

Schüler-SimuLab
Kursreihe stochastische
Simulationen
Kurs 2

Eine Alternative zum Linearen
Kongruenz-Generator :
Der Inverse Kongruenz-Generator

Stefan Hartmann

Forschungszentrum caesar

18. September 2006

Ein paar einleitende Worte

Im Kurs über die Erzeugung von Pseudo-Zufallszahlen (Kurs 5) haben wir uns ja in erster Linie mit dem Linearen Kongruenz-Generator beschäftigt. Dabei sind uns Schwächen aufgefallen, insbesondere dahingehend, dass in der zweidimensionalen Darstellung eine unerwünschte parallele Linienstruktur (und entsprechend in der dreidimensionalen Darstellung eine unerwünschte parallele Ebenenstruktur) erkennbar war. Wir werden uns in diesem Zusatzkurs mit einem weiteren Generator beschäftigen, der zumindestens diese Schwäche nicht mehr hat, nämlich mit dem sogenannten Inversen Kongruenz-Generator. Die Mathematik, die dahinter steckt, ist deutlich anspruchsvoller als die einfache „Division mit Rest“, die man für den Linearen Kongruenz-Generator als Grundlage brauchte. Wir müssen uns die Kongruenzrechnung etwas detaillierter anschauen, die wir ja im ersten Kurs schon kurz kennengelernt haben. Insbesondere müssen wir genauer lernen, wie man „modulo m “ rechnet. Dort werden wir sehen, dass einige Gleichungen leider nicht lösbar sind, von denen wir eigentlich gerne hätten, dass sie lösbar wären. Dieses Problem tritt beim Rechnen „modulo p “ nicht mehr auf, wenn p eine Primzahl ist. In diesem Zusammenhang wird der sogenannte Euklidische Algorithmus eine wichtige Rolle spielen, den ihr vielleicht aus der Schule bereits kennt. Wie beim Linearen Kongruenz-Generator werden wir uns auch beim Inversen Kongruenz-Generator Gedanken über die Periodenlänge machen und ein kleines Programm schreiben, mit dem man bei gegebenen Parametern schnell überprüfen kann, ob die Periodenlänge maximal ist. Anschließend wollen wir dann auch den Algorithmus des Inversen Kongruenz-Generators programmieren und damit eigene Pseudo-Zufallszahlen erzeugen. Ihr werdet also neben einer neuen Methode zur Erzeugung von Pseudo-Zufallszahlen auch eine ganze Menge an (vermutlich für euch neuer) interessanter Mathematik kennenlernen.

Ich wünsche euch viel Freude dabei und hoffe, dass euch der Kurs gefällt.

Bonn, den 16.05.2004

Stefan Hartmann

Kapitel 1

Der Inverse Kongruenz-Generator: der Algorithmus

Zur Erinnerung sei hier noch einmal der Algorithmus des Linearen Kongruenz-Generators erwähnt:

1. Schritt: Wähle

- einen **Modul** m mit $m > 0$,
- einen **Multiplikator** a mit $0 \leq a < m$,
- eine **Verschiebung** c mit $0 \leq c < m$,
- einen **Startwert** X_0 mit $0 \leq X_0 < m$.

2. Schritt: Berechne für $i \geq 0$:

$$X_{i+1} = \text{Rest}_m(a \cdot X_i + c).$$

3. Schritt: Breche ab, sobald eine Periode eintritt.

Wir werden nun den **Inversen Kongruenz-Generator** kennenlernen. Der Inverse Kongruenz-Generator arbeitet nach dem folgenden Algorithmus, der rein formal dem Algorithmus des Linearen Kongruenz-Generators sehr ähnelt:

Inverser Kongruenz-Generator

1. Schritt: Wähle

- einen **(Primzahl)-Modul p** mit $p > 0$, wobei p eine Primzahl ist,
- einen **Multiplikator a** mit $0 \leq a < p$,
- eine **Verschiebung c** mit $0 \leq c < p$,
- einen **Startwert X_0** mit $0 \leq X_0 < p$.

2. Schritt: Berechne für $i \geq 0$:

$$X_{i+1} = \text{Rest}_p(a \cdot X_i^{-1} + c).$$

3. Schritt: Breche ab, sobald eine Periode eintritt.

Soviel ändert sich also (scheinbar) nicht: Wir wählen den Modul m als Primzahl p , was wir vorher aber auch schon hätten tun können und nehmen statt X_i den Ausdruck X_i^{-1} , ansonsten bleibt alles gleich. Aber die Auswirkungen sind enorm, wie wir noch sehen werden!

Natürlich müssen wir noch klären, was wir unter dem Ausdruck X_i^{-1} verstehen und wie wir diesen Ausdruck berechnen. Daher wiederholen und vertiefen wir zunächst die Kongruenzrechnung und überlegen uns, wie man modulo einer Primzahl p auch Divisionen durchführen darf. Damit können wir in der Menge der ganzen Zahlen modulo p die gleichen Rechenoperationen durchführen wie in der Menge der reellen Zahlen.

Kapitel 2

Die Mathematik dahinter: Kongruenzrechnung und Euklidischer Algorithmus

Zunächst erinnern wir uns noch einmal daran, was wir überhaupt unter der Kongruenzrechnung verstehen. Wann sind zwei Zahlen „kongruent modulo m “?

