ETH Lecture 401-2673-00L Numerical Methods for **CSE**

Final Examination

Autumn Term 2021

Aug 31, 2022, 9:30, HG G 1

Don't panic!

		Grade
Family Name		
First Name		
Department		
Legi Nr.		
Date	Aug 31, 2022	

Points:

Prb. No.	1	2	3	Total
max	26	27	25	78
achvd				

(100% = 65 pts. , ≈40% (passed) = 26 pts.)

- **Duration: 180 minutes + 45 minutes advance reading**
- Cell phones and other communication devices are not allowed. Make sure that they are turned off and stowed away in your bag.
- No own notes or other aids are permitted during the exam!
- You may cite theorems, lemmas, and equations from the lecture document by specifying their precise number in the supplied PDF.
- *Write clearly with a non-erasable pen. Do not use red pen or green pen.* No more than one solution can be handed in per problem. Invalid attempts should be clearly crossed out.
- Write your name on every problem sheet in the blank line at the top of the pages.
- Write clean solutions of theoretical tasks in the boxes with the green frame.

  Only the contents of solution boxes will be taken into account when grading the exam. Thus, **bring glue and scissors** in case you want to replace one of your solutions. No scrap paper must be handed in along with the exam paper.
- If a tasks is to be answered in a free-text box, include relevant preparatory considerations, auxiliary computations, etc. in your solution.
- Get a general idea of the problems. Pay attention to the number of points awarded for each subtask. It is roughly correlated with the amount of work the task will require.
- **If you have failed to solve a sub-problem, do not give up on the entire problem**, but try the next one. Dependencies between sub-problems are indicated where relevant.
- For coding tasks, for which no solution boxes are provided, only the code files will be examined for grading.

- Codes will be graded like theoretical problems: Partial solutions and correct approaches and ideas will be rewarded as long as they are evident from the code. Hence, it is advisable to write tidy and well-structured codes and add some comments.
- In coding tasks, even if you could not fully implement a function, you may still call it in a following coding task.
- Wrong ticks in multiple-choice-type problems may lead to points being deducted.

At the beginning of the exam

Follow the steps listed below. This is not counted as exam time.

You are not allowed to start the exam until you have a clear indication from an assistant or the Professor.

1. Upon entering the examination room, starting from the far end of the room sit down at any available desk.
2. Put your ETH ID card ("legi") on the table.
3. Fill in the blanks on the cover page of the problem sheet. Do not turn pages yet!
4. On your desk you will find scrap paper.
5. The computer screen should show an exam selection page. You can change the language of the page (upper right). In case you have requested a US keyboard, you can change the keyboard layout (bottom right corner).
6. Please provide your information (last name, first name, legi-nr.), select the correct exam from the drop-down menu, and continue.
7. Log into Moodle as a member of ETH Zürich using your NETHZ username and password.
8. You should now see the start page of the Moodle exam. You can not start the exam, but you can open the available resources (C++ Reference, Eigen documentation, Lecture document). Navigate between windows using alt+tab. You can also arrange the windows side by side.
9. Wait until everybody is finished with the previous steps.
10. Only when you are asked so, turn the pages of the problem sheets. Then, you may read through all the problems and also click through the provided documents. However, *during the advance reading time you must not touch the keyboard or a pen*!
11. You are allowed to remove the staples, but you must not forget to *write your name on every page*.
12. After the 45 minutes of advance reading, the exam password will be communicated and you can begin the exam.

Instructions concerning CodeExpert

- When you enter the password and start the Moodle exam, a timer of 180 minutes is started. Once the timer expires, your attempt is submitted and you can no longer make changes to your answers. Your answers are automatically saved at regular intervals.
- You will see the Code Expert environment (CE) embedded in a small window. It may take a moment to load. This is the coding part of the first problem.
- In the top right corner of the CE, you see a blue icon that allows you to view the CE in full screen mode.

- At the bottom of the CE, you can click 'Console' to get access to the 'Run' and 'Test' buttons. At the right side, you can click 'Task' to view a short task description. At the left side, you can click 'Project file system' to see the available files for this problem.
- Under 'Task' on the right hand side, there is a 'message' icon. If it shows a notification symbol, please check the announcement sent by the examiners.
- To navigate to the next problem, first minimize the CE (click the blue icon in the top right corner), and then click 'Next page'. You can return to a previous problem by pressing 'Previous page'.
- At the final problem of the exam, you will see a 'Finish attempt...'. If you click this button, you will see a summary of your attempt. Then you can decide to submit your solution, or to return to your attempt.
- Concerning implementation in C++ you will only be asked to complete classes or (member) functions in existing header files. The sections of the code you have to supply will be marked by

  ```
  // TO DO: ...
  // START

  // END
  ```

  Insert your code between these two markers. Some variables may already have been defined earlier in the template code.
- Note that codes rejected by the compiler will incur a point penalty.
- *C++-code* that has been commented out will not be considered as part of your solution. *Text* comments may be taken into account, however.
- You are free to edit the file 'main.cpp', but its contents will not be considered as part of your solution.
- Each problem has test cases in 'tests.cpp'. These tests do not determine your grade.

At the end of the examination

1. **Do not log out and do not turn off the computer!**
2. Make sure that you have written you name on the top of all pages of the exam paper.
3. Put all your written solution in the exam envelopes and leave those on the desk along with the computer form.
4. Wait until your row will be called to leave the room.

Problems

- If you have problems with the computer, please raise your hand to get support, right in the beginning of the exam, if possible.
- In case your computer fails irrevocably ("freezes") you will be assigned another one.

Throughout the exam use the notations introduced in class, in particular [Lecture → **??**]:

- $(\mathbf{A})_{i,j}$ to refer to the entry of the matrix $\mathbf{A} \in \mathbb{K}^{m,n}$ at position (i, j).
- $(\mathbf{A})_{:,i}$ to designate the i-column of the matrix $\mathbf{A}$,
- $(\mathbf{A})_{i,:}$ to denote the i-th row of the matrix $\mathbf{A}$,
- $(\mathbf{A})_{i:j,k:\ell}$ to single out the sub-matrix $\left[(\mathbf{A})_{r,s}\right]_{\substack{i\leq r\leq j\\ k\leq s\leq \ell}}$ of the matrix $\mathbf{A}$,

- $\mathbf{A}^\top$ for the transposed matrix,
- $\otimes$ to denote the Kronecker product,
- $\cdot$ as an alternative way to write the Euclidean inner product of two vectors,
- $(\mathbf{x})_k$ to reference the k-th entry of the vector $\mathbf{x}$,
- $\mathbf{e}_j \in \mathbb{R}^n$ to write the j-th Cartesian coordinate vector,
- $\mathbf{I}$ ($\mathbf{I}_n$) to denote the identity matrix (of size $n \times n$)
- $\mathbf{O}$ to write a zero matrix,
- $\mathcal{P}_n$ for the space of (univariate polynomials of degree $\leq n$),
- and superscript indices in brackets to denote iterates: $\mathbf{x}^{(k)}$, etc.

