

ETH Lectures 401-0663-00L Numerical Methods for Computer Science
401-2673-00L Numerical Methods for CSE

Homework Problems and Projects

Prof. R. Hiptmair, SAM, ETH Zurich

Autumn Term 2021, Version of October 2, 2022
(C) Seminar für Angewandte Mathematik, ETH Zürich

[Link](#) to the current version of this homework collection

0.1 General Information

0.1.1 Weekly Homework Assignments

All problems will be published in this single “.pdf” file. Every week, we publish a list of problem numbers on the lecture Moodle page. These are the current assignments for that week. Details are given on the lecture Moodle page.

0.1.2 Importance of Homework

Homework assignments are **not** mandatory. However, it is **very important** that you constantly exercise with the material you learn. “Solving” the homework assignments one week before the main exam by looking at the solutions will likely result in failure.

We provide hints and solutions from the beginning. It is your responsibility to look at the solutions only if you are stuck with a difficult problem and you tried to find a solution for a sufficient amount of time.

Most homework problems come with an estimate for the time required for its solution. These times are what we expect a bright and well-prepared student to require for that problem during an exam.

0.1.3 Corrections and Grading of Assignments

Homework assignments will not be graded.

However, you may submit your handwritten solutions by handing them to your tutor directly or uploading them via the Moodle upload interface, see the lecture Moodle page for instructions. The assistants will

Problem 4-2: FFT and least squares

This problem deals with an application of fast Fourier transformation in the context of a least squares (data fitting) problems.

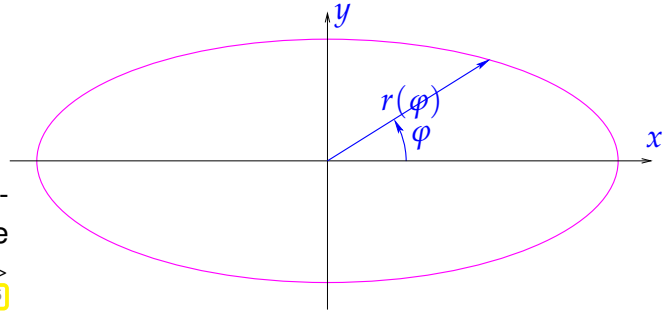
You should be familiar with both DFT and the idea of least-squares fitting

The orbit of a planet around the sun is a closed, planar curve $\mathcal{C} \subset \mathbb{R}^2$, which can be described by a so-called polar representation, closely related to polar coordinates:

$$\mathcal{C} := \left\{ \begin{bmatrix} d(\varphi) \cos \varphi \\ d(\varphi) \sin \varphi \end{bmatrix} : 0 \leq \varphi < 2\pi \right\},$$

where $d : [0, 2\pi] \rightarrow \mathbb{R}^+$ is a given function of the polar angle φ providing the distance of a point on the curve from the origin in the direction of that angle. ▷

Fig. 25



We put the sun in the origin of the coordinate system. For equispaced polar angles ($\rightarrow$ Fig. 25) $\varphi_j = 2\pi \frac{j}{n}$, $j = 0, \dots, n-1$, $n \in \mathbb{N}$ the distance of the planet from the sun is measured and found to be $d_j \in \mathbb{R}$, $j = 0, \dots, n-1$. We now discuss a numerical method providing the planet's path in polar representation.

Write $\mathcal{P}_m^{\mathbb{C}}$, $m \in \mathbb{N}$, for the space of real **trigonometric polynomials** of degree $\leq m$. These are 2π -periodic functions $\mathbb{R} \rightarrow \mathbb{R}$ of the form

$$p(t) = \sum_{k=0}^m c_k \cos(kt), \quad c_k \in \mathbb{R}. \quad (4.2.1)$$

The idea is to approximate the distance $d(\varphi)$ of the planet from the sun as a function of the angle by a trigonometric polynomial $p^* \in \mathcal{P}_m^{\mathbb{C}}$, which minimized the sum of squares of "distance mismatches":

$$p^* = \operatorname{argmin}_{p \in \mathcal{P}_m^{\mathbb{C}}} \sum_{j=0}^{n-1} |p(\varphi_j) - d_j|^2. \quad (4.2.2)$$

Throughout, we assume that the number of distance data is more than twice as big as the degree of p^* : $2m < n$.

