Computermathematik (für Informatik)

4. Übungsblatt
3. 12. 2008

Die heutigen Übungen sollen mit dem Computeralgebrasystem **Sage** gelöst werden.

Die Lösung der Beispiele soll auf möglichst kompakte Weise erfolgen. Wenn zum Beispiel eine Funktion für mehrere Werte berechnet werden soll, soll das mittels einer geeigneten Schleifen oder Listen Operation erfolgen, und **nicht** alle Werte einzeln eingetippt werden.

Zwischenergebnisse welche in einem weiteren Berechnungsschritt benötigt werden, sollen in eine Variable gespeichert und weiterverwendet werden (**nicht** neu eintippen).

Beispiel 28

Wir betrachten die durch

$$a_0 = 1 \qquad a_{n+1} = \frac{1}{4} a_n^2 + 1$$

rekursiv gegebene Folge. Aus Analysis T1 (Beispiel 15b) wissen Sie dass $\lim_{n\to\infty} a_n = 2$

1. Bestimmen Sie die ersten 20 Folgenglieder mit 1000 Bits Genauigkeit.
2. Bestimmen Sie das erste Folgenglied a_{n_ε} dass von 2 einen Abstand von weniger als ε hat. Für $\varepsilon = 10^{-2}, 10^{-3}, 10^{-4}$. Geben Sie auch jeweils den Index n_ϵ an.
3. Bestimmen Sie das erste Folgenglied a_{n_ε} das vom nächsten Folgenglied $a_{n_\varepsilon+1}$ einen Abstand von weniger als ε hat. Für $\varepsilon = 10^{-6}, 10^{-7}, 10^{-8}$. Geben Sie auch jeweils den Index n_ε an.
4. Wir bilden eine neue Folge b_n durch

   $$b_n = a_{n+2} - \frac{(a_{n+2} - a_{n+1})^2}{a_{n+2} - 2a_{n+1} + a_n},$$

   dann hat b_n genau den selben Grenzwert wie die Folge a_n.

   Berechnen Sie Punkt (b) für die Folge b_n. Was fällt auf?

Iterative Lösung

```
def bsp28a(n):
   R = RealField(1000)
   a = R(1)
   print a
   for n in xrange(n - 1):
       a = 1/4 * a^2 + 1
       print a
```

```
bsp28a(20)
```

```
1.00000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000
1.25000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000
1.39062500000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
```

0000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000
1.48345947265625000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000
1.55016300175338983535766601562500000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000
1.60075133300127002444236679057354422184289433062076568603515625000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000
1.64060120752633571890953101408328742609406818997365207796747684796
5073901256044127889080486148998261342057958245277404785156250000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000
1.67289308053421772018435197500449088576613505000717940436724974446
1678315824660760875205075499913723500729864459189799560449423298904
8434838212987640845437189971748783180567927728458254604783807146241
9653981255491415058855864117504097521305084228515625000000000000000
00000000000000000000000000000
1.69964281472481616372623651477441453175131337980062885583595585489
1400926520854045808061853547804313714597361852360553243423075209518
5789553507523631238431436264081254488579878783894925658818658215913
0548763356799755745124470337242844543359674566259315555036947032460
05496569726996407447095673968
1.72219642441142394120551229560007386939064415510573221494507646061
5091212580094518944890798746233130127729718134341214498856032992140
0240590135037557117225346931961622788028984464720589789111822061544
3832710792958912636499240503251184111017461446111433161905872393679
88470808229754141795917912824
1.74149013106387336418837717713450828555063371301840876002513575632
2681126243659781861247440355181153526234625779939908217834733613240
1855023259446126464070878199663698382813148657949859352109074620405
4937984708824489999682972027429095707807779695216095106866284171341
00548017715677768536168922242
1.75819696914821670693493379110587778542315320856700980005710105093
9353948380008076065188914754880366603070170096705292879367998752963
1749463809176591087588975735506066797076012531728090216771497443296
7336260664546375870408100970728574979033843684730552184424831359362
39756204346029639879827960250
1.77281414558049382269957298392267575284913812659747030418739124434
8037750425970098847306496162486183530965799816337797333557007884254
5106844826732577967829351594897420602258974212423948043165590051657
5434787437008036655085887520747334393308277860715808285097425691710
11221234780913743289851600433
1.78571749869257408631770580522707939917469795102307891534172645084
5461092000632752735683937817068235707518567591309940557465457715369
8068192684800998240408748643756553119972275251807181527640840563158
3469433781077430842402916064717315943399276296773010800783617940698
85268661025909162206006729865
1.79719674628421583341935953806478902613204595833849903227797974621
9167442703232916837678695851270968986767573436229044028181446189959
4093046125557728330415583870319692544572198156929889852287307969647
8196660374124003417209920103833020883606933090586735850703819042206
07303751495722268993104010287
1.80747903621364301451167016759861626998326619738136842772294874740

754682962409282059859791666992694343428067720109196820843781341025265688477603641462105455072602295316836334512124152017981116515673981437325201405860356428937814510547542576064019962647781923860080924652585323020121988578719552201.816745116587949958970254637052961112756913212464061140792405578576297256040360344470409284121292058038472880899621764254708282127581700397456910805626534562178431628291626192643410289962100655941449092082543409288571475812381579098530082376099648192122327840457455152044426644206578446497500463401.825140704661540972292226539134440895257169067181200220136179183465848762181365115977664034811611661262637451190445817225967539746948415017641005712049837952088058113861541360226563056282094530634155223545108849685539577316182378666434439064986932641868021372571301743357890131765297599411574765701.832784647953106582056548534186994613500976027967171669022893376563350749227461803234024424144917652879405778142891207703898674033839921759271524952124921974218894692690225755036296744767578416972518750443276102352661078220876502930726199734935027463260630228314075792067034676347156154718935776301.839774891443148207751046958824018248416981618429475009977556082228941516582532578913230426582210142501355307184255286389640933733449435875612383214464585944090380846421278315983962937590093735417724752929423074016513636681731246812552477899487189768057200824220638108945376996898416800803876784 0
```
def bsp28b(epsilon):
   a = 1.0
   n = 0
   while abs(a - 2) > epsilon:
       n += 1
       a = 1/4.0 * a^2.0 + 1.0
   return a, n
```

```
for i in range(-2, -5, -1):
   epsilon = 10^i
   a_n, n = bsp28b(epsilon)
   html("$\\varepsilon = 10^{%d},\quad n_\\varepsilon = %i,\quad
a_{n_\\varepsilon} = %f$" % (i, n, a_n))
```

