Numerical Methods for CSE
Final Exam - Summer 2019

Prof. Rima Alaifari

ETH Zürich, D-MATH

Exam Organisers:
Soumil Gurjar, Pratyuksh Bansal

August 26, 2019

Exercise 1. *Sparsity Preservation* (26 pts) [*Template: 1.cpp*]

Consider the following problem: Given $\mathbf{A} \in \mathbb{R}^{m,n}$, $m \geq n$, $\text{rank}(\mathbf{A}) = n$, $\mathbf{b} \in \mathbb{R}^{m}$, we want to obtain the unique least squares solution of the linear system of equations $\mathbf{Ax} = \mathbf{b}$.

(a) (2 pts) Implement a C++ function

`VectorXd normalEquationSolve(const MatrixXd &A, const VectorXd &b);`

that returns the least squares solution of $\mathbf{Ax} = \mathbf{b}$ by solving the normal equations.

Solution:

```
VectorXd normalEquationSolve(const MatrixXd &A, const VectorXd &b){
    if (b. size () != A.rows() ) throw std::runtime_error("Dimension mismatch");  // Cholesky solver
    VectorXd x = (A.transpose()*A).llt().solve(A.transpose()*b);
    return x ;
}
```

question_1/codes/1_sol.cpp

(b) (2 pts) Now consider a matrix $\mathbf{A}$ which is sparse. While performing the computation of the solution by normal equations, the sparsity is not necessarily preserved. Construct an example of a sparse $n \times n$ matrix such that the system matrix of the normal equations has no zero entries.

Solution: For example, considering a sparse matrix

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix},$$

when computing the solution by normal equations, we perform the matrix product $\mathbf{A}^{\top}\mathbf{A}$, which would be a dense matrix of only ones.

(c) (6 pts) Through an augmented system $\mathbf{Cy} = \mathbf{d}$, sparsity can be maintained. Implement two C++ functions,

`SparseMatrix<double> computeC(SparseMatrix<double> &A, const VectorXd &b);`

and

`VectorXd computeRHS(SparseMatrix<double> &A, const VectorXd &b);`

that respectively compute the *sparse* matrix $\mathbf{C}$ and RHS-vector $\mathbf{d}$.

Solution:

```
SparseMatrix<double> computeC(SparseMatrix<double> &A, const VectorXd &b){
    int m = A.rows();
    int n = A.cols();
    int sizeC = m+n;
    std :: vector < Triplet <double > > triplets_C ;
    triplets_C.reserve(m+2*A.nonZeros()) ;

    // Fill the triplets with elements from A and A_transpose
    for(int i = 0; i < A.outerSize(); ++i){
        for(SparseMatrix<double>::InnerIterator it(A,i); it; ++it){
            triplets_C.emplace_back(it.row(),it.col()+m,it.value());
            triplets_C.emplace_back(it.col()+m,it.row(),it.value());
        }
    }

    //Fill the triplets corresponding to the Identity Block
    for (int k=0; k<m; ++k){
        triplets_C.push_back(Triplet<double>(k ,k ,-1.0));
    }

    SparseMatrix <double> C(sizeC, sizeC) ;
    C.setFromTriplets(triplets_C.begin(), triplets_C.end());
    C.makeCompressed();
    return C;
}
```

question_1/codes/1_sol.cpp

```
VectorXd computeRHS(SparseMatrix<double> &A, const VectorXd &b){
    int m = A.rows();
    int n = A.cols();
    int sizeC = m+n;
    VectorXd d = VectorXd :: Zero(sizeC);
    d.head(m) = b;

    return d ;
}
```

question_1/codes/1_sol.cpp

(d) (4 pts) Implement a C++ function

`VectorXd computeSolution(SparseMatrix<double> &A, const VectorXd &b);`

that solves the modified system using a standard sparse solver (SparseLU) and returns the required solution vector $\mathbf{x}$ corresponding to the original system $\mathbf{A}\mathbf{x} = \mathbf{b}$.

Solution:

```
VectorXd computeSolution(SparseMatrix<double> &A, const VectorXd &b){
    VectorXd x, y;
    VectorXd d = computeRHS(A, b);
    SparseMatrix<double> C = computeC(A, b);
    // Perform sparse elimination
    SparseLU<SparseMatrix<double>> solver(C);
    y = solver.solve(d);

    return x = y.tail(A.cols()); //The actual solution vector 'x' is the last 'n'
   elements of y
}
```

question_1/codes/1_sol.cpp

(e) (6 pts) Now we consider a specific example of the 1D Poisson equation, $-\ddot{u} = f$ on $\Omega = [0, 1]$, with Dirichlet boundary conditions $u(0) = u(1) = 0$. We apply a finite difference method, which allows us to obtain the solution to this problem by solving a linear system of equations. This is done in the following way:

The independent variable, $x \in \Omega$, is discretized by choosing an equidistant grid $\Omega_N = \{x_j\}_{j=0}^{N+1} \in \Omega$, where the grid points are given by $x_j = j\Delta x$, and where the spacing between the grid points is referred to as the mesh width, $\Delta x = \frac{1}{N+1}$. This means that $x_0 = 0$ is the left endpoint, and $x_{N+1} = 1$ is the right endpoint. In between, there are N internal points in the interval [0, 1].

