Praktikum

Parallele Programmierung

Wintersemester 2015/16

Scientific Computing / Angewandte Informatik
Bergische Universität Wuppertal

Prof. Dr. Andreas Frommer,
Dr. Karsten Kahl,
Prof. Dr. Bruno Lang

1 Voraussetzungen

Für die Bearbeitung der Aufgaben dieses Praktikums sollten Sie (mindestens) eine der Programmiersprachen C/C++ oder Fortran77/Fortran90/Fortran95 beherrschen. Gegebenenfalls sollten Sie sich diese Kenntnisse *jetzt* aneignen.
Der Besuch der Vorlesung Parallele Algorithmen wird nicht explizit vorausgesetzt, jedoch dürfte die Kenntnis der Vorlesungsinhalte die Bearbeitung der Aufgaben deutlich erleichtern. Es ist vielleicht hilfreich, die Mitschrift von Kommilitonen zu kopieren. Ein Skript zur Vorlesung finden Sie als „Skript AUD II“ auf der Home-Page des Praktikums

`http://www-ai.math.uni-wuppertal.de/SciComp/teaching/ss14/praktikum`

Die Aufgaben sollten in Zweiergruppen bearbeitet werden.
Für das Erreichen des Scheins sind die Bearbeitung

- des „einführenden“ Blocks (Aufgaben 1–3) sowie
- *zweier* der drei thematischen Blöcke
  - *Direkte Lösung linearer Systeme* (Aufgaben 4–5)
  - *Iterative Lösung linearer Systeme* (Aufgaben 6–7)
  - *Berechnung von Eigenwerten und -vektoren* (**zwei** der Aufgaben 8–10)

hinreichend. Dabei ist

- bei den Aufgaben 1, 2 und 4 nur eine Parallelisierung mit MPI,
- bei den übrigen Aufgaben (3, 5–10) sowohl eine Parallelisierung mit MPI als auch eine Parallelisierung mit OpenMP durchzuführen.

Die Abgabetermine für die bearbeiteten Aufgaben (Vorführung und Listings) werden in der Vorbesprechung bekanntgegeben.

2 Literatur

Die parallelen Programme sollen zum Einen mit Hilfe der MPI-Kommunikationsbibliothek (*Message Passing Interface*) erstellt werden. Als Einstieg in MPI könnte etwa die unter

`http://moss.csc.ncsu.edu/~mueller/cluster/mpi.guide.pdf`

erhältliche Kurzeinführung von Peter Pacheco dienen, evtl. ergänzt durch Material aus

`http://www-unix.mcs.anl.gov/mpi/tutorial/`

Darüber hinaus bietet

`http://www-unix.mcs.anl.gov/mpi/`

einen Einstieg in die „MPI-Welt“ mit vielen weiteren Dokumenten (insbesondere den Definitionen der Standards MPI-1.1 und MPI-2.0) und frei verfügbaren MPI-Implementierungen (u.a. für Linux). Dort sind auch Verweise auf die Lehrbücher von Gropp, Lusk und Skjellum bzw. Pacheco zu finden.
Es ist empfehlenswert, zumindest Pachecos Kurzeinführung durchzuarbeiten, bevor mit der MPI-Programmierung begonnen wird.

Für die Programmierung mit OpenMP sollte die unter

`http://www.openmp.org`

erhältliche Dokumentation (insbesondere der OpenMP-„Standard“ für die jeweils benutzte Programmiersprache) verwendet werden.

3 Allgemeine Hinweise

- Die Aufgaben müssen natürlich nicht in der angegebenen Reihenfolge gelöst werden. Es ist jedoch empfehlenswert, zumindest die einführenden Aufgaben 1–3 zuerst zu bearbeiten.
- Die nachfolgenden Beschreibungen (z.B. der Datenverteilungen etc.) beziehen sich auf die MPI-Parallelisierung. Bei OpenMP liegt ein *gemeinsamer* Speicher zugrunde, so dass die Daten nicht verteilt werden müssen. Dies führt zu einer gegenüber MPI deutlich vereinfachten Parallelisierung.
- Für die OpenMP-Parallelisierung gehen Sie am besten von einem *seriellen* Programm aus, in das dann geeignete OpenMP-Direktiven eingefügt werden.
- Führen Sie numerische Rechnungen grundsätzlich in *doppelter Genauigkeit* durch und speichern Sie die Daten (reelle Zahlen, Vektoren und Matrizen) in einem entsprechenden Format (`double` bei C/C++ bzw. `double precision` bei Fortran).

4 Block 0 — Einführende Aufgaben

Aufgabe 1 (1 Punkt)

Schreiben Sie ein Programm, welches auf jedem Prozess die Anzahl der aktiven Prozesse sowie die Nummer des eigenen Prozesses bestimmt und ausgibt.
Testen Sie Ihr Programm für $p = 1, 2, 4, 7$.
Auf $p = 4$ Prozessen sollte die Ausgabe etwa wie folgt aussehen:

```
Program running on  4 processes, I am process no.  0
Program running on  4 processes, I am process no.  3
Program running on  4 processes, I am process no.  1
Program running on  4 processes, I am process no.  2
```

Aufgabe 2 (2 Punkte)

Schreiben Sie ein Programm, welches

a) in Prozess 0 eine Zahl vom Benutzer einliest und

b) diese dann durch den Prozess*ring* an alle anderen Prozesse weiterreicht und dabei in jedem Prozess verdoppelt und ausgibt.

Testen Sie Ihr Programm für $p = 1, 2, 4, 7$.

Hinweis: Die Prozesse sind standardmäßig als Ring konfiguriert.

Auf $p = 4$ Prozessen sollte die Ausgabe etwa wie folgt aussehen:

```
Enter a number in [ 0, 1000 ] : 12.5
Process  0 got              12.5000 and computed              25.0000
Process  1 got              25.0000 and computed              50.0000
Process  3 got             100.0000 and computed             200.0000
Process  2 got              50.0000 and computed             100.0000
```

Aufgabe 3 (4 Punkte)

Implementieren Sie die Simpson-Regel mit n Teilintervallen zur Berechnung von

$$s \approx \int_a^b f(x)\,dx$$

und messen Sie die benötigte Zeit.
Testen Sie Ihr Programm für $[a, b] = [0, 2]$,

$$f(x) = x^{16}$$

und $n = 512, 1024, 65536$ und $p = 1, 4, 7$.