By default, vectors are regarded as column vectors.

Problem 0-1: Compact Storage Format for QR-Factorization

It is essential for the efficiency of orthogonalization techniques that orthogonal matrices, which commonly occur as factors in matrix products. are stored as products of elementary orthogonal transformations like Householder reflections or Givens rotations, see [Lecture → **??**]. This approach is adopted in all numerical libraries including EIGEN. This problem addresses some related algorithmic issues.

This problem assumes familiarity with orthogonal matrices [Lecture → **??**] and Householder reflections [Lecture → **??**].

A legacy FORTRAN-77 numerical library routine returns a raw array (type **double** $*$) of n^2 **double** floating-point numbers, $n \in \mathbb{N}$, which represents an $n \times n$-matrix $\mathbf{A} \in \mathbb{R}^{n,n}$ according to the following specification:

(I) The array encodes an $n \times n$ matrix $\mathbf{M} \in \mathbb{R}^{n,n}$ in column-major format [Lecture → **??**].

(II) The column vectors $\mathbf{u}_j \in \mathbb{R}^n$, $j = 1, \dots, n-1$, defined as[1]

$$(\mathbf{u}_j)_k := \begin{cases} 0 & \text{for } k < j\,, \\ \sqrt{1 - \sum\limits_{\ell=k+1}^{n} (\mathbf{M})_{\ell,j}^2} & \text{for } k = j\,, \\ (\mathbf{M})_{k,j} & \text{for } k > j\,, \end{cases} \qquad j = 1, \dots, n-1\,, \tag{0.1.1}$$

are the building blocks for the product matrix

$$\mathbf{Q} = (\mathbf{I}_n - 2\mathbf{u}_1\mathbf{u}_1^\top)(\mathbf{I}_n - 2\mathbf{u}_2\mathbf{u}_2^\top) \cdot \dots \cdot (\mathbf{I}_n - 2\mathbf{u}_{n-1}\mathbf{u}_{n-1}^\top)\,. \tag{0.1.2}$$

(III) The matrix $\mathbf{A}$ is given as $\mathbf{A} = \mathbf{QR}$, where $\mathbf{R} \in \mathbb{R}^{n,n}$ is the *upper triangular* part of $\mathbf{M}$:

$$(\mathbf{R})_{k,j} := \begin{cases} (\mathbf{M})_{k,j} & \text{for } k \le j\,, \\ 0 & \text{else}, \end{cases} \qquad k, j \in \{1, \dots, n\}\,. \tag{0.1.3}$$

Your task is to write an EIGEN-based wrapper class for the raw data that offers certain operations with the matrix $\mathbf{A}$. The class definition is a follows:

C++ code 0.1.4: Class definition of CompactStorageQR

```cpp
class CompactStorageQR {
 public:
  // Constructor taking raw data array, which has to be persistent
  CompactStorageQR(const double *data, unsigned int n) : n_(n), M_(data, n, n) {
    for (unsigned int k = 0; k < n_ - 1; ++k) {
      double sn{0};
      for (unsigned int j = k + 1; j < n_; ++j) {
        const double t{M_(j, k)};
        sn += t * t;
      }
      if (sn > 1.0) {
        throw std::runtime_error(
            "CompactStorageQR: Illegal subdiagonal column norm!");
      }
    }
```

[1] A sum whose lower summation bound is larger than the upper summation bound is understood to evaluate to zero.

```cpp
  }
  // Determinant of the matrix stored in this object
  double det() const;
  // Right multiplication with another matrix
  Eigen::MatrixXd matmult(const Eigen::MatrixXd &X) const;
  // Solution of linear systems of equations
  Eigen::MatrixXd solve(const Eigen::MatrixXd &B) const;

 private:
  unsigned int n_;  // Matrix dimensions
  // Raw data wrapped into a matrix, see [Lecture → ??]
  Eigen::Map<const Eigen::MatrixXd> M_;
};
```

(0-1.a) ⚁ (10 pts.) In the file `compactstorageqr.hpp` implement the method

```cpp
Eigen::MatrixXd CompactStorageQR::matmult(
   const Eigen::MatrixXd &X) const;
```

which returns the matrix product $\mathbf{A}\cdot\mathbf{X}$ of the matrix $\mathbf{A}\in\mathbb{R}^{n,n}$ stored in the **CompactStorageQR** object and of a matrix $\mathbf{X}\in\mathbb{R}^{n,k}$, $k\in\mathbb{N}$, passed in `X`. Your implementation must not incur asymptotic computational cost worse than $O(n^2k)$ for $n,k\to\infty$.

SOLUTION of (0-1.a):

By the associativity of matrix multiplication

$$\mathbf{A}\mathbf{X}=\mathbf{Q}(\mathbf{R}\mathbf{X})=\mathbf{Q}_1(\mathbf{Q}_2(\ldots\mathbf{Q}_{n-1}(\mathbf{R}\mathbf{X})\ldots))\;,\quad \mathbf{Q}_j:=\mathbf{I}_n-2\mathbf{u}_j\mathbf{u}_j^\top\;.$$

As discussed in [Lecture → **??**], for the sake of efficiency the multiplication of a matrix $\mathbf{Y}\in\mathbb{R}^{n,k}$ with $\mathbf{Q}_j$ has to be implemented as follows

$$\mathbf{Q}_j\mathbf{Y}=\mathbf{Y}-2\mathbf{u}_j\left(\mathbf{u}_j^\top\mathbf{Y}\right)\;,$$

which requires an asymptotic computational effort of $O(nk)$ for $n,k\to\infty$. These matrix multiplications have to be embedded into an outer loop over $j=n-1,n-2,\ldots,1$.