Definition 2.1 (kongruent modulo m) *Es sei $m > 0$ beliebig vorgegeben. Dann nennen wir zwei ganze Zahlen a und b **kongruent modulo m** , wenn sie bei der Division durch m den gleichen Rest lassen.*

Hier die Schreibweise für „ a ist kongruent zu b modulo m “:

$$a \equiv b \pmod{m}.$$

Zum Glück – so haben wir im ersten Kurs gesehen – gibt es auch eine etwas bequemere Variante um zu überprüfen, ob zwei Zahlen kongruent zueinander sind:

Satz 2.2 *Es gilt genau dann $a \equiv b \pmod{m}$, wenn m ein Teiler von $a - b$ ist.*

Den Beweis dieser Aussage findet man im Skript zu Kurs 5.

Was bedeutet das nun? Wenn wir feststellen wollen, ob a und b kongruent modulo m sind, müssen wir einfach deren Differenz bilden, also $a - b$, und schauen, ob diese Differenz ein Vielfaches von m ist. Wenn ja, dann sind a und b kongruent modulo m . Wenn nein, dann sind a und b nicht kongruent modulo m . Umgekehrt: Wie erhalten wir aus einer beliebigen ganzen Zahl a alle Zahlen, die kongruent zu a modulo m sind? Indem wir beliebig oft m zu a addieren bzw. von a subtrahieren. Mit anderen Worten: Alle Zahlen, die einen Abstand zu a haben, der gerade ein Vielfaches von m ist, sind kongruent zu a modulo m .

Zum Beispiel sind -15 und 24 kongruent modulo 13 , weil ihre Differenz $(-15) - 24 = -39$ ein Vielfaches von 13 ist.

Diese Beziehungen kann man nun geschickt ausnutzen, wenn man „modulo m “ rechnet.

Beispiele:

Zum Beispiel gilt:

$$8 + 5 = 13 \equiv 2 \pmod{11}$$

und

$$8 + 3 + 12 \equiv 11 + 1 \equiv 1 \pmod{11}.$$

Klar? Man darf also „zwischendurch“ immer die Kongruenzen ausnutzen, die einem das Rechnen erheblich erleichtern können.

Die Gleichung

$$8 + x \equiv 0 \pmod{11}$$

hat die Lösung: $x \equiv 3 \pmod{11}$, d.h. es gilt:

$$-8 \equiv 3 \pmod{11}.$$

Allgemeiner gilt:

Satz 2.3 *Es seien $m \in \mathbb{N}$ und $a \in \mathbb{Z}$ beliebig gewählt. Dann ist die Gleichung*

$$a + x \equiv 0 \pmod{m}$$

immer lösbar.

Man kann also sagen: Egal, wie $m \in \mathbb{N}$ aussieht, es gibt in jedem Fall zu jedem $a \in \mathbb{Z}$ ein $-a \in \mathbb{Z}$, also ein **Inverses bezüglich der Addition**, für das

$$a + (-a) \equiv 0 \pmod{m}$$

gilt. Hierbei ist 0 (und natürlich alle ganzen Zahlen, die kongruent dazu sind) das sogenannte **neutrale Element bezüglich der Addition**, d.h. für alle $b \in \mathbb{Z}$ gilt:

$$b + 0 \equiv b \pmod{m}.$$

Das **neutrale Element bezüglich der Multiplikation** ist 1 (und natürlich alle ganzen Zahlen, die kongruent dazu sind), d.h. für alle $b \in \mathbb{Z}$ gilt:

$$b \cdot 1 \equiv b \pmod{m}.$$

Die Frage ist jetzt: Gibt es auch zu jedem $a \in \mathbb{Z}$, $a \not\equiv 0 \pmod{m}$, ein **Inverses bezüglich der Multiplikation**, also ein $a^{-1} \in \mathbb{Z}$, für das

$$a \cdot a^{-1} \equiv 1 \pmod{m}$$

gilt?

Es stellt sich also die folgende Frage:

Es seien $m \in \mathbb{N}$ und $a \in \mathbb{Z}$, $a \not\equiv 0 \pmod{m}$ beliebig gewählt. Ist dann die Gleichung

$$a \cdot x \equiv 1 \pmod{m}$$

immer lösbar?

Testet das doch einfach mal:

Gruppenaufgabe:

Ist die Gleichung

$$3 \cdot x \equiv 1 \pmod{15}$$

lösbar?

Man sieht: Man muss anscheinend etwas für a und m voraussetzen, damit die Gleichung

$$a \cdot x \equiv 1 \pmod{m}$$

lösbar ist. Man kann zeigen, dass diese Gleichung genau dann lösbar ist, wenn

m und a teilerfremd sind. Wir wollen aber möglichst, dass die Gleichung „immer“ (d.h. für alle $a \in \mathbb{Z}$, die keine Vielfachen von m sind) lösbar ist, damit wir uns nicht jedes Mal Gedanken über die Teilerfremdheit machen müssen. Daher wählen wir m im Folgenden als Primzahl und bezeichnen diesen Modul ab jetzt (suggestiv) mit p . Eine Primzahl p ist zu allen Zahlen teilerfremd, die keine Vielfachen von p sind. Wenn wir nun für a nur Zahlen zulassen, die echt kleiner als p sind (und auf die können wir uns ja beschränken, da es für alle Zahlen x eine Zahl a mit $0 \leq a < p$ gibt, die kongruent zu x modulo p ist!), dann haben wir die wichtige Aussage:

Satz 2.4 *Es sei p eine Primzahl. Dann ist für alle $a \in \mathbb{Z}$ mit $0 < a < p$ die Gleichung*

$$a \cdot x \equiv 1 \pmod{p}$$

immer lösbar.

