Übung 01: Riemannsche Zeta Funktion

Tobias Blesgen und Leonardo Thome

05.05.2021

Riemannsche Zeta Funktion

Definition der Riemmannschen Zeta Funktion

Die allgemeine Definition der Reimannschen Zeta Funktion ist:

$$\zeta(s) = \sum_{k=1}^{\infty} \frac{1}{k^s}, s \in \mathbb{C}\backslash\{0\} \tag{1}$$

Im Folgenden wollen wir uns genauer mit der Riemannschen Zeta Funktion von 2 beschäftigen, also $\zeta(2)$.

$$\zeta(2) = \sum_{k=1}^{\infty} \frac{1}{k^2} \tag{2}$$

Konvergenzverhalten der Riemannschen Zeta Funktion

Für die Konvergenz bzw. das Konvergenzverhalten der Riemannschen Zeta Funktion betrachten wir den Wert von: $\limsup_{k\to\infty}(|\frac{s_{k+1}-s}{s_k-s}|)$ mit den Reihengliedern $s_k = \frac{1}{k^2}$, bzw $s_{k+1} = \frac{1}{(k+1)^2}$ und dem bekannten Grenzwert $s = \frac{\pi^2}{6}$.

$$\begin{aligned}
\limsup_{k\to\infty}(|\frac{s_{k+1}-s}{s_k-s}|) &= \limsup_{k\to\infty}(|\frac{\frac{1}{(k+1)^2}-s}{\frac{1}{k^2}-s}|) \\
&= \limsup_{k\to\infty}(|\frac{\frac{1-(k+1)^2 s}{(k+1)^2}}{\frac{1-k^2 s}{k^2}}|) \\
&= \limsup_{k\to\infty}(|\frac{k^2-k^2(k+1)^2 s}{(k+1)^2-k^2(k+1)^2 s}|) \quad \text{, zwei mal l'Hospital} \\
&= \limsup_{k\to\infty}(|\frac{2-12k^2 s-12ks-2s}{2-12k^2 s-12ks-2s}|) \\
&= \limsup_{k\to\infty}(|1|) \\
&= 1
\end{aligned}$$

Da nun $\limsup_{k\to\infty}(|\frac{s_{k+1}-s}{s_k-s}|) = 1$ ist, handelt es sich um ein unterlineares Konvergenzverhalten, somit konvergiert die Reihe langsamer als linear. Da sich analog zur obrigen Umformung ergibt, dass die Riemannschen Zeta Funktion ebenfalls die Gleichung $\lim_{k\to\infty}(|\frac{s_{k+2}-s_{k+1}}{s_{k+1}-s_k}|) = 1$ erfüllt, konvergiert die Reihe sogar logarithmisch. [1]

[1] Beide Bedingungen stammen von dem Wikipediaartikel "Konvergenzgeschwindigkeit"

Numerische Näherungsverfahren

Die einfachste Implementierung einer Näherung erfolgt durch das direkte Implementieren der Reihe über ihre Summe. Diese ist in der Implementation des naiven Verfahrens zu betrachten. Dabei stößt man jedoch schnell auf zwei große Probleme:

1. Die Konvergenz der Reihe in ihrer grundlegenden Form ist nicht sehr schnell und braucht daher viele Summenschritte, bis ein ausreichend genauer Wert erreicht wird.
2. Das Addieren immer kleiner werdenden Zahlen ist für den Menschen kein Problem, jedoch hat der Computer nur eine begrenzte Anzahl an Stellen für eine Zahl. So sind die Zahlen irgendwann zu klein, um auf die größere vorherige Zahl addiert zu werden. Dies ist in der Regel kein Problem, da zu dem Zeitpunkt die gewünschte Genauigkeit erreicht wurde. Jedoch tritt das Problem der langsamen Konvergenz auf, durch das der Wert zu jenem Zeitpunkt noch zu ungenau ist.

Durch diese Probleme ist die Implementation des naiven Verfahrens für unsere Zwecke zu langsam und zu ungenau.

Um diese Probleme zu vermeiden, kann ein Verfahren mittels alternierender Reihe nach Borwein genutzt werden. Durch das Alternieren der Reihe eine schnellere Konvergenz gegeben ist.

