Institut für Wissenschaftliches Rechnen
Technische Universität Braunschweig
Prof. Hermann G. Matthies, Ph. D.
Dr. Elmar Zander

Sommersemester 2010
14. November 2014

Weiterführendes Programmieren

Finden von Fehlern und Entwickeln eines Programmes für die LU-Zerlegung

Aufgabenblatt 2

Übung 1: *Suchen*

Diese Aufgabe muss im Zusammenhang mit der letzten Aufgabe auf dem ersten Aufgabenblatt gelöst werden. Häufig sortiert man Listen aus einem ganz bestimmten Grund, nämlich um schneller bestimmte Einträge in der Liste finden zu können. In dieser Aufgabe soll ein Programm `contains` entwickelt werden, das zwei Listen einliest und zu jedem Eintrag in der zweiten Liste eine Suche in der ersten Liste durchführt. Dabei soll das Programm kontrollieren, ob jeder Eintrag aus der zweiten Liste in der ersten Liste vorhanden ist.

Dazu sollen Sie zwei verschiedene Suchalgorithmen implementieren, die lineare und die binäre Suche, welche im Folgenden als Pseudocode gegeben sind:

```
linear_search(data,search_number)
{
  for every Number in data
    if(search_number == Number)
      return Position of number in data
  return -1
}
```

Listing 1: Pseudocode zur linearen Suche

Die lineare Suche kann sowohl auf unsortierten Daten als auch auf sortierten Daten angewendet werden. Die binären Suche kann nur auf sortierte Daten angewendet werden und ist eine Abwandlung der Suche in einem Binärbaum. Der Baum wird nicht explizit erzeugt, sondern entsteht durch die wiederholte Teilung des Vektors `daten`.

```
binary_search(data,search_number)
{
  left   = 0
  right  = n-1
  while (right > left)
    middle = (right+left)/2
    if (search_number == data[middle])
        return middle
    else
        if (search_number > data[middle])
            left = middle+1;
        else
            right = middle-1;

```

```
  if search_number == data[left]
    return left
  else
    return -1
}
```

Listing 2: Pseudocode zur binäre Suche

Zum Suchen in einem Array von Zahlen implementieren Sie bitte beide Algorithmen jeweils als Funktion.

```
int linear_search(int search_number, int n, int data[]);
int binary_search(int search_number, int n, int data[]);
```

Als Eingabe soll die zu suchende Zahl, die Anzahl der in Daten gespeicherten Zahlen, und schließlich ein Vektor mit den Daten gegeben werden. Als Resultat soll die Position der Suchzahl in den Daten zurückgegeben werden, falls sie vorhanden ist, und `-1` sonst.

Aufgabe a) Schreiben Sie ein Programm `contains`, das zwei Listen aus zwei Dateien einliest und die oben beschriebene Funktionalität zur Verfügung stellt. Es soll wieder ein Kommandozeilenparameter zur Verfügung gestellt werden, mit dem ein Algorithmus selektiert werden kann. □

Aufgabe b) Schätzen Sie die Komplexität der linearen Suche und der binären Suche in Abhängigkeit von der Eingabegröße n ab. Wann ist die lineare Suche geeignet, wann die binäre? □

Aufgabe c) Bauen Sie eine Zeitmessung (wieder mit dem Parameter `-t`) in das Programm ein und messen Sie die Zeit, die für 100,1000,1500,2000,2500... Suchoperationen benötigt wird. Stellen Sie die Messergebnisse erneut graphisch dar und stellen sie die Ausführungszeit für beide Algorithmen gegenüber. □

Zweidimensionale Felder – Matrix

Auf diesem Aufgabenblatt werden Algorithmen implementiert, die ein zweidimensionales Array benötigen. Dazu werden wir zu Beginn kurz einige C-Codefragmente diskutieren, um den Umgang mit solchen Konstrukten klar zu machen.

```
double  array[2];
```

Listing 3: Einfaches statisches Array

```
int n=2;
double* array=(double*)malloc(sizeof(double)*n);
...
free(array);
```

Listing 4: Einfaches dynamisches Array

Von dem letzten Aufgabenblatt wissen Sie, dass die beiden Definitionen in den Listings 3 und 4 das gleiche Ergebnis haben, also für den weiteren Verlauf eines Programmes keinen Unterschied machen. Die Definition in Listing 3 hat den großen Nachteil, dass sie ein Array statischer Größe anlegt; die Größe des erzeugten Arrays ist immer zwei. Die Größe des Arrays in Listing 4 ist dagegen über eine Variable (`n`) steuerbar.
Will man nun ein zweidimensionales Array von dynamischer Größe erzeugen, so wird der Aufwand dafür nochmals größer.

```
1 double  array[5][5];
```

