Ludwig-Maximilians-Universität München
Mathematisches Institut
Gregor Kleen

Programmieren II für Studierende der Mathematik

Blatt 12 – Lösungsvorschlag

Aufgabe 13 Wir bezeichnen im Folgenden die i-te Zeile einer Matrix M mit m^i, die j-te Spalte einer Matrix M mit m_j und den j-ten Eintrag der i-ten Zeile einer Matrix M mit m^i_j.

Das zyklische Jacobi-Verfahren dient zur Berechnung der Eigenwerte und Eigenvektoren einer symmetrischen Matrix $A = \left(a^i_j\right)_{i,j=0,\dots,n-1} \in \mathbb{R}^{n\times n}$. Wir beobachten zunächst, dass A als symmetrischer Matrix ähnlich ist zu einer Diagonalmatrix, d.h. es existiert U mit $A' = U^\top AU$ diagonal. Aus U und A' lassen sich die Eigenwerte bzw. die Eigenvektoren der Matrix A ablesen.

Es ist U das Produkt einer Folge von Rotationsmatrizen Q, die sich von der Einheitsmatrix jeweils in den folgenden vier Einträgen unterscheiden:

$$\begin{aligned} q^i_i = q^j_j = c \\ q^i_j = -q^j_i = s \end{aligned}$$

Ansatz des zyklischen Jacobi Verfahrens ist es nun die Matrix A sukzessive den Transformationen $A^{\text{neu}} = Q^\top A^{\text{alt}} Q$ zu unterwerfen und hierbei Buch zu führen über $U^{\text{neu}} = U^{\text{alt}} Q$. c und s werden für jedes Q hierbei geeignet gewählt sodass ein $\left(a^{\text{alt}}\right)^i_j \neq 0$ auf $\left(a^{\text{neu}}\right)^i_j = 0$ gesetzt wird. Ist $\left(a^{\text{alt}}\right)^i_j = 0$ bereits der Fall, so wählen wir $c = 1$ und $s = 0$. Mit dem Ansatz $c = st$ und den Nebenbedingungen $c^2 + s^2 = 1$, $c > 0$ und $s \in (-c, c]$ erhält man:

$$\begin{aligned} \tau &= \frac{\left(a^{\text{alt}}\right)^j_j - \left(a^{\text{alt}}\right)^i_i}{2\left(a^{\text{alt}}\right)^i_j} \\ t &= \frac{\operatorname{sign}(\tau)}{|\tau| + \sqrt{1+\tau^2}} \\ c &= \frac{1}{\sqrt{1+t^2}} \\ s &= ct \end{aligned}$$

Bei der Multiplikation von A^{alt} mit Q werden jeweils nur die i-te Zeile und die j-te Spalte verändert, wie folgt:

$$\begin{aligned} a'_i &= ca^{\text{alt}}_i - sa^{\text{alt}}_j \\ a'_j &= sa^{\text{alt}}_i + ca^{\text{alt}}_j \\ (a'')^i &= c(a')^i - s(a')^j \\ (a'')^j &= s(a')^i + c(a')^j \\ \left(a^{\text{neu}}\right)^k_l &= \begin{cases} (a')^k_l & l \in \{i,j\}, k \notin \{i,j\} \\ (a'')^k_l & k \in \{i,j\} \\ \left(a^{\text{alt}}\right)^k_l & \text{sonst} \end{cases} \qquad k = 0,1,\dots \quad l = 0,1,\dots \end{aligned}$$

Ebenso für U:

$$u_i^{\text{neu}} = cu_i^{\text{alt}} - su_j^{\text{alt}}$$
$$u_j^{\text{neu}} = su_i^{\text{alt}} + cu_j^{\text{alt}}$$

In einem *Zyklus* sollen zeilenweise die Indexpaare (i, j) der oberen rechten Hälfte von A druchlaufen werden, d.h. $j = i + 1, \ldots, n - 1$ mit $i = 0, \ldots, n - 1$.