Beschreibung: Bei der Simpson-Regel wird der Integrationsbereich $[a, b]$ durch $n + 1$ äquidistante Punkte

$$x_i = a + ih \ , \quad i = 0, \ldots, n,$$

mit $h = (b - a)/n$ in n Intervalle $[x_i, x_{i+1}]$ mit zugehörigen Mittelpunkten

$$x_{i+\frac{1}{2}} = \frac{x_i + x_{i+1}}{2} \ , \quad i = 0, \ldots, n - 1,$$

zerlegt und das Integral durch die Summe

$$s = \frac{h}{6} \sum_{i=0}^{n-1} \Big(f(x_i) + 4f(x_{i+\frac{1}{2}}) + f(x_{i+1}) \Big)$$

approximiert.

Hinweis: Um eine aussagekräftige Laufzeit zu erhalten, sollten Sie vor Beginn der Zeitmessung alle Prozesse mit einer Barriere synchronisieren und am Ende das Maximum über die lokalen Zeiten bestimmen.

Auf $p = 4$ Prozessen sollte die Ausgabe etwa wie folgt aussehen:

```
Enter the interval bounds (a, b) : 0 2
Enter the number of subintervals (n) : 65536
Simpsons Rule for [     0.0000,     2.0000 ] with    65536 subintervals
   yielded the approximation  7710.117647058816
   and took     0.042057 seconds
```

5 Block 1 — Direkte Lösung linearer Systeme

Aufgabe 4 (6 Punkte)

Eine Matrix $A \in \mathbb{R}^{n \times n}$ heißt *spaltenzyklisch* auf die Prozesse $P_1, \ldots, P_p$ verteilt, falls P_i genau die Spalten j von A mit $j \equiv i \bmod p$ enthält.

a) Schreiben Sie eine Routine `DistributeColumns`, welche eine in Prozess P_i vollständig vorliegende Matrix $A \in \mathbb{R}^{m \times n}$ spaltenzyklisch auf die Prozesse verteilt.

b) Schreiben Sie eine Routine `CollectColumns`, welche eine spaltenzyklisch auf die Prozesse verteilte Matrix $A \in \mathbb{R}^{m \times n}$ in einem Prozess P_i rekonstruiert.

c) Schreiben Sie eine Routine `Transpose`, welche zu einer spaltenzyklisch auf die Prozesse verteilten Matrix $A \in \mathbb{R}^{m \times n}$ die spaltenzyklisch auf die Prozesse verteilte Transponierte A^T bestimmt, *ohne A oder A^T in einem einzigen Prozess aufzubauen.*

Es sollen jeweils *keine einzelnen Elemente* der Matrix verschickt, sondern so wenige Kommunikations*operationen* wie möglich verwendet werden.
Testen Sie Ihre Routinen, indem Sie etwa für verschiedene m, n und $p = 1, 3, 4$ Prozesse die Matrix

$$A = (a_{ij}) \in \mathbb{R}^{m \times n} \quad \text{mit } a_{ij} = i + \frac{j}{1000}$$

jeweils

i. lokal in einem Prozess P_i erzeugen,

ii. von dort aus mit `DistributeColumns` spaltenzyklisch verteilen,

iii. mit `Transpose` (verteilt) transponieren,

iv. die Matrix A^T mit `CollectColumns` wieder in P_i einsammeln und

v. dort mit A^T vergleichen.

Aufgabe 5 (10 Punkte)

Implementieren Sie einen parallelen Löser für lineare Gleichungssysteme der Gestalt

$$AX = B, \quad A \in \mathbb{R}^{n \times n}, B \in \mathbb{R}^{n \times q},$$

so dass mehrere Systeme

$$Ax^{(i)} = b^{(i)}, \quad A \in \mathbb{R}^{n \times n}, b^{(i)} \in \mathbb{R}^n \ (i = 1, \ldots, q),$$

mit gleicher Matrix A gelöst werden können.
Gehen Sie hierzu folgendermaßen vor.

a) Realisieren sie den Gauß-Algorithmus mit Spaltenpivotsuche. Transformieren Sie dabei zusätzlich die rechten Seiten B gleich mit, d.h. arbeiten Sie mit der Matrix $(\,A\,|\,B\,)$ statt A. Gehen Sie hierfür von der folgenden seriellen (kji-)Form des Gauß-Algorithmus aus:

```
für k = 1 : n
    { Spaltenpivotsuche }
    bestimme betragsgrößtes Element a_{p_k,k} in A(k : n, k)
    vertausche Zeilen k und p_k von A
    { Multiplikatoren bestimmen }
    für i = k + 1 : n
        a_ik := a_ik / a_kk
    für j = k + 1 : n + q
        { (−a_kj)-faches der Multiplikatorenspalte zu Spalte j addieren }
        für i = k + 1 : n
            a_ij := a_ij − a_ik a_kj
```

Verwenden Sie eine spaltenzyklische Speicherung der Daten.

b) Im obigen Gauß-Algorithmus wird $(\,A\,|\,B\,)$ überschrieben durch

$$(\,U\,|\,\widetilde{B}\,) = (\,L^{-1}P^TA\,|\,L^{-1}P^TB\,)$$

mit $P^TA = LU$. Dabei ist P eine durch die Vertauschungen bestimmte Permutationsmatrix, L eine untere und U eine obere Dreiecksmatrix.

Realisieren Sie den jetzt noch notwendigen Dreieckslöser $UX = \widetilde{B}$.

Gehen Sie hierfür von der folgenden seriellen (ij-)Form aus:

```
für i = n : −1 : 1
    für k = 1 : q
        s^(k) := 0
    für j = i + 1 : n
        für k = 1 : q
            s^(k) := s^(k) + a_ij x_j^(k)
    für k = 1 : q
        x_i^(k) := (a_{i,n+k} − s^(k)) / a_ii
```

c) Für $A \in \mathbb{R}^{m\times n}$ ist die *Frobenius-Norm* $\|A\|_F$ definiert durch

$$\|A\|_F := \left(\sum_{i=1}^{m}\sum_{j=1}^{n} a_{ij}^2\right)^{1/2}.$$

Berechnen Sie jeweils $\|LU - P^TA\|_F$ und $\|\widetilde{B} - UX\|_F$. Verwenden Sie hierzu ggf. die Routine `CollectColumns` aus Aufgabe 4.

