HMMatrix

Version: 2017-08-01

Vektoren und Matrizen in C und C++

Tilman Küpper (tilman.kuepper@hm.edu)

Inhalt

1. Einleitung

Die Funktions- und Klassenbibliothek HMMatrix dient zur Arbeit mit Vektoren und quadratischen Matrizen in den Programmiersprachen C und C++. Für beide Sprachen werden eigene Schnittstellen (Header-Dateien) zur Verfügung gestellt. Grundlegende Operationen wie die Addition und Multiplikation von Vektoren und Matrizen sind mit HMMatrix möglich, ebenso wie QR- und LU-Zerlegungen. Zur Ein- und Ausgabe auf dem Bildschirm und in Dateiform existieren verschiedene Funktionen.

In den neueren Versionen von HMMatrix wurden Funktionen zur Berechnung von Eigenwerten, Eigenvektoren und Polynomen hinzugefügt: Die Berechnung von Eigenwerten und Eigenvektoren geschieht über das zyklische Jakobiverfahren (für symmetrische Matrizen) oder die von-Mises-Vektoriteration, siehe Anhang B für weitere Details. Funktionswerte und Nullstellen von Polynomen können nun berechnet werden, charakteristische Polynome von Matrizen ebenfalls. Weitere Funktionen dienen zur Lösung linearer Gleichungssysteme. Multiplikationen größerer Matrizen können bei Bedarf auf mehrere Prozessorkerne verteilt und so oft deutlich beschleunigt werden, siehe Anhang C.

Bei C-Projekten sind die beiden Dateien **matrix.c** und **matrix.h** dem Projekt hinzuzufügen, bei C++-Projekten ist zusätzlich die Datei **matrix.hpp** erforderlich. Die folgende Abbildung zeigt Microsoft Visual Studio 2017 mit einem geöffneten C++-Projekt. Auf der linken Seite ist der Projektmappen-Explorer sichtbar, wo die genannten Dateien aufgelistet sind.

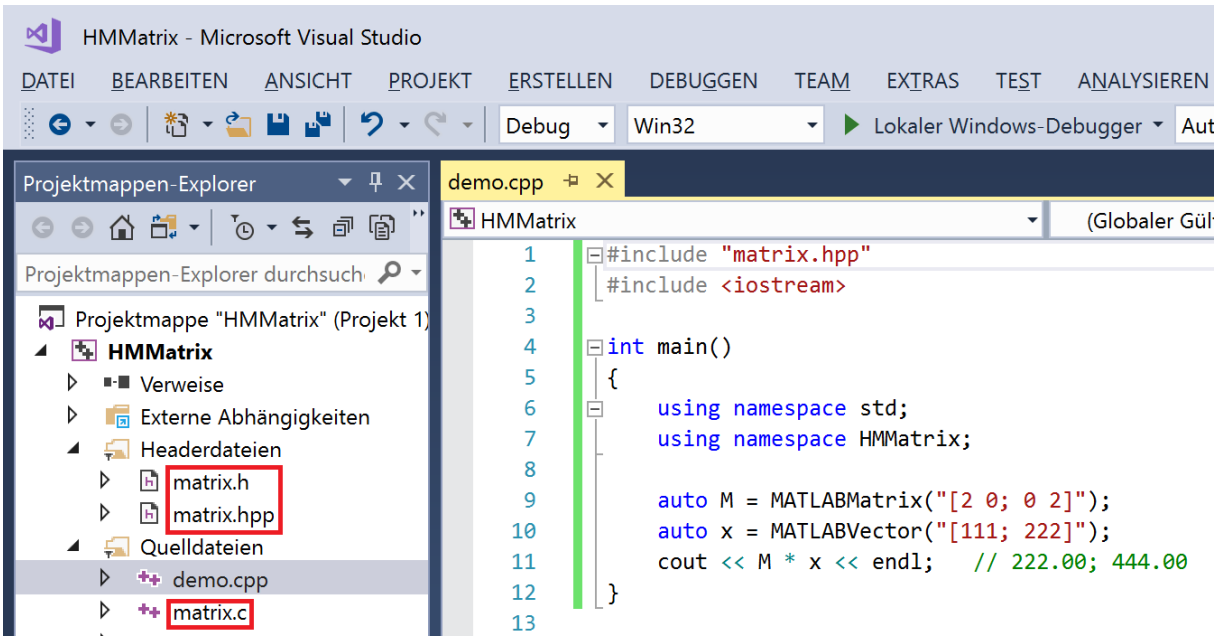


Abbildung 1 – Benutzeroberfläche von Microsoft Visual Studio 2017

2. Programmierung in C

2.1. Beispiele

```
/* ------------------------------------------------------------ */
/* Programmbeispiel: Vektoren & Matrizen                        */
/* ------------------------------------------------------------ */
#include "matrix.h"
#define DIM 5

int main(void)
{
    double vect[DIM], prod[DIM];            /* Vektoren definieren */
    double matr[DIM][DIM];                  /* Matrix definieren */

    v_init(DIM, vect, MAT_INIT_RANDOM);     /* Vektor mit Zufallszahlen (0…1) füllen */
    m_init(DIM, matr, MAT_INIT_IDENTITY);   /* Einheitsmatrix erzeugen */
    m_fmul(DIM, matr, 2.0);                 /* Alle Matrixelemente verdoppeln */
    m_vmul(DIM, prod, matr, vect);          /* Matrix mit Vektor multiplizieren */

    /* Matrix und Vektoren auf Konsole ausgeben */
    printf("matr = \n"); m_print(DIM, matr); printf("\n");
    printf("vect = \n"); v_print(DIM, vect); printf("\n");
    printf("prod = \n"); v_print(DIM, prod); printf("\n");

    /* Matrix invertieren und auf Konsole ausgeben */
    printf("Inverse Matrix = \n");
    m_invert(DIM, matr);
    m_print(DIM, matr);

    return 0;
}
```

```
/* ------------------------------------------------------------ */
/* Programmbeispiel: lineares Gleichungssystem                  */
/* ------------------------------------------------------------ */
#include "matrix.hpp"
#include <stdio.h>
#define DIM 3

int main(void)
{
    double x[DIM];
    double y[DIM] = {280, 340, 190};        /* Vektor y mit Werten belegen */
    double a[DIM][DIM] =                    /* Matrix A mit Werten belegen */
    {
        {2, 4, 6},
        {3, 5, 7},
        {4, 6, 1}
    };

    if(MAT_OK == lin_solve(DIM, x, a, y))   /* Gleichungssystem A x = y lösen */
        v_print(DIM, x);                    /* Ausgabe: 10.00; 20.00; 30.00 */
    else
        printf("Gleichungssystem kann nicht geloest werden!\n");
    return 0;
}
```

2.2. Funktionsübersicht

Funktionen zur Arbeit mit Matrizen

- `void m_init(size_t dim, void *mat, int type);`
  Die Matrix "mat" wird initialisiert. Für "type" sind folgende Werte möglich: MAT_INIT_NULL, MAT_INIT_IDENTITY, MAT_INIT_RANDOM, MAT_INIT_HILBERT, MAT_INIT_INV_HILBERT.

- `void m_fill(size_t dim, void *mat, double value);`
  Alle Elemente der Matrix "mat" werden auf den angegebenen Wert gesetzt.

- `void m_print(size_t dim, const void *mat);`
  Die Elemente der Matrix "mat" werden auf dem Bildschirm ausgegeben. Das Format kann über die Konstante MAT_PRN_FMT eingestellt werden.

- `void m_copy(size_t dim, void *mresult, const void *mat);`
  Der Inhalt der Matrix "mat" wird nach "mresult" kopiert.

- `int m_equal(size_t dim, const void *mat1, const void *mat2);`
  Falls "mat1" und "mat2" identisch sind, wird MAT_OK zurückgegeben, sonst MAT_ERR_NOTEQ.

- `void m_mul(size_t dim, void *mresult, const void *mat1, const void *mat2);`
  Das Produkt der Matrizen "mat1" und "mat2" wird in der Matrix "mresult" abgelegt.

- `void m_fmul(size_t dim, void *mat, double f);`
  Alle Elemente der Matrix "mat" werden mit dem Faktor "f" multipliziert.

- `void m_add(size_t dim, void *mresult, const void *mat1, const void *mat2);`
  "mat1" und "mat2" werden addiert. Das Ergebnis wird in der Matrix "mresult" abgelegt.

- `void m_fadd(size_t dim, void *mresult, const void *mat, double f);`
  Das "f-fache" der Matrix "mat" wird zur Matrix "mresult" addiert.

