

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

Chapter 4

Filtering Algorithms

Problem 4-1: Autofocus with FFT

In this problem, we will use 2D frequency analysis to find the “best focused” image among a collection of out-of-focus photos. To that end, we will implement an algorithm based on 2D DFT as introduced in [Lecture → Section 4.2.5].

This problem takes for granted that you know about the 2D discrete Fourier transform and its connection with discrete convolutions, see [Lecture → Section 4.2.2] and [Lecture → Section 4.2.5].

Let a grey-scale image consisting of $n \times m$ pixels be given as a matrix $\mathbf{P} \in \mathbb{R}^{n,m}$ as in [Lecture → Ex. 4.2.5.16]; each element of the matrix indicates the gray-value of the pixel as a number between 0 and v_{max} . This image is regarded as the “perfect image”.

If a camera is poorly focused due to an inadequate arrangement of the lenses, it will record a blurred image $\mathbf{B} \in \mathbb{R}^{n,m}$. As before [Lecture → Eq. (4.2.5.17)], the blurring operation can be modeled through the 2D (periodic) discrete convolution and can be undone provided that the point spread function (PSF) is known. However, in this problem we must not assume any knowledge of the PSF.

Assume that the only data available to us is the “black-box” C++ function (declared in `autofocus.hpp`):

```
MatrixXd set_focus(double f);
```

which returns the potentially blurred image $\mathbf{B}(f)$, when the focus parameter f is set to a particular value. The actual operation of `set_focus` is utterly obscure. The focus parameter can be read as the distance of the lenses of a camera that can be changed through turning on and off a stepper motor. The following sequence of images illustrates the impact of setting different focus parameters:



Fig. 16



Fig. 17



Fig. 18



Fig. 19

The problem of *autofocusing* is to determine the value of the focus parameter f , which yields an image $\mathbf{B}(f)$ with the least blur. The idea of the autofocus algorithm is the following: *The less the image is marred by blur, the larger its “high frequency content”*. The latter can be found out based on the discrete Fourier transform, more precisely, its 2D version.

To translate the autofocus idea into an algorithm we have to give a quantitative meaning to the notion of “high frequency content” of a (blurred) image $\mathbf{C}$, which is done by looking at the second moments of its 2D discrete Fourier transform:

$$V(\mathbf{C}) = \sum_{k_1=0}^{n-1} \sum_{k_2=0}^{m-1} \left(\left(\frac{n}{2} - \left| k_1 - \frac{n}{2} \right| \right)^2 + \left(\frac{m}{2} - \left| k_2 - \frac{m}{2} \right| \right)^2 \right) |\hat{\mathbf{C}}_{k_1, k_2}|^2.$$

Here, $\hat{\mathbf{C}} \in \mathbb{C}^{n,m}$ stand for the 2D DFT of the image $\mathbf{C}$. Hence, the autofocus policy can be rephrased as

find $f \in \mathbb{R}$ such that $V(\mathbf{B}(f))$ becomes maximal.

(4-1.a) ☐ (10 min.) This sub-problem supplies us with a tool to write an image file from a C++ code. Write a C++ function

```
void save_image(double focus);
```

that saves the image $\mathbf{B}(f)$ returned by `set_focus()` for $f = 0, 1, 2, 3$ in **Portable Graymap (PGM)** format to the file `./cx_out/image_focus<f>.png`, where `<f>` is a string containing the argument `focus`.

For this task you can use objects of type `PGMObject` (found in “`pgm.hpp`”). You can find example of usages of this class in “`examples/pgm_example.cpp`”.

SOLUTION of (4-1.a):

C++11-code 4.1.1: Rendering of images with different focus settings.

```
2 void save_image(double focus) {
3     // Create empty object
4     PGMObject q;
5
6     // TO DO: (4-1.a) Read matrix of image generated
7     // by "set_focus" and save as an image in format ".pgm".
8     // START
9     // Set data using function "set_data"
10    // Data obtained from "set_focus"
```

```

11     q.set_data(set_focus(focus));
12
13     // Create and save file
14     std::stringstream ss;
15     ss << "./cx_out/image_focus="
16         << (int) focus
17         << ".pgm";
18     std::ofstream file(ss.str());
19     file << q;
20     // END
21 }

```

Get it on  [GitLab \(autofocus.hpp\)](#).