Nun macht die folgende Bezeichnung Sinn:

Bezeichnung 2.5 ($a^{-1} \pmod{p}$)

Es sei $a \in \mathbb{Z}$ vorgegeben. Man schreibt für $x \in \mathbb{Z}$:

$$x \equiv a^{-1} \pmod{p},$$

wenn

$$a \cdot x \equiv 1 \pmod{p}$$

gilt. Im Falle $a \equiv 0 \pmod{p}$ existiert kein solches $x \in \mathbb{Z}$. In diesem Fall setzen wir vereinbarungsgemäß: $a^{-1} := 0$.

Gruppenaufgabe:

- (a) Überlege dir, dass $a^{-1} \pmod{p}$ in \mathbb{Z} (außer im Fall $a = 0$ nach Vereinbarung) nicht eindeutig bestimmt ist, wenn es existiert. Nenne mindestens zwei Zahlen $x \in \mathbb{Z}$, für die

$$x \cdot 5 \equiv 1 \pmod{11}$$

gilt.

- (b) Nehmen wir einmal an, wir haben ein $x \in \mathbb{Z}$ mit der Eigenschaft $a \cdot x \equiv 1 \pmod{p}$ gefunden. Wie bekommen wir dann alle anderen?
- (c) Durch welche zusätzliche Forderung können wir erreichen, dass $a^{-1} \pmod{p}$ eindeutig bestimmt ist?

Wir schreiben häufig auch einfach a^{-1} statt $a^{-1} \pmod{p}$, wenn klar ist, welcher Modul p gerade betrachtet wird. Wenn wir im Folgenden von a^{-1} oder $a^{-1} \pmod{p}$ reden, meinen wir in der Regel das eindeutig bestimmte $a^{-1} \pmod{p}$ mit $0 \leq a^{-1} < p$. Dennoch sollte man immer im Hinterkopf behalten, dass es auch andere $x \in \mathbb{Z}$ mit $a \cdot x \equiv 1 \pmod{p}$ gibt, die sich um Vielfache von p unterscheiden.

Wegen $a \cdot a^{-1} \equiv 1 \pmod{p}$ könnten wir statt a^{-1} (vielleicht für euch intuitiver) $\frac{1}{a}$ schreiben, also so wie in der Menge der reellen Zahlen, aber in der Kongruenzrechnung ist diese Bezeichnung eher unüblich. Dadurch wird aber klarer, wie man eine Division modulo p durchführt. Man setzt nämlich:

$$a : b \stackrel{\text{def}}{=} \frac{a}{b} \stackrel{\text{def}}{=} a \cdot b^{-1}.$$

Es ist klar, dass dieser Ausdruck nur für $b \neq k \cdot p$ ($k \in \mathbb{Z}$) sinnvoll ist, also nur für $b \not\equiv 0 \pmod{p}$. Dies entspricht unserer gewohnten Regel, dass man „nicht durch 0 teilen darf“.

Die Division üben wir jetzt noch etwas:

Beispiel

Wir wollen $4 : 5 \pmod{7}$ berechnen. Dazu ermitteln wir zunächst durch Ausprobieren das Inverse von 5 modulo 7. Wir testen also:

$$\begin{aligned}1 \cdot 5 &\equiv 5 \not\equiv 1 \pmod{7}, \\2 \cdot 5 &\equiv 10 \not\equiv 1 \pmod{7}, \\3 \cdot 5 &\equiv 16 \equiv 1 \pmod{7}.\end{aligned}$$

Daher gilt:

$$5^{-1} \equiv 3 \pmod{7},$$

also:

$$4 : 5 \equiv 4 \cdot 5^{-1} \equiv 4 \cdot 3 \equiv 12 \equiv 5 \pmod{7}.$$

Aufgabe 1:

Berechne:

$$\begin{aligned}5 : 6 &\pmod{11}, \\5 : 2 &\pmod{7}, \\\frac{6}{3} &\pmod{13}, \\\frac{2}{3} &\pmod{13}.\end{aligned}$$

Vielleicht habt ihr anhand der Aufgabe auch gemerkt: Wenn es sich um einen Primzahlmodul handelt, darf man wie gewöhnt kürzen und erweitern, aber nur mit Zahlen, die keine Vielfache der Primzahl sind. Denn wenn man Zähler und Nenner mit einem Vielfachen von p erweitert, werden beide automatisch kongruent zu 0!

Die Frage, die sich jetzt natürlich aufdrängt, ist die folgende:

Wie berechnet man allgemein $a^{-1} \pmod{p}$?

Für kleine Primzahlen p kann man $a^{-1} \pmod{p}$ erraten (siehe oben). Für große Primzahlen ist das nicht mehr so einfach. Da müsste man alle Zahlen bis p durchprobieren und das könnte unter Umständen ziemlich lange dauern. Für große Primzahlen p brauchen wir als mathematisches Hilfsmittel zur Berechnung von $a^{-1} \pmod{p}$ den **Euklidischen Algorithmus**. Es handelt sich dabei um die bereits bekannte Division mit Rest, die man iterativ so lange durchführt, bis irgendwann mal kein Rest mehr vorhanden ist. Dabei geht man nach dem folgenden Algorithmus vor:

Euklidischer Algorithmus

1. Schritt: Wähle zwei ganze Zahlen b_0 und a_0 mit $a_0 \neq 0$.

[Beginn der Schleife]

2. Schritt: Teile für $i \geq 0$ die Zahl b_i durch a_i mit Rest:

$$b_i = q_i \cdot a_i + r_i \quad \text{mit } q_i, r_i \in \mathbb{Z} \text{ und } 0 \leq r_i < a_i.$$

Setze dann: $b_{i+1} := a_i$ und $a_{i+1} := r_i$.