- `void m_swap_row(size_t dim, void *mat, size_t row1, size_t row2);`
  Die Zeilen "row1" und "row2" der Matrix "mat" werden vertauscht.

- `void m_mul_row(size_t dim, void *mat, size_t row, double factor);`
  Alle Elemente der Zeile "row" werden mit einem Faktor multipliziert.

- `void m_mul_col(size_t dim, void *mat, size_t col, double factor);`
  Alle Elemente der Spalte "col" werden mit einem Faktor multipliziert.

- `void m_sub_row(size_t dim, void *mat, size_t result, size_t sub, double f);`
  Von der Zeile "result" wird das "f-fache" der Zeile "sub" abgezogen.

- `void m_transpose(size_t dim, void *mat);`
  Die Matrix "mat" wird an der Hauptdiagonalen gespiegelt (transponiert).

- `int m_invert(size_t dim, void *mat);`
  Die Matrix "mat" wird invertiert. Im Fehlerfall ist der Rückgabewert MAT_ERR_MEMORY (nicht genügend freier Speicher) oder MAT_ERR_NOINV (inverse Matrix existiert nicht). Andernfalls wird die inverse Matrix in "mat" abgelegt und der Rückgabewert ist MAT_OK.

- `double m_det(size_t dim, const void *mat, int *pResult);`

  Die Determinante der Matrix "mat" wird berechnet und zurückgegeben. Falls nicht genügend freier Speicher zur Verfügung steht, wird als Ergebnis 0 zurückgegeben und *pResult auf MAT_ERR_MEMORY gesetzt, andernfalls auf MAT_OK. Es ist möglich, für pResult NULL zu übergeben, falls keine Erfolgs-/Fehlermeldung erforderlich ist.

- `int m_is_symmetric(size_t dim, const void *mat, double epsilon);`

  Ist die Matrix "mat" symmetrisch (Rückgabe = MAT_OK) oder ist sie nicht symmetrisch (Rückgabe = MAT_ERR_NOTEQ)? Bei einer symmetrischen Matrix weichen die an der Hauptdiagonalen gespiegelten Matrixelemente maximal um den Betrag "epsilon" voneinander ab.

- `void m_make_symmetric(size_t dim, void *mat, int mode);`

  Die Matrix "mat" wird symmetrisch gemacht. Entweder dadurch, dass die Elemente oberhalb der Hauptdiagonalen nach unten kopiert werden (mode = MAT_USE_UPPER) oder dadurch, dass die Elemente unterhalb der Hauptdiagonalen nach oben kopiert werden (mode = MAT_USE_LOWER) oder dadurch, dass die Mittelwerte aller an der Hauptdiagonalen gespiegelten Elemente berechnet werden (mode = MAT_USE_MEAN).

- `void m_get_row(size_t dim, size_t row, const void *mat, double *vec);`

  Einen einzelnen Zeilenvektor der Matrix "mat" zurückgeben.

- `void m_get_col(size_t dim, size_t col, const void *mat, double *vec);`

  Einen einzelnen Spaltenvektor der Matrix "mat" zurückgeben.

- `void m_get_diag(size_t dim, const void *mat, double *vec);`

  Die Hauptdiagonalelemente von "mat" in Form eines Vektors zurückgeben.

- `int m_charpol(size_t dim, const void *mat, double *pol);`

  Die Koeffizienten des charakteristischen Polynoms der Matrix "mat" werden ermittelt und in den Vektor "pol" geschrieben. Im Parameter "dim" ist die Dimension der Matrix "mat" zu übergeben. Hinweis: Es werden insgesamt (dim + 1) Koeffizienten ermittelt und in "pol" abgelegt. Der Ergebnisvektor "pol" muss daher (dim + 1) Elemente besitzen! Die Rückgabe ist MAT_OK oder MAT_ERR_MEMORY (zu wenig freier Speicher). Zur Berechnung des charakteristischen Polynoms wird der Algorithmus von Samuelson-Berkowitz benutzt.

- `double m_get_max(size_t dim, const void *mat);`

  Das maximale Element der Matrix "mat" wird zurückgegeben.

- `double m_get_min(size_t dim, const void *mat);`

  Das minimale Element der Matrix "mat" wird zurückgegeben.

- `int m_writecsv2(size_t dim, const void *mat, FILE *file);`

  Die Matrix "mat" wird im CSV-Format in die bereits geöffnete Datei "file" geschrieben. Die Datei bleibt nach dem Schreiben geöffnet. Rückgabewert == MAT_OK, falls die Matrix erfolgreich ausgegeben wurde oder MAT_ERR_FWRITE, falls ein Fehler bei der Ausgabe aufgetreten ist.

- `int m_writecsv(size_t dim, const void *mat, const char *filename);`

  Es wird die Datei "filename" erstellt und die Matrix "mat" im CSV-Format gespeichert. Rückgabewert == MAT_OK, falls die Ausgabe erfolgreich war, MAT_ERR_FOPEN bzw. MAT_ERR_FWRITE, falls ein Fehler aufgetreten ist.

- `int m_readcsv(size_t dim, void *mat, const char *filename);`

  Die Matrix "mat" wird aus einer CSV-Datei eingelesen. Falls die Matrix korrekt eingelesen wurde, ist der Rückgabewert == MAT_OK, ansonsten MAT_ERR_FORMAT oder MAT_ERR_FOPEN.

- `int m_readcsv2(size_t dim, void *mat, FILE *file);`

  Die Matrix "mat" wird aus der bereits geöffneten CSV-Datei "file" eingelesen, danach bleibt die Datei geöffnet. Falls die Matrix korrekt eingelesen wurde, ist der Rückgabewert == MAT_OK, sonst MAT_ERR_FORMAT.

Zerlegung von Matrizen, Eigenwerte

- `int m_qr_decomp(size_t dim, const void *mat, void *q, void *r);`

  Es wird die QR-Zerlegung der Matrix "mat" ermittelt. Der Rückgabewert ist entweder MAT_OK oder MAT_ERR_MEMORY (zu wenig freier Speicher).

- `int m_lu_decomp(size_t dim, const void *mat, void *lu, void *p, double *sgn);`

  Es wird die LU-Zerlegung P · MAT = L · U der Matrix "mat" ermittelt. Die Ergebnismatrix "lu" enthält im oberen Dreieck die Matrix U und unterhalb der Hauptdiagonalen L. Die Diagonalelemente von L sind nicht in "lu" abgelegt; sie sind alle gleich eins. Die während der Zerlegung durchgeführten Zeilenvertauschungen sind in der Permutationsmatrix "p" wiedergegeben. Mittels "sgn" wird der Wert $(-1)^n$ zurückgegeben, wobei "n" die Anzahl der durchgeführten Zeilenvertauschungen ist. Die Rückgabe ist MAT_OK bzw. MAT_ERR_NOINV (bei singulärer Matrix "mat").

- `int m_jacobi(size_t dim, const void *mat, void *eig, void *vec);`

  Mittels zyklischem Jacobi-Verfahren werden alle Eigenwerte und -vektoren der symmetrischen Matrix "mat" ermittelt. Die Eigenwerte werden auf der Hauptdiagonalen der Matrix "eig", die Eigenvektoren in den jeweiligen Spalten der Matrix "vec" abgelegt. Die Eigenwerte sind nicht nach ihrer Größe sortiert. Die Rückgabe der Funktion "m_jacobi" ist entweder MAT_OK, MAT_ERR_MEMORY oder MAT_ERR_ITFAIL (wenn das Jacobi-Verfahren nach "MAT_MAX_ITER" Iterationen kein Ergebnis mit der Genauigkeit "MAT_EPSILON" ermitteln konnte).

  Achtung: Die Funktion "m_jacobi" darf nur für symmetrische Matrizen "mat" aufgerufen werden. Wenn die Matrix "mat" nicht symmetrisch ist, werden keine korrekten Ergebnisse berechnet!

- ```
  int m_jacobi2(size_t dim, const void *mat, void *eig, void *vec,
                double epsilon, size_t max_iter);
  ```

  Mittels zyklischem Jacobi-Verfahren werden alle Eigenwerte und -vektoren der symmetrischen Matrix "mat" ermittelt. Die Eigenwerte werden auf der Hauptdiagonalen der Matrix "eig", die Eigenvektoren in den jeweiligen Spalten der Matrix "vec" abgelegt. Die Eigenwerte sind nicht nach ihrer Größe sortiert. Die Rückgabe der Funktion "m_jacobi2" ist entweder MAT_OK, MAT_ERR_MEMORY oder MAT_ERR_ITFAIL (wenn das Jacobi-Verfahren innerhalb von "max_iter" Iterationen kein Ergebnis mit der Genauigkeit "epsilon" ermitteln konnte).