The file `fft.hpp` supplies a few FFT-based convenience functions. Among them is `fft2r`, which performs the 2D discrete Fourier transform [Lecture → Eq. (4.2.5.4)] of a real-values matrix, cf. [Lecture → Code 4.2.5.6], which results in a complex matrix however.

C++ code 4.1.2: 2D DFT of a `Eigen::MatrixXd`

```

2  Eigen::MatrixXcd fftf(const Eigen::MatrixXd &X) {
3      const long m = X.rows(), n = X.cols();
4      Eigen::MatrixXcd Y(m, n); // result
5      Eigen::FFT<double> fft; // auxiliary FFT object
6      for (long j = 0; j < n; ++j) {
7          Eigen::VectorXd Xj = X.col(j);
8          Y.col(j) = fft.fwd(Xj);
9      }
10     return Y;
11 }

```

```

2  Eigen::MatrixXcd fft2r(const Eigen::MatrixXd &X) {
3      return fft(fftf(X).transpose()).transpose();
4  }

```

(4-1.b)  (20 min.) The two-dimensional discrete Fourier transform can be performed using the C++ function `fft2r()`. Write a C++ function

```
void plot_freq(double focus);
```

that creates 3D plots of the (modulus of the) 2D DFTs of the images obtained in sub-problem (4-1.a). Clamp the data between 0 and 8000, this means when a value exceeds 8000, then set it to 8000.

HINT 1 for (4-1.b): `MATPLOTLIBCPP` provides many facilities for rendering 2D data, for instance `plot_surface()`, see [Github page](#). 

SOLUTION of (4-1.b):

C++11-code 4.1.3: Plotting of spectrum of the images.

```

2  void plot_freq(double focus) {
3      int a = 0;
4      int b = 8000;

```

```

5 | auto clamp = [a, b](double x) { return x < a ? a : x > b ? b : x; };
6 |
7 | // TO DO: (4-1.b) compute D containing the spectrum of
8 | // set_focus(focus).
9 | // "clamp" the data between 0 and 8000.
10 | // START
11 | MatrixXd D = fft2r(set_focus(focus)).cwiseAbs().unaryExpr(clamp) / b;
12 | // END
13 |
14 | // Plot values of X.
15 | // Axis labels
16 | std::vector<double> xticks(5);
17 | for (int i = 0; i < 5; ++i) {
18 |     xticks[i] = i * (D.cols() - 1) / 4;
19 | }
20 | std::vector<double> yticks(5);
21 | for (int i = 0; i < 5; ++i) {
22 |     yticks[i] = i * (D.rows() - 1) / 4;
23 | }
24 | std::vector<std::string> labels{"-1", "-0.5", "0", "0.5", "1"};
25 |
26 | plt::figure();
27 | plt::imshow(D, {"cmap", "viridis"}, {"origin", "lower"});
28 | plt::colorbar();
29 | plt::xticks(xticks, labels);
30 | plt::yticks(yticks, labels);
31 | std::stringstream ss;
32 | ss << "Spectrum with f = " << focus << ".";
33 | plt::title(ss.str().c_str());
34 | std::stringstream ss2;
35 | ss2 << "./cx_out/spectrum_focus=" << focus << ".eps";
36 | plt::savefig(ss2.str().c_str());

```

Get it on  GitLab (autofocus.hpp).

