Programmierpraktikum im WS 2010/2011

Denny Otten, V5-142, dotten@math.uni-bielefeld.de

4. Datentypen und Datenstrukturen

1 Elementare Datentypen

Numerische Datentypen

Standardmäßig interpretiert Matlab eine Zahl als `double` (d.h. als eine Fließkommazahl von 64 Bit Länge), für Details dazu siehe die Online-Hilfe: `>> helpwin` und dann
`Matlab->User Guide->Programming Fundamentals->Classes (Data Types)->Numeric Classes`
Es gibt aber auch andere Datentypen wie
`int8, uint8, int16, uint16, int32, uint32, int64, uint64` für ganze Zahlen,
`single` als weiteren Fließkommadatentyp mit geringerem Speicheraufwand,
`logical` als Boolschen Datentyp und
`char` für Strings (Texte).
In numerischen Berechnungen werden meist nur `double`,`char` und `logical` benutzt.

Aufgabe 1. *Schaue Dir in der Online Hilfe (*`>> helpwin`*) unter* `Matlab->Functions->Programming->Data Types` *die Dokumentation zu den oben erwähnten Datentypen an. Erstelle Variablen der verschiedenen Typen und vergleiche mit* `whos` *oder im Workspace-Browser den Speicherbedarf.*

Ein weiterer numerischer Datentyp sind die komplexen Zahlen. Matlab kann mit komplexen Zahlen ganz einfach rechnen, es ist dabei nichts weiter zu beachten, als das, was man sowieso schon wegen der komplexen Zahlen beachten muss (z. B. auf welchem Zweig des Logarithmus man ist,...). Wie Du schon gesehen hast kannst Du sie auf verschiedene Arten erzeugen:

```
>> re = 0.7; im = 1.4;
>> z = re + im*i
>> z = re + im*j
>> z = 0.7 + 1.4i
>> z = complex(re,im)
>> z = re + im*sqrt(-1)
```

Beachte, dass man aufpassen muss, ob man `i` oder `j` irgendwo vorher umdefiniert hat (z. B. in einer Schleife!).

Zeichenketten (Strings)

Strings (bzw. den Datentyp `char`) kennt ihr schon von den vorherigen Übungsblättern. Ein String steht zwischen Hochkommata ':
`>> s1 = 'Hallo'` ist eine der einfachsten Methoden einen String zu erzeugen. Um Strings aneinander zu hängen (engl.: *concatenate*) kann man sie einfach in ein Array schreiben:
`>> text = [s1, ' Du Da.']`\\Man kann sie auch mehrzeilig machen, dafür muss man aber beachten, dass es sich wieder um Matrizen handelt, also *jede Zeile und jede Spalte gleich lang sein muss*. Folgendes geht also nicht:
`>> text2 = ['Hallo';'Du Da.']`, da die beiden Zeilen nicht gleich lang sind. Du kannst hier entweder Leerzeichen von Hand einfügen (also `text2=['Hallo ';'Du Da.']`) oder aber den Befehl `char` benutzen (siehe `help char`).
Wie ihr schon bei den Kontrollstrukturen gesehen habt, kann man strings in `switch` Verzweigungen benutzen.
Besonders wichtig sind Strings für die Ausgabe. Dazu dienen die Funktionen `disp` zusammen mit `sprintf` oder `fprintf`. Um Zahlen mit `disp` darzustellen, müssen sie erst mit der Funktion `num2str` in Strings umgewandelt werden:

```
>> erg = [102 97 108 115 99 104];
>> disp(['Die Ergebnisse sind: ' num2str(erg)]); % so ist es richtig
>> disp(['Die Ergebnisse sind: ' erg]);          % beliebter Fehler, erg ist kein String!
```

Die ensprechende Anweisung mit `fprintf` sieht so aus, dabei wird für jeden Wert in `erg` eine Zeile geschrieben:

```
>> fprintf('Das Ergebnis ist: %g\n', erg);
```