  Achtung: Die Funktion "m_jacobi2" darf nur für symmetrische Matrizen "mat" aufgerufen werden. Ansonsten werden keine korrekten Ergebnisse berechnet!

- `int m_mises(size_t dim, const void *mat, double *eig, double *vec);`

  Der betragsmäßig größte Eigenwert der Matrix "mat" und der dazu passende Eigenvektor werden ermittelt und über die (Zeiger-)Parameter "eig" und "vec" an den Aufrufer zurückgegeben. Der Rückgabewert der Funktion "m_mises" ist MAT_OK, MAT_ERR_MEMORY oder MAT_ERR_ITFAIL (falls nach "MAT_MAX_ITER" Iterationen kein Ergebnis mit der Genauigkeit "MAT_EPSILON" ermittelt werden konnte).

  Achtung: Die Berechnung des Eigenwerts bzw. des Eigenvektors erfolgt über eine von-Mises-Vektoriteration. Insbesondere bei doppelten, eng benachbarten oder komplexen Eigenwerten konvergiert das Verfahren nur langsam oder überhaupt nicht. In diesen Fällen ist es sinnvoll, statt
  ```

"m_mises" die Funktion "m_mises2" aufzurufen und dort das Iterationsende über die Parameter "max_iter" bzw. "epsilon" gezielt einzustellen.

- ```
  int m_mises2(size_t dim, const void *mat, double *eig, double *vec,
               size_t max_iter, double epsilon);
  ```

  Der betragsmäßig größte Eigenwert der Matrix "mat" und der dazu passende Eigenvektor werden ermittelt und über die (Zeiger-)Parameter "eig" und "vec" an den Aufrufer zurückgegeben. Der Rückgabewert der Funktion "m_mises2" ist MAT_OK, MAT_ERR_MEMORY oder MAT_ERR_ITFAIL (falls nach "max_iter" Iterationen kein Ergebnis mit der Genauigkeit "epsilon" ermittelt werden konnte).

  Achtung: Die Berechnung des Eigenwerts bzw. des Eigenvektors erfolgt über eine von-Mises-Iteration. Bei doppelten, eng benachbarten oder komplexen Eigenwerten konvergiert das Verfahren nur langsam oder gar nicht. In solchen Fällen sollten die Parameter "max_iter" und "epsilon" auf geeignete Werte gesetzt werden, um ein Einfrieren des Programmablaufs zu vermeiden.

Funktionen zur Arbeit mit Vektoren

- ```
  void v_init(size_t dim, double *vec, int type);
  ```

  Der Vektor "vec" wird initialisiert. Für "type" sind folgende Werte möglich: MAT_INIT_NULL, MAT_INIT_RANDOM.

- ```
  void v_fill(size_t dim, double *vec, double value);
  ```

  Alle Elemente des Vektors "vec" werden auf den angegebenen Wert gesetzt.

- ```
  void v_print(size_t dim, const double *vec);
  ```

  Die Elemente des Vektors "vec" werden auf dem Bildschirm ausgegeben. Das Format kann über die Konstante MAT_PRN_FMT eingestellt werden.

- ```
  int v_equal(size_t dim, const double *v1, const double *v2);
  ```

  Falls beide Vektoren gleich sind, wird MAT_OK zurückgegeben, andernfalls MAT_ERR_NOTEQ.

- ```
  void v_fadd(size_t dim, double *vresult, const double *v1, double f);
  ```

  Das "f-fache" des Vektors "v1" wird zum Vektor "vresult" addiert.

- ```
  void v_fmul(size_t dim, double *vresult, double factor);
  ```

  Der Vektor "vresult" wird mit "factor" multipliziert.

- ```
  double v_dot(size_t dim, const double *v1, const double *v2);
  ```

  Das Skalarprodukt der Vektoren "v1" und "v2" wird berechnet.

- ```
  void v_cross(double *vresult, const double *v1, const double *v2);
  ```

  Das Kreuzprodukt der dreidimensionalen (!) Vektoren "v1" und "v2" wird berechnet und im Vektor "vresult" abgespeichert.

- ```
  double v_abs(size_t dim, const double *v1);
  ```

  Der Betrag des Vektors "v1" wird berechnet.

- ```
  int v_normalize(size_t dim, double *v1);
  ```

  Der Vektor "v1" wird normalisiert (Rückgabe == MAT_OK). Hat "v1" die Länge null, bleibt "v1" unverändert (Rückgabe == MAT_ERR_VNORM).

- ```
  double v_get_max(size_t dim, const double *vec);
  ```

  Das maximale Element des Vektors "vec" wird zurückgegeben.

- `double v_get_min(size_t dim, const double *vec);`

  Das minimale Element des Vektors "vec" wird zurückgegeben.

- `void m_vmul(size_t dim, double *vresult, const void *mat, const double *v);`

  Das Produkt der Matrix "mat" und des Vektors "v" wird im Vektor "vresult" abgelegt.

Polynome

- `HM_COMPLEX v_cpolyval(size_t dim, const HM_COMPLEX *a, HM_COMPLEX x);`

  Der Wert des Polynoms $a_0 \cdot x^n + a_1 \cdot x^{n-1} + a_2 \cdot x^{n-2} \ldots + a_n$ wird für den angegebenen komplexen Wert "x" berechnet. Die Anzahl der komplexen Koeffizienten ist im Parameter "dim" zu übergeben. Die Koeffizienten sowie das Ergebnis werden HM_COMPLEX-Strukturen gespeichert:

  ```
  typedef struct tagHM_COMPLEX
  {
      double val[2];   /* val[0] = Realteil, val[1] = Imaginärteil */
  } HM_COMPLEX;
  ```

- `double v_dpolyval(size_t dim, const double *a, double x);`

  Der Wert des Polynoms $a_0 \cdot x^n + a_1 \cdot x^{n-1} + a_2 \cdot x^{n-2} \ldots + a_n$ wird für den angegebenen reellen Wert "x" berechnet. Die Anzahl der reellen Koeffizienten ist im Parameter "dim" zu übergeben.

- `int v_croots(size_t dim, const HM_COMPLEX *a, HM_COMPLEX *roots);`

  Alle reellen und komplexen Nullstellen des Polynoms $a_0 \cdot x^n + a_1 \cdot x^{n-1} + a_2 \cdot x^{n-2} \ldots + a_n$ werden ermittelt und im Vektor "roots" abgelegt. Die komplexen Koeffizienten des Polynoms sind in a[0], a[1] ... a[n] zu übergeben, deren Anzahl ist im Parameter "dim" zu übergeben. Hinweis: Ein Polynom mit z Koeffizienten besitzt genau (z-1) Nullstellen. Der Ergebnisvektor "roots" hat also ein Element weniger als Vektor "a". Rückgabe: MAT_OK, MAT_ERR_ITFAIL (Algorithmus konvergiert nicht) oder MAT_ERR_DIM (falls dim < 2 oder a[0] == 0).

  Die Nullstellenbestimmung basiert auf dem Verfahren von Weierstraß-Durand-Kerner.

- `int v_croots2(size_t dim, const HM_COMPLEX *a, HM_COMPLEX *roots, double epsilon, size_t max_iter, HM_COMPLEX wdk_init);`

  Siehe v_croots(). Bei dieser Variante kann zusätzlich die gewünschte Genauigkeit, die maximale Anzahl der Iterationen sowie der Startwert für das verwendete Weierstraß-Durand-Kerner-Iterationsverfahren angegeben werden. Rückgabe: MAT_OK, MAT_ERR_ITFAIL (der Algorithmus konvergiert nicht) oder MAT_ERR_DIM (falls dim < 2 oder a[0] == 0).

- `int v_droots(size_t dim, const double *a, HM_COMPLEX *roots);`

  Alle reellen und komplexen Nullstellen des Polynoms $a_0 \cdot x^n + a_1 \cdot x^{n-1} + a_2 \cdot x^{n-2} \ldots + a_n$ werden ermittelt und im Vektor "roots" abgelegt. Die reellen Koeffizienten des Polynoms sind in a[0] ... a[n] zu übergeben, deren Anzahl ist im Parameter "dim" zu übergeben. Hinweis: Ein Polynom mit z Koeffizienten besitzt genau (z-1) Nullstellen. Der Ergebnisvektor "roots" hat also ein Element weniger als Vektor "a". Rückgabe: MAT_OK, MAT_ERR_ITFAIL (Algorithmus konvergiert nicht), MAT_ERR_DIM (falls dim < 2 oder a[0] == 0) oder MAT_ERR_MEMORY (freier Speicher reicht nicht aus). Die Nullstellenbestimmung basiert auf dem Verfahren von Weierstraß-Durand-Kerner.

- `int v_droots2(size_t dim, const double *a, HM_COMPLEX *roots, double epsilon, size_t max_iter, HM_COMPLEX wdk_init);`

  Siehe v_droots(). Bei dieser Variante kann zusätzlich die gewünschte Genauigkeit, die maximale Anzahl der Iterationen sowie der Startwert für das verwendete Weierstraß-Durand-Kerner-Iterationsverfahren angegeben werden. Rückgabe: MAT_OK, MAT_ERR_ITFAIL (der Algorithmus konvergiert nicht), MAT_ERR_DIM (falls dim < 2 oder a[0] == 0) oder MAT_ERR_MEMORY (freier Speicher reicht nicht aus).

Lineare Gleichungssysteme

- `int lin_solve(size_t dim, double *x, const void *a, const double *y);`

  Das lineare Gleichungssystem A · x = y wird gelöst. Es wird zunächst eine QR-Zerlegung der Matrix durchgeführt und anschließend die Lösung des Gleichungssystems durch Rückwärts-Einsetzen bestimmt. Mögliche Rückgabewerte sind MAT_OK, MAT_ERR_MEMORY und MAT_ERR_NOINV. Bitte die Kommentare zur Funktion "lin_solve_qr" ebenfalls beachten.