Listing 5: Zweidimensionales statisches Array

```
1 int n=5, i;
2 double** array=(double**)malloc(sizeof(double*)*n);
3 array[0]=(double*)calloc(sizeof(double),n*n);
4 for(i=1; i<5; i++)
5   array[i] = array[i-1] + n;
6 ...
7 free(array[0]);
8 free(array);
```

Listing 6: Zweidimensionales dynamisches Array

Der Code in Listing 6 zeigt diese Komplikation, denn es muss zunächst Speicher für die Zeiger auf die eigentlichen Daten vorbereitet werden und diese dann mit entsprechenden Zeilenanfangszeigern "befüllt" werden. Dieser Code in Listing 6 erzeugt tatsächlich beliebig große quadratische Matrizen gesteuert über den Parameter `n`.

Übung 2: *Dynamische 2-dimensionale Arrays*

Das soeben beschriebene Vorgehen zur Erzeugung eines Datenfeldes der Größe $n \times n$ soll nun als Funktionen umgesetzt werden. Dazu geben wir Ihnen wieder die erforderliche Header-Dateien vor:

```
1 #ifndef LINSTRUCT__H
2 #define LINSTRUCT__H
3 double* create_vector(int size);
4 void free_vector(double*);
5 double** create_square_matrix(int size);
6 void free_square_matrix(double**);
7 #endif
```

Listing 7: lin_struct.h

Aufgabe a) Kommentieren Sie die Header-Datei, so dass man versteht, was die einzelnen Funktionen anbieten. □

Aufgabe b) Implementieren sie die Funktionen, die in der Headerdatei definiert sind in einer Datei mit Namen `lin_struct.c`. □

Aufgabe c) Schreiben Sie ein Testprogramm, das ein Array anlegt und es benutzt. □

Aufgabe d) Schreiben Sie ein Makefile in dem auch eine Bibliothek `liblin_struct.a` erzeugt wird.

Bemerkung: `Make` bietet für die Erzeugung von Bibliotheken ebenfalls eine eingebaute Regel an, die wie folgt benutzt wird:

```
liblin_struct.a:liblin_struct.a(matrix_free.o create_square_matrix.o)
```

Listing 8: Exemplarische Zeile eines Makefiles zur Definition einer Bibliothek

Die Zeile in Listing 8 liest sich etwa so: *Die Bibliothek `liblin_struct.a` enthält `matrix_free.o` und `create_square_matrix.o`* Um solche Bibliotheken zu nutzen muss man sie am Kompiler mit `-l` angeben. Mehr dazu erfahren Sie an geeigneter Stelle.

□

In der folgenden Aufgabe werden Sie die `lin_struct` Bibliothek praktisch einsetzen.

Übung 3: *Fehler in einem Programm finden*

Da Matrizen positive Dimensionen haben müssen, kann die Funktion `create_square_matrix` nach einem Aufruf mit size$<= 0$ keine verwendbare Matrix zurückliefern und daher insgesamt die Programmausführung im Allgemeinen nicht sinnvoll fortgesetzt werden.

```
#include <assert.h>
double** create_square_matrix(int size)
{
  ...
  assert(size>0);
  ...
}
```

Listing 9: Anwendung einer Assertion

Aufgabe a) Schreiben Sie ein Programm `matrix_test`, das die nach Listing 9 modizizierte Funktion `create_square_matrix` mit dem Argument `-1` aufruft. □

Wenn Sie das Programm nun kompilieren und ausführen, dann erscheint der gewollte Fehler:

```
matrix_test: create_square_matrix.c:12:
                create_square_matrix : Assertion 'size>0' failed.
```

Der Fehler ist gefolgt von einem `Aborted (core dumped)`. Der letzte Satz (`core dumped`) besagt, dass das Betriebssystem ein Speicherabbild des Programms erzeugt hat. Dieses Speicherabbild befindet sich in der Datei `core`. Das Speicherabbild kann man dazu benutzen, um den Fehler zu finden, der bei der Programmausführung aufgetreten ist. Dazu müssen Sie die Datei in einen sogenannten Debugger (zu Deutsch *Fehlerfinder*) laden. Dazu wollen wir in der nächsten Aufgabe den `gdb` (GNU Debugger) benutzen.

