Molekularbiologie und Genetik II (für Studierende der Bioinformatik): matlab/Octave-Tutorial

Peter N. ROBINSON
peter.robinson@charite.de

28. Juni 2008

1 Einführung

In diesem Tutorial werden wir die in den Vorlesungen besprochenen Skripten erklären und erweitern. Jedes der in den Vorlesung besprochenen Skripte wird in Hinblick auf matlab-Syntax erklärt. Studenten sollen dann die Skripten erweitern und anpassen, um inhaltliche Aspekte der besprochenen populationsgenetischen Themen zu veranschaulichen.

Zahlreiche matlab/octave Tutorials sind kostenfrei im Internet erhältlich. S. hierzu vor allem http://www.matlab.com, http://www.gnu.org/software/octave/ und natürlich http://www.google.com.

matlab-Grundlagen

MATLAB (Matrix Laboratory) ist ein Softwarepaket für numerische Berechnungen (vor allem aber nicht nur mit Matrizen) und für die Visualisierung von Daten im technisch-wissenschaftlichen Bereich. Gnu Octave ist ein open-source-Softwarepaket, dessen Syntax weit(est)gehend mit demjenigen von matlab identisch ist. Für die hier vorgestellten Skripte kann entweder matlab oder Octave verwendet werden.

Im folgenden werde ich statt "matlab oder octave" der Einfachheit halber lediglich "matlab" schreiben.

matlab kann interaktiv verwendet werden, wobei Anweisungen direkt über die Tastatur eingegeben und sofort ausgeführt werden. Alternativ (und für größere Projekte empfehlenswert) können die Kommandos in m-Files gespeichert werden. Sobald diese im Commandfenster aufgerufen werden, führt sie der MATLAB-Interpreter wie ein Programm aus. Es wird bei den m-Files zwischen einfachen Skripten und Funktionen unterschieden. Ein Skript stellt eine Reihe von einzelnen Anweisungen dar, die nacheinander ausgeführt werden. m-Files können jedoch alternativ eine Funktion enthalten (deren Name mit dem des m-Files übereinstimmen muss). Solche Funktionen können dann vom Commandfenster oder innerhalb anderer m-Files ausgerufen werden.

2 logphase.m

Es handelt sich hier um ein denkbar einfaches Skript, das eine lineare Funktion ohne viel Schnörkel plottet. Das %-Zeichen bedeutet, dass eine Kommentarzeile folgt.

```
% logphase.m
% stellt die logarithmische Wachstumsphase von Bakterien in
% Kultur dar.

% x0 = anfaengliche Dichte von Bakterien
% r  = Wachstumsrate

x0 = 1.0;
r =0.01;
t = linspace(0,200,10);  % 10 linear verteilte Punkte zw. 0 und 200

lnx = log(x0) + r*t;     % Vektoroperation, alles auf einmal!

plot(t,lnx,'ro-');    % plotx, y in rot('r'), Datenpunkte als Kreise ('o')
                        % verbunden mit Linie ('-')
axis([ 0 300 0 3]); % Achsen einstellen [xmin xmax ymin ymax]
xlabel('Zeit');       % Beschriftung der X-Achse
ylabel('Wachstum');
```

```
19 title('Logarithmische Wachstumsphase');
```

Aufgaben:

- Lernen Sie das wichtigste Kommando in matlab kennen: `help`. Z.B. mit `help linspace` erfahren Sie, wie die eingebaute matlab-Funktion `linspace` funktioniert.
- Wie funktioniert linspace?
- Verwenden Sie linspace, um die Folge $(1,2,3,4,5)$ der Variablen x zuzuweisen und die Folge $(0,7,14,21,28,35,42)$ der Variablen y.

Ein wichtiger Unterschied zwischen matlab und vielen anderen Programmiersprachen wie Java besteht in der Verwendung vektorisierter Operationen in matlab. Da die Variable `t` ein Vektor ist, wird die Anweisung `log(x0) + r*t;` für alle Elemente des Vektors `t` gleichzeitig ausgeführt und die Ergebnisse wiederum als Vektor in der Variablen `lnx`. Verwenden Sie nun das Commandfenster in matlab um $\sqrt{t}$ für alle Werte von t mit einer Anweisung zu bestimmen.