- ```
  int lin_solve_qr(size_t dim, double *x,
                   const void *q, const void *r, const double *y);
  ```

  Das lineare Gleichungssystem (QR) · x = y wird gelöst. Vor dem Aufruf von "lin_solve_qr" muss zunächst eine QR-Zerlegung der Matrix erfolgen, zum Beispiel mittels "m_qr_decomp". (Alternativ kann auch die Funktion "lin_solve" aufgerufen werden, wo beide Schritte - QR-Zerlegung und die die Lösung des Gleichungssystems - gemeinsam durchgeführt werden.) Der Rückgabewert beträgt MAT_OK, falls das Gleichungssystem gelöst werden konnte, ansonsten MAT_ERR_NOINV.

Schnelle Fouriertransformation (FFT, Fast Fourier Transform)

- `int v_cfft(size_t n, HM_COMPLEX *f);`

  Die schnelle Fouriertransformation (FFT, Fast Fourier Transform) wird für den Vektor "f" mit "n" komplexen Elementen berechnet. Achtung: Die Elementanzahl "n" muss eine Zweierpotenz sein, sonst wird die Funktion mit dem Fehlercode MAT_ERR_DIM abgebrochen. Das Ergebnis der FFT wird wiederum im Vektor "f" zurückgegeben. Die Elemente des Vektors "f" werden also durch den Aufruf von "v_cfft" überschrieben. Mögliche Rückgabewerte: MAT_OK, MAT_ERR_MEMORY oder MAT_ERR_DIM.

- `int v_dfft(size_t n, const double *f, HM_COMPLEX *y);`

  Die schnelle Fouriertransformation (FFT, Fast Fourier Transform) wird für den Vektor "f" mit "n" reellen Elementen berechnet. Achtung: Die Elementanzahl "n" muss eine Zweierpotenz sein, sonst wird die Funktion mit dem Fehlercode MAT_ERR_DIM abgebrochen. Das Ergebnis der FFT wird in den Vektor "y" geschrieben, welcher Platz für "n" komplexe Elemente des Typs HM_COMPLEX bieten muss. Mögliche Rückgabewerte sind: MAT_OK, MAT_ERR_MEMORY oder MAT_ERR_DIM.

- `int v_ifft(size_t n, HM_COMPLEX *y);`

  Die IFFT (Inverse Fast Fourier Transform) wird für den Vektor "y" mit "n" komplexen Elementen berechnet. Achtung: Die Elementanzahl "n" muss eine Zweierpotenz sein, sonst wird "v_ifft" mit dem Fehler MAT_ERR_DIM abgebrochen. Das Ergebnis der IFFT wird im Vektor "y" zurückgegeben, die Elemente des Vektors "y" werden also durch den Aufruf von "v_ifft" überschrieben. Mögliche Rückgabewerte sind: MAT_OK, MAT_ERR_MEMORY oder MAT_ERR_DIM.
  ```

3. Programmierung in C++

3.1. Beispiele

```
// ------------------------------------------------------------
// Programmbeispiel: Vektoren & Matrizen
// ------------------------------------------------------------
#include <iostream>
#include "matrix.hpp"
using namespace std;
using namespace HMMatrix;                    // Namespace von HMMatrix

int main()
{
    const size_t dim = 5;
    Vector vect(dim), prod(dim);             // Vektoren definieren
    Matrix matr(dim);                        // Matrix definieren

    vect.Init(MAT_INIT_RANDOM);              // Mit Zufallszahlen (0...1) füllen
    matr.Init(MAT_INIT_IDENTITY);            // Einheitsmatrix erzeugen
    matr *= 2.0;                             // Alle Matrixelemente verdoppeln
    prod = matr * vect;                      // Matrix mit Vektor multiplizieren

    cout << "matr = \n" << matr << "\n";
    cout << "vect = \n" << vect << "\n\n";
    cout << "prod = \n" << prod << "\n\n";

    // Inverse Matrix mit 5 Nachkommastellen und Feldbreite von 9 Zeichen ausgeben
    cout << MatrixFormat(9, 5);
    cout << "Inverse Matrix = \n" << matr.Inverse() << "\n";
}
```

```
// ------------------------------------------------------------
// Programmbeispiel: lineares Gleichungssystem
// ------------------------------------------------------------
#include <iostream>
#include "matrix.hpp"
using namespace std;
using namespace HMMatrix;

int main()
{
    try
    {
        Matrix a(3);
        a.Row(0) = 2, 4, 6;                  // Matrix A mit Werten belegen
        a.Row(1) = 3, 5, 7;
        a.Row(2) = 4, 6, 1;

        Vector x(3), y(3);
        y.Elements() = 280, 340, 190;        // Vektor y mit Werten belegen

        x = LinSolve(a, y);                  // Gleichungssystem A x = y lösen
        cout << "x = " << x << "\n";         // Ausgabe: 10.0; 20.0; 30.0
    }
    catch(const exception& err)
    {
        cout << err.what() << "\n";
    }
}
```

3.2. Klassen- und Funktionsübersicht

Konstruktoren

Die Matrix- und Vektor-Klassen sind im Namespace HMMatrix definiert. Den Konstruktoren wird die gewünschte Dimension als Parameter übergeben.

```
HMMatrix::Vector vect(3);              // Dreidimensionaler Vektor
HMMatrix::Matrix matr(5);              // Definition einer 5x5-Matrix

using namespace HMMatrix;
Matrix a(3), q(3), r(3), l(3), u(3), p(3);
Vector x(3), y(3), z(3);
```

Zugriff auf Vektor- und Matrixelemente

Zum Zugriff auf Vektor- und Matrixelemente existieren verschiedene Methoden. Zu beachten ist, dass – wie in C++ üblich – generell nullbasierte Indizes verwendet werden, vect(0) ist das erste Element des Vektors.

```
x(0) = 1.5; x(1) = 2.5; x(2) = 3.5;    // Elemente separat setzen
y.Elements() = 1.5, 2.5, 3.5;          // Kompletten Vektor setzen

a(0, 0) = 9.9; a(0, 1) = 8.8;          // Elemente separat setzen
a.Row(1) = 6.6, 5.5, 4.4;              // Komplette Matrixzeile setzen

double matmin = matr.GetMin();         // Kleinstes Matrixelement ermitteln
double matmax = matr.GetMax();         // Größtes Matrixelement ermitteln
double vecmin = vect.GetMin();         // Kleinstes Vektorelement ermitteln
double vecmax = vect.GetMax();         // Größtes Vektorelement ermitteln

x = a.GetRow(0);                       // Zeilenvektor ermitteln
y = a.GetCol(0);                       // Spaltenvektor ermitteln
z = a.GetDiag();                       // Hauptdiagonale (als Vektor) ermitteln

// Kompletten Vektor initialisieren
vect.Fill(1.5);                        // Alle Elemente auf bestimmten Wert
x.Init(MAT_INIT_NULL);                 // Alle Elemente null
y.Init(MAT_INIT_RANDOM);               // Zufallszahlen (0...1)

// Komplette Matrix initialisieren
matr.Fill(-2.5);                       // Alle Elemente auf bestimmten Wert
a.Init(MAT_INIT_NULL);                 // Alle Elemente null
q.Init(MAT_INIT_RANDOM);               // Zufallszahlen (0...1)
r.Init(MAT_INIT_IDENTITY);             // Einheitsmatrix
l.Init(MAT_INIT_HILBERT);              // Hilbertmatrix
u.Init(MAT_INIT_INV_HILBERT);          // Inverse Hilbertmatrix
```