3. Schritt: Breche ab, sobald der Rest r_i gleich 0 ist. Gehe zurück zu 2.

[Ende der Schleife]

In schematischer Form sieht der euklidische Algorithmus so aus:

$$\begin{aligned}
 b_0 &= q_0 \cdot \underbrace{a_0}_{=:b_1} + \underbrace{r_0}_{=:a_1} , & 0 \leq r_0 < a_0 & , \\
 b_1 &= q_1 \cdot \underbrace{a_1}_{=:b_2} + \underbrace{r_1}_{=:a_2} , & 0 \leq r_1 < a_1 & , \\
 b_2 &= q_2 \cdot \underbrace{a_2}_{=:b_3} + \underbrace{r_2}_{=:a_3} , & 0 \leq r_2 < a_2 & , \\
 &\vdots & & \vdots \\
 b_{n-1} &= q_{n-1} \cdot \underbrace{a_{n-1}}_{=:b_n} + \underbrace{r_{n-1}}_{=:a_n} , & 0 \leq r_{n-1} < a_{n-1} & , \\
 b_n &= q_n \cdot a_n .
 \end{aligned}$$

Man kann sich nun fragen, ob der Algorithmus immer abbricht. Das ist ja zunächst einmal nicht klar.

Gruppenaufgabe:

Überlege dir, dass der Euklidische Algorithmus in jedem Fall irgendwann abbricht, egal wie man b_0 und $a_0 \neq 0$ auch immer wählt. Tipp: Betrachte mal die Folge $(r_n)_{n \in \mathbb{N}_0}$ der Reste. Was kannst du über diese Folge aussagen?

Den Euklidischen Algorithmus wollen wir nun dazu nutzen um den größten gemeinsamen Teiler zweier ganzer Zahlen zu berechnen. Was der größte gemeinsame Teiler zweier natürlicher Zahlen ist, wisst ihr aus der Schule. Wir verallgemeinern und präzisieren diese Definition hier etwas:

Bezeichnung 2.6 ($ggT(a, b)$)

Der **größte gemeinsame Teiler** zweier ganzer Zahlen a und b (wobei nicht beide Zahlen gleich 0 sind) ist die größte natürliche Zahl, die a und b ohne Rest teilt. Man verwendet dafür die Bezeichnung $ggT(a, b)$.

Aus technischen Gründen setzen wir zudem $ggT(0, 0) := 0$.

Wir nennen a und b **relativ prim**, wenn $ggT(a, b) = 1$ gilt.

Beispiele:

$$ggT(12, 18) = 6,$$

$$ggT(-4, 14) = 2,$$

$$ggT(5, 0) = 5.$$

Man kann nun mit Hilfe des Euklidischen Algorithmus den größten gemeinsamen Teiler zweier ganzer Zahlen, die nicht beide gleich 0 sind, berechnen. Dazu benötigen wir einen kleinen Hilfssatz:

Satz 2.7 *Es gilt für zwei ganze Zahlen a und b die folgende Beziehung:*

$$ggT(a, b) = ggT(b, a - b),$$

d.h. man kann $ggT(a, b)$ sukzessive ausrechnen, indem man wiederholt von der größeren der beiden Zahlen die kleinere abzieht und dann jeweils den größten gemeinsamen Teiler von der kleineren der beiden Zahlen und der Differenz der beiden Zahlen bildet.

Beispiel:

Es gilt:

$$\begin{aligned} ggT(48, 30) &= ggT(30, 18) \\ &= ggT(18, 12) \\ &= ggT(12, 6) \\ &= ggT(6, 6) \\ &= 6. \end{aligned}$$

Mit dem Euklidischen Algorithmus macht man nun genau diese sogenannte „Wechselwegnahme“, um den größten gemeinsamen Teiler zweier ganzer Zahlen zu bestimmen, nur in einer „Speed-Version“.

Denn: Haben wir eine Division mit Rest durchgeführt:

$$a = bq + r,$$

so gilt nach dem obigen Satz:

$$ggT(a, b) = ggT(b, a - b) = ggT(b, a - 2b) = \dots = ggT(b, a - qb) = ggT(b, r).$$

Gruppenaufgabe:

Überlege dir nun, dass mit den Bezeichnungen aus dem Schema des Euklidischen Algorithmus folgendes gilt:

$$ggT(b_0, a_0) = a_n = r_{n-1}.$$

Beispiel:

Wir möchten mit Hilfe des Euklidischen Algorithmus gerne $ggT(84, 315)$ bestimmen.

Führen wir den Euklidischen Algorithmus für $b_0 = 315$ und $a_0 = 84$ durch, so erhalten wir das folgende Schema:

$$315 = 3 \cdot 84 + 63$$

$$84 = 1 \cdot 63 + 21$$

$$63 = 3 \cdot 21 + 0.$$

Wie wir uns gerade in der Aufgabe überlegt haben, gilt also:

$$ggT(315, 84) = 21.$$

Aufgabe 2:

Berechne mit Hilfe des Euklidischen Algorithmus den größten gemeinsamen Teiler von

- (a) 308 und 70,
- (b) 396 und 210.