Die Berechnung soll abgebrochen werden, sobald eine vorgegebene Anzahl von Zykeln $\text{cyc}_{\max} = 50$ überschritten wird, oder gilt:

$$N\left(A^{\text{neu}}\right) := 2\sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1}\left|\left(a^{\text{neu}}\right)_j^i\right|^2 \leq \varepsilon \cdot N(A)$$

Es gilt:

$$N\left(A^{\text{neu}}\right) = N\left(A^{\text{alt}}\right) - 2\left|\left(a^{\text{alt}}\right)_j^i\right|^2$$

Nach Abschluss des Verfahrens lässt sich eine Näherung für die Eigenwerte in der Diagonale von A^{neu} und Näherungen für die Eigenvektoren in den Spalten von U^{neu} ablesen.

Erstellen Sie eine Funktion `jacobi` die das zyklische Jacobi-Verfahren durchführt, wie beschrieben. Rückgabewert der Funktion soll sein, ob das Verfahren erfolgreich war.

Rechnen Sie die folgenden Beispiele:

a) $A = \begin{pmatrix} 5 & 4 & 1 & 1 \\ 4 & 5 & 1 & 1 \\ 1 & 1 & 4 & 2 \\ 1 & 1 & 2 & 4 \end{pmatrix}$

b) $B = \begin{pmatrix} 5 & 1 & -2 & 0 & -2 & 5 \\ 1 & 6 & -3 & 2 & 0 & 6 \\ -2 & -3 & 8 & -5 & -6 & 0 \\ 0 & 2 & -5 & 5 & 1 & -2 \\ -2 & 0 & -6 & 1 & 6 & -3 \\ 5 & 6 & 0 & -2 & -3 & 8 \end{pmatrix}$