Iteratoren

Vektoren und Matrizen können mit Iteratoren vollständig durchlaufen werden.

```
// Alle Elemente eines Vektors durchlaufen
for(auto v_it = vect.begin(); v_it != vect.end(); ++v_it)
    cout << *v_it << "\n";

// Alle Elemente einer Matrix durchlaufen
for(auto m_it = matr.begin(); m_it != matr.end(); ++m_it)
    cout << *m_it << "\n";
```

Kompatibilität mit std::vector

Vektoren unterstützen die Methoden size(), push_back(), pop_back(), clear() und empty(). Die Bedeutung dieser Methoden entspricht den gleichnamigen Methoden bei std::vector.

```
Vector my_vec = MATLABVector("[1, 2, 3]");
my_vec.push_back(10);
my_vec.push_back(20);                    // my_vec ist jetzt [1, 2, 3, 10, 20]
my_vec.pop_back();
my_vec.pop_back();                       // my_vec ist jetzt [1, 2, 3]

auto sz = my_vec.size();                 // sz == 3
my_vec.clear();
auto em = my_vec.empty();                // em == true
```

Teile von Vektoren extrahieren

Die Methode Vector::Slice() extrahiert Teile von Vektoren.

```
// Von Position 0 ausgehend 5 Elemente zurückgeben, Schrittweite = 1 (Standardwert)
Vector svec = MATLABVector("[0 1 2 3 4 5]");
Vector sl_0 = svec.Slice(0, 5);          // sl_0 ist jetzt: [0, 1, 2, 3, 4]

// Von Position 1 ausgehend 3 Elemente zurückgeben, Schrittweite = 1 (Standardwert)
Vector sl_1 = svec.Slice(1, 3);          // sl_1 ist jetzt: [1, 2, 3]

// Von Position 1 ausgehend 3 Elemente zurückgeben, Schrittweite = 2
Vector sl_2 = svec.Slice(1, 3, 2);       // sl_2 ist jetzt: [1, 3, 5]
```

Symmetrische Matrizen

Über die Methode MakeSymmetric(mode) kann eine unsymmetrische Matrix in eine symmetrische Matrix umgewandelt werden.

```
#include "matrix.hpp"
#include <iostream>

int main(void)
{
    using namespace HMMatrix;
    using namespace std;

    Matrix v(3), d(3);
    Matrix m = MATLABMatrix("[1, 2, 3; 4, 5, 6; 7, 8, 9]");
    m.MakeSymmetric(MAT_USE_MEAN);
    if(m.IsSymmetric()) cout << "Matrix ist nun symmetrisch:\n" << m;
}
```

Zum Parameter „mode“ vergl. die Erläuterungen zur C-Funktion m_make_symmetric() weiter oben: Es stehen die Varianten MAT_USE_MEAN, MAT_USE_UPPER und MAT_USE_LOWER zur Verfügung.

Arithmetische Operatoren

Zur Addition, Subtraktion und Multiplikation von Vektoren existieren die üblichen Operatoren.

```
x *= 2;                                  // Alle Elemente verdoppeln
y += x;                                  // Vektor x zu y hinzuaddieren
y = x + x;                               // Vektoren addieren

double s = x * x;                        // Skalarprodukt
y = CrossProduct(x, y);                  // Vektorprodukt
y = ((x - y) * 3 + y) / 2.5;             // Mehrere Operationen in einer Zeile
```

Für Matrizen gilt Entsprechendes.

```
x = a * y;                          // Matrix mal Vektor
a = l * u;                          // Matrizen multiplizieren
a = l + u;                          // Matrizen addieren
a = l - u;                          // Matrizen subtrahieren

a *= 2;                             // Alle Matrixelemente verdoppeln...
a /= 2;                             // ...und wieder halbieren
```

Vergleichsoperatoren

Vektoren und Matrizen können miteinander verglichen, Matrizen auf Symmetrie geprüft werden. Da hierbei letztlich einzelne Fließkommazahlen miteinander verglichen bzw. auf Gleichheit geprüft werden, ist eine gewisse Vorsicht bezgl. Rundungsfehlern geboten.

```
if(l == u) cout << "Matrizen sind gleich";
if(l != u) cout << "Matrizen sind nicht gleich";

if(x == y) cout << "Vektoren sind gleich";
if(x != y) cout << "Vektoren sind nicht gleich";

if(a.IsSymmetric()) cout << "Matrix ist symmetrisch";
```

Ein- und Ausgabe

Vektoren und Matrizen können auf C++-Streams ausgegeben werden. Standardmäßig erfolgt die Ausgabe 2 Nachkommastellen bei einer Feldbreite von 8 Zeichen. Das Ausgabeformat kann aber bei Bedarf verändert werden. Matrizen können darüber hinaus in CSV-Dateien gespeichert werden (kompatibel zu Microsoft Excel) und aus Zeichenketten im „MATLAB-Format“ eingelesen werden.

```
cout << MatrixFormat(5, 3);         // 5 Zeichen Feldbreite, 3 Nachkommastellen
cout << a << "\n" << x;             // Ausgabe erfolgt nun im neuen Format

WriteCSV(a, "test.csv");            // Speichern im CSV-Format
ReadCSV(l, "test.csv");             // CSV-Datei wieder einlesen

try
{
    a = MATLABMatrix("[1 2; 3 4]"); // Matrix im „MATLAB-Format“ einlesen
    x = MATLABVector("[1,2,33,0]"); // Vektor im „MATLAB-Format“ einlesen
}
catch(std::runtime_error& err)
{
    std::cout << err.what();        // Ausnahmefehler bei ungültigem Format
}
```

Charakteristisches Polynom

Die Methode CharPol() berechnet das charakteristische Polynom einer Matrix und gibt dessen Koeffizienten als Vektor zurück. Zur Berechnung des charakteristischen Polynoms wird der Algorithmus von Samuelson-Berkowitz eingesetzt (vergl. Anhang A).

```
a.Row(0) = 0, -2, -1;
a.Row(1) = 2,  2,  1;
a.Row(2) = 0,  2,  2;

HMMatrix::Vector pol = a.CharPol();
std::cout << pol << "\n";           // Ausgabe: 1.00; -4.00; 6.00; -4.00
```

Polynome

Alle reellen und komplexen Nullstellen eines Polynoms $a_0 \cdot x^n + a_1 \cdot x^{n-1} + a_2 \cdot x^{n-2} \ldots + a_n$ können mit der Methode Roots() ermittelt werden. Die Koeffizienten des Polynoms müssen als Elemente eines Vektors vorliegen. Die Rückgabe der Nullstellen erfolgt als std::vector< std::complex<double> >.

Zur Berechnung des Funktionswerts an einer Stelle x existiert darüber hinaus die Methode PolyVal(). Die Koeffizienten des Polynoms werden auch hier in Form von Vektorelementen übergeben. Das nachfolgende Beispiel zeigt die Berechnung der beiden reellen Nullstellen der Funktion $y(x) = x^2 - 1$ und anschließend die Berechnung der beiden komplexen Nullstellen von $y(x) = x^2 + 1$.

```
HMMatrix::Vector pol(3);
pol.Elements() = 1, 0, -1;          // y(x) = 1·x² + 0·x - 1
double y1 = pol.PolyVal(1.0);       // Funktionswert für x = 1.0 berechnen

auto nst = pol.Roots();             // Rückgabe erfolgt als: vector< complex<double> >
std::cout << nst[0] << ", ";        // Zwei reelle Nullstellen bei +1 und -1
std::cout << nst[1] << "\n";

pol.Elements() = 1, 0, +1;          // y(x) = 1·x² + 0·x + 1
nst = pol.Roots();                  // Rückgabe erfolgt als:  vector< complex<double> >

std::cout << nst[0] << ", ";        // Zwei komplexe Nullstellen bei +i und -i
std::cout << nst[1] << "\n";
```

Weitere Operationen

Weitere Operationen gestatten die Berechnung transponierter oder inverser Matrizen sowie von Determinanten. Auch QR- und LU-Zerlegungen von Matrizen sind möglich.

```
double d = a.Determinant();         // Determinante
a.SwapRow(1, 2);                    // Zeilen vertauschen

a = l.GetTranspose();               // Transponierte Matrix (Variante 1)
l.Transpose();                      // Transponierte Matrix (Variante 2)

a = l.Inverse();                    // Inverse Matrix (Variante 1)
l.Invert();                         // Inverse Matrix (Variante 2)

a.QRDecomp(q, r);                   // QR-Zerlegung
double sgn = a.LUDecomp(l, u, p);   // LU-Zerlegung
```

Während der Berechnung der LU-Zerlegung werden geeignete Zeilenvertauschungen durchgeführt, die in der Permutationsmatrix P wiedergegeben sind ($P \cdot MAT = L \cdot U$). Der Rückgabewert der Methode LUDecomp() beträgt $(-1)^n$, wobei n die Anzahl der Zeilenvertauschungen ist.