(4-2.a) □ (20 min.) Recast this task as a standard linear least squares problem

$$\mathbf{x}^* = \operatorname{argmin}_{\mathbf{x} \in \mathbb{R}^?} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2, \quad (4.2.3)$$

with a suitable matrix $\mathbf{A} \in \mathbb{R}^{?,?}$ and vectors $\mathbf{b}, \mathbf{x} \in \mathbb{R}^?$.

SOLUTION of (4-2.a):

Of course, the coefficients of the trigonometric polynomial p are the unknown parameters to be estimates. Those will be collected in the vector $\mathbf{x}$:

$$\mathbf{x} := [c_0, \dots, c_m] \in \mathbb{R}^{m+1}.$$

We have

$$p(\varphi_j) = p\left(2\pi \frac{j}{n}\right) = \sum_{k=0}^m c_k \cos\left(2\pi k \frac{j}{n}\right) = \sum_{k=0}^m c_k \cos\left(\frac{2\pi}{n} kj\right), \quad j = 0, \dots, n-1.$$

Hence,

$$\sum_{j=0}^{n-1} |p(\varphi_j) - d_j|^2 = \left\| \begin{bmatrix} \sum_{k=0}^m c_k \cos\left(\frac{2\pi}{n}k \cdot 0\right) - d_0 \\ \sum_{k=0}^m c_k \cos\left(\frac{2\pi}{n}k \cdot 1\right) - d_1 \\ \vdots \\ \sum_{k=0}^m c_k \cos\left(\frac{2\pi}{n}k(n-1)\right) - d_{n-1} \end{bmatrix} \right\|_2^2 \stackrel{!}{=} \|\mathbf{Ax} - \mathbf{b}\|_2^2.$$

Recalling the definition of the Euclidean norm and the rules of matrix $\times$ vector multiplication, we find

$$\mathbf{A} = \begin{bmatrix} \cos(0) & \cos(0) & \cos(0) & \dots & \cos(0) \\ \cos(0) & \cos\left(\frac{2\pi}{n}\right) & \cos\left(\frac{4\pi}{n}\right) & \dots & \cos\left(\frac{2\pi m}{n}\right) \\ \cos(0) & \cos\left(\frac{4\pi}{n}\right) & \cos\left(\frac{8\pi}{n}\right) & \dots & \cos\left(\frac{4\pi m}{n}\right) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \cos(0) & \cos\left(\frac{2\pi(n-1)}{n}\right) & \cos\left(\frac{4\pi(n-1)}{n}\right) & \dots & \cos\left(\frac{2\pi(n-1)m}{n}\right) \end{bmatrix} \in \mathbb{R}^{n,m+1}, \mathbf{b} = \begin{bmatrix} d_0 \\ d_1 \\ \vdots \\ d_{n-1} \end{bmatrix} \in \mathbb{R}^n.$$

More compactly, letting $\cos$ act on a matrix componentwise, we can write

$$\mathbf{A} = \left[\cos\left(\frac{2\pi}{n}\right) \ell k \right]_{\substack{\ell=0,\dots,n-1 \\ k=0,\dots,m}} = \cos\left(\frac{2\pi}{n} \begin{bmatrix} 0 \\ 1 \\ \vdots \\ n-1 \end{bmatrix} [0 \ 1 \ \dots \ m] \right) \in \mathbb{R}^{n,m+1}. \quad (4.2.4)$$

▲

(4-2.b) 🕒 (30 min.) [depends on Sub-problem (4-2.a)]

State the **normal equations** [Lecture $\rightarrow$ Thm. 3.1.2.1] for the linear least-squares problem found in Sub-problem (4-2.a) in explicit form.

