KV Wissenschaftliches Rechnen, WS 2024/25

Hausübung 3

Einleitung Im git-Repository finden Sie die Klasse TridiagSparseMatrix<T> im Header `SCTridiagSparseMatrix.h`. In dieser Klasse ist eine Tridiagonalmatrix auf Basis der SparseMatrix<T> implementiert. Diese Klasse kann für diese Hausübung verwendet werden.

Es wird die Diskretisierung des Konvektions-Diffusionsproblems ähnlich der zweiten Hausübung betrachtet, $u : (0, 1) \to \mathbb{R}$ mit

$$-ku'' + bu' = f, \qquad\qquad u(0) = u(1) = 0, \tag{1}$$

mit Parametern

$$k = 10, \qquad b = 50, \qquad f(x) = \begin{cases} 0 & \text{falls } x < 0.5, \\ 1 & \text{falls } x \geq 0.5 \end{cases}. \tag{2}$$

Nach einer Diskretisierung über Finite Differenzen

$$-u''(x_i) \simeq \frac{-u_{i-1} + 2u_i - u_{i+1}}{h^2}, \qquad\qquad u'(x_i) \simeq \frac{u_i - u_{i-1}}{h}, \tag{3}$$

ergibt sich das tridiagonale Gleichungssystem $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ mit

$$a_{ii} = 2k/h^2 + b/h, \qquad a_{i\,i-1} = -k/h^2 - b/h \qquad a_{i-1\,i} = -k/h^2, \qquad b_i = f(x_i). \tag{4}$$

Abzugeben sind

- `SCTridiagLUSolver.h` und `SCGaussSeidelSolver.h` Files, enthalten Quellcode für Aufgabe 1 und 2,
- andere Header aus der SC-Bibliothek nur falls geändert, in diesem Fall Änderungen bzw Ergänzungen beschreiben,
- Quellcode mit `main`-Routine, in der die Aufgaben umgesetzt werden,
- Protokoll als `.pdf` File.

Aufgabe 1 (Klasse TridiagLUSolver) – 4 Punkte Die LU-Faktorisierung für Tridiagonalmatrizen lässt sich innerhalb der Tridiagonalstruktur umsetzen. In diesem Fall brauchen für $\mathbf{L}$ nur die Sub-Diagonale und für $\mathbf{U}$ die Diagonale und Super-Diagonale gespeichert werden. Das Verfahren zur LU-Faktorisierung vereinfacht sich stark und ist mit Aufwand $O(n)$ Operationen umsetzbar.

Erstellen Sie eine entsprechende Solver-Klasse TridiagLUSolver<T>, die von LinearOperator<T> abgeleitet ist

```
namespace SC {
    template <typename T = double>
    class TridiagLUSolver : public LinearOperator<T> {
    // ....
    };
}
```

Folgende Spezifikationen müssen erfüllt sein

1. **1 Punkt – benötigter Speicher** der Speicheraufwand für die Faktorisierung ist $O(n)$, zusätzlich wird eine Referenz **const** TridiagSparseMatrix<T> & als Member geführt,

2. **1 Punkt** – Default- und Copy-Construktor, Destruktor und Zuweisungsoperator müssen implementiert oder ausgeschlossen (= **delete**) sein, zusätzlich wird ein Konstruktor für gegebene Tridiagonalmatrix **const** TridiagSparseMatrix<T> & benötigt,

3. **1 Punkt** – **Faktorisierung** passiert z.B. im Konstruktor, muss $O(n)$ Operationen benötigen, OHNE Pivot-Suche (kein Fehler-Handling bei Division durch Null notwendig),

4. **1 Punkt** – **Anwendung** die Apply-Funktionalität muss überladen werden,

   ```
   void Apply(const Vector<T> &x, Vector<T> &r, T factor = 1.)
   const override
   ```

   der Gesamtaufwand muss $O(n)$ sein.

