied to a vector, and how that relates to the information encoded in the SVD.

The SVD captures how the matrix transforms a unit circle into an ellipse/ellipsoid. The **A**-matrix has a simple geometric interpretation: it scales the $x$-axis by 2 and then performs a rotation by $\theta$, which means that the scaling of this ellipse is described by the singular values

$$\Sigma = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}.$$

(In this order, the SVD mandates it.) The **V**-matrix (right singular vector) describes what vector gets scaled by 2, and which one by 1. h The **U**-matrix gives the axes following the transformation. Thus,

$$\mathbf{V}^T = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \text{and} \quad \mathbf{U} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \quad \text{and}$$

### 2.2 SVD: Shape and Storage (4 pts)

Consider the (*full* / non-economy) SVD of a matrix $A$ of shape $n \times m$. Give an expression for $A$ in terms of this decomposition and denote the shape of all matrices. (2 pts)

$$\underbrace{A}_{n\times m} = \underbrace{\mathbf{U}}_{n\times n}\underbrace{\Sigma}_{n\times m}\underbrace{\mathbf{V}^T}_{m\times m}$$

Assuming that the entries of $A$ are stored in single precision, how many bytes of memory are needed to store $A$? How many bytes are needed to store the nonzero entries of the full SVD? (2 pts)

**A** : $4nm$ bytes, SVD: $4(n^2 + m^2 + \min(n, m))$.

### 2.3 Power Iteration (2 pts)

Give a concrete example of for a matrix $\mathbf{A} \in \mathbb{R}^{2\times 2}$ where the power iteration will generally never converge when started with a random initial vector. Explain why this happens (1-2 sentences are enough, this question just happens to be on a partially empty page.)

Two possible answers:

$$\mathbf{A}_1 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \qquad \mathbf{A}_2 = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

$\mathbf{A}_1$ swaps the $x$ and $y$ coordinates. It has eigenvalues $\lambda = \pm 1$ with equal magnitude, so power iteration oscillates between the two eigenspaces rather than converging to a dominant eigenvector.

$\mathbf{A}_2$ is a $90^\circ$ rotation matrix with eigenvalues $\lambda = \pm i$ (purely imaginary). Power iteration using real arithmetic cannot represent the complex eigenvectors, so it will cycle indefinitely without converging.

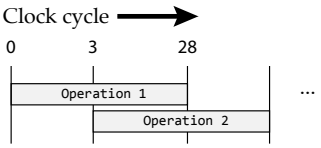
# 3 Numerical Code and Processors (16 pts)

In this exercise, we will revisit how processors execute numerical programs. We will work with a processor that only has two instructions:

- `LOAD(a)`, which loads a value from memory address `a` and stores the result in a register. `LOAD` has a *latency* of **100** clock cycles and a *throughput* of **1** per cycle.
- `ADD(a, b)`, which adds the values of two registers and returns the result. `ADD` has a *latency* of **3** clock cycles and a *throughput* of **1** per cycle

At each clock cycle, the CPU inspects the current instruction and does one of two things: when all of the inputs of the operation are available, it immediately starts the operation and advances to the next program instruction. Otherwise, it waits for all inputs to be ready and *does not* advance to the next instruction. Registers (`r0`, `r1`, etc.) and memory addresses (`a0`, `a1`, etc.) are always available, and accessing them has no cost. There is no limit to how many operations the CPU can execute in parallel except for the discussed constraints on latency, throughput, and the inputs of operations being ready. The CPU has no cache, so each `LOAD` operation takes the same amount of time.

In the following, you can answer questions either in textual form, or as a timeline as shown on the right (as you prefer).




## 3.1 Cycle counting

(4 pts) Consider a program that determines the sum of 4 values stored at memory addresses `a0`, `a1`, `a2`, and `a3`.

```
r0 = LOAD(a0)
r1 = LOAD(a1)
r2 = ADD(r0, r1)
r3 = LOAD(a2)
r4 = ADD(r2, r3)
r5 = LOAD(a3)
r6 = ADD(r4, r5)
```

How many clock cycles are needed until the final result `r6` is available?

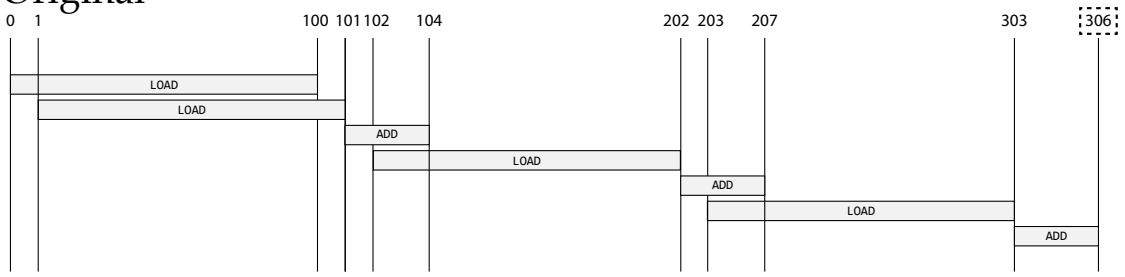
306 clock cycles. See the next page for a timeline.