$\varepsilon = 10^{-2}, \quad n_\varepsilon = 392, \quad a_{n_\varepsilon} = 1.990019$

$\varepsilon = 10^{-3}, \quad n_\varepsilon = 3989, \quad a_{n_\varepsilon} = 1.999000$

$\varepsilon = 10^{-4}, \quad n_\varepsilon = 39987, \quad a_{n_\varepsilon} = 1.999900$

```
def bsp28c(epsilon):
   a_0 = 1.0
   a_1 = 1/4 * a_0 ^ 2 + 1
   n = 0
   while abs(a_0 - a_1) > epsilon:
       n += 1
       a_0, a_1 = a_1, 1/4 * a_1^2 + 1
   return a_0, n
```

```
for i in range(-6, -9, -1):
   epsilon = 10^i
   a_n, n = bsp28c(epsilon)
   html("$\\varepsilon = 10^{%d},\quad n_\\varepsilon = %i,\quad
a_{n_\\varepsilon} = %f$" % (i, n, a_n))
```

$\varepsilon = 10^{-6}, \quad n_\varepsilon = 1990, \quad a_{n_\varepsilon} = 1.998000$

$\varepsilon = 10^{-7}, \quad n_\varepsilon = 6314, \quad a_{n_\varepsilon} = 1.999368$

$\varepsilon = 10^{-8}, \quad n_\varepsilon = 19988, \quad a_{n_\varepsilon} = 1.999800$

```
def bsp28d(epsilon):
   a_0 = 1.0
   a_1 = 1/4 * a_0 ^ 2 + 1
   a_2 = 1/4 * a_1 ^ 2 + 1

   b = a_2 - (a_2 - a_1)^2 / (a_2 - 2 * a_1 + a_0)

   n = 0
   while abs(b - 2) > epsilon:
       n += 1
       a_0, a_1, a_2 = a_1, a_2, 1/4.0 * a_2 ^ 2.0 + 1.0
       b = a_2 - (a_2 - a_1)^2 / (a_2 - 2 * a_1 + a_0)
   return b, n
```

```
for i in range(-2, -5, -1):
   epsilon = 10^i
   b_n, n = bsp28d(epsilon)
   html("$\\varepsilon = 10^{%d},\quad n_\\varepsilon = %i,\quad
b_{n_\\varepsilon} = %f$" % (i, n, b_n))
```

$\varepsilon = 10^{-2}, \quad n_\varepsilon = 192, \quad b_{n_\varepsilon} = 1.990028$

$\varepsilon = 10^{-3}, \quad n_\varepsilon = 1990, \quad b_{n_\varepsilon} = 1.999000$

$\varepsilon = 10^{-4}, \quad n_\varepsilon = 19982, \quad b_{n_\varepsilon} = 1.999900$

Lösung mit Generatoren

Ein Generator ist ein Python Konstrukt, dass eine Folge von Werten erzeugt.

Man erzeugt einen Generator, wie eine normale Python Funktion, allerdings wird das Schlüsselwort **yield** anstatt **return** verwendet um Werte zurückzugeben.

Wir definieren uns einen Generator für die natürlichen Zahlen:

```
def count():
   n = 0
   while True:
       yield n
       n += 1
```

Aufrufen der Funktion **count()** erzeugt jetzt ein Generator Objekt.

```
nat = count()
type(nat)
```

```
<type 'generator'>
```

Mit der Methode **.next()** berechnet man jeweils einen Wert des Generators:

```
nat.next(); nat.next(); nat.next()
```

```
0
1
2
```

Mit der Funktion **iter** kann man Listen in Generatoren umwandeln.

```
list_g = iter([1..4])
```

Wenn der Generator kein neues Element mehr erzeugen kann, bekommt man die Fehlermeldung **StopIteration**.

```
list_g.next(); list_g.next(); list_g.next()
```
```
1
2
3
```

Eine **Generator Comprehension** ist das äquivalent einer List Comprehension nur eben für Generatoren.

Hier erzeugen wir uns aus dem Generator für die natürlichen Zahlen einen neuen Generator für die ungeraden Quadratzahlen.

```
odd_squares = (x^2 for x in count() if x % 2 == 1)
```

```
odd_squares.next(); odd_squares.next(); odd_squares.next()
```
```
1
9
25
```

Die Funktion **take** nimmt einen Generator **g** und eine natürliche Zahl **n**, und gibt eine Liste der nächsten n Elemente des Generators zurück.

```
def take(g, n):
   l = []
   for i in xrange(n):
       try:
           l.append(g.next())
       except StopIteration:
           break
   return l
```

```
take(odd_squares, 5)
```
```
[49, 81, 121, 169, 225]
```
```
g = iter([1..10])
```

```
take(g, 3); take(g, 3); take(g, 3); take(g, 3)
```
```
[1, 2, 3]
[4, 5, 6]
[7, 8, 9]
[10]
```