Aufgabe b) Starten Sie den GNU Debugger mit

```
gdb matrix_test core
```

Sie sehen, dass ein Programm startet, welches mehrere Meldungen ausgibt. Der Debugger wird uns dabei helfen herauszufinden, was passiert ist; in welcher Funktion der Fehler aufgetreten ist. Geben Sie dazu

`where` ein. □

Bemerkung: Die Ausgabe liest sich rückwärts, `#0 0xffffe430 __kernel_vsyscall` war der letzte Aufruf Ihres Programms, bevor es beendet wurde. Als erstes wurde die `main`-Funktion auf aufgerufen (`#5 0x08048488 in main ()`). Sie sehen im Debugger nun die Aufrufhierarchie (den Funktionsstack) Ihres Programms zum Zeitpunkt des Fehlers. Die `#0` besagt, dass dieser Funktionsaufruf als letztes auf den Funktionsstack gelegt wurde. Die Zahl `0xffffe430` gibt die Adresse der Funktion im Speicher an (diese kann sich bei verschiedenen Compilern verändern) und `__kernel_vsyscall` ist der symbolische Name der Funktion. Sie sehen, was zu erwarten war, denn

```
#4  0x080484ec in create_square_matrix ()
```

besagt, dass `main` die Funktion `create_square_matrix` aufgerufen hat und dann

```
#3  0xf7dbc5be in __assert_fail () from /lib/xxx/cmov/libc.so.6
```

die Funktion hinter dem symbolischen Namen `__assert_fail` aufgerufen wurde.

Aufgabe c) Um mehr zu erfahren müssen Sie das Programm erneut kompilieren, diesmal mit einem besonderen *Flag*, die den Compiler sogenannte Debugsymbole erzeugen lässt. Beenden Sie dazu den Debugger mit der Eingabe von [q] und [enter]. Um diese Symbole zu erzeugen müssen Sie das Projekt mit `CFLAGS=-g` erneut kompilieren. Starten Sie also `make CFLAGS=-g`. Wenn `make` jetzt die Ausgabe `make: 'matrix_test' is up to date.` produziert, dann hat es zwar eigentlich recht, allerdings wollen wir trotzdem, dass das Projekt neu übersetzt wird. Daher löschen Sie einfach die Datei `create_square_matrix.o`, kompilieren Sie dann erneut und führen Sie das erzeugte Programm aus. □

Aufgabe d) Der Debugger hat viele Möglichkeiten, die Sie bis jetzt noch nicht gesehen haben. Sie können z.B. in die Funktionen hineingucken und einzelne Werte von Variablen anschauen. Geben Sie `up` ein; der Debugger geht mit diesem Befehl im Stack nach oben.
Der `gdb`-Befehl `list` gibt jetzt den Programmcode in der Umgebung des Fehlers aus. Eine Variable können Sie sich mit dem Befehl `print` ausgeben lassen; z.B.

```
print size
```

produziert bei mir die Ausgabe `$1 = -1`. □

Aufgabe e) Gehen Sie auf die Seite `http://www.digilife.be/quickreferences/quickrefs.htm` und drucken Sie die "GDB Quick Reference". □

Bemerkung: Bis jetzt haben Sie den GNU Debugger eingesetzt um das Speicherabbild eines bereits abgestürzten Programmes zu analysieren. Der Debugger kann aber auch dazu eingesetzt werden, das Verhalten laufender Programmen zu analysieren.

Übung 4: *Fehler in einem laufenden Programm finden*

In der letzten Aufgabe haben wir uns einzig mit der Analyse eines abgestürzten Programms beschäftigt. Jetzt wollen wir *unter die Haube eines laufenden Motors* gucken. Dazu brauchen wir zunächst wieder ein geeignetes Forschungsobjekt:

```
1 int main()
2 {
3   int i;
4   for(i=0;i!=1;i=i+2)
5     i=i-2;
6 }
```

Listing 10: Programmcode – `loop.c`

Bemerkung: Das Programm in Listing 10 sieht zwar wirklich komisch aus, aber es ist ein legales C Programm, welches sich von allein nie beendet, sondern immer `i` um 2 erhöht und sofort wieder zurück setzt.