Testen Sie Ihr Verfahren anhand der Matrizen

$$A = (a_{ij}) \in \mathbb{R}^{n \times n} \quad \text{mit} \quad a_{ij} = \frac{i+j}{n} \cdot \sin \frac{ij\pi}{n+1}$$
$$B = (b_{ij}) \in \mathbb{R}^{n \times q} \quad \text{mit} \quad b_{ij} = i + j$$

für $n = 100, 500$ und $q = 1, 10$ auf $p = 1, 3, 4$ Prozessen.

6 Block 2 — Iterative Lösung linearer Systeme

Aufgabe 6 (8 Punkte)

Implementieren Sie das Red-Black-SSOR-Verfahren mit Conrad-Wallach-Trick zum Lösen der diskretisierten Laplace-Gleichung

$$-u_{i,j-1} - u_{i-1,j} + 4u_{ij} - u_{i+1,j} - u_{i,j+1} = h^2 f_{ij} \ , \quad i,j = 1:N, \tag{1}$$

mit den Randbedingungen

$$\begin{aligned} u_{0j} &= g_{0j}, \quad u_{N+1,j} = g_{N+1,j}, \quad j = 1:N, \\ u_{i0} &= g_{i0}, \quad u_{i,N+1} = g_{i,N+1}, \quad i = 1:N. \end{aligned}$$

Dabei sei $N \in \mathbb{N}$, $h = 1/(N+1)$ und

$$u_{ij} = u(ih, jh),\ f_{ij} = f(ih, jh),\ g_{ij} = g(ih, jh), \quad i,j = 0:N+1.$$

Beschreibung: Ordnen Sie den Gitterpunkten (ih, jh) des äquidistanten Gitters

$$\Omega_h = \{(ih, jh) \ : \ i,j = 0:N+1\}$$

die Farben Rot und Schwarz zu durch

$$\begin{aligned} (ih, jh) &\quad \text{ist rot, falls} \quad i+j \equiv 0 \bmod 2, \\ (ih, jh) &\quad \text{ist schwarz, falls} \quad i+j \equiv 1 \bmod 2. \end{aligned}$$

Ein Iterationsschritt des Red-Black-SSOR-Verfahrens besitzt die Gestalt:

für alle roten Gitterpunkte

$$u_{ij}^{k+\frac{1}{2}} := \frac{\omega}{4}\left(h^2 f_{ij} + u_{i,j-1}^{k} + u_{i-1,j}^{k} + u_{i+1,j}^{k} + u_{i,j+1}^{k}\right) + (1-\omega)u_{ij}^{k} \tag{2}$$

für alle schwarzen Gitterpunkte

$$u_{ij}^{k+\frac{1}{2}} := \frac{\omega}{4}\left(h^2 f_{ij} + u_{i,j-1}^{k+\frac{1}{2}} + u_{i-1,j}^{k+\frac{1}{2}} + u_{i+1,j}^{k+\frac{1}{2}} + u_{i,j+1}^{k+\frac{1}{2}}\right) + (1-\omega)u_{ij}^{k} \tag{3}$$

für alle schwarzen Gitterpunkte

$$u_{ij}^{k+1} := \frac{\omega}{4}\left(h^2 f_{ij} + u_{i,j-1}^{k+\frac{1}{2}} + u_{i-1,j}^{k+\frac{1}{2}} + u_{i+1,j}^{k+\frac{1}{2}} + u_{i,j+1}^{k+\frac{1}{2}}\right) + (1-\omega)u_{ij}^{k+\frac{1}{2}} \tag{4}$$

für alle roten Gitterpunkte

$$u_{ij}^{k+1} := \frac{\omega}{4}\left(h^2 f_{ij} + u_{i,j-1}^{k+1} + u_{i-1,j}^{k+1} + u_{i+1,j}^{k+1} + u_{i,j+1}^{k+1}\right) + (1-\omega)u_{ij}^{k+\frac{1}{2}} \tag{5}$$

Die Aufdatierungen der roten Gitterpunkte sind voneinander unabhängig und deshalb parallel ausführbar, da diese nur von Werten in den jeweils benachbarten schwarzen Gitterpunkten und dem alten Wert im zugehörigen roten Gitterpunkt abhängen. Entsprechendes gilt für die Aufdatierung der schwarzen Gitterpunkte.

Hinweise: Beachten Sie, dass sowohl Größen aus (3) in (4) als auch Größen aus (5) in (2) (für den nächsten Iterationsschritt $k+1$) verwendet werden können (*Conrad-Wallach-Trick*).

Verwenden Sie eine *zweidimensionale* Aufteilung von Ω_h, so dass jedes Teilgebiet von Ω_h ein Rechteck von $\ell \times m$ Gitterpunkten darstellt, und ordnen Sie jedem der $p = P \cdot Q$ Prozesse $P_{\ell m}$, $\ell = 1 : P$, $m = 1 : Q$, ein Teilgebiet zu. Benutzen Sie lokal ein zweidimensionales Feld $u(0 : \ell + 1, 0 : m + 1)$ für das Teilgebiet und dessen Rand der Breite 1.

Konstruieren Sie einen geeigneten Datentyp zum Austausch der Werte in den roten und schwarzen Randpunkten. Zur Vereinfachung können Sie ℓ und m als gerade annehmen. Realisieren Sie die Kommunikation mit `MPI_CART_SHIFT`.

Brechen sie die Iteration ab, wenn sich zwei aufeinander folgende Iterierte in der Euklid-Norm ($\|v\|_2 = \left(\sum_{i=1}^{n} v_i^2\right)^{1/2}$ für $v \in \mathbb{R}^n$) um weniger als ε unterscheiden.

Testen Sie Ihr Verfahren für $N = 24, 48, 120$, $P = 2$, $Q = 3$, $\omega = 1$, $f \equiv 0$,

$$g(x,y) = \begin{cases} 1-x & \text{für} \quad y = 0 \\ x & \text{für} \quad y = 1 \\ 1-y & \text{für} \quad x = 0 \\ y & \text{für} \quad x = 1 \end{cases}$$

und den Startvektor $u^0 = 0$. Die exakte Lösung ist die Funktion $u(x,y) = 2xy - x - y + 1$. Geben Sie am Ende die Euklidnorm des Fehlervektors $e := (u_{ij})_{i,j=1:N} - (u(ih, jh))_{i,j=1:N} \in \mathbb{R}^{N^2}$ aus.

Aufgabe 7 (8 Punkte)

Implementieren Sie das parallele cg-Verfahren zur Lösung des in Aufgabe 6 beschriebenen linearen Gleichungssystems (1), der sog. diskretisierten Laplace-Gleichung.
Vergleichen Sie die Laufzeit des Verfahrens *ohne Präkonditionierung* mit der des *Red-Black-SSOR-präkonditionierten* Verfahrens.