Zurück zum Beispiel: Wir definieren uns einen Generator für die Folge:

$$a_0 = 1 \quad a_{n+1} = \frac{1}{4}a_n^2 + 1.$$

RDF ist der Typ der Fließkommazahlen mit doppelter Präzision. Das enspricht in etwa dem Type **double** in C.

```
def sequence(ring = RDF):
   a = ring(1)
   while True:
       yield a
       a = 1/4 * a^2 + 1
```

```
an = sequence(RR)
```

```
an.next(); an.next(); an.next(); an.next();
```
```
1.00000000000000
1.25000000000000
1.39062500000000
```

1.48345947265625

```
alist = take(sequence(ring = RealField(1000)), 20)
for a in alist:
   print a
```

```
1.00000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000
1.25000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000
1.39062500000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000
1.48345947265625000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000
1.55016300175338983535766601562500000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000
1.60075133300127002444236679057354422184289433062076568603515625000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000
1.64060120752633571890953101408328742609406818997365207796747684796
5073901256044127889080486148998261342057958245277404785156250000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000
1.67289308053421772018435197500449088576613505000717940436724974446
1678315824660760875205075499913723500729864459189799560449423298904
8434838212987640845437189971748783180567927728458254604783807146241
9653981255491415058855864117504097521305084228515625000000000000000
00000000000000000000000000000
1.69964281472481616372623651477441453175131337980062885583595585489
1400926520854045808061853547804313714597361852360553243423075209518
5789553507523631238431436264081254488579878783894925658818658215913
0548763356799755745124470337242844543359674566259315555036947032460
05496569726996407447095673968
1.72219642441142394120551229560007386939064415510573221494507646061
5091212580094518944890798746233130127729718134341214498856032992140
0240590135037557117225346931961622788028984464720589789111822061544
3832710792958912636499240503251184111017461446111433161905872393679
88470808229754141795917912824
1.74149013106387336418837717713450828555063371301840876002513575632
2681126243659781861247440355181153526234625779939908217834733613240
1855023259446126464070878199663698382813148657949859352109074620405
4937984708824489999682972027429095707807779695216095106866284171341
00548017715677768536168922242
1.75819696914821670693493379110587778542315320856700980005710105093
9353948380008076065188914754880366603070170096705292879367998752963
1749463809176591087588975735506066797076012531728090216771497443296
```

```
    7336260664546375870408100970728574979033843684730552184424831359362
    3975620434602963987982796025 0
    1.772814145580493822699572983922675752849138126597470304187391244 34
    8037750425970098847306496162486183530965799816337797333557007884254
    5106844826732577967829351594897420602258974212423948043165590051657
    5434787437008036655085887520747334393308277860715808285097425691710
    1122123478091374328985160043 3
    1.785717498692574086317705805227079399174697951023078915341726450 84
    5461092000632752735683937817068235707518567591309940557465457715369
    8068192684800998240408748643756553119972275251807181527640840563158
    3469433781077430842402916064717315943399276296773010800783617940698
    8526866102590916220600672986 5
    1.797196746284215833419359538064789026132045958338499032277979746 21
    9167442703232916837678695851270968986767573436229044028181446189959
    4093046125557728330415583870319692544572198156929889852287307969647
    8196660374124003417209920103833020883606933090586735850703819042206
    0730375149572226899310401028 7
    1.807479036213643014511670167598616269983266197381368427722948747 40
    7546829624092820598597916669926943434280677201091968208437813410252
    6568847760364146210545507260229531683633451212415201798111651567398
    1437325201405860356428937814510547542576064019962647781923860080924
    6525853230201219885787195522 0
    1.816745116587949958970254637052961112756913212464061140792405578 57
    6297256040360344470409284121292058038472880899621764254708282127581
    7003974569108056265345621784316282916261926434102899621006559414490
    9208254340928857147581238157909853008237609964819212232784045745515
    2044426644206578446497500463 4
    1.825140704661540972292226539134440895257169067181200220136179183 46
    5848762181365115977664034811611661262637451190445817225967539746948
    4150176410057120498379520880581138615413602265630562820945306341552
    2354510884968553957731618237866643443906498693264186802137257130174
    3357890131765297599411574765 7
    1.832784647953106582056548534186994613500976027967171669022893376 56
    3350749227461803234024424144917652879405778142891207703898674033839
    9217592715249521249219742188946926902257550362967447675784169725187
    5044327610235266107822087650293072619973493502746326063022831407579
    2067034676347156154718935776 3
    1.839774891443148207751046958824018248416981618429475009977556082 22
    8941516582532578913230426582210142501355307184255286389640933733449
    4358756123832144645859440903808464212783159839629375900937354177247
    5292942307401651363668173124681255247789948718976805720082422063810
    8945376996898416800803876784 0
```

```
bn = enumerate(sequence())
```

```
bn.next(); bn.next(); bn.next();
```

```
    (0, 1.0)
    (1, 1.25)
    (2, 1.390625)
```

```
def seq_lim(g, epsilon, limit):
   h = enumerate(g)
   n, a = h.next()

   while abs(limit - a) >= epsilon:
      n, a = h.next()

   return n, a
```

```
def seq_accel(g):
   a_0 = g.next()
```

```
    a_1 = g.next()
    a_2 = g.next()

    while True:
        b = a_2 - (a_2 - a_1)^2 / (a_2 - 2*a_1 + a_0)
        yield b
        a_0, a_1, a_2 = a_1, a_2, g.next()
```

```
epsilon = [10^(-2), 10^(-3), 10^(-4)]
for e in epsilon:
    print seq_lim(sequence(), e, 2), seq_lim(seq_accel(sequence()),
e, 2)
```