(Dabei steht `%g` für kompakte Fixpunkt Notation und `\n` für einen Zeilenumbruch, siehe auch `doc fprintf`.)
Achtung: Der ganz ähnlich funktionierende Befehl `sprintf` erzeugt nur einen String und keine Ausgabe, diese muss dann von `disp` übernommen werden. Oft ist `disp` die einfachste Möglichkeit der Ausgabe. Bei komplizierten Formatierungen oder Ausgaben, die viele `num2str` Statements enthalten, ist `fprintf` oft übersichtlicher.
Eine weitere häufig verwendete Funktion ist `strcmp`, die Strings miteinander vergleicht. Sie wird z.B. in `if`–Verzweigungen gebraucht:

```
tier = 'hund'; % oder 'katze' etc.
if strcmp(tier, 'hund')
  disp('Ein Hund');
end
```

Aufgabe 2. *Schaue Dir die Online-Dokumentation* `>> doc strings` *zu Strings an. Experimentiere mit dem Vergleich von Strings mit* `==` *und* `strcmp`*. Schaue Dir auch die zugehörige Hilfe an. (Vielmehr erfährt man unter*
`Matlab->User Guide->Programming Fundamentals->Classes (Data Types)->Characters and Strings`*)*

2 Datenstrukturen (siehe auch `Matlab->Getting Started->Programming->Other Data Structures`)

Matrizen (siehe auch `Matlab->Getting Started->Matrices and Arrays->`)

Die wichtigste Datenstruktur in Matlab, die dem Programm auch seinen Namen gibt ist die Matrix. Außer eine Matrix direkt einzugeben, bietet Matlab verschiedene Funktionen zur Erzeugung von speziellen Matrizen an. Die wichtigsten sind `zeros, ones, rand, eye`. Zusammen mit dem `:`-Operator lassen sich damit viele Matrizen ohne Benutzung von `for`–Schleifen definieren, was häufig schneller und manchmal auch lesbarer ist.

```
>> M = [1:4; 16:-4:1; -5:2:2]
>> A = 10*(1:4)'*ones(1,5)+ones(4,1)*(1:5)
```

Auf die Größe von Matrizen (Zeilen, Spalten) kann mit `size` und die Länge von Vektoren mit `length` zugegriffen werden. *Was ergibt* `length` *angewandt auf eine Matrix?*

Aufgabe 3. *Erstelle eine* 4×2 *Matrix, deren Einträge alle gleich der Eulerschen Zahl* e *sind.*

Aufgabe 4. *Schreibe eine Funktion, die nach Eingabe der Spalten- und Zeilenzahl (*< 10*) eine Matrix derselben Form wie* `A` *oben mit Hilfe von* `for`*-Schleifen erzeugt.*

Zugriff auf Matrixelemente

Man kann auf Matrixelemente nicht nur über das Paar `(z, s)` (z=Zeilen, s=Spalten) sondern auch über einen linearen Index `l=(s-1)*Zeilen + z` zugreifen. Diesen Index erhält man, indem man die Elemente spaltenweise durchzählt.

```
>> A(3:6)      % die Eintraege 3 bis 6 der Matrix A bzgl. des linearen Indizes
```

Aufgabe 5. *Schreibe eine Funktion* `subtoind` *die aus einem Paar von Indizes* (z, s) *den linearen Index* l *berechnet und zurückliefert. Überlege, welcher Eingabeparameter außer den Indizes* `z, s` *noch benötigt wird. Teste die Funktion, indem Du in einer Schleife die Ausgabe für alle Indizes* $z, s < 10$ *mit der Ausgabe der Matlab Funktion* `sub2ind` *vergleichst.*
Erstelle auch die entsprechende Umkehrfunktion `indtosub` *und vergleiche diese mit* `ind2sub`*. (***Zusatzaufgabe:** *Gestalte die Funktion so, dass nicht nur ein Paar* `(z, s)` *sondern mehrere (also ein Array von) Indizexpaaren eingegeben werden können.)*

Auf dem ersten Zettel hast Du schon Beispiele für den Zugriff mit `:` (colon) auf Teilbereiche von Matrizen kennengelernt. Matlab ist noch viel flexibler: der Zugriff kann ganz allgemein mit Indexarrays erfolgen. Dabei bleibt die Struktur des Indexarrays erhalten.

```
>> M = [1:4; 5:8; 9:12; 13:16]
>> M(1:end-1, :)
>> M(:, 2:3)
>> I=1:3
>> M(I, end:-1:end-2)
>> I=[3,5;7,9]  % ein Matrix-Array
>> M(I)
>> M(:)         % verwandelt M in einen Spaltenvektor
```

