

Reed-Solomon Codes

Digitale AV Technik, MIB 5

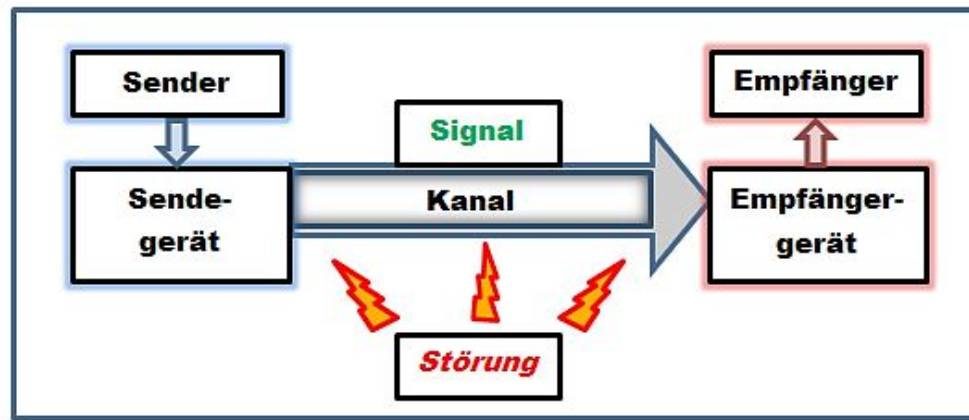
Aus Sicht der Informationstheorie

<i>Problem</i>	Kompression	Fehlerkorrektur
<i>Ziel</i>	Effizienz	Verlässlichkeit
<i>Anwendung</i>	Quellencodierung	Kanalcodierung

Algorithmische Perspektive

<i>Problem</i>	Fehlerkorrektur
<i>Algorithmen</i>	Hamming code
	Reed-Solomon Code
	Turbo-Code

Fehlererkennung



Störungen sind unvermeidlich. Hamming Codes ermöglichen die Korrektur eines einzelnen Bitfehlers. Aber geht auch mehr?

Einführung in Reed-Solomon Codes

"We're going to talk about going from Galois fields to Reed-Solomon codes. We must be mad. Really, I mean, so many of you have said: 'Just do Reed-Solomon, you know. You've done Hamming codes. They [Reed-Solomon] can't be that much more complex?'

Oh yes they can!"

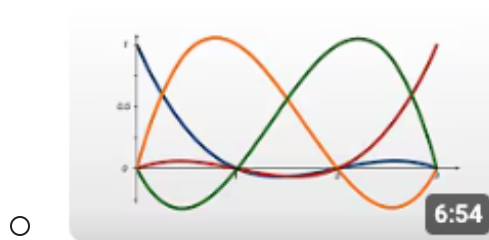
David Brailsford in [Reed Solomon Encoding - Computerphile](#)

Weitere Quellen

- Francesco Mazzoli: [The essence of Reed-Solomon coding](#)
- Die folgenden Folien enthalten Screenshots aus diesen YouTube Videos:



- [What are Reed-Solomon Codes? How computers recover lost data](#) by vcubingx



- [Lagrange Interpolation](#) by Dr. Will Wood

Geschichte der Reed-Solomon Codes

- Originalpaper: [Polynomial Codes Over Certain Finite Fields](#) by *I. S. Reed and G. Solomon*, 1960
- Effiziente Dekodierung [erst in den 1970ern](#) möglich
- Einsatz in der Praxis:
 - [NASA voyager](#) (launched 1977)
 - [CD](#) 1980
 - QR Codes

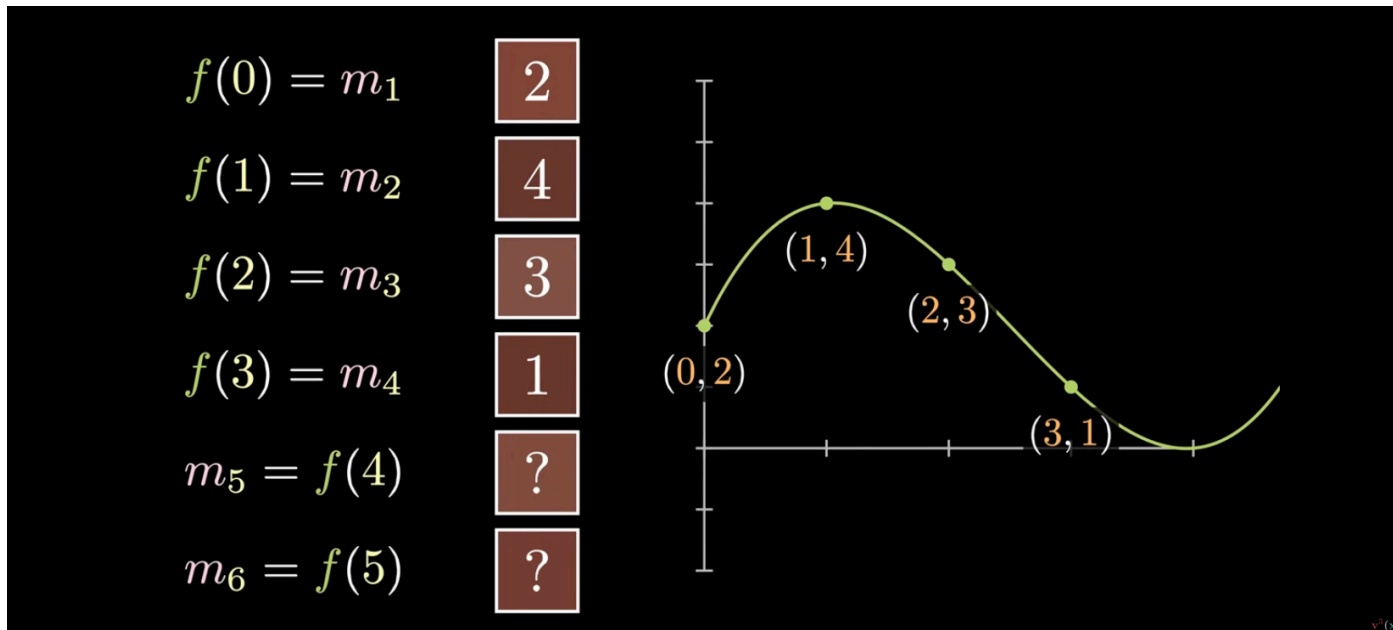


derivative work: Dzucconi (talk) [CD_autolev_crop.jpg](#): [Ubern00b](#), CC BY-SA 3.0, via Wikimedia Commons



Mathematische Grundlagen: Polynome

- Idee: Nachricht definiert ein Polynom
- Man sendet mehr Information als notwendig
- Wenn Daten fehlen, lässt sich das Polynom trotzdem rekonstruieren



Mathematische Grundlagen: Lagrange Polynome (1)

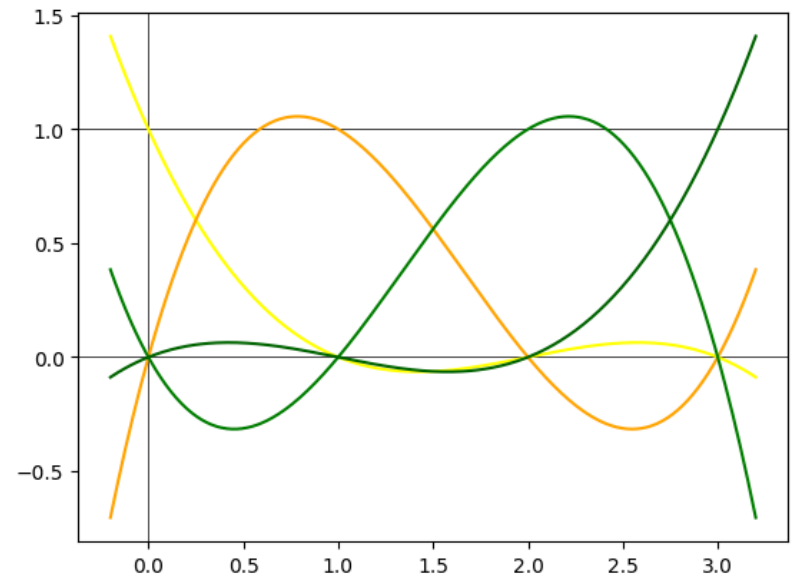
Ein Polynom als Kombination aus anderen (sog. Lagrange) Polynomen definieren.

Diese lassen sich einfach ermitteln und sind eindeutig: **Es gibt genau ein Polynom vom Grad $n - 1$ durch n Stützstellen.**

- Beispiel:
 - Stützstellen: $x_0 = 0, x_1 = 1, x_2 = 2, \dots$ usw.
 - Werte aus der Nachricht:
 $y_0 = 2, y_1 = 4, y_2 = 3, y_3 = 1$
- Lagrange Polynome haben für eine Stützstelle den Wert 1, für alle anderen den Wert 0. Dadurch lassen sie sich einfach ermitteln.

Mathematische Grundlagen: Lagrange Polynome (2)

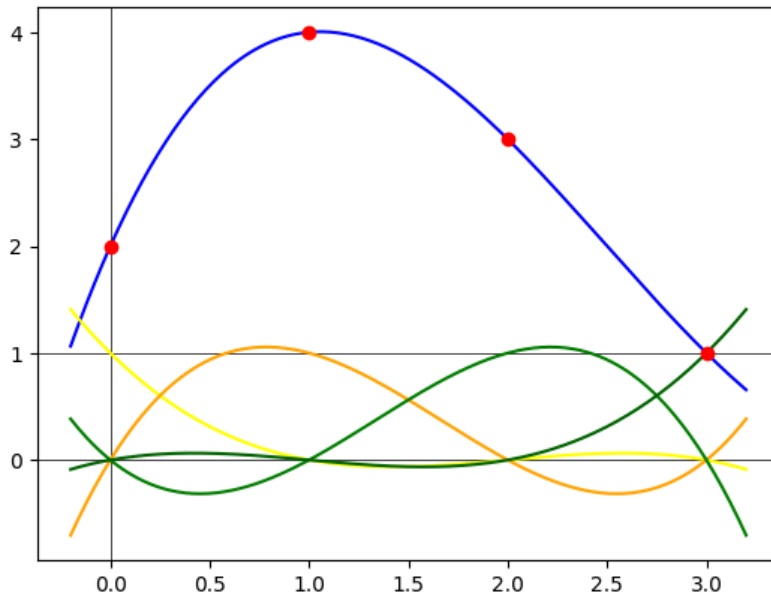
Wir arbeiten mit vier
Stützstellen, also hat das
Polynom max. Grad 3
Die Lagrange
Basispolynome l_0 bis l_3
sind fest definiert und
hängen nur vom Grad ab.



Mathematische Grundlagen: Lagrange Polynome (3)

Das Lagrange Polynom durch alle vier Stützstellen ergibt sich als gewichtete Kombination der Basispolynome. Die Gewichte sind die **y-Werte der Stützstellen**.

$$\begin{aligned} l(x) &= y_0 * l_0 + y_1 * l_1 + y_2 * l_2 + y_3 * l_3 \\ &= 2 * l_0 + 4 * l_1 + 3 * l_2 + 1 * l_3 \end{aligned}$$



Mathematische Grundlagen: Lagrange Polynome (4)

Allgemein lässt sich also das Polynom so berechnen:

$$L(x) = \sum_{i=0}^n y_i \cdot \ell_i(x),$$

wobei

$$\ell_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^n \frac{x - x_j}{x_i - x_j}$$

Mathematische Grundlagen: Lagrange Polynome (5)

Python Code (Teil 1): [lagrange.py](#)

```
def lagrange(x, x_i, y_i):  
    n = len(x_i)  
    m = len(x)  
    y = np.zeros(m)  
    for i in range(n):  
        p = lagrange_polynomial(i, x, x_i)  
        y += y_i[i] * p  
    return y
```

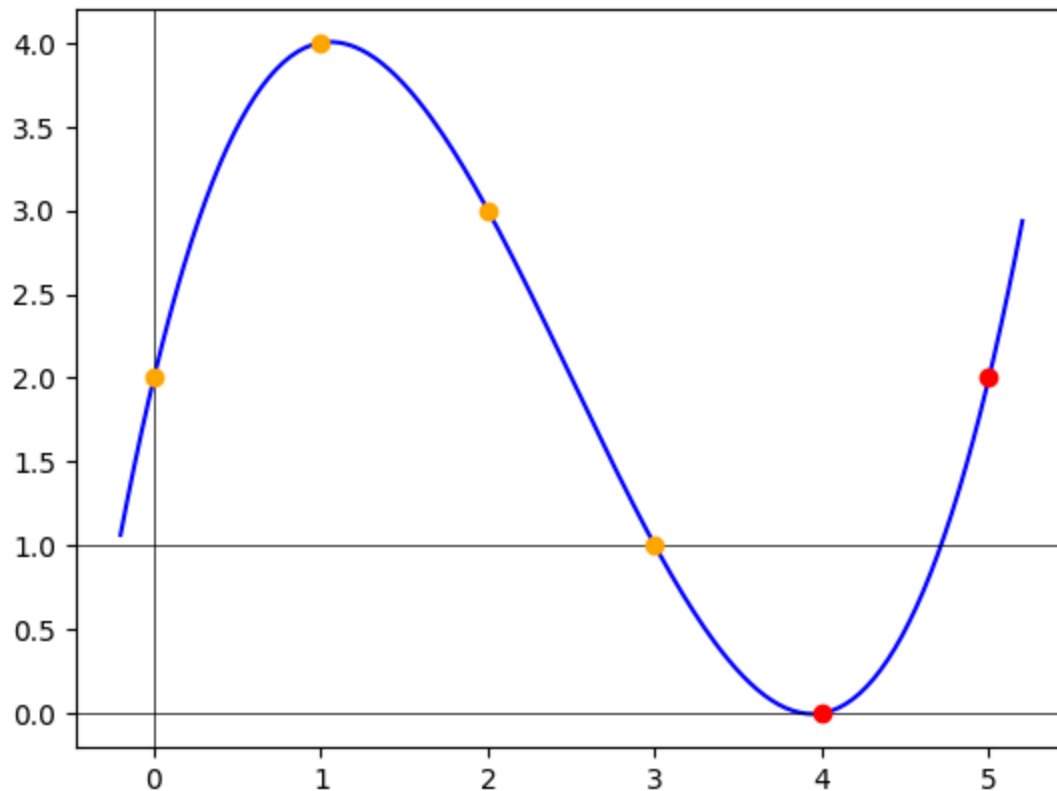
Mathematische Grundlagen: Lagrange Polynome (6)

Python Code (Teil 2): [lagrange.py](#)

```
def lagrange_polynomial(j, x, x_i):  
    n = len(x_i)  
    p = 1  
    for m in range(n):  
        if m != j:  
            p *= (x - x_i[m]) / (x_i[j] - x_i[m])  
    return p
```

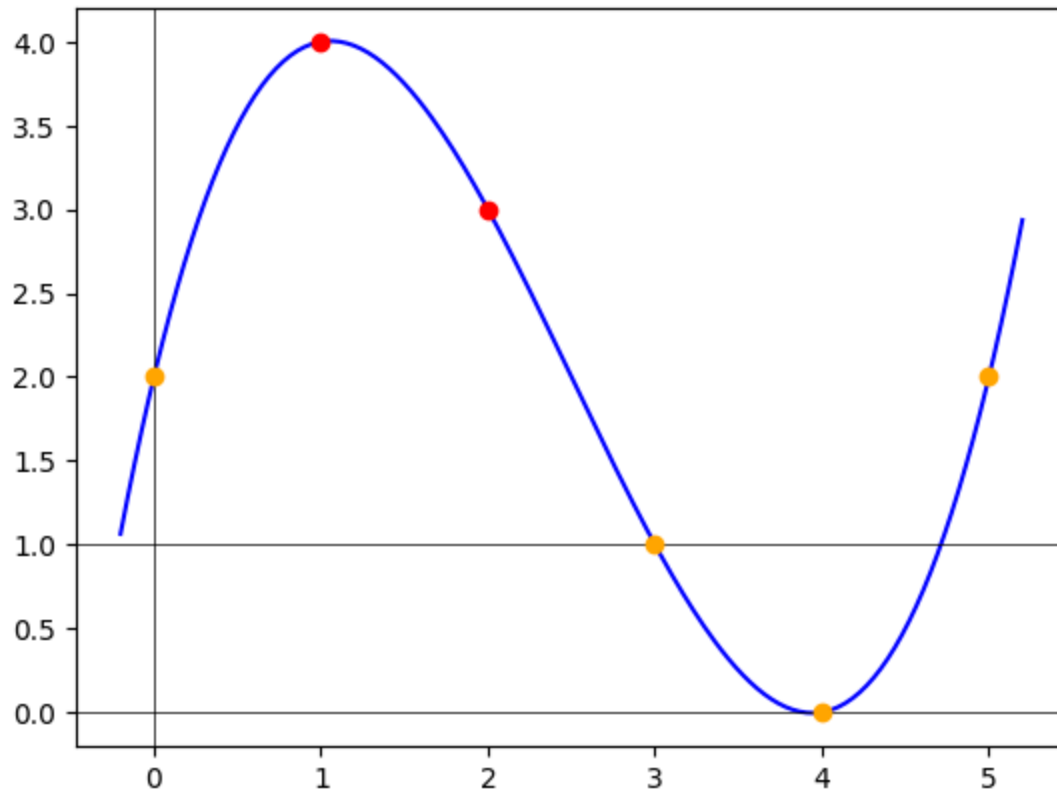
Mathematische Grundlagen: Lagrange Polynome (7)

Das Polynom ist eindeutig und kann aus vier Punkten (orange) rekonstruiert werden:



Mathematische Grundlagen: Lagrange Polynome (8)

Also auch aus anderen vier Punkten (orange):



Dann könnte man das doch so direkt zur Fehlerkorrektur nutzen, oder?

Problem

Ein Code besteht aus einem endlichen Alphabet. Die reellen Zahlen sind unendlich.

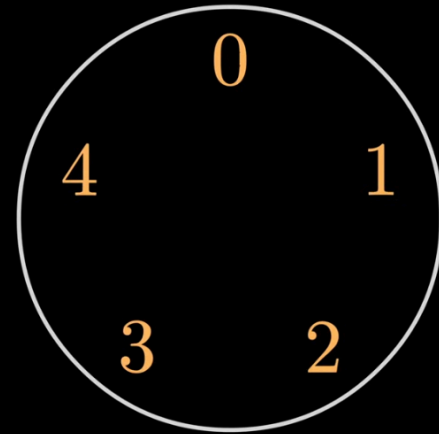
Lösung: Rechnen mit Modulo

Modular Arithmetic $(+, -, \times, \div) \mod p$

$$1 + (-3) \equiv 3 \mod 5$$

$$a + \underbrace{(-a)} \equiv 0 \mod 5$$

Additive inverse



$v^3(x)$

from [What are Reed-Solomon Codes? How computers recover lost data by vcubingx](#)

Lösung: Rechnen auf einem endlichen Körper

Mathematisch spricht man von einem **endlichen Körper** oder Galois-Körper (engl. finite or Galois field). Man kann mit Addieren und Multiplizieren den Zahlenbereich nicht verlassen.

Also der gleiche Ansatz wie bei CRC.

Fazit Fehlerkorrektur

- Reed-Solomon (RS) Codes gibt es in vielen Varianten. Wir haben nur das zugrundeliegende Prinzip betrachtet. Sie arbeiten aber immer mit Symbolen die aus mehreren Bits bestehen.
- RS Codes nutzen ein festgelegtes Generatorpolynom, das sowohl dem Sender (Encoder) als auch dem Empfänger (decoder) bekannt sein muss.
- RS Codes können sowohl falsche Werte erkennen, als auch fehlende Werte ersetzen.

Fazit und Ausblick

- In neueren Anwendungsbereichen werden RS-Codes zunehmend durch leistungsfähigere Codes wie die [Low-Density-Parity-Check-Codes \(LDPC\)](#) oder [Turbo-Codes \(TPC\)](#) abgelöst. Quelle: [Wikipedia](#)