Man kann aber mit dem Euklidischen Algorithmus noch mehr erreichen als die Berechnung des größten gemeinsamen Teilers zweier ganzer Zahlen. Das ist auch dringend nötig, denn bisher haben wir auch noch nicht viel gewonnen. Vielleicht fragt ihr euch schon ungeduldig: Was bitte hat der größte gemeinsame Teiler mit dem Ausdruck $x^{-1} \pmod{p}$ zu tun, den wir ja eigentlich mit dem Euklidischen Algorithmus berechnen wollten? Nun ja, mehr als man anfangs glaubt, und der Schlüssel auf dem Weg dorthin wird durch den

folgenden Satz gegeben:

Satz 2.8 *Der größte gemeinsame Teiler $ggT(a, b)$ von zwei ganzen Zahlen a und b kann als Linearkombination von a und b mit ganzzahligen Koeffizienten dargestellt werden, d.h. es gibt $c, d \in \mathbb{Z}$ mit*

$$ggT(a, b) = ca + db.$$

Insbesondere gilt: Wenn a und b relativ prim sind, dann hat die Gleichung

$$1 = xa + yb$$

eine ganzzahlige Lösung (x, y) .

Man beachte, dass die $c, d \in \mathbb{Z}$ mit

$$ggT(a, b) = ca + db$$

nicht eindeutig bestimmt sind. Erfüllt das Paar (c, d) diese Gleichung, so auch alle Paare (c', d') mit

$$c' = c + kb \quad \text{und} \quad d' = d - ka$$

mit einem beliebigen $k \in \mathbb{Z}$.

Die Frage ist jetzt natürlich: Wie berechne ich c und d ? Aber bevor wir uns damit beschäftigen, wollen wir uns erst mal vergewissern, dass diese Darstellung dann auch tatsächlich zum Ziel führt! Also: Wir haben einen Primzahlmodul p gewählt, haben X_i mit $0 \leq X_i < p$ aus dem Algorithmus bestimmt und wollen jetzt X_i^{-1} bestimmen. Im Falle $X_i = 0$ gilt ja nach Vereinbarung: $X_i^{-1} := 0$, und im Falle $0 < X_i < p$ sind X_i und p teilerfremd, da p ja eine Primzahl ist. Daher gilt: $ggT(X_i, p) = 1$, und es gibt nach dem

obigen Satz zwei ganze Zahlen c und d mit

$$c \cdot X_i + d \cdot p = 1.$$

Nun rechnen wir diese Gleichung modulo p . Es gilt: $p \equiv 0 \pmod{p}$, also auch $d \cdot p \equiv 0 \pmod{p}$ und daher:

$$c \cdot X_i \equiv c \cdot X_i + \underbrace{d \cdot p}_{\equiv 0 \pmod{p}} \equiv 1 \pmod{p}.$$

Die bedeutet aber gerade gemäß Bezeichnung 1:

$$c = X_i^{-1} \pmod{p},$$

d.h. wir haben ein X_i^{-1} gefunden!

Wie kommt man auf die ganzzahlige Linearkombination?

Jetzt bleibt also wirklich nur noch zu klären, wie wir es schaffen mit Hilfe des Euklidischen Algorithmus zwei ganze Zahlen c und d zu finden, so dass für zwei fest gewählte ganze Zahlen a und b die Beziehung

$$ca + db = ggT(a, b)$$

gilt. Wir werden dabei die Bestimmung auf zwei Arten durchführen: Erst so, wie wir es am besten per Hand ausrechnen und dann so, wie wir es am geeignetsten auf dem Computer in Form eines Algorithmus programmieren.

Wir geben aber nicht **nur** den Computeralgorithmus an, weil dieser wenig intuitiv ist. Zunächst kümmern wir uns also darum, wie wir die Linearkombination am einsichtigsten per Handrechnung ermitteln.

Im Falle $a = 0 = b$ ist es klar, dass wir $c = 0 = d$ wählen können, da nach Vereinbarung $ggT(0, 0) = 0$ gilt.

Andernfalls gelte etwa $a \neq 0$. Dann können wir für $a_0 := a$ und $b_0 := b$ den Euklidischen Algorithmus durchführen und erhalten das Schema des Euklidischen Algorithmus:

$$\begin{aligned}
 b_0 &= q_0 \cdot \underbrace{a_0}_{=:b_1} + \underbrace{r_0}_{=:a_1} \quad , & 0 \leq r_0 < a_0 \quad , \\
 b_1 &= q_1 \cdot \underbrace{a_1}_{=:b_2} + \underbrace{r_1}_{=:a_2} \quad , & 0 \leq r_1 < a_1 \quad , \\
 b_2 &= q_2 \cdot \underbrace{a_2}_{=:b_3} + \underbrace{r_2}_{=:a_3} \quad , & 0 \leq r_2 < a_2 \quad , \\
 &\vdots & & \vdots \\
 b_{n-2} &= q_{n-2} \cdot \underbrace{a_{n-2}}_{=:b_{n-1}} + \underbrace{r_{n-2}}_{=:a_{n-1}} \quad , & 0 \leq r_{n-2} < a_{n-2} \quad , \\
 b_{n-1} &= q_{n-1} \cdot \underbrace{a_{n-1}}_{=:b_n} + \underbrace{r_{n-1}}_{=:a_n} \quad , & 0 \leq r_{n-1} < a_{n-1} \quad , \\
 b_n &= q_n \cdot a_n \quad .
 \end{aligned}$$