Man kann das auch gut dazu benutzen, um Zeilen oder Spalten einer Matrix zu vertauschen.
`>> M = M(:, [end, 1:end-1]);  % tauscht die letzte Spalte nach vorne`
Durch Zuweisung des leeren Arrays `[]` löscht man einzelne Zeilen oder Spalten.
`>> M(2, :) = [];   % loescht die 2. Zeile`
Man kann auch Arrays vom Datentyp `logical` benutzen um auf Matrixelemente zuzugreifen. Dabei wird das Array dann als Maske aufgefasst, die auf die Matrix gelegt wird (übrig bleiben dann nur noch die Einträge, die unter den Einsen liegen):

```
>> M = [1:3; 4:6; 7:9]
>> Mask = logical([1 0 1; 0 1 0;1 0 1]) % erzeugt eine logische Matrix
>> M(Mask)
>> I = logical([1,0,1])  % logisches Array erzeugen
>> J = logical([1,1,0])  % logisches Array erzeugen
>> M(I,J)
>> I = [1,0,1]           % das ist ein numerisches (kein logisches) Array!
>> M(I, :)               % Fehler, denn
>> class(I)              % I ist kein logisches, sondern ein double-Array!
>> I = logical(I)        % konvertiere I in ein logisches Array
>> islogical(I)          % ok, I ist ein logisches Array
>> M(I, :)               % jetzt geht's
```

Logische Arrays kann man auch durch logische Operationen erzeugen und die entsprechenden Indizes kann man mit `find` bestimmen.

```
>> M=rand(5)
>> L=(M<0.5)              % logisches Array erzeugen
>> M(L)
>> sum(L(:))              % Anzahl der Elemente, mit M<0.5
>> [Z, S]=find(M<0.5)     % Indizes, mit M<0.5
>> [Z, S]=find(L==true)   % das sollte das gleiche ergeben
>> for i=1:length(Z)
 M(Z(i),S(i))             % Z,S sind die Zeilen- und Spaltenindizes
end                       % siehe  help find
>> K=find(M<0.5)          % wenn man aber den linearen Index benutzt
>> M(K)                   % dann erhaelt man das gleiche wie M(L)
>> M(K) = pi              % setzt alle Elemente <0.5 auf pi
>> M(K) = 1:sum(L(:))     % was macht das?
```

Mit den Funktionen `any` und `all` kann man entscheiden, ob ein oder alle Elemente die Bedingung erfüllen. Das Ergebnis ist vom Datentyp logical.

```
>> any(M(:) < 0.5)       % ist ein Element <0.5 ?
>> b=all(M(:) < 0.5)     % sind alle Element <0.5 ?
>> islogical(b)
>> any(M < 0.5)          % Achtung, hier wird die Bedingung pro Spalte getestet
```

Aufgabe 6. *Für eine zufällige* 5×5*–Matrix* M *führe folgende Aktionen durch: Vertausche die zweite und die vierte Zeile; lösche die dritte Spalte; definiere die Matrix, die nur aus den vier Eckelementen von* M *besteht und schliesslich zähle, wieviele Einträge zwischen* 0.2 *und* 0.8 *liegen und gib diese Einträge aus.*

Für den Zugriff auf spezielle Bereiche einer Matrix stellt Matlab die Funktionen `diag, triu, tril` zur Verfügung.

```
>> M=rand(6)
>> d=diag(M)
>> u=diag(M,1)
>> l=diag(M,-1)
```

Aufgabe 7. *Finde heraus, was* `diag`, `triu` *und* `tril` *machen und teste die Funktionen an verschiedenen Beispielen.*

Aufgabe 8. *Schreibe eine* `for`*-Schleife, um die Matrix* $\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$ *zu erzeugen.*
Versuche nun, das gleiche unter Benutzung von `eye` *und* `fliplr` *zu erreichen (*`help fliplr`*). Geht das auch mit* `flipud`*?*

Funktionen von Matrizen

Matlab stellt eine Vielzahl von Funktionen zur Verfügung, deren Eingabe eine Matrix ist. Die wichtigsten sind `norm, sum, prod, min` und `max`.