C++ code 0.1.5: Method `matmult()` of CompactStorageQR

```cpp
Eigen::MatrixXd CompactStorageQR::matmult(const Eigen::MatrixXd &X) const {
  assert((X.rows() == n_) && "Wrong size of matrix factor");
  Eigen::MatrixXd Y(n_, X.rows());
  // START Student code
  // Multiplication with R-factor
  Y = M_.template triangularView<Eigen::Upper>() * X;
  // Multiplication with Q-factors
  Eigen::VectorXd u{Eigen::VectorXd::Zero(n_)};
  for (int k = n_ - 2; k >= 0; --k) {
    u.tail(n_ - k - 1) = M_.block(k + 1, k, n_ - k - 1, 1);
    u[k] = std::sqrt(1.0 - u.squaredNorm());
    Y -= 2 * u * (u.transpose() * Y);
  }
  // END Student code
  return Y;
}
```

An even more efficient implementation is possible by taking into account that the first $j-1$ entries of the vector $\mathbf{u}_j$ are zero.

▲

(0-1.b) ⚂ (10 pts.) In the file `compactstorageqr.hpp` supply the missing parts of the implementation of

```
Eigen::MatrixXd CompactStorageQR::solve(const Eigen::MatrixXd &B)
    const;
```

which is to compute $\mathbf{A}^{-1}\mathbf{B}$ for $\mathbf{A} \in \mathbb{R}^{n,n}$ stored in the current **CompactStorageQR** object and $\mathbf{B} \in \mathbb{R}^{n,k}$ contained in the argument `B`. Again, the asymptotic complexity of your implementation must be $O(n^2k)$ for $n, k \to \infty$.

HINT 1 for (0-1.b): You must exploit that the factor matrices in (0.1.2) are Householder matrices [Lecture → **??**], in particular, *orthogonal*. ⌟

SOLUTION of (0-1.b):

Based on the factorization of $\mathbf{A}$,

$$\mathbf{A} = \mathbf{Q}\mathbf{R} = \mathbf{Q}_1\mathbf{Q}_2 \cdot \cdots \cdot \mathbf{Q}_{n-1}\mathbf{R}\,, \quad \mathbf{Q}_j := \mathbf{I}_n - 2\mathbf{u}_j\mathbf{u}_j^\top\,,$$

we have

$$\mathbf{A}^{-1}\mathbf{B} = \mathbf{R}^{-1}\mathbf{Q}_{n-1}^{-1}\mathbf{Q}_{n-2}^{-1} \cdot \cdots \cdot \mathbf{Q}_1^{-1}\mathbf{B}\,.$$

Note that, since the vectors $\mathbf{u}_j$ from (0.1.1) are normalized, $\left\|\mathbf{u}_j\right\|_2 = 1$, the matrices $\mathbf{Q}_j$ are *orthogonal* and symmetric **Householder matrices** [Lecture → **??**] satisfying

$$\mathbf{Q}_j^{-1} = \mathbf{Q}_j^\top = \mathbf{Q}_j\,, \quad j = 1, \dots, n-1\,. \tag{0.1.6}$$

We infer

$$\mathbf{A}^{-1}\mathbf{B} = \mathbf{R}^{-1}\mathbf{Q}_{n-1}\mathbf{Q}_{n-2} \cdot \cdots \cdot \mathbf{Q}_1\mathbf{B}\,,$$

which suggests an implementation very similar to `CompactStorageQR::matmult()` in Code 0.1.5. Of course $\mathbf{R}^{-1}\mathbf{Y}$ for $\mathbf{Y} \in \mathbb{R}^{n,k}$ is obtained by solving the k linear systems of equations $\mathbf{R}\mathbf{x} = (\mathbf{Y})_\ell$, $\ell = 1, \dots, k$, which can be done with an asymptotic computational effort of $O(n^2)$ for $n \to \infty$, because $\mathbf{R}$ is *upper triangular*.

C++ code 0.1.7: Method `solve()` of class CompactStorageQR

```cpp
Eigen::MatrixXd CompactStorageQR::solve(const Eigen::MatrixXd &B) const {
  assert((B.rows() == n_) && "Wrong size of right-hand side");
  Eigen::MatrixXd X{B};
  // START student code
  // Check invertibility by inspecting diagonal elements of R-factor
  const double atol = 1.0E-16;
  const double rtol = 1.0E-8;
  double maxrii = std::abs(M_(0, 0));
  for (unsigned int k = 1; k < n_; ++k) {
    const double tmp = std::abs(M_(k, k));
    if (maxrii < tmp) {
      maxrii = tmp;
    }
  }
  for (unsigned int k = 1; k < n_; ++k) {
    const double rii = std::abs(M_(k, k));
    if ((rii < rtol * maxrii) || (rii < atol)) {
      throw std::runtime_error("CompactStorageQR::solve: matrix not regular");
    }
  }
  // Multiplication with Q-factors
  Eigen::VectorXd u(n_);
  for (unsigned int k = 0; k < n_ - 1; ++k) {
    double sn{0};
    // Retrieve u_j
    for (unsigned int j = k + 1; j < n_; ++j) {
      u[j] = M_(j, k);
      sn += u[j] * u[j];
    }
    u[k] = std::sqrt(1.0 - sn);
    // Multiplication with current Q-factor
    X -= 2 * u * (u.transpose() * X);
    u[k] = 0.0;
  }
  // Solve upper triangular linear system RX = Q^T B
  X = M_.template triangularView<Eigen::Upper>().solve(X);
  // END student code
  return X;
}
```

This code also checks for the invertibility of the matrix $\mathbf{A}$, which was not requested in this sub-problem.

Inspecting the loop we see that the total asymptotic computational cost of `solve()` for $\mathbf{B} \in \mathbb{R}^{n,k}$ is $O(n^2k)$, which is optimal.

▲

(0-1.c) (6 pts.) In the file `compactstorageqr.hpp` write the code for the method

```
double CompactStorageQR::det() const;
```

which gives the **determinant** $\det \mathbf{A}$ of the matrix $\mathbf{A} \in \mathbb{R}^{n,n}$ stored in the current **CompactStorageQR** object.