The solution will be approximated numerically on this grid, by a vector $\mathbf{u} = \{u_j\}_1^N$ of internal points, augmented by the boundary values, $u_0 = 0$ and $u_{N+1} = 0$. The vector $\mathbf{u}$ approximates the solution to the differential equation, according to $u_j \approx u(x_j)$. Now, the second order derivative $-\ddot{u}$ is approximated by a symmetric finite difference scheme,

$$-\ddot{u}(x_j) = -\left(\frac{u_{j-1} - 2u_j + u_{j+1}}{(\Delta x)^2}\right).$$

Applying the finite difference method for all the interior points, obtain a linear system of equations of the form

$$L_N \mathbf{u} = \mathbf{f},$$

where the vector $\mathbf{f} = (f(x_1), f(x_2), \ldots, f(x_N))^\top$, $\mathbf{u} = (u_1, u_2, \ldots, u_N)^\top$ and $L_N \in \mathbb{R}^{N,N}$. Explicitly write out the matrix L_N.

Solution: A symmetric finite difference scheme leads to a linear system of equations:

$$\left(\frac{0 + 2u_1 - u_2}{(\Delta x)^2}\right) = f(x_1)$$

$$\left(\frac{-u_{j-1} + 2u_j - u_{j+1}}{(\Delta x)^2}\right) = f(x_j) \qquad j = 2 \ldots N-1$$

$$\left(\frac{-u_{N-1} - 2u_N - 0}{(\Delta x)^2}\right) = f(x_N),$$

where the first and last equations include the zero-valued boundary conditions, and the middle equation is the generic equation, only involving internal points.

This linear system is conveniently represented in matrix-vector form $L_N \mathbf{u} = \mathbf{f}$, as

$$\frac{1}{(\Delta x)^2}\begin{pmatrix} 2 & -1 & 0 & 0 & \cdots & & \cdots & 0 \\ -1 & 2 & -1 & 0 & \cdots & & & 0 \\ 0 & -1 & 2 & -1 & \ddots & & & \\ \vdots & & \ddots & \ddots & \ddots & & & \\ & & & \ddots & \ddots & & \ddots & \\ & & & & -1 & 2 & -1 & 0 \\ \vdots & & & & & -1 & 2 & -1 \\ 0 & \cdots & & & & 0 & -1 & 2 \end{pmatrix} \begin{pmatrix} u_1 \\ \vdots \\ \\ \\ \vdots \\ u_N \end{pmatrix} = \begin{pmatrix} f(x_1) \\ \vdots \\ \\ \\ \vdots \\ f(x_N) \end{pmatrix}$$

Thus, the $N \times N$ tridiagonal matrix can be written

$$L_N = \frac{1}{(\Delta x)^2}\mathbf{T}$$

with

$$\mathbf{T} = tridiag(-1 \quad 2 \quad -1).$$

(f) (6 pts) For the 1D Poisson problem described above, if $f = \pi^2 \sin(\pi x)$ and $N = 25, 50, 100, ..., 1600$, implement two functions, `constructDenseA_Poisson` and `constructSparseA_Poisson`, that construct the matrix L_N in dense and sparse format respectively for each mesh-width. Solve the corresponding linear system $L_N \mathbf{u} = \mathbf{f}$, using the normal equations from sub-problem (a), as well as the modified system from sub-problem (d). Compare the two solutions to verify your implementation, and also compare their run-times for each value of N. Explain your observations.

Hint: A dense matrix `A_dense` can be converted into a sparse matrix `A_sparse` by using `A_sparse = A_dense.sparseView()`. Moreover, the code to compare the solutions and run-times is already implemented within the template.

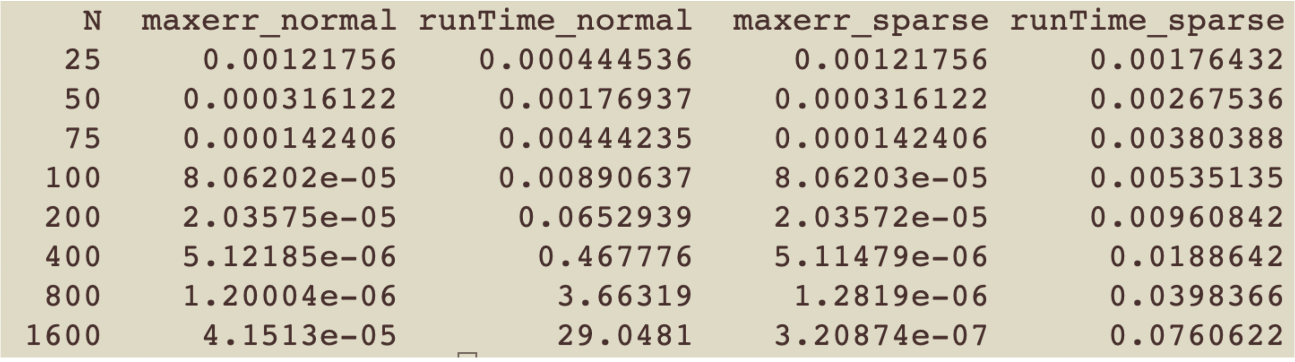
Solution:

The exact solution to this problem is $u = \sin(\pi x)$. Thus, at the interior nodes, we have the exact solution $u_i = \sin(\pi x_i)$

```
//Construct dense matrix 'A' obtained through Finite Difference Method for the poisson problem
MatrixXd constructDenseA_Poisson(int N){
    // Mesh width
    double delta_x = 1.0/(N+1);
    MatrixXd A_dense = MatrixXd::Zero(N,N);

    A_dense.diagonal() = VectorXd::Constant(N,2);
    A_dense.diagonal<1>() = VectorXd::Constant(N-1,-1);
    A_dense.diagonal<-1>() = VectorXd::Constant(N-1,-1);
    A_dense *= 1/(delta_x*delta_x);

    return A_dense;
}

//Construct sparse matrix 'A' obtained through Finite Difference Method for the poisson problem
SparseMatrix <double> constructSparseA_Poisson(MatrixXd &A_dense, int N){
    SparseMatrix <double> A_sparse = A_dense.sparseView(); //Convert dense matrix A into a sparse form
    return A_sparse;
}
```

question_1/codes/1_sol.cpp

RunTime Comparison: For $N \geq 75$, we observe that the run-time for the augmented system method is lower than that of the normal system method. The benefit of the augmented system and preservation of sparsity can be clearly seen for the cases with $N = 200, 400, 800, 1600$.



```
   N  maxerr_normal runTime_normal  maxerr_sparse runTime_sparse
  25     0.00121756    0.000444536     0.00121756     0.00176432
  50    0.000316122     0.00176937    0.000316122     0.00267536
  75    0.000142406     0.00444235    0.000142406     0.00380388
 100    8.06202e-05     0.00890637    8.06203e-05     0.00535135
 200    2.03575e-05      0.0652939    2.03572e-05     0.00960842
 400    5.12185e-06       0.467776    5.11479e-06      0.0188642
 800    1.20004e-06        3.66319     1.2819e-06      0.0398366
1600     4.1513e-05        29.0481    3.20874e-07      0.0760622
```

Figure 1: Run-time comparison between Normal Equation method and Augmented System method

Exercise 2. *Interpolation* (28 pts) [*Template: 2.cpp*]

Consider the function

$$f(x) = \begin{cases} f_1(x) := 2 + \frac{1}{1+25(6x-1)^2}, & x \in [0, \frac{1}{3}), \\ f_2(x) := 1 + \cos(\pi x), & x \in [\frac{1}{3}, 1]. \end{cases} \tag{1}$$

Let $I^p_{\mathcal{T}_N}[f]$ be a piecewise Lagrange interpolant of the function f, for a mesh $\mathcal{T}_N$ on the interval $[0, 1]$. The mesh $\mathcal{T}_N$ has N cells and the local polynomial degree $p \in \mathbb{N}_0$ of the interpolant is the same for each cell in the mesh.
Two functions are provided, see `2.cpp`, to generate equidistant and Chebyshev interpolation nodes in the reference interval $[0, 1]$. Let $t = \{t_0, t_1, \ldots, t_p\} \subset [0, 1]$ denote the set of these reference nodes.

(a) (2 pts) Is $f \in C^0([0, 1])$? Explain.

Solution: No, $f \notin C^0([0, 1])$. The given function is discontinuous at $x = 1/3$,

$$f_1(\frac{1}{3}) = 2 + \frac{1}{26} \approx 2.04 \neq 1.5 = f_2(\frac{1}{3}).$$

It is continuous for $x \in [0, 1]\backslash\{\frac{1}{3}\}$.

(b) (5 pts) For $N = 2$, design the interpolant:

- i) specify the mesh and
- ii) specify the strategy to generate local interpolation nodes,

such that the interpolation error $\|I^p_{\mathcal{T}_N}[f] - f\|_{L^\infty([0,1])}$ is small for an arbitrarily large p. Support your answer.

Solution:

- i) For $N = 2$, we choose the mesh vertices $\{\mathbf{0}, \mathbf{\frac{1}{3}}, \mathbf{1}\}$. For any other choice of mesh vertices $\{0, \xi, 1\}$, $\xi \neq \frac{1}{3}$, the point of discontinuity $x = \frac{1}{3}$ is included in either $[0, \xi]$ or $[\xi, 1]$. Therefore, the function is not smooth in one of the two cells and increasing the polynomial degree p of the interpolation will not reduce the interpolation error.
- ii) In the cell $(\frac{1}{3}, 1)$, we can simply generate *equidistant* local interpolation nodes. In the cell $(0, \frac{1}{3})$, the given function is an affine transformation of the **Runge's function**, thus, Lagrange interpolation is unstable for equidistant nodes but stable for *Chebyshev* nodes, for a large polynomial degree p.

