Übung 01: Riemannsche Zeta Funktion

Tobias Blesgen und Leonardo Thome

4/14/2021

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 $\zeta(2)$.

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

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.

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:

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

bzw. für s = 2

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

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

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
    long double sum = 0;
    for (long int i = 1; i < 1000; i++)
    {
        sum += 1.0/(i*i);
      if(i<=100){
          x[i] = i;
          y[i] = sum;
      }
    }
    Rprintf("Die Zahl ergibt sich als %.15Lf .\n", sum);
    return List::create(Named("x") = x, Named("y") = y);
}
```

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: %.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\n", summe);
    return List::create(Named("x") = x, Named("y") = y);
    }
```