Übung 01: Riemannsche Zeta Funktion

Tobias Blesgen und Leonardo Thome

28.04.2021

Riemannsche Zeta Funktion

Definition der Riemmannschen Zeta Funktion

Die allgemeine Definition der Reimannschen Zeta Funktion ist:

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

Im folgenden wollen wir uns genauer mit 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 betracheten 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 bekanntem Grenzwert $s = \frac{\pi^2}{6}$ [1].

$$\limsup_{k\to\infty}(|\frac{s_{k+1}-s}{s_k-s}|) = \limsup_{k\to\infty}(|\frac{\frac{1}{k^2}-s}{\frac{1}{(k+1)^2}-s}|) \tag{3}$$

$$= \limsup_{k\to\infty}(|\frac{\frac{1-(k+1)^2 s}{(k+1)^2}}{\frac{1-k^2 s}{(k^2}}|) \tag{4}$$

$$= \limsup_{k\to\infty}(|\frac{k^2-k^2(k+1)^2 s}{(k+1)^2-k^2(k+1)^2 s}|), l'Hospital \tag{5}$$

$$= \limsup_{k\to\infty}(|\frac{2-12k^2 s-12ks-2s}{2-12k^2 s-12ks-2s}|) \tag{6}$$

$$= \limsup_{k\to\infty}(|1|) = 1 \tag{7}$$

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.

[1] J.-P. Serre: A course in arithmetic, Springer-Verlag New York (1973), ISBN 978-0-387-90040-7, S. 91.

Numerische Näherungsverfahren

Die einfachtes Implementierung einer Näherung erfolgt durch das direkte Implementieren der Reihe über ihre Summe. Diese ist in der Impelmentation des naives Verfahren 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 außreichend genauer Wert erreicht wird.
2. Das Addieren immer kleiner Zahlen ist für den Menschen kein Problem jedoch hat der Computer nur eine begrenzte Anhalen 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 hier das Problem der langsamen Konvergenz auf durch das der Wert zu jenem Zeitpunk noch zu ungenau ist.

Durch diese Probleme ist die Impelmentation des naives Verfahren für unsere Zwecke zu langsam und zu ungenau.

Um diese Probleme zu vermeiden kann ein Verfahren mittels alternierder Reihe nach Borwein genutzt werden. Da durch das alternieren der Reihe ist eine schnellere Konvergenz gegeben ist.

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

$$\zeta(s) = \frac{1}{1-2^{1-s}} \sum_{k=1}^{\infty} \frac{-1^{k-1}}{k^s} \tag{8}$$

bzw. für s = 2

$$\zeta(s) = 2 \sum_{k=1}^{\infty} \frac{(-1)^{k-1}}{k^2} \tag{9}$$

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

Da wir eine Genauigkeit von 15 Dezimalstellen haben wollen hat die Reihe nach Borwein zwar eine vertrettbare, 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 dem vorherigen Reihenglieds ersetzt, zu beachten ist das wir zufor das erste Reihenglied bestimmen und aus der Summe ziehen müssen um nicht durch 0 zu teilen.

$$S'_k = \frac{1}{2} * (S_k + S_{n+1}) \tag{10}$$

Für unseren Fall:

$$S'_k = (-1)^{k-1} \frac{1}{2} (\frac{1}{k^2} + \frac{1}{(k-1)^2}) \tag{11}$$

Und somit unsere neue Summe:

$$\zeta(s) = 2 * (0.5 + \sum_{k=2}^{\infty} \frac{1}{k^2} - \frac{1}{(k-1)^2}) \tag{12}$$

Durch die Impementierung in Implementation der Reihe nach Borwein mit Eulerschem Verfahren zur Mittelwertbildung können wir die Reihe nun neu bestimmen. Dabei fällt auf das zwar die laufzeit wesendlich

[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.

kürzer ist, jedoch wir ein nur noch auf 13 Dezimalstellen genaues Ergebnis erhalten, da dies der Problemstellung nicht genügt ist die Reihe nach Borwein die best gefundenste Lösung. Bei einer Probelmstellung wo die lautzeit relavant wird und eine Genauigkeit von 13 Dezimalstellen genügen, würde sich die Reihe nach Borwein mit Mittelwertbildung nach Euler anbieten.

Impelmentation des naives Verfahren:

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 sum = 0;
    for (int i = 1; i < 100; i++)
    {
        sum += 1.0/(i*i);