(c) (2 pts) Consider a *uniform* mesh $\mathcal{T}_M$ on the interval $[a, b]$. Implement a function

```
Eigen::MatrixXd genNodes(const double a,
                         const double b,
                         const int M,
                         const Eigen::VectorXd &t);

```

that uses a vector of reference nodes $\mathtt{t} \in \mathbb{R}^n$ and generates a matrix $\mathbb{R}^{n\times M}$ of interpolation nodes in the mesh $\mathcal{T}_M$.

Hint: Map the reference nodes to a given cell.

Solution:

```cpp
Eigen::MatrixXd genNodes(const double a, const double b, const int M, const Eigen::VectorXd &knots)
{
    const int n = knots.size();
    double H = (b-a)/M;

    Eigen::MatrixXd nodes(n,M);
    for (int j=0; j<M; j++) {
        nodes.col(j) = (a + j*H) + knots.array()*H;
    }

    return nodes;
}
```

question_2/codes/2_sol.cpp

(d) (8 pts) Given the declaration of a `class` **Newton**, implement the member functions:

- i) `Interpolate` : Computes the coefficients of a piecewise Lagrange interpolant in the Newton basis.
- ii) `Evaluate` : Uses the *Horner scheme* to evaluate a piecewise Lagrange interpolant in the Newton basis.

Solution:

```cpp
void Newton::Interpolate(const Eigen::MatrixXd &y)
{
    int n = nodes.rows();
    int M = nodes.cols();
    assert(y.rows() == n);
    assert(y.cols() == M);

    // set-up and solve linear system
    for (int l=0; l<M; l++)
    {
        Eigen::MatrixXd A(n,n);
        A.setZero();

        for (int i=0; i<n; i++) {
            A(i,0) = 1;
            for (int j=1; j<=i; j++) {
                A(i,j) = A(i,j-1)*(nodes(i,l) - nodes(j-1,l));
            }
        }

        alpha.col(l) = A.triangularView<Eigen::Lower>().solve(y.col(l));
    }

    // solve with divided difference method
    /*for (int l=0; l<M; l++) {

        alpha.col(l) = y.col(l);
        for (int j=0; j<n-1; ++j) {
            for (int i=n-1; i>j; --i) {
                alpha(i,l) = (alpha(i,l) - alpha(i-1,l)) / (nodes(i,l) - nodes(i-1-j,l));
            }
        }
    }*/
}
```

question_2/codes/2_sol.cpp

```
double Newton::Evaluate(const int l, double x)
{
    int n = alpha.rows();
    double y = alpha(n-1,l);
    for (int i = n-2; i >= 0; --i) { // Horner scheme
        y = y * (x - nodes(i,l)) + alpha(i,l);
    }

    return y;
}
```

question_2/codes/2_sol.cpp

(e) (8 pts) Use the components from sub-problems (c) and (d), and write a `C++` function to compute a piecewise Lagrange interpolant of the given function $f(x)$ from (1) on a uniform mesh $\mathcal{T}_N$. This function should also evaluate and return the interpolation error.

Hint: Use the given member function `EvalError`, of `class` **Newton**, to compute the interpolation error.

Solution:

```
// Given function f
auto f = [](const double x)
           {
                if (x < 1./3.)
                    return (1.0 / (1.0 + 25.0 * (6*x-1) * (6*x-1)));
                else
                    return (1.0 + cos(M_PI*x));
           };

// function f1
auto f1m = [](const Eigen::MatrixXd x)
            {
                return (1.0 / (1.0 + 25.0 * (6*x.array()-1) * (6*x.array()-1))).
    matrix();
            };

// function f2
auto f2m = [](const Eigen::MatrixXd x)
            {
                return (1.0 + cos(M_PI*x.array())).matrix();
            };

////TASK E
// Computes the piecewise Lagrange interpolant
// and evaluates the interpolation error
double piecewiseLagrange_cheb(const int p, const int l)
{
    double a = 0;
    double c = 1./3.;
    double b = 1;

    int N1 = std::pow(2,l);
    int N2 = 2*N1;

    // generate grid
    Eigen::VectorXd grid = Eigen::VectorXd::LinSpaced(N1+N2+1, a, b);

    // generate reference nodes
    Eigen::VectorXd knotsCheb = genKnotsCheb(p+1);
    Eigen::VectorXd knotsEqui = genKnotsEqui(p+1);

    // generate nodes
    Eigen::MatrixXd x(p+1, N1+N2);
    x.block(0, 0, p+1, N1) = genNodes(a, c, N1, knotsCheb);
    x.block(0, N1, p+1, N2) = genNodes(c, b, N2, knotsEqui);

    // generate data
    Eigen::MatrixXd y(p+1, N1+N2);
    y.block(0, 0, p+1, N1) = f1m(x.block(0, 0, p+1, N1));
    y.block(0, N1, p+1, N2) = f2m(x.block(0, N1, p+1, N2));

    // solve interpolant
    Newton ipolf(grid, x);
    ipolf.Interpolate(y);

    // compute interpolation error
    double error = ipolf.EvalError(f);

    return error;
}
```