Aufgabe 9.

- *Finde mit der Hilfe und durch Ausprobieren heraus, was die Befehle* `norm, sum, prod, min` *bewirken.*
- *Bestimme den minimalen/maximalen Eintrag einer mit* `rand` *zufällig erstellten Matrix.*
- *Erstelle mit* `magic(n)` *für verschiedene Grössen ein magisches Quadrat (*`help magic`*) und überprüfe die magischen Eigenschaften (**Hinweis:*** `sum, diag, fliplr, '` *sollten helfen).*
- *Finde heraus, was die Funktionen* `cumsum` *und* `cumprod` *tun. Benutze dazu zunächst Vektoren und anschließend Matrizen.*

Auf weitere Funktionen wie `eig, det`, die in der Linearen Algebra gebraucht werden, werden wir später noch eingehen.

Cell arrays

Vektoren und Matrizen sind Arrays deren Elemente alle denselben Typ haben. Mehr Flexibilität bieten `cell arrays`: der Typ jedes Elementes kann darin verschieden sein. Ansonsten funktionieren viele Dinge bei `cell arrays` genauso wie bei Matrizen. Ein cell array kann mit `c=cell(n,m)` vordimensioniert werden und der Zugriff erfolgt anders als bei Matrizen nicht mit runden Klammern `()` sondern mit geschweiften Klammern {}. (Mit `()` erhält man wieder ein Cell-Array!)

```
>> c = {}                                          % leeres cell array erzeugen
>> cM = cell(3,5)                                  % cell Array der Groesse 3x5 erzeugen
>> cM = { [1,2,3] , 'b'; cell(2,2), @(x) x.^3 }    % die Elemente haben verschiedenen Typ
>> c = {'rot','gruen','blau'}                      % ein cell Vektor
>> c = {'rot', 1, eye(3)}                          % ... noch einer
>> c{1} = 9                                        % setze erstes Element auf 9
>> cM{2,2}(5)                                      % fuehre die Funktion an 2,2 aus
>> cM{2,1}{2,2}=8                                  % schachteln ist kein Problem
>> celldisp(cM)                                    % gibt den Inhalt des cell-arrays aus
```

Wie bei Matrizen ist auch der Zugriff auf Bereiche mit `c{m:n}` möglich und die Größe wird mit `size` bestimmt. (Ausprobieren!) Eine schöne Möglichkeit der Visualisierung des Inhaltes eines cell arrays bietet die Funktion `cellplot`.

```
>> cellplot(c);
>> cellplot(cM);
```

Aufgabe 10. *Erstelle ein cell array von 5 Funktionshandles (Hinweis: erstelle die Funktionshandles entweder mit anonymen Funktionen oder als Handle auf eine vorhandene Funktion), sowie ein cell array von 5 Linienstilen (Hinweis: etwa* `'c+'`*, siehe dazu in der Online Hilfe (*`>> doc LineSpec`*) nach.)*
Benutze eine Schleife, um die Funktionen mit den entsprechenden Linienstilen mit `plot` *(Hinweis: siehe erstes Arbeitsblatt) zusammen in eine Grafik zu zeichnen.*

Schließlich ist es noch möglich eine Funktion auf jedes Element eines Cell-Arrays anzuwenden:

```
>> for i = 3:10
 c{i-2,1} = magic(i);           % erzeuge Zell-Array mit magischen Quadraten
end                             % Wieso muss man hier Cell-Arrays benutzen?
>> D=cellfun(@(M)sum(M(1,:)),c) % berechnet die Zeilensumme der ersten Zeile jeder Matrix
```

Strukturen

Eine besonders wichtige Datenstruktur, die sich - wie der Name schon sagt - zum Strukturieren von Daten eignet, ist die Struktur (`struct`). In ihr können verschiedene Daten, die zu einem Objekt gehören zusammengefasst werden. Zum Beispiel kann man verschiedene Daten über eine Person zusammenfassen:

```
% erzeuge eine Struktur person mit den Feldern name, alter, wohnort, lieblingsfunktion
>> person.name = 'Maria';
>> person.alter = 19;
>> person.wohnort = 'Bielefeld';
>> person.lieblingsfunktion = @(x) exp(x);
>> person       % Anzeige der Variablen person
>> disp(person) % Ausgabe des Inhalts der Variable person
>> person(2).name = 'Alex'; % erzeuge eine weitere person
>> disp(person) % jetzt wird nur noch die Datenstruktur angezeigt
>> person(1)    % greift auf die erste person zu
>> person(2)    % die zweite person
```