### 3.2 Optimizations (10 pts)

Implement a better version of this program that runs in the minimal amount of time. Provide both the source code of the program and the combined number of cycles.

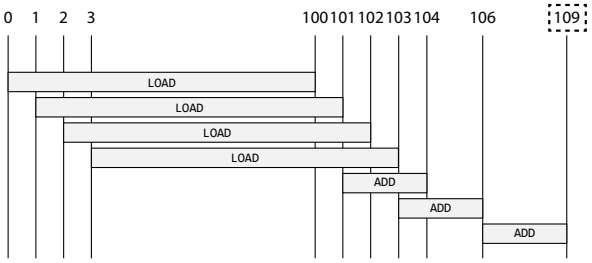
The improved program runs in only 109 clock cycles:

Original




Better:

```
r0 = LOAD(a0)
r1 = LOAD(a1)
r2 = LOAD(a2)
r3 = LOAD(a3)
r4 = ADD(r0, r1)
r5 = ADD(r2, r3)
r6 = ADD(r4, r5)
```




### 3.3 Cache (2 pts)

Suppose now that the `LOAD` instruction benefits from the presence of a cache. The values `a0`, `a1`, `a2`, and `a3` are all directly next to each other in memory, so loading any one of them causes all to land in the cache.

The latency of a `LOAD` operation drops to just `1` cycle when the referenced memory region is cached at the time when the `LOAD` operation starts. How much faster does the original algorithm become? How much faster does your optimized version become?

Loads 3 and 4 of the original version benefit from caching. The critical path now becomes: LOAD1+2 (101 cycles), followed by 3 ADDs that are serialized because of interdependencies (9 cycles), which adds up to 110.
There is no benefit for the optimized version of the algorithm since we are bottlenecked by the first load, so it still takes 109 cycles.

# 4 Linear System Construction (20 pts)

## 4.1 Constrained least squares (10 pts)

You are well-acquainted with methods to solve *linear least squares* problems, e.g. to obtain a solution $\mathbf{x}_{\text{sol}} \in \mathbb{R}^3$ given a matrix $\mathbf{A} \in \mathbb{R}^{n\times 3}$ and right-hand side $\mathbf{b} \in \mathbb{R}^n$, where

$$\mathbf{A} = \begin{bmatrix} | & | & | \\ \mathbf{a}_1 & \mathbf{a}_2 & \mathbf{a}_3 \\ | & | & | \end{bmatrix} \quad \text{and} \quad \mathbf{x}_{\text{sol}} = \operatorname*{argmin}_{\mathbf{x}\in\mathbb{R}^3} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|.$$

Suppose that we have additional information about the problem that is not yet reflected in the above equation: valid solutions *must* satisfy $x_3 = 1$, and other answers are not acceptable. Specify the linear system that must be solved to obtain $\mathbf{x}_{\text{sol}}$ satisfying this constraint *after* transformation by the normal equations.

$$\operatorname*{argmin}_{\mathbf{x}} \left\| \mathbf{A} \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix} - \mathbf{b} \right\| = \operatorname*{argmin}_{\mathbf{x}} \left\| \underbrace{\begin{bmatrix} | & | \\ \mathbf{a}_1 & \mathbf{a}_2 \\ | & | \end{bmatrix}}_{=\hat{\mathbf{A}}} \underbrace{\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}}_{=\hat{\mathbf{x}}} - (\mathbf{b} - \mathbf{a}_3) \right\|,$$

and therefore

$$\hat{\mathbf{x}} = \mathbf{B}^{-1}\hat{\mathbf{A}}^T(\mathbf{b} - \mathbf{a}_3), \quad \text{where} \quad \mathbf{B} = \hat{\mathbf{A}}^T\hat{\mathbf{A}} = \begin{bmatrix} \mathbf{a}_1^T\mathbf{a}_1 & \mathbf{a}_1^T\mathbf{a}_2 \\ \mathbf{a}_2^T\mathbf{a}_1 & \mathbf{a}_2^T\mathbf{a}_2 \end{bmatrix}.$$