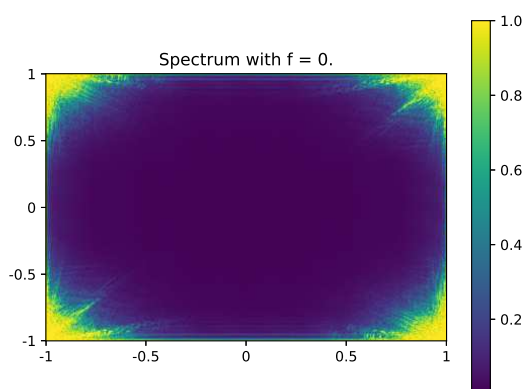


Fig. 20

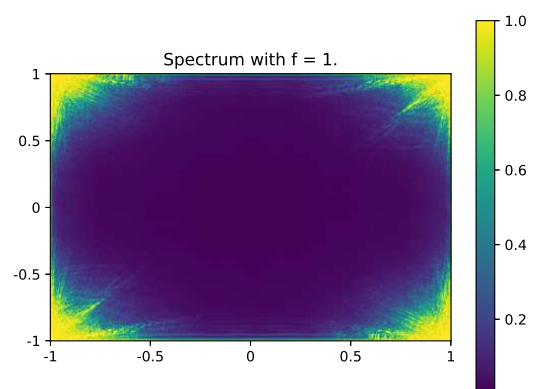


Fig. 21

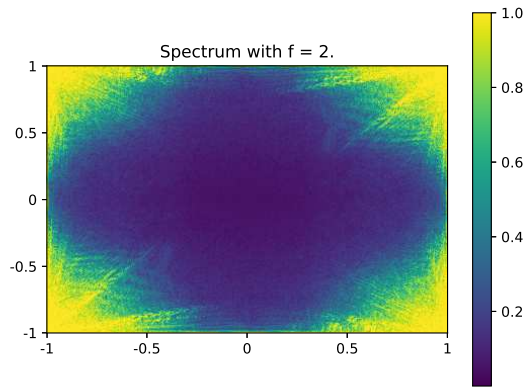


Fig. 22

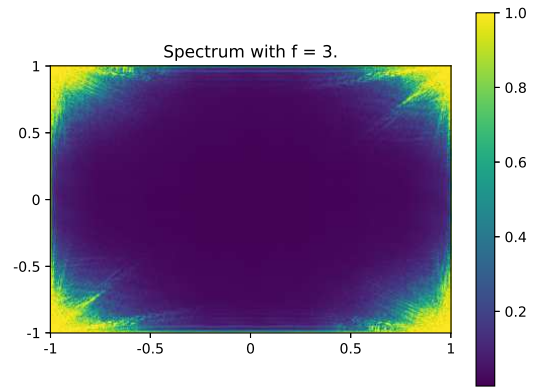


Fig. 23

(4-1.c) (15 min.) In the plots generated in Sub-problem (4-1.b), mark (or explain) the regions corresponding to the high and low frequencies.

HINT 1 for (4-1.c): Look at [Lecture → § 4.2.3.6] and [Lecture → Fig. 126] to understand the relationship of Fourier coefficients and low and high frequency content of a signal.

SOLUTION of (4-1.c):

Centres of the plots correspond to the high frequencies, whereas corners correspond to the low frequencies.

From the plots we see that all the images are rich in “low-frequency content”, because the dominant 2D frequencies are concentrated in the corners of the matrix arising from 2D DFT. However, one can also observe that for some values of the autofocus parameter f also “high-frequency modes” in the center of the matrix become more important.

(4-1.d) (25 min.) Write a C++ function

```
double high_frequency_content(const MatrixXd & C);
```

that returns the value $V(\mathbf{C})$ for a given matrix $\mathbf{C}$. Write a C++ function

```
void plotV();
```

that plots the function $V(\mathbf{B}(f))$ in terms of the focus parameter f . Use 100 equidistantly spaced sampling points in the interval $[0, 5]$.