Lineare Gleichungssysteme

Zur Lösung linearer Gleichungssysteme des Typs $A \cdot \vec{x} = \vec{y}$ gibt es die beiden Funktionen LinSolve() und LinSolveQR(). Zunächst wird die QR-Zerlegung der Matrix A bestimmt, anschließend folgt die Lösung des Gleichungssystems durch Rückwärts-Einsetzen. Die Funktion LinSolve() führt beide Schritte nacheinander aus.

Falls die QR-Zerlegung der Matrix A bereits bekannt sein sollte, bietet sich der Aufruf der zweiten Funktion LinSolveQR() an: Hier werden statt der Matrix A die beiden Teilmatrizen Q und R als Parameter übergeben.

```
#include <iostream>
#include "matrix.hpp"

int main()
{
    using namespace std;
    using namespace HMMatrix;

    Matrix a(3), q(3), r(3);
    Vector x(3), y(3);

    a.Row(0) = 1, 3, 2;
    a.Row(1) = 2, 4, 4;
    a.Row(2) = 3, 2, 1;

    y.Elements() = 26, 44, 20;

    x = LinSolve(a, y);
    cout << x << "\n";                // Ausgabe: 2.0, 4.0, 6.0

    a.QRDecomp(q, r);
    x = LinSolveQR(q, r, y);
    cout << x << "\n";                // Ausgabe: 2.0, 4.0, 6.0
}
```

Ausnahmefehler (Exceptions)

Fehler, die bei der Arbeit mit Vektoren und Matrizen auftreten, werden durch Ausnahmefehler des Typs std::runtime_error angezeigt. Als Beispiel sei der Versuch gezeigt, eine singuläre Matrix zu invertieren. Solche Ausnahmefehler können mittels try/catch ausgewertet werden.

```
#include <iostream>
#include "matrix.hpp"

int main()
{
    try
    {
        HMMatrix::Matrix a(5);
        a.Init(MAT_INIT_NULL);
        a.Invert();                   // Ausnahmefehler wird ausgelöst
    }
    catch(std::runtime_error& err)
    {
        std::cout << err.what();      // "Matrix inversion failed"
    }
}
```

Schnelle Fouriertransformation (FFT, Fast Fourier Transform)

Die FFT berechnet, vereinfacht ausgedrückt, welche Frequenzen im Originalsignal enthalten sind, welche Sinusschwingungen also mit welcher Phase addiert werden müssten, um auf das ursprüngliche Signal zu kommen. Konkret erhält man für jeden Frequenzpunkt (von der Frequenz 0 bis zur halben Abtastfrequenz) eine komplexe Zahl, die Amplitude und Phase der entsprechenden Sinusschwingung repräsentiert.[1]

Das folgende Beispiel nutzt die Funktion FFT() zur Ausgabe eines Amplitudenspektrums. Zur Berechnung der inversen FFT existiert zudem die Funktion IFFT(), die hier allerdings nicht benötigt wird.

```
#include "matrix.hpp"
#include <iostream>

int main()
{
    using namespace HMMatrix;

    const size_t dim = 1024;
    double t = 0, delta_t = 0.001;

    // Sinussignal mit einer Frequenz von 200 Hz, mit etwas Rauschen überlagert
    Vector y(dim);
    for(size_t i = 0; i < dim; ++i, t += delta_t)
        y[i] = sin(6.28319 * 200. * t) + 2. * rand() / RAND_MAX - 1.;

    // Achtung: Die Elementanzahl muss eine Zweierpotenz sein!
    auto Y = FFT(y);

    // Amplitudenspektrum auf Konsole ausgeben
    for(size_t i = 0; i < dim / 2; ++i)
    {
        double f = 1.0 * i / (dim * delta_t);   // Frequenz
        double a = 2.0 * abs(Y[i]) / dim;       // Amplitude
        std::cout << f << "\t" << a << "\n";
    }
}
```

[1] Aus: www.mikrocontroller.net, Spektralanalyse mit der FFT, abgerufen am 20. April 2015.

Anhang A: Algorithmus von Samuelson-Berkowitz

Zur Berechnung der Determinante und des charakteristischen Polynoms einer Matrix A wird der Algorithmus von Samuelson-Berkowitz verwendet. Dieser ist in der Literatur[2] und auch bei Wikipedia[3] dokumentiert.

Die Implementierung in HMMatrix basiert auf der Darstellung in Wikipedia. Die hier verwendeten Zeilenvektoren und nicht-quadratischen Matrizen werden allerdings von HMMatrix nicht unterstützt. Das Verfahren muss daher so angepasst werden, dass es auch mit quadratischen Matrizen und Spaltenvektoren funktioniert[4]:

Eingabe:

n×n-Matrix $A = \begin{pmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,n} \end{pmatrix}$

Ausgabe:

Charakteristisches Polynom der Matrix A

Ablauf:

1 **Lokale Variablen:** $\overrightarrow{cp} = \begin{pmatrix} cp_1 \\ \vdots \\ cp_{n+1} \end{pmatrix}$ und $\vec{q} = \begin{pmatrix} q_1 \\ \vdots \\ q_{n+1} \end{pmatrix}$

2 **Initialisierung:** $\overrightarrow{cp} = \begin{pmatrix} 1 \\ -a_{1,1} \\ 0 \\ \vdots \\ 0 \end{pmatrix}$, $\vec{q} = \mathbf{0}$

3 **Für alle** i **von** 1 **bis** (n-1)

4 $q_1 = 1,\ q_2 = -a_{i+1,i+1}$

5 **Für alle** k **von** 1 **bis** i

6 $q_{k+2} = -\vec{r}(A,i) \circ \left[Z(A,i)^{k-1} \cdot \vec{s}(A,i)\right]$ (mit dem Skalarprodukt ∘)

7 $\overrightarrow{cp} = T(\vec{q}, i+2) \cdot \overrightarrow{cp}$

8 **Rückgabe:** $\overrightarrow{cp}$

Mit den folgenden Abkürzungen:

$$\vec{r}(A,i) = \begin{pmatrix} r_1 \\ \vdots \\ r_i \\ r_{i+1} \\ \vdots \\ r_n \end{pmatrix} = \begin{pmatrix} a_{(i+1),1} \\ \vdots \\ a_{(i+1),i} \\ 0 \\ \vdots \\ 0 \end{pmatrix} \qquad \vec{s}(A,i) = \begin{pmatrix} s_1 \\ \vdots \\ s_i \\ s_{i+1} \\ \vdots \\ s_n \end{pmatrix} = \begin{pmatrix} a_{1,(i+1)} \\ \vdots \\ a_{i,(i+1)} \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

[2] Siehe z. B. http://dis.um.es/~domingo/08/CD/27May/Posters/ALopez/poster.pdf

[3] Seite „Algorithmus von Samuelson-Berkowitz“. In: Wikipedia, Die freie Enzyklopädie. Stand: 10. März 2013, 18:04 UTC. URL: http://de.wikipedia.org/w/index.php?title=Algorithmus_von_Samuelson-Berkowitz&oldid=115222780 (Abgerufen: 8. Juli 2013, 09:51 UTC)

[4] In der folgenden Darstellung wird wie in der Literatur üblich eine eins-basierte Indizierung der Vektor- und Matrix-Elemente verwendet. HMMatrix verwendet dagegen eine null-basierte Indizierung.