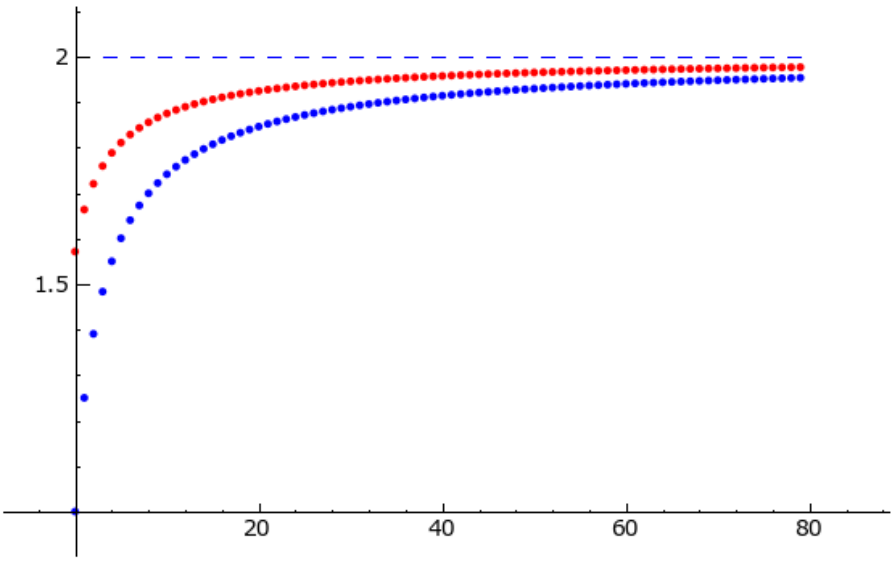
```
(392, 1.99001896664) (192, 1.99002825175)
(3989, 1.99900001548) (1990, 1.99900043432)
(39987, 1.99990000091) (19982, 1.99990000001)
```

```
def seq_diff(g, epsilon):
    h = enumerate(g)
    n0, a0 = h.next()
    n1, a1 = h.next()

    while abs(a0 - a1) >= epsilon:
        n0, a0 = n1, a1
        n1, a1 = h.next()

    return n0, a0
```

```
epsilon = [10^(-4), 10^(-6), 10^(-8)]
for e in epsilon:
    print seq_diff(sequence(), e), seq_diff(seq_accel(sequence()),
e)
```

```
(192, 1.98000641036) (134, 1.98593746254)
(1990, 1.99800036876) (1404, 1.99858609494)
(19988, 1.99980000671) (9619, 1.99979235363)
```

```
n = 80
s1 = take(sequence(), n)
s2 = take(seq_accel(sequence()), n)

list_plot(s1) + \
list_plot(s2, rgbcolor = 'red') + \
line([(0, 2),(80, 2)], linestyle = '--')
```



Beispiel 29a

Trapezregel

1. Schreiben Sie eine Funktion **trapez(f, a, b, n)** die das Integral

$$\int_a^b f(x)dx$$

näherungsweise mit Hilfe der Trapezregel mit Schrittweite $h = \frac{b-a}{n}$ berechnet.
Die Trapezregel besagt:

$$\int_a^b f(x)dx \approx h \cdot \sum_{i=1}^{n} f\left(a + h \cdot \frac{2i-1}{2}\right).$$

2. Die Fehlerabschätzung für diese Approximation lautet:

$$|E(f)| \leq \frac{b-a}{24} h^2 \max_{a \leq x \leq b} |f''(x)|$$

Erweitern Sie ihre Funktion **trapez**, so dass sie sowohl eine Näherung für das Integral, als auch die Fehlerabschätzung berechnet.

Überprüfen Sie ihre Funktion anhand einiger Beispiele, und vergleichen Sie das Ergebnis mit der eingebauten Funktion **integral_numerical**.

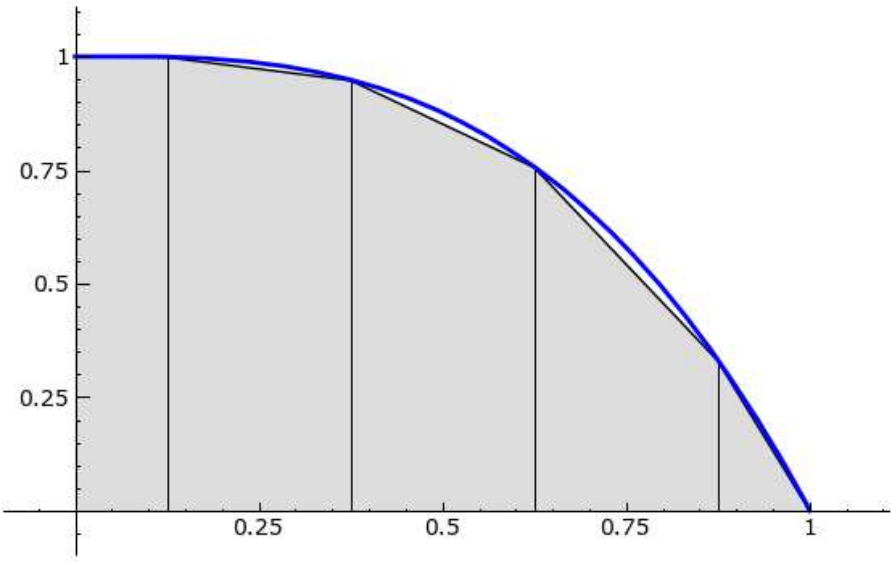
```
def trapez(f, a, b, n):
   h = RDF((b - a) / n)
   a = RDF(a)
   b = RDF(b)

   integral_approx =  h * sum([f(a + h * (2*i - 1)/2) for i in
[1..n]])

   return integral_approx
```

```
var('x')
trapez(x^2, 0, 1, 100)
```