HINT 1 for (4-2.b): Use Euler's formula

$$\cos(2\pi x) = \frac{1}{2}(e^{2\pi i x} + e^{-2\pi i x}),$$

together with the geometric sum formula [Lecture $\rightarrow$ Eq. (4.2.1.9)]

$$\sum_{\ell=0}^{n-1} q^\ell = \frac{1-q^n}{1-q}, \quad (4.2.5)$$

applied with $q := \exp(2\pi i \frac{v}{n})$, $n \in \mathbb{Z}$, which yields, cf. [Lecture $\rightarrow$ § 4.2.1.6],

$$\sum_{\ell=0}^{n-1} \exp\left(\frac{2\pi i}{n} \ell v\right) = \begin{cases} n & \text{, if } v \in n\mathbb{Z}, \\ 0 & \text{else.} \end{cases}$$

┘

SOLUTION of (4-2.b):

As we have learned in [Lecture $\rightarrow$ Section 3.1.2], [Lecture $\rightarrow$ Thm. 3.1.2.1], the normal equations for a linear least-squares problem $\|\mathbf{Ax} - \mathbf{b}\| \rightarrow \min$ are

$$\mathbf{A}^\top \mathbf{Ax} = \mathbf{A}^\top \mathbf{b}. \quad (4.2.6)$$

The entries of $\mathbf{A}^\top \mathbf{A} \in \mathbb{R}^{m+1, m+1}$ are inner products of columns of $\mathbf{A}$ as given in (4.2.4): for $j, k \in \{0, \dots, m\}$,

$$\begin{aligned} (\mathbf{A}^\top \mathbf{A})_{j,k} &= \sum_{\ell=0}^{n-1} \cos\left(\frac{2\pi j\ell}{n}\right) \cos\left(\frac{2\pi k\ell}{n}\right) \\ &= \frac{1}{4} \sum_{\ell=0}^{n-1} \left(e^{\frac{2\pi j\ell i}{n}} + e^{-\frac{2\pi j\ell i}{n}} \right) \left(e^{\frac{2\pi k\ell i}{n}} + e^{-\frac{2\pi k\ell i}{n}} \right) \\ &= \frac{1}{4} \sum_{\ell=0}^{n-1} \left(e^{\frac{2\pi(j+k)\ell i}{n}} + e^{\frac{2\pi(j-k)\ell i}{n}} + e^{\frac{2\pi(k-j)\ell i}{n}} + e^{-\frac{2\pi(j+k)\ell i}{n}} \right) = \begin{cases} n, & j = k = 0, \\ n/2, & j = k \neq 0, \\ 0, & j \neq k. \end{cases} \end{aligned}$$

Note that the assumption $2m < n$ rules out $j + k = n$. As a consequence we find that $\mathbf{A}^\top \mathbf{A}$ is *diagonal*!

$$\mathbf{A}^\top \mathbf{A} = \begin{bmatrix} n & & & \\ & n/2 & & \\ & & \ddots & \\ & & & n/2 \end{bmatrix} \in \mathbb{R}^{m+1, m+1}. \quad (4.2.7)$$



(4-2.c) 🕒 (20 min.) [depends on Sub-problem (4-2.a), Sub-problem (4-2.b)]

Write a short EIGEN-based C++ function

bool testNormEqMatrix(unsigned int n, unsigned int m);

that test, whether the formula for the matrix product $\mathbf{A}^\top \mathbf{A}$ that you have found in Sub-problem (4-2.b) is correct.

To that end, form the dense matrix $\mathbf{A}$ as an object of type **Eigen::MatrixXd**, compute $\mathbf{A}^\top \mathbf{A}$ and compare with the matrix given by your formula. Of course, the comparison has to be done in a numerically sound way following the advice of [Lecture → Rem. 1.5.3.15].

SOLUTION of (4-2.c):

The formula (4.2.4) can be implemented directly in EIGEN, which supplies componentwise application of *cos* to the entries of a matrix 🐼 **EIGEN documentation**. The test of relative closeness can rely on the Frobeniusnorm provided by EIGEN's `norm()` method, but any other matrix norm can be used as well. As tolerance we may set $mn \cdot \text{TOL}$ with $\text{TOL} \ll 1$, for instance $\text{TOL} \approx \text{EPS}$ or slightly larger.