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

    }
    // 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 summenext = 1;
```

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

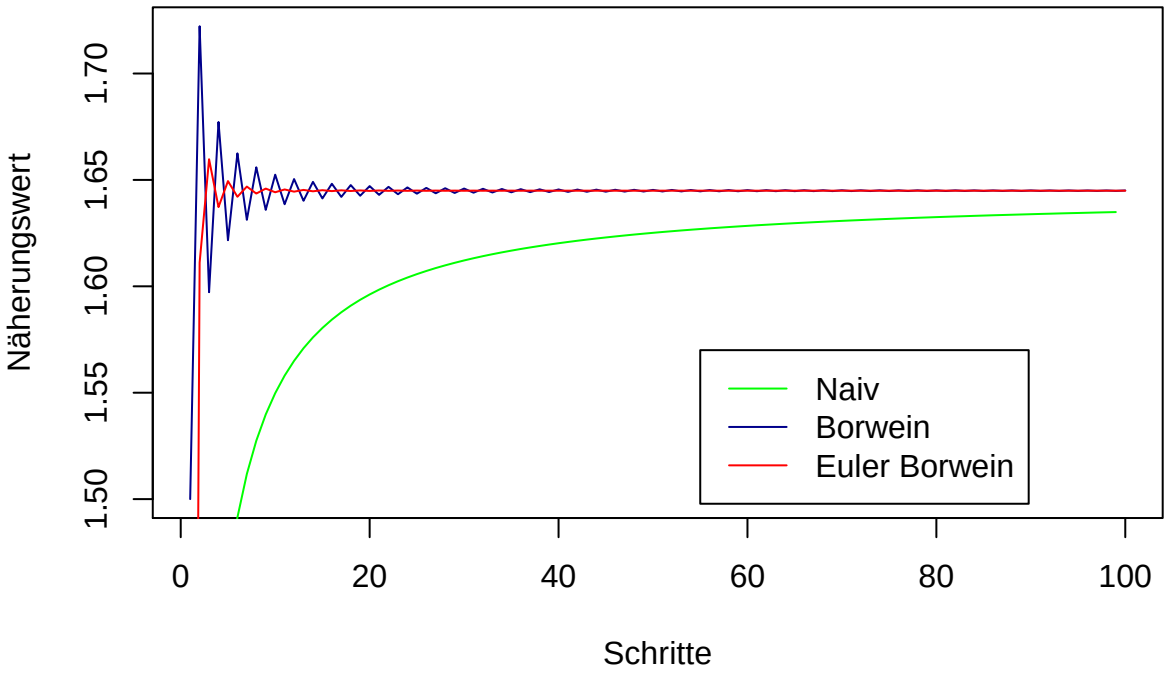
Implementation der Reihe nach Borwein mit Eulerschem Verfahren zur Mittelwertbildung

```
#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 summenext = 2;
    double vorzeichen = -1;
    int k = 2;
    while(fabs(summenext-summe)>pow(10,-15)){
        summe += vorzeichen *0.5*(pow(k,-s)-pow(k-1,-s));
        vorzeichen *= -1;
        summenext = 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] = summenext/ (1-pow(2,1-s));
        }
        k++;

    }
    summe /= (1-pow(2,1-s));
```

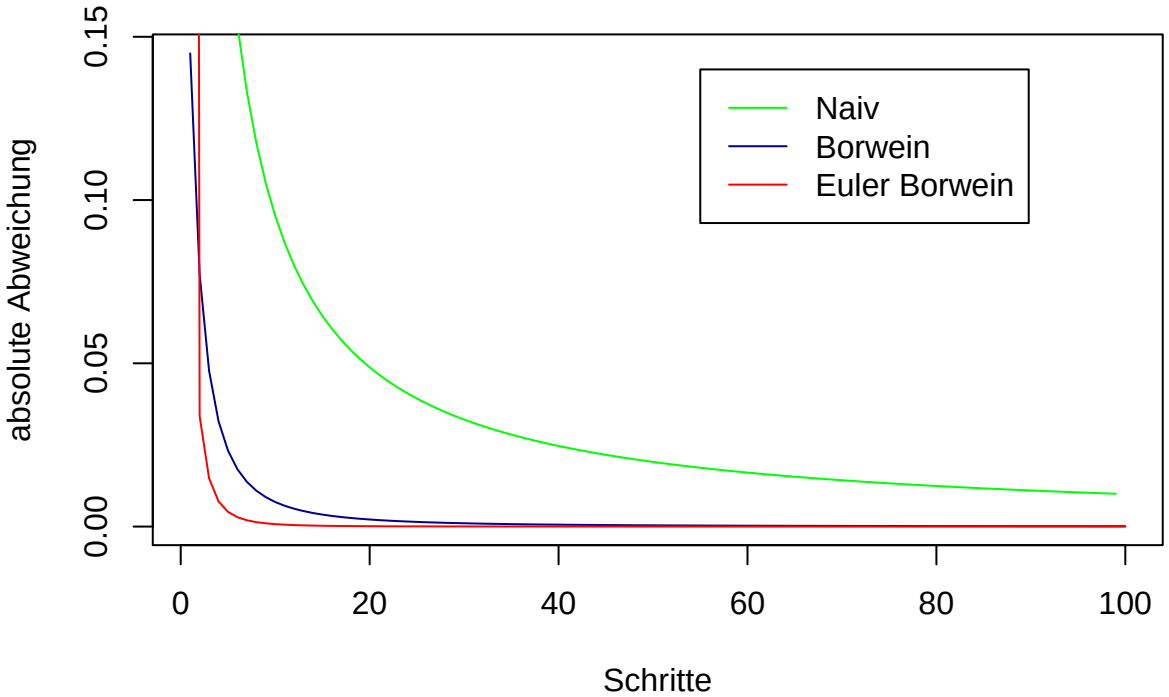
```
    Rprintf("Das Verfahren nach Borwein mit Eulerschem Verfahren zur Mittelwertbildung ergibt %.16lf\n"
    // Rückgabe für eine grafische Wiedergabe
    return List::create(Named("x") = x, Named("y") = y);
    }
```

```
## Das Verfahren nach Borwein mit Eulerschem Verfahren zur 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

wird deutlich, dass der das naive Summenverfahren deutlich länger als die beiden Implementationen nach Borwein braucht um gute Ergebnisser zu liefern.