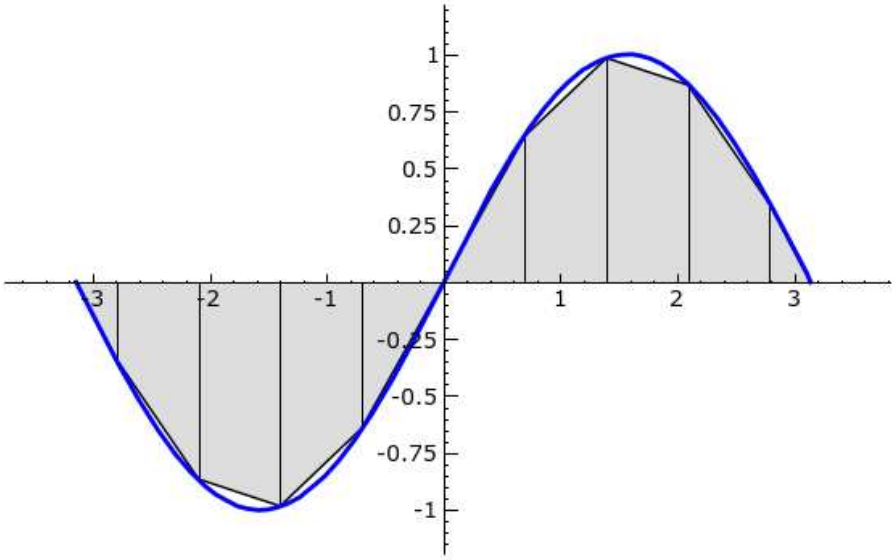
```
0.333325
```

```
numerical_integral(x^2, 0, 1)
```

```
(0.33333333333333331, 3.7007434154171879e-15)
```

```
trapez(1 - x^3, 0, 1, 100)
```
```
    0.7500125
```
```
numerical_integral(1 - x^3, 0, 1)
```
```
    (0.75000000000000011, 8.3266726846886756e-15)
```
```
trapez(e^x, 0, 1, 100)
```
```
    0.0542101327528
```
```
numerical_integral(e^x, 0, 1)
```
```
    (0.054286809695038393, 6.0270466056875793e-16)
```
```
def plot_trapez(f, a, b, n):
   h = RDF((b - a) / n)
   points = [(a, 0), (a, RDF(f(a)))] + \
            [(a + h * (2*i - 1)/2, RDF(f(a + h * (2*i - 1)/2))) for
i in [1..n]] + \
            [(b, RDF(f(b))), (b, 0)]
   p = polygon(points, rgbcolor = '#ddd')
   for point in points:
       p += line2d([(point[0], 0), point], rgbcolor = 'black')
   for i in xrange(len(points) - 1):
       p += line2d([points[i], points[i + 1]], rgbcolor = 'black')
   p += plot(f, a, b, thickness = 2)
   return p
```
```
plot_trapez(1 - x^3, 0, 1, 4).show(xmin = 0, ymin = 0)
```


```
plot_trapez(sin(x), -pi, pi, 9)
```




```
def trapez(f, a, b, n):
   h = RDF((b - a) / n)
   a = RDF(a)
   b = RDF(b)

   integral_approx =  h * sum([f(a + h * (2*i - 1)/2) for i in
[1..n]])

   f2(x) = f.diff().diff()

   maximum, estimate = find_maximum_on_interval(lambda x:
abs(f2(x)), a, b)

   error_approx = RDF((b - a) / 24 * h^2 * maximum)
   return (integral_approx, error_approx)
```

```
var('x')
time trapez(x^2, 0, 1, 10)
```

```
    (0.3325, 0.000833333333333)
    Time: CPU 2.02 s, Wall: 4.14 s
```

```
time trapez(1 - x^3, 0, 1, 10)
```

```
    (0.75125, 0.00249999992825)
    CPU time: 1.92 s,  Wall time: 4.36 s
```

```
numerical_integral(1 - x^3, 0, 1)
```

```
    (0.75000000000000011, 8.3266726846886756e-15)
```

```
time trapez(e^x, 0, 1, 20)
```

```
    (0.0524144255673, 0.035345983166)
    CPU time: 1.73 s,  Wall time: 4.72 s
```

```
numerical_integral(e^x, 0, 1)
```

```
    (0.054286809695038393, 6.0270466056875793e-16)
```

Die obige Implementierung von trapez ist sehr langsam. Man kann die Performance dramatisch verbessern, wenn man die Funktionen f und f'' in eine Form konvertiert, die für numerische Berechnungen optimiert ist, das geht mit der Funktion **fast_float**, aus dem Paket **sage.ext.fast_eval**.

```
def trapez_fast(f, a, b, n):
```

```
    h = RDF((b - a) / n)
    a = RDF(a)
    b = RDF(b)

    from sage.ext.fast_eval import fast_float, is_fast_float
    # Konvertiere Symbolische Ausdruecke in eine Form
    # die effizienter fÃƒÆ'Ã‚Â¼r Fliesskommazahlen ist.
    if not is_fast_float(f):
        ff =  fast_float(f)
    else:
        ff = f

    integral_approx =  h * sum([ff(a + h * (2*i - 1)/2) for i in
[1..n]])

    try:
        f2 = f.diff().diff()
        if not is_fast_float(f2):
            f2 = fast_float(f2)

        maximum, estimate = find_maximum_on_interval(lambda x:
abs(f2(x)), a, b)

        error_approx = RDF((b - a) / 24 * h^2 * maximum)
        return (integral_approx, error_approx)

    except AttributeError:
        # Wenn f kein symbolischer Ausdruck ist, koennen wir
        # keine Fehlerschranke berechnen.
        return (integral_approx, "no error estimate available")
```

```
time trapez_fast(1 - x^3, 0, 1, 500)
```

```
    (0.7500005, 9.999999713e-07)
    CPU time: 0.06 s,  Wall time: 0.15 s
```

```
time trapez_fast(x^2, 0, 1, 500)
```

```
    (0.333333, 3.33333333333e-07)
    CPU time: 0.07 s,  Wall time: 0.26 s
```

```
time trapez_fast(sin(x), 0, pi, 500)
```

```
    (2.00000328987, 5.16771278005e-06)
    CPU time: 0.06 s,  Wall time: 0.16 s
```

```
def f(x):
    return sin(x)

time trapez_fast(f, 0, pi, 500)
```

```
    (2.00000328987, 'no error estimate available')
    Time: CPU 0.16 s, Wall: 0.16 s
```