SOLUTION of (4-1.d):

C++11-code 4.1.4: Computation of high-frequency content.

```
2 double high_frequency_content(const MatrixXd& M) {
3     int n = M.rows(), m = M.cols();
4     double V = 0;
5     // TO DO: (4-1.d) compute V(M).
```

```

6 // START
7 for (unsigned int i = 0; i < M.rows(); ++i) {
8     for (unsigned int j = 0; j < M.cols(); ++j) {
9         double a = n / 2. - std::abs(i - n / 2.);
10        double b = m / 2. - std::abs(j - m / 2.);
11        V += (a * a + b * b) * M(i, j) * M(i, j);
12    }
13 }
14 // END
15 return V;
16 }

```

Get it on  GitLab (autofocus.hpp).

C++11-code 4.1.5: Plot creation for $V(\mathbf{B}(f))$.

```

2 void plotV() {
3     unsigned int N = 20;
4
5     VectorXd x(N), y(N);
6
7     // TO DO: (4-1.d) Plot  $V(\mathbf{B}(f))$ .
8     // START
9     for (unsigned int i = 0; i < N; ++i) {
10        double V = high_frequency_content(
11            // Find 2D spectrum of matrix  $\mathbf{B}(t)$ 
12            fft2r(
13                // Evaluate set_focus at equidistant points
14                set_focus(5. / (N - 1) * i))
15            .cwiseAbs());
16        x(i) = 5. / (N - 1) * i;
17        y(i) = V;
18    }
19
20    // END
21    plt::figure();
22    plt::title("High frequency content.");
23    plt::plot(x, y, "r+", {"label", "V(\mathbf{B}(f))"});
24    plt::xlabel("f");
25    plt::ylabel("V(\mathbf{B}(f))");
26    plt::savefig("./cx_out/focus_plot.eps");
27 }

```

Get it on  GitLab (autofocus.hpp).

