- Unix – Befehle
  - Verzeichniswechsel
  - Erstellen, löschen, kopieren oder verschieben von Dateien/Verzeichnissen
  - Grobe Bearbeitung von Dateien
  - Ändern von Benutzerrechten
- AWK / SED
  - Manipulieren von Dateien und Datensätzen
- SHELL – Skripte
  - Zusammenfassen verschiedenster Befehle in ein Skript
- FORTRAN 95
  - Programme zum Lösen gestellter Probleme
- GMT
  - Plotten von Daten, Karten,…
  - Manipulieren von Daten (z.B. interpolieren)
- LATEX
  - Zusammenfassen von Ergebnissen in Textform (z.B. Bericht)
  - Erstellen von Tabellen, Formeln
  - Einbinden von Grafiken

Warum gerade diese Programme??

Alle sind kostenlos und universell einsetzbar !!!!

Prinzipielles Konzept

Der Stream-Editor SED

Liest Text-Dateien zeilenweise und gibt das Ergebnis zeilenweise aus. Eignet sich perfekt für das Suchen/Ersetzen von Zeichenketten. Dabei kann SED Befehle von der Kommandozeile ausführen oder ein Skript, welches SED Befehle beinhaltet, abarbeiten. Die Syntax ist wie folgt:

Kommandozeile	sed ‘BEFEHLE‘ [FILE]	oder cat [FILE] | sed ‘BEFEHLE‘		
SED-Skript	sed –f SED-SKRIPT [FILE]	oder cat [FILE] | sed –f SED-SKRIPT	SED-SKRIPT:	Befehl1 Befehl2 …..

-n Ausgabe nur bei Kommando s oder p

Mögliche Kommandos

Suche/Ersetze Zeichenkette	s/{suche}/{ersetze}/g
Suche/Ersetze einzelne Zeichen	y/{suche}/{ersetze}/
Zeile löschen	/{suche}/d
Zeile schreiben	/{suche}/p

Beispiele:

```
sed 's/,/ /g'
```
In der ganzen Datei werden , durch ein Leerzeichen ersetzt. Alle Zeilen werden ausgegeben.

```
sed –n 's/,/ /gp'
```
In der ganzen Datei werden , durch ein Leerzeichen ersetzt. Nur die modifizierten Dateien werden ausgegeben.

```
sed –n '/,/p'
```
Es werden nur die Zeilen ausgeschrieben, die ein Komma beinhalten.

Einschränkungen

Einschränkung der Anwendung (innerhalb des Befehls vor dem Kommando anzugeben!):

`1`	Erste Zeile
`$`	Letzte Zeile
`!1`	Nicht die erste Zeile
`1,$`	Erste bis letzte Zeile
`20,40`	20te bis 40te Zeile
`!20,40`	Alle Zeilen bis auf die Zeilen 20 bis 40
`/TEXT/`	Alle Zeilen, die die Zeichenkette TEXT beinhalten
`/TEXT/!`	Alle Zeilen, die NICHT die Zeichenkette TEXT beinhalten

`/[Tt][Ee][Xx][Tt]/`	Alle Zeilen, die nicht die Zeichenkette TEXT beinhalten (egal ob Gross- oder Kleinschreibung)
`/^Anfang/,/^Ende/`	Alle Zeilen zwischen den Zeilen, die Anfang am Zeilenanfang und Ende am Zeilenfang zu stehen haben
`/$Anfang/,/$Ende/`	Alle Zeilen zwischen den Zeilen, die Anfang am Zeilenanfang und Ende am Zeilenfang zu stehen haben
`/^Anfang/,/Ende/!`	Alle Zeilen ausserhalb zwischen den Zeilen, die Anfang am Zeilenanfang und Ende am Zeilenfang zu stehen haben

Beispiele:

```
sed '10,20 s/=/ /g; s/,/\./g'
```

Für die Zeilen 10 bis 20 wird = durch ein Leerzeichen ausgetauscht. Für alle Zeilen wird noch , durch . ersetzt.

```
sed '10,20 {s/=/ /g; s/,/\./g}'
```

Für die Zeilen 10 bis 20 wird = durch Leerzeichen ausgetauscht und , wird durch . ersetzt.

Reguläre Ausdrücke

Ein regulärer Ausdruck (englisch regular expression, Abkürzung: RegExp oder Regex) ist in der theoretischen Informatik eine Zeichenkette, die der Beschreibung von Mengen von Zeichenketten mit Hilfe bestimmter syntaktischer Regeln dient. Reguläre Ausdrücke können als Filterkriterien in der Textsuche verwendet werden, indem der Text mit dem Muster des regulären Ausdrucks abgeglichen wird. Quelle: Wikipedia

`.`	1 beliebiges Zeichen
`x*`	0-∞ Wiederholungen des Teils x davor
`^`	Zeilenanfang
`$`	Zeilenende
`\x`	Metazeichen x schützen
`[abc],[a-z]`	Menge von Zeichen ([a-z]=Zeichenbereich)
`[^abc],[^a-z]`	Menge von Zeichen negieren

`\(...\)`	Zeichenkette merken
`x\{m,n\}`	m-n Wiederholungen des Teils x davor
`x\{m,\}`	m-∞ Wiederholungen des Teils x davor
`x\{m\}`	m Wiederholungen

Beispiel:

C++ Kommentare (//….) durch C-Kommentare (/*….*/) ersetzen:

Ein regulärer Ausdruck (englisch regular expression, Abkürzung: RegExp oder Regex) ist in der theoretischen Informatik eine Zeichenkette, die der Beschreibung von Mengen von Zeichenketten mit Hilfe bestimmter syntaktischer Regeln dient. Reguläre Ausdrücke können als Filterkriterien in der Textsuche verwendet werden, indem der Text mit dem Muster des regulären Ausdrucks abgeglichen wird. Quelle: Wikipedia

`.`	1 beliebiges Zeichen
`x*`	0-∞ Wiederholungen des Teils x davor
`^`	Zeilenanfang
`$`	Zeilenende
`\x`	Metazeichen x schützen
`[abc],[a-z]`	Menge von Zeichen ([a-z]=Zeichenbereich)
`[^abc],[^a-z]`	Menge von Zeichen negieren

`\(...\)`	Zeichenkette merken
`x\{m,n\}`	m-n Wiederholungen des Teils x davor
`x\{m,\}`	m-∞ Wiederholungen des Teils x davor
`x\{m\}`	m Wiederholungen

Beispiel:

C++ Kommentare (//….) durch C-Kommentare (/*….*/) ersetzen:

```
sed 's/            /          /'
```

Reguläre Ausdrücke

Ein regulärer Ausdruck (englisch regular expression, Abkürzung: RegExp oder Regex) ist in der theoretischen Informatik eine Zeichenkette, die der Beschreibung von Mengen von Zeichenketten mit Hilfe bestimmter syntaktischer Regeln dient. Reguläre Ausdrücke können als Filterkriterien in der Textsuche verwendet werden, indem der Text mit dem Muster des regulären Ausdrucks abgeglichen wird. Quelle: Wikipedia

`.`	1 beliebiges Zeichen
`x*`	0-∞ Wiederholungen des Teils x davor
`^`	Zeilenanfang
`$`	Zeilenende
`\x`	Metazeichen x schützen
`[abc],[a-z]`	Menge von Zeichen ([a-z]=Zeichenbereich)
`[^abc],[^a-z]`	Menge von Zeichen negieren

`\(...\)`	Zeichenkette merken
`x\{m,n\}`	m-n Wiederholungen des Teils x davor
`x\{m,\}`	m-∞ Wiederholungen des Teils x davor
`x\{m\}`	m Wiederholungen

Beispiel:

C++ Kommentare (//….) durch C-Kommentare (/*….*/) ersetzen:

```
sed 's/\/\/        /          /`
```

Ein regulärer Ausdruck (englisch regular expression, Abkürzung: RegExp oder Regex) ist in der theoretischen Informatik eine Zeichenkette, die der Beschreibung von Mengen von Zeichenketten mit Hilfe bestimmter syntaktischer Regeln dient. Reguläre Ausdrücke können als Filterkriterien in der Textsuche verwendet werden, indem der Text mit dem Muster des regulären Ausdrucks abgeglichen wird. Quelle: Wikipedia

`.`	1 beliebiges Zeichen
`x*`	0-∞ Wiederholungen des Teils x davor
`^`	Zeilenanfang
`$`	Zeilenende
`\x`	Metazeichen x schützen
`[abc],[a-z]`	Menge von Zeichen ([a-z]=Zeichenbereich)
`[^abc],[^a-z]`	Menge von Zeichen negieren

`\(...\)`	Zeichenkette merken
`x\{m,n\}`	m-n Wiederholungen des Teils x davor
`x\{m,\}`	m-∞ Wiederholungen des Teils x davor
`x\{m\}`	m Wiederholungen

Beispiel:

C++ Kommentare (//….) durch C-Kommentare (/*….*/) ersetzen:

```
sed 's/\/\/\(.*\)$/        /*\1*/'
```

Ein regulärer Ausdruck (englisch regular expression, Abkürzung: RegExp oder Regex) ist in der theoretischen Informatik eine Zeichenkette, die der Beschreibung von Mengen von Zeichenketten mit Hilfe bestimmter syntaktischer Regeln dient. Reguläre Ausdrücke können als Filterkriterien in der Textsuche verwendet werden, indem der Text mit dem Muster des regulären Ausdrucks abgeglichen wird. Quelle: Wikipedia

`.`	1 beliebiges Zeichen
`x*`	0-∞ Wiederholungen des Teils x davor
`^`	Zeilenanfang
`$`	Zeilenende
`\x`	Metazeichen x schützen
`[abc],[a-z]`	Menge von Zeichen ([a-z]=Zeichenbereich)
`[^abc],[^a-z]`	Menge von Zeichen negieren

`\(...\)`	Zeichenkette merken
`x\{m,n\}`	m-n Wiederholungen des Teils x davor
`x\{m,\}`	m-∞ Wiederholungen des Teils x davor
`x\{m\}`	m Wiederholungen

Beispiel:

C++ Kommentare (//….) durch C-Kommentare (/*….*/) ersetzen:

```
sed 's/\/\/\(.*\)$/\/*\1*\//'
```

Reguläre Ausdrücke

Ein regulärer Ausdruck (englisch regular expression, Abkürzung: RegExp oder Regex) ist in der theoretischen Informatik eine Zeichenkette, die der Beschreibung von Mengen von Zeichenketten mit Hilfe bestimmter syntaktischer Regeln dient. Reguläre Ausdrücke können als Filterkriterien in der Textsuche verwendet werden, indem der Text mit dem Muster des regulären Ausdrucks abgeglichen wird. Quelle: Wikipedia

`.`	1 beliebiges Zeichen
`x*`	0-∞ Wiederholungen des Teils x davor
`^`	Zeilenanfang
`$`	Zeilenende
`\x`	Metazeichen x schützen
`[abc],[a-z]`	Menge von Zeichen ([a-z]=Zeichenbereich)
`[^abc],[^a-z]`	Menge von Zeichen negieren

`\(...\)`	Zeichenkette merken
`x\{m,n\}`	m-n Wiederholungen des Teils x davor
`x\{m,\}`	m-∞ Wiederholungen des Teils x davor
`x\{m\}`	m Wiederholungen

Beispiel:

C++ Kommentare (//….) durch C-Kommentare (/*….*/) ersetzen:

```
sed 's/\/\/\(.*\)$/\/*\1*\//'
```

```
sed 's|//\(.*\)$|/*\1*/|'
```

Prinzipielles Konzept

Benannt nach Alfred v. **A**ho, Peter J. **W**einberger und Brian W. **K**ernighan. Liest Textdateien zeilenweise ein und erlaubt individuelle Operationen auf Spalten und Zeilen. Die Ausgabe erfolgt individuell nach Wünschen des Users. Wie beim SED können die Befehle via Kommandozeile gegeben werden oder via Skript:

Kommandozeile	awk ‘{BEFEHLE}‘ [FILE]	oder	cat [FILE] | awk ‘{BEFEHLE}‘
AWK-Skript	awk –f AWK-SKRIPT [FILE]	oder	cat [FILE] | awk –f AWK-SKRIPT

Die Befehle erfolgen in C-Syntax. Es gibt Variablen (auch arrays), Schleifen, Verzweigungen und Funktionen

Beispiele für awk in der Kommandozeile:

```
awk 'BEGIN{}{print $0}END{}' test.txt
```

Gibt alle Zeilen unverändert aus.

Zugriff auf Spalten via $:

```
awk '{print $1}' test.txt
```

Gibt die erste Spalte aus.

Zeilenzähler NR:

```
awk '{print NR}' test.txt
```

Gibt die aktuelle Zeilennummer aus.

Spaltenzähler NF:

```
awk '{print NF}' test.txt
```

Gibt die Spaltenanzahl der aktuellen Zeile aus.

Mit BEGIN{} und END{} werden einmalig Operationen beim Starten bzw. Beenden von AWK durchgeführt.

Prinzipielles Konzept

Benannt nach Alfred v. **A**ho, Peter J. **W**einberger und Brian W. **K**ernighan. Liest Textdateien zeilenweise ein und erlaubt individuelle Operationen auf Spalten und Zeilen. Die Ausgabe erfolgt individuell nach Wünschen des Users. Wie beim SED können die Befehle via Kommandozeile gegeben werden oder via Skript:

Kommandozeile	awk ‘{BEFEHLE}‘ [FILE]	oder	cat [FILE] | awk ‘{BEFEHLE}‘
AWK-Skript	awk –f AWK-SKRIPT [FILE]	oder	cat [FILE] | awk –f AWK-SKRIPT

Die Befehle erfolgen in C-Syntax. Es gibt Variablen (auch arrays), Schleifen, Verzweigungen und Funktionen

Beispiel für awk als Skript:

```
#!/bin/awk –f
BEGIN{

}
{
print $0
}
END{

}
```

Variablen

In AWK besteht die Möglichkeit, Werte in Variablen abzuspeichern. Im Gegensatz zu Programmiersprachen wie Fortran, C,… ist keine Variablendeklaration notwendig, was das Arbeiten mit Variablen sehr einfach macht.

Beispiele:

```
awk '{var=$1;print var}' test.txt
```

Variable var bekommt den Wert der ersten Spalte (der aktuellen Zeile) und wird immer ausgegeben.

```
awk '{var[1]=$1;var[2]=$2;print var[1], var[2]}' test.txt
```

Das erste Element des Arrays var bekommt den Wert der ersten Spalte und das zweite Element den Wert der zweiten Spalte. Anschliessend werden beide Werte ausgegeben. Arrays können in awk beliebig viele Dimensionen haben. Der Aufruf des jeweiligen Feldes erfolgt über den Index in der eckigen Klammer.

Math. Operationen auf Variablen

Auch in AWK besteht die Möglichkeit von math. Operationen auf Variablen (Addition (+), Subtraktion(-), Multiplikation(*), Division(/), Modulus(%)):

```
awk '{var=$1+$2;print var}' test.txt
```

Variable var bekommt den Wert der Summe der ersten und der zweiten Spalte und wird immer ausgegeben.
Differenz, Multiplikation und Division ergeben sich analog.

```
awk '{var=$1%$2;print var}' test.txt
```

Variable var bekommt den Wert des Restes der Division der ersten und der zweiten Spalte und wird immer ausgegeben.

```
 $1  $2  $1%$2
 10  5   0
 10  6   4
-10  6   -4
```

Des Weiteren besteht auch die Möglichkeit, math. Operationen in Form von Zuweisungen (+=,-=,*=,/=) durchzuführen:

```
awk '{var=0;var+=10;print var}' test.txt
```

Variable var bekommt den Wert 0 und wird dann um 10 erhöht.

Wenn eine Variable nur um eins erhöht (++) oder verringert (--) werden soll, dann kann folgende Syntax verwendet werden:

```
awk '{var=0;var++;print var}' test.txt
```

Variable var bekommt den Wert 0 und wird dann um 1 erhöht.

Potenzen werden mittels ^ oder ** angegeben:

```
awk '{var=5^2;print var}' test.txt
```

Variable var bekommt den Wert 25.

Verzweigungen

Verzweigungen werden, wie in den meisten Sprachen, mit „if“ durchgeführt. Dafür stehen folgende Vergleichsoperatoren zur Verfügung:

==	Gleich	<	Kleiner	<=	Kleinergleich	||	Oder
!=	Ungleich	>	Grösser	>=	Grössergleich	&&	Und

```
awk '{if($1<$2)print $1}' test.txt
```
Es wird nur die erste Spalte ausgegeben, wenn diese kleiner als die zweite Spalte ist.

```
awk '{if($1<$2){print $1}else{print $2}}' test.txt
```
Es wird nur die erste Spalte ausgegeben, wenn diese kleiner als die zweite Spalte ist sonst wird die zweite Spalte ausgegeben.

Es besteht auch die Möglichkeit der Mehrfachverzweigung:

```
awk '{if($1<$2){print $1}else{if($3>$2){print $2}else{print $3}}}' test.txt
```
Wenn die erste Spalte kleiner ist als die Zweite wird die Erste ausgegeben. Ist diese Bedingung nicht erfüllt, wird überprüft, ob die dritte Spalte grösser ist als die zweite Spalte. Ist dies der Fall, dann wird die zweite Spalte ausgegeben. Ist dies nicht der Fall, dann wird die dritte Spalte ausgegeben.
Mehrfachverzweigungen sind unbegrenzt möglich, werden aber aufgrund der geschweiften Klammern zunehmend unübersichtlich.

Schleifen

In awk gibt es kopfgesteuerte und fussgesteuerte Schleifen sowie Zählschleifen.

Kopfgesteuert:

```
awk '{i=1;while(i<5){print i;i++;};}' test.txt
```

Es wird 1 bis 4 untereinander ausgegeben. Diese Ausgabe wird so oft wieder holt, wie test.txt Zeilen hat

```
awk '{i=1;do{print i;i++;}while(i<5)}' test.txt
```

Es wird 1 bis 4 untereinander ausgegeben. Diese Ausgabe wird so oft wieder holt, wie test.txt Zeilen hat

```
awk '{for(i=1;i<=5;i++){print i;}}' test.txt
```

Es wird 1 bis 5 untereinander ausgegeben. Diese Ausgabe wird so oft wieder holt, wie test.txt Zeilen hat

```
awk '{a[0]=1;a[1]=2;a[2]=3;for(i in a){print a[i]}}' test.txt
```

Es wird 1 bis 3 untereinander ausgegeben. Diese Ausgabe wird so oft wieder holt, wie test.txt Zeilen hat

Schleifen können ausserdem mit folgenden Befehlen abgebrochen werden:

`break`	Verlassen der Schleife
`next`	Starten des nächsten Schleifendurchlaufs
`exit`	Stoppen der Schleife und des AWK-Skripts

Builtin Funktionen

In awk gibt es bereits vordefinierte Funktion für strings und math. Operationen.

String-Funktionen:

Funktion	Beschreibung
`n=sub(suche, ersetze [,ziel])`	Sucht und ersetzt strings (zu vergleichen mit sed ‘s///‘). n bekommt den Wert der Anzahl an Austauschen. Als optionales Argument „ziel“ kann eine Variable angegeben werden. Ist keine Variable angegeben, wird automatisch in $0 gesucht (gesamte Zeile).
`n=gsub(suche, ersetze [,ziel])`	Gleiche Funktion wie sub, erlaubt allerdings mehrere Operationen pro String. (zu vergleichen mit sed ‘s///g‘)
`n=index(ziel,suche)`	n bekommt den Positionswert, wo das erste Mal die zu suchende Zeichenkette in der Zeile gefunden wurde.
`n=length(string)`	n bekommt den Wert der Länge des Strings. Wenn string=$0, dann bekommt n den Wert der Zeilenlänge.
`tolower(string)`	Wandelt alle Buchstaben in Kleinbuchstaben um und gibt diese Zeichenkette an string zurück.
`toupper(string)`	Wandelt alle Buchstaben in Grossbuchstaben um und gibt diese Zeichenkette an string zurück.

Builtin Funktionen

In awk gibt es bereits vordefinierte Funktion für strings und math. Operationen.

Beispiele an folgenden String: echo „Das ist ein Test Test“ | awk ….

AWK – Kommando	Ausgabe
`awk '{sub("Test","test")}'`	
`awk '{print sub("Test","test")}'`	1
`awk '{print gsub("Test","test")}'`	2
`awk '{var=$0;sub("Test","test",var);print var}'`	Das ist ein test Test
`awk '{var=$0;gsub("Test","test",var);print var}'`	Das ist ein test test
`awk '{print index($0,"Test")}'`	13
`awk '{print length($0)}'`	21
`awk '{print tolower($0)}'`	das ist ein test test
`awk '{print toupper($0)}'`	DAS IST EIN TEST TEST

Builtin Funktionen

In awk gibt es bereits vordefinierte Funktion für strings und math. Operationen.

Math.-Funktionen:

Funktion	Beschreibung
`sin(x)`	Sinus von x in Radians
`cos(x)`	Cosinus von x in Radians
`atan(x,y)`	Arcustangens von x/y
`exp(x)`	Exponentialfunktion von x
`log(x)`	ln von x
`sqrt(x)`	Wurzel von x
`rand()`	Zufallszahl zwischen 0 und 1 (Gleichverteilt)
`srand()`	Aktueller Zeitwert als Startwert
`srand(x)`	x als Startwert

Builtin Funktionen

In awk gibt es bereits vordefinierte Funktion für strings und math. Operationen.

…. aber keine Funktionen für tan(x), asin(x), acos(x) und log. Was tun? Die Lösung bieten selbstdefinierte Funktionen.

Zuerst als Kommandozeile:

Arcussinus

```
awk 'function asin(x){pi=atan2(0,-1);return atan2(x,sqrt(1-x*x))/pi*180}{print asin($1)}'
```

Arcuscosinus

```
awk 'function acos(x){pi=atan2(0,-1);return atan2(sqrt(1-x*x),x)/pi*180}{print acos($1)}'
```

Tangens

```
awk 'function tan(x){pi=atan2(0,-1);return atan2(x/180*Pi,1)}{print tan($1)}'
```

Log (Logarithmus zur Basis 10)

```
awk 'function log10(x){return log(x)/log(10)}{print log10($1)}'
```

$$\log_{10}(x)*\log_e(10) = \log_e(x)$$

$$\exp\{\log_{10}(x)*\log_e(10)\} = \exp\{\log_e(x)\}$$

$$\exp\{\log_e(10)*\log_{10}(x)\} = x$$

$$\exp\{\log_e(10)\}^{\log_{10}(x)} = x$$

$$10^{\log_{10}(x)} = x$$

$$x = x$$

Builtin Funktionen

In awk gibt es bereits vordefinierte Funktion für strings und math. Operationen.

…. aber keine Funktionen für tan(x), asin(x) und acos(x). Was tun? Die Lösung bieten selbstdefinierte Funktionen.

Und nun als Skript:

Arcussinus

```
#!/bin/awk -f
{
print asin($1)
}
function asin(x){
pi=atan2(0,-1);
return atan2\
(x,sqrt(1-x*x))/pi*180
}
```

Arcuscosinus

```
#!/bin/awk -f
{
print acos($1)
}
function acos(x){
pi=atan2(0,-1);
return atan2\
(sqrt(1-x*x),x)/pi*180
}
```

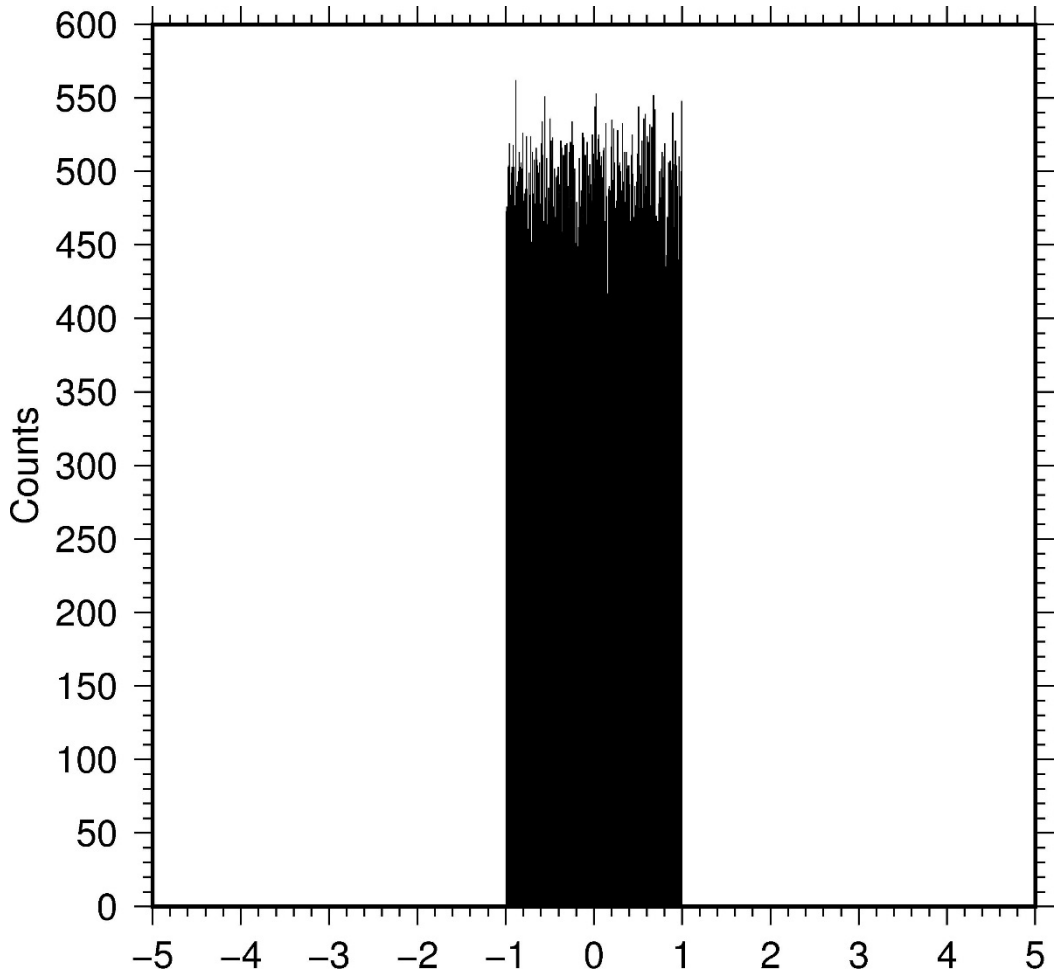
Tangens

```
#!/bin/awk -f
{
print tan($1)
}
function tan(x){
pi=atan2(0,-1);
return atan2(x/180*pi,1)
}
```

Zufallszahlen

Abschliessend zu den Funktionen noch ein Beispiel zu den Zufallszahlen. Mit rand können gleichverteilte Zufallszahlen im gewünschten Bereich erstellt werden (z.B. zwischen -1 und 1):

```
echo 100000 | awk '{srand();for(i=1;i<=$1;i++){print rand()*2-1}}'
```



Zufallszahlen

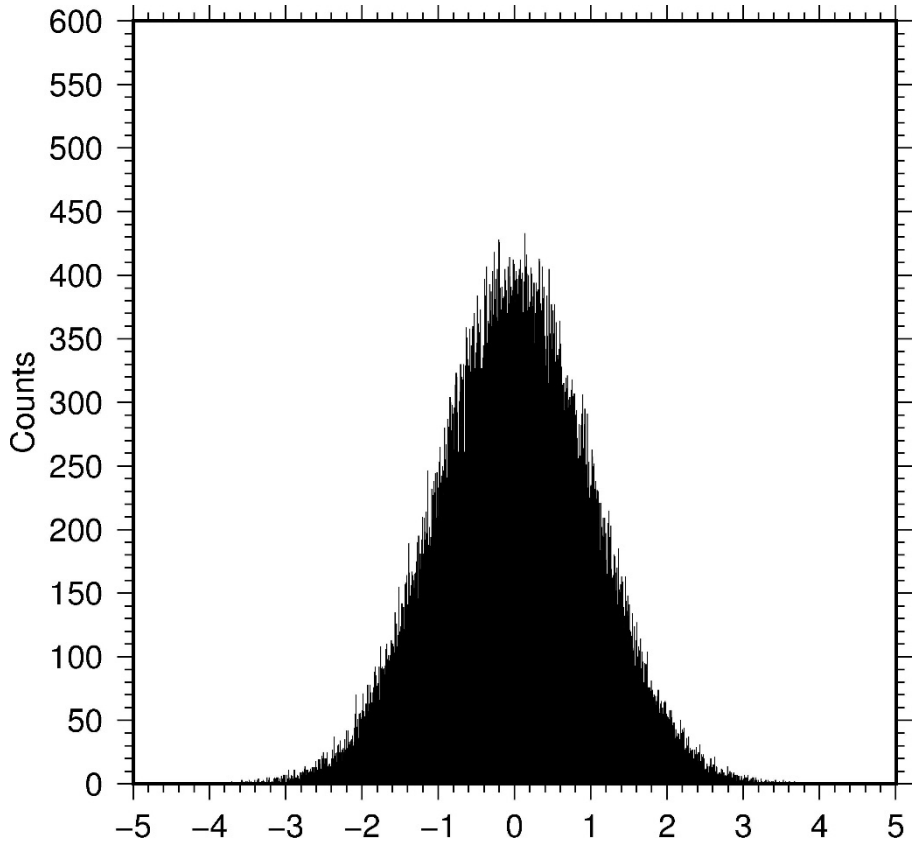
Abschliessend zu den Funktionen noch ein Beispiel zu den Zufallszahlen. Mit rand können gleichverteilte Zufallszahlen im gewünschten Bereich erstellt werden (z.B. zwischen -1 und 1):

```
echo 100000 | awk '{srand();for(i=1;i<=$1;i++){print rand()*2-1}}'
```

Wie generiert man aber nun Zufallszahlen, die normalverteilt sind? Einen einfach zu implementierender Algorithmus ist die Polar-Methode von George Marsaglia:

```
echo 50000 | awk '{srand();for(i=1;i<=$1;i++)\
{do{x1=2.0*rand()-1; x2=2.0*rand()-1;s=x1*x1+x2*x2;}\
while ((s>1) || (s==0)) rnd=sqrt(-2.0*log(s)/s);\
print x1*rnd,"\n", x2*rnd}}'
```

Diese Normalverteilung ist zentriert bei 0 und hat die Standardabweichung von 1.



Zufallszahlen

Abschliessend zu den Funktionen noch ein Beispiel zu den Zufallszahlen. Mit rand können gleichverteilte Zufallszahlen im gewünschten Bereich erstellt werden (z.B. zwischen -1 und 1):

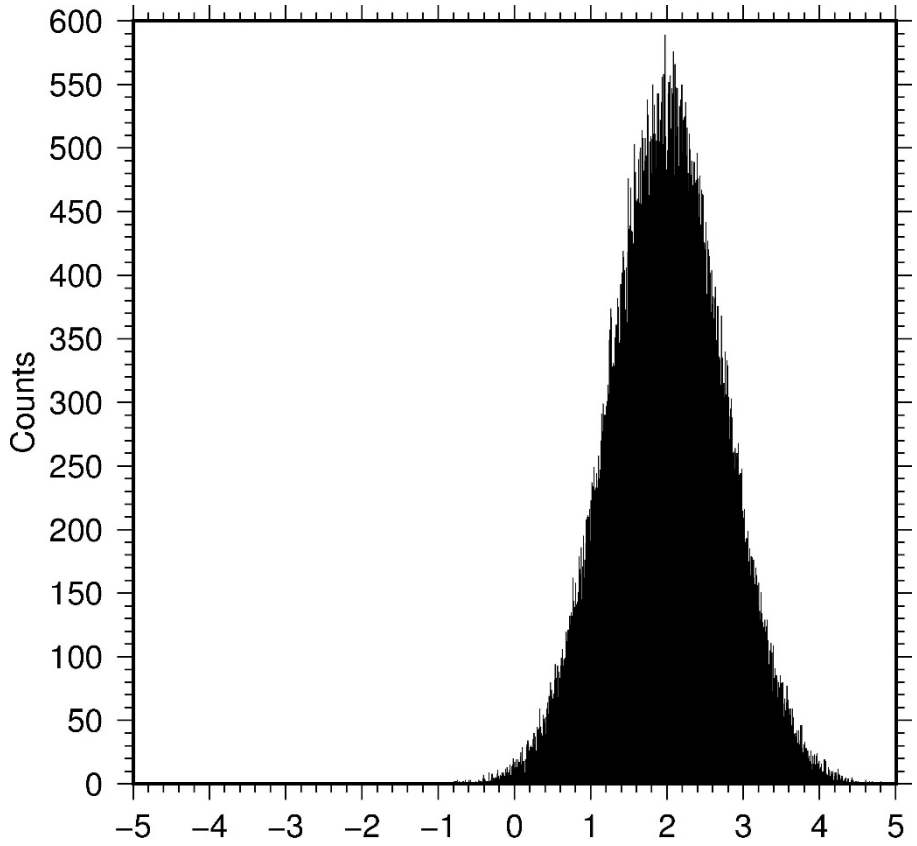
```
echo 100000 | awk '{srand();for(i=1;i<=$1;i++){print rand()*2-1}}'
```

Wie generiert man aber nun Zufallszahlen, die normalverteilt sind? Einen einfach zu implementierender Algorithmus ist die Polar-Methode von George Marsaglia:

```
echo 50000 | awk '{srand();for(i=1;i<=$1;i++)\
{do{x1=2.0*rand()-1; x2=2.0*rand()-1;s=x1*x1+x2*x2;}\
while ((s>1) || (s==0)) rnd=sqrt(-2.0*log(s)/s);\
print x1*rnd,"\n", x2*rnd}}'
```

Diese Normalverteilung ist zentriert bei 0 und hat die Standardabweichung von 1. Eine Multiplikation mit der Zufallszahl vergrössert oder verringert die Standardabweichung. Addition oder Subtraktion einer konstanten Zahl auf bzw. der Zufallszahl bewirken eine Verschiebung des Histogramms in positive oder negative Richtung.

```
echo 100000 | awk '{srand();for(i=1;i<=$1;i++)\
{do{x1=2.0*rand()-1; x2=2.0*rand()-1;s=x1*x1+x2*x2;}\
while ((s>1) || (s==0)) rnd=sqrt(-2.0*log(s)/s);\
print x1*rnd*0.75+2,"\n", x2*rnd*0.75+2}}'
```



Formatierte Ausgabe

Auch in awk können Variablen in gewünschten Formaten ausgegeben werden (· steht für ein Leerzeichen).

Strings

Format	Ausgabe
`printf("%s","Hallo");`	`Hallo`
`printf("%10s","Hallo");`	`····Hallo`
`printf("%.2s","Hallo");`	`Ha`
`printf("%-10s","Hallo");`	`Hallo····`

Integer (ganzzahlig)

Format	Ausgabe
`printf("%d",50);`	`50`
`printf("%10d",50);`	`·····50`
`printf("%-10d",50);`	`50·····`
`printf("%.5d",50);`	`00050`
`printf("%05d",50);`	`00050`

Floats (Gleitkomma)

Format	Ausgabe
`printf("%f",3.14159265);`	`3.141593`
`printf("%10.8f",3.14159265);`	`3.14159265`
`printf("%8.4f",3.14159265);`	`··3.1416`
`printf("%08.4f",3.14159265);`	`003.1416`
`printf("%e",3.14159265);`	`3.141593e+00`
`printf("%14.8e",3.14159265);`	`3.14159265e+00`
`printf("%14.4e",3.14159265);`	`···3.1416e+00`
`printf("%014.4e",3.14159265);`	`00003.1416e+00`

Allgemeine Formate

Format	Bedeutung
`printf("\n");`	Zeilenumbruch
`printf("\t");`	Horizontaler Tab
`printf("\v");`	Vertikaler Tab
`printf("\b");`	Backspace (Cursor ein Zeichen nach links)

Definition

In der Informatik ist die Shell die Schnittstelle zwischen Benutzer und Computer. In UNIX existieren verschiedene Typen von Shells. Grundlegend differenziert man zwischen der Bash und der C-Shell. Der Funktionsumfang ist relativ ähnlich jedoch orientiert sich die C-Shell in der Syntax mehr an C. Das ist ganz vorteilhaft, da AWK sich auch an C orientiert.

Was kann alles eine Shell?

- Ausführen von Kommandos
- Bedingungen (if,switch) und Schleifen (while,foreach)
- Interne Kommandos (cd,ls,…)
- Interne Variablen ($HOME, $SHELL)
- Manipulation der Umgebungsvariablen für neue Prozesse ($PATH,…)
- Ein- und Ausgabeumlenkung
- Starten mehrere Prozesse und deren Verbindung über Pipes
- Starten von Prozessen im Hintergrund (z.B. durch anhängen eines &)
- Verschieben von Prozessen aus den Vordergrund in den Hintergrund und umgekehrt (Ctrl+Z und bg für Background)
- Wiederholung und Editieren früherer Kommandos (History)
- Eingebautes Kommando zur Durchführung von Berechnungen

Verzweigungen

Wie in AWK, so gibt es auch in der SHELL Verzweigungen deren Syntax (C-SHELL) ähnlich zu AWK ist.

```
#!/bin/tcsh
set i=1
if($i == 1)echo $i
```

```
#!/bin/tcsh
set i=1
if($i == 1)then
  echo $i
endif
```

```
#!/bin/tcsh
set i=1
if($i == 1)then
  echo $i
else
  echo "Keine 1"
endif
```

Zwischen den Operator und den Operanden muss immer ein Leerzeichen stehen!

Als Operanden sind nur Strings und Integer zulässig!

Zusätzlich zu if gibt es auch noch das switch Kommando:

```
#!/bin/tcsh

set i=5
switch ($i)
case 5:
  echo "5"
  breaksw
default:
  echo "Nicht 5"
  breaksw
endsw
```

Es können beliebig viele case-Optionen aufgelistet werden.

Die möglichen Operatoren sind die Gleichen wie bei AWK:

==	Gleich	<	Kleiner	<=	Kleinergleich	||	Oder
!=	Ungleich	>	Grösser	>=	Grössergleich	&&	Und

Verzweigungen

Es besteht aber auch die Möglichkeit gezielt abzufragen, ob z.B Dateien oder Verzeichnisse existieren:

Ausdruck	liefert TRUE, wenn
-d datei	datei ein Verzeichnis (**d**irectory) ist
-e datei	datei existiert (**e**xistence)
-f datei	datei eine normale Datei (**f**ile) ist
-o datei	datei dem Benutzer gehört (**o**wnership)
-r datei	datei gelesen (**r**ead) werden darf
-w datei	datei beschrieben (**w**rite) werden darf
-x datei	datei ausgeführt (e**x**ecute) werden darf
-z datei	datei leer (**z**ero) ist

Beispiel:

```
#!/bin/tcsh
If(-e „test.dat“)then
  echo „test.dat“ existiert
endif
```

Schleifen

Es gibt auch hier Bedingungsschleifen (while) und Zählschleifen (foreach)

```
#!/bin/tcsh
set i=1
while($i < 10)
  echo $i
  @ i++
end
```

Zwischen den Operator und den Operanden muss immer ein Leerzeichen stehen!

```
#!/bin/tcsh
set i=1
while($i < 10)
  echo $i
  @ i=$i + 5
end
```

```
#!/bin/tcsh

foreach i ( 1 2 3 4 5 6 7 )
   echo $i
end
```

```
#!/bin/tcsh

foreach i ( `seq 1 1 7` )
   echo $i
end
```

`` ` `` gibt an der Stelle das Ergbnis des eingeschlossenen Kommandos aus. seq ist ein Zähler mit simpler Syntax: seq start increment ende. Start, Ende und Increment können Integer oder Float sein.

Argumente

Bei dem Aufruf von SHELL-Skripten können auch Argumente mit übergeben werden. Dafür gilt folgende Syntax:

```
./mein_script.tcsh argument1 argument2 ....
```

Im Skript wird das Argument mittels $-Zeichen angesprochen. Hierfür ein Beispiel. Wir rufen das Skript „schreibe_argument.tcsh“ mit den Argumenten 1, 5, und 20 auf:

```
./schreibe_argument.tcsh 1 5 20
```

Das Skript selber hat folgende Gestalt:

```
#!/bin/tcsh
if ( $# > 0 )then
  foreach i ( `seq 1 1 $#`)
      echo $i
  end
endif
```

In dem Skript wird zuerst überprüft, ob Argumente mit übergeben wurden. Ist dies der Fall, wird in der foreach-Schleife jedes Argument mittels dem Befehl echo ausgegeben:

```
1
5
20
```

Beispiel

Ein kleines Beispiel für ein Shell-Skript. Im aktuellen Verzeichnis befinden sich die Dateien datei1 und datei2. Diese beiden Dateien sollen umbenannt werden in out1 und out2.

`mv datei? out?` geht nicht -> datei1 würde nach datei2 verschoben.

Die Lösung des “Problems“ ist mit folgenden Skript gegeben:

```
#!/bin/tcsh
foreach filename( `ls datei?` )
  set fileout=`echo $filename | sed 's/datei/out/'`
  \mv $filename $fileout
  echo \mv $filename $fileout
end
```

Zusammenfassung

Mit SED und AWK lassen sich Dateien gezielt manipulieren.

SED eignet sich dabei eher um Zeichenketten auszutauschen (sed ‘s/ / /g‘) oder um ganze Zeilen mit der betreffenden Zeichenkette zu löschen (sed ‘//d‘) oder auszuschreiben (sed –n ‘/ /p‘).

Mit AWK ist es dagegen möglich, einzelne Zeilen / Spalten in Abhängigkeit von deren Inhalt oder Zeilen- / Spaltennummer in einer ASCII-Datei zu modifizieren. Dafür stehen dem User Variablen, Verzweigungen und Schleifen zur Verfügung.

Mit der Kombination von UNIX-Kommandos, SED und AWK können ASCII-Dateien in jede gewünschte Form manipuliert werden.

Damit diese Kommandos nicht jedes Mal neu in die SHELL eingegeben werden müssen, kann man sie in SHELL-Skripten auflisten. In dieser Vorlesung wird die TCSH als SHELL verwendet, da Diese sich wie AWK an der C-Syntax orientiert. In jeder SHELL gibt es auch Variablen, Verzweigungen und Schleifen. Des Weiteren ist es möglich SHELL-Skripte mit Argumenten aufzurufen.