## 4.2 Polynomial fitting with derivatives (10 pts)

Consider the following cubic polynomial

$$p(x) = ax^3 + bx^2 + cx + d.$$

Our goal is to find the coefficients $a$, $b$, $c$, and $d$ so that the polynomial not only passes through two given points, but that its derivative also takes on a particular value at these locations. In other words, we want that

$$p(x_1) = y_1, \quad p(x_2) = y_2, \quad p'(x_1) = d_1, \quad \text{and } p'(x_2) = d_2.$$

Express this as an $\mathbf{A}\mathbf{x} = \mathbf{b}$-style linear system and give the specific entries of every term.

$$\begin{bmatrix} x_1^3 & x_1^2 & x_1^1 & 1 \\ x_2^3 & x_2^2 & x_2^1 & 1 \\ 3x_1^2 & 2x_1^1 & 1 & 0 \\ 3x_2^2 & 2x_2^1 & 1 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ d_1 \\ d_2 \end{bmatrix}$$

# 5 Automatic differentiation (32 pts)

In this section, your task is to extend the *MyTorch* framework from homework 4 with differentiable `load` and `store` operations that indirectly access another array:

- The function `load(src, idx)` fetches multiple entries from `src` as specified by an integer array `idx`:

```
def load(src, idx):
    return src[idx]
```

- The function `store(dst, idx, val)` first copies the input array `dst` and then returns a modified version, where entries with indices `idx` have been overwritten by corresponding values in `val`.

```
def store(dst, idx, val):
    tmp = array(dst) # Copy input
    tmp[idx] = val
    return tmp
```

The helper functions `array()` and `index()` construct or copy a floating point or integer array. Here is an example use:

```
a0 = array([5.0, 6.0, 3.0, 2.0])
a1 = load(a0, index([1, 2, 0]))
a2 = store(a1, index([1, 0]), array([2.0, 1.0]))
# At this point, we have
# a1 == array([6.0, 3.0, 5.0]) and a2 == array([1.0, 2.0, 5.0])
```

What remains to be done are the forward and backward derivative functions. Recall that these receive the original function arguments, and the backward derivative additionally receives the previously computed function output `out`. Their task is to use the `.grad` field of relevant arguments to compute and return a suitable derivative per input or output. For simplicity, you may assume that the entries of the `idx` array do not contain duplicate indices. The derivative of an integer array is always zero. In the case of `store()`, both the target `dst` and the source `val` may carry derivatives.

### 5.1 (a). Forward derivative of `load()` (2 pts)

```
@mytorch_fwd
def load(src, idx);
    return src.grad[idx]
```

### 5.1 (b). Backward derivative of `load()` (6 pts)

```
@mytorch_bwd
def load(src, idx, out);
    tmp = array(src.grad)
    tmp[idx] += out.grad
    return tmp, 0
```

**5.1 (c). Forward derivative of `store()` (2 pts)**

```
@mytorch_fwd
def store(dst, idx, val);
    tmp = array(dst.grad)
    tmp[idx] = val.grad
    return tmp
```

**5.1 (d). Backward derivative of `store()` (10 pts)**

```
@mytorch_bwd
def store(dst, idx, val, out);
    grad_val = out.grad[idx]
    grad_dst = array(out.grad)
    grad_dst[idx] = 0
    return grad_dst, 0, grad_val
```

**5.1 (e). Why could duplicate indices be problematic during differentiation? (2 pts)**

Possible answers (getting one of these is enough): In `store()`, these competing indices would overwrite each other. For correctness, the forward and reverse mode derivative would need to reproduce the somewhat arbitrary choice of which element "wins", which may not be efficiently possible. If the computation runs in parallel, that arbitrary decision may even be non-deterministic. In the reverse-mode derivative of gather, the implementation would have to combine multiple gradients reaching the same index.

**5.2. Reverse-mode tape traversal (10 pts).** The left code fragment implements the function $(\mathtt{a},\mathtt{b}) \mapsto e^{(\mathtt{ab})/2} + \mathtt{b}$ using temporaries `c`, `d`, `e`, and `f`.

Suppose that this computation was recorded using tape-based AD. Complete the resulting derivative computed by a reverse-mode tape traversal on the right, which processes associated derivative variables $\delta\mathtt{a} \ldots \delta\mathtt{f}$ that have been zero-initialized. Reuse previously computed (primal) program variables whenever this is possible.

```
c = a · b
d = c / 2
e = exp(d)
f = e + b
```

$\delta$f = 1
$\delta$e += 1·$\delta$f = 1
$\delta$b += 1·$\delta$f = 1
$\delta$d += e·$\delta$e = e
$\delta$c += $\frac{1}{2}$ · $\delta$d = e/2
$\delta$a += b·$\delta$c = be/2
$\delta$b += a·$\delta$c = 1 + ae/2