jacobi.cpp

```cpp
#include <valarray>
#include <algorithm>
#include <iostream>
#include <iomanip>

using namespace std;

template<class T> class matrix {
  private:
    valarray<T> v;
    size_t m, n;
```

```
  public:
    matrix(size_t m_ = 0, size_t n_ = 0, T val_ = T())
      : v(val_, m_ * n_),m(m_),n(n_) {}

    size_t nrows() const { return m; }
    size_t ncolumns() const { return n; }

    matrix<T>& operator=(T x) { v = x; return *this; }

    class matrix_slice : public slice {
      friend class matrix;
      private:
        valarray<T>& v;

        matrix_slice(valarray<T>& v_, size_t start, size_t size, size_t stride)
          : slice(start, size, stride), v(v_) {}

      public:
        T& operator[](size_t i)
          { return v[start() + i*stride()]; }

        matrix_slice& operator=(const matrix_slice&) = delete;
        matrix_slice& operator=(valarray<T> w) {
          v[static_cast<slice>(*this)] = w;
          return *this;
        }
        matrix_slice& operator=(T x) {
          v[static_cast<slice>(*this)] = x;
          return *this;
        }

        operator valarray<T>() {
          return v[static_cast<slice>(*this)];
        }
    };

    matrix_slice operator[](size_t i) { return row(i); }
    matrix_slice row(size_t i)
      { return matrix_slice{v, i*n, n, 1}; }
    matrix_slice column(size_t j)
      { return matrix_slice{v, j, m, n}; }

    matrix_slice diag()
      { return matrix_slice{v, 0, min(m, n), n + 1}; }

    auto begin() { return std::begin(v); }
    auto end() { return std::end(v); }

    friend istream& operator>>(istream& stream, matrix<T>& a) {
      for (T& x: a)
        stream >> x;
      return stream;
    }
};

using Vektor = valarray<double>;
```

```cpp
using Matrix = matrix<double>;

bool jacobi(Matrix a, Vektor& ew, Matrix& ev, int maxcyc = 50, double eps = 1e-12) {
  size_t n{a.nrows()};
  double sum_ndq{0};

  for (size_t i = 0; i < n; i++)
    for (size_t j = i + 1; j < n; j++)
      sum_ndq += a[i][j] * a[i][j];
  sum_ndq *= 2;

  double sum_ndq_start{sum_ndq};

  ev = 0;
  ev.diag() = 1;

  bool success = false;
  for (int cyc = 0; cyc < maxcyc; cyc++) {
    for (size_t i = 0; i < n; i++)
      for (size_t j = i + 1; j < n; j++) {
        double c, s, t, tau;
        if (abs(a[i][j]) <= eps) {
          c = 1;
          s = 0;
        } else {
          tau = (a[j][j] - a[i][i]) / (2 * a[i][j]);
          t = (tau >= 0 ? 1 : -1) / (abs(tau) + sqrt(1 + tau * tau));
          c = 1 / sqrt(1 + t * t);
          s = c * t;
        }

        sum_ndq -= 2 * a[i][j] * a[i][j];

        Vektor temp_i{a.column(i)};
        Vektor temp_j{a.column(j)};
        a.column(i) = c * temp_i - s * temp_j;
        a.column(j) = s * temp_i + c * temp_j;

        temp_i = a.row(i);
        temp_j = a.row(j);
        a.row(i) = c * temp_i - s * temp_j;
        a.row(j) = s * temp_i + c * temp_j;

        temp_i = ev.column(i);
        temp_j = ev.column(j);
        ev.column(i) = c * temp_i - s * temp_j;
        ev.column(j) = s * temp_i + c * temp_j;
      }

    if (sum_ndq <= eps * sum_ndq_start) {
      success = true;
      break;
    }
  }

  ew = a.diag();
  return success;
```

```
}

int main() {
  size_t n;
  cout << "n: "; cin >> n;

  Matrix a(n, n);
  cout << "Matrix: " << endl;
  cin >> a;

  Matrix ev(n, n);
  Vektor ew(n);
  if (jacobi(a, ew, ev))
    cout << "Jacobi-Verfahren erfolgreich" << endl;
  else
    cout << "Jacobi-Verfahren nicht erfolgreich" << endl;

  cout << "Eigenwerte:" << endl;
  for (double x: ew)
    cout << "  " << setprecision(4) << setw(8) << fixed << x << endl;
  cout << endl;

  cout << "Eigenvektoren:" << endl;
  for (size_t i = 0; i < n; i++) {
    cout << "  ";
    for (size_t j = 0; j < n; j++)
      cout << setprecision(4) << setw(8) << fixed << ev[i][j];
    cout << endl;
  }
}
```

```
n: 4
Matrix:
5  4  1  1
4  5  1  1
1  1  4  2
1  1  2  4
Jacobi-Verfahren erfolgreich
Eigenwerte:
    1.0000
   10.0000
    5.0000
    2.0000

Eigenvektoren:
    0.7071  0.6325 -0.3162  0.0000
   -0.7071  0.6325 -0.3162  0.0000
    0.0000  0.3162  0.6325 -0.7071
    0.0000  0.3162  0.6325  0.7071
```

```
n: 6
Matrix:
 5  1 -2  0 -2  5
 1  6 -3  2  0  6
```

```
-2 -3  8 -5 -6  0
 0  2 -5  5  1 -2
-2  0 -6  1  6 -3
 5  6  0 -2 -3  8
Jacobi-Verfahren erfolgreich
Eigenwerte:
   -1.5987
   -1.5987
   16.1427
    4.4560
    4.4560
   16.1427

Eigenvektoren:
   -0.2001  0.4691 -0.0854 -0.6209 -0.4161  0.4171
   -0.4610  0.3273 -0.3143  0.5528  0.2775  0.4460
    0.2632  0.5924  0.7216  0.2349  0.0600  0.0173
    0.4691  0.2001 -0.4171  0.4161 -0.6209 -0.0854
    0.3273  0.4610 -0.4460 -0.2775  0.5528 -0.3143
    0.5924 -0.2632 -0.0173 -0.0600  0.2349  0.7216
```