Beschreibung: Das cg-Verfahren kann zur Lösung linearer Gleichungssysteme $Ax = b$ mit symmetrischer, positiv definiter Matrix $A \in \mathbb{R}^{n \times n}$ (d.h. $y^T Ay > 0$ für alle $0 \neq y \in \mathbb{R}^n$) eingesetzt werden. Eine der Standard-Formulierungen des Verfahrens ist nachfolgend angegeben:

{ Startvektor }
wähle $x^0 \in \mathbb{R}^n$ beliebig
{ Anfangsresiduum }
$r^0 := b - Ax^0$
$\rho_0 := \|r^0\|_2^2$
{ erste „Suchrichtung“ }
$s^0 := r^0$
für $k = 0, 1, \ldots,$ bis Abbruchkriterium erfüllt ist
    $q^k := As^k$
    $\alpha_k := \rho_k \,/\, (\,(s^k)^T q^k\,)$
    { neue Iterierte }
    $x^{k+1} := x^k + \alpha_k s^k$
    { „billiges“ Aufdatieren des Residuums $r^{k+1} = b - Ax^{k+1}$ }
    $r^{k+1} := r^k - \alpha_k q^k$
    $\rho_{k+1} := \|r^{k+1}\|_2^2$
    { neue Suchrichtung }
    $\beta_k := -\rho_{k+1}/\rho_k$
    $s^{k+1} := r^{k+1} - \beta_k s^k$

Das cg-Verfahren liefert bei rundungsfehlerfreier Rechnung nach spätestens n Schritten die exakte Lösung x^* des Gleichungssystems. In der Praxis wird das Verfahren allerdings nicht in dieser Weise verwendet, sondern als iteratives Verfahren, das nach (i.a. wesentlich) weniger als n Schritten abgebrochen wird, sobald eine hinreichend gute Näherung für die Lösung bestimmt ist.

Die Konvergenzgeschwindigkeit dieses iterativen Verfahrens hängt stark von der Verteilung der Eigenwerte von A ab. Man kann zeigen, dass in der Norm $\|x\|_A := (x^T Ax)^{1/2}$ für die Fehler der Iterierten $x^k \in \mathbb{R}^n$ gilt:

$$\|x^k - x^*\|_A \leq \mu^k \cdot 2\|x^0 - x^*\|_A \ ,$$

wobei

$$\mu = \frac{\kappa(A) - 1}{\kappa(A) + 1} < 1$$

ist mit

$$\kappa(A) = \frac{\lambda_{\max}(A)}{\lambda_{\min}(A)} \qquad (\textit{Kondition} \text{ von } A)$$

($\lambda_{\max}$ größter, $\lambda_{\min}$ kleinster Eigenwert von A). Diese Kondition kann bereits für kleine Matrizen sehr groß sein; dann liegt μ sehr nahe bei 1, und das cg-Verfahren konvergiert sehr langsam.

Aus diesem Grunde geht man meist vom System $Ax = b$ auf ein „äquivalentes“ System $By = c$ mit besser konditionierter Matrix B über, etwa, indem man mit einer nichtsingulären Matrix S^{-1} durchmultipliziert:

$$\underbrace{S^{-1}A(S^T)^{-1}}_{B} \cdot \underbrace{S^T x}_{y} = \underbrace{S^{-1}b}_{c} \ .$$

Gelingt es, S so zu wählen, dass $T := SS^T \approx A$ gilt, so konvergiert das cg-Verfahren für das „*präkonditionierte*“ System $By = c$ deutlich schneller als für das ursprüngliche System $Ax = b$. Für die praktische Anwendbarkeit ist es wesentlich, dass dieses präkonditionierte cg-Verfahren auch durchgeführt werden kann, *ohne die Matrix B explizit zu berechnen* (siehe folgenden Algorithmus).

{ Startvektor }
wähle $x^0 \in \mathbb{R}^n$ beliebig
{ Anfangsresiduum }
$r^0 := b - Ax^0$
löse das System $Tz^0 = r^0$
{ erste „Suchrichtung“ }
$\rho_0 := (r^0)^T z^0$ { $= \|\hat{r}^0\|_2^2$ mit $\hat{r}^0 = c - By^0$, wobei $y^0 = S^T x^0$ }
$s^0 := z^0$
für $k = 0, 1, \ldots,$ bis Abbruchkriterium erfüllt ist
    $q^k := As^k$
    $\alpha_k := \rho_k \,/\, ((s^k)^T q^k)$
    { neue Iterierte }
    $x^{k+1} := x^k + \alpha_k s^k$
    { „billiges“ Aufdatieren des Residuums $r^{k+1} = b - Ax^{k+1}$ }
    $r^{k+1} := r^k - \alpha_k q^k$
    löse das System $Tz^{k+1} = r^{k+1}$
    $\rho_{k+1} := (r^{k+1})^T z^{k+1}$ { $= \|\hat{r}^{k=1}\|_2^2$ mit $\hat{r}^{k+1} = c - By^{k+1}$, $y^{k+1} = S^T x^{k+1}$ }
    { neue Suchrichtung }
    $\beta_k := -\rho_{k+1}/\rho_k$
    $s^{k+1} := z^{k+1} - \beta_k s^k$

Bei dieser Formulierung erhält man auch statt einer Näherung für die Lösung y^* von $By = c$ direkt eine Näherung für die (eigentlich interessierende) Lösung x^* von $Ax = b$. Der einzige Mehraufwand im Vergleich zum nicht präkonditionierten Verfahren besteht darin, dass *in jedem cg-Schritt* ein linearen Gleichungssystem mit der Matrix T gelöst werden muss.

Bei der Red-Black-SSOR-Präkonditionierung wählt man die Matrix S so, dass das Lösen des Systems $Tz = r$ genau der Durchführung von $\nu \geq 1$ Schritten des Red-Black-SSOR-Verfahrens für das System $Az = r$ mit dem Startvektor $z^{(0)} = 0$ entspricht.