$$\text{n×n-Matrix } Z(A,i) = \begin{pmatrix} z_{1,1} & \cdots & z_{1,n} \\ \vdots & \ddots & \vdots \\ z_{n,1} & \cdots & z_{n,n} \end{pmatrix} = \begin{pmatrix} a_{1,1} & \cdots & a_{1,i} & 0 & \cdots & 0 \\ \vdots & & \vdots & \vdots & & \vdots \\ a_{i,1} & \cdots & a_{i,i} & 0 & \cdots & 0 \\ 0 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & 0 & \cdots & 0 \end{pmatrix}$$

$$\text{(n+1)×(n+1)-Matrix } T(\vec{q},i) = \begin{pmatrix} t_{1,1} & \cdots & t_{1,n+1} \\ \vdots & \ddots & \vdots \\ t_{n+1,1} & \cdots & t_{n+1,n+1} \end{pmatrix} = \begin{pmatrix} q_1 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ q_2 & q_1 & \cdots & 0 & 0 & \cdots & 0 \\ q_3 & q_2 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & & \vdots & \vdots & & \vdots \\ q_{i-1} & q_{i-2} & \cdots & q_1 & 0 & \cdots & 0 \\ q_i & q_{i-1} & \cdots & q_2 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & & \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \end{pmatrix}$$

Anhang B: Eigenwerte und Eigenvektoren

Eigenwerte und Eigenvektoren reeller symmetrischer Matrizen, Jacobiverfahren

Eine reelle symmetrische Matrix ist als normale Matrix diagonalisierbar mit einem vollen Satz Eigenvektoren, wobei die Eigenvektoren zu verschiedenen Eigenwerten orthogonal sind.[5]

HMMatrix[6] stellt zur Berechnung der Eigenwerte und Eigenvektoren von reellen symmetrischen Matrizen in der Programmiersprache C die Funktionen m_jacobi() und m_jacobi2() zur Verfügung und in der Programmiersprache C++ die Methode Matrix::Jacobi(). Diese Funktionen nutzen zur Berechnung der Eigenwerte und Eigenvektoren das zyklische Jacobiverfahren.[7] Zwei Beispiele in den Programmiersprachen C++ und C zeigen die Anwendung der genannten Funktionen:

```
// Jacobiverfahren in C++
#include "matrix.hpp"
#include <iostream>

int main()
{
    using namespace std;
    using namespace HMMatrix;

    Matrix a(3), eig(3), vec(3);
    a.Row(0) =  2, -1,  0;
    a.Row(1) = -1,  2, -1;
    a.Row(2) =  0, -1,  2;

    a.Jacobi(eig, vec);
    // Die drei Eigenvektoren stehen in den Spalten von "vec",
    // die Eigenwerte auf der Hauptdiagonalen von "eig".

    cout << MatrixFormat(6, 3);
    for(size_t i = 0; i < 3; ++i)
    {
        cout << i + 1 << ". Eigenwert: ";
        cout << eig(i, i) << "\n";

        cout << i + 1 << ". Eigenvektor: ";
        cout << vec.GetCol(i) << "\n";
    }
}
```

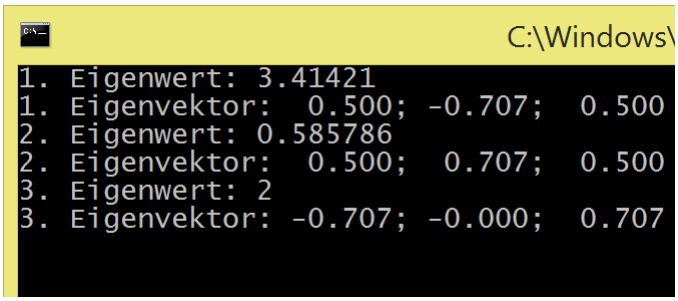


Abbildung 2 – Ausgabe des C++-Programmbeispiels zum Jacobiverfahren

[5] Aus: Wikipedia, Die freie Enzyklopädie. Stand: 12. Mai 2014, 19:19 UTC. URL: http://de.wikipedia.org/w/index.php?title=Symmetrische_Matrix&oldid=130343678 (Abgerufen: 21. Juli 2014, 09:21 UTC)

[6] Download im Internet: http://kuepper.userweb.mwn.de/software.htm

[7] Siehe z. B. Heinrich Sormann, Numerische Methoden in der Physik (515.421), Skriptum WS2012/13, Kapitel 7, Technische Universität Graz

```c
/* Jacobiverfahren in C */
#include "matrix.h"
#include <stdio.h>
#define DIM 3

int main(void)
{
    double v[DIM], eig[DIM][DIM], vec[DIM][DIM];
    double a[DIM][DIM] =
    {
            {  2, -1,  0 },
            { -1,  2, -1 },
            {  0, -1,  2 }
    };

    m_jacobi(DIM, a, eig, vec);
    /* Die drei Eigenvektoren stehen in den Spalten von "vec",
       die Eigenwerte auf der Hauptdiagonalen von "eig". */

    for(size_t i = 0; i < 3; ++i)
    {
        printf("%d. Eigenwert: %f\n", (int)i + 1, eig[i][i]);
        printf("%d. Eigenvektor: ", (int)i + 1);
        m_get_col(DIM, i, vec, v);
        v_print(DIM, v);
    }

    return 0;
}
```

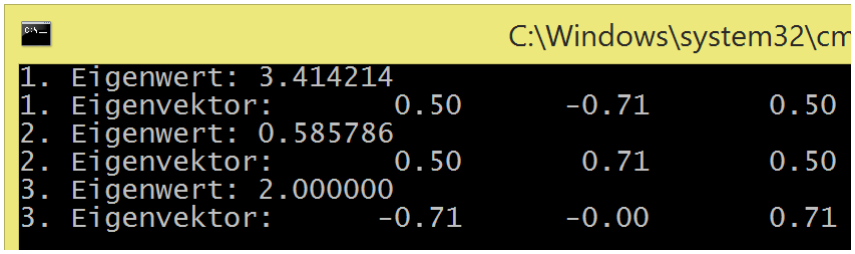


Abbildung 3 – Ausgabe des C-Programmbeispiels zum Jacobiverfahren

Betragsgrößter Eigenwert einer Matrix, von-Mises-Vektoriteration

Die Potenzmethode, Vektoriteration oder von-Mises-Iteration (nach Richard von Mises) ist ein numerisches Verfahren zur Berechnung des betragsgrößten Eigenwertes und des dazugehörigen Eigenvektors einer Matrix.[8]

HMMatrix stellt die C-Funktionen m_mises(), m_mises2() sowie die C++-Methode Matrix::Mises() zur Verfügung, um den betragsgrößten Eigenwert (inkl. Eigenvektor) einer Matrix zu berechnen.

Insbesondere bei doppelten, eng benachbarten oder komplexen Eigenwerten konvergiert das Verfahren oft nur langsam oder überhaupt nicht. In diesen Fällen ist es sinnvoll, statt "m_mises" die Funktion "m_mises2" aufzurufen (in C-Programmen) und dort das Iterationsende über die beiden Parameter "max_iter" (maximale Anzahl der Iterationen) bzw. "epsilon" (gewünschte Genauigkeit) gezielt einzustellen. Bei der Methode Matrix::Mises() (in C++-Programmen) existieren zu diesem Zweck zwei optionale Parameter.

[8] Aus: Wikipedia, Die freie Enzyklopädie. Stand: 21. Juni 2014, 15:03 UTC. URL: http://de.wikipedia.org/w/index.php?title=Potenzmethode&oldid=131501250 (Abgerufen: 21. Juli 2014, 11:40 UTC)

```
// ------------------------------------------------------
// von-Mises-Vektoriteration in C++
// ------------------------------------------------------
#include "matrix.hpp"
#include <iostream>

int main()
{
    using namespace std;
    using namespace HMMatrix;

    Matrix a(3);
    Vector v(3);
    double e;

    a.Row(0) =  2, -1,  0;
    a.Row(1) = -1,  2, -1;
    a.Row(2) =  0, -1,  2;
    v = a.Mises(e);

    cout << MatrixFormat(6, 3);
    cout << "Eigenwert mit groesstem Betrag: " << e;
    cout << "\nDazugehoeriger Eigenvektor: " << v;
}
```

```
C:\Windows\system32\cmd.ex
Eigenwert mit groesstem Betrag: 3.41421
Dazugehoeriger Eigenvektor: -0.500;  0.707; -0.500
```

```
/* ------------------------------------------------------ */
/* von-Mises-Vektoriteration in C                         */
/* ------------------------------------------------------ */
#include "matrix.h"
#include <stdio.h>
#define DIM 3

int main(void)
{
    double e, v[DIM];
    double a[DIM][DIM] =
    {
        { 2, -1,  0},
        {-1,  2, -1},
        { 0, -1,  2}
    };

    m_mises(DIM, a, &e, v);
    printf("Eigenwert mit groesstem Betrag: %f\n", e);
    printf("Dazugehoeriger Eigenvektor:");
    v_print(DIM, v);
    return 0;
}
```

```
C:\Windows\system32\cmd.exe
Eigenwert mit groesstem Betrag: 3.414214
Dazugehoeriger Eigenvektor:     -0.50       0.71      -0.50
```

Berechnung von Eigenwerten über das charakteristische Polynom

Nicht-symmetrische Matrizen besitzen im Allgemeinen komplexe Eigenwerte und Eigenvektoren. Diese können mit den bisher vorgestellten Funktionen bzw. Methoden allerdings nicht ermittelt werden.