C++11-code 4.2.8: Implementation of `testNormEqMatrix()`.

```
2 bool testNormEqMatrix(unsigned int n, unsigned int m) {
3     // TODO: (4-2.c) Test if the two definitions, given in (4.2.4) and
4     // (4.2.7), of  $A^T * A$  are equal.
5
6     // START
7     Eigen::MatrixXd A(n, m + 1);
8
9     // Initializing the vectors described in (4.2.4)
10    Eigen::VectorXd n_linear = Eigen::VectorXd::LinSpaced(n, 0, n - 1);
```

```

11 Eigen::VectorXd m_linear = Eigen::VectorXd::LinSpaced(m + 1, 0, m);
12
13 // Building A as described in (4.2.4)
14 A = 2.0 * M_PI / n * n_linear * m_linear.transpose();
15 A = A.array().cos().matrix();
16
17 // Using the knowledge gained in (4-2.b) to initialize A^T * A
18 // directly as a
19 // diagonal matrix.
20 Eigen::MatrixXcd ATA = Eigen::MatrixXcd::Zero(m + 1, m + 1);
21 ATA.diagonal() = Eigen::VectorXd::Constant(m + 1, n / 2.0);
22 ATA(0, 0) = n;
23
24 // Comparing the two approaches to calculate A^T * A.
25 // Test for relative smallness of the Frobenius norm of the
26 // difference of the
27 // matrices
28 if ((ATA - A.transpose() * A).norm() < 1e-10 * m * n) {
29     return true;
30 }
31 // END
32
33 return false;
34 }

```

Get it on  GitLab (fftlsq.hpp).



(4-2.d) (45 min.) [depends on Sub-problem (4-2.b)]

Write a C++ function

```
VectorXd find_c(const VectorXd & d, unsigned int m);
```

that computes the coefficients c_k , $k = 0, \dots, m$, of p^* in the form (4.2.1) for arbitrary inputs $d_j \in \mathbb{R}$, $j = 0, \dots, n-1$. These are passed as vector $\mathbf{d}$.

We aim for an efficient implementation: the function must not have asymptotic complexity worse than $O(n \log n)$ in terms of the problem size parameters $m, n \rightarrow \infty$.

HINT 1 for (4-2.d): Note the identity

$$\sum_{k=0}^{n-1} e^{\frac{2\pi i}{n} jk} d_k = \overline{\sum_{k=0}^{n-1} e^{-\frac{2\pi i}{n} jk} d_k}, \quad j = 0, \dots, n-1, \quad (4.2.9)$$

where the over-line denotes the complex conjugation. 

SOLUTION of (4-2.d):

Of course, we solve the normal equations $\mathbf{A}^\top \mathbf{A} \mathbf{x} = \mathbf{a}^\top \mathbf{b}$ to obtain the coefficients of p^* . From (4.2.7) we already know that $\mathbf{A}^\top \mathbf{A}$ is diagonal. So there is no linear system to be solved, just a scaling of the right-hand side vector to perform. Yet, we have to compute that right-hand side vector $\mathbf{A}^\top \mathbf{b}$. If we did it using straightforward matrix $\times$ vector multiplication, we would incur a computational cost of $O(mn)$ for $m, n \rightarrow \infty$.

To understand how $\mathbf{A}^\top \mathbf{b}$ can be formed more efficiently, we compute the j -th entry of the right hand-side vector of the normal equations: $\mathbf{A}^\top \mathbf{b}$ is:

$$\begin{aligned} (\mathbf{A}^\top \mathbf{b})_j &= \sum_{k=0}^{n-1} \cos\left(\frac{2\pi jk}{n}\right) d_k = \frac{1}{2} \sum_{k=0}^{n-1} \left(e^{-\frac{2\pi i}{n} jk} + e^{\frac{2\pi i}{n} jk} \right) d_k = \frac{1}{2} \sum_{k=0}^{n-1} e^{-\frac{2\pi i}{n} jk} d_k + \frac{1}{2} \sum_{k=0}^{n-1} e^{\frac{2\pi i}{n} jk} d_k \\ &= \frac{1}{2} \left((\mathbf{F}_n \mathbf{b})_j + \overline{(\mathbf{F}_n \mathbf{b})_j} \right) = (\operatorname{Re}(\mathbf{F}_n \mathbf{b}))_j, \end{aligned}$$