Hinweise: Das im cg-Verfahren benötigte Matrix-Vektor-Produkt $q = As$ ergibt sich für die diskretisierte Laplace-Gleichung mit der (für zweidimensionale äquidistante Diskretisierungen üblichen) zweidimensionalen Indizierung der Vektoren durch

$$q_{ij} := -s_{i,j-1} - s_{i-1,j} + 4s_{ij} - s_{i+1,j} - s_{i,j+1} \ , \quad i,j = 1 : N \ .$$

Verwenden Sie zur Parallelisierung die in Aufgabe 6 vorgeschlagene zweidimensionale Aufteilung des Gitters Ω_h in rechteckige Gebiete.

Testen Sie Ihr Verfahren mit dem in Aufgabe 6 angegebenen Problem. Bestimmen Sie, welche Anzahl ν von Red-Black-SSOR-Schritten als Präkonditionierer für dieses Problem „vernünftig“ ist.

7 Block 3 — Berechnung von Eigenwerten und -vektoren

Aufgabe 8 (10 Punkte)

Implementieren Sie eine parallele Variante des zyklischen Jacobi-Verfahrens zur Bestimmung der Eigenwerte *und Eigenvektoren* einer symmetrischen Matrix $A \in \mathbb{R}^{n\times n}$ (n gerade) auf $p = n/2$ Prozessen.

Beschreibung: Beim Jacobi-Verfahren wird auf die Matrix A eine Folge von *Jacobi-Rotationen*

$$A \equiv A^{(0)} \longrightarrow J_1^T A^{(0)} J_1 =: A^{(1)} \longrightarrow J_2^T A^{(1)} J_2 =: A^{(2)} \longrightarrow \cdots$$

angewandt. Dabei ist

$$J_k \equiv J(p_k, q_k, \theta_k) := \begin{pmatrix} 1 & & & & & & & & & & \\ & \ddots & & & & & & & & & \\ & & 1 & & & & & & & & \\ & & & c_k & 0 & \cdots & 0 & s_k & & & \\ & & & 0 & 1 & & & 0 & & & \\ & & & \vdots & & \ddots & & \vdots & & & \\ & & & 0 & & & 1 & 0 & & & \\ & & & -s_k & 0 & \cdots & 0 & c_k & & & \\ & & & & & & & & 1 & & \\ & & & & & & & & & \ddots & \\ & & & & & & & & & & 1 \end{pmatrix} \in \mathbb{R}^{n\times n}$$

(der Eintrag s_k befindet sich an der (p_k, q_k)-Position in J_k) eine Rotation um den Winkel θ_k in der (p_k, q_k)-Ebene mit $c_k = \cos\theta_k$ und $s_k = \sin\theta_k$.

Bei geeigneter Wahl des Winkels θ_k kann man durch die Anwendung der Rotation

$$A^{(k-1)} \longrightarrow J(p_k, q_k, \theta_k)^T A^{(k-1)} J(p_k, q_k, \theta_k) = A^{(k)}$$

den Eintrag in der (p_k, q_k)-Position von A zu Null machen. Wählt man etwa für $a_{p_k,q_k} \neq 0$ speziell den Winkel θ_k so, dass

$$t_k \equiv \tan\theta_k = \frac{\operatorname{sign}(\tau)}{|\tau| + \sqrt{1+\tau^2}} \qquad \text{mit} \qquad \tau := \frac{a_{q_k q_k}^{(k-1)} - a_{p_k p_k}^{(k-1)}}{2a_{p_k q_k}^{(k-1)}} \ , \tag{6}$$

so gilt $a_{p_k q_k}^{(k)} = 0$. (Zur Anwendung der Rotation wird der Winkel θ_k wegen

$$c_k = \frac{1}{\sqrt{1+t_k^2}} \quad \text{und} \quad s_k = c_k \cdot t_k \tag{7}$$

nicht explizit benötigt.) Im Fall $a_{p_k q_k}^{(k-1)} = 0$ wird man natürlich $\theta_k = 0$ wählen, also keine „echte“ Rotation durchführen.

Ein *Sweep* des zyklischen Jacobi-Verfahrens besteht nun darin, in geeigneter Reihenfolge jedes der *Außen*diagonalelemente von A genau einmal zu Null zu machen. Wegen der (während des Verfahrens erhaltenen) Symmetrie wird dabei mit $a_{p_k q_k}$ auch gleichzeitig $a_{q_k p_k}$ zu Null, so dass $n(n-1)/2$ Rotationen pro Sweep genügen. (Es gibt mehrere Varianten des zyklischen Jacobi-Verfahrens, die sich hauptsächlich in der Wahl der Reihenfolge für die Indexpaare (p_k, q_k) unterscheiden.) Die Erfahrung zeigt (und für einige zyklische Varianten kann die Konvergenz auch bewiesen werden), dass die Matrix A nach wenigen solcher Sweeps (typisch: höchstens 10) „praktisch“ Diagonalgestalt besitzt mit sehr guten Approximationen λ_i der Eigenwerte an den Diagonalpositionen a_{ii}.

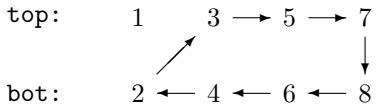
Parallelität wird durch gleichzeitige Bestimmung und Anwendung von Rotationen erzielt. Rotationen $J(p_{k_i}, q_{k_i}, \theta_{k_i})$ sind vertauschbar (und damit in jeder Reihenfolge anwendbar, insbesondere parallel), wenn ihre Indexpaare $\{p_{k_i}, q_{k_i}\}$ paarweise disjunkt sind. Das folgende Vorgehen liefert $n-1$ Mengen von jeweils $n/2$ Indexpaaren, so dass

- die zu den Indexpaaren jeder Menge gehörigen Rotationen parallel anwendbar sind und
- die $n-1$ Mengen zusammen genau einen Sweep ergeben (d.h. jedes Außendiagonalelement wird genau einmal zu Null gemacht); insbesondere tritt kein Indexpaar doppelt auf.

Die erste Menge ist durch die Indexpaare $(1,2)$, $(3,4)$, $\ldots$, $(n-1,n)$ gegeben, siehe nachfolgende Skizze für $n = 8$.

`top:`	1	3	5	7
`bot:`	2	4	6	8

Für die zweite Menge verschiebt man alle Indizes (außer der 1) um einen Platz im Uhrzeigersinn



und erhält

`top:`	1	2	3	5
`bot:`	4	6	8	7

,

entsprechend den Indexpaaren $(1,4)$, $(2,6)$, etc. Die nächste „Verschiebe-Runde“ ergibt

`top:`	1	4	2	3
`bot:`	6	8	7	5

mit den Indexpaaren $(1,6)$, $(4,8)$, usw. Nach $n-1$ solchen Runden ist wieder die Ausgangssituation hergestellt.