question_2/codes/2_sol.cpp

or if equidistant nodes are used:

```
double piecewiseLagrange_equi(const int p, const int l)
{
    double a = 0;
    double c = 1./3.;
    double b = 1;

    int N1 = std::pow(2,l);
    int N2 = 2*N1;

    // generate grid
    Eigen::VectorXd grid = Eigen::VectorXd::LinSpaced(N1+N2+1, a, b);

    // generate reference nodes
    Eigen::VectorXd knotsEqui = genKnotsEqui(p+1);

    // generate nodes
    Eigen::MatrixXd x(p+1, N1+N2);
    x.block(0, 0, p+1, N1) = genNodes(a, c, N1, knotsEqui);
    x.block(0, N1, p+1, N2) = genNodes(c, b, N2, knotsEqui);

    // generate data
    Eigen::MatrixXd y(p+1, N1+N2);
    y.block(0, 0, p+1, N1) = f1m(x.block(0, 0, p+1, N1));
    y.block(0, N1, p+1, N2) = f2m(x.block(0, N1, p+1, N2));

    // solve interpolant
    Newton ipolf(grid, x);
    ipolf.Interpolate(y);

    // compute interpolation error
    double error = ipolf.EvalError(f);

    return error;
}
```

question_2/codes/2_sol.cpp

(f) (3 pts) Implement a C++ function to study the convergence of your interpolant $I^p_{\mathcal{T}_N}$ with respect to p, for $p = 1, 2, 3, \ldots, 20$. Explain the results from your convergence test.

Solution:

```
void convergenceWrtp()
{
    int l=0;

    std::cout << "\n\nEquidistant nodes, convergence with respect to p:" << std::endl
    ;
    for (int p=1; p<21; p++)
    {
        double error = piecewiseLagrange_equi(p, l);
        std::cout << p << "\t" << error << std::endl;
    }

    std::cout << "\n\nChebyshev nodes, convergence with respect to p:" << std::endl;
    for (int p=1; p<21; p++)
    {
        double error = piecewiseLagrange_cheb(p, l);
        std::cout << p << "\t" << error << std::endl;
    }
}
```

question_2/codes/2_sol.cpp

If equidistant nodes are used to interpolate the given function in the range $[0, 1/3]$, we do not observe convergence with respect to the local polynomial degree p because of the **Runge phoenomenon**. Whereas, when the Chebyshev points are used, the interpolation scheme converges with respect to p.

Exercise 3. *Singly Diagonally Implicit Runge-Kutta Methods* (26 pts) [*Template: 3.cpp*]

For general implicit Runge-Kutta methods, the k_i's cannot be evaluated successively since they are coupled in the system of implicit equations that is given for their determination. One way to decouple a non-linear system is to use diagonally implicit Runge-Kutta methods. A special family of those methods are Singly Diagonally Implicit Runge-Kutta methods (SDIRK), in which the matrix of the method is lower triangular and all the diagonal entries are equal. Continuing in this direction, we define the following one parameter family of SDIRK-methods:

$$\begin{array}{c|cc} \gamma & \gamma & 0 \\ 1-\gamma & 1-2\gamma & \gamma \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$$

Table I: Singly Diagonally Implicit Runge-Kutta

(a) (3 pts) Determine the value(s) of γ for which the corresponding Runge-Kutta method is of at least (consistency) order 3.

Hint: In addition to conditions necessary for a consistency order 2, the following conditions are necessary for a consistency order 3:

$$\sum_{i=1}^{s} b_i c_i^2 = \frac{1}{3} \qquad \text{and} \qquad \sum_{i=1}^{s} \sum_{j=1}^{s} b_i a_{ij} c_j = \frac{1}{6},$$

where $\{a_{ij}\}_{i,j=1}^{s}$, $\{b_i\}_{i=1}^{s}$, $\{c_j\}_{j=1}^{s}$ are entries of a standard Butcher tableau.

Solution: The following equations have to be satisfied to obtain a consistency order of at least 3:

$$\sum_{i=1}^{s} b_i = 1$$

$$\sum_{i=1}^{s} b_i c_i = \frac{1}{2}$$

$$\sum_{i=1}^{s} b_i c_i^2 = \frac{1}{3}$$

$$\sum_{i=1}^{s} \sum_{j=1}^{s} b_i a_{ij} c_j = \frac{1}{6}$$

Therefore, we have:

$$\frac{1}{2} + \frac{1}{2} = 1$$

$$\frac{1}{2}\gamma + \frac{1}{2}(1-\gamma) = \frac{1}{2}$$

$$\frac{1}{2}\gamma^2 + \frac{1}{2}(1-\gamma)^2 = \frac{1}{3}$$

$$\frac{1}{2}\gamma^2 + 0 + \frac{1}{2}\gamma(1-2\gamma) + \frac{1}{2}\gamma(1-\gamma) = \frac{1}{6}$$

Therefore, we have that the method will be of order 3 provided γ satisfies the equation

$$\gamma - \gamma^2 = \frac{1}{6}$$

that is, for,

$$\gamma = \frac{3 \pm \sqrt{3}}{6}$$

(b) (4 pts) Write down the iterations of the Runge-Kutta method defined in Table I, when applied to the ODE

$$\dot{y} = \lambda y, \qquad y(0) = 1, \qquad \lambda \in \mathbb{C}$$

HINT: Use the stability function and the fact that the ODE is linear.