3 logistic.m

Eine anonyme Funktion besteht aus einem einzigen MATLAB-Ausdruck hat aber beliebig viele Ein-und Ausgabeparameter. Anonyme Funktionen können auf der MATLAB-Kommandozeile definiert werden. Sie erlauben, schnell einfache Funktioen zu definieren, ohne ein File zu editieren. Die Zeile `f=@(t) C.*exp(r.*t) ./ (1 + C.*exp(r.*t )./ K);` weist der Variablen f eine anonyme Funktion zu, welche die logistische Gleichung berechnet.

$$f = \frac{Ce^{rt}}{1+Ce^{rt}/K}$$

Die Schreibweise `./` und `.*` bedeutet elementweise Division und Multiplikation, so dass man mit der einzigen Zeile `x=f(t)` die x-Werte für den gesamten Vektor t auf einmal berechnet.

```
1  % logistic.m
2  % Stellt logarithmisches Wachstum dar.
3  % x0 = anfaengliche Dichte
4  % r  = Wachstumsrate
5  % K  = Kapazitaet
6
7
8  x0=0.1;
9  r=0.01;
10 K=1.5;
11
12 t=linspace(0,750,100); % 100 linear verteilte Punkte zw. 0 und 200
13 C=(K*x0)/(K-x0);
14 f=@(t) C.*exp(r.*t) ./ (1 + C.*exp(r.*t)./K);
15 x=f(t);                % Vektoroperation, alles auf einmal!
16 plot(t,x,'ro-');       % plot x,y in rot ('r'), Datenpunkt als Kreise ('o')
17                        % verbunden mit Linie ('-')
18 axis([0 750 0 2]);     % Achsen einstellen [xmin xmax,ymin ymax]
19 xlabel('Zeit')         % Beschriftung der X-Achse
20 ylabel('Wachstum')     % Beschriftung der Y-Achse
21 title('Logistisches Wachstum');
```

Aufgabe: Verändere das Skript und beobachte das Verhalten des Systems mit unterschiedlichen Parametern. Was passiert, wenn x0 > K? Welcher Parameter bestimmt, wie schnell x seinen Gleichgewichtswert annimmt (bzw. sich ihm annähert)?

4 Genetische Drift

gdrift.m

```
% gdrift.m
% Genetische Drift simulieren


N=100; % Populationsgroesse
p=0.5; % Anteil Typ A in Population
ngenerations = 200;


% Initialisiere Vektor mit Werten fuer N Individuen in
% der Population: Allel A: 1, Allel a: 0
pop=rand(N,1)<p;



A=zeros(ngenerations,1);% Vektor mit Anzahl von Allel A Individuen

for i=1:ngenerations
    A(i) = sum(pop);
  ind=ceil(N*rand(N,1));
  pop=pop(ind);
end


% Ergebnis plotten
clf; % clear current figure.
gen = [1:ngenerations]';
plot(gen,A,'ro-');

axis([ 0 ngenerations 0 100]); % Achsen einstellen [xmin xmax ymin ymax]
xlabel('Generation');       % Beschriftung der X-Achse
ylabel('Anteil der Individuen mit Allel A');
title('Genetische Drift');
```

Das folgende Kommando vereinigt mehrere Operationen.

```
pop=rand(N,1)<p;
```

Das Kommando rand(1,N) vereinbart einen $1 \times N$-Vektor und initialisiert die Elemente mit Zufallszahlen $\in (0,1)$. Das Kommando rand(1,N)<p prüft dann, ob die so initialisierten Element kleiner 0.5 (p) ist und liefert das Ergebnis dieses Vergleiches als 0 (falsch) bzw. 1(wahr) zurück. Zum Beispiel

$$\boxed{rand(3,1)} = \begin{bmatrix} 0.44725 \\ 0.54643 \\ 0.44388 \end{bmatrix}$$

und

$$\boxed{rand(3,1) < 0.5} \Rightarrow \begin{bmatrix} 0.44725 < 0.5 \\ 0.54643 < 0.5 \\ 0.44388 < 0.5 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