Nun fangen wir mit der vorletzten Zeile an:

$$b_{n-1} = q_{n-1} \cdot a_{n-1} + r_{n-1}$$

und stellen nach r_{n-1} um:

$$r_{n-1} = b_{n-1} - q_{n-1} \cdot a_{n-1}. \tag{2.1}$$

Langfristig soll da ja sowas stehen wie:

$$ggT(a, b) = r_{n-1} = c b_0 + d a_0 = cb + da$$

mit zwei ganzen Zahlen c und d . Nun versuchen wir sukzessive b_{n-1} und a_{n-1} durch einen Ausdruck mit b_{n-2} und a_{n-2} , dann durch einen Ausdruck mit b_{n-3} und a_{n-3} , usw. zu ersetzen, bis wir schließlich bei einem Ausdruck mit

b_0 und a_0 ankommen.

Zunächst wissen wir ja, dass $b_{n-1} = a_{n-2}$ und $a_{n-1} = r_{n-2}$ ist, d.h. wir können (2.1) auch so schreiben:

$$r_{n-1} = a_{n-2} - q_{n-1} \cdot r_{n-2}. \quad (2.2)$$

Dann nehmen wir uns die drittletzte Zeile vor:

$$b_{n-2} = q_{n-2} \cdot a_{n-2} + r_{n-2}$$

und stellen nach r_{n-2} um:

$$r_{n-2} = b_{n-2} - q_{n-2} \cdot a_{n-2}. \quad (2.3)$$

Nun setzen wir (2.3) in (2.2) ein und erhalten:

$$\begin{aligned} r_{n-1} &= a_{n-2} - q_{n-1} \cdot (b_{n-2} - q_{n-2} \cdot a_{n-2}) \\ &= -q_{n-1} \cdot b_{n-2} + (1 - q_{n-1} \cdot q_{n-2}) \cdot a_{n-2}. \end{aligned}$$

Unser erstes Ziel ist erreicht: Es tauchen nur noch Terme mit b_{n-2} und a_{n-2} auf.

So fährt man nun weiter fort. Am besten sieht man das Vorgehen natürlich an einem konkreten Beispiel:

Beispiel:

Wir möchten $ggT(3370, 315)$ bestimmen und $ggT(3370, 315)$ als ganzzahlige Linearkombination von 3370 und 315 darstellen.

Zunächst wenden wir den Euklidischen Algorithmus an und erhalten:

$$3370 = 10 \cdot 315 + 220 \quad , \quad 0 \leq 220 < 315 \quad ,$$

$$315 = 1 \cdot 220 + 95 \quad , \quad 0 \leq 95 < 220 \quad ,$$

$$220 = 2 \cdot 95 + 30 \quad , \quad 0 \leq 30 < 95 \quad ,$$

$$95 = 3 \cdot 30 + 5 \quad , \quad 0 \leq 5 < 30 \quad ,$$

$$30 = 6 \cdot 5 \quad .$$

Daher gilt:

$$ggT(3370, 315) = 5.$$

Durch fortwährendes „Ineinander-Einsetzen“ bekommt man jetzt die gewünschte Darstellung:

Man beginnt mit der vorletzten Gleichung und löst nach dem Rest auf. Dann löst man aus die drittletzte Gleichung nach dem Rest auf und setzt dies in die umgestellte vorletzte Gleichung ein. Das Gleiche (Auflösen nach dem Rest und Einsetzen) macht man dann mit der viertletzten Gleichung, bis man am Ende alle Gleichungen „abgearbeitet“ hat:

$$\begin{aligned}
5 &= 95 - 3 \cdot 30 \\
&= 95 - 3 \cdot (220 - 2 \cdot 95) \\
&= 7 \cdot 95 - 3 \cdot 220 \\
&= 7 \cdot (315 - 1 \cdot 220) - 3 \cdot 220 \\
&= (-10) \cdot 220 + 7 \cdot 315 \\
&= (-10) \cdot (3370 - 10 \cdot 315) + 7 \cdot 315 \\
&= 107 \cdot 315 - 10 \cdot 3370,
\end{aligned}$$

d.h. wir haben eine Darstellung

$$ggT(3370, 315) = 5 = c \cdot 3370 + d \cdot 315$$

mit

$$c = -10 \quad \text{und} \quad d = 107.$$

Aufgabe 3:

Bestimme mit der soeben dargestellten Methode den größten gemeinsamen Teiler $ggT(32174, 6418)$ und stelle $ggT(32174, 6418)$ als ganzzahlige Linearkombination von 32174 und 6418 dar!

Noch einmal zur Erinnerung: Mit dieser Methode können wir jetzt für alle Primzahlen p und alle $0 < X_i < p$ ein $X_i^{-1} \pmod{p}$ berechnen, also eine ganze Zahl c , für die

$$c \cdot X_i \equiv 1 \pmod{p}$$

gilt. Denn: Da X_i und p teilerfremd sind, gibt es ganze Zahlen c und d mit

$$1 = ggT(X_i, p) = c \cdot X_i + d \cdot p.$$

Rechnet man modulo p , so erhält man:

$$1 \equiv c \cdot X_i \pmod{p},$$

und es gilt in der Tat:

$$c \equiv X_i^{-1} \pmod{p}.$$

Gruppenaufgabe:

Es sei ein Primzahlmodul $p = 647$ gewählt.

