First Name:

Last Name:

SCIPER ID:

Signature:

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 every sheet of paper. 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 on these sheets and indicate if an answer is continued on another page.

Multiple-choice items are each worth one point, while incorrect answers deduct one point. The final score of multiple-choice question is clamped to a non-negative number.

Regulations:

With the exception of one (1) page of personal notes written on a A4 page, 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.

The 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. Floating point arithmetic	25	
2. Large-scale computations	25	
3. Linear systems	25	
4. Nonlinear methods	25	
Total	**100**	

Course Name:	Numerical Methods in Visual Computing	**Date:**	27.01.2017
Course Number:	CS 328	**Time:**	8:15-11:15
Lecturer:	Wenzel Jakob		

Fall 2016 exam solution

NumPy/SciPy command reference

The following set of NumPy/SciPy commands may come in handy when completing the Python programming portions of this exam:

The "at" (`@`) operator. Given two correctly-shaped 2D arrays `A` and `B`, the expression `A @ B` performs a matrix multiplication. If the second part is a 1D vector `b`, the expression `A @ b` instead performs a matrix-vector product.

Summing over arrays. The expression `np.sum(A)` sums over all entries of a NumPy array $\mathbf{A} \in \mathbb{R}^{m \times n}$ and returns the resulting scalar. The command `np.sum(A, axis=i)` only sums along the i-th axis of the array; for instance, `np.sum(A, axis=0)` returns a n-dimensional vector containing sums of rows, and `np.sum(A, axis=1)` returns a m-dimensional vector containing sums of columns.

QR decompositions. The following statement computes the QR decomposition of a matrix `A` and records the resulting orthogonal matrix `Q` and triangular part `R`.

```
Q, R = la.qr(A, mode = 'economic')
```

Solving triangular systems. The following statement solves a triangular linear system of the form $\mathbf{Mx} = \mathbf{z}$. The placeholder "`???`" should be replaced with `True` if the system is lower triangular and `False` if it is upper triangular.

```
x = la.solve_triangular(M, z, lower=???)
```

Fall 2016 exam solution

1 Floating point arithmetic (25 pts)

1.1 Multiple choice (5 pts)

The following multiple-choice questions cover characteristic properties of floating point computations following the IEEE-754 standard. Circle all items that are true.

A. Signed 32 bit integers are exactly representable as signed 32 bit (single precision) floating point numbers.

B. A loop of the form

```
i = 0.0
while i < N: # N is a positive floating point value
    i += 1
```

may not terminate even if we can assume that `N` is a finite.

C. On current processors, arithmetic involving denormalized floating point values generally runs slower than arithmetic involving normalized values.

D. Transcendental operations (`np.sin()`, `np.cos()`, etc.) provide the same accuracy guarantees as elementary operations such as addition or multiplication.

E. Pivoting strategies used by matrix decompositions reorder the rows and/or columns of a linear system to avoid dividing by values of small magnitude, thereby increasing numerical stability.

Solution: B, C, E

1.2 Arithmetic rules (7 pts)

Which of the following statements are guaranteed to hold when implemented in IEEE-754 floating point arithmetic regardless of which floating point constants `x` and `y` are provided? Assume that no overflow/underflow occurs and that no denormalized values are encountered during the calculation. (circle your choices)

A. `x+y == y+x`

B. `x+x == 2*x`

C. `x + y - y == x`

D. `x / y == x * (1 / y)`

E. `(x / 2) * 2 == x`

F. `(x * 2) * (y / 2) == x * y`

G. `(x-y)*(x+y) == x*x - y*y`

Solution: A, B, E, F

Fall 2016 exam solution

1.3 Hypotenuse (13 pts)

Most modern mathematical libraries provide a function named `hypot(x, y)`, which computes the hypotenuse of two numbers x and y (a.k.a. Pythagorean addition).

While the hypotenuse h is simply defined as

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

real-world `hypot()` implementations will normally be more complex than a direct transcription of the above formula. The snippet below gives the Python code of a typical implementation that is mathematically equivalent:

```
def hypot(x, y):
    x = abs(x)
    y = abs(y)
    a = min(x, y)
    b = max(x, y)
    r = a / b
    return b * np.sqrt(1 + r * r)
```

Explain what serious problem in the original formula this more involved implementation addresses. How does it ensure that the problem cannot arise anymore? You can assume that the variables x and y are similar in magnitude.

Solution: Very small or very large values can easily over or under-flow in the squaring operation of the original formula, giving bogus results. The `hypot()` implementation shown below first computes the ratio `t` $\in [0,1]$, which can never overflow even if squared.
Underflow (`r*r == 0`) can still happen, which happens when one argument is much larger than the other. In this case `np.sqrt(1 + r*r) == 1` and the function returns the larger value.