Aufgabe a) Kompilieren Sie das Programm mit dem *Flag* für das *Debugging* `make CFLAGS=-g loop` und starten Sie den GNU Debugger mit diesem Programm `gdb ./loop`. Der Debugger gibt wieder einige Meldungen aus und wartet dann auf Ihre Eingabe. Geben Sie `run` ein, um die Ausführung des Programms `loop` im Debugger zu starten. Da das Programm ja nie aufhören wird zu laufen, müssen Sie es unterbrechen; das erreichen Sie mit der Tastenkombination [strg+c]. Jetzt erscheint folgender Text:

```
1 Program received signal SIGINT, Interrupt.
2 0x0804838e in main () at loop.c:4
3 4         for(i=0;i!=1;i=i+2)
```

oder

```
1 Program received signal SIGINT, Interrupt.
2 main () at loop.c:5
3 5           i=i-2;
```

Welcher der beiden Texte erscheint, hängt davon ab, zu welchem Zeitpunkt Sie das Programm in seiner Ausführung unterbrechen.

Sie sind mit dem Debugger nun direkt an der Stelle der Ausführung. Das Programm befindet sich, wie zu erwarten war, in Zeile 4 oder 5 des Codes. Lassen Sie sich den Wert der Variable `i` mit `print i` ausgeben, dann lassen Sie das Programm mit `next` einen Schritt machen und lassen Sie sich den Wert von `i` erneut ausgeben. Macht die Ausgabe für Sie Sinn? Warum ist der Wert entweder `-2` oder `0`? □

Bemerkung: An diesem einfachen Beispiel können sie sehen, dass Programme manchmal nicht machen, was man intuitiv denkt. Merken Sie sich, dass der Debugger ihr Freund ist.

Übung 5: *LU-Zerlegung*

In dieser Aufgabe soll nun etwas mehr mit Matrizen gerechnet werden. Es wird ein Verfahren zur Lösung von linearen Gleichungssystemen implementiert, das vielseitiger ist als die simple Gauss-Elimination.

Dieses Verfahren nennt sich LU-Zerlegung. Weitere Informationen zur LU-Zerlegung finden Sie in allen Lehrbüchern zur numerischen Mathematik.

Bemerkung: Wir werden Matrizen mit großen lateinischen Buchstaben bezeichnen und ihre Einträge mit den entsprechenden kleinen Buchstaben. Zum Beispiel ist $a_{1,3}$ das Element von A in der ersten Zeile und dritten Spalte.

Definition: Sei $n \in \mathbb{N}$. Wir sagen, eine $n \times n$- Matrix ist *LU*-**zerlegbar**, falls es eine untere Dreiecksmatrix L und eine obere Dreiecksmatrix U gibt mit den Eigenschaften

$$\begin{aligned} (i) \quad & l_{ii} = 1 \quad \text{für } i = 0, \ldots, n-1, \\ (ii) \quad & u_{ii} \neq 0 \quad \text{für } i = 0, \ldots, n-1, \\ (iii) \quad & A = LU \end{aligned}$$

Beispiel:

$$\begin{pmatrix} 1 & 1 & 2 \\ 2 & 3 & 1 \\ 3 & -1 & -1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & -4 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 2 \\ 0 & 1 & -3 \\ 0 & 0 & -19 \end{pmatrix}$$

Da die Diagonalelemente der in einer LU-Zerlegung vorkommenden unteren Dreiecksmatrizen alle 1 sind, braucht man sie nicht explizit abzuspeichern. Daher ist eine vernünftige Datenstruktur zur Speicherung einer LU-Zerlegung wieder eine Matrix, die wie folgt aussieht:

$$\begin{pmatrix} u_{0,0} & u_{0,1} & u_{0,2} & u_{0,3} & u_{0,4} & \ldots & u_{0,n-1} \\ l_{1,0} & u_{1,1} & u_{1,2} & u_{1,3} & u_{1,4} & \ldots & u_{1,n-1} \\ l_{2,0} & l_{2,1} & u_{2,2} & u_{2,3} & u_{2,4} & \ldots & u_{2,n-1} \\ l_{3,0} & l_{3,1} & l_{3,2} & u_{3,3} & u_{3,4} & \ldots & u_{3,n-1} \\ l_{4,0} & l_{4,1} & l_{4,2} & l_{4,3} & u_{4,4} & \ldots & u_{4,n-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ l_{n-1,0} & l_{n-1,1} & l_{n-1,2} & l_{n-1,3} & l_{n-1,4} & \ldots & u_{n-1,n-1} \end{pmatrix}$$