In alternierender Form[2] lässt sich die Reihe wie folgt schreiben:

$$\zeta(s) = \frac{1}{1 - 2^{1-s}} \sum_{k=1}^{\infty} \frac{-1^{k-1}}{k^s} \tag{3}$$

bzw. für s = 2

$$\zeta(2) = 2 \sum_{k=1}^{\infty} \frac{(-1)^{k-1}}{k^2} \tag{4}$$

Diese Form der Reihe und die einhergehende Berechnung sind unter Implementation der Reihe nach Borwein zu finden.

Da wir eine Genauigkeit von 15 Dezimalstellen haben wollen, hat die Reihe nach Borwein zwar eine vertretbare, aber denoch recht lange Laufzeit. Zum Verkürzen der Laufzeit wenden wir das Eulersche Verfahren zur Mittelwertbildung auf unsere alternierende Reihe an. Dabei werden die einzelnen Reihenglieder durch den Mittelwert des Reihenglieds selber und des vorherigen Reihenglieds ersetzt. Zu beachten ist, dass wir zuvor das erste Reihenglied bestimmen und aus der Summe ziehen müssen, um nicht durch 0 zu teilen.

$$S'_k = \frac{1}{2} * (S_k + S_{k-1}) \tag{5}$$

Für unseren Fall:

$$S'_k = (-1)^{k-1} \frac{1}{2} (\frac{1}{k^2} - \frac{1}{(k-1)^2}) \tag{6}$$

Und somit unsere neue Summe:

$$\zeta(2) = 2 * (0.5 + \sum_{k=2}^{\infty} (-1)^{k-1} (\frac{1}{k^2} - \frac{1}{(k-1)^2}) \tag{7}$$

Durch die Implementierung der Reihe nach Borwein mit dem Eulerschen Verfahren zur Mittelwertbildung können wir die Reihe nun neu bestimmen. Dabei fällt auf, dass zwar die Laufzeit wesentlich kürzer ist,

[2]Quelle: P. Borwein: An efficient algorithm for the Riemann zeta function. In Théra, Michel A. (ed.). Constructive, Experimental, and Nonlinear Analysis (PDF). Conference Proceedings, Canadian Mathematical Society. 27. Providence, RI: American Mathematical Society, on behalf of the Canadian Mathematical Society. pp. 29–34. ISBN 978-0-8218-2167-1.

jedoch wir ein nur noch auf 13 Dezimalstellen genaues Ergebnis erhalten. Da dies der Problemstellung nicht genügt, ist die Reihe nach Borwein die beste Lösung, die wir gefunden haben. Bei einer Problemstellung, bei der die Laufzeit relevant wird und eine Genauigkeit von 13 Dezimalstellen genügt, würde sich die Reihe nach Borwein mit Mittelwertbildung nach Euler anbieten.

Implementation des naiven Verfahrens:

Das naive Verfahren konvergiert nur sehr langsam. Da wir es hier nur nutzen, um später die ersten 100 Schritte grafisch mit den anderen Verfahren vergleichen zu können, laufen wir an dieser Stelle auch nur über diese ersten 100 Summenglieder, um unnötige Wartezeiten zu vermeiden.

```
#include <Rcpp.h>

using namespace Rcpp;

//[[Rcpp::export]]
Rcpp::List naiveZetafunktion(){
  // Array der ersten 100 Werte:
    Rcpp::NumericVector x(100);
    Rcpp::NumericVector y(100);
  // Quelltext
    long double summe = 0;
    for (int i = 1; i < 100; i++)
    {
        summe += 1.0/(i*i);

        // Einsammeln der ersten 100 Werte
          x[i] = i;
          y[i] = summe;

    }
    // Rückgabe für eine grafische Wiedergabe
    return List::create(Named("x") = x, Named("y") = y);
}
```