Strukturen werden häufig als Übergabeparameter an Funktionen benutzt um lange Parameterlisten zu vermeiden und zusammengehörige Informationen zu bündeln.
Strukturen und cell arrays sind insofern ähnlich, dass sie Variablen verschiedenen Typs zusammenfassen. Daher gibt es auch Funktionen um zwischen `struct` und `cell array` zu konvertieren: `help struct2cell`, `help cell2struct`

```
>> c = struct2cell(person)
>> s = cell2struct(c, {'name', 'alter', 'wohnort', 'lieblingsfunktion}, 1)
```

Aufgabe 11. *Erstelle (und teste) eine Funktion, die mit den Eingabeparametern* `Name` *und* `Alter` *eine Struktur* `Person` *zurückgibt, die aus den Feldern* `Name, Alter` *und* `Wohnort` *besteht. Dabei soll als Wohnort zufällig (Hinweis: benutze* `rand`*) mit gleicher Wahrscheinlichkeit* `'Muenchen'` *oder* `'Hamburg'` *zugewiesen werden.*

Aufgabe 12 (**Zusatzaufgabe für sehr Motivierte**). *Erstelle eine Simulation über das Umzugsverhalten von* `N` *Personen über J Jahre, die zufällig gemäß einer vorgegebenen Wahrscheinlichkeit zwischen München und Hamburg umziehen. Erstelle dazu mit der Funktion aus der vorherigen Aufgabe ein Strukurarray, das* `N` *Personen enthält (Zum Erzeugen bieten sich* `for`*–Schleifen, sowie ein* `cell`*–Array mit den Namen an). Gib nun Wahrscheinlichkeiten* `pHM, pMH` $\in [0, 1]$ *vor, mit der Personen von Hamburg nach München bzw. von München nach Hamburg ziehen.*
Simuliere nun das Verhalten in einer Schleife über `J` *Jahre, indem Du aufgrund des aktuellen Wohnortes und einer zufälligen Zahl (*`rand`*) gemäß der Umzugswahrscheinlichkeit* `pMH` *oder* `pHM` *entscheidest, welches der neue Wohnort ist. Berücksichtige auch das Altern indem Du das Feld* `Alter` *jeder Person in jedem Jahr aktualisierst.*
Du kannst auch einen Friedhof als weiteren Wohnort hinzufügen, der das Ableben simuliert. (Für gewöhnlich sollte man von dort nicht wieder zurückkommen können.)
Zum Visualisieren bietet sich zum Beispiel `pie` *an (siehe* `help pie`*).*

Zusatz: Mehrdimensionale Arrays

Außer Vektoren und Matrizen können auch höherdimensionale Arrays erzeugt werden:

```
>> A=zeros(3,4,5)  % 3-Tupel
>> A=rand(2,2,3)
>> A=ones(2,3,2,2) % 4-Tupel
```

Der Zugriff erfolgt hier wieder entweder über ein Tupel $(i_1, \ldots, i_p)$ oder einen linearen Index. Die Funktionen `sub2ind`, `ind2sub` ermöglichen auch in diesem Fall die Umrechnung zwischen beiden Indizierungsarten. In der Online Hilfe (`>> helpwin`) findest Du unter
`Matlab->Programming->Data Structures->Multidimensional Arrays`
eine schöne Visualisierung des Elementzugriffs bei mehrdimensionalen Arrays.
Genau das gleiche ist auch mit Cell Arrays möglich.