HINT 1 for (0-1.c): From linear algebra remember that $\det(\mathbf{AX}) = \det \mathbf{A} \cdot \det \mathbf{X}$ for all $\mathbf{AX} \in \mathbb{R}^{n,n}$. ⌟

SOLUTION of (0-1.c):

Thanks for multiplication theorem for determinants the factorization

$$\mathbf{A} = \mathbf{QR} = \mathbf{Q}_1\mathbf{Q}_2 \cdot \cdots \cdot \mathbf{Q}_{n-1}\mathbf{R}\,, \quad \mathbf{Q}_j := \mathbf{I}_n - 2\mathbf{u}_j\mathbf{u}_j^\top\,,$$

immediately implies

$$\det \mathbf{A} = \prod_{j=1}^{n-1} \det \mathbf{Q}_j \cdot \det \mathbf{R}\,.$$

Note that

- the matrices $\mathbf{Q}_j$ are *orthogonal* and symmetric **Householder matrices** [Lecture → **??**] satisfying $\det \mathbf{Q}_j = -1$,
- the determinant of the *upper triangular* matrix $\mathbf{R}$ is equal to the product of its diagonal elements.

Remark. To see that $\det \mathbf{Q}_j = -1$, let $\mathbf{U}_j \in \mathbb{R}^{n,n}$ an *orthogonal* matrix with first column $\mathbf{u}_j$:

$$\blacktriangleright \quad \mathbf{U}_j\mathbf{Q}_j\mathbf{U}_j = \mathbf{U}_j^\top(\mathbf{I}_n - 2\mathbf{u}_j\mathbf{u}_j^\top)\mathbf{U}_j = \operatorname{diag}(-1,1,\dots,1)\,.$$

C++ code 0.1.8: Method det of class CompactStorageQR

```cpp
double CompactStorageQR::det() const {
  double d;
  // START Student code
  // The determinant is just the product of the diagonal entries of the
  // R-factor, because the determinant of the Q-factor is ±1.
  d = 1.0;
  for (unsigned int j = 0; j < n_; ++j) {
    d *= M_(j, j);
  }
  // Sign correction, because Househiolder transformations
  // have determinant -1 and n-1 of them are inviolved.
  d *= ((n_ % 2 == 0) ? -1.0 : 1.0);
  // END Student code
  return d;
}
```

▲

End Problem 0-1 , 26 pts.

Problem 0-2: Periodic Collocation for a Non-Linear Differential Equation

This problem examines the so-called collocation method for the approximate solution of differential equations. Since we look for periodic solutions, it is natural to use linear combinations of trigonometric polynomials as trial space. Equations are obtained by requiring that the equations remains satisfied in finitely many points.

This problem requires the techniques from [Lecture → **??**] and [Lecture → **??**] and involves substantial implementation in C++ using EIGEN.

We look for 1-periodic solutions $u : [0,1] \to \mathbb{C}$, $u = u(t)$, of the differential equation

$$-\frac{d^2u}{dt^2}(t) + u(t)^3 = \sin(\pi t)\ , \quad 0 \le t < 1\ .$$

To approximate them we replace u with

$$u_N(t) := \sum_{j=0}^{N} x_j \cos(2\pi j t) \quad \text{for some} \quad N \in \mathbb{N}\ , \tag{0.2.1}$$

and try to determine the unknown coefficients $x_j \in \mathbb{R}$, $j = 0, \dots, N$, by solving the **collocation equations**

$$-\frac{d^2u_N}{dt^2}(t_k) + u_N(t_k)^3 = \sin(\pi t_k)\ , \quad k = 0, \dots, N\ , \quad t_k := \frac{k}{N+1}\ . \tag{0.2.2}$$

(0-2.a) ⚁ (7 pts.) For plotting $t \mapsto u_N(t)$ with u_N as in (0.2.1) we need an *efficient* C++ function

```
Eigen::VectorXd eval_uN(const Eigen::VectorXd &x, unsigned int M);
```

which, given the coefficients $x_j \in \mathbb{R}$, $j = 0, \dots, N$, in (0.2.1) through the vector `x` and a number $M > N$ in `M`, returns the vector

$$\left[u_N\left(\frac{k}{M}\right)\right]_{k=0}^{M-1} \in \mathbb{R}^M\ .$$

"Efficient" stipulates that the computational cost of `eval_uN()` must scale like $O(M \log M)$ for $M \to \infty$.

HINT 1 for (0-2.a): Rewrite (0.2.1) using that $\cos(\xi) = \operatorname{Re}\exp(-\imath\xi)$ for all $\xi \in \mathbb{R}$. ⌟

SOLUTION of (0-2.a):

Since $x_j \in \mathbb{R}$ as indicated in the hint the formula (0.2.1) can be converted into

$$u_N(t) = \operatorname{Re}\left\{\sum_{j=0}^{N} x_j e^{2\pi\imath j t}\right\}\ , \quad t \in \mathbb{R}\ ,$$

which means

$$u_N\left(\frac{k}{M}\right) = \operatorname{Re}\left\{\sum_{j=0}^{N} x_j e^{-2\pi\imath j \frac{kj}{M}}\right\}\ , \quad k = 0, \dots, M-1\ . \tag{0.2.3}$$

Next recall the discrete Fourier transform from [Lecture → **??**]:

Definition [Lecture → ??]. Discrete Fourier transform

The linear map $\mathsf{DFT}_n : \mathbb{C}^n \mapsto \mathbb{C}^n$, $\mathsf{DFT}_n(\mathbf{y}) := \mathbf{F}_n\mathbf{y}$, $\mathbf{y} \in \mathbb{C}^n$, is called **discrete Fourier transform** (DFT), i.e. for $[c_0, \dots, c_{n-1}] := \mathsf{DFT}_n(\mathbf{y})$

$$c_k = \sum_{j=0}^{n-1} y_j\, \omega_n^{kj} = \sum_{j=0}^{n-1} y_j \exp\left(-2\pi\imath\frac{kj}{n}\right) \quad , \quad k = 0, \dots, n-1\,. \qquad \text{[Lecture} \to \textbf{??}\text{]}$$

Inspired by (0.2.3) we aim to apply it for $n = M$. Since $M > N$ this entails zero-padding of the vectors of coefficients. We define

$$\widetilde{x}_j := \begin{cases} x_j & \text{for } j \in \{0, \dots, N\}, \\ 0 & \text{for } j \in \{N+1, \dots, M-1\}\,. \end{cases} \quad , \qquad j = 0, \dots, M-1\,,$$

and collect these numbers into the vector $\widetilde{\mathbf{x}} \in \mathbb{R}^M$. Then (0.2.3) can be recast as

$$\left[u_N\left(\frac{k}{M}\right)\right]_{k=0}^{M-1} = \mathrm{Re}\{\mathsf{DFT}_M(\widetilde{\mathbf{x}})\}\,.$$

The implementation based on EIGEN's built-in FFT-based DFT facilities [Lecture → **??**] is straightforward.

C++ code 0.2.4: Evaluation of trigonometric sum (0.2.1) **at M equidistant points**

```cpp
Eigen::VectorXd eval_uN(const Eigen::VectorXd &x, unsigned int M) {
  unsigned int N = x.size() - 1;
  assert((M > N) && "Number of evaluation points must be > N");
  Eigen::VectorXd u(M);
  // START: Student solution
  // Zero-pad coefficient vector and copy it into a complex-valued vector
  Eigen::VectorXcd xt = Eigen::VectorXcd::Zero(M);
  xt.head(N + 1) = x;
  // Compute DFT of the vector xt
  Eigen::FFT<double> fft;
  Eigen::VectorXcd y = fft.fwd(xt);
  u = y.real();
  // END: student solution
  return u;
}
```

Note that we have to introduce vectors of type **Eigen::VectorXcd**, because EIGEN's FFT only operates on those.

▲

(0-2.b) ⚁ (6 pts.) Write the collocation equations (0.2.2) in the standard form $F(\mathbf{x}) = \mathbf{0}$ of a non-linear system of equations for an *explicitly given* $F : \mathbb{R}^n \to \mathbb{R}^n$ and suitable $\mathbf{x}$ and $n \in \mathbb{N}$.

$$n = \boxed{\quad\quad}\,, \quad \mathbf{x} = \begin{bmatrix} \boxed{\quad\quad} \end{bmatrix}\,,$$

$$F(\mathbf{x}) = \boxed{} \; .$$

HINT 1 for (0-2.b): Remember the derivatives $\cos' = -\sin$ and $\sin' = \cos$. ⌟

SOLUTION of (0-2.b):

By the chain rule we find for any $\alpha \in \mathbb{R}$

$$\frac{d^2}{dt^2}\{t \mapsto \cos(2\pi j t)\}(t) = -(2\pi j)^2 \cos(2\pi j t) \; .$$

Using this, it is straightforward from (0.2.2) that

$$n = N+1 \, , \quad \mathbf{x} = \begin{bmatrix} x_0 \\ \vdots \\ x_N \end{bmatrix} ,$$

$$F(\mathbf{x}) = \left[\sum_{j=0}^{N} x_j \cdot 4\pi^2 j^2 \cos\left(2\pi \tfrac{jk}{N+1}\right) + \left(\sum_{j=0}^{N} x_j \cdot \cos\left(2\pi \tfrac{jk}{N+1}\right)\right)^3 - \sin\left(\pi \tfrac{k}{N+1}\right) \right]_{k=0}^{N} \; . \tag{0.2.5}$$

▲

(0-2.c) ⚄ (7 pts.) [depends on Sub-problem (0-2.b)]

Relying on the function `eval_uN()` from Sub-problem (0-2.a) implement another C++ function in `periodiccollocation.hpp`,

```
Eigen::VectorXd eval_F(const Eigen::VectorXd &x);
```

that returns the vector $F(\mathbf{x})$ when given the argument $\mathbf{x}$ in `x`.

SOLUTION of (0-2.c):

Given $\mathbf{x} = [x_0, \dots, x_N]^\top \in \mathbb{R}^{N+1}$ we define

$$\begin{aligned} \mathbf{d} &:= \left[4\pi^2 j^2 x_j\right]_{j=0}^{N} \in \mathbb{R}^{N+1} \, , \\ \mathbf{y} &:= \texttt{eval\_uN}(\mathbf{d}, N+1) \, , \\ \mathbf{u} &:= \texttt{eval\_uN}(\mathbf{x}, N+1) \, . \end{aligned}$$

Then, from (0.2.5) it is immediate that

$$F(\mathbf{x}) = \mathbf{y} + \left[(\mathbf{u})_k^3\right]_{k=0}^N - \left[\sin(\pi\tfrac{k}{N+1})\right]_{k=0}^N .$$

This can be translated into C++ directly.

C++ code 0.2.6: Evaluation of F, with $F(\mathbf{x}) = \mathbf{0} \triangleq$ collocation equations (0.2.2)

```
Eigen::VectorXd eval_F(const Eigen::VectorXd &x) {
  const unsigned int N = x.size() - 1;
  Eigen::VectorXd Fx;
  // START Student code
  Eigen::ArrayXd s = Eigen::ArrayXd::LinSpaced(N + 1, 0, N);
  // Contribution $\left[\sin(\pi\frac{k}{N+1})\right]_k$
  const Eigen::VectorXd rhs = ((M_PI / (N + 1)) * s).sin().matrix();
  // Vector $\left[x_j \cdot 4\pi^2 j^2\right]_j$
  s = 4 * M_PI * M_PI * s * s;
  const Eigen::VectorXd d{(x.array() * s).matrix()};
  Fx = eval_uN(d, N + 1) + (eval_uN(x, N + 1).array().pow(3)).matrix() - rhs;
  // END Student code
  return Fx;
}
```

Of course, this is only one of many ways how this function can be implemented.

▲

(0-2.d) (7 pts.) [depends on Sub-problem (0-2.b)]

We want to try Newton's method to solve $F(\mathbf{x}) = \mathbf{0}$ with F from Sub-problem (0-2.b). To that end we need a function

```
Eigen::MatrixXd eval_DF(const Eigen::VectorXd &x);
```

that returns the Jacobian $\mathsf{D}F(\mathbf{x})$, when `x` contains the argument $\mathbf{x}$. Write such a function in the file `periodiccollocation.hpp`

HINT 1 for (0-2.d): The function `eval_uN()` from Sub-problem (0-2.a) may be used.

SOLUTION of (0-2.d):

For the function $F(\mathbf{x}) = [F_0(\mathbf{x}), \dots, F_N(\mathbf{x})]^\top$ encoding (0.2.2) and found in Sub-problem (0-2.b) the Jacobian $\mathsf{D}F(\mathbf{x}) \in \mathbb{R}^{N+1,N+1}$ for $\mathbf{x} \in \mathbb{R}^{N+1}$ is the matrix $\left[\frac{\partial F_k}{\partial x_j}(\mathbf{x})\right]_{k,j=0}^N$.

From (0.2.5) we directly see, $\mathbf{x} = [x_0, \dots, x_N]$,

$$F_k(\mathbf{x}) = \sum_{\ell=0}^N x_\ell \cdot 4\pi^2\ell^2 \cos(2\pi\ell\tfrac{k}{N+1}) + \left(\sum_{\ell=0}^N x_\ell \cdot \cos(2\pi\ell\tfrac{k}{N+1})\right)^3 - \sin(\pi\tfrac{k}{N+1}) .$$

Forming the partial derivative with respect to x_j is straightforward using the chain rule on the second summand:

$$\frac{\partial F_k}{\partial x_j}(\mathbf{x}) = 4\pi^2 j^2 \cos(2\pi j \tfrac{k}{N+1}) + 3\underbrace{\Big(\sum_{\ell=0}^{N} x_\ell \cdot \cos(2\pi\ell \tfrac{k}{N+1})\Big)}_{=u_N(\frac{k}{N+1})}{}^2 \cos(2\pi j \tfrac{k}{N+1}) \;.$$

C++ code 0.2.7: Evaluation of the Jacobian $\mathrm{D}\,F(\mathbf{x})$

```cpp
Eigen::MatrixXd eval_DF(const Eigen::VectorXd &x) {
  const unsigned int N = x.size() - 1;
  const Eigen::VectorXd u{eval_uN(x, N + 1)};
  Eigen::MatrixXd J(N + 1, N + 1);
  double fourpisq = 4 * M_PI * M_PI;
  for (unsigned int k = 0; k <= N; ++k) {
    const double fac = (2 * M_PI * k) / static_cast<double>(N + 1);
    const double uNksq = 3.0 * u[k] * u[k];
    for (unsigned int j = 0; j <= N; ++j) {
      J(k, j) = (fourpisq * j * j + uNksq) * std::cos(fac * j);
    }
  }
  return J;
}
```

▲

End Problem 0-2 , 27 pts.

Problem 0-3: Closed Curve Interpolation with Quadratic Splines

The reconstruction of curves and surfaces from given points is a central task in computer-aided design (CAD). Here we focus on the problem of finding a closed and C^1-smooth curve running through a sequence of given points. That curve should be modeled by means of a periodic quadratic spline function.

This problem relies on [Lecture → **??**] and has a focus on implementation in C++ using EIGEN.

We are given n, $n \in \mathbb{N}$, mutually different points $\mathbf{p}^1, \dots, \mathbf{p}^n \in \mathbb{R}^2$ in the plane. Defining a knot sequence $\mathcal{M} := \{0 = t_0 < t_1 < \dots < t_{n-1} < t_n = 1\}$ as [2]

$$t_j := \frac{\sum_{k=1}^{j} \left\| \mathbf{p}^k - \mathbf{p}^{k-1} \right\|_2}{\sum_{k=1}^{n} \left\| \mathbf{p}^k - \mathbf{p}^{k-1} \right\|_2} \quad (\, \mathbf{p}^0 := \mathbf{p}^n \,)\,, \quad j = 0, \dots, n\,, \tag{0.3.1}$$

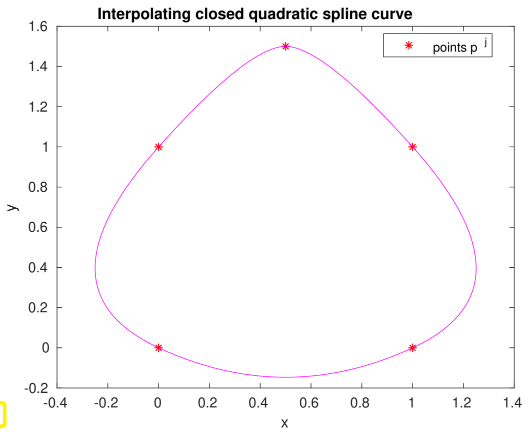

Fig. 1

we want to find a spline curve $\mathbf{s} : [0,1] \to \mathbb{R}^2$ satisfying

$$\mathbf{s} \in (\mathcal{S}_{2,\mathcal{M}})^2\,, \quad \text{c.f. [Lecture} \to \textbf{??}], \tag{0.3.2}$$

$$\mathbf{s}(t_j) = \mathbf{p}^j \quad \text{for all} \quad j = 1, \dots, n\,, \tag{0.3.3}$$

$$\mathbf{s}(t_0) = \mathbf{p}^n \quad \text{and} \quad \mathbf{s}'(t_0) = \mathbf{s}'(t_n)\,, \tag{0.3.4}$$

where $'$ indicates differentiation with respect to the curve parameter. The requirement (0.3.4) ensures that $\mathbf{s}$ describes a *closed* *C^1-smooth* curve.

Throughout, for $\mathbf{S}$ use the following local representation on knot intervals:

$$\mathbf{s}|_{]t_{j-1},t_j[}(t) = \mathbf{y}^{j-1}(1-\tau) + \mathbf{x}^j h_j \tau(1-\tau) + \mathbf{y}^j \tau\,, \quad \tau := \frac{t - t_{j-1}}{h_j}\,, \quad \begin{array}{l} t_{j-1} < t < t_j\,, \\ h_j := t_j - t_{j-1}\,, \end{array} \tag{0.3.5}$$

with unknown vector-valued coefficients $\mathbf{y}^j \in \mathbb{R}^2$, $j = 0, \dots, n$, and $\mathbf{x}^j \in \mathbb{R}^2$, $j = 1, \dots, n$.

(0-3.a) ⚀ (3 pts.) What are the benefits of using the local representation (0.3.5)?

SOLUTION of (0-3.a):

Since

$$t = t_{j-1} \;\Rightarrow\; \tau = 0\,, \quad t = t_j \;\Rightarrow\; \tau = 1\,, \tag{0.3.6}$$

we find

$$\mathbf{s}|_{]t_{j-1},t_j[}(t_{j-1}) = \mathbf{y}^{j-1} \quad , \quad \mathbf{s}|_{]t_{j-1},t_j[}(t_j) = \mathbf{y}^j\,, \quad j = 1, \dots, n\,. \tag{0.3.7}$$

Thus

- the uniqueness of the coefficients $\mathbf{y}^j$ immediately leads to

$$\mathbf{s}|_{]t_{j-1},t_j[}(t_j) = \mathbf{y}^j = \mathbf{s}|_{]t_j,t_{j+1}[}(t_j)\,, \quad j = 1, \dots, n-1\,,$$

which guarantees the continuity of $\mathbf{s}$, and

[2] Sums whose lower summation bound exceeds the upper are supposed to evaluate to zero.

- the interpolation conditions (0.3.3)/(0.3.4) can easily be enforced by setting

$$\mathbf{y}^j := \mathbf{p}^j \,, \quad j = 1,\dots,n \quad , \quad \mathbf{y}^0 := \mathbf{p}^n \,. \tag{0.3.8}$$

As a consequence of (0.3.8) the vector coefficients $\mathbf{y}^j$, $j = 0,\dots,n$, can be assumed to be known.

▲

(0-3.b) (15 pts.)

In explicit form state the linear system of equations

- whose coefficient matrix and right-hand side vector are to be given in terms of the point locations $\mathbf{p}^j \in \mathbb{R}^2$, $j = 1,\dots,n$ and the knot positions t_k, $k = 0,\dots,n$, and
- whose solution provides the vector $\left[(\mathbf{x}^1)_1, (\mathbf{x}^2)_1, \dots, (\mathbf{x}^n)_1\right]^\top \in \mathbb{R}^n$ of first components of the vector-valued coefficients $\mathbf{x}^j \in \mathbb{R}^2$, $j = 1,\dots,n$, in (0.3.5).

$$\begin{bmatrix} \quad \end{bmatrix} \begin{bmatrix} (\mathbf{x}^1)_1 \\ \vdots \\ \vdots \\ (\mathbf{x}^n)_1 \end{bmatrix} = \begin{bmatrix} \quad \end{bmatrix} .$$

SOLUTION of (0-3.b):

We start with the important observation that, assuming the knot sequence $\mathcal{M} := \{0 = t_0 < t_1 < \dots < t_{n-1} < t_n = 1\}$ to be given, the first component of $\mathbf{s}$ depends only on the first components of the points $\mathbf{p}^j$. Thus, for the sake of brevity let us write

$$s(t) := (\mathbf{s}(t))_1 \,, \quad x_j := \left(\mathbf{x}^j\right)_1 \,, \quad j = 1,\dots,n \,, \quad p_j := \begin{cases} (\mathbf{p}^j)_1 & \text{for } j = 1,\dots,n \,, \\ (\mathbf{p}_n)_1 & \text{for } j = 0 \,. \end{cases}$$

In light of (0.3.8) we get the local representation

$$s|_{]t_{j-1},t_j[}(t) = p_{j-1}(1-\tau) + x_j h_j \tau (1-\tau) + p_j \tau \,, \quad \tau := \frac{t - t_{j-1}}{h_j} \,, \quad \begin{array}{l} t_{j-1} < t < t_j \,, \\ h_j := t_j - t_{j-1} \,. \end{array} \tag{0.3.9}$$

As we have seen, the continuity conditions at the knots are implied by this representation. What remains to be satisfied is the *continuity of the first derivative s'* plus the second part of (0.3.4). This amounts to the n conditions

$$s'\big|_{]t_{j-1},t_j[}(t_j) = s'\big|_{]t_j,t_{j+1}[}(t_j)\,, \quad j = 1,\dots,n-1\,, \tag{0.3.10a}$$
$$s'\big|_{]t_0,t_1[}(t_0) = s'\big|_{]t_{n-1},t_n[}(t_n)\,. \tag{0.3.10b}$$

By differentiating the local representation (0.3.9) using the chain rule and $\frac{d\tau}{dt} = h_j^{-1}$, we deduce

$$s'\big|_{]t_{j-1},t_j[}(t) = -h_j^{-1}p_{j-1} + x_j(1-2\tau) + h_j^{-1}p_j\,, \quad \tau := \frac{t-t_{j-1}}{h_j}\,, \quad t_{j-1} < t < t_j\,. \tag{0.3.11}$$

Owing to (0.3.6) this means that

$$s'\big|_{]t_{j-1},t_j[}(t_j) = -h_j^{-1}p_{j-1} - x_j + h_j^{-1}p_j\,, \qquad s'\big|_{]t_{j-1},t_j[}(t_{j-1}) = -h_j^{-1}p_{j-1} + x_j + h_j^{-1}p_j\,, \tag{0.3.12}$$
$$s'\big|_{]t_j,t_{j+1}[}(t_{j+1}) = -h_{j+1}^{-1}p_j - x_{j+1} + h_{j+1}^{-1}p_{j+1}\,, \quad s'\big|_{]t_j,t_{j+1}[}(t_j) = -h_{j+1}^{-1}p_j + x_{j+1} + h_{j+1}^{-1}p_{j+1}\,. \tag{0.3.13}$$

We plug these expressions into (0.3.10) and obtain linear equations

$$\text{(0.3.10b)} \quad \Rightarrow \quad -h_1^{-1}p_0 + x_1 + h_1^{-1}p_1 = -h_n^{-1}p_{n-1} - x_n + h_n^{-1}p_n\,, \tag{0.3.14}$$
$$\text{(0.3.10a)} \quad \Rightarrow \quad -h_j^{-1}p_{j-1} - x_j + h_j^{-1}p_j = -h_{j+1}^{-1}p_j + x_{j+1} + h_{j+1}^{-1}p_{j+1},\ j = 1,\dots,n-1\,. \tag{0.3.15}$$

Arranging the unknowns on the left side we get

$$x_1 + x_n = h_1^{-1}p_0 - h_1^{-1}p_1 - h_n^{-1}p_{n-1} + h_n^{-1}p_n\,, \tag{0.3.16}$$
$$x_j + x_{j+1} = -h_j^{-1}p_{j-1} + h_j^{-1}p_j + h_{j+1}^{-1}p_j - h_{j+1}^{-1}p_{j+1}\,, \quad j = 1,\dots,n-1\,. \tag{0.3.17}$$

This is an $n \times n$ linear system of equations for the unknowns $x_1,\dots,x_n$. Its matrix-vector form is given by

$$\begin{bmatrix} 1 & 0 & \dots & \dots & 0 & 1 \\ 1 & 1 & 0 & \dots & & 0 \\ 0 & \ddots & \ddots & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & 1 & 1 & 0 \\ 0 & \dots & \dots & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} \frac{p_n - p_{n-1}}{h_n} - \frac{p_1 - p_0}{h_1} \\ \frac{p_1 - p_0}{h_1} - \frac{p_2 - p_1}{h_2} \\ \frac{p_2 - p_1}{h_2} - \frac{p_3 - p_2}{h_3} \\ \vdots \\ \vdots \\ \frac{p_{n-1} - p_{n-2}}{h_{n-1}} - \frac{p_n - p_{n-1}}{h_n} \end{bmatrix}\,. \tag{0.3.18}$$

Remove the abbreviations:

$$\begin{bmatrix} 1 & 0 & \dots & \dots & 0 & 1 \\ 1 & 1 & 0 & \dots & & 0 \\ 0 & \ddots & \ddots & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & 1 & 1 & 0 \\ 0 & \dots & \dots & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} (\mathbf{x}^1)_1 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ (\mathbf{x}^n)_1 \end{bmatrix} = \begin{bmatrix} \frac{(\mathbf{p}^n)_1 - (\mathbf{p}^{n-1})_1}{h_n} - \frac{(\mathbf{p}^1)_1 - (\mathbf{p}^n)_1}{h_1} \\ \frac{(\mathbf{p}^1)_1 - (\mathbf{p}^n)_1}{h_1} - \frac{(\mathbf{p}^2)_1 - (\mathbf{p}^1)_1}{h_2} \\ \frac{(\mathbf{p}^2)_1 - (\mathbf{p}^1)_1}{h_2} - \frac{(\mathbf{p}^3)_1 - (\mathbf{p}^2)_1}{h_3} \\ \vdots \\ \vdots \\ \frac{(\mathbf{p}^{n-1})_1 - (\mathbf{p}^{n-2})_1}{h_{n-1}} - \frac{(\mathbf{p}^n)_1 - (\mathbf{p}^{n-1})_1}{h_n} \end{bmatrix}\,. \tag{0.3.19}$$

▲

In an object-oriented C++ code, for handling interpolating closed C^1-smooth curves as specified above we rely on the following data type:

C++ code 0.3.20: Class definition of ClosedQuadraticSplineCurve

```
class ClosedQuadraticSplineCurve {
 public:
  // Constructor
  explicit ClosedQuadraticSplineCurve(
      const Eigen::Matrix<double, 2, Eigen::Dynamic> &p);
  ~ClosedQuadraticSplineCurve() = default;
  ClosedQuadraticSplineCurve() = delete;
  ClosedQuadraticSplineCurve(const ClosedQuadraticSplineCurve &) = delete;
  ClosedQuadraticSplineCurve(const ClosedQuadraticSplineCurve &&) = delete;
  ClosedQuadraticSplineCurve &operator=(const ClosedQuadraticSplineCurve &) =
      delete;
  // Point evaluation operator for sorted parameter arguments
  Eigen::Matrix<double, 2, Eigen::Dynamic> curve_points(
      const Eigen::VectorXd &v) const;

 private:
  // Number of points to be interpolated
  unsigned int n_;
  // Coordinates of interpolated points, duplicate: p^0 = p^n
  Eigen::Matrix<double, 2, Eigen::Dynamic> p_;
  // Knot sequence
  Eigen::VectorXd t_;
  // Coefficients in local representation
  Eigen::Matrix<double, 2, Eigen::Dynamic> x_;
};
```

The constructor initializes the columns of the matrix-valued class variable p_ with the coordinate vectors $\mathbf{p}^j$, $j = 1, \dots, n$, the class variable t_ with the vector $[t_0, \dots, t_n]^\top$ as defined in (0.3.1), and it computes the vector-valued coefficients $\mathbf{x}^j$ from (0.3.5) and stores them in the columns of x_.

(0-3.c) ⚃ (7 pts.) Complete the code in `periodicquadraticsplines.hpp` to achieve an *efficient* implementation, that is, with cost scaling linearly in the number of data and in the number n of points, of the member function

```
Eigen::Matrix<double, 2, Eigen::Dynamic>
ClosedQuadraticSplineCurve::curve_points(const Eigen::VectorXd &v)
    const;
```

that takes a *sorted* sequence $(0 \leq v_1 \leq v_2 \leq \cdots \leq v_{N-1} \leq v_N \leq 1)$, $N \in \mathbb{N}$, of parameter values as v and returns the sequence $(\mathbf{s}(v_1), \mathbf{s}(v_2), \dots, \mathbf{s}(v_N))$ as the columns of a $2 \times N$-matrix.

SOLUTION of (0-3.c):

The key to an efficient implementation is to exploit the fact that the v-array is sorted. Thus, we can find the knot interval, in which the next v_j is located with a total number of $O(n+N)$ comparisons. Once we have found that $v_k \in [t_{j-1}, t_j[$, we can simply use the formula (0.3.5) to compute $\mathbf{s}(v_k)$.

C++ code 0.3.21: Class definition of ClosedQuadraticSplineCurve

```cpp
Eigen::Matrix<double, 2, Eigen::Dynamic>
ClosedQuadraticSplineCurve::curve_points(const Eigen::VectorXd &v) const {
  unsigned int N = v.size();
  assert(N > 0);
  // Matrix containing points to be computed
  Eigen::Matrix<double, 2, Eigen::Dynamic> s(2, N);
  // Lengths of knot intervals
  const Eigen::VectorXd h = t_.tail(n_) - t_.head(n_);
  // START Student code
  // Run through all the provided parameter values
  unsigned int knot_idx = 0;
  for (unsigned int k = 0; k < N; ++k) {
    assert(((v[k] >= 0.0) && (v[k] < 1.0)) && "Out of range!");
    if (k > 0) {
      assert(v[k] >= v[k - 1] && "Parameter values not sorted!");
    }
    // Find right knot interval: knot_idx stores index of knot to the
       right of
    // current parameter value
    while ((knot_idx < n_) && (v[k] >= t_[knot_idx])) {
      knot_idx++;
    }
    assert(knot_idx > 0);
    const double tau = (v[k] - t_[knot_idx - 1]) / h[knot_idx - 1];
    s.col(k) = ((1.0 - tau) * p_.col(knot_idx - 1)) +
               (h[knot_idx - 1] * tau * (1 - tau)) * x_.col(knot_idx - 1) +
               (tau * p_.col(knot_idx));
  }
  // END Student code
  return s;
}
```

The consistency checks for the parameter values `v[k]` are optional.

▲

End Problem 0-3 , 25 pts.