where $\mathbf{F}_n$ stands for the **Fourier matrix**, see [Lecture → Eq. (4.2.1.13)]. This tells us that $\mathbf{A}^\top \mathbf{b} \in \mathbb{R}^n$ is the real parts of a section of the discrete Fourier transform [Lecture → Section 4.2] of the data vector $\mathbf{b} = [d_0, \dots, d_{n-1}]^\top$. As we have learned in [Lecture → Section 4.3] that DFT is available with an effort of $O(n \log n)$ for $n \rightarrow \infty$.

Hence, if $m > O(\log n)$, this is a more efficient way to compute $\mathbf{A}^\top \mathbf{b}$ than direct multiplication.

C++11-code 4.2.10: Implementation of `find_c`.

```

2  Eigen::VectorXd find_c(const Eigen::VectorXd &d, unsigned int m) {
3      unsigned int n = d.size();
4
5      // We will use a real to complex, discrete Fourier transform.
6      Eigen::FFT<double> fft;
7      Eigen::VectorXd rhs;
8
9      // TODO: (4-2.d) find the coefficients
10     // START
11
12     // Computing DFT of d
13     Eigen::VectorXcd fourier = fft.fwd(d);
14
15     // Gathering what is important for the rhs.
16     rhs = fourier.real().head(m + 1);
17
18     // Solving normal equation by inverting diagonal matrix.
19     rhs /= n;
20     rhs.tail(m) *= 2.0;
21     // END
22
23     return rhs;
24 }

```

Get it on  [GitLab \(fftlsq.hpp\)](https://gitlab.com/fftlsq).



(4-2.e) (30 min.) [depends on Sub-problem (4-2.d)]

Since Kepler we know that the orbits of planets around the sun are ellipses. Choose $n = 100$ and sample the distance data from the polar form of an ellipse:

$$d(\varphi) = \frac{1}{\sqrt{1 - c^2 \cos^2 \varphi}}, \quad \varphi \in [0, 2\pi], \quad c = 0.8. \quad (4.2.11)$$

1. Using your implementation of `find_c` compute and tabulate the coefficients c_k of p^* for $m = 1, 2, 3$.
2. Using `MATPLOTLIBCPP`'s `plot()` function create a plot of the ellipse given by (4.2.11).

3. Into the same plot insert the curves described by the trigonometric polynomials p^* of degrees $m = 1, 2, 3$.

This should be accomplished by a C++ function

```
void fitEllipse(void);
```

HINT 1 for (4-2.e): Actually `MATPLOTLIBCPP`'s `plot()` function can merely draw polygons. Therefore you have to give it many points on the ellipse and the approximating curves: use, for instance,

$$\begin{bmatrix} d(\frac{2\pi}{N}j) \cos(\frac{2\pi}{N}j) \\ d(\frac{2\pi}{N}j) \sin(\frac{2\pi}{N}j) \end{bmatrix}, \quad j = 0, \dots, N-1, \quad N = 10^4.$$

SOLUTION of (4-2.e):

