Ana I-1: Kurze Anleitung zum Einstieg in Julia, Induktion

Folgende Begriffe, Konzepte und Sätze sollten am Ende des Sheets verstanden sein:

- Julia-Grundlagen
- Vollständige Induktion
- Vollständige Induktion mit allgemeinem Startindex
- Starke Induktion

Kurze Anleitung zum Einstieg in Julia

Herzlich Willkommen in den Hands-on Workshops! Der erste Teil des Sheets (entspricht dem vorab hochgeladenen Sheet "Einführungsblatt_Julia") soll Ihnen die grundsätzliche Bedienung eines Jupyter-Sheets (das ist die Dateiform unserer digitalen Übungsblätter) erklären und in die Grundrechenarten in Julia einführen. Die Aufgaben werden benötigt, um die kommenden Sheets erfolgreich bearbeiten zu können. Es werden keinerlei Vorkenntnisse im Programmieren vorausgesetzt. Weitere Anwendungen und Befehle in Julia, die wir im ganzen Semester benötigen werden (z.B. Funktionen, Plots) werden Ihnen dann ausführlich auf dem ersten Sheet zur Linearen Algebra I erklärt.

Vorbereitung

Grundsätzliche Tipps zur Bearbeitung der Sheets: Wenn man links neben eine Zelle klickt und dann a bzw. b drückt, wird eine neue Zelle (obendrüber bzw. untendrunter) eingefügt. Alternativ kann man oben in der Leiste das + Symbol anklicken. Die neue Zelle ist dann automatisch eine "Code" Zelle, also zur Durchführung von Rechnungen mit Eingabe von Julia Befehlen geeignet. Wenn Sie in dieser Zelle nichts rechnen wollen, sondern erklärenden Text zu den Lösungen einfügen möchten, können Sie die Zelle im Reiter oben in eine "Markdown" Zelle umwandeln. Um den Befehl in einer Code Zeile auszuführen, muss Shift und Enter gedrückt werden. Um eine Markdown Zelle in "fertigen Text" umzuwandeln, drückt man ebenfalls Shift und Enter. Möchte man die Markdown-Zelle später nocheinmal bearbeiten, kommt man durch Doppelklick darauf wieder in den Bearbeitungsmodus.

Aufgabe 1 ***

- a) Erzeugen Sie unter dieser grünen Zelle eine neue Code Zelle, in der Sie berechnen

wieviele Minuten 1 Jahr hat (kein Schaltjahr).
*Hinweis: Die Grundrechenarten werden in Julia einfach mit +, - , *, / eingegeben*
Fehlerbehebung: Falls nach Drücken von Shift und Enter kein Ergebnis kommt, gehen Sie sicher, dass Julia überhaupt schon gestartet wurde, oben rechts sollte "Julia 1.4.2" stehen. Falls dort "No Kernel" steht, dann klicken Sie auf "No Kernel" und wählen Sie "Julia 1.4.2" in der Liste aus, ggf. mehrmals hintereinander.

- b) Erzeugen Sie darunter eine Markdown Zelle in der Sie einen Antwortsatz schreiben und die Rechnung kurz erläutern.

Ein Semikolon am Ende des Befehls (bzw. der Rechnung) führt dazu, dass der Befehl ausgeführt wird, aber kein Ergebnis angezeigt wird. Wenn mehrere Befehle untereinander stehen, werden alle ausgeführt aber nur das Ergebnis des letzten angezeigt. Wenn auch das Ergebnis eines vorherigen Befehls aus der gleichen Zelle angezeigt werden soll, muss die entsprechende Rechnung in Klammern `(  )` gesetzt werden und der Befehl `println` davor geschrieben werden. Alternativ können mehrere Rechnungen mit Kommata getrennt in eine Zeile geschrieben werden, dann werden alle outputs in einem Tupel angeordnet angezeigt.
Kommentare innerhalb von Code-Zellen müssen mit `#` davor gekennzeichnet werden, damit Julia weiß, dass hier nichts zu rechnen ist. Wenn Sie Ihr Sheet als **PDF exportieren** wollen, ist unserer Erfahrung nach der beste Weg so: oben links unter *File* dann *Save and Export Notebook As...* auswählen, und zwar zunächst als HTML Export (nicht PDF!!). Dieser wird in Ihrem Download Ordner gespeichert und wenn Sie die Datei in einem Browser öffnen können Sie sie dort dann als PDF exportieren oder als PDF drucken (z.B. durch Drücken von Strg+P).

Aufgabe 2 ***

- a) Erzeugen Sie eine neue Code Zelle, in der Sie berechnen wieviele Minuten ein Nicht-Schaltjahr bzw ein Schaltjahr hat, sodass beide Ergebnisse angezeigt werden. Probieren Sie es einmal mit Kommata getrennt und einmal mit `println` . Schreiben Sie in die gleiche Zelle einen kurzen Kommentar neben die Rechnung, um zu kennzeichnen, welche Rechnung sich auf das Schaltjahr bezieht bzw. auf das Nicht-Schaltjahr.
- b) Exportieren Sie nun das (bis hierhin) bearbeitete Sheet als PDF.

Bevor wir nun richtig anfangen zu rechnen, laden wir zunächst das package `SymPy` . Mit `SymPy` können wir symbolisch rechnen, beispielsweise bekommen wir bei Rechnungen mit rationalen Zahlen oder Quadratwurzeln dann exakte Ergebnisse und keine gerundeten Fließkommazahlen. (Für Interessierte hier noch der Link zur Dokumentation.)
Hinweis: Es kann ein paar Minuten dauern, bis SymPy fertig geladen ist, und Julia wieder bereit (*idle*) ist. Das Package muss nur einmal zu Beginn der Bearbeitung des sheets geladen werden.

In [3]:
```
# bitte führe diese Zelle einmal aus (reinklicken und Shift+Enter)

using SymPy
```

Grundlegende Rechenoperationen

Brüche können mit `/` eingegeben werden, dann rechnet Julia allerdings mit gerundeten Fließkommazahlen. Soll exakt gerechnet werden, dann müssen alle Brüche, die vorkommen mit `//` eingegeben werden.

Aufgabe 3 *** Berechnen Sie in der Code Zelle unten $\frac{5}{3}+\frac{17}{11}$. Vergleichen Sie die Eingabe der drei verschiedenen Befehle

- `5/3+17/11`
- `5//3+17//11`
- `5//3+17/11`

In [4]:
```
# Lösung Aufgabe 3
```

In den folgenden Beispielen illustrieren wir, dass auch bei der Eingabe von Quadratwurzeln oder trigonemtrischen Ausdrücken zwischen symbolischen Ausdrücken und näherungsweisen Rechnungen unterschieden werden kann:

In [5]:
```
sqrt(2) # Quadratwurzel aus 2 (sqrt steht für das englische Wort square root)
```

Out[5]:
```
1.4142135623730951
```

In [6]:
```
sympy.sqrt(2) # So geben wir Wurzel aus 2 an, wenn damit symbolisch gerechnet werden
```

Out[6]: $\sqrt{2}$

In [7]:
```
pi # Die Zahl pi
```

Out[7]:
```
π = 3.1415926535897...
```

In [8]:
```
sympy.pi # So geben wir pi ein, wenn damit symbolisch gerechnet werden soll
```

Out[8]: π

In [9]:
```
cos(2) # So erhalten wir ein gerundetes Ergebnis
```

Out[9]:
```
-0.4161468365471424
```

In [10]:
```
sympy.cos(2) # So geben wir cos(2) ein, wenn damit symolisch gerechnet werden soll
```

Out[10]: $\cos\left(2\right)$

In [11]:
```
sympy.sin(sympy.pi) # Exaktes Ergebnis
```

Out[11]: 0

In [12]:
```
sin(pi) # wenn man nicht symbolisch rechnet, erhält man die Ergebnisse nur näherungsw
```

Out[12]:
```
1.2246467991473532e-16
```

In [13]:
```
sympy.sin(pi) # bei gemischten Ausdrücken rechnet Julia nicht symbolisch (hier wurde
```

Out[13]: $1.22464679914735 \cdot 10^{-16}$

Freiwillige Aufgabe *** Recherchieren Sie im Internet, was Maschinengenauigkeit ist und welche Maschinengenauigkeit mit heutigen Computern erreicht wird. Sie werden dann merken, dass Julia hier mit $1.22... * 10^{-16}$ ein innerhalb der Maschinengenauigkeit korrektes Ergebnis liefert.

Aufgabe 4 ***

- a) Berechne unten $\frac{5}{3} + \sqrt{17 + \sin(3)}$ einmal symbolisch und einmal näherungsweise. Welche der beiden Rechnungen hilft weiter, wenn wir in einer Anwendung wissen möchten, ob der Ausdruck kleiner als 6 ist?
- b) Berechne unten $\cos(\frac{\pi}{2})$ einmal symbolisch und einmal näherungsweise. Welche der beiden Rechnungen hilft weiter, um mit Sicherheit sagen zu können, dass hier eine Nullstelle der Kosinus-Funktion vorliegt?

In [14]:
```
#a
```

In [15]:
```
#a
```

In [16]:
```
#b
```

In [17]:
```
#b
```

Wir listen nun noch weitere nützliche Grundrechenarten auf:

- `x^(1/n)` berechnet (näherungsweise) $\sqrt[n]{x}$ für eine nichtnegative Zahl x und eine natürliche Zahl n
- `factorial(n)` berechnet die Fakultät $n!$ einer natürlichen Zahl n
- `abs(x)` berechnet den Absolutbetrag $|x|$ einer Zahl x
- `n%m` berechnet den Rest von n geteilt durch m
- `sum(f(i) for i=1:n)` berechnet $\sum_{i=1}^{n} f(i)$, also beispielsweise `sum(i for i=1:10)` für $\sum_{i=1}^{10} i$

Aufgabe 5 ***

- a) Berechnen Sie unten $|13^7 - 13!|$.
- b) Berechnen Sie den Rest von 140 geteilt durch 13.
- c) Berechnen Sie die Summe der ersten 100 Quadratzahlen.

In [18]:
```
#a
```

In [19]:
```
#b
```

In [20]:
```
#c
```

Freiwillige Aufgabe *** Wenn man eine Maschine für sich rechnen lässt, sollte man sicherstellen, dass man sich auf das Ergebnis verlassen kann, d.h. dass der eingegebene Befehl für die entsprechende Anwendung ausgelegt ist. Geben Sie in Julia einmal `2^64` und einmal `2.0^64` ein. Sie werden sehen, dass im ersten Fall ein falsches Ergebnis und im zweiten Fall ein (näherungsweise) korrektes Ergebnis ausgegeben wird. Dies liegt an der Verwendung der verschiedenen Datentypen Int64 bzw. Float64 ein (im ersten Fall werden die ganzen Zahlen bit-weise in Darstellung zur Basis 2 dargestellt, und da nur 64 Bits zur Verfügung stehen, ist die Zahl 2^64 schlicht zu groß!). Wenn man ein genaues Ergebnis möchte, müsste man mit dem Datentyp BigInt rechnen. Geben Sie dazu big(2)^64 ein und vergleiche mit dem Ergebnis im Datentyp Float64. Lesen Sie sich [hier](https://einocs.julialang.org/en/v1/base/numbers/#Data-Formats) in der Julia Dokumentation etwas in die verschiedenen Datentypen ein.

Induktion

Wir kommen nun zum zweiten Teil des Sheets.

Varianten der vollständigen Induktion

Wir stellen nun noch zwei Varianten der vollständigen Induktion vor. Die erste Variante zielt auf Anwendungen, in denen die Aussage $A(n)$ nur für fast alle $n \in \mathbb{N}$ wahr ist, d.h. es gibt höchstens endlich viele $m \in \mathbb{N}$, in denen $A(m)$ falsch ist, aber ab einem bestimmen Wert $n_0 \in \mathbb{N}$ gilt $A(n)$ für alle $n \geq n_0$.

Vollständige Induktion mit allgemeinem Startindex
Um eine Aussage $A(n)$ für alle $n \in \mathbb{N}$ mit $n \geq n_0$ zu beweisen, genügt es zu zeigen

- IA: $A(n_0)$ gilt
- IS: für alle $n \in \mathbb{N}$ mit $n \geq n_0$ gilt: $A(n) \Rightarrow A(n+1)$

Die starke Induktion dagegen zielt auf Anwendungen, in denen wir $A(n+1)$ allein mit dem Wissen von $A(n)$ nicht zeigen können, sondern auch noch auf $A(n-1), A(n-2), \ldots, A(2), A(1)$ zurückgreifen müssen:

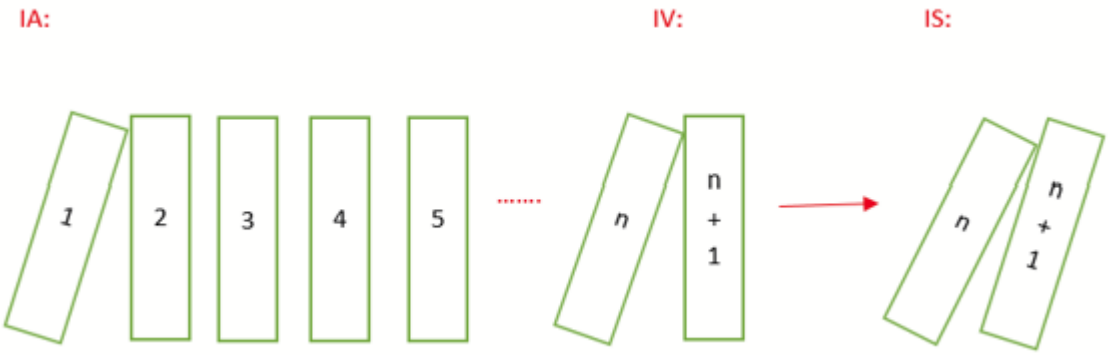
Starkes Induktionsprinzip Um eine Aussage $A(n)$ für alle $n \in \mathbb{N}$ zu beweisen, genügt es zu zeigen

- IA: $A(1)$ gilt
- IS: für alle $n \in \mathbb{N}$ gilt: falls $A(k)$ für alle $k \in \mathbb{N}$ mit $k \leq n$ gilt, dann gilt auch $A(n+1)$

Auch die starke Induktion kann mit allgemeinem Startindex n_0 verwendet werden.

Man kann sich das Induktionsprinzip als eine (unendliche) Reihe von Dominosteinen vorstellen. Damit der Induktionsschritt gilt, müssen die Dominosteine nah genug aneinander stehen, sodass sichergestellt ist, dass wenn ein Stein umfällt immer auch der nächste umfällt. Der Induktionsanfang ist unverzichtbar, damit überhaupt ein erster Stein umfällt.

Eine andere Visualisierung findet sich in folgender Grafik:



Schleifen

Schleifen werden gebraucht, um Anweisungsblöcke mehrfach zu wiederholen, solange eine Bedingung erfüllt ist oder eine Abbruchbedingung noch nicht erfüllt ist. Die erste Schleifenart, die wir kennenlernen ist, die sogenannte while-Schleife:

```
while B
    S
end
```

Dieser Block besagt, dass solange die Bedingung B erfüllt ist, Block S wiederholt werden soll.

In [21]:

```
i = 1

while i <= 10 #solange i<=10
    println(i) #gebe i aus
    i = i + 1 #erhöhe i um 1
end
```

```
1
2
3
4
5
6
7
8
9
10
```

Beim Verwenden von Schleifen sollten wir immer sicherstellen, dass die Iterationen irgendwann abbrechen. Beispielsweise würde folgender Code nicht abbrechen und endlos "Hello" ausgeben, da die Bedingung "true" immer wahr ist. Das kann den Computer gegebenenfalls zum Abstürzen bringen.

```
while true
    println("Hello")
end
```

Wir kommen nun zur zweiten Schleifenart und können die while-Schleife oben mit einer for-Schleife lesbarer schreiben:

In [22]:

```
for i = 1:10
    println(i)
end
```

```
1
2
3
4
5
6
7
8
9
```

```
10
```

Die Variable i heißt Zählvariable und hat zuerst den Wert 1 und erhöht sich dann in jeder Iteration um 1.

Vielleicht kennen Sie aus der Schule bereits die Fibonacci-Folge. Die Fibonacci-Folge $(f_i)_{i\in\mathbb{N}}$ wird rekursiv definiert wie folgt:

- $f_1 := 1$
- $f_2 := 1$
- $f_{i+2} := f_{i+1} + f_i$ für $i = 1, 2, 3, \ldots$

Fibonacci-Zahlen sind also die Elemente der Folge, welche mit den Zahlen 1 und 1 beginnt und das jeweils nächste Folgenglied die Summe der zwei vorherigen Folgenglieder ist, also ist $f_3 = f_2 + f_1 = 1 + 1 = 2$ und $f_4 = f_3 + f_2 = 2 + 1 = 3$ usw.

Mit der folgenden Schleife berechnen wir die ersten 10 Fibonacci-Zahlen:

In [23]:
```
fib_1 = 1
fib_2 = 1

for i = 1:10
    println(fib_1)
    Summe = fib_1 + fib_2 # addiere die beiden letzten Folgenglieder
    fib_1 = fib_2 # deklariere fib_1 neu, damit wir in der Folge eins weiterrücken
    fib_2 = Summe # neues letztes Folgenglied
end
```

```
1
1
2
3
5
8
13
21
34
55
```

Aufgabe 6 *** Berechnen Sie mithilfe einer while-Schleife die kleinste Fibonacci-Zahl, welche größer als 10.000 ist. *Hinweis: $\leq$ wird in Julia geschrieben, indem man die Zeichen < und = ohne Leerzeichen hintereinander setzt.*

In [24]:
```
#LÖSUNG ZU 6
```

Aufgabe 7 *** In dieser Aufgabe soll gezeigt werden, dass für alle $n \in \mathbb{N}$ gilt:

$$\sum_{k=1}^{2n-1} (-1)^{k+1} k^2 = n(2n-1)$$

- a) Stellen Sie $A(n)$ auf und zeige von Hand, dass $A(n)$ für $n = 1, 2, 3$ gilt.
- b) `[Julia]` Überprüfen Sie in Julia mithilfe einer for-Schleife, dass $A(n)$ für alle $n \leq 20$ gilt.
- c) Beweisen Sie mit vollständiger Induktion, dass $A(n)$ für alle $n \in \mathbb{N}$ gilt. Achten Sie darauf, im Induktionsschritt sauber aufzuschreiben, was zu zeigen ist (insbesondere: was sagt $A(n+1)$ aus?).
- d) (freiwillig): Finden Sie einen Beweis der Aussage, der keine vollständige Induktion verwendet.

LÖSUNG AUFGABE 7

Aufgabe 8 ***

Es sei $(a_n)_{n\geq 0}$ eine Folge von ganzen Zahlen, die wie folgt definiert ist: $a_0 = 1$, $a_1 = 4$ und

$$a_{n+2} = 2a_{n+1} - a_n.$$

- a) `[Julia]` Berechnen Sie mit Julia die ersten 10-20 Terme von (a_n) und stellen Sie eine Vermutung für a_n.
- b) Beweisen Sie Ihre Vermutung mittels Induktion.

LÖSUNG AUFGABE 8

Aufgabe 9 Beweisen Sie anhand von starker Induktion die folgende Aussage:
Jede natürliche Zahl $n \in \mathbb{N}$ mit $n \geq 2$ lässt sich als Produkt endlich vieler Primzahlen schreiben (dabei lassen wir auch zu, dass das Produkt nur aus einem einzelnen Faktor besteht).

Hinweis: im Induktionsschritt bietet sich eine Fallunterscheidung an, ob $n + 1$ eine Primzahl ist.

LÖSUNG AUFGABE 9