Das folgende Kommando initialisiert einen Vektor `A`. Der Vektor hat `ngenerations` Reihen und 1 Spalte. Das Element `A(i)` wird jeweils die Anzahl von Individuen mit einem A-Allel in Generation `i` enthalten (die Anzahl der Individuen mit einem a-Allel ist dann `N-A(i)`).

```
A=zeros(ngenerations,1);
```

Die folgende For-Schleife wird `ngenerations`-mal durchgeführt und berechnet jeweils die Anzahl von Individuen in der Bevölkerung mit einem A-Allel in Generation `i`.

```
for i=1:ngenerations
    A(i) = sum(pop);
    ind=ceil(N*rand(N,1));
    pop=pop(ind);
end
```

Der Vektor pop hat N Elemente (eins für jedes Individuum in der Bevölkerung). pop(i) ist 1, falls Individuum i ein A-Allel bzw. 0, falls Individuum i ein a-Allel hat. Das Kommando sum(pop) berechnet die Summe aller Elemente von pop, was in diesem Fall dasselbe ist wie die Anzahl der A-Allele ist.

Das nächste Kommando verbindet 3 Schritte in einer Zeile. rand(N,1) haben wir oben kennengelernt. N*rand(N,1) multipliziert die Zufallszahlen $\in (0,1)$ mal N, was Zahlen $\in (0,N)$ erzeugt. Das ceil-Kommando erhöht jede reelle Zahl auf die nächstgrößere

Ganzzahl. Zum Beispiel

$$\boxed{rand(3,1)} = \begin{bmatrix} 0.039425 \\ 0.762976 \\ 0.432788 \end{bmatrix} \Rightarrow \boxed{3 * rand(3,1)} = \begin{bmatrix} 0.11827 \\ 2.28893 \\ 1.29837 \end{bmatrix} \Rightarrow \boxed{ceil(3 * rand(3,1))} = \begin{bmatrix} 1 \\ 3 \\ 2 \end{bmatrix}$$

Da matlab (im Gegensatz zu den meisten Computersprachen wie z.B. C und Java) die Indizierung von Arrays mit 1 anfangen lässt, können wir diese Zahlen als zufällige Vektorindices verwenden (da sie zwischen 1 und N liegen). Nach dem Kommando `ind=ceil(N*rand(N,1));` können wir also pop verwenden, um die Eltern der Individuen in der nächsten Generation per Zufall auszuwählen. `pop(j)` enthält den Index des Elternteils von Individuum `j`. Mit dem nächsten Kommando `pop=pop(ind)` vertauschen wir die Indices. Zum Beispiel könnte pop in Generation i folgende Werte für eine Population mit 5 Individuen enthalten:

$$pop = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

Das Ergebnis von `ind=ceil(N*rand(N,1));` könnte folgendermaßen ausfallen:

$$ind = \begin{bmatrix} 4 \\ 5 \\ 1 \\ 3 \\ 2 \end{bmatrix}$$

Das Kommando `pop(ind)` liefert die Element von `pop` nach der Reihenfolge von `ind` zurück, so

$$pop(ind) = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

Das Ergebnis wird schließlich wieder der Variablen `pop` für die nächste Generation zugewiesen Schließlich wird das Ergebnis geplottet:

```
clf; % clear current figure.
gen = [1:ngenerations]';
plot(gen,A,'ro-');

axis([ 0 ngenerations 0 100]); % Achsen einstellen [xmin xmax ymin ymax]
xlabel('Generation');       % Beschriftung der X-Achse
ylabel('Anteil der Individuen mit Allel A');
title('Genetische Drift');
```

Aufgabe: Studenten sollen insbesondere die folgenden drei Variablen verändern, um den Einfluss der Parameter Populationsgröße (`N`) und Zeit (`ngenerationen`) und p (anfänglicher Anteil des Allels A in der Population) auf die genetische Drift zu untersuchen.

