

SOLUTION of (4-3.e):

By the **convolution theorem** [Lecture $\rightarrow$ Thm. 4.2.2.2], (4.3.7) implies

$$\text{DFT}_{N+1}\mathbf{p} = \text{DFT}_{N+1}(\hat{\mathbf{u}}) \odot \text{DFT}_{N+1}(\hat{\mathbf{v}}),$$

where $\odot$ stands for componentwise multiplication of vectors. Hence, formally we get

$$\hat{\mathbf{v}} = \text{DFT}_{N+1}^{-1}(\text{DFT}_{N+1}(\mathbf{p}) \oslash \text{DFT}_{N+1}(\hat{\mathbf{u}})), \quad (4.3.8)$$

with componentwise division written as $\oslash$.

However, two complications arise and those have to be taken care of in a code.

- (i) If $(\text{DFT}_{N+1}\hat{\mathbf{u}})_j = 0$ for some $j = 0, \dots, N$ (C++ indexing), the componentwise division will not be possible.
- (ii) The result vector $\hat{\mathbf{v}}$ may not be a valid coefficient vector for a polynomial $v \in \mathcal{P}_n$ (notations as in the first hint), because $(\hat{\mathbf{v}})_j = 0$ is not guaranteed for $j > n$.

To deal with (i), we have to check, whether $(\text{DFT}_{N+1}\hat{\mathbf{u}})_j = 0$ for some $j = 0, \dots, N$. If also $(\text{DFT}_{N+1}\hat{\mathbf{p}})_j = 0$ then set the result of the division to zero, otherwise u does not divide p . Complication (ii) can be detected by a simple test: If any $(\hat{\mathbf{v}})_j, j > n$, does not vanish we conclude that u does not divide p .

C++ code 4.3.9: Solution of (4-3.e)

```
2 Eigen::VectorXd polyDiv(const Eigen::VectorXd &uv, const Eigen::VectorXd &u) {
3     // Initialization
4     const unsigned int mn = uv.size();
5     const unsigned int m = u.size();
6     // need well behaved input
7     if (mn < m) {
8         std::cerr << "uv can't be divided by u\n";
9         return Eigen::VectorXd(0);
10    }
11    const unsigned int dim = mn;
12    const unsigned int n = mn - m + 1;
13
14    Eigen::VectorXd v;
15
16    // TODO: (4-3.e) Divide polynomials uv and u efficiently.
17    // START
18
19    // zero padding
20    Eigen::VectorXd uv_tmp = uv;
21    uv_tmp.conservativeResizeLike(Eigen::VectorXd::Zero(dim));
22    Eigen::VectorXd u_tmp = u;
23    u_tmp.conservativeResizeLike(Eigen::VectorXd::Zero(dim));
24
25    // Specify that real DFTs should only return/expect the conjugate
26    // symmetric
27    // half of the complex input/output
28    Eigen::FFT<Eigen::FFTOption::Scaled | Eigen::FFTOption::HalfSpectrum> fft;
29
30    // Complex outputs only have size floor(dim/2) + 1
31    Eigen::VectorXcd uv_tmp_ = fft.fwd(uv_tmp);
32    Eigen::VectorXcd u_tmp_ = fft.fwd(u_tmp);
```

```

32 // check divisibility: case(i)
33 // - using Eigen::FFTOption::HalfSpectrum halves the necessary
34 // iterations below since each iteration effectively
35 // checks both position i as well as (dim-i-1)
36 for (unsigned int i = 0; i < dim / 2 + 1; ++i) {
37     if (abs(uv_tmp_(i)) < 1e-13) {
38         if (abs(u_tmp_(i)) < 1e-13) {
39             // make cwiseQuotient at i-th position equal 0
40             uv_tmp_(i) = 0.0; // complex assignment (0., 0.)
41             u_tmp_(i) = 1.0; // complex assignment (1., 0.)
42         } else {
43             std::cerr << "uv can't be divided by u\n";
44             return Eigen::VectorXd(0);
45         }
46     }
47 }
48 }
49
50 // The coefficient-wise product of conjugate symmetric
51 // vectors is itself conjugate symmetric...
52 Eigen::VectorXcd tmp = uv_tmp_.cwiseQuotient(u_tmp_);
53 // ... so we can use a more efficient Complex-To-Real Inverse FFT
54 v = fft.inv(tmp, dim); // since v is real, a C2R is triggered and
55                        // we need to specify the full output dimension
56
57 // check divisibility: case(ii)
58 for (unsigned int i = n; i < dim; ++i) {
59     if (abs(v(i)) > 1e-13) {
60         std::cerr << "uv can't be divided by u\n";
61         return Eigen::VectorXd(0);
62     }
63 }
64
65 // reshape v to a suitable size
66 v.conservativeResizeLike(Eigen::VectorXd::Zero(n));
67 // (mn-1) - (m-1) + 1
68 // END
69
70 return v;
71 }

```

Get it on 🐙 [GitLab](#) (polydiv.hpp).