Fall 2016 exam solution

2 Large-scale computations (25 pts)

2.1 Vectorization (15 pts)

While analyzing a slow-running program, you stumble across the following inefficient loops that iterate over the individual entries of large vectors and/or matrices. Replace them with equivalent (and more compact) vectorized Python code.

In the code fragments given below, `x` denotes a 1-dimensional NumPy array (i.e. a vector) and `A` is a 2-dimensional $n \times n$ NumPy array (i.e. a square matrix).

(5 pts each)

(i)

```
z = np.zeros(x.shape[0])
for i in range(x.shape[0]):
   z[i] = x[i] * x[i]
```

Solution:

```
z = x*x # or x**2
```

(ii)

```
z = 0
for i in range(A.shape[0]):
    for j in range(A.shape[1]):
        z += A[i, j] * x[i] * x[j]
```

Solution:

```
z = x @ (A @ x) # Many other possibilities
```

(iii)

```
z = np.zeros(A.shape[1])
for j in range(A.shape[1]):
    temp = 0
    for i in range(A.shape[0]):
        temp += A[i, j] * A[i, j]
    z[j] = np.sqrt(temp)
```

Solution:

```
z = np.sqrt(np.sum(A*A, axis=0)) # Many other possibilities
```

Fall 2016 exam solution

2.2 Performance implications of subtle program changes (10 pts)

While refactoring a program that performs arithmetic involving a large 1000×1000 matrix A and a 1000-dimensional vector x, you discover that version 2 of the two functionally equivalent algorithms below runs consistently faster. List all reasons you can think of that would explain this difference, and relate them to specific parts of the code.

```
# Version 1 (slow)
for j in range(A.shape[1]):
    for i in range(A.shape[0]):
        if x[i] > 0:
            A[i, j] += 1.0
```

```
# Version 2 (fast)
for i in range(A.shape[0]):
    for j in range(A.shape[1]):
        if x[i] > 0:
            A[i, j] += 1.0
```

Solution: Two concepts which the answer should contain:

a) A performance decrease could indicate that version 1 of the loop is not accessing memory sequentially, thereby foregoing the benefits of the cache hierarchy of modern processors. In this case, simply swapping the loops will lead to a noticeable speedup.

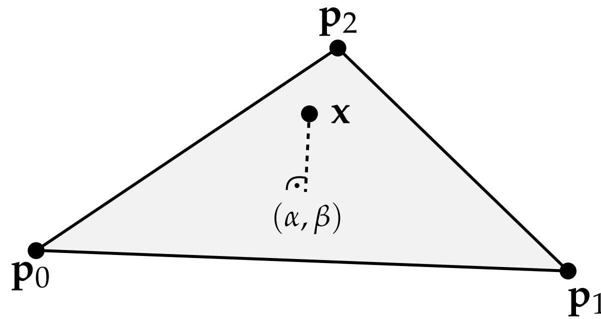
b) Version 1 involves branching based on numbers that change at a high frequency (every iteration). This reduces the effectiveness of branch prediction in modern processors, leading to penalties in execution time.

(the fact that more potentially non-cached loads from **x** are necessary in the first version is also a small factor, but that is not an answer we are looking for here)

Fall 2016 exam solution

3 Linear systems (25 pts)

3.1 Least squares in 3D (14 pts)



Let the positions $\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3 \in \mathbb{R}^3$ denote the corners of a triangle in three dimensions. Any point on the triangle can be expressed as a linear combination of the corner positions using local coordinates (α, β):

$$\mathbf{p}(\alpha, \beta) := \mathbf{p}_0 + \alpha(\mathbf{p}_1 - \mathbf{p}_0) + \beta(\mathbf{p}_2 - \mathbf{p}_0)$$

Given an arbitrary point $\mathbf{x} \in \mathbb{R}^3$ that is close to the triangle, it's often useful be able to find the values of α and β, whose associated position is *closest* to $\mathbf{x}$ according to the $\|\cdot\|_2$-norm.

(i) Re-formulate this computation as a least squares problem and express it in its standard form (i.e. $\mathbf{Ax} \approx \mathbf{b}$). (7 pts)

Solution:

$$\begin{pmatrix} | & | \\ \mathbf{p}_1 - \mathbf{p}_0 & \mathbf{p}_2 - \mathbf{p}_0 \\ | & | \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} | \\ \mathbf{x} - \mathbf{p}_0 \\ | \end{pmatrix}$$

Fall 2016 exam solution

(ii) Derive the associated normal equations. (7 pts)

Solution: Normal equations solution:

$$\begin{pmatrix} \|\mathbf{p}_1 - \mathbf{p}_0\|^2 & \langle \mathbf{p}_1 - \mathbf{p}_0, \mathbf{p}_2 - \mathbf{p}_0 \rangle \\ \langle \mathbf{p}_2 - \mathbf{p}_0, \mathbf{p}_1 - \mathbf{p}_0 \rangle & \|\mathbf{p}_2 - \mathbf{p}_0\|^2 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \langle \mathbf{p}_1 - \mathbf{p}_0, \mathbf{x} - \mathbf{p}_0 \rangle \\ \langle \mathbf{p}_2 - \mathbf{p}_0, \mathbf{x} - \mathbf{p}_0 \rangle \end{pmatrix}$$

3.2 Condition numbers and the Singular Value Decomposition (5 pts)

Suppose we are given a matrix $\mathbf{A}$ which satisfies $\operatorname{cond}(\mathbf{A}) = 1$. What kind of a matrix is $\mathbf{A}$? Justify your answer. It may be helpful to think about $\mathbf{A}$'s singular value decomposition.

Solution: Let $\mathbf{U\Sigma V}^T = \mathbf{A}$ be the SVD of $\mathbf{A}$. $\operatorname{cond} \mathbf{A} = 1$ implies that $\sigma_1 = \sigma_n =: \sigma \in \mathbb{R}$. This means that $\mathbf{A} = \sigma \mathbf{UV}^T$ (involving a matrix product of two orthogonal matrices) is simply a scaled orthogonal matrix (i.e. a matrix with orthogonal rows and columns, but whose rows and columns are not necessarily unit length).

Fall 2016 exam solution

3.3 Solving least squares problems using the QR decomposition (6 pts)

Suppose that a large number of least squares problems of the form

$$\begin{aligned} \mathbf{A}\mathbf{x}_1 &\approx \mathbf{b}_1 \\ \mathbf{A}\mathbf{x}_2 &\approx \mathbf{b}_2 \\ \mathbf{A}\mathbf{x}_3 &\approx \mathbf{b}_3 \\ \vdots\ &\quad \vdots \end{aligned}$$

involving the same tall matrix $\mathbf{A}$ must be solved for different right-hand sides $\mathbf{b}_1, \ldots \mathbf{b}_n$. Complete the partial Python implementation below so that it that solves this problem while making efficient use of the QR decomposition.

```
def qrsolve(A, b_list):
    # Solutions will be collected in this list
    x_list = []









    # For every right-hand side
    for b in b_list:
        # Solve a least squares problem to find 'x' given 'b'









        # Append result to the list of solutions
        x_list.append(x)
    # Return all solutions
    return x_list
```

Fall 2016 exam solution

Solution:

```
def qrsolve(A, b_list):
    # Solutions will be collected in this list
    x_list = []
    # Precompute factorization
    Q, R = la.qr(A, mode='economic')
    # For every right-hand side
    for b in b_list:
        # Solve a least squares problem to find 'x' given 'b'
        temp = Q.T @ b
        x = la.solve_triangular(R, temp, lower = False)
        # Append result to the list of solutions
        x_list.append(x)
    # Return all solutions
    return x_list
```

Fall 2016 exam solution

4 Nonlinear methods (25 pts)

4.1 Comparison of root finding methods (6 pts)

Please fill in all entries of the following table. You may assume that the function in question is smooth, and that a starting point or a bracketing interval containing a root is given.

Method	Function evaluations per iteration[a]	Convergence speed[b]	Derivative needed? Yes/No	Always converges? Yes/No
Secant				
Newton				
Bisection				

[a] Assuming an optimized implementation. Include evaluations of the derivative.
[b] Write 1 for the fastest convergence rate, 3 for the slowest, and 2 for the one in between.

Solution:

Method	Evals	Convergence	Derivative	Always converges
Secant	1	2	N	N
Newton	2	1	Y	N
Bisection	1	3	N	Y

4.2 Reverse engineering a nonlinear algorithm (19 pts)

You are reading some code from another programmer and you find numerical calculations without documentation and cryptic variable names. You eventually isolate the following iterative calculation that seems to be the most important part:

```
def compute_it(a):
    x = 1
    for i in range(10):
        x = (a + 2 * x**3) / (3 * x**2)
    return x
```

(Note that `a**b` refers to the power operation a^b)

(i) What algorithm is being used? (2 pts)

Solution: Newton's method.

Fall 2016 exam solution

(ii) What equation is being solved? Explain the code in terms of the standard equation for the algorithm. (12 pts)

Solution: $f(x) = x^3 - a$ is being solved. The iteration formula for Newton's method is $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{x_n^3 - a}{3x_n^2} = \frac{-x_n^3 + a + 3x_n^3}{3x_n^2} = \frac{a + 2x_n^3}{3x_n^2}$.

(iii) What value does the iteration converge to? If there is more than one possible answer, any of them is fine (5 pts)

Solution: It converges to $\sqrt[3]{a}$.

Fall 2016 exam solution