5 Hardy-Weinberg-Gesetz: Ausbreitung eines günstigen Allels in einer Population

In der Vorlesung bzw. im SKript besprachen wir ein auf dem Hardy-Weinberg-Gesetz basierendes Modell für die positive Selektion in einer Population. Es wurde gezeigt, dass die Häufigkeit des günstigen Allels `A` in der Generation $n+1$ anhand von dessen Häufigkeit in der Generation n durch folgende Formel berechnet werden kann

$$p_{n+1} = \frac{p_n^2(1+s) + p_n q_n (1+hs)}{1+s(p_n^2 + 2hp_n q_n)} \tag{1}$$

Hierbei gibt s den Selektionsvorteil an. Wir können den Effekt eines dominanten, rezessiven oder intermediären Allels mit unterschiedlichen Werten von h simulieren:

- $h = 1$: dominant

- $h = 0$: rezessiv
- $h = 1/2$: intermediär

Wir verwenden den folgenden matlab-Code, um die Ausbreitung eines solchen Allels in einer Population zu simulieren und visualisieren. Wir sehen hier zum ersten Mal ein m-File, das eine Funktion definiert. Die Funktion setzt folgendem matlab-Code simulieren. Die im m-File definiert anonyme Funktion `p_n1` setzt Gleichung 1 um:

```
function A = select(p,s,h,ngenerations)

p_n1=@(p,q,h,s) (p^2*(1+s) + p*q*(1+h*s))/(1+s*(p^2 + 2*h*p*q));

A = [p ];

for i=2:ngenerations
    q=1-p;
    p = p_n1(p,q,h,s);
    A = [ A; p  ];
end
```

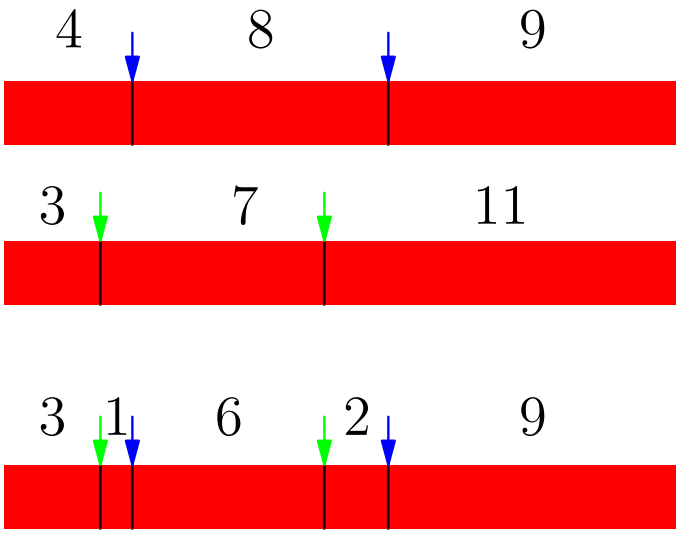
Der Vektor A wird mit p_0 initialisiert. In jedem Durchlauf der For-Schleife wird ein neuer Wert für p berechnet und ans Ende von A angefügt. Nach Beendigung der For-Schleife wird A zurückgegeben. Die Simulation wird dann mit folgenden Kommandos durchgeführt:

```
s=0.1;                  % Selektionsvorteil
ngenerations = 1200;    % Anzahl der Generationen fuer die Simulation
t= [1:ngenerations]';   % Zeitpunkte fuers Plotten

p=0.01;                 % Anfaengliche Frequenz von A


h=0; % rezessiv
A=select(p,s,h,ngenerations);

clf;
plot(t,A,'r-');
axis([0 ngenerations 0 1.05]);
```

Aufgabe: Wie schnell breitet sich ein günstiges Allel `A` in einer Population aus? Welchen Einfluss hat es darauf, ob das Allel dominant, rezessiv oder intermediär wirkt? Welchen Einfluss hat der Parameter s?

6 Das Doppelverdau-Problem

Gegeben sei ein DNA-Segment, das jeweils mit Enzym A, Enzym B bzw. beiden Enzymen verdaut wird. Das Ziel ist es, die Positionen der Schnittstellen in dem Fragment anhand des Schnittmusters zu bestimmen. Wenn zum Beispiel eine DNA-Segment folgendermaßen von Enzymen A und B geschnitten wird



, beobachten wir lediglich die Länge der einzelnen Fragmente:

- Verdau mit A: $dA = \{9,8,4\}$
- Verdau mit B: $dB = \{11,7,3\}$
- Doppelverdau: $dX = \{9,6,3,2,1\}$

Der Algorithmus soll aus den beobachteten Daten die Positionen der Schnittstellen (d.h., Anordnung der Fragmente) bestimmen.

