

Übung 01: Riemannsche Zeta Funktion

Tobias Blesgen und Leonardo Thome

4/14/2021

Riemannsche Zeta Funktion

Definition der Riemannschen Zeta Funktion

Die allgemeine Definition der Riemannschen Zeta Funktion ist:

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

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

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

Numerische Näherungsverfahren

Die einfache Implementierung einer Näherung erfolgt durch das direkte Implementieren der Reihe über ihre Summe. Diese ist in der Implementierung 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 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 hier das Problem der langsamen Konvergenz auf durch das der Wert zu jenem Zeitpunkt noch zu ungenau ist.

Durch diese Probleme ist die Implementierung 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. Da durch das Alternieren der Reihe eine schnellere Konvergenz gegeben ist.

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.

In alternierender Form lässt sich die Reihe wie folgt schreiben:

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

bzw. für $s = 2$

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

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

Impelmentation des naives Verfahren:

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

using namespace Rcpp;

//[[Rcpp::export]]
Rcpp::List zetafunktion(){
    // Array der ersten 100 Werte:
    Rcpp::NumericVector x(100);
    Rcpp::NumericVector y(100);
    // Quelltext
    int s = 2;
    long double sum = 0;
    for (long int i = 0; i < 10000000; i++)
    {
        sum += 1.0/pow(i,2);
        if(i<=100){
            x[i] = i;
            y[i] = sum/= (1-pow(2,1-s));
        }
    }
    Rprintf("Die Zahl ergibt sich als %.15Lf .\n", sum);
    return List::create(Named("x") = x, Named("y") = y);
}
```

Implementation der Reihe nach Borwein 2?

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

using namespace Rcpp;

//[[Rcpp::export]]
Rcpp::List borwein(int s, int N){
```

```

// Werte vector
Rcpp::NumericVector Werte(N*2+1);
// Algorithmus

Werte[0] = 0.0;

for (int k = 1; k < N; k+=2)
{
    Werte[k+1] = Werte[k]+pow(k,-s);

}
for (int k = 2; k < N; k+=2)
{
    Werte[N+k+1] = Werte[N+k] - pow(k,-s);
}

int skalierung = 1-pow(2,1-s);
for (int i = 0; i< 2*N+1; i++){
    Werte[i] = Werte[i]/skalierung;
}
Rprintf("Das Verfahren nach Borwein ergibt: %.16f \n",Werte[2*N]);
Rprintf("Fuck\n");
return List::create(Named("Werte") = Werte);
}

```

Implementation der Reihe nach Borwein

```

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

//[[Rcpp::export]]
long double borwein2(int s, int N){
    long double summe1 = 0;
    for (int k = 1; k < N; k+=2)
    {
        summe1 += pow(k,-s);
    }
    for (int k = 2; k < N; k+=2)
    {
        summe1 -= pow(k,-s);
    }
    summe1/= (1-pow(2,1-s));
    Rprintf("Das Verfahren nach Borwein scs: %.16Lf \n",summe1);
    return summe1;
}

```

Implementation der Reihe nach Borwein Test

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

using namespace Rcpp;

//[[Rcpp::export]]
Rcpp::List borweinTest(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;
    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 scs: %.16Lf \n",summe);
  return List::create(Named("x") = x, Named("y") = y);
}
```

Implementation der Reihe nach Borwein Euler

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

using namespace Rcpp;

//[[Rcpp::export]]
Rcpp::List borweinTestEuler(int s){
  // Array der ersten 100 Werte:
  Rcpp::NumericVector x(100);
  Rcpp::NumericVector y(100);
  // Quelltext
  double summe = pow(1,-s)*0.5;
  double summenext = 2;
  double vorzeichen = 1;
```

```

int k = 1;
while(fabs(summenext-summe)>pow(10,-16)){
    summe += vorzeichen *0.5*(pow(k,-s)-pow(k+1,-s));
    vorzeichen *= -1;
    summenext = vorzeichen* 0.5*(pow(k+1,-s)-pow(k+2,-s)) + summe;
    if(k<=100){
        x[k-1] = k;
        y[k-1] = summenext/= (1-pow(2,1-s));
    }
    k++;
}
summe /= (1-pow(2,1-s));
Rprintf("Schritte %d\n",k);
Rprintf("Das Verfahren ergibt %lf", summe);
return List::create(Named("x") = x, Named("y") = y);
}

```

```
euler = borweinTestEuler(2)
```

```

## Schritte 262144
## Das Verfahren ergibt 1.644934

```

```
borwein = borweinTest(2)
```

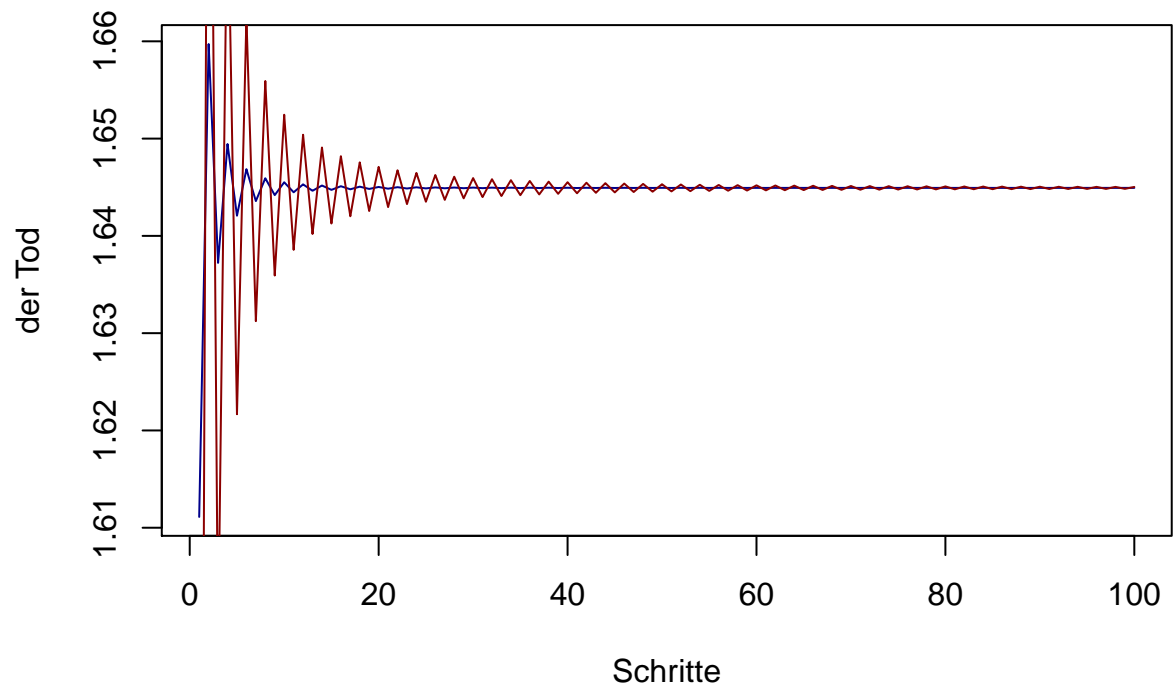
```
## Das Verfahren nach Borwein scs: 1.6449340668482266
```

```
# naiv = zetafunktion()
```

```

plot(euler$x, euler$y, "l", xlab = "Schritte", ylab = "der Tod", col = "dark blue")
lines(borwein$x, borwein$y, "l", col = "dark red")

```



```
# lines(naiv$x, naiv$y, "l", col = "green")
```