Mit dieser Reihenfolge führt man Sweeps (jeweils $n-1$ Runden) mit der Matrix A durch, bis die Außendiagonal-„Norm“

$$\|A\|_{\text{off}} := \left(\sum_{i=1}^{n} \sum_{j \neq i} a_{ij}^2 \right)^{1/2}$$

eine vorgegebene Schranke ε unterschreitet.

Die Eigenvektoren $q^{(i)}$ erhält man, indem man die Rotationen gleichzeitig in einer orthogonalen Matrix Q akkumuliert:

$$I_n \equiv Q^{(0)} \longrightarrow Q^{(0)} J_1 =: Q^{(1)} \longrightarrow Q^{(1)} J_2 =: Q^{(2)} \longrightarrow \cdots$$

(I_n: Einheitsmatrix). Wenn A „hinreichend diagonal“ ist, bilden die Spalten $q^{(i)}$ von Q gute Näherungen für Eigenvektoren zu den λ_i.

Zusammen ergibt sich der folgende (zunächst serielle) Algorithmus für ein zyklisches Jacobi-Verfahren:

```
Q := I_n
top := (1, 3, ..., n-1)
bot := (2, 4, ..., n)
solange ||A||_off ≥ ε
    { durchlaufe die Paare-Mengen des Sweeps }
    für s = 1 : n-1
        { durchlaufe die Indexpaare der Menge }
        für k = 1 : n/2
            { Rotation bestimmen }
            (p_k, q_k) := (top(k), bot(k))
            berechne t_k aus a_{q_k q_k}, a_{p_k p_k} und a_{p_k q_k} gemäß (6)
            berechne c_k und s_k gemäß (7)
            { Rotation anwenden }
            A := A · J(p_k, q_k, θ_k)
            A := J(p_k, q_k, θ_k)^T · A
            Q := Q · J(p_k, q_k, θ_k)
        verschiebe( top, bot )
```

Dabei bezeichnet $\texttt{top}(1:\frac{n}{2})$ die „obere“ und $\texttt{bot}(1:\frac{n}{2})$ die „untere“ Hälfte der Indizes in der „aktuellen“ Runde (siehe Skizzen), und `verschiebe` führt die zum Übergang in die nächste Runde notwenigen Indexverschiebungen durch.

Parallelität wird durch die gleichzeitige Bestimmung und Anwendung der Rotationen jeder Menge erzielt. Hierzu wird die Matrix A spaltenweise auf die Prozesse aufgeteilt, wie durch die aktuellen Indexpaare angegeben. In der ersten Runde jedes Sweeps besitzt also Prozess 1 die Spalten 1 und 2, Prozess 2 die Spalten 3 und 4, usw., in der zweiten Runde besitzt Prozess 1 die Spalten 1 und 4, Prozess 2 die Spalten 2 und 6, usw. Dies bedeutet, dass zur Anwendung der $n/2$ Rotationen einer Menge

- jeder Prozess die „seinem“ Indexpaar der Menge entsprechende Rotation bestimmen und von rechts her auf die Matrix A anwenden kann und hierfür nur lokale Daten benötigt,
- jeder Prozess *alle* Rotationen der Menge von links her auf seine beiden Spalten von A anwenden muss und
- die Spalten von A zwischen den Runden in gleicher Weise wie die Indizes zwischen den Prozessoren verschoben werden müssen.

Die Akkumulation von Q kann ebenfalls parallel erfolgen, wenn man die Spalten der Matrix Q ebenso wie die Spalten von A verteilt und mit jeder Runde entsprechend verschiebt. Hier wird die Parallelisierung dadurch erleichtert, dass Q nur von rechts her transformiert werden muss.

Hinweis: Anwenden einer Rotation von links, $A \longrightarrow J(p,q,\theta)^T \cdot A$, ändert nur die Zeilen p und q von A. Analog werden bei Anwendung von rechts, $A \longrightarrow A \cdot J(p,q,\theta)$, nur die Spalten p und q modifiziert. Nutzt man dies aus, dann erfordert die Anwendung der Rotation (von links oder rechts) $4n$ Multiplikationen und $2n$ Additionen. Daher wird die Anwendung einer Rotation *nie als Matrix-Matrix-Produkt* realisiert, was rund $2n^3$ Operationen erfordern würde.

Zur Überprüfung der Ergebnisse lassen Sie das *Residuum*

$$\|AQ - Q\Lambda\|_F$$

sowie die Abweichung

$$\|Q^TQ - I\|_F$$

der Eigenvektoren von der Orthogonalität ausgeben. Dabei ist $\Lambda = \mathrm{diag}(\lambda_1, \ldots, \lambda_n) \in \mathbb{R}^{n\times n}$ (λ_i: Näherungen für die Eigenwerte von A) und $Q = (\, q^{(1)} \mid \ldots \mid q^{(n)} \,) \in \mathbb{R}^{n\times n}$ ($q^{(i)}$: Näherungen zugehöriger orthonormierter Eigenvektoren).
Testen Sie das Verfahren mit der Matrix

$$A = \begin{pmatrix} n & n-1 & n-2 & \cdots & 1 \\ n-1 & n-1 & n-2 & \cdots & 1 \\ n-2 & n-2 & n-2 & \cdots & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & 1 & \cdots & 1 \end{pmatrix} \in \mathbb{R}^{n\times n} \qquad (n = 2p)$$

und $\varepsilon = 10^{-10}$ auf $p = 1, 3, 4, 8$ Prozessen.
Für $p = 4$ (also $n = 8$) besitzt diese Matrix die Eigenwerte

```
 0.258735930272134
 0.287520089775684
 0.345844044326706
 0.457762962433225
 0.688385684834675
 1.258287827212875
 3.338165566772758
29.365297894371945
```

und ein zum Eigenwert 0.258735930272134 gehöriger Eigenvektor ist durch

```
 0.0891316083075325
-0.2553571073253758
 0.3870952140163506
-0.4665539670857883
 0.4830020216355103
-0.4342179767567612
 0.3267903880321401
-0.1752279465957299
```

gegeben.

Aufgabe 9 (10 Punkte)

Implementieren Sie ein zweistufiges paralleles Verfahren zur Berechnung aller Eigenwerte einer symmetrischen Matrix $A = (a_{ij}) \in \mathbb{R}^{n \times n}$. Verwenden Sie hierfür eine spaltenzyklische Abspeicherung des *unteren Dreiecks* von A.

a) Transformieren Sie in der ersten Stufe die Matrix A mit Hilfe des Householder-Verfahrens auf Tridiagonalgestalt.