Beispiel 30

Das Interpolations Polynom $p(x)$ zu den Stützen

$$(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$$

ist das eindeutige Polynom n-ten Grades, für das gilt:

$$p(x_i) = y_i.$$

Das heisst das Polynom $p(x)$ geht durch alle Punkte (x_i, y_i).

1. Berechnen Sie das Interpolations Polynom für die Stützwerte:

   $$(1, 5), (3, 2), (6, 7), (7, 2), (8, 2)$$

2. Schreiben Sie eine Funktion in Sage, die für n im Interval $[-1, 1]$ gleichmässig verteilte Werte x_i die Funktion

   $$f(x) = \frac{1}{1 + 25x^2}$$

   berechnet, und aus diesen Daten das zugehörige Interpolationspolynom bestimmt.

   Verleichen Sie die Interpolationspolynome für $n = 6, 10, 20$ graphisch mit der Funktion $f(x)$. Wird die Funktion gut approximiert?

3. Machen Sie das selbe wie in Punkt (b) nur wählen Sie diesmal n Stützstellen der Form

   $$x_i = \cos\left(\frac{2i-1}{2n}\pi\right).$$

   Wird die Funktion gut approximiert?

Hinweis: Verwenden Sie die Maxima Funktion **lagrange** aus dem Maxima Paket **interpol** zum berechnen des Interpolationspolynoms.

```
maxima.load('interpol')
```

```
"/local/data/huss/software/sage-3.3.alpha0/local/share/maxima/5.16.
/share/numeric/interpol.mac"
```

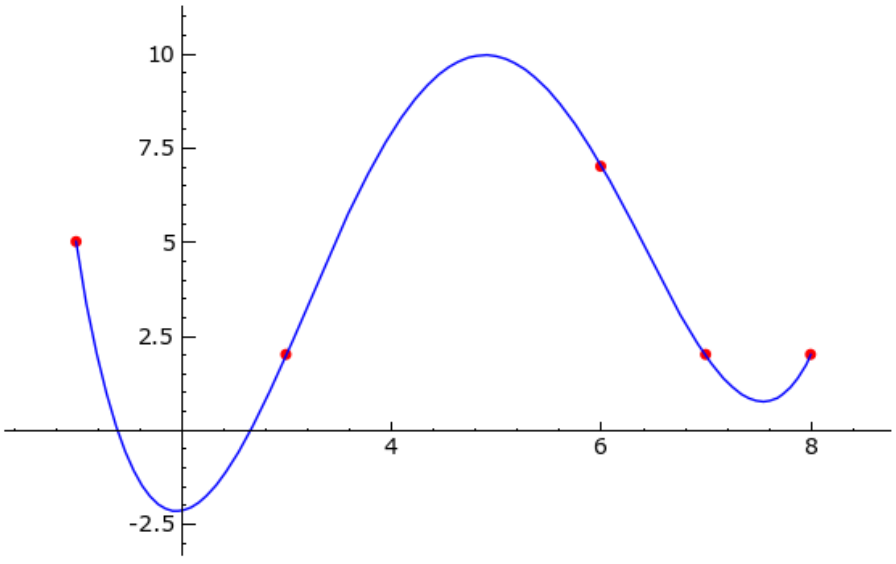
```
points = [(7, 2), (8, 2), (1, 5), (3, 2), (6, 7)]
f = maxima.lagrange(points).sage()
view(f)
```

$$\frac{73x^4}{420} - \frac{701x^3}{210} + \frac{8957x^2}{420} - \frac{5288x}{105} + \frac{186}{5}$$

```
reduce(lambda x, y: x and y, [bool(f(x[0]) == x[1]) for x in
points])
```

```
True
```

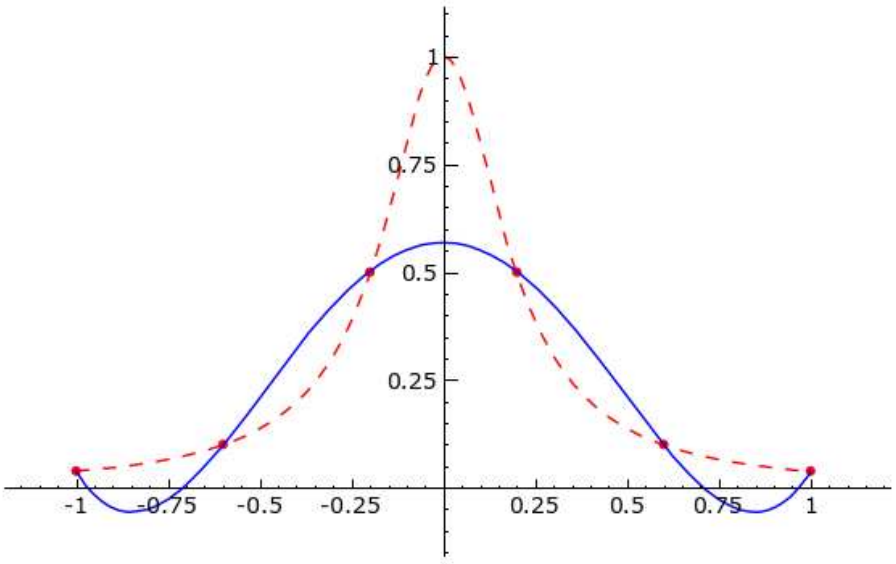
```
p1 = list_plot(points, pointsize = 20, rgbcolor = 'red')
p2 = plot(f, (x, 1, 8))
(p1 + p2).show()
```



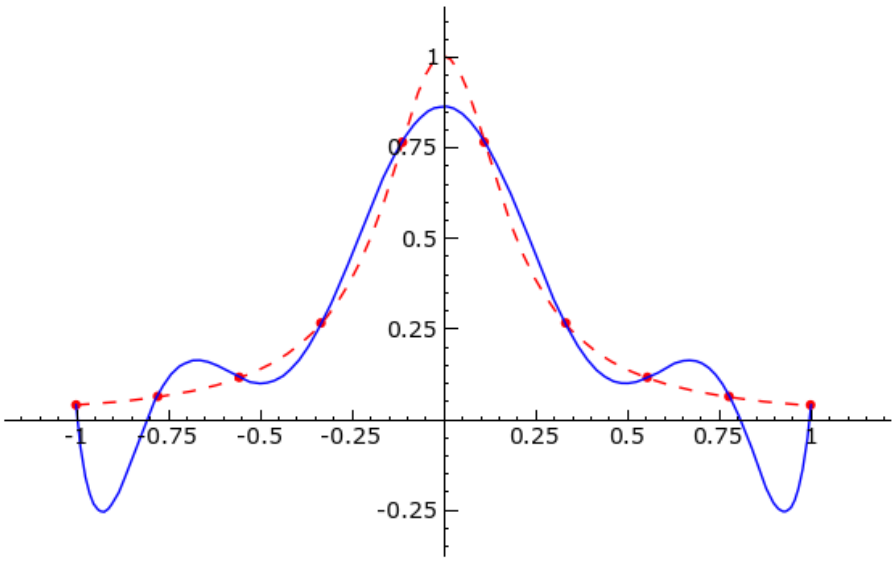

```
def f(x):
   return RDF(1/(1+25*x^2))
```

```
maxima.load('interpol')

def bsp30b(n):
   data = [(x, f(x)) for x in xsrange(-1, 1, 2/(n - 1),
include_endpoint = True)]
   polynom = maxima.lagrange(data).sage()
   g = plot(f, -1, 1, rgbcolor = 'red', linestyle='--')
   g += plot(polynom, -1, 1)
   g += list_plot(data, rgbcolor = 'red', pointsize = 15)
   return g
```

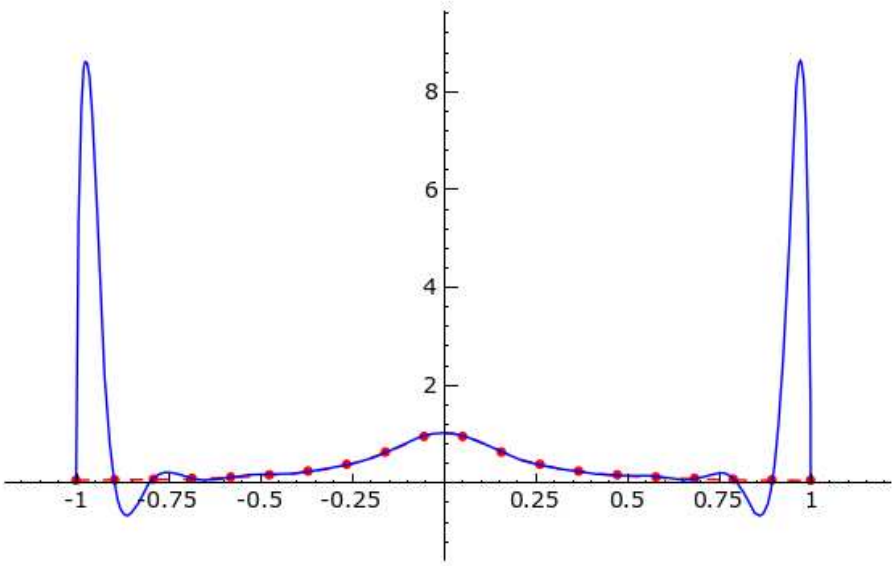
```
bsp30b(6)
```




```
bsp30b(10)
```

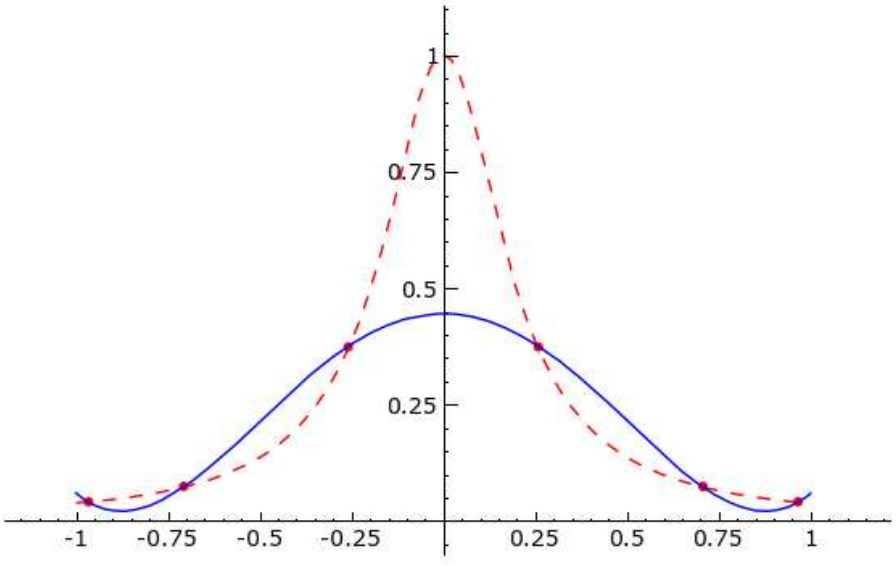



```
bsp30b(20)
```

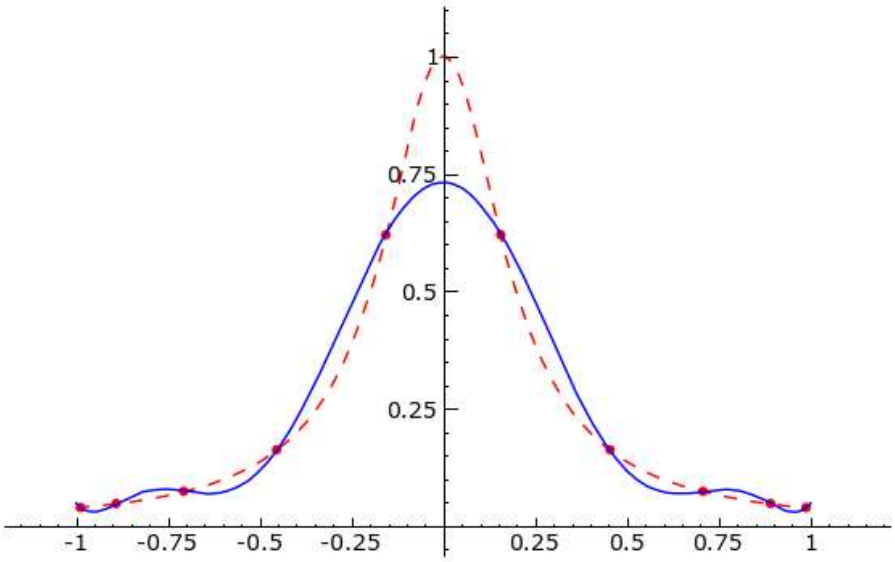



```
maxima.load('interpol')

def bsp30c(n):
   data = [(x, f(x)) for x in [RDF(cos((2*i-1)/(2*n)*pi)) for i in
xrange(1, n + 1)]]
   polynom = maxima.lagrange(data).sage()
   g = plot(f, -1, 1, rgbcolor = 'red', linestyle='--')
   g += plot(polynom, -1, 1)
   g += list_plot(data, rgbcolor = 'red', pointsize = 15)
   return g
```

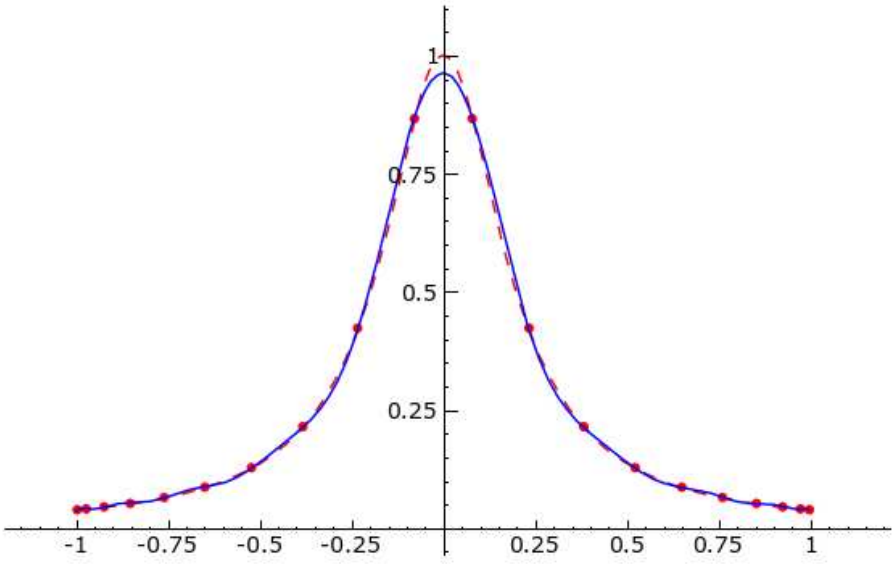
```
bsp30c(6)
```




```
bsp30c(10)
```




```
bsp30c(20)
```




```
maxima.load('interpol')

def f(x):
   return RDF(1/(1+25*x^2))

def interpolate_data(n, points = 'uniform'):
   if points == 'uniform':
       return [(x, f(x)) for x in xsrange(-1, 1, 2/(n - 1),
include_endpoint = True)]
   else:
       return [(x, f(x)) for x in [RDF(cos((2*i-1)/(2*n)*pi)) for i
in xrange(1, n + 1)]]

def interpolate_polynom(data):
   return maxima.lagrange(data).sage()

def interpolate_plot(n, points = 'uniform', color = 'red'):
   data = interpolate_data(n, points = points)
   p1 = plot(f, -1, 1, thickness = 2, linestyle = '--')
   p2 = list_plot(data, pointsize = 20, rgbcolor = color)
```

```
    p3 = plot(interpolate_polynom(data), -1, 1, rgbcolor = color)
    return p1 + p2 + p3

@interact
def bsp30_interact(n = slider(3,
vmax=30,step_size=1,default=3,label="interpolation order"), all =
False, points = ['uniform', 'chebyshev']):
    if all:
        p = Graphics()
        for i in xrange(3, n + 1, 2):
            p += interpolate_plot(i, points = points, color =
hue(i/n))
        p
    else:
        p = interpolate_plot(n, points = points)

    p.show(ymin = 0, ymax = 1)
```

interpolation order

all ☐

points uniform chebyshev