Aufgabe 13 (**Zusatzaufgabe**)**.** *Erweitere die Funktionen* `indtosub` *und* `subtoind`*, die Du in Aufgabe 5 erstellt hast, damit sie auch für höherdimensionale Arrays die Umwandlung des Indizes in den linearen Index und umgekehrt erledigen. (Hier bietet sich ein Blick in die Hilfe zu* `varargin` *an!)*

Zusatz: Dünnbesetzte Matrizen

Wenn man mit Matrizen rechnen möchte, die nur wenige Einträge ($< 5\%$) ungleich null haben, bietet es sich an, den Datentyp einer **Sparsematrix** zu benutzen. In diesem Datentyp werden nur die Nicht-Nullelemente zusammen mit ihren Indizes gespeichert. Solche Matrizen entstehen z.B. bei der numerischen Berechnung von Lösungen partieller Differentialgleichungen. Die Speicherung in einer Sparsematrix bedeutet zum einen eine große Einsparung an Hauptspeicher zum anderen werden so Matrixoperationen wie `+, -, .*, ./, *` nur für die Nicht-Nullelemente durchgeführt und sind damit wesentlich schneller. Außerdem stellt Matlab effiziente Algorithmen zur Lösung linearer Gleichungssysteme mit Sparsematrizen zur Verfügung.

Sparsematrizen erzeugen

Sparsematrizen können direkt durch Angabe der Zeilen-, Spaltenindizes und der Werte erzeugt werden (siehe: `help sparse`):

```
>> val = rand(5,1);     % 5 Nichtnull Werte
>> Z = 1:5;             % Zeilenindizes
>> S = [2,4,1,3,3];     % Spaltenindizes
>> sM = sparse(Z,S,val) % sM m x n Matrix wobei m = max(Z) and n = max(S).
>> fM = full(sM)        % Konversion Sparse->Full Matrix
>> sparse(fM)           % das geht auch umgekehrt: Full->Sparse
>> issparse(sM)         % sM ist eine Sparsematrix
>> issparse(fM)         % fM nicht
>> [m,n] = size(sM)     % wie ueblich funktioniert hier auch size
>> nnz(sM)              % Number of NonZeros
```

Wie Du gerade gesehen hast, lässt sich die Größe der Sparsematrix genau wie bei vollbesetzten Matrizen mit `size` bestimmen, die Anzahl Nichtnullelemente erhälst Du mit `nnz` (Number of NonZeros).
Mit Hilfe der Funktion `find` (siehe: `help find`) kann man die Arrays `Z, S, val` aus der Sparsematrix `S` extrahieren:

```
>> [Z,S,val]    = find(sM);
>> [rows, cols] = size(sM);          % Zeilen und Spaltenzahl bestimmen
>> sM2 = sparse(Z,S,val,rows,cols); % rows und cols ist wichtig, siehe unten
>> sM3 = sparse(rows, cols)         % entspricht sparse([],[],[],rows,cols,0) und erzeugt
                                    % eine leere rows x cols Sparsematrix
>> sM3(1,1) = 1;                    % 1 in den Eintrag links oben
>> [Z,S,val] = find(sM3);
>> A=sparse(Z,S,val);               % scheint dasselbe wie sM3 zu sein, aber...
>> full(A), full(sM3)
```

Natürlich ist die Datenstruktur der Sparsematrizen für dünnbesetzte Matrizen optimiert und sollte keinesfalls blindlings für alle Matrizen benutzt werden:

```
>> clear
>> fM = rand(100);   % erzeuge eine vollbesetzte 100 x 100 Matrix
>> sM = sparse(fM);  % erzeuge daraus eine Sparsematrix
>> whos              % der Speicherbedarf der Sparsematrix ist viel schlechter
```

Auch bei Verwendung von Sparsematrizen empfiehlt es sich Speicher vorzureservieren

```
>> nzmax = 10; rows = 100; cols = 200;     % max. Anzahl Nichtnullelemente
>> S = sparse([],[],[],rows,cols, nzmax)   % erzeugt leere rows x cols Matrix
>> S = spalloc(rows, cols, nzmax)          % tut das gleiche
>> whos                                    % Speicher fuer 10 doubles vorreserviert
>> S = sparse([],[],[],rows,cols, 5);
>> whos                                    % Speicher fuer 5 doubles vorreserviert
```

