Übung 05: Spektrum einer Klaviersaite

Tobias Blesgen und Leonardo Thome

27.06.2021

Schlägt man eine Klaviersaite an, so verhält sie sich nach:

$$\frac{\partial^2\psi(x,t)}{\partial t^2} = c^2\frac{\partial^2\psi(x,t)}{\partial x^2} - \frac{\gamma}{l}|\frac{\partial\psi(x,t)}{\partial t}|(\frac{\partial\psi(x,t)}{\partial t}). \tag{1}$$

Wir wollen dieses Verhalten im Folgenden numerisch untersuchen. Hierzu schreiben wir Gleichung 1 mit $\Xi = \frac{x}{l}, \phi = \frac{\psi}{l}, \tau = \frac{tc}{l}$ um zu einer Dimensionslosen Gleichung:

$$\frac{\partial^2\phi(\Xi,\tau)}{\partial\tau^2} = \frac{\partial^2\phi(\Xi,\tau)}{\partial\Xi^2} - \gamma|\frac{\partial\phi(\Xi,\tau)}{\partial\tau}|(\frac{\phi(\Xi,\tau)}{\partial\tau}). \tag{2}$$

Um dieses Problem analytisch angehen zu können, müssen wir die Differentiale annähern und schreiben hierzu die Gleichung 2 um zu:

$$\frac{\psi_{i+1,j} - 2\psi_{i,j} + \psi_{i-1,j}}{\Delta\tau^2} = \frac{\psi_{i,j+1} - 2\psi_{i,j} + \psi_{i,j-1}}{\Delta\Xi^2} - \gamma|\frac{\psi_{i,j} - \psi_{i-1,j}}{\Delta\tau}|\frac{\psi_{i,j} - \psi_{i-1,j}}{\Delta\tau} \tag{3}$$

Dies lässt sich weiter umstellen zu:

$$\psi_{i+1,j} = \frac{\Delta\tau^2}{\Delta\Xi^2}(\psi_{i,j+1} - 2\psi_{i,j} + \psi_{i,j-1}) + 2\psi_{i,j} - \psi_{i-1,j} - \gamma|\psi_{i,j} - \psi_{i-1,j}|(\psi_{i,j} - \psi_{i-1,j}) \tag{4}$$

Diese Gleichung wird nun verwendet um iterativ den nächsten Funktionswert an jedem Ort zu bestimmen. Als Randbedingung werden die Ränder, also $\Xi = 0$ und $\Xi = Xi_{max}$ auf 0 gesetzt, da sie im Klavier befestigt sind.

Implementation des DGS nach dem Runge-Kutta 2 Verfahren

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

using namespace Rcpp;


//[[Rcpp::export]]
NumericMatrix klaviersaite2(const double gamma, const int xSchritte,
                                          const double dt, const int
```

```
                                                zeitSchritte){
  // Array der Werte zur späteren Ausgabe
    NumericMatrix matrix(zeitSchritte, xSchritte);
  // Quelltext
    // Startwerte

    for (int i = 0; i<xSchritte; i++){
      matrix(0,i) = 0.0;
      matrix(1,i) = 0.0;
    }
    matrix(0,(int)(0.26*(xSchritte-1))) = 0.01;
    matrix(1,(int)(0.26*(xSchritte-1))) = 0.01;



    // Funktionsdurchläufe
    double dx = 1.0/(xSchritte - 1.0);
    Rprintf("o: %f\n", dx);
    double C = dt*dt/(dx*dx);

    for (int i = 2; i<zeitSchritte; i++){
      // Randbedingungen
      matrix(i,0) = 0.0;
      matrix(i,xSchritte-1) = 0.0;
      for (int j = 1; j<xSchritte-1; j++){
        matrix(i,j) = C*(matrix(i-1,j+1) - 2*matrix(i-1,j) +
          matrix(i-1,j-1)) + 2*matrix(i-1,j) - matrix(i-2,j) -
          gamma * fabs(matrix(i-1,j)-matrix(i-2,j)) *
          (matrix(i-1,j)-matrix(i-2,j));
      }
    }

  // Rückgabe für eine grafische Wiedergabe
    return matrix;
}
```

```
## o: 0.002004
```

```
## o: 0.001011
```

FFT

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

using namespace Rcpp;

double pi = 3.14159;
std::complex<double> I = 1i;
```

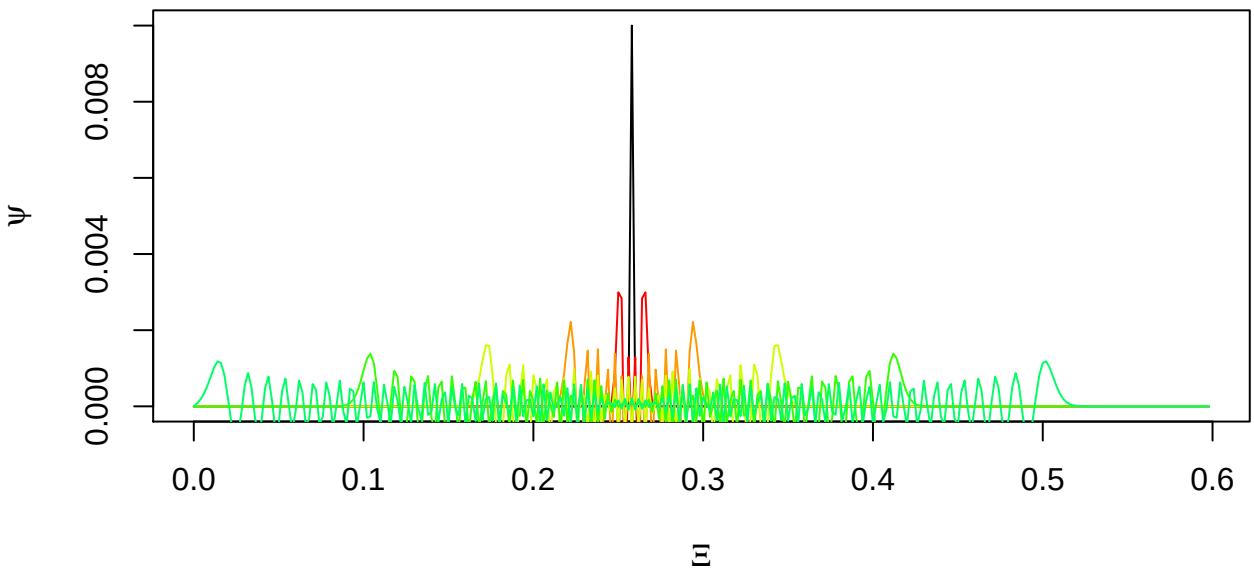


Abbildung 1: Zeitliche Entwicklung

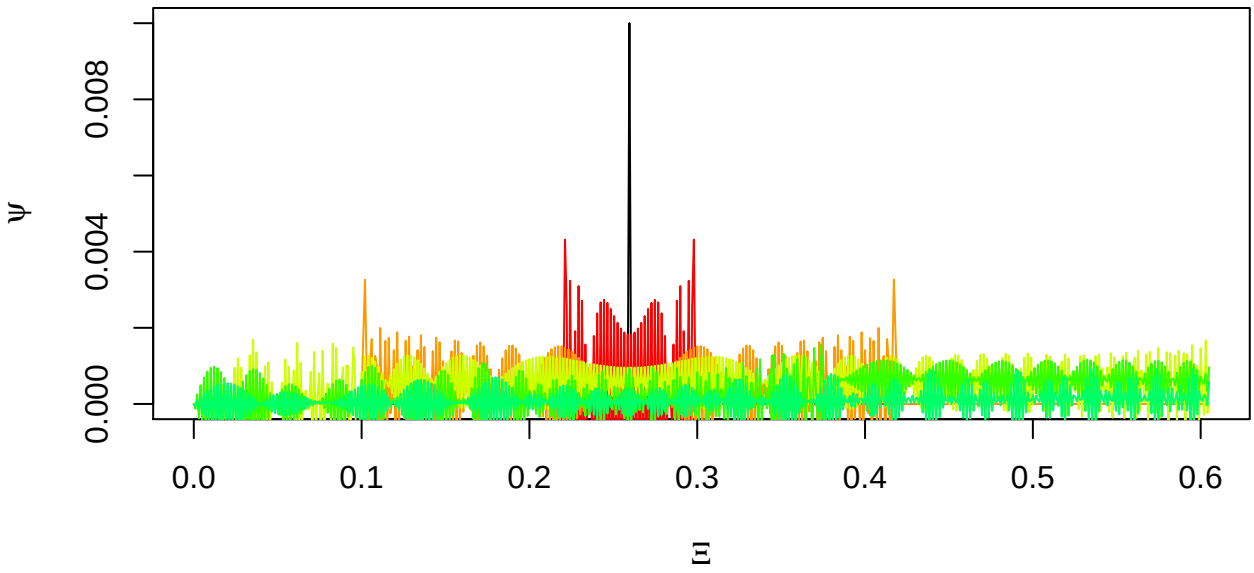


Abbildung 2: Zeitliche Entwicklung

```
// Generiere die w Faktoren
std::vector<std::complex<double>> wInit(int n){
  std::vector<std::complex<double>> W(n);
  for (int a = 0; a < n; a++)
    {
        W[a]= (-2.*pi*I*(double)a)/((double)n);
    }
  return W;
}

// Funktion zum Umsortieren der verdrehten Terme
void fftResort(const int n, int* k){
    int l = 0, m;
    for (int i = 0; i <= n-2; i++)
    {
        k[i] = l;
        m = n/2;
        while (m <= l)
        {
            l -= m;
            m /= 2;
        }
        l += m;
    }
    k[n-1] = n-1;
}

// Fast Fourier Transform - Funktion (nimmt die z_i und gibt die g_i unsortiert aus)
std::vector<std::complex<double>> fft (std::vector<std::complex<double>> z, const int r){
    const int n = pow(2,r);
    int m = n/2;
    int K = 1;
    std::vector<std::complex<double>> w = wInit(n);
    int a, b;

    for (int i = 0; i < r; i++)
    {
        for (int k = 0; k < K; k++)
        {
            for (int j = 0; j < m; j++)
            {
                a = 2*k*m +j;
                b = a + m;
                z[a] += z[b];
                z[b] = w[K*j]*(z[a] - 2.0*z[b]);
            }
        }
        m /= 2;
        K *= 2;
    }
    int index[n];
    fftResort(n, index);
    for (int i = 0; i < n; i++)
```

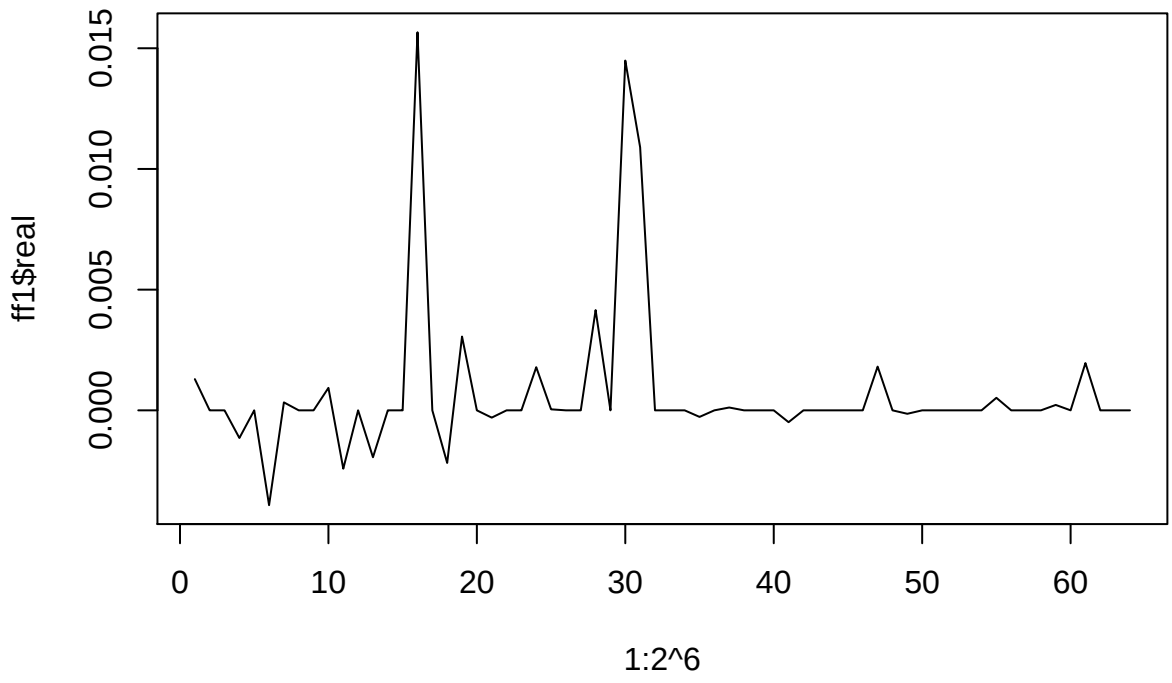
```
    {
        z[i]= z[index[i]] / sqrt(n);
    }
    Rprintf("%f\n", real(z[0]));
    return z;
}

// FFT Aufruf
//[[Rcpp::export]]
Rcpp::List fastFourier(int r,std::vector<std::complex<double>> a){
    int n = pow(2,r);
    a = fft(a,r);
    Rprintf("%f\n", real(a[0]));
    std::vector<std::complex<double>> re(n);
    std::vector<std::complex<double>> im(n);
    for (int i = 0; i < n; i++)
    {
        re[i] = real(a[i]);
        im[i] = imag(a[i]);
    }
    return List::create(Named("real") = re, Named("imag") = im);
}
```

```
ff1 = fastFourier(6, datenMatrix[,c(300:363)])
```

```
## 0.001293
## 0.001293
```

```
plot(1:2^6, ff1$real,"l")
```



Fazit

Wir erhalten bei den meisten Werten ein nicht chaotisches Verhalten zwischen θ und $\dot{\theta}$, können jedoch auch chaotische Situationen konstruieren. Dies ist im physikalischen Sinne auch nachvollziehbar, da die Eigenrotation des Mondes durch geschickt gewählte Bahnen stark von der Radiusvariation beeinflusst werden kann.

Wir können aber sehen, dass dieses Verhalten nicht immer eintritt und wir im Allgemeinen ein strukturiertes Verhalten vorfinden.

Literatur

[1] Keplersche Gesetze https://de.wikipedia.org/wiki/Keplersche_Gesetze, Stand 22.06.2021