Implementation der Reihe nach Borwein

```
#include <Rcpp.h>
#include <math.h>

using namespace Rcpp;

//[[Rcpp::export]]
Rcpp::List borwein(int s){
  // Array der ersten 100 Werte:
    Rcpp::NumericVector x(100);
    Rcpp::NumericVector y(100);
  // Quelltext
    long double summe = 0;
    long double summe2 = 1;           // Summe mit nächstem Summenglied
```

```
    long double vorzeichen = 1;
    int k = 1;
    while(fabs(summe2-summe)>pow(10,-16)){
        summe += vorzeichen *pow(k,-s);
        vorzeichen *= -1;
        summe2 = vorzeichen *pow(k+1,-s) + summe;
        // Einsammeln der ersten 100 Werte
        if(k<=100){
          x[k-1] = k;
          y[k-1] = summe2/ (1-pow(2,1-s));
        }
        k++;
    }
    summe /= (1-pow(2,1-s));          // Korrektur um den Vorfaktor
    Rprintf("Das Verfahren nach Borwein: %.16Lf \n",summe);
    // Rückgabe für eine grafische Wiedergabe
    return List::create(Named("x") = x, Named("y") = y);
}
```

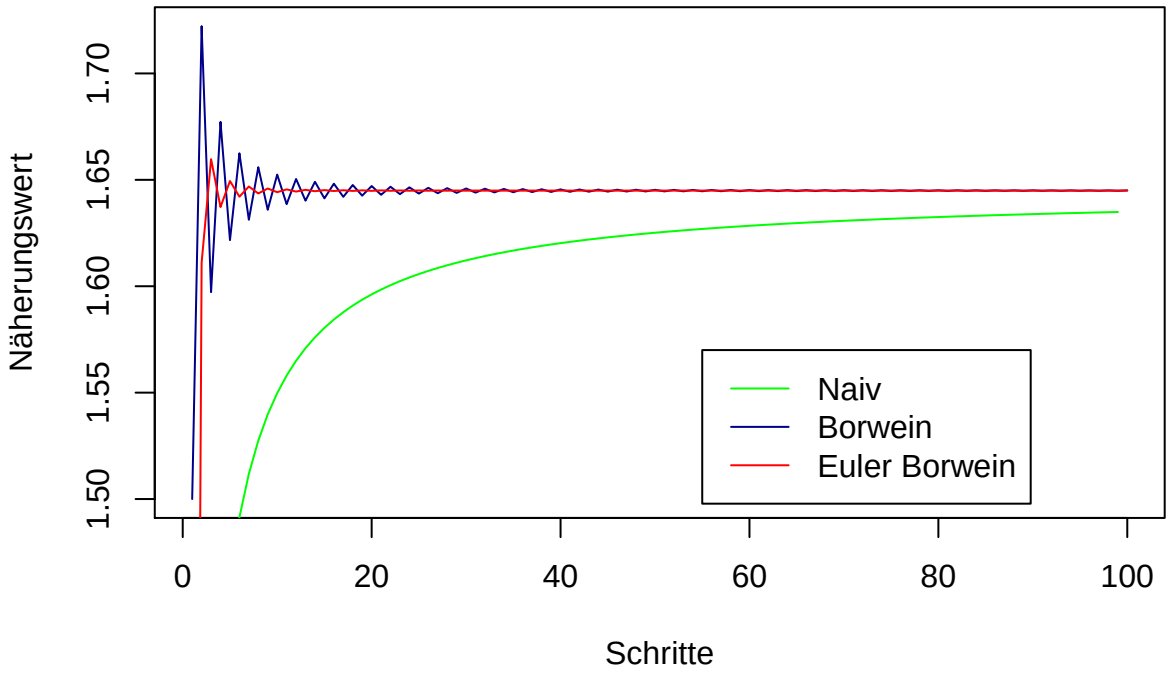
Implementation der Reihe nach Borwein mit Eulerschem Verfahren zur Mittelwertbildung

```
#include <Rcpp.h>
#include <math.h>

using namespace Rcpp;

//[[Rcpp::export]]
Rcpp::List BorweinEuler(int s){
  // Array der ersten 100 Werte:
    Rcpp::NumericVector x(100);
    Rcpp::NumericVector y(100);
  // Quelltext
    double summe = pow(1,-s)*0.5;   // Hälfte des ersten Glieds
    double summe2 = 2;              // Summe mit nächstem Summenglied
    double vorzeichen = -1;
    int k = 2;
    while(fabs(summe2-summe)>pow(10,-15)){
        summe += vorzeichen *0.5*(pow(k,-s)-pow(k-1,-s));
        vorzeichen *= -1;
        summe2 = vorzeichen* 0.5*(pow(k+1,-s)-pow(k,-s)) + summe;
        // Einsammeln der ersten 100 Werte
        if(k<=100){
          x[k-1] = k;
          y[k-1] = summe2/ (1-pow(2,1-s));
        }
        k++;

    }
    summe /= (1-pow(2,1-s));        // Korrektur um den Vorfaktor
```

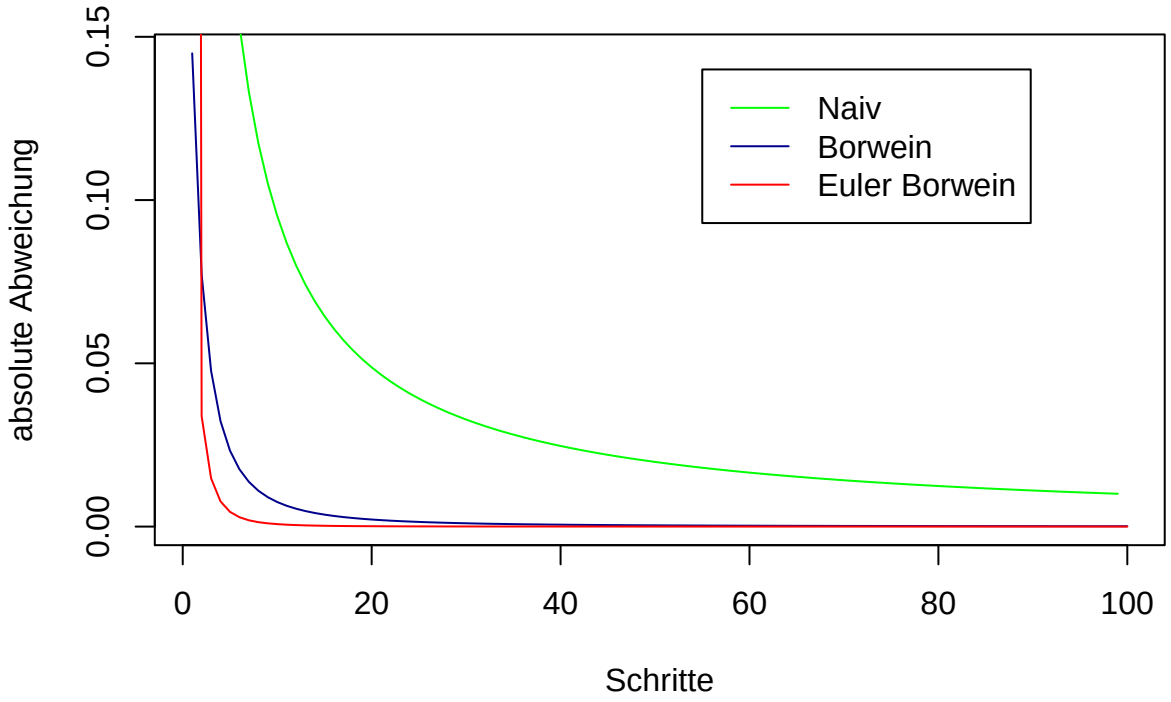
```
    Rprintf(
      "Das Verfahren mit der Eulerschen Mittelwertbildung ergibt \n %.16lf\n"
      , summe);
    // Rückgabe für eine grafische Wiedergabe
    return List::create(Named("x") = x, Named("y") = y);
    }
```

```
## Das Verfahren mit der Eulerschen Mittelwertbildung ergibt
##  1.6449340668482426
```

```
## Das Verfahren nach Borwein: 1.6449340668482266
```



Um nun das Konvergenzverhalten besser betrachten zu können, können wir uns direkt die Differenz zum Endwert grafisch darstellen lassen:



Es fällt auf, dass das naive Summenverfahren deutlich länger als die beiden Implementationen nach Borwein braucht, um gute Ergebnisse zu liefern. Dieses Beispiel verdeutlicht die große Relevanz des korrekten Implementationsverfahrens.