Solution: Notice that we have

$$y_{n+1} = S(\lambda h) y_n$$

where $S(z)$ is the stability function of the Runge-Kutta method.

The stability function of any Runge-Kutta method is given by

$$S(z) := \frac{\det\left(\mathbf{I} - z\mathbf{A} + z\mathbf{1}\mathbf{b}^\top\right)}{\det\left(\mathbf{I} - z\mathbf{A}\right)}.$$

Therefore, setting $\mathbf{A}$ and $\mathbf{b}$ as given in table I, we have

$$\det\left(\mathbf{I} - z\mathbf{A}\right) = (1 - z\gamma)^2$$

and

$$\det\left(\mathbf{I} - z\mathbf{A} + z\mathbf{1}\mathbf{b}^\top\right) = 1 + z(1 - 2\gamma) + z^2\left(\frac{1}{2} - 2\gamma + \gamma^2\right)$$

So

$$y_{n+1} = \frac{1 + \lambda h(1 - 2\gamma) + (\lambda h)^2\left(\frac{1}{2} - 2\gamma + \gamma^2\right)}{(1 - \lambda h\gamma)^2} y_n$$

(c) (3 pts) For which value(s) of γ obtained from sub-problem (a) (the value(s) with which the method is of order 3), can we conjecture the corresponding Runge-Kutta method to be A-stable? Explain.

Hint: You may use the plotting script *'stabilityRegion_plot.cpp'* to visualize the region of stability for a chosen value of γ.

Solution: From fig. 2 and fig. 3, we can conjecture that the method is A-stable only for $\gamma = \frac{3+\sqrt{3}}{6}$ as it is stable in the entire left half of the complex plane ($|S(z)| < 1 \; \forall z \in \mathbb{C}^-$).

```
////TASK C
// This function computes |S(z)| for a chosen value of 'gamma'
double computeSMag(std::complex<double> z, double gamma){
    double Smag;
    std::complex<double> Stab_func;

    //To-do [task (c)]: Write stability function in terms of 'z' and gamma
    Stab_func = (1.0+z*(1.0-2.0*gamma)+z*z*(0.5-2.0*gamma+pow(gamma,2)))/(pow((1.0-z*
    gamma),2));

    Smag = std::abs(Stab_func);

    return Smag;
}
////END OF TASK C

```

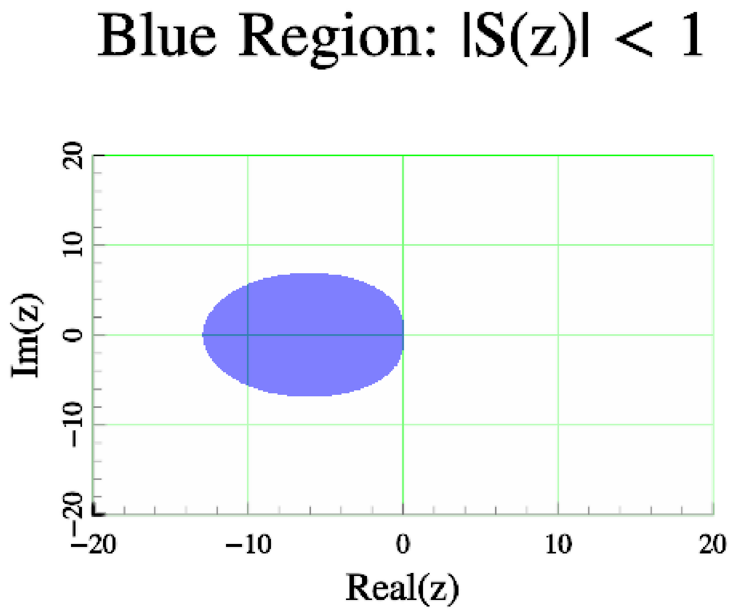


Figure 2: Stability region for $\gamma = \frac{3-\sqrt{3}}{6}$ (Blue is stable region)

```
int main() {
    //To-do [task (c)]: : Modify value for gamma in next line
    double gamma = (3.0-sqrt(3.0))/6.0;
    //double gamma = (3.0+sqrt(3.0))/6.0;
    //double gamma = 1.0;

    const std::complex<double> I(0.0, 1.0); //complex number ’iota’=sqrt(-1)
    double x_min = -20.0;
    double x_max = 20.0;
    double y_min = -20.0;
    double y_max = 20.0;
    int N_x = 1001;
    int N_y = 1001;
    double delta_x = (x_max - x_min)/(N_x-1);
    double delta_y = (y_max - y_min)/(N_y-1);
    VectorXd x_grid = VectorXd::LinSpaced(N_x, x_min, x_max);
    VectorXd y_grid = VectorXd::LinSpaced(N_y, y_min, y_max);

    // generate stability curve
    MatrixXcd z_grid(N_x,N_y);
    MatrixXd Smag_grid(N_x,N_y);
    MatrixXd Smag_grid_boolean(N_x,N_y);
    Smag_grid_boolean.setOnes();
    for(int k=0; k < N_y; ++k){
        for(int j=0; j < N_x; ++j){
            z_grid(j,k) = x_grid(j)+I*y_grid(k); // z = x + iota * y;
            Smag_grid(j,k) = computeSMag(z_grid(j,k),gamma); // Compute |S(z)| for given ’gamma’
            if (Smag_grid(j,k) >= 1.0) Smag_grid_boolean(j,k) = 0.0; //Create a boolean variable that takes value = 0.0 wherever |S(z)| >= 1 and takes value = 1.0 wherever |S(z)| < 1.
        }
    }