Die Berechnung der komplexen Eigenwerte einer nicht-symmetrischen Matrix ist mit HMMatrix über eine Nullstellenbestimmung des charakteristischen Polynoms möglich. Vorsicht ist bei größeren Matrizen geboten und auch dann, wenn die Eigenwerte mit hoher Genauigkeit benötigt werden. Dieser Rechenweg ist numerisch nicht stabil!

Die folgenden Beispiele zeigen die Bestimmung der Eigenwerte einer 3x3-Matrix. Das erste Beispiel in der Programmiersprache C, das zweite Beispiel in C++. Die in den Beispielen verwendete Matrix besitzt zwei komplexe Eigenwerte $\lambda_{1,2} = 2 \pm 3i$ und einen reellen Eigenwert $\lambda_3 = 2$.

```c
/* ---------------------------------------------------- */
/* Berechnung komplexer Eigenwerte in C                 */
/* ---------------------------------------------------- */
#include "matrix.h"
#include <stdio.h>
#define DIM 3

int main(void)
{
   int i;
   double mat[DIM][DIM] = { { 2, -1, 2 }, { 1, 2, -2 }, { -2, 2, 2 } };
   double char_pol[DIM + 1];
   HM_COMPLEX eigval[DIM];

   /* Reelle und komplexe Eigenwerte berechnen und auf dem Bildschirm ausgeben */
   m_charpol(DIM, mat, char_pol);
   v_droots(DIM + 1, char_pol, eigval);
   for(i = 0; i < DIM; ++i)   /* Real- und Imaginärteil der Eigenwerte ausgeben */
      printf("%.2f + (%.2fi)\n", MAT_REAL(eigval[i]), MAT_IMAG(eigval[i]));
   return 0;
}
```

```
2.00 + (-3.00i)
2.00 + (3.00i)
2.00 + (0.00i)
```

```cpp
// ----------------------------------------------------
// Berechnung komplexer Eigenwerte in C++
// ----------------------------------------------------
#include <iostream>
#include "matrix.hpp"

int main()
{
   using namespace std;
   using namespace HMMatrix;

   Matrix mat(3);
   mat.Row(0) =  2, -1,  2;
   mat.Row(1) =  1,  2, -2;
   mat.Row(2) = -2,  2,  2;

   // Reelle und komplexe Eigenwerte berechnen und auf dem Bildschirm ausgeben
   auto char_pol = mat.CharPol();
   auto eigenval = char_pol.Roots();
   for(auto it = eigenval.begin(); it != eigenval.end(); ++it)
   {
      complex<double> e = *it;
      cout << e << "\n";
   }
}
```

```
(2,-3)
(2,3)
(2,0)
```

Anhang C: Parallelisierung mit OpenMP

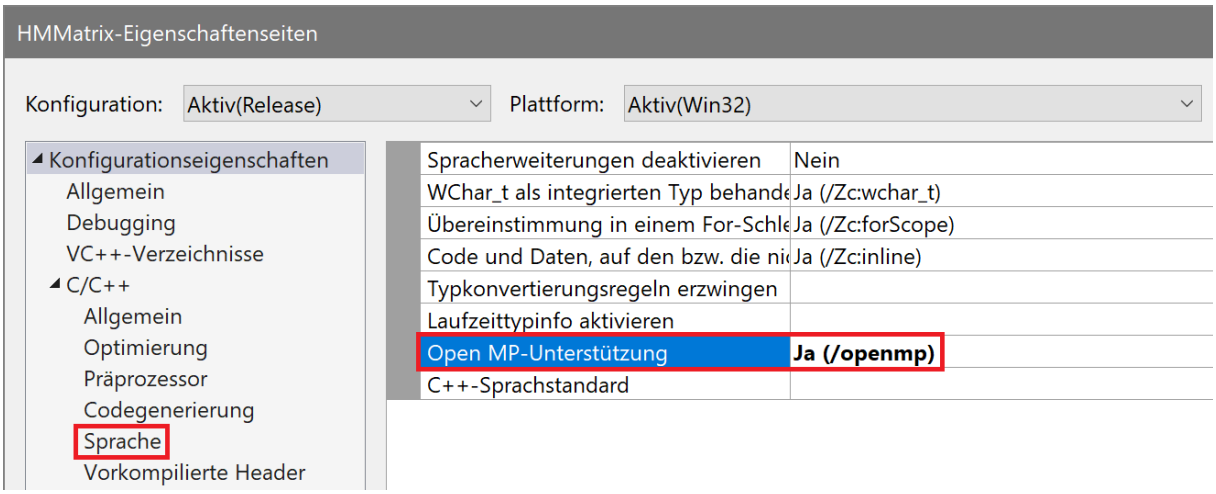
Im Sommer 2017 wurden verschiedene Berechnungsfunktionen hinsichtlich der Ablaufgeschwindigkeit optimiert. Zum Beispiel die Matrix-Multiplikation und die Berechnung des Skalarprodukts.

Um die Matrix-Multiplikation von größeren Matrizen (Dimension > 50) weiter zu beschleunigen, kann HMMatrix zudem mit OpenMP-Unterstützung übersetzt werden. Dadurch wird innerhalb der Multiplikationsfunktion eine Berechnungsschleife parallelisiert und auf mehrere Prozessorkerne verteilt.

Unabhängig davon gilt immer: Wenn eine hohe Berechnungsgeschwindigkeit erforderlich ist, sollte das Projekt nicht im Debug- sondern im Release-Modus erstellt werden.

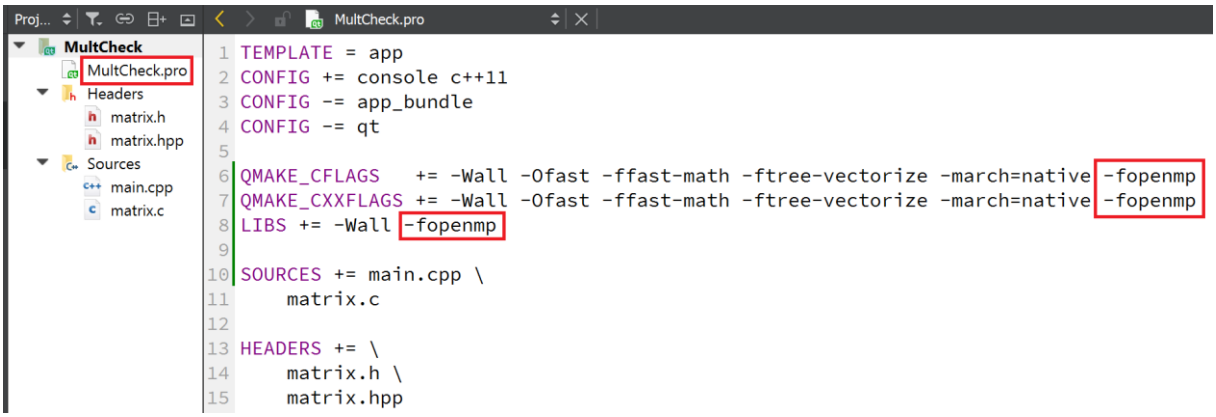
Microsoft Visual Studio

OpenMP wird über die Eigenschaftsseiten des Projekts aktiviert. Der entsprechende Eintrag befindet sich im Bereich C/C++ → Sprache:



Qt Creator

Über die Projektdatei (.pro-Datei) können zusätzliche Aufrufparameter an Compiler und Linker übermittelt werden. Die folgende Abbildung zeigt, wie der Parameter „-fopenmp“ gesetzt wird, welcher sowohl beim GNU C-/C++-Compiler (unter Microsoft Windows auch als „MinGW“ bekannt) als auch bei Clang die OpenMP-Unterstützung aktiviert:



Anhang D: Kontakt, Lizenz

Tilman Küpper
tilman.kuepper@hm.edu

Hochschule München
Fakultät für Maschinenbau, Fahrzeugtechnik, Flugzeugtechnik
Dachauer Straße 98 b
D-80335 München

http://kuepper.userweb.mwn.de/

Die Funktions- und Klassenbibliothek HMMatrix darf unter Beachtung der folgenden Lizenzbedingungen frei verwendet und weitergegeben werden: