MuPAD Entwicklung
AutoMATH Institut
Warburger Straße 100
33095 Paderborn

www.mupad.de/schule
Email: schule@mupad.de

Eine Einführung in die MuPAD-Bibliothek analysis

Autoren: Mirko Igel
Sandra Meier-Tokic
Markus Thewes

Revision: 1.0, Oktober 1999
1.1, Februar 2001: Umstellung auf MuPAD 2.0 (frankp)
1.2, November 2002 : Umstellung auf MuPAD 2.5 (dellaere)
1.3, Mai 2004: Umstellung auf MuPAD Pro 3.0 (Kai Gehrs – schule@mupad.de)

MuPAD Version: MuPAD 3.0

Inhaltsverzeichnis

1. Vorbereitungen

Bevor wir in die Arbeit mit der Bibliothek einsteigen können, müssen wir in unserer MuPAD-Sitzung einige Vorbereitungen treffen. Zunächst lesen wir die Bibliothek ein:

- `loadlib("analysis")`

```
TRUE
```

Damit wir mit den Operatoren der Bibliothek arbeiten können, geben wir ein:

- `analysis::init():`

Um einfacher auf die Prozeduren der Bibliothek zugreifen zu können, d.h. um statt `analysis::Prozedur` einfach `Prozedur` schreiben zu können exportieren wir `analysis`:

- `export(analysis):`

2. Das Arbeiten mit Folgen in MuPAD

2.1 Definition von Folgen

Um in MuPAD die Folge $a_n = n^2$ zu definieren, geben wir in MuPAD folgenden Befehl ein:

- `a := n +-> n^2`

$$n \to n^2$$

Es wird eine reelle Folge mit dem Bezeichner `a` definiert, also eine Abbildung von **N** nach **R**.
Das 12. Folgenglied erhalten wir dann über die folgende Eingabe:

- `a(12)`

```
144
```

MuPAD liefert unmittelbar das Ergebnis (hier: `144`) zurück. Folgende Befehle jedoch führen zu Fehlern:

- `a(0)`

```
Error: analysis::Folge ist nur fuer natuerliche Werte definiert
[analysis::Folge::func_call]
```

- `a(-2.1)`

```
Error: analysis::Folge ist nur fuer natuerliche Werte definiert
[analysis::Folge::func_call]
```

Warum treten Fehler auf? Wie oben beschrieben, behandelt MuPAD die von uns definierte Abbildung wie eine reelle Folge, d.h. die Abbildung ist nur auf **N** wohldefiniert. 0 und –2.1 liegen somit nicht im Definitionsbereich der Folge `a(n)`.

Etwas anders verhält es sich mit dem Wert 12.0. Für MuPAD entspricht 12.0 nicht der natürlichen Zahl 12, sondern einem Näherungswert vom Typ `DOM_FLOAT`. Das bedeutet, daß auch die nachfolgende Eingabe zu einem Fehler führt:

- `a(12.0)`

```
Error: analysis::Folge ist nur fuer natuerliche Werte definiert
[analysis::Folge::func_call]
```

Wir müssen also darauf achten, dass beim Aufruf `a(n)` stets das `n` eine natürliche Zahl ist oder sich zu einer natürlichen Zahl auswertet, wie im Bespiel:

- `a(cos(0) + 2)`

```
9
```

2.2 Rechnen mit Folgen, Erstellung von Wertetabellen

Nachdem wir reelle Folgen definiert haben, möchten wir natürlich auch mit ihnen „rechnen“. Zunächst definieren wir uns einige Folgen:

- `a := n +-> 1/n`

$$n \to \frac{1}{n}$$

- `b := n +-> sin(n)`

$$n \to \sin(n)$$

- `c := n +-> (-1)^n / (n^2)`

$$n \to \frac{(-1)^n}{n^2}$$

- `10 * a`

$$n \to \frac{10}{n}$$

Nun aber zu den Grundrechenarten. Folgen können addiert, subtrahiert, multipliziert und dividiert werden. Hier einige Beispiele, bei denen die oben definierten Folgen benutzt werden: Eine neue Folge d wird definiert als Multiplikation der Summe von a und b mit c:

- `d := (a + b) * c`

$$n \to \frac{(-1)^n \cdot \left(\sin(n) + \frac{1}{n}\right)}{n^2}$$

Das erste Folgenglied von d erhalten wir durch:

- `d(1)`

$$-\sin(1) - 1$$

Die Quotientenfolge von a und b lautet:

- `a/b`

$$n \to \frac{1}{n \cdot \sin(n)}$$

Die Folge a wird skalar mit 3 multipliziert, was auf die Folge $a_n := \frac{3}{n}$ führt:

- `a * 3`

$$n \to \frac{3}{n}$$

Die Komposition ist für Folgen ebenfalls definiert. Hierbei müssen wir beachten, daß eine Hintereinanderausführung von Folgen im allgemeinen *nicht* möglich ist, denn Folgen haben natürliche Zahlen als Definitionsmenge. Es ist aber nur in Ausnahmefällen so, daß Folgen auch wieder natürliche Zahlen als Wertemenge haben (z.B. die Folge `n +-> n^2`). Viel wahrscheinlicher ist der Fall, daß eine Folge auch Folgenglieder aus **R\N** hat. Es ist aber möglich, Folgen *von rechts* mit Funktionen zu verknüpfen, wodurch eine Folge definiert wird:

- `b := m +-> m^2`

$$m \to m^2$$

- `sin@b`

$$m \to \sin\left(m^2\right)$$

Hingegen ist

- `b@sin`

```
Error: Illegale Komposition. [analysis::Folge::_fconcat]
```

eine unerlaubte Komposition, denn b ist eine Folge von **N** nach **N**, der Sinus hingegen bildet von **R** nach **R** ab, d.h. $b \circ \sin$ ist nicht wohldefiniert. Wir können Folgen auch radizieren und potenzieren:

- `sqrt(b)`

$$m$$

- `b^2`

$$m \to m^4$$

Als nächstes zeigen wir, wie man bequem eine Wertetabelle zu einer Folge erhalten kann:

- `wertetabelle(n +-> sin(n), n = 2..6)`

$$[[2, \sin(2)], [3, \sin(3)], [4, \sin(4)], [5, \sin(5)], [6, \sin(6)]]$$

Oder an bestimmen Zwischenstellen:

- `wertetabelle(n +-> sin(n), n = 1..2, 34, 41..43)`

$$[[1, \sin(1)], [2, \sin(2)], [34, \sin(34)], [41, \sin(41)], [42, \sin(42)], [43, \sin(43)]]$$

Die Einträge der Wertetabelle können wir mit `float` in Paare von Fließkommazahlen umwandeln:

- `float(%)`

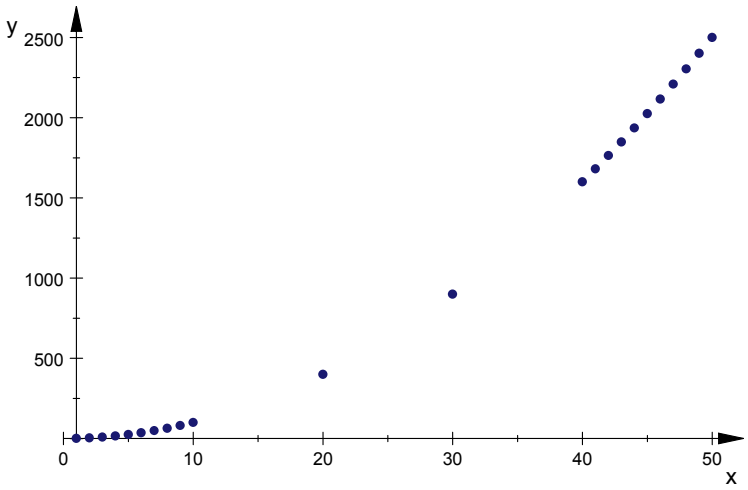
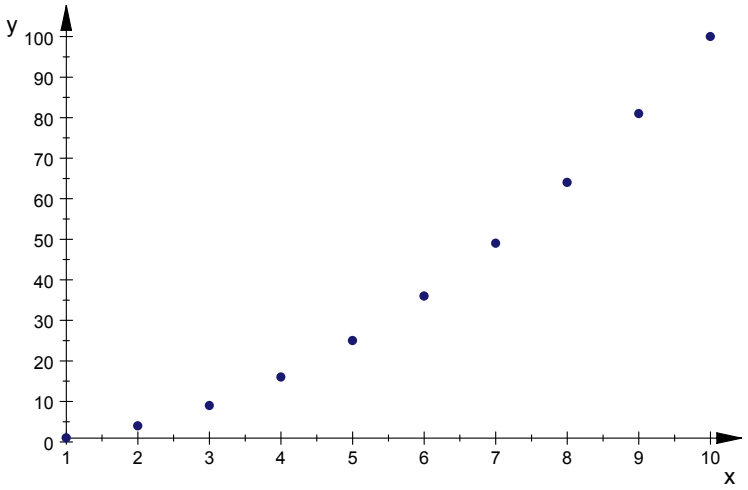
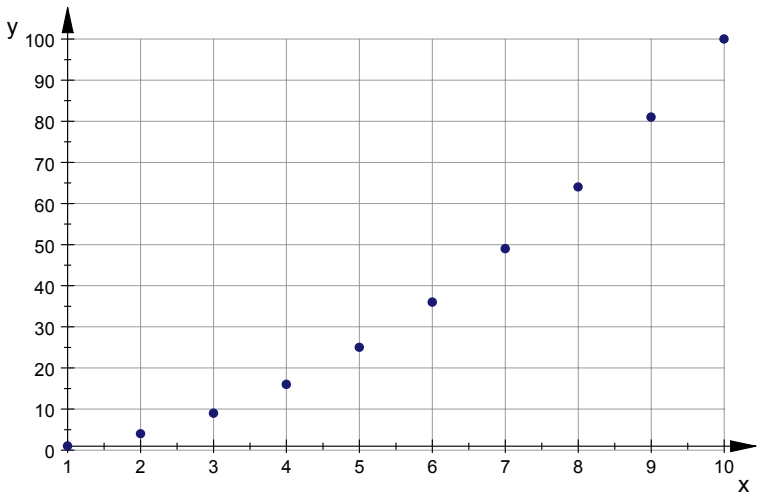
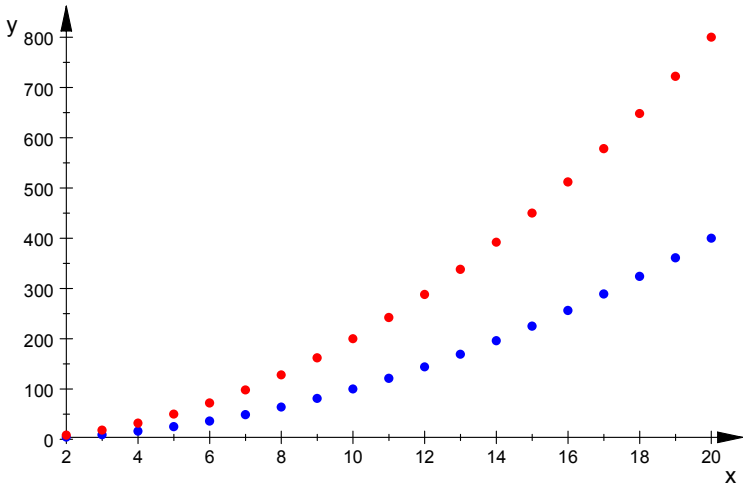
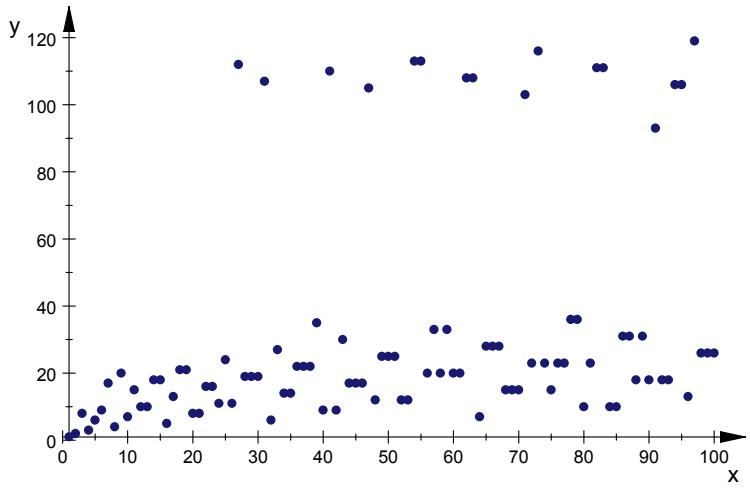
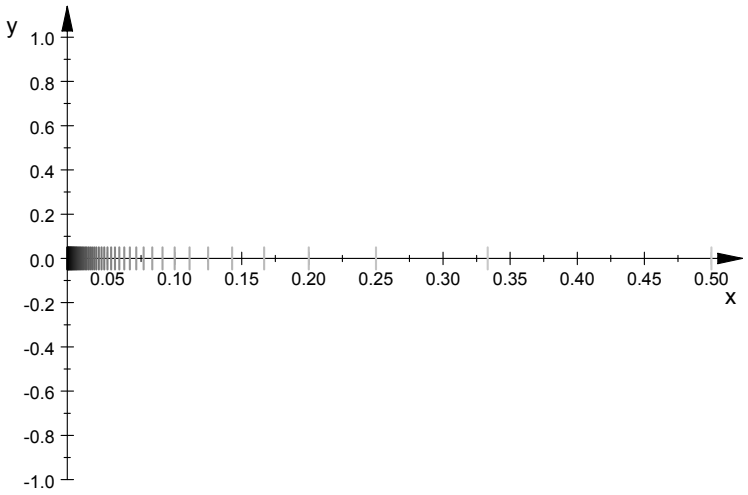
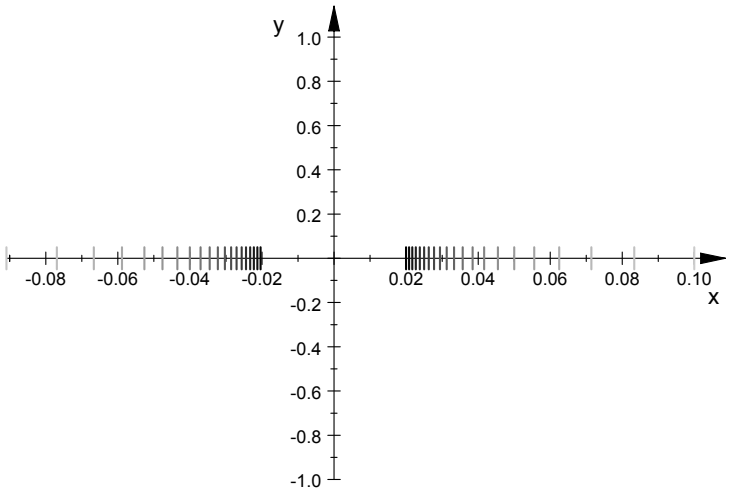
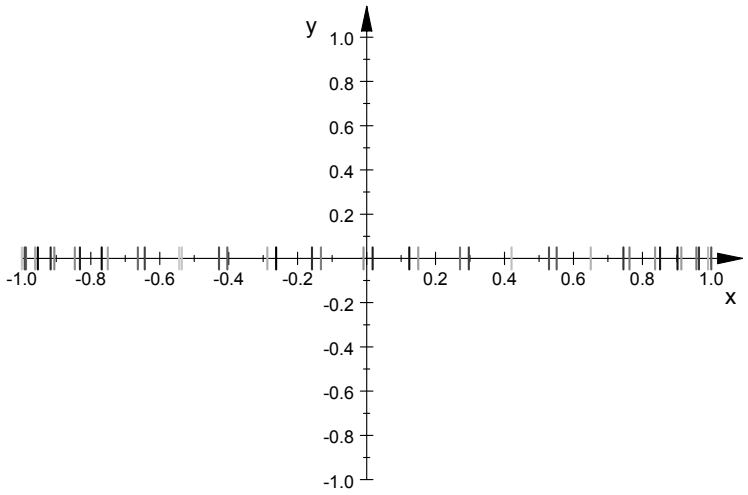
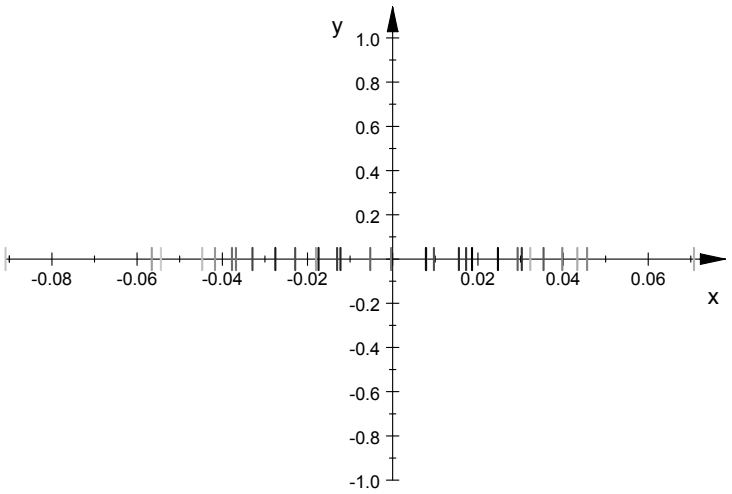
```
[[1.0, 0.8414709848], [2.0, 0.9092974268], [34.0, 0.5290826861],
[41.0, -0.1586226688], [42.0, -0.9165215479], [43.0, -0.8317747427]]
```

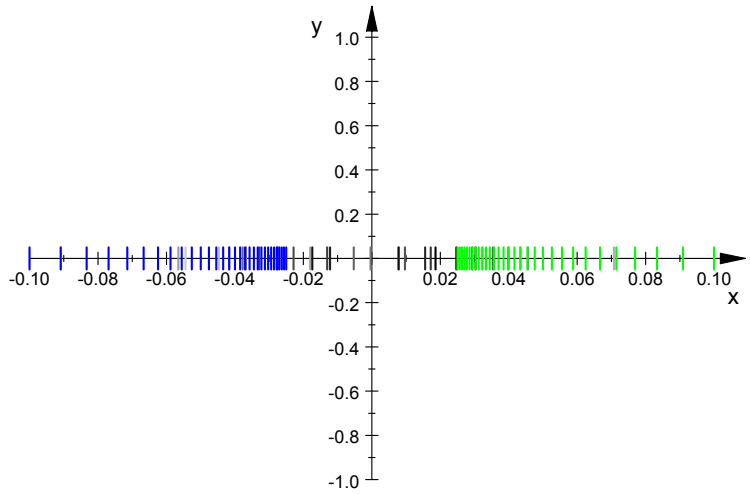
2.3 Visualisierung von Folgen

Eine Liste von 2-elementigen Listen von Zahlen, wie sie uns beispielsweise die obige Prozedur `wertetabelle` liefert, kann so umgewandelt werden, daß sie als Punktmenge graphisch angezeigt werden kann. Dazu ein Beispiel:

- `a := n +-> 1/n:`

`meineTabelle` wird in der folgenden Eingabe mit Hilfe der Prozedur `graph` in ein Objekt vom Typ `plot::PointList` umgewandelt, so dass sie mittels `plot` als Grafik ausgegeben werden kann:

- `meineTabelle := wertetabelle(a, 1..10, 20, 30, 40..50):`
- ```
  meinePunkte := graph(meineTabelle):
  plot(meinePunkte)
  ```
  ```



Keine Angst! Um Folgen zeichnen zu lassen, müssen wir nicht unbedingt über den Umweg gehen, die einzelnen Punkte vorab zu berechnen. Wir können auch mittels `graph` direkt ein Graphikobjekt erzeugen, das dann mittels `plot` auf den Bildschirm gebracht werden kann. Hierzu ein Beispiel:

- `plot(graph(a, 1..10))`



Die folgende Eingabe zeichnet zudem noch ein Gitter als Hintergrund:

- `plot(graph(a, 1..10), GridVisible)`



Bei der Angabe GridVisible handelt es sich um eine Grafikoption, die auf der Hilfeseite des MuPAD-Attributs GridVisible dokumentiert ist.

Natürlich lässt die Prozedur plot auch zu, direkt mehrere Folgen in ein gemeinsames Koordinatensystem zu zeichnen:

```
plot( graph(a, 2..20, Color = RGB::Blue),
      graph(2*a, 2..20, Color = RGB::Red) )
```



Die Folgen werden in den entsprechenden Farben angezeigt. Eine Liste der verfügbaren vordefinierten Farben erhält man übrigens mit RGB::ColorNames().

Bei gewissen Folgen ist der Verlauf der Folgenglieder aufgrund der starken Streuung nur schlecht auszumachen. In solchen Fällen bietet es sich an, mittels DrawMode=Vertical Verbindungslinien zwischen den Punkten und der x-Achse zeichnen zu lassen.

Als Beispiel definieren wir uns eine Folge, die das sogenannte „Collatz“-Phänomen[1] beschreibt: Nimm eine natürliche Zahl m und definiere die „Collatzfolge“ wie folgt:

$$c_1 := m,\ c_{n+1} := \begin{cases} 1, \text{falls } c_n = 1 & \text{(a)} \\ \dfrac{c_n}{2}, \text{falls } c_n \text{ gerade} & \text{(b)} \\ 3c_n + 1, \text{sonst.} & \text{(c)} \end{cases}$$

In MuPAD können wir diese Folge folgendermaßen in Form einer MuPAD-Prozedur formulieren:

[1] Angeblich geht diese Folge auf den deutschen Mathematiker Collatz zurück

```
collatz := proc(n, list = [])
// Baut eine Liste gem. obiger Definition auf, die zunächst leer ist.
begin
    if n = 1 then
        // Fall (a):
        return( list.[1] )
    elif n mod 2 = 0 then
        // Fall(b):
        return( collatz(n/2, list.[n]) )
    else
        // Fall(c):
        return( collatz(3*n + 1, list.[n]) )
    end
end:
```

Die Prozedur `collatz` liefert eine Liste der Glieder der „Collatzfolge“ für den Startwert `n`. Da m stets eine natürliche Zahl ist und die Collatzfolgen scheinbar immer terminieren[2], definiert folgender Befehl eine Folge:

```
a := n +-> hold( nops(collatz(n)) ):
```

Diese Eingabe leistet folgendes: `collatz(n)` gibt die Glieder der Collatzfolge zum Startwert `n` in Form einer Liste zurück. `nops` liefert die Anzahl der Elemente dieser Liste, also die „Länge“ der Collatzfolge. `hold` verzögert die Auswertung bis zum tatsächlichen Aufruf. Somit definiert `a` eine Folge, die einem Startwert `n` die „Länge“ der Collatzfolge zu diesem Startwert zuordnet:

```
collWerte := wertetabelle(a, n = 1..100):
```

```
collPunkte := graph(collWerte):
plot( collPunkte )
```



Eine weitere Möglichkeit, Folgen in MuPAD zu visualisieren, stellt uns die Prozedur `Folge::visLim` zur Verfügung. Ein Beispiel:

```
plot(Folge::visLim( n +-> 1/n, n = 2..50 ))
```

[2] Bis heute ist kein Beweis dafür erbracht, ob diese Folgen wirklich für jedes n aus **N** terminiert.



Sicher ist bei dieser Darstellung die y-Achse bedeutungslos, jedoch liefert uns MuPAD nicht die Möglichkeit, einzelne Achsen auszublenden.

Was bedeutet nun der Farbverlauf in der obigen Grafik? Um bei stark oszillierendem Verlauf unterscheiden zu können, wann welches Folgenglied auftritt, spiegelt der Farbverlauf die „Historie“ der Glieder wider (heller bedeutet dabei "früher", dunkler bedeutet "später"). So ist es möglich, einen Eindruck davon zu bekommen, wie sich die Werte der Folgenglieder „häufen“. Wenn wir die rechte Grenze des Indexbereiches „höher“ setzen, zum Beispiel von 50 auf 100, dann wird dies noch deutlicher:

- ```
  vl := Folge::visLim( n +-> (-1)^n*1/n, n = 10..50 ):
  plot(vl)
  ```




- ```
  plot( Folge::visLim(n +-> sin(n), n = 10..50) )
  ```



Hier machen wir keine Häufungen aus, möglicherweise ist diese Folge nicht konvergent. Untersuchen wir ein weiteres Beispiel:

- `a := n +-> sin(n) / n:`

Es gilt bekanntlich die Beziehung: $\frac{-1}{n} \le \frac{\sin(n)}{n} \le \frac{1}{n}$, $\forall n \in \mathbf{N}$, die wir im folgenden als "Sandwich"-Beziehung bezeichnen. Auf den ersten Blick hilft `Folge::visLim` hier nicht weiter:

- `avl := Folge::visLim(a, n = 10..40): plot(avl)`



Obige "Sandwich"-Beziehung läßt sich nun mit `Folge::visLim` wie folgt umsetzen:

- `b := n +-> 1/n:`
- ```
  untereSchranke := Folge::visLim(-b, n = 10..40, Color = RGB::Blue):
  obereSchranke  := Folge::visLim( b, n = 10..40, Color = RGB::Green):
  ```
- `plot(avl, untereSchranke, obereSchranke)`
  ```



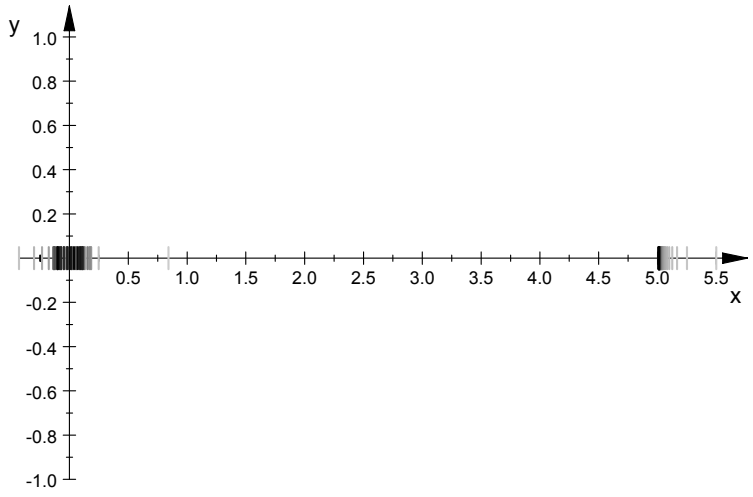
so bekommen wir ein besseres Gefühl dafür, wie sich die Folge `a` verhält.

Im folgenden Beispiel können wir noch einmal sehen, daß es möglich ist, auch *bedingte* Folgen zu definieren (siehe hierzu auch die auf Seite 7 Collatz-Folge):

- ```
  spezFolge := proc(n)
  begin
      if n mod 2 = 0 then
          // Fall (a):
          return( 5 + 1/n )
      else
          // Fall (b):
          return( sin(n)/sqrt(n) )
      end
  end:
  ```

Dies entspricht der mathematischen Definition: $a_n := \begin{cases} 5 + \frac{1}{n}, \text{falls n gerade (a)} \\ \frac{\sin(n)}{\sqrt{n}}, \text{sonst} \qquad \text{(b)} \end{cases}$

- ```
  a := n +-> hold(spezFolge(n)):
  plot( Folge::visLim(a, n = 1..100) )
  ```



Abschließend noch ein Beispiel dafür, wie rekursive Folgen realisiert werden können.

$fib_0 := 0; fib_1 := 1$

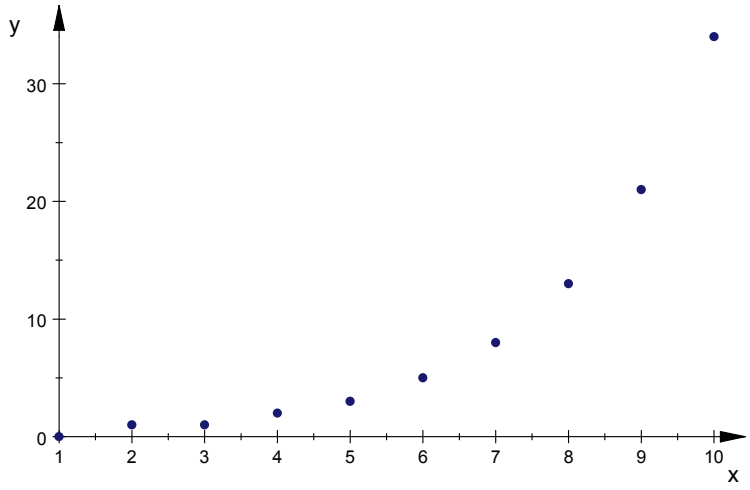
$fib_{n+1} := fib_{n-1} + fib_n, \forall 1 < n \in \mathbf{N}$

Obige Folge liefert die sogenannten *Fibonacci*-Zahlen. In MuPAD können wir das wie folgt umsetzen:

```
fib := proc(n)
    local fib_rek;
begin
    // Rekursiver Teil:
    fib_rek := proc(n, vorherige, diese)
    begin
        if n = 0 then // Bei 0 gebe vorherige zurück
            return( vorherige )
        else // Sonst gebe (vorherige + diese) = naechste
             // weiter in die Rekursion:
           return( fib_rek(n - 1, diese, vorherige + diese) )
        end
    end:
    return( fib_rek(n, 0, 1) )
end:
```

```
af := n +-> hold(fib(n - 1)):
```

```
plot(graph(af, n = 1..10))
```



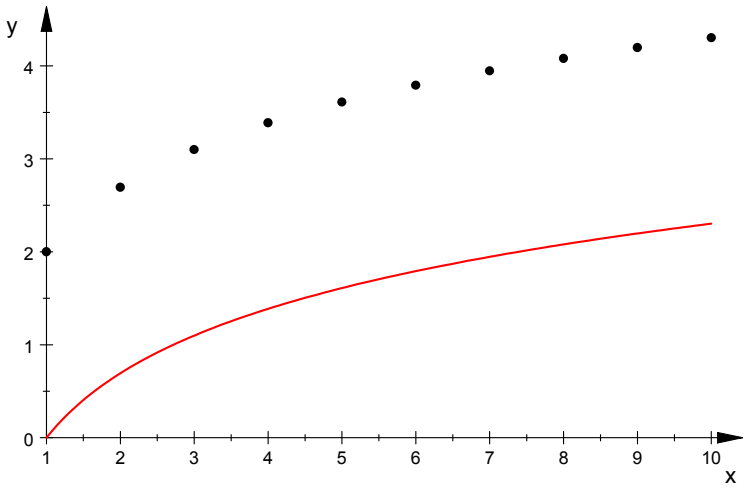
3. Das Arbeiten mit Funktionen in MuPAD

3.1 Definition und Visualisierung von Funktionen

Mit diesem Objekt werden wir reelle Funktionen definieren, also Abbildungen f: **R** -> **R**. Beginnen wir mit unserem ersten Beispiel:

```
f := x :-> ln(x): a := n +-> ln(n):
```

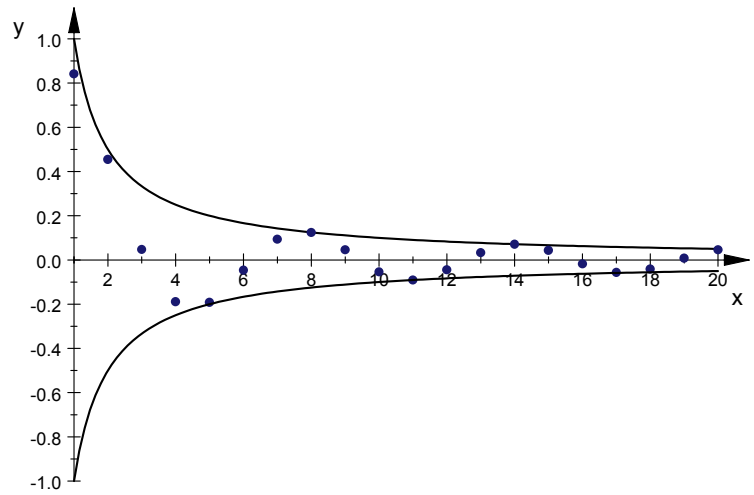
```
plot(
  graph(f, 1..10, Color = RGB::Red),
  graph(a + 2, 1..10, Color = RGB::Black)
)
```



3.2 Unterschiede zwischen Funktion und Folge

Es ist wichtig, den Unterschied zwischen Folgen und Funktionen in zu beachten: In MuPAD verwendet man bei Folgen den Operator `+->` und bei Funktionen den Operator `:->` .
Zurück zum Bild oben: MuPAD hat die Objekte in ihrer „korrekten“ Darstellung gezeichnet, der Graph der Folge `a` ist eine Menge nicht verbundener Punkte, der Graph der Funktion `f` ist als durchgängige Linie visualisiert. Wir können uns nun ein Beispiel, das wir bei den Folgen (siehe S. 5) schon behandelt hatten, mit Hilfe von Funktionen visualisieren:

```
schranke := k :-> 1/k: a := n +-> sin(n)/n:
f := x :-> sin(x)/x:
plot( graph(schranke, 1..20, Color = RGB::Black),
      graph(a, 1..20),
      graph(-schranke, 1..20, Color = RGB::Black) )
```



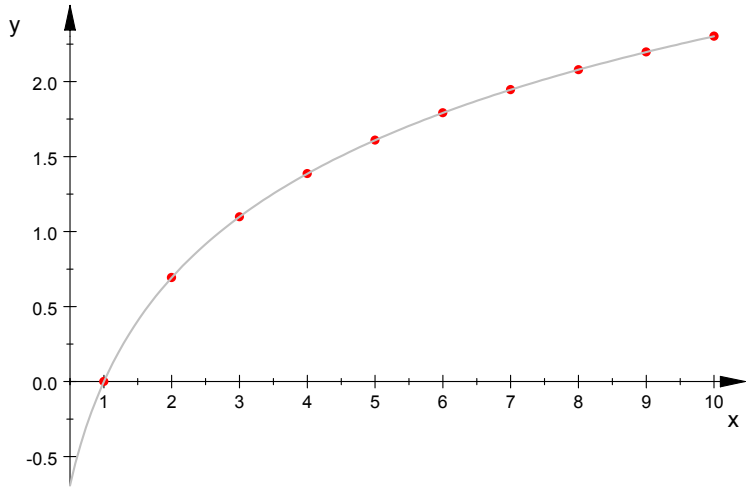
Der Verlauf des grauen Funktionsgraphen läßt vermuten, daß die „Sandwich“-Beziehung auf Seite 10 sogar für $\mathbf{R}^+$ gilt.

Ebenso wie bei den Folgen kann auch hier die Ausgabe verzögert werden, indem zunächst ein Grafikobjekt erzeugt wird. So können verschiedene Objekte mit verschiedenen Definitionsbereichen gezeichnet werden. Im folgenden Beispiel werden der Graph der auf dem ganzzahligen Bereich [1;10] definierte Folge `b` und der Graph der auf dem Bereich [0,5;10] definierten Funktion `g` gezeichnet:

- ```
  b := n +-> ln(n): g := x :-> ln(x):
  ```

- ```
  bPlot := graph(b, 1..10, Color = RGB::Red):
  gPlot := graph(g, 0.5..10, Color = RGB::Gray):
  ```

- ```
  plot(bPlot, gPlot)
  ```



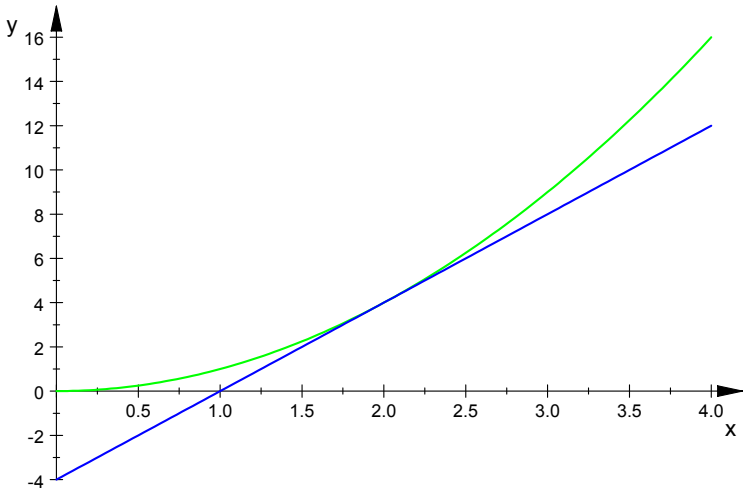
3.3 Tangenten und Geraden

Für Funktionen stellt `analysis` eine Vielzahl von vordefinierten Prozeduren zur Verfügung. Im folgenden werden wir uns auf *grundlegende* Prozeduren beschränken, mit deren Hilfe wir selber Prozeduren entwickeln können.

MuPAD bietet die Möglichkeit, bei einer gegebenen differenzierbaren Funktion die Tangente an den Graphen dieser Funktion an einer Stelle x_0 zu bestimmen:

- ```
  f := x :-> x^2: tf2 := tangente(f, Stelle = 2)
  ```

$$x \to 4 \cdot x - 4$$

- ```
  gf:=graph(f, Bereich = 0..4):
  gtf2:=graph(tf2, Bereich = 0..4):
  gf::Color:= RGB::Green:
  plot(gf,gtf2)
  ```
  ```



Über diese Prozedur erhalten wie die *allgemeine* Tangentengleichung von f wie folgt:

- `t := tangente(f, Stelle = Xo)`

$$x \to 2 \cdot x \cdot \mathrm{Xo} - \mathrm{Xo}^2$$

Entsprechend können mit der Prozedur `gerade` Geraden durch 2 Punkte oder durch Angabe einer Steigung m und einem Punkt bestimmt werden:

- `gerade(m = 2, [0, 3], x)`

$$x \to 2 \cdot x + 3$$

- `gerade([2, 3], [4, 4], x)`

$$x \to \frac{x}{2} + 2$$

3.4 Differenzenquotienten und Grenzwerte

Um Grenzwerte von Funktionen zu untersuchen, können wir eine Hintereinanderausführung von Folgen und Funktionen durchführen:

- ```
  f := x :-> sign(x - 1):
  a := n +-> 1+1/n: b := n +-> 1-1/n:
  ```
- `limit(f@a)`

  1
- `limit(f@b)`

  -1
  ```

Wie wir sehen, sind mit a und b Folgen gefunden, die zwar beide gegen 1 konvergieren, wobei die Grenzwerte von f@a und f@b jedoch nicht übereinstimmen. Damit haben wir gezeigt, daß f in x = 1 nicht stetig ist.
Um „zu Fuß“ abzuleiten, stellt die Bibliothek analysis die Prozedur differenzenQuotient zur Verfügung:

- `dQ := differenzenQuotient(x :-> x^2, Xo, h)`

$$\frac{-\mathrm{Xo}^2 + (h + \mathrm{Xo})^2}{h}$$

- `limit(dQ, h = 0)`

$$2 \cdot \mathrm{Xo}$$

An der Stelle x = 3:

- `dQ := differenzenQuotient(x :-> x^2, Xo = 3, h)`

$$\frac{(h+3)^2 - 9}{h}$$

- `limit(dQ, h = 0)`

$$6$$

Es ist ebenfalls möglich, in dem Differenzenquotienten für h eine Folge einzusetzen. Damit kann beispielsweise untersucht werden, ob eine Funktion an einer Stelle differenzierbar ist. Wir führen das an einem Beispiel vor:

- ```
  delete h:
  f := x :-> sqrt(1 + abs(x)):
  dQ := differenzenQuotient(f, Xo = 0, h):
  ```

Mit Hilfe von ersetzen können wir Substitutionen durchführen. Das nutzen wir um in dQ den Parameter h durch 1/n bzw. -1/n zu ersetzen:

- ```
  af := n +-> ersetzen(dQ, h = 1/n):
  ag := n +-> ersetzen(dQ, h = -1/n):
  ```

Nun können wir die Differenzierbarkeit von f an der Stelle x = 0 überprüfen, indem wir n gegen unendlich (bzw. h gegen 0) gehen lassen:

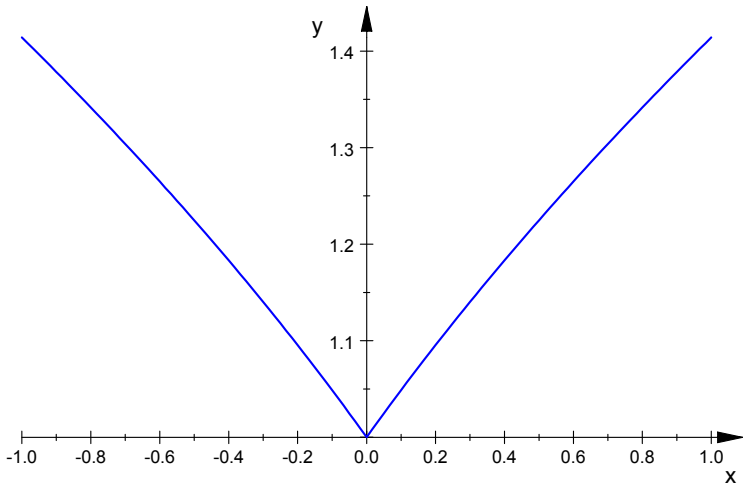
- `limit(af, n = infinity); limit(ag, n = infinity)`

$$1$$

$$-1$$

Wie man sieht, sind an der Stelle x = 0 linksseitiger und rechtsseitiger Grenzwert des Differenzenquotienten dQ nicht identisch, also ist die Funktion f dort nicht differenzierbar. Diesen Sachverhalt wollen wir noch visualisieren:

```
plot(graph(f, Bereich = -1..1))
```



3.5 Zusammengesetzte Funktionen

3.5.1 Definieren und Visualisieren von zusammengesetzen Funktionen

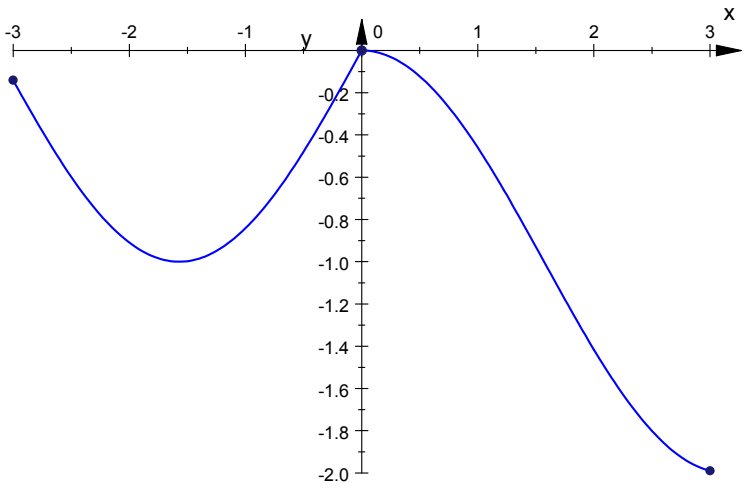
Mit zusFunktion werden zusammengesetzte Funktionen in MuPAD realisiert. In der Tat handelt es sich neben Funktion und Folge um einen weiteren Datentyp. Er unterscheidet sich jedoch in einem wichtigen Gesichtspunkt von den beiden anderen: Für Objekte vom Typ zusFunktionen sind keine Operationen wie Addition, Subtraktion, etc. definiert. Dieser Datentyp dient weitgehend zur Visualisierung. Zur analytischen Behandlung müssen die Teilfunktionen untersucht werden, so wie es auch beim Rechnen „zu fuß“ gehandhabt wird. Wir erzeugen Objekte dieses Typs mit dem &-> Operator unter Angabe der Teilfunktionen und der zugehörigen Teilintervalle. Letztere sind in der Form Dom::Interval(a,b) einzugeben, falls es sich um ein offenes Intervall handelt. Falls es sich um ein abgeschlossenes oder halboffenes Intervall handelt, sind a,b durch [a],[b] oder [a],b bzw. a,[b] entsprechend zu ersetzen. Wir geben unmittelbar einige Beispiele an:

```
fz := x &-> ( sin(x),     Dom::Interval(-infinity, 0),
              cos(x) - 1, Dom::Interval([0], infinity)   )
```

```
                +-                                    -+
                |                                      |
                |    sin(x),    x in (-infinity, 0)    |
                |                                      |
                |                                      |
                | cos(x) - 1,   x in [0, infinity)     |
                +-                                    -+
```

Beim Zeichnen können die bereits bekannten Befehle graph und plot verwendet werden (siehe u.a. S. 5 und S. 12). Beispielsweise erhalten wir den Graph von fz auf dem Bereich [-3;3] unmittelbar über:

- `plot(graph(fz, Bereich = -3..3))`



Es ist möglich, die einzelnen Funktionen und Intervalle aus einer zusFunktion zu extrahieren:

- `zusFunktion::intervalle(fz)`

$$[(-\infty, 0), [0, \infty)]$$

- `zusFunktion::funktionen(fz)`

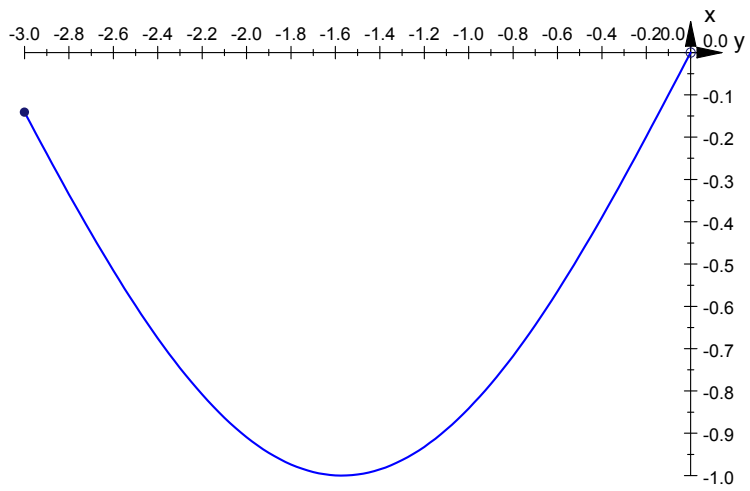
$$[x \to \sin(x), x \to \cos(x) - 1]$$

Dies erleichtert die getrennte Untersuchung der einzelnen Funktionen.
Um die Teilfunktionen in unterschiedlichen Farben zeichnen zu können, gibt es die Hilfsprozedur zusFunktionen::teilFunktionen, die die einzelnen Teilfunktionen mit ihren Definitionsbereichen extrahiert:

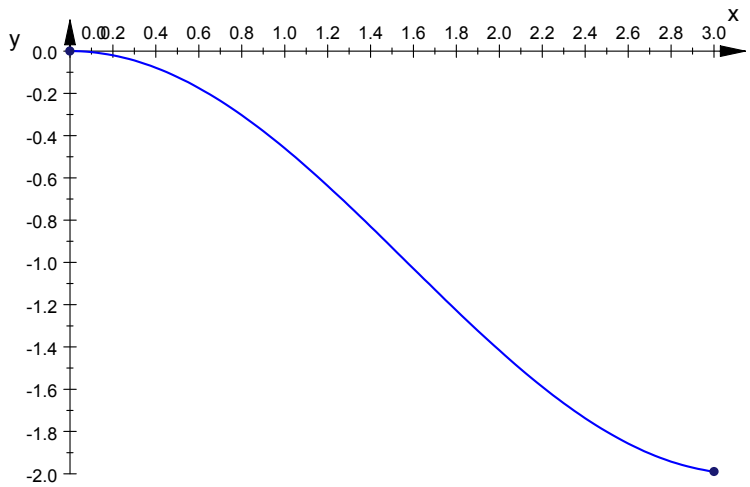
- `teilf := zusFunktion::teilFunktionen(fz)`

```
--                                                                  --
|  +-                                -+  +-                       -+  |
|  | sin(x), x in (-infinity, 0) |, | cos(x) - 1, x in [0, infinity) |  |
-- +-                                -+  +-                       -+ --
```

- `plot(graph(teilf[1], Bereich = -3..3))`



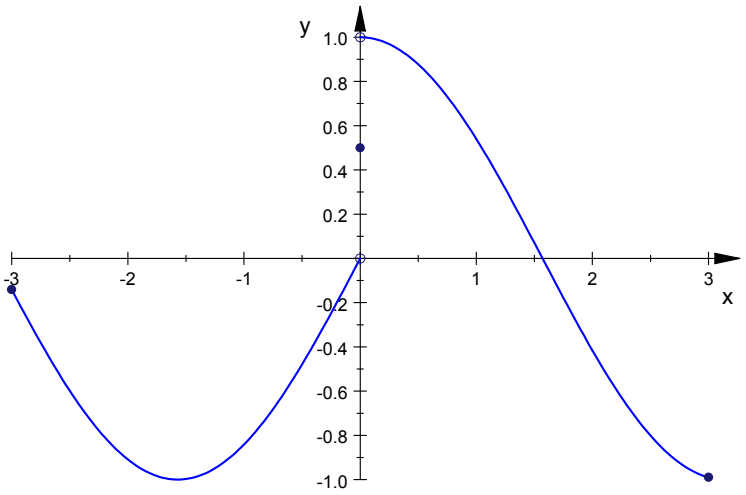
- `plot(graph(teilf[2], Bereich = -3..3))`



Zeichnet man Graphen von zusammengesetzten Funktionen, so fallen ggf. Kreise an den Intervallgrenzen auf, die entweder ausgefüllt oder unausgefüllt sind. Ein unausgefüllter Kreis deutet auf einen Punkt hin, der *nicht* zum Graphen gehört. Ein ausgefüllter Kreis bedeutet ein Punkt, der zum Graphen gehört. Dazu ein weiteres Beispiel, bei welchem die Teilfunktionen auf offenen Intervallen bzw. in einem einzelnen Punkt definiert sind:

- ```
  fz := x &-> ( sin(x), Dom::Interval(-infinity, 0),
                cos(x), Dom::Interval(0, infinity),
                0.5*x, 0 )
  ```

```
+-                                  -+
|                                    |
|  sin(x), x in (-infinity, 0)       |
|                                    |
|                                    |
|  cos(x),  x in (0, infinity)       |
|                                    |
|   0.5 x,        x in {0}           |
+-                                  -+
```

- `plot(graph(fz, Bereich = -3..3))`
```



MuPAD erkennt, ob tatsächlich eine wohldefinierte *Funktion* definiert wird, d.h. ob die Definitionsmengen paarweise disjunkt sind:

- `fz2 := x &-> (sin(x), Dom::Interval(-3, 1), x^2, Dom::Interval(0, 3))`

```
Error: Die Definitionsbereiche der Teilfunktionen ueberdecken sich
[analysis::zusFunktion::new]
```

Ebenso tritt ein Fehler auf, wenn die Funktion an einer Stelle ausgewertet werden soll, an der sie nicht definiert ist, wie im folgenden Beispiel:

- `fz2 := x &-> (sin(x), Dom::Interval(-3, 1), x^2, Dom::Interval(1, 3))`

```
+-                          -+
|                            |
|  sin(x), x in (-3, 1)      |
|                            |
|     2                      |
|    x ,    x in (1, 3)      |
+-                          -+
```

- `fz2(1)`

```
Error: Die Funktion ist an der Stelle 1 nicht definiert [analy-
sis::zusFunktion::func_call]
```

4. Ableitung und unbestimmte Integration

4.1 Ableitung

Die MuPAD-Prozedur `diff` dient dazu, die Ableitung einer Funktion zu bestimmen. `diff` kann in der Weise, wie wir es aus dem Umgang mit MuPAD gewohnt sind, auch auf Funktionen der Bibliothek `analysis` angewandt werden:

- `f := x :-> sin(x) * cos(x)`

$$x \to \cos(x) \cdot \sin(x)$$

- `diff(f)`

$$x \to \cos(x)^2 - \sin(x)^2$$

Wie wir sehen, erhalten wir durch `diff` nur die Ableitungsfunktion, aber keinerlei Hinweise auf die Berechnung. Um in „kleinen Schritten“ ableiten zu können, nutzen wir die neue Prozedur `ableitung`. Sie gibt einen Baum zurück, der die wichtigsten Informationen für das Ableiten enthält. Um diesen Baum sichtbar zu machen, benutzen wir den Befehl `expose` :

- `ablBaum := ableitung(f):`
- `expose(ablBaum)`

```
diff(x :-> cos(x)*sin(x))
 |
 `-- Produktregel
      |
      +-- (u * v)' = u'v + v'u
      |
      +-- x :-> cos(x)
      |    |
      |    `--  = u, u' = sin(x)*(-1)
      |
      +-- x :-> sin(x)
      |    |
      |    `--  = v, v' = cos(x)
      |
      `-- x :-> (sin(x)*(-1))*sin(x) + cos(x)*cos(x)
           |
           `-- Endergebnis
```

Auf die Zwischenergebnisse des Baumes `ablBaum` können wir mittels der Prozedur `knoten` (siehe Seite 33) zugreifen, eine Prozedur, die eine Liste dieser Zwischenergebnisse zurückliefert:

- `zwischenErg := knoten(ablBaum)`

$$\left["\text{Produktregel}", \cos(x), \sin(x), \frac{\partial}{\partial x}\cos(x)\cdot\sin(x) + \frac{\partial}{\partial x}\sin(x)\cdot\cos(x)\right]$$

Hierdurch erhalten wir Zugriff auf die Zwischenergebnisse und können die Rechnung nachvollziehen:

- ```
  u := zwischenErg[2]: v := zwischenErg[3]:
  u'*v + v'*u
  ```

$$\cos(x)\cdot\sin(x)' + \sin(x)\cdot\cos(x)'$$

Das Endergebnis, das im letzten Knoten des Baumes steht, wird nicht evaluiert. Erst durch die nachträgliche Verwendung der MuPAD-Prozedur `eval` erfolgen eventuelle Zusammenfassungen oder Vereinfachungen:

- `eval(zwischenErg[4])`

$$\cos(x)^2 - \sin(x)^2$$

4.2 Unbestimmte Integration

MuPAD stellt zur unbestimmten Integration die MuPAD-Prozedur `int` zur Verfügung, bei welcher als erstes Argument die zu integrierende Funktion und als zweites Argument die Variable erwartet wird, nach der integriert werden soll:

- `int(x^2*sin(x),x)`

$$2\cdot x\cdot\sin(x) - \cos(x)\cdot\left(x^2 - 2\right)$$

Diese Prozedur `int` kann aber auch auf Funktionen der Bibliothek `analysis` angewandt werden.

- `int(x :-> x^2*sin(x))`

$$x \to 2\cdot x\cdot\sin(x) - \cos(x)\cdot\left(x^2 - 2\right)$$

Die Prozedur `int` bestimmt zwar eine Stammfunktion, liefert aber keinerlei Hinweise über die Berechnungshistorie. Aus diesem Grund stellt `analysis` zur unbestimmten Integration die Prozeduren `integriere` und `intTipp` zur Verfügung, mit denen die Berechnungshistorie verfolgt und selbständig durchgeführt werden kann.
  ```

4.2.1 Die Prozedur intTipp

Die Prozedur intTipp gibt Tips und konkrete Hinweise, wie eine Stammfunktion bestimmt werden kann. Dabei ist zu beachten, daß die Integration auf heuristische Verfahrensweisen beruht, wie sie in der Schule angewendet werden. Die Verfahren beherrschen die gängigsten Schulaufgaben, es können jedoch leicht Integrationsaufgaben angegeben werden, für die diese Prozedur keine Hilfe angeben kann.
intTipp gibt einen Baum zurück, der die wichtigsten Informationen für das Integrieren enthält. Um diesen Baum sichtbar zu machen, benutzen wir den Befehl expose.
Nun einige Beispiele zum Arbeiten mit der Prozedur intTip:

- `ergebnisBaum := intTip(x^2 + sin(x)):`
- `expose(ergebnisBaum)`

```
int (x :-> sin(x) + x^2)
|
`-- Summenregel
    |
    +-- int ( S1 + S2 ) dx = int( S1 ) dx + int( S2 ) dx
    |
    +-- x :-> sin(x)
    |   |
    |   `-- = S1
    |
    `-- x :-> x^2
        |
        `-- = S2
```

Die Integrationsregel bzw. das Integrationsverfahren steht immer im ersten Knoten:

- `integrationsregel := knoten(ergebnisBaum)[1]`

```
"Summenregel"
```

Die einzelnen Summanden können wie folgt extrahiert werden.

- `ersterSummand := knoten(ergebnisBaum)[2]`

$$\sin(x)$$

- `zweiterSummand := knoten(ergebnisBaum)[3]`

$$x^2$$

Bestimmen wir nun das Integrationsverfahren für den ersten und den zweiten Summanden:

- ```
  tip_ersterSummand := intTip(ersterSummand):
  knoten(tip_ersterSummand)[1]
  ```

```
"Grundintegral"
```
```

- ```
  tip_zweiterSummand := intTip(zweiterSummand):
  knoten(tip_zweiterSummand)[1]
  ```

  ```
  "Grundintegral"
  ```

Mit Grundintegralen sind solche Integrale gemeint, die unmittelbar aus dem Hauptsatz der Infinitesimalrechnung hergeleitet werden.

`intTip` erkennt auch, ob eine Stammfunktion mit Hilfe der partiellen Integration bestimmt werden kann:

- ```
  ergebnisBaum := intTipp(x^2*sin(x)): expose(ergebnisBaum)
  ```

  ```
  int (x :-> x^2*sin(x))
  |
  `-- partielle Integration
      |
      +-- int (u v') dx = u * v - int(u' v) dx
      |
      +-- x :-> x^2
      |   |
      |   `-- = u
      |
      `-- x :-> sin(x)
          |
          `-- = v'
  ```

- ```
  integrationsverfahren := knoten(ergebnisBaum)[1]
  ```

  ```
  "partielle Integration"
  ```

- ```
  u := knoten(ergebnisBaum)[2]
  ```

  $$x^2$$

- ```
  vStrich := knoten(ergebnisBaum)[3]
  ```

  $$\sin(x)$$

In bestimmten Fällen erkennt `intTip`, ob die Stammfunktion mit Hilfe der Substitution bestimmt werden kann. Dazu ein Beispiel:

- ```
  f := x/sqrt(25-x^2)
  ```

  $$\frac{x}{\sqrt{-x^2+25}}$$

- ```
  ergebnisBaum := intTip(f): expose(ergebnisBaum)
  ```
  ```

```
int (x :-> x*(x^2*(-1) + 25)^((-1/2)))
 |
 `-- Integration durch Substitution
     |
     +-- int ( f(g(x)) * g'(x) dx = int ( f(z1) )dz1
     |
     `-- z1 = x :-> x^2*(-1) + 25
          |
          `-- Subsitutionsgleichung
```

- `substitutionsgleichung := knoten(ergebnisBaum)[2]`

$$z1 = -x^2 + 25$$

4.2.2 Die Prozedur `integriere`

Die Prozedur `integriere` ermöglicht Integrationsansätze für Funktionen durchzuführen, die mit Hilfe der partiellen Integration, der Substitution, der Polynomdivision bzw. Partialbruchzerlegung bzw. der Summenregel integriert werden können. Im folgenden gelten für die partielle Integration folgende Bezeichnungen: $\int uv'$ = u v - $\int u'v$ oder $\int vu'$ = u v - $\int v'u$.

`integriere` erwartet neben der zu integrierenden Funktion auch die Angabe der gewünschten Integrationsart in Form eines Argumentes `Modus = "Schlüsselwort"`. Hierfür kommen die Werte `PartInt, Partialbruch, Substitution` und `Summe` in Frage.
Als erstes betrachten wir ein Beispiel für die partielle Integration, müssen also `Modus = "PartInt"` setzen. Hierbei benötigen wir noch ein drittes Argument, um festzulegen auf welche Art und Weise der Integrand in das Produkt uv' zerlegt werden soll:

- `ergebnisBaum := integriere(x*cos(x), Modus = "PartInt", "u" = x):`

Die Ausgabe erfolgt wie bei `intTipp` wieder in einer Baumstruktur:

- `expose(ergebnisBaum)`

```
int (x :-> x*cos(x))
|
`-- Partielle Integration
    |
    +-- int (u v') dx = u * v - int(u' v) dx
    |
    +-- x :-> x
    |   |
    |   `-- gewaehltes u, u' = 1
    |
    +-- x :-> cos(x)
    |   |
    |   `-- = v' , v = sin(x)
    |
    +-- x :-> x*sin(x)
    |   |
    |   `-- = u v
    |
    +-- x :-> sin(x)
    |   |
    |   `-- Restintegrand u' v
    |
    `-- x :-> x*sin(x) + int(sin(x), x)*(-1)
        |
        `-- Ergebnis
```

Dem obigen Baum können wir nun sämtliche Informationen für die partielle Integration entnehmen. Neben dem Produkt `u v` und dem Restintegranden `u' v` erhalten wir auch insbesondere eine unevaluierte, d.h. unausgewertete Stammfunktion der übergebenen Funktion. Unevaluiert deshalb, damit wir die Struktur, die aus der Integrationsregel resultiert, hier im konkreten Endergebnis wiederfinden. Mit dem MuPAD-Befehl `eval` können wir aber diesen Ausdruck nachträglich evaluieren und damit weiterarbeiten:

- `u_v := knoten(ergebnisBaum)[4]`

$$x \cdot \sin(x)$$

- `restintegrand := knoten(ergebnisBaum)[5]`

$$\sin(x)$$

- `meinErgebnis := u_v - int(restintegrand, x)`

$$\cos(x) + x \cdot \sin(x)$$

Die genaue Untersuchung des Restintegranden haben wir schon in dem ersten Beispiel von `intTipp` durchgeführt. Wir überprüfen, ob die obige Rechnung korrekt war:

- `unevaluiertesErgebnis := knoten(ergebnisBaum)[6]`

$$x \cdot \sin(x) - \int \sin(x)\, \mathrm{d}x$$

- `evaluiertesErgebnis := eval(unevaluiertesErgebnis)`

$$\cos(x) + x \cdot \sin(x)$$

- `istgleich( meinErgebnis = evaluiertesErgebnis )`

```
TRUE
```

Die Prozedur `istgleich` wird auf Seite 32 beschrieben.

Neben der partiellen Integration kann `integriere` auch den Substitutionsansatz für die Integration durchführen. Dabei werden noch als drittes und viertes Argument die Substitutionsgleichung und Ihre Umkehrung erwartet :

-
```
f := exp(5*x-10):
ergebnisBaum := integriere(f, Modus="Substitution", z =5*x-10,
                           x = (z+10)/5):
```

-
```
expose(ergebnisBaum)
```

```
int (x :-> exp(x*5 + (-10)))
      |
      `-- Integration durch Substitution
          |
          +-- int (f(g( x))* g'(x)) dx = int(f (z)) dz
          |
          +-- x :-> x*5 + (-10)
          |   |
          |   `-- = z :  Substitution, x = z*1/5 + 2
          |
          +-- dx = (dz*1)/5
          |   |
          |   `-- Differential
          |
          +-- z :-> exp(z)*1/5
          |   |
          |   `-- Integrand der substituierten Funktion
          |
          +-- z :-> int (exp(z)*1/5, z)
          |   |
          |   `-- int ueber den Integrand der substituierten
          |       Funktion
          |
          `-- x :-> subs(int (exp(z)*1/5, z), z = x*5 + (-10))
              |
              `-- Ergebnis
```

Bei vielen Schulbeispielen wird die Umkehrfunktion der Substitutionsgleichung nicht benötigt. In solchen Fällen kann die Eingabe auch folgendermaßen durchgeführt werden:

-
```
f := exp(5*x-10):
ergebnisBaum := integriere(f,Modus="Substitution",z =5*x-10,"Implizit"):
```

Das liefert die identische Ausgabe wie beim vorherigen Beispiel, weshalb wir sie hier unterdrücken.

Allen Aufrufen von `integriere` ist es gemein, daß in dem ersten Parameter die Funktion steht, von der eine Stammfunktion bestimmt werden soll. Der zweite Parameter hat die Form `Modus = "Schlüsselwort"`. Jedes Integrationsverfahren besitzt ein eigenes Schlüsselwort. Der zweite Parameter läßt sich der folgenden Tabelle entnehmen:

Summenregel	`Modus = "Summe"`
Substitution	`Modus = "Substitution"`
partielle Integration	`Modus = "PartInt"`
Partialbruchzerlegung	`Modus = "PartialBruch"`

Tabelle 1: Schlüsselwörter für `integriere`

5. Weitere Funktionalität

5.1 Weitere Funktionen der Bibliothek

Die Bibliothek stellt weitere Funktionen zur Verfügung, die sich auf die oben beschriebenen Grundfunktionen zurückführen lassen. Im folgenden werden Verfahren zur Nullstellenbestimmung und zur numerischen Integration exemplarisch vorgestellt.

5.1.1 Numerischen Bestimmung von Nullstellen

Name der Funktion	Beschreibung
`analysis::nsNewton`	Führt das Newtonverfahren für eine Funktion mit einem gegebenen Startwert durch.
`analysis::nsBisektion`	Führt eine Bisektion für eine Funktion durch, wobei ein Bereich übergeben werden muss, auf dem die Funktion einen Vorzeichenwechsel hat.
`analysis::nsSekantenVerfahren`	Führt das Sekantenverfahren für eine Funktion durch. Es müssen zwei Startwerte übergeben werden.

Beispiel:

- `nsNewton(x^2-4, Start = 1.0)`

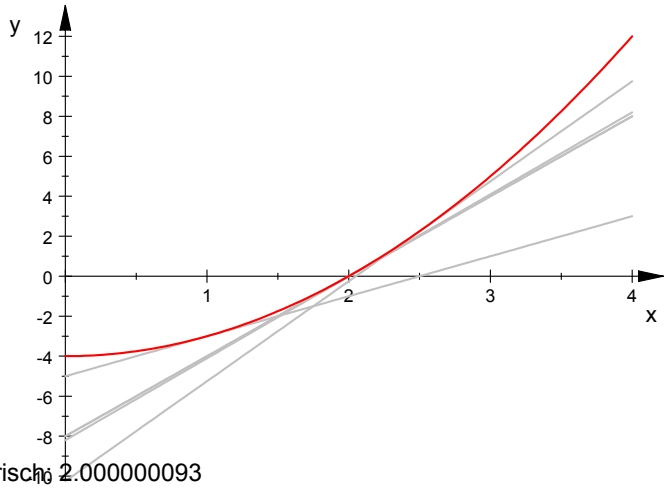
$$[1.0, 2.5, 2.05, 2.000609756, 2.000000093, 2.0]$$

5.1.2 Visualisierung zur numerischen Bestimmung von Nullstellen

Name der Funktion	Beschreibung
analysis::graphNewton	Visualisiert das Newtonverfahren für eine Funktion mit einem gegebenen Startwert.
analysis::graphBisektion	Visualisiert eine Bisektion für eine Funktion, wobei ein Bereich übergeben werden muß, auf dem die Funktion einen Vorzeichenwechsel hat.
analysis::graphSekantenVerfahren	Visualisiert das Sekantenverfahren für eine Funktion. Es müssen zwei Startwerte übergeben werden.

Beispiel: Wir wollen mit dem Newton-Verfahren eine Nullstelle der Funktion f(x) = x^2 - 4 im Bereich x = 0..4 finden, wobei wir als Start x = 1 versuchen und eine Genauigkeit von 0.001 verlangen. Die letzten beiden Argumente sind übrigens optional, d.h. man kann sie weglassen, wobei MuPAD dann hierfür eine Heuristik anwendet.

- ```
  newt := graphNewton(x^2-4, Start = 1.0, x = 0..4, 0.001):
  plot(newt)
  ```

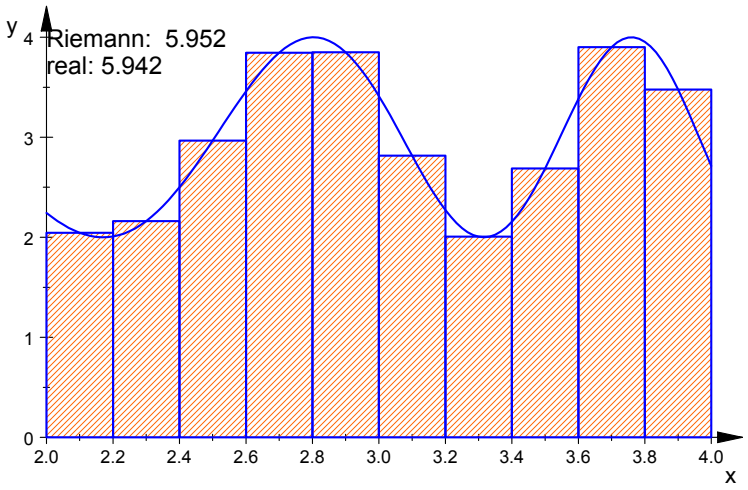


Numerisch: 2.000000093

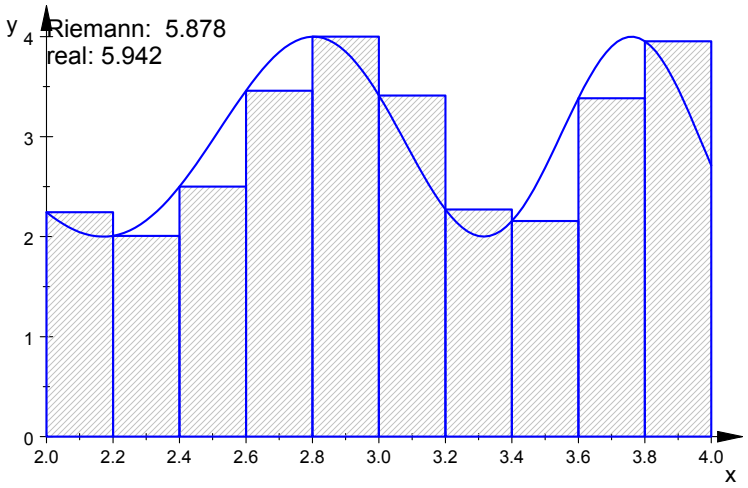
5.1.3 Numerische Integration

Hier sind einige Funktionen bereits in der MuPAD-Bibliothek student enthalten:

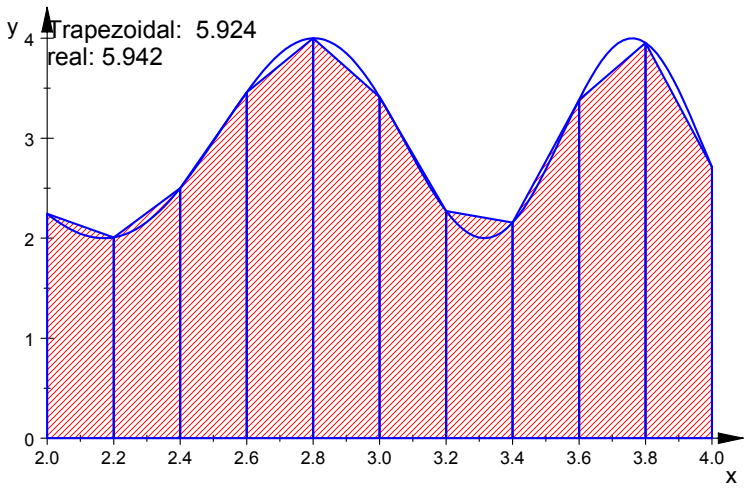
Name der Funktion	Beschreibung
student::riemann	Bestimmt einen numerischen Näherungswert für das Integral einer Funktion über einem gegebenen Bereich (kompaktes Intervall) bzw. unter Verwendung einer Liste von Stützstellen mit dem Riemannverfahren.
student::trapezoid	Bestimmt einen numerischen Näherungswert für das Integral einer Funktion über einem gegebenen Bereich (kompaktes Intervall) bzw. unter Verwendung einer Liste von Stützstellen mit dem Sehnentrapezverfahren.
student::simpson	Bestimmt einen numerischen Näherungswert für das Integral einer Funktion über einem gegebenen Bereich (kompaktes Intervall) mit der Simpsonregel.
  ```

Diese Funktionen erwarten als erstes Argument den Integranden, dann das Integrationsintervall und als letztes Argument die gewünschte Anzahl der Teilintervalle. Für das Riemannverfahren könnte das z.B. so aussehen:

- `riemann(sin(x), x = 0.0..3.0, 10)`

$$0.3 \cdot \left(\sum_{i3=0}^{9} \sin\left(0.3 \cdot i3 + 0.15\right) \right)$$

- `float(%)`

$$1.997474604$$

Man beachte, daß diese Funktionen den Datentyp `Funktion` der `analysis` Bibliothek nicht kennen!

5.1.4 Visualisierung zur numerischen Integration

Entsprechend stellt die student Bibliothek Funktionen zur Visualisierung der oben genannten Funktionen bereit:

Name der Funktion	**Beschreibung**
`student::plotRiemann`	Visualisiert das Riemannverfahren für eine Funktion auf einem gegebenen Bereich bei gegebener Anzahl von Rechtecken bzw. einer gegebenen Liste von Stützstellen.
`plotTrapezoid`	Visualisiert das Trapezverfahren für eine Funktion auf einem gegebenen Bereich bei gegebener Anzahl von Trapezen bzw. einer gegebenen Liste von Stützstellen.
`student::plotSimpson`	Visualisiert das Simpsonverfahren für eine Funktion auf einem gegebenen Bereich bei gegebener Anzahl von Parabeln.

Diese Funktionen erwarten als erstes Argument den Integranden, dann das Integrationsintervall und als letztes Argument die gewünschte Anzahl der Teilintervalle. Für das Riemannverfahren kann noch ein optionales Argument `Left`, `Middle` oder `Right` angegeben werden. Hierdurch wird die Höhe des jeweiligen Rechtecks durch den Funktionswert links, mittig oder rechts im zugehörigen Teilintervall bestimmt. Wird dieses Argument weggelassen, so tritt automatisch die Voreinstellung `Middle` in Kraft.

Beispiele:

- ```
  trap := student::plotRiemann(sin(x^2)+3, x = 2..4, 10):
  plot(trap)
  ```
  ```



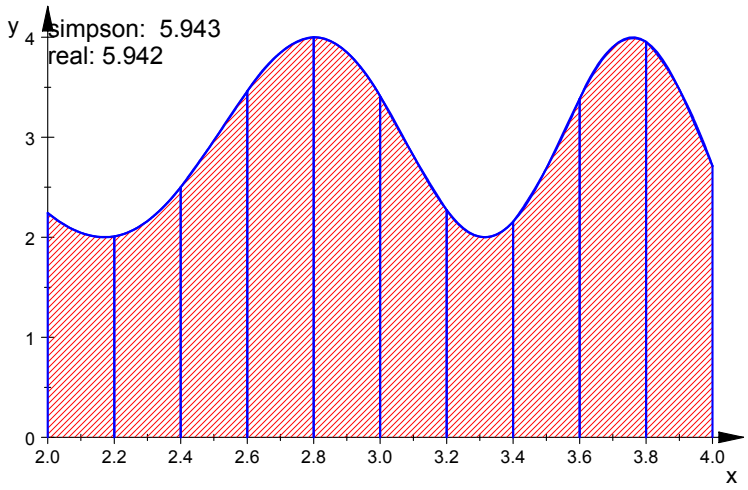
- `trap := student::plotRiemann(sin(x^2)+3, x = 2..4, 10, Left): plot(trap)`



- `trap := student::plotTrapezoid(sin(x^2)+3, x = 2..4, 10): plot(trap)`



- `trap := student::plotSimpson(sin(x^2)+3, x = 2..4, 10): plot(trap)`



5.2 Nützliches

Name der Funktion	Beschreibung
analysis::istgleich	Überprüft, ob zwei Ausdrücke mathematisch äquivalent sind.
analysis::knoten	Liefert eine Liste mit Knotenwerten für eine spezielle Baumstruktur zurück.

5.2.1 Die Prozedur istgleich

Im allgemeinen ist das Testen auf mathematische Äquivalenz in Computer-Algebra-Systemen ein unentscheidbares Problem. Eine Prozedur zum Vereinfachen von Ausdrücken - wie z.B. die MuPAD-Prozedur simplify - ist nicht so stark, als das sie zum "universellen" Testen auf mathematische Äquivalenz eingesetzt werden kann. Aus diesem Grund stellt die Bibliothek analysis die Prozedur istgleich zur Verfügung, die die gängigsten Äquivalenzbeziehungen verifizieren kann. Hier ein Beispiel:

- `term1 := arctan(x) + arctan(y)`

$$\arctan\left(x\right)+\arctan\left(y\right)$$

- `term2 := arctan( (x+y)/(1-x*y) )`

$$\arctan\left(\frac{x+y}{-x\cdot y+1}\right)$$

Die obigen beiden Termen sind äquivalent zueinander, was uns die Prozedur istgleich bestätigt:

- `istgleich ( x :-> term1 = x :-> term2 )`

TRUE

Liegt keine Äquivalenz vor, so wird FALSE zurückgegeben:

- `istgleich( x = x+3 )`

FALSE

5.2.2 Die Prozedur knoten

Die Bildschirmausgaben von ableitung, integriere und intTip (siehe Kapitel 4. auf Seite 21) erfolgen in einer speziellen Baumstruktur. Um die Knotenwerte dieser speziellen Baumstruktur auszulesen, stellt analysis die Prozedur knoten zur Verfügung. Diese liefert eine Liste mit den Knotenwerten zurück, wie in dem folgenden Beispiel:

- `ergebnisBaum := intTip(4*sin(x)): expose(ergebnisBaum)`

```
int (x :-> sin(x)*4)
 |
 `-- Faktorregel
      |
      +-- int ( s * f(x) ) dx = s * int( f(x) ) dx, s aus R
      |
      +-- 4
      |    |
      |    `-- = s
      |
       `-- sin(x)
           |
           `-- = f(x)
```

- `knoten(ergebnisBaum)`

$$\left["Faktorregel",\ 4,\ \sin(x)\right]$$

5.3 Interessante Phänomene

Diese i-malige Hintereinanderausführung des Cosinus, die wir für einen festen Startwert x in Form von $\cos_i(x) := (\overset{i}{\underset{k=1}{\circ}} \cos)(x)$ schreiben können, soll in MuPAD umgesetzt werden. Dazu schreiben wir uns eine MuPAD Prozedur mit den Namen cosi:

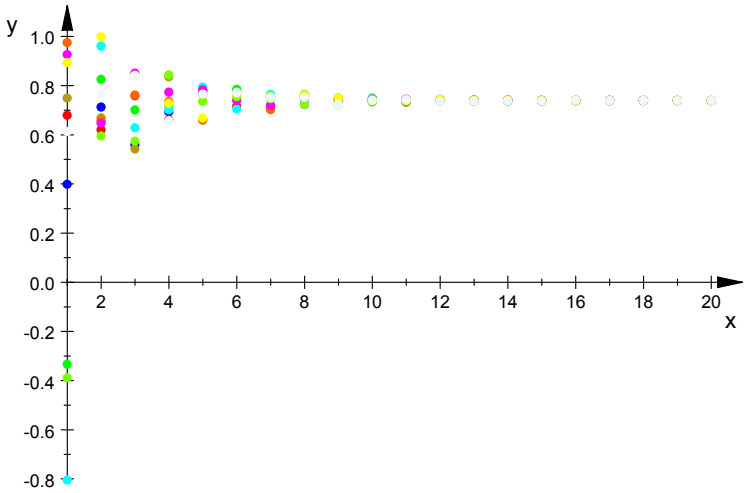
- `cosi := n -> (cos@@n)( float(random()) ):`

und definieren die Folge $n \mapsto \mathrm{cosi}(n)$:

- `a := n +-> hold(cosi(n)):`

Wir lassen uns den Graphen der Folge a auf dem Bereich [1;20] zeichnen:

```
plot(
   graph(a, 1..20, Color = RGB::ColorList[i]) $ i=1..10
)
```



Eine interessante Frage in diesem Zusammenhang, nämlich ob `a` einen Fixpunkt besitzt, kann nun vom interessierten Leser mit MuPAD behandelt werden.

Index

N

P

R

S

T

U

V

W

Z