Als Grundgerüst zur Programmierung kann das folgende Schema verwendet werden:

for $i = 0, \ldots, n-1$
  $l_{i,i} = 1$
for $j = 0, \ldots, n-1$
  for $i = 0, \ldots, j$
    $u_{i,j} := a_{i,j} - \sum_{k=0}^{i-1} l_{i,k} u_{k,j}$
  for $i = j+1, \ldots, n-1$
    $l_{i,j} := \frac{1}{u_{j,j}} \left(a_{i,j} - \sum_{k=0}^{j-1} l_{i,k} u_{k,j} \right)$

Bemerkung: Wenn sie sich das obige Schema genauer ansehen, und einige Schritte von Hand durchrechnen, werden Sie feststellen, dass die Einträge u_{ij} und l_{ij} bereits berechnet sind, wenn sie benutzt werden. Ausserdem wird jeder Eintrag a_{ij} nur einmal benutzt. Daher ist es möglich, die zerlegte Matrix in dem selben Speicherbereich abzuspeichern wie die Originalmatrix A. Eine solche Zerlegung mit Abspeicherung in der selben Stelle nennt man im Englischen “in place”. Besonders bei großen Matrizen ist es wichtig, keinen Speicherplatz für eine zusätzliche Lösungsmatrix reservieren zu müssen.

Wir geben Ihnen im Folgenden eine Funktion `lu_decomp` vor, welche das beschriebene Verfahren (fehlerhaft) implementiert. An das von Ihnen korrigierte `lu_decomp` kann eine Matrix A übergeben werden, welche nach dem Aufruf dann die LU-Zerlegung von A enthält. Die Funktion `lu_decomp` erkennt, ob eine gegebene Matrix A singulär sein könnte. Eine definitive Antwort wird erst die Verbesserung durch eine Pivotsuche in der nächsten Aufgabe geben.

Aufgabe a) Laden Sie aus `http://www.wire.tu-bs.de/ADV/files/lu_decomp_error/` die C-Funktion `lu_decomp` herunter, welche eine LU-Zerlegung einer $n \times n$-Matrix A zu berechnen **soll**. Die angegebene Datei hat Fehler. Es ist nun Ihre Aufgabe mit den erlernten *debugging* Techniken, diesen Fehler zu finden und zu korrigieren.
Testen Sie Ihre Programmkorrektur mit der oben angegebenen Beispielzerlegung. □

Lineare Gleichungssysteme $Ly = b$, für welche L eine untere Dreiecksmatrix und $b \in \mathbb{R}$ ist, sind besonders leicht zu lösen:

$$\begin{pmatrix} 1 & & & & & \\ l_{1,0} & 1 & & & & \\ l_{2,0} & l_{2,1} & 1 & & & \\ l_{3,0} & l_{3,1} & l_{3,2} & 1 & & \\ \vdots & \vdots & \vdots & \vdots & \ddots & \\ l_{n-1,0} & l_{n-1,1} & l_{n-1,2} & l_{n-1,3} & \cdots & 1 \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{n-1} \end{pmatrix}$$

Wir sehen aus der ersten Zeile, dass $y_0 = b_0$ ist. Nachdem man den Wert für y_0 in die zweiten Zeile eingesetzt hat, erhält man sofort $y_1 = b_1 - l_{1,0}b_0$. Jetzt können die Werte für y_0 und y_1 in die dritte Zeile eingesetzt werden, um y_2 zu berechnen usw. Diese Methode funktioniert analog für Gleichungssysteme mit oberen Dreiecksmatrizen.

Sei nun $A = LU$ die LU-Zerlegung der Matrix A. Soll das lineare Gleichungssystem $Ax = b$ gelöst werden, so kann man zunächst das Gleichungssystem $Ly = b$ und anschließend das System $Ux = y$ lösen.

Aufgabe b) Man schreibe eine C-Funktion `lu_solve`, um ein lineares Gleichungssystem $Ax = b$ mit n Unbekannten zu lösen. An `lu_solve` sollen die LU-Zerlegung der Matrix A und der Vektor b übergeben werden. Nach dem Aufruf von `lu_solve` soll b die Komponenten des Lösungsvektors x enthalten (in-place Lösung). □

Als Grundgerüst für das Programm soll folgende Rekursion benutzt werden:

$$y_0 := b_0 \text{und } y_i := b_i - \sum_{j=0}^{i-1} l_{i,j} y_j \text{für } i = 1, \ldots, n-1,$$

sowie

$$x_n - 1 := \frac{y_n - 1}{u_{n-1,n-1}} \text{und } x_i := \frac{1}{u_{i,i}} \left(y_i - \sum_{j=i+1}^{n-1} u_{i,j} x_j \right) \text{for } i = n-2, \ldots, 0.$$

`lu_solve` kann dadurch getestet werden, dass man das Gleichungssystem

$$\begin{pmatrix} 1 & 1 & 2 \\ 2 & 3 & 1 \\ 3 & -1 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 4 \\ 6 \\ 1 \end{pmatrix}$$

löst. (Die Lösung ist: $x = y = z = 1$.)