Berechne $20^{-1} \pmod{647}$.

Nun könnten wir diesen Algorithmus natürlich als Computerprogramm implementieren. Das Ganze sähe dann so aus, dass der Computer zunächst den Euklidischen Algorithmus durchführt und dann rückwärts die ganzzahlige Linearkombination bildet. Aber so etwas ist natürlich unschön!

Wesentlich besser wäre es doch, man könnte beides gleichzeitig machen: die sukzessive Division mit Rest und das Auffinden der Darstellung des größten gemeinsamen Teilers als Linearkombination. Um dies zu erreichen, versuchen wir im Folgenden eine Rekursionsformel herzuleiten. (Dies wird im Präsenzkurs nicht gemacht.)

Wir setzen: $r_{-2} := b_0$ und $r_{-1} := a_0$.

Offenbar gibt es, das sieht man sofort am Euklidischen Algorithmus, für alle $i \in \{-2, -1, 0, 1, \dots\}$ ganze Zahlen s_i und t_i mit

$$r_i = s_i \cdot a_0 + t_i \cdot b_0.$$

Diese Gleichung wollen wir ja für $i = n$ lösen, das ist unser Ziel. Vielleicht gelingt es uns ja, eine Rekursionsformel für die Koeffizienten s_i und t_i herzuleiten, die wir dann leicht programmieren können und die parallel zum Euklidischen Algorithmus abläuft.

Nun, wegen

$$a_i = r_{i-1} \quad \text{und} \quad b_i = a_{i-1} = r_{i-2}$$

folgt aus

$$b_i = q_i \cdot a_i + r_i$$

die Beziehung:

$$r_{i-2} = q_i \cdot r_{i-1} + r_i,$$

also:

$$r_i = -q_i \cdot r_{i-1} + r_{i-2}.$$

Wir erhalten daher:

$$\begin{aligned} s_i \cdot a_0 + t_i \cdot b_0 &= -q_i \cdot (s_{i-1} \cdot a_0 + t_{i-1} \cdot b_0) + s_{i-2} \cdot a_0 + t_{i-2} \cdot b_0 \\ &= (s_{i-2} - q_i \cdot s_{i-1}) \cdot a_0 + (t_{i-2} - q_i \cdot t_{i-1}) \cdot b_0, \end{aligned}$$

also können wir die folgende Rekursion festlegen:

$$s_i := s_{i-2} - q_i \cdot s_{i-1},$$

$$t_i := t_{i-2} - q_i \cdot t_{i-1}.$$

Da man immer auf die letzten beiden Werte zugreift, benötigen wir für diese Rekursion zwei Startwertpaare, (s_{-2}, t_{-2}) und (s_{-1}, t_{-1}) .

Wegen

$$r_{-2} = b_0 = 1 \cdot b_0 + 0 \cdot a_0$$

und

$$r_{-1} = a_0 = 0 \cdot b_0 + 1 \cdot a_0$$

setzen wir:

$$s_{-2} := 0 \quad \text{und} \quad t_{-2} := 1$$

und

$$s_{-1} := 1 \quad \text{und} \quad t_{-1} := 0.$$

Kommen wir zu unserem Problem zurück: Wir wollen zu einem gegebenen Primzahlmodul ein Inverses $X^{-1} \pmod{p}$ bestimmen. Wenn wir $a_0 = X$ und $b_0 = p$ setzen, dann suchen wir s_{n-1} und t_{n-1} mit

$$1 = s_{n-1} \cdot a_0 + t_{n-1} \cdot b_0 = s_{n-1} \cdot X + t_{n-1} \cdot p.$$

Im nächsten Schritt rechnen wir modulo p . Dadurch fällt das t_{n-1} weg! Wir sind also eigentlich nur an der Folge der s_i interessiert, um dann schließlich auf $s_{n-1} \equiv X^{-1} \pmod{p}$ zu kommen. Daher brauchen wir nur die Rekursion der Folge $(s_i)_{i \geq 0}$.

Ein Algorithmus, der zu einem gegebenen Primzahlmodul p und einer ganzen Zahl X die Inverse $X^{-1} \pmod{p}$ berechnet, könnte nun wie folgt aussehen:

1. Schritt: Setze (wir fangen bei 0 anstatt bei -2 an, machen aber das Gleiche wie oben):

$$r_0 = p,$$

$$r_1 = X$$

$$s_0 = 0,$$

$$s_1 = 1.$$

[**Beginn der Schleife**]

Mache das folgende so lange, wie $r_1 > 1$ gilt:

2. Schritt: Berechne:

$$r_2 = \text{Rest}_{r_0}(r_1),$$

$$s_2 = s_0 - \text{INT}(r_0/r_1) \cdot s_1.$$

3. Schritt: Gehe zum nächsten Schritt und setze die Variablen zurück:

$$s_0 = s_1,$$

$$s_1 = s_2,$$

$$r_0 = r_1,$$

$$r_1 = r_2.$$

[**Ende der Schleife**]

4. Schritt: Setze:

$$X^{-1} = \text{Rest}_p(s_1)$$

und beende das Programm.