Im folgenden matlab-Skript werden wir eine Lösung durch "Brachialgewalt" probierne.

- $\mathscr{A}$: Menge aller Permutationen nach Verdau durch A
- $\mathscr{B}$: Menge aller Permutationen nach Verdau durch B
- $\mathscr{A}\mathscr{B}$: Menge aller Permutationen vom Doppelverdau

doubledigest

Wir verwenden folgende Funktion, die alle möglichen Permutationen von $\mathscr{A}$, $\mathscr{B}$ und $\mathscr{A}\mathscr{B}$ miteinander vergleicht, bis eine kompatible Kombination gefunden wird.

```
function [a,b,ab] = doubledigest(A,B,AB)
%function doubledigest(A,B,AB)
%A: Set of fragment lengths from digestion by restriction enzyme A
%B: Set of fragment lengths from digestion by restriction enzyme B
%AB: Set of fragment lengths following double digestion by both A & B
%returns [a,b,ab] a permutation (order) compatible with data
%assumption: Each fragment length in A and B is distinct (not necessarily
%            true but OK for this demonstration program)

pA=perms(A); % all permutations of A
pB=perms(B);
pAB=perms(AB);

for i=1:length(pA)
    for j=1:length(pAB)
        if compatible(pA(i,:),pAB(j,:))
            for k=1:length(pB)
                if compatible(pB(k,:),pAB(j,:))
                    a=pA(i,:);
                    b=pB(k,:);
                    ab=pAB(j,:);
                    return;
                end
            end
        end
    end
end
% If we get here, no compatible combination was found
% return an error message and terminate function
error('No compatible order of restriction sites was found');
```

Aufgaben: Verwenden Sie help, um Informationen über die eingebaute Funktion perms zu erhalten. Zeigen Sie hiermit alle Permutationen der Menge lstinline!Z=['A' 'B' 'C']! an. Weisen Sie das Ergebnis von perms(Z) der Variablen pZ zu. Verwenden Sie size(pZ), um die Größe dieser Matrix zu bestimmen. Sie können nun mit pZ(1,:) die erste Permutation (d.h., die erste Reihe) extrahieren.

compatible

Die o.g. Funktion verwendet eine "private" Funktion, die in demselben m-File definiert ist und daher nur innerhalb dieses Files sichtbar ist.

```
function c = compatible(x,ab)
% x is vector of fragment lengths from single digest (A or B)
% ab is a vector of fragment lengths from double digestion
% return c=1 if compatible, c=0 if not compatible

cAB=cumsum(ab);
cX=cumsum(x);
% If x is compatible with AB then the cumulative sum of lengths will
% be a subset of the cumulative sum of lengths of the double digestion
% mem=ismember(i,j) returns a vector as long as i with elements
%   mem(x)=1 if i(x) is a member of j and mem(x)=0 otherwise
mem=ismember(cX,cAB);
% If compatible, mem has all 1. Then the sum of the elements is equal
% to the length
c = sum(mem)==length(mem);
return;
```

Die Funktionsweise dieser Funktion ist im Folgenden erklärt.

- AB, korrekte Reihenfolge: 3 – 1 – 6 – 2 – 9
- Kumulative Summe : 3 – 4 – 10 – 12 – 21
- A korrekte Reihenfolge: 4 – 8 – 9
- A, kumulative Summe: 4 – 12 – 21
- `ismem(A,AB)` liefert Vektor zurück dessen Einträge angeben, ob A(i) Mitglied von AB ist
- Falls Reihenfolge von A mit der von AB übereinstimmt, enthält mem nur '1'
- `c` wird dann auf 1 (wahr) gesetzt

Wir können die Analyse nun durch die folgenden Anweisungen durchführen:

```
%test doubledigeast code
%digestion by enzyme A
A=[9 8 4 ];
%digestion by enzyme B
B=[11 7 3];
%double digenstion
AB=[9 6 3 2 1];


[a,b,ab]=doubledigest(A,B,AB);
```