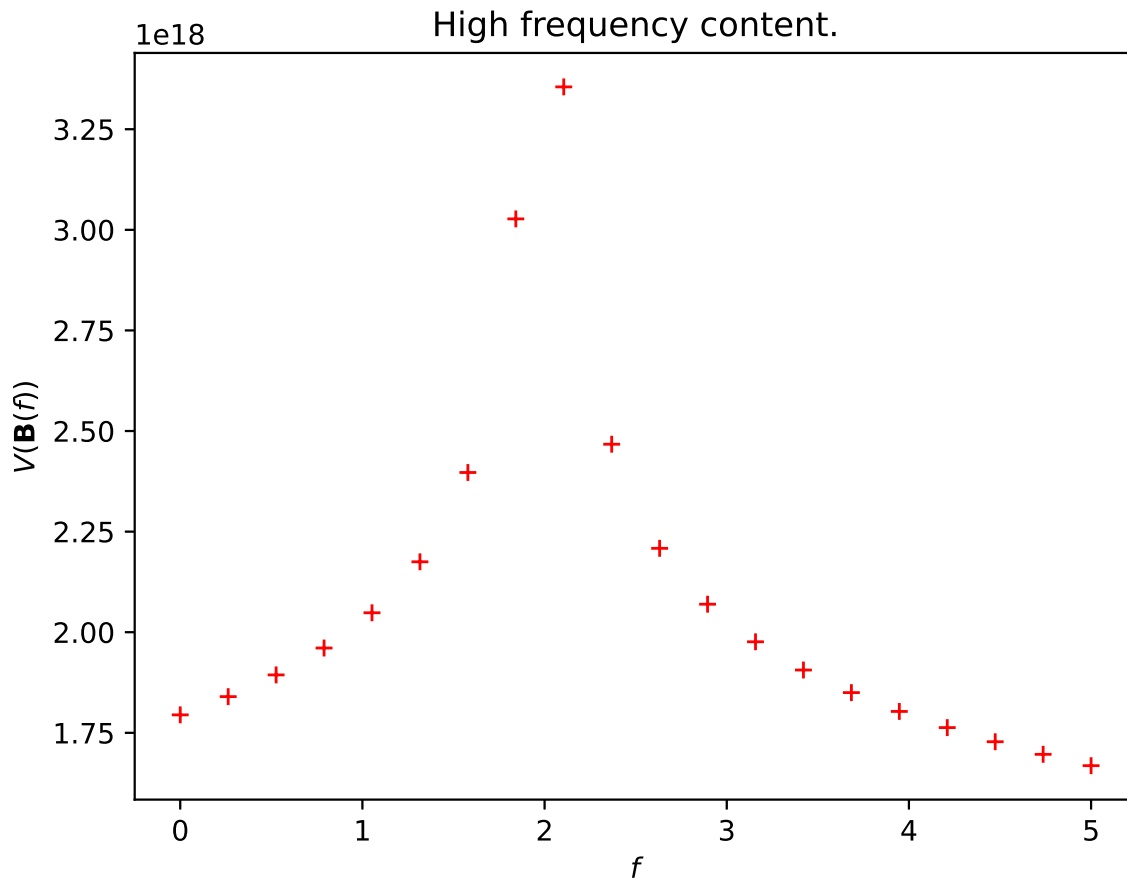


Fig. 24

There is a pronounced peak of $f \rightarrow V(\mathbf{C}(f))$ for a particular value of f ! For that value the blurred picture has maximal high-frequency content.



(4-1.e) (40 min.) Write an *efficient* (you want the auto-focus procedure to take not longer than a few seconds!) C++ function

```
double autofocus();
```

that numerically determines optimal focus parameter $f_0 \in [0, 5]$ for which the 2nd moment $V(\mathbf{B}(f))$ is maximized. What is the resulting focus parameter f_0 ?

To locate the change of slope of $V(\mathbf{B}(f))$ in $[0, 5]$ you should use the **bisection algorithm** [Lecture → Section 8.4.1] for finding a zero of the derivative $f \mapsto \frac{dV(\mathbf{B}(f))}{df}$, which will, hopefully, mark the location of the maximum of $V(f)$.

The bisection algorithm for finding a zero of a continuous function $\varphi : [a, b] \rightarrow \mathbb{R}$ with $\varphi(a) \cdot \varphi(b) < 0$ is rather simple and lucidly demonstrated in [Lecture → Code 8.4.1.2].

Of course, the derivative is not available, so that we have to approximate it crudely by means of a difference quotient

$$\frac{dV(f)}{df} \approx \frac{V(\mathbf{B}(f + \delta f)) - V(\mathbf{B}(f - \delta f))}{2\delta f}.$$

You can assume that the smallest step of the auto-focus mechanism is 0.05 and so the natural choice

for δf is $\delta f = 0.05$. This maximal resolution also tells you a priori how many bisection steps should be performed.

Use the structure of the function $V(\mathbf{B}(f))$ that you observed in sub-problem (4-1.d). Use as few evaluations of $V(\mathbf{B}(f))$ as possible!

SOLUTION of (4-1.e):

The auto-focus determines that focus parameter should be set to $f \approx 2.0$.

C++11-code 4.1.6: Finding best value of focus parameter.

```

2  double autofocus() {
3      // Minimum focus
4      const double min_focus = 0;
5      // Maximum focus
6      const double max_focus = 5;
7      // Min step
8      const double min_step = 0.05;
9      // Starting guess
10     double f0 = (max_focus - min_focus) / 2.;
11     // Finite differences increment
12     double df = min_step;
13     // Starting step
14     double step = max_focus / 2.;
15     // Max number of iteration
16     unsigned int Niter = std::ceil(std::log2((max_focus - min_focus) / min_step));
17     // TO DO: (4-1.e) Use bisection method to find best focus.
18     // START
19     // Returns V(B(f))
20     auto computeV = [](double focus) {
21         return high_frequency_content(fft2r(set_focus(focus)).cwiseAbs());
22     };
23
24     // Bisection method
25     for (unsigned int i = 0; i < Niter; ++i) {
26         double dV = computeV(f0 + df) - computeV(f0 - df);
27
28         step = step / 2.;
29         f0 = f0 + (dV > 0 ? 1 : -1) * step;
30     }
31     // END
32     return f0;
33 }

```

Get it on  GitLab (autofocus.hpp).



End Problem 4-1 , 110 min.