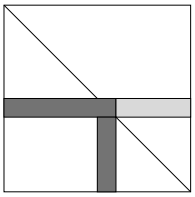
Beschreibung: Gehen Sie von der folgenden seriellen Form des Householder-Verfahrens aus:

für $k = 1 : n-2$
  $x := A(k+1:n, k) \in \mathbb{R}^{n-k}$
  wenn $x \neq 0$
    $\xi := \begin{cases} \text{sign}(x_1)\|x\|_2 & \text{für} \quad x_1 \neq 0 \\ \|x\|_2 & \text{für} \quad x_1 = 0 \end{cases}$
    $y := (x_1 + \xi, x_2, \ldots, x_{n-k})^T$
    $\tau := 2/\|y\|_2^2$
    $A(k+1:n, k) := (-\xi, 0, \ldots, 0)^T$
    $v := \tau A(k+1:n, k+1:n)y$
    $z := v - \frac{\tau}{2}(y^T v)y$
    für $j = k+1 : n$
      für $i = j : n$
        $a_{ij} := a_{ij} - y_{i-k}z_{j-k} - z_{i-k}y_{j-k}$

Dabei wird für festes k gerade die k-te Spalte bzw. Zeile von A „tridiagonalisiert“. Die Schleife über j ist eine symmetrische Rang-2-Modifikation, bei der nur das *untere Dreieck* von $A - yz^T - zy^T$ berechnet wird.

Hinweis: Beachten Sie, dass für ein paralleles Householder-Verfahren mit spaltenzyklischer Abspeicherung des unteren Dreiecks von A ein Algorithmus zur Matrix-Vektor-Multiplikation

$$v \quad = \quad \tau \cdot \left(\quad \right) \cdot y$$



verwendet werden muss, der nur das untere Dreieck von A benötigt.

b) Die Eigenwerte einer symmetrischen Tridiagonalmatrix

$$T = \begin{pmatrix} a_1 & b_2 & & \\ b_2 & a_2 & \ddots & \\ & \ddots & \ddots & b_n \\ & & b_n & a_n \end{pmatrix} \in \mathbb{R}^{n \times n}$$

mit $b_i \neq 0$ für $i = 2 : n$ können wie folgt durch ein Bisektionsverfahren bestimmt werden (zweite Stufe der Berechnungen).

Beschreibung: Definiert man zu $\mu \in \mathbb{R}$ die Größen

$$\begin{aligned} p_0(\mu) &:= 1 \\ p_1(\mu) &:= a_1 - \mu \\ p_i(\mu) &:= (a_i - \mu)p_{i-1}(\mu) - b_i^2 p_{i-2}(\mu) \ , \quad i = 2 : n, \end{aligned}$$

so ist die Anzahl $s(\mu)$ der Vorzeichenwechsel in der Folge $p_0(\mu), \ldots, p_n(\mu)$ (Nullen werden ignoriert) gerade die Anzahl der Eigenwerte λ_j von T, welche kleiner als μ sind.

Das Bisektionsverfahren startet mit einem Intervall $[a, b]$, welches mit Sicherheit alle Eigenwerte enthält (also $s(a) = 0$, $s(b) = n$). Dieses unterteilt man in der Mitte $c = (a+b)/2$ und bestimmt $s(c)$. Ist $s(a) < s(c)$, so enthält $[a, c]$ Eigenwerte und wird auf die gleiche Weise (rekursiv) behandelt. Ist $s(c) < s(b)$, so wird $[c, b]$ weiter betrachtet. Die Rekursion wird abgebrochen, wenn der Durchmesser des Intervalls $[a, b]$ kleiner als ε ist.

```
Bisect( [a,b], s_a, s_b )
{ bestimmt alle Eigenwerte im Intervall [a,b]; es ist s_a = s(a), s_b = s(b) }
wenn b − a < ε
    dann ist (a + b)/2 eine Näherung für s_b − s_a Eigenwerte
sonst
    c := (a + b)/2
    s_c := s(c)
    wenn s_c > s_a
        Bisect( [a,c], s_a, s_c )
    wenn s_c < s_b
        Bisect( [c,b], s_c, s_b )
```

Ein geeignetes, alle Eigenwerte enthaltendes, Startintervall ist zum Beispiel durch $[-\|T\|_\infty, +\|T\|_\infty]$ gegeben mit

$$\|A\|_\infty := \max_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}| \qquad \text{für} \quad A = (a_{ij}) \in \mathbb{R}^{m\times n} \ .$$

Implementieren Sie eine parallele Variante des Bisektionsverfahrens, bei der ein Prozess (der master)

- eine Liste der noch zu bearbeitenden Intervalle verwaltet,
- den übrigen Prozessen (den slaves) jeweils ein noch zu bearbeitendes Intervall zuschickt und
- von jedem Slave die noch weiter zu bearbeitende(n) Hälfte(n) des ihm übergebenen Intervalls zurück erhält

und alle anderen Prozesse (die slaves)

- jeweils ein noch zu bearbeitendes Intervall vom master empfangen,
- durch Berechnung von $s(c)$ die noch weiter zu bearbeitende(n) Hälfte(n) dieses Intervalls bestimmen und
- diese dann wieder an den master senden.

Hinweis: Die Berechnung der Folge $p_0(\mu), \ldots, p_n(\mu)$ kann leicht zu Überlauf führen. Dies läßt sich vermeiden, indem man statt dessen die Folge $q_i(\mu) := p_i(\mu)/p_{i-1}(\mu)$ verwendet, welche der Rekursionsgleichung

$$\begin{aligned} q_1(\mu) &:= a_1 - \mu \\ q_i(\mu) &:= a_i - \mu - \frac{b_i^2}{q_{i-1}(\mu)} \ , \quad i = 2:n, \end{aligned}$$

genügt. Dabei sind Glieder $q_i(\mu) = 0$ durch sehr kleine positive Zahlen zu ersetzen, um Division durch Null auszuschließen. Dann erhält man $s(\mu)$ als Anzahl der negativen Glieder in der Folge $q_1(\mu), \ldots, q_n(\mu)$.