Kapitel 3

Die Eigenschaften des Inversen Kongruenz-Generators

Wir sollten uns so langsam mal Gedanken darüber machen, was uns dieser Inverse Kongruenz-Generator eigentlich bringt. Hierbei stellen sich zwei wichtige Fragenkomplexe:

1. Frage nach der Verbesserung der bisherigen Nachteile:

Was für tolle Eigenschaften hat er denn, die der Lineare Kongruenz-Generator nicht hat? Werden die Schwächen des Linearen Kongruenz-Generators (die linearen Strukturen) bereinigt? Kann man das auch optisch feststellen? Gibt es auch ansonsten keine unerwünschten Musterbildungen?

2. Frage nach dem Erhalt der bisherigen Vorteile:

Können wir die Vorteile des Linearen Kongruenz-Generators retten, die zum Beispiel darin bestanden, dass wir ganz genau wussten, unter welchen Umständen man eine maximale Periodenlänge erhält?

(Wenn dies nicht der Fall wäre, könnte man zurecht die Frage stellen, ob die Vorteile des Inversen Kongruenz-Generators wirklich so groß sind, dass wir die eventuellen Nachteile in Kauf nehmen.)

Zunächst einmal zur zweiten Fragekategorie: Wir haben Glück!

Wie beim Linearen Kongruenz-Generator ist die maximale Periodenlänge gleich der Größe des Moduls, hier also gleich p . Die Voraussetzungen, wann wir eine maximale Periodenlänge haben, kann man wie beim Linearen Kongruenz-Generator ganz genau angeben.

Dazu zitiere ich mal einen Satz:

Satz 3.1 (Eichenauer/Lehn, 1986) *Es seien a, c und p die Parameter des Inversen Kongruenz-Generators. Dann gilt für die Periodenlänge λ (beginnend bei $X_0 = 0$) folgendes:*

- (i) Wenn es eine ganze Zahl q mit $q^2 \equiv 4a + c^2 \not\equiv 0 \pmod{p}$ gibt, dann ist die um eins vermehrte Periodenlänge $\lambda + 1$ ein Teiler von $p - 1$.*
- (ii) Wenn $4a + c^2 \equiv 0 \pmod{p}$ gilt, dann ist die Periodenlänge λ gleich $p - 1$.*
- (iii) Wenn es keine ganze Zahl q mit $q^2 \equiv 4a + c^2 \pmod{p}$ gibt, dann ist die um eins vermehrte Periodenlänge $\lambda + 1$ ein Teiler von $p + 1$.*

Man sieht also: Nur in Fall (iii) kann man also eine maximale Periodenlänge erwarten, denn dann gilt im Idealfall: $\lambda + 1 = p + 1$, also: $\lambda = p$. Dass dieser Fall tatsächlich eintreten kann, soll in der nächsten Aufgabe zunächst einmal ohne Computerprogramm erfahren werden:

Aufgabe 4:

Prüfe nach, dass der Inverse Kongruenz-Generator mit $a = 1$, $c = 2$ und $p = 11$ maximale Periodenlänge hat, indem du für $X_0 = 0$ die Folgenglieder X_1, \dots, X_{10} berechnest und verifizierst, dass sie alle Zahlen zwischen 1 und 10 annehmen.

Das Problem ist nun:

Auch wenn der Fall (iii) eintritt, d.h. auch wenn es keine ganze Zahl q mit $q^2 \equiv 4a + c^2 \pmod{p}$ gibt, kann man sich längst nicht sicher sein, dass die Periodenlänge maximal ist! Man weiß dann nur, dass die um eins vermehrte Periodenlänge ein Teiler von $p+1$ ist, aber nicht unbedingt, dass dies gleich $p+1$ selbst ist.

Der folgende Teil zur algorithmischen Ermittlung der Perionenlänge ist für sehr interessierte Leser als Zusatz zum Nachlesen in diesem Skript gedacht; er wird im Präsenzkurs nicht behandelt:

In der nächsten Aufgabe kannst du sehen, dass es tatsächlich passieren kann, dass die Periodenlänge nicht maximal ist, auch wenn es keine ganze Zahl q mit $q^2 \equiv 4a + c^2 \pmod{p}$ gibt. Dort ist also die Bedingung (iii) aus dem letzten Satz erfüllt, und λ ist leider nicht gleich p . Stattdessen ist $\lambda + 1$ ein **echter** Teiler von $p + 1$, d.h. die Periodenlänge ist nicht maximal:

Zusatzaufgabe:

Prüfe nach, dass der Inverse Kongruenz-Generator mit $a = 3$, $c = 7$ und $p = 11$ die Bedingung erfüllt, dass keine ganze Zahl q mit $q^2 \equiv 4a + c^2 \pmod{p}$ existiert. Zeige nun durch Berechnung der Folgenglieder bis zur ersten Periode, dass dieser Generator trotzdem keine maximale Periodenlänge hat! Zeige, dass die um eins vermehrte Periodenlänge $\lambda + 1$ aber ein Teiler von $p + 1$ ist, wie es der Satz ja aussagt.

Wie kann man nun, ohne den Generator selbst programmieren zu müssen, herausfinden, ob für gegebene Parameter a , c und p die Periodenlänge maximal ist?

Es gibt einen sehr einfachen Algorithmus dafür, den ich kurz angeben möchte und den du dann in Excel umsetzen sollst.

Gegeben seien eine Primzahl p sowie zwei ganze Zahlen a und c mit $0 \leq a, c < p$.

1. Schritt: Überprüfe, ob für **alle** $0 \leq q < p$ die Beziehung

$