```

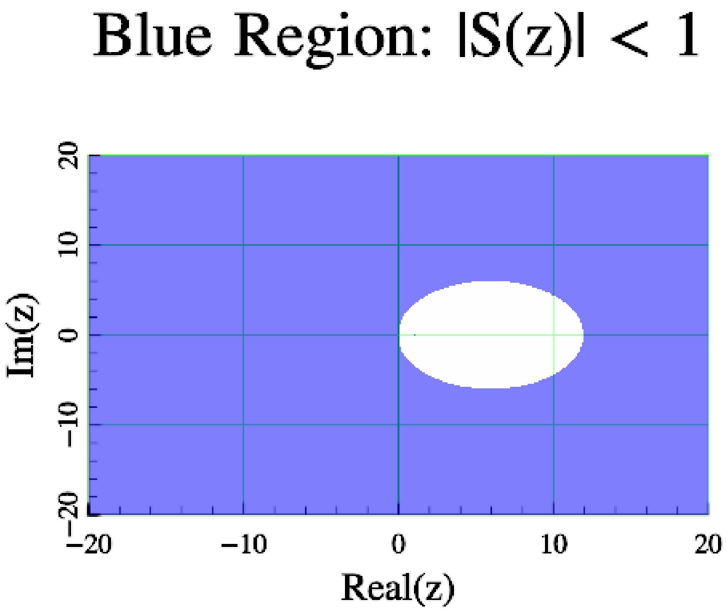


Figure 3: Stability region for $\gamma = \frac{3+\sqrt{3}}{6}$ (Blue is stable region)

```cpp
    // Plot stability function using MathGL (Blue region indicates region of |S(z)| < 1 )
    mglData x_dat(x_grid.data(), x_grid.size());
    mglData y_dat(y_grid.data(), y_grid.size());
    mglData dat(Smag_grid_boolean.rows(), Smag_grid_boolean.cols(), Smag_grid_boolean.data());
    mglGraph gr;
    gr.Title("Blue Region: |S(z)| < 1");
    gr.SetRanges(x_min,x_max,y_min,y_max);
    gr.Grid("xy","g");
    gr.Axis(); gr.Label('x',"Real(z)",0); gr.Label('y',"Im(z)",0);
    gr.Alpha(true);
    gr.SetRange('c',0.0,1.0);
    gr.Tile(x_dat,y_dat,dat,"wb");
    gr.WriteFrame("stabilityPlot1.png");
    //gr.WriteFrame("stabilityPlot2.png");

    return 0;
```

question_3/codes/stabilityRegion_plot_sol.cpp

(d) (2 pts) We consider a scalar linear initial value problem of second order

$$\ddot{y} + \dot{y} + y = 0, \qquad y(0) = 1, \quad \dot{y}(0) = 0 \tag{2}$$

that should be solved numerically using the SDIRK-method described in Table I.

Formulate (2) as an initial value problem (IVP) for a linear first order system.

Solution: Define $z_1 = y$, $z_2 = \dot{y}$, then the initial value problem

$$\ddot{y} + \dot{y} + y = 0, \qquad y(0) = 1, \quad \dot{y}(0) = 0 \tag{3}$$

is equivalent to the first order system

$$\begin{aligned} \dot{z}_1 &= z_2 \\ \dot{z}_2 &= -z_1 - z_2, \end{aligned} \tag{4}$$

with initial values $z_1(0) = 1$, $z_2(0) = 0$.

(e) (6 pts) Implement a `C++/Eigen` function

```
template <class StateType>
StateType sdirkStep(const StateType & z0, double h, double gamma);
```

that realizes the numerical evolution of one step of the SDIRK-method for the IVP obtained in sub-problem (d), starting from the initial value `z0` and returning the value after a time step of size h.

Hint: For an SDIRK-method, compute stage k_1 first and then use it to compute stage k_2.