Dokumentieren Sie im Protokoll jedenfalls, wie die Faktorisierung gespeichert wird, wie die Faktorisierung berechnet wird, und wie die Apply-Funktionalität umgesetzt ist. *Hinweis: Testen Sie Ihre Implementierung an obigem Beispiel mit sehr kleinem n, etwa $n = 3 \ldots 5$. Diese Tests brauchen nicht protokolliert werden.*

Aufgabe 2 (Klasse TridiagGaussSeidelSolver) – 3 Punkte Setzen Sie das Gauss-Seidel-Verfahren für Tridiagonalmatrizen um. In diesem Fall braucht nur eine Referenz **const** TridiagSparseMatrix<T> & als Member-Variable gespeichert werden, zusätzlich zwei Variablen um die zu erreichende Genauigkeit **double** acc und die maximale Anzahl an Iterationen **int** maxit zu speichern. Das Gauss-Seidel-Verfahren vereinfacht sich insoweit dass ein Iterationsschritt mit Aufwand $O(n)$ Operationen umsetzbar ist. Das Verfahren soll abgebrochen werden, sobald das Residuum unter $|\boldsymbol{r}| <$ acc fällt.

Erstellen Sie eine entsprechende Solver-Klasse TridiagGaussSeidelSolver<T>, die von LinearOperator<T> abgeleitet ist

```
namespace SC {
    template <typename T = double>
    class TridiagGaussSeidelSolver : public LinearOperator<T> {
    // ....
    };
}
```

Folgende Spezifikationen müssen erfüllt sein

1. **1 Punkt** – Default- und Copy-Construktor, Destruktor und Zuweisungsoperator müssen implementiert oder ausgeschlossen (= **delete**) sein, zusätzlich wird ein Konstruktor für gegebene Tridiagonalmatrix **const** TridiagSparseMatrix<T> & a, Genauigkeit **double** acc=1e−6 und maximale Iterationszahl **int** maxit=1000000 benötigt, verwenden Sie die angegebenen Default-Werte,

2. **2 Punkte** – **Anwendung** die Apply-Funktionalität muss entsprechend dem Gauss-Seidel-Verfahren überladen werden,

   ```
   void Apply(const Vector<T> &x, Vector<T> &r, T factor = 1.)
   const override
   ```

   der Gesamtaufwand für eine Iteration muss $O(n)$ sein.

Dokumentieren Sie im Protokoll jedenfalls, wie die Apply-Funktionalität umgesetzt ist. *Hinweis: Testen Sie Ihre Implementierung an obigem Beispiel mit sehr kleinem n, etwa $n = 3 \ldots 5$. Diese Tests brauchen nicht protokolliert werden.*

Aufgabe 3 (Anwendungsbeispiel) – 3 Punkte Lösen Sie das diskretisierte Konvektions-Diffusionsproblem mit einer der oben beschriebenen Löser für $n = 100$ und visualisieren Sie das Ergebnis einmal ohne Konvektionsterm ($b = 0$) und einmal mit Konvektion ($b = 50$).

Dokumentieren Sie die Laufzeit in s für beide Verfahren beginnend für $n = 100$, und danach wenn n wiederholt verdoppelt wird. Stellen Sie Laufzeit über Anzahl der Unbekannten in einem doppelt logarithmischen Plot dar! Verdoppeln Sie n so lange, bis die Laufzeit über zwei bis drei Minuten betragen würde (**Release-Mode verwenden; Ausgabe während der Iteration unterbinden**, bei der LU-Faktorisierung kann die Anzahl der Unbekannten auch in jedem Schritt verzehnfacht werden); wählen Sie die maximale Iterationszahl im Gauss-Seidel-Löser entsprechend hoch.

Für $b = 0$ ist das System symmetrisch positiv definit, es kann das CG-Verfahren verwendet werden. Lösen Sie auch mit dem CGSolver<T>, wenn Sie die TridiagSparseMatrix<T> als Operator übergeben, und vergleichen Sie auch hier die Laufzeiten für $n = 100, 200, 400, \ldots$. Stellen Sie diesen Zusammenhang auch im doppelt logarithmischen Plot dar!

Punkteverteilung Aufgabe 3:

- **1 Punkt** – Lösen für $n = 100$ und Visualisierung der Lösung für $b = 0$ und $b = 50$.
- Vergleich und Darstellung der Laufzeiten über Anzahl der Unbekannten für die drei Verfahren bei $b = 0$ **1 Punkt**. Bei optimaler Implementierung sehen Sie (etwa) drei Geraden mit Steigungen $k = 1, 2, 3$ im loglog-Plot – dies entspricht der erwarteten Komplexität $O(n)$, $O(n^2)$ und $O(n^3)$. Verifizieren Sie dies und ordnen Sie die erwarteten Komplexitäten den drei Verfahren zu **1 Punkt**.