Bemerkung: Der Speicherverbrauch einer Sparsematrix in Matlab ist wie folgt:

$$\underbrace{nnz \times 8Byte}_{\text{Einträge}} + \underbrace{nnz \times 4Byte}_{\text{Positionen jeder Spalte}} + \underbrace{cols \times 4Byte}_{\text{erster nicht Null Eintrag jeder Spalte}} + \underbrace{4Byte}_{\textit{Datenstruktur}}$$

Es gibt auch spezielle Funktionen zum Erzeugen von Sparsematrizen: `spones, spdiags, sprand, speye`

```
>> S = sprand(20,10, 0.05) % erzeugt 20 x 10 Matrix mit Wkeit ist 0.05 fuer Eintrag ~=0
>> I = spones(S)           % erzeugt das Belegungsmuster von S
>> T = S';                 % transponiere S
>> L = logical(I);         % verwandle L in eine logische Matrix
>> T(L)                    % dann funktioniert der Zugriff mit dem Belegungsmuster
>> L = logical(S);         % das geht auch direkt durch Umwandlung von S in logical
>> T(L)
```

In der Numerik wird die Funktion `spdiags` häufig benutzt um Sparsematrizen durch Angabe der Vektoren auf den Diagonalen zu erzeugen:

```
>> n = 10;
>> e = ones(n,1);                          % Spaltenvektor mit 1en
>> D = spdiags([e, -2*e, e], -1:1, n, n) % siehe: help spdiags
>> full(D)
>> spy(D)
>> d = spdiags([-e, e], [-1, 1], n, n)
>> full(d)
>> spy(d)
>> m = 8;
>> r = (1:max(n,m)).';                   % r muss ein Spaltenvektor sein
>> d = spdiags([-r, r], [-1, 2], n, m)  % das geht auch wenn m~=n
>> full(d)
```

Mit vollbesetzten Matrizen geht ähnliches durch Addition von Diagonalmatrizen, die mit der Funktion `diag` erzeugt werden.

```
>> e = ones(n,1);
>> A = diag(1:n+1) + diag(-e,1) + diag(e,-1) % vollbesetzte Matrix
>> size(A)                                   % der Dimension (n+1) x (n+1)
              % mit spdiags muessen alle Eingabevektoren diesselbe Laenge haben
>> S = spdiags([-[4711; e], (1:n+1)', [e; 0815]], [1, 0, -1], n+1, n+1);
>> full(S)    % manche verschwinden aber einfach!
```

Aufgabe 14. *Erzeuge die Matrix*

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

unter Verwendung von `spdiags` *als Sparsematrix.*
Verallgemeinere den Code zu einer Funktion, die nach Eingabe von `n` *eine* $n \times (n+2)$ *Sparsematrix der obigen Form erzeugt.*

Zugriff auf Matrixelemente

Der Zugriff auf Elemente einer Sparsematrix erfolgt wie bei vollbesetzten Matrizen mit `S(i,j)` oder `S(k)`. Das Ergebnis eines Zugriffs auf einen Bereich ist wieder eine Sparsematrix. Auf Diagonalen kann ebenfalls mit `diag` zugegriffen werden.

```
>> S(2, 3)-S(12)    % z=2, s=3, l=12
>> S(:, 1:3)        % das Ergebnis ist wieder eine Sparsematrix
>> S(1:2, 2:4)
>> diag(S)          % Hauptdiagonale
>> diag(S, 2)       % 2. Nebendiagonale
```

Visualisierung von Sparsematrizen

Eine gute Möglichkeit, um Sparsematrizen zu veranschaulichen indem die Nichtnull Elemente farbig markiert werden, bietet die Funktion `spy`.

```
>> spy(S)           % Zeigt das Belegungsmuster von S an
```

Aufgabe 15. *Visualisiere eine Sparsematrix, die Du mit der Funktion aus Aufgabe 14 mit der Eingabe* `n=30` *erzeugt hast mit* `spy`.
Erzeuge eine zufällige Sparsematrix der Dimension 40×40 *mit ca. 160 Einträgen und visualisiere diese mit* `spy`.

Weitere Hinweise zur Implementierung von Sparsematrizen in Matlab finden sich in der Online Hilfe unter `Matlab->Mathematics->Sparse Matrices` oder in dem Artikel
`www.mathworks.com/access/helpdesk/help/pdf_doc/otherdocs/simax.pdf`