Aufgabe c) Testen Sie auch das folgende Gleichungssystem. Hier wird es nun schief gehen, die Lösung dieses Problems wird in der nächsten Aufgabe behandelt: *Beispiel:*

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 10 \\ 20 \\ 30 \end{pmatrix} \tag{1}$$

□

Übung 6: *Pivotierung*

Das oben beschriebene Schema setzt voraus, dass die Diagonalelemente der Matrix U nicht 0 sind. Dies war im letzten Beispiel nicht der Fall, wodurch das Verfahren daher scheitern musste. Das Verfahren kann erweitert werden, so dass für alle regulären $n \times n$-Matrizen eine Zerlegung möglich wird. Grundlage hierfür ist der folgende Satz:

Satz: Sei $n \in \mathbb{N}$ und A eine reguläre $n \times n$- Matrix mit reellen Einträgen. Dann gibt es eine Permutationsmatrix P, eine untere Dreiecksmatrix L, deren Diagonalelemente alle 1 sind, und eine obere Dreiecksmatrix U, so dass gilt:

$$PA = LU.$$

Bei gegebenem P sind L und U eindeutig bestimmt.

Aufgabe a) Man modifiziere `lu_decomp` wie folgt: Beim Berechnen der LU-Zerlegung wird ein Zeilenindexvektor `piv` berechnet und zurückgegeben. In diesem Vektor wird die Permutation der Zeilenindizes gespeichert.

Das entsprechende Schema zur Zerlegung der Matrix sieht wie folgt aus:

for $i = 0, \ldots, n-1$
  piv[i]=i
for $j = 0, \ldots, n-1$
  for $i = 0, \ldots, j$
    $u_{piv[i],j} := a_{piv[i],j} - \sum_{k=0}^{i-1} l_{piv[i],k} u_{piv[k],j}$
  for $i = j+1, \ldots, n-1$
    $u_{piv[i],j} := a_{piv[i],j} - \sum_{k=0}^{j-1} l_{piv[i],k} u_{piv[k],j}$
  determine index p of Pivot
  swap(piv[j],piv[p])
  for $i = j+1, \ldots, n-1$
    $l_{piv[i],j} := \frac{1}{u_{piv[j],j}} u_{piv[i],j}$

Dabei wird das Pivot-Element so bestimmt, dass es der betragsmäßig größte Eintrag $u_{k,j}$ mit $k = j, \cdots, n-1$ ist. Dieses Vorgehen vermeidet, dass die Zeilen in der Matrix (durch teures kopieren) vertauscht werden müssen.

Bemerkung: Mit dem neuen Verfahren erhält man folgende Zerlegung der Beispielmatrix mit `piv[0]=2, piv[1]=1, piv[2]=0` :

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & 1 & 2 \\ 2 & 3 & 1 \\ 3 & -1 & -1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ \frac{2}{3} & 1 & 0 \\ \frac{1}{3} & \frac{4}{11} & 1 \end{pmatrix} \begin{pmatrix} 3 & -1 & -1 \\ 0 & \frac{11}{3} & \frac{5}{3} \\ 0 & 0 & \frac{57}{33} \end{pmatrix}$$

□

Aufgabe b) Verändern Sie nun die Funktion `lu_solve` so, dass sie mit der um die Pivotierung erweiterten LU-Zerlegung arbeitet. Dazu muss sowohl der erste Zugriffsindex auf die Matrix als auch der Zugriffsindex auf die rechte Seite über die Permutation erfolgen. Da das so erhaltene Ergebnis permutiert ist, muss zur Erzeugung der korrekten Lösung noch ein weiterer Vektor zum Umkopieren benutzt werden. Testen Sie wieder mit den oben angegebenen Testgleichungen. □

Aufgabe c) Testen Sie nun Ihre Routine zum Lösen von linearen Gleichungssystemen anhand der Gleichung $Ax = b$ mit einer 10×10 Matrix mit zufälligen Einträgen. Kontrollieren Sie das Ergebnis, indem Sie die Matrix mit dem Ergebnisvektor multiplizieren und mit der rechten Seite vergleichen. □