Solution: Let

$$\mathbf{A} := \begin{pmatrix} 0 & 1 \\ -1 & -1 \end{pmatrix}.$$

Then

$$\begin{aligned} k_1 &= \mathbf{A}\mathbf{y}_0 + h\gamma\mathbf{A}k_1 \\ k_2 &= \mathbf{A}\mathbf{y}_0 + h(1-2\gamma)\mathbf{A}k_1 + h\mathbf{A}k_2, \end{aligned}$$

so

$$\begin{aligned} k_1 &= (\mathbf{I} - h\gamma\mathbf{A})^{-1}\mathbf{A}\mathbf{y}_0 \\ k_2 &= (\mathbf{I} - h\gamma\mathbf{A})^{-1}(\mathbf{A}\mathbf{y}_0 + h(1-2\gamma)\mathbf{A}k_1). \end{aligned}$$

```
template <class StateType>
StateType sdirkStep(const StateType & z0, double h, double gamma) {
    // Matrix A for evaluation of f
    Eigen::Matrix2d A;
    A << 0.,1.,-1.,-1.;

    // Reuse factorization
    auto A_lu = (Eigen::Matrix2d::Identity()-h*gamma*A).partialPivLu();
    Eigen::Vector2d az = A*z0;

    // Increments
    Eigen::Vector2d k1 = A_lu.solve(az);
    Eigen::Vector2d k2 = A_lu.solve(az + h*(1-2*gamma)*A*k1);

    // Next step
    StateType z1 = z0 + h*0.5*(k1 + k2);
    return z1;
}
```

question_3/codes/3_sol.cpp

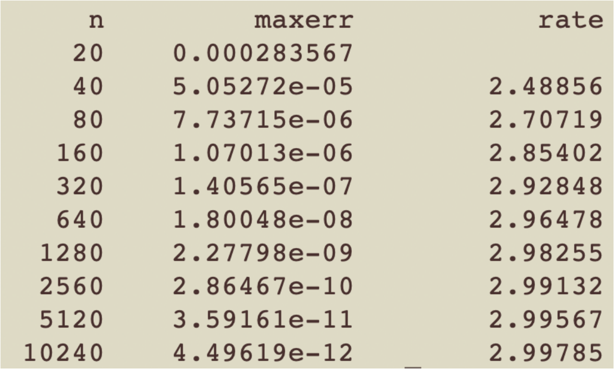
(f) (8 pts) Conduct a numerical experiment to deduce the convergence order of the method by computing the global error of the *A-stable* SDIRK method for the first order IVP from sub-problem (d) at the end time. Choose `T=10` as end time and `N=20,40,80,...,10240` as steps. Report the observed order of convergence.

Solution:

```
////TASK F
template <class StateType>
std::vector<StateType> sdirkSolve(const StateType & z0, unsigned int N, double T, double gamma) {
    // Solution vector
    std::vector<StateType> res(N+1);
    // Equidistant step size
    const double h = T/N;

    // Push initial data
    res.at(0) = z0;

    // Main loop
    for(unsigned int i = 1; i <= N; ++i) {
        res.at(i) = sdirkStep(res.at(i-1), h, gamma);
    }

    return res;
}
////END OF TASK F

int main() {
    // Initial data z0 = [y(0), y'(0)]
    Eigen::Vector2d z0;
    z0 << 1,0;
    // Final time
    const double T = 10;
    // TO-DO task (f): Modify value of parameter gamma such that method is A-stable
    const double gamma = (3.+std::sqrt(3.)) / 6.;
    // Mesh sizes
    std::vector<int> N = {20, 40, 80, 160, 320, 640, 1280, 2560, 5120, 10240};
    // Exact solution (only y(t)) given initial data  z0 and t
   auto yex = [&z0] (double t) {
        return 1./3.*std::exp(-t/2.) * ( 3.*z0(0) * std::cos( std::sqrt(3.)*t/2. ) +
        std::sqrt(3.)*z0(0) * std::sin( std::sqrt(3.)*t/2. ) +
        2.*std::sqrt(3.)*z0(1) * std::sin( std::sqrt(3.)*t/2. ) );
    };

    // Store old error for rate computation
    double errold = 0;
    std::cout << std::setw(15) << "n" << std::setw(15) << "maxerr" << std::setw(15) << "rate" << std::endl;
    // Loop over all meshes
    for(unsigned int i = 0; i < N.size(); ++i) {
        int n = N.at(i);
        // Get solution
        auto sol =  sdirkSolve(z0, n, T, gamma);
        // Compute error
        double err = std::abs(sol.back()(0) - yex(T));

        // I/O
        std::cout << std::setw(15) << n << std::setw(15) << err;
        if(i > 0) std::cout << std::setw(15) << std::log2(errold /err);
        std::cout << std::endl;

        // Store old error
        errold = err;
    }
}
```

question_3/codes/3_sol.cpp

As seen in fig. 4, the numerically estimated order of convergence is 3, which is consistent with the expected order from sub-problem (a).



```
    n         maxerr            rate
   20    0.000283567
   40    5.05272e-05         2.48856
   80    7.73715e-06         2.70719
  160    1.07013e-06         2.85402
  320    1.40565e-07         2.92848
  640    1.80048e-08         2.96478
 1280    2.27798e-09         2.98255
 2560    2.86467e-10         2.99132
 5120    3.59161e-11         2.99567
10240    4.49619e-12         2.99785
```

Figure 4: Observed rate of convergence for SDIRK method