Testen Sie Ihre Verfahren anhand der Matrizen

$$A = \begin{pmatrix} n & n-1 & n-2 & \cdots & 1 \\ n-1 & n-1 & n-2 & \cdots & 1 \\ n-2 & n-2 & n-2 & \cdots & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & 1 & \cdots & 1 \end{pmatrix} \in \mathbb{R}^{n\times n}$$

für $n = 50, 100$ auf $p = 1, 3, 4$ Prozessen mit $\varepsilon = 10^{-12}$.
Die zweite Stufe alleine kann mit den Tridiagonalmatrizen

$$T = \begin{pmatrix} 2 & 1 & & & \\ 1 & 2 & 1 & & \\ & 1 & 2 & \ddots & \\ & & \ddots & \ddots & 1 \\ & & & 1 & 2 \end{pmatrix} \in \mathbb{R}^{n\times n}$$

mit den Eigenwerten

$$\lambda_k = 4\sin^2 \frac{k\pi}{2(n+1)}\ , \quad k = 1 : n,$$

und

$$T = \begin{pmatrix} 10 & 1 & & & & & & & & & \\ 1 & 9 & 1 & & & & & & & & \\ & 1 & 8 & \ddots & & & & & & & \\ & & \ddots & \ddots & 1 & & & & & & \\ & & & 1 & 1 & 1 & & & & & \\ & & & & 1 & 0 & 1 & & & & \\ & & & & & 1 & 1 & 1 & & & \\ & & & & & & 1 & \ddots & \ddots & & \\ & & & & & & & \ddots & 8 & 1 & \\ & & & & & & & & 1 & 9 & 1 \\ & & & & & & & & & 1 & 10 \end{pmatrix} \in \mathbb{R}^{21\times 21}$$

mit Eigenwerten

```
-1.12544152211998
 0.25380581709668
 0.94753436752929
 1.78932135269508
 2.13020921936251
 2.96105888418573
 3.04309929257883
 3.99604820138363
 4.00435402344086
 4.99978247774291
 5.00024442500192
```

```
 6.00021752225710
 6.00023403158417
 7.00395179861638
 7.00395220952868
 8.03894111581427
 8.03894112282902
 9.21067864730492
 9.21067864736133
10.74619418290332
10.74619418290339
```

getestet werden.

Aufgabe 10 (10 Punkte)

Implementieren Sie das Lanczos-Verfahren zur Bestimmung der extremalen Eigenwerte *und Eigenvektoren* einer symmetrischen, dünn besetzten Matrix $A \in \mathbb{R}^{n\times n}$ auf p Prozessen.

Beschreibung: Das Lanczos-Verfahren erweitert, ausgehend von einem normierten Startvektor v^1, die vorhandene Orthonormalbasis (ONB) $\{v^1, \ldots, v^k\}$ des Krylov-Unterraums $K_k(A, v^1) = \text{span}(v^1, Av^1, \ldots, A^{k-1}v^1)$ zu einer ONB $\{v^1, \ldots, v^{k+1}\}$ von $K_{k+1}(A, v^1)$. Die algorithmische Formulierung

für $k = 1, 2, \ldots$, bis das Abbruchkriterium erfüllt ist

$$
\begin{aligned}
u^k &= Av^k \\
\alpha_k &= \langle u^k, v^k \rangle \\
\widetilde{v}^{k+1} &= q^k - \alpha_k v^k - \beta_k v^{k-1} \qquad (\text{mit } \beta_1 = 0,\ v^0 = 0) \\
\beta_{k+1} &= \|\widetilde{v}^{k+1}\|_2 \\
v^{k+1} &= \widetilde{v}^{k+1}/\beta_{k+1}
\end{aligned}
$$

lässt sich mit Hilfe der Matrizen $V_k = \left[v^1|v^2|\ldots|v^k\right]$ und der symmetrischen Tridiagonalmatrizen

$$
T_k = \begin{pmatrix}
\alpha_1 & \beta_2 & & & \\
\beta_2 & \alpha_2 & \beta_3 & & \\
 & \ddots & \ddots & \ddots & \\
 & & \beta_{k-1} & \alpha_{k-1} & \beta_k \\
 & & & \beta_k & \alpha_k
\end{pmatrix}
$$

zusammenfassen zu

$$
AV_k = V_k T_k + \beta_{k+1} \cdot v^{k+1} \cdot e_{k+1}^T \quad \text{mit} \quad e_{k+1}^T = (\underbrace{0, \ldots, 0}_{k\text{-mal}}, 1).
$$

Die Eigenwerte $\lambda_1, \ldots, \lambda_k$ von T_k sind Approximationen an die Eigenwerte von A. Ist $Q_k = (q_{ij}) \in \mathbb{R}^{k\times k}$ die Matrix, deren Spalten die zugehörigen Eigenvektoren von T_k

darstellen, so sind die Spalten w^j von $W_k = V_k \cdot Q_k$ approximative Eigenvektoren von A, und es gilt

$$\|Aw^j - \lambda_j w^j\|_2 \leq \beta_{k+1} \cdot |q_{kj}|. \tag{8}$$

Parallelität wird im Wesentlichen durch die Parallelisierung der Schritte im Lanczos-Verfahren erzielt. Sie sollen deshalb alle Berechnungen von Vektoren sowie die Matrix–Vektor-Multiplikation parallel durchführen.

Die Bestimmung der Eigenwerte und -vektoren von T_k soll seriell erfolgen, entweder durch eigenhändige Programmierung des Jacobi-Verfahrens (vgl. Aufg. 8) oder unter Verwendung einer einzubindenden Standard-Routine, etwa `DSTEV` aus LAPACK.

Die Bestimmung der approximativen Eigenvektoren von A soll dann wieder parallel erfolgen.

Geben Sie für jedes k die jeweils konvergierten Eigenwerte an, d.h. die Eigenwerte, in welchen die rechte Seite in (8) kleiner als eine vorgegeben Schranke ϵ ist. Berechnen Sie zur Kontrolle jeweils auch die linke Seite von (8).

Das Lanczos-Verfahren wird abgebrochen, wenn eine zuvor festgelegte Anzahl m von approximativen Eigenpaaren (λ_j, w^j) konvergiert ist, d.h. wenn die durch (8) gegebene Schranke für diese Paare kleiner als ϵ ist.

Hinweis: Bei dieser Aufgabe kommt es ganz wesentlich darauf an, wie die Matrix–Vektor-Multiplikation mit A realisiert wird. Für die vorliegende Aufgabe ist es ausreichend, wenn Sie für A die Matrix der diskreten Laplace-Gleichung auf dem Einheitsquadrat verwenden (s. Aufgabe 6).