C++11-code 4.2.12: Implementation of `fitEllipse()`.

```

2  void fitEllipse() {
3      unsigned int n = 10;
4      unsigned int m = 3;
5
6      // Test points
7      const unsigned int npoints = 10;
8      Eigen::VectorXd d(npoints);
9      d << 0.987214, 1.03579, 0.997689, 0.917471, 1.00474, 0.92209, 1.03517,
10         1.08863, 0.904992, 0.956089;
11
12     plt::figure();
13
14     // TODO: (4-2.e) Tabulate the coefficients of find_c for all m=1,2,3
15     // plot the ellipse and also the curves of the trigonometric
16     // polynomials
17     // START
18     const double c = 0.8;
19     auto d_func =
20         [c](const double phi) { // no need to capture c because it is
21             constexpr
22             assert(0. <= phi && phi <= 2. * M_PI);
23             return 1. / std::sqrt(1. - std::pow(c * std::cos(phi), 2));
24         };
25
26     // Sample distance data, we reuse d from above
27     n = 100;
28     d.resize(n);
29     for (std::size_t i = 0; i < n; ++i) {
30         d(i) = d_func((2. * M_PI * i) / n);
31     }
32
33     plt::title("Orbit of planet");
34     plt::xlim(-2, 2);
35     plt::ylim(-2, 2);
36
37     // Plot the ellipse given by (5.2.10) using 10000 points
38     constexpr unsigned int N = 10000;
39     Eigen::VectorXd x_ellipse(N), y_ellipse(N);
40     double temp_d;
41     for (std::size_t i = 0; i < N; ++i) {
42         temp_d = d_func((2.0 * M_PI * i) / N);
43     }

```

```

41 x_ellipse(i) = temp_d * std::cos((2.0 * M_PI * i) / N);
42 y_ellipse(i) = temp_d * std::sin((2.0 * M_PI * i) / N);
43 }
44 plt::plot(x_ellipse, y_ellipse, "k", {"label", "exact orbit"});
45
46 // Calculate, tabulate and plot the coefficients in one go
47 x_ellipse.resize(n);
48 y_ellipse.resize(n);
49
50 // Create vector of colors for plots
51 std::vector<std::string> colors = {"m", "y--", "r"};
52 std::cout << "degree of polynomial | coefficients" << std::endl;
53 for (m = 1; m <= 3; ++m) {
54     Eigen::VectorXd coefficients = find_c(d, m);
55
56     std::cout << std::setw(20) << m << " | " << std::setw(5)
57         << coefficients.transpose() << std::endl;
58
59     Eigen::VectorXd e = eval_p(coefficients, n);
60     for (std::size_t i = 0; i < n; ++i) {
61         x_ellipse(i) = e(i) * std::cos((2 * M_PI * i) / n);
62         y_ellipse(i) = e(i) * std::sin((2 * M_PI * i) / n);
63     }
64     const std::string label = "fitted trig. poly. for m = " + std::to_string(m);
65     plt::plot(x_ellipse, y_ellipse, colors[m - 1], {"label", label});
66 }
67
68 plt::xlabel("x");
69 plt::ylabel("y");
70 plt::legend();
71
72 // END
73 plt::savefig("cx_out/orbit.png");
74 }

```

Get it on  GitLab (fftlsq.hpp).

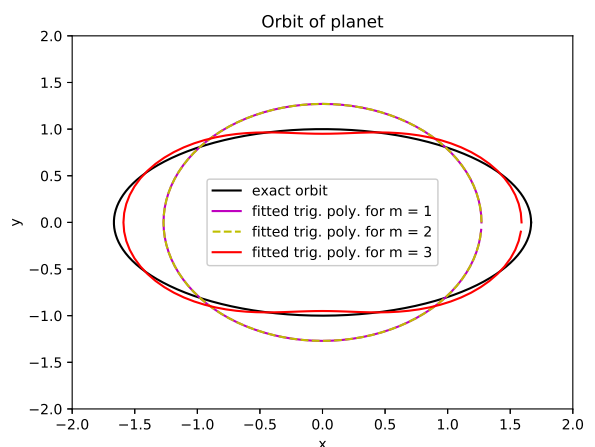
Picture produced by the function 

These are the coefficients of the fitted trigonometric polynomials:

m	$[c_0, \dots, c_{m-1}]^T$
1	$[1.27025]^T$
2	$[1.27025 \ 3.60268 \cdot 10^{-17}]^T$
3	$[1.27025 \ 3.60268 \cdot 10^{-17} \ 0.320124]^T$

We see, as expected, a better fit for larger degree m . The fact $c_1 = 0$ is due to the symmetry of the ellipse.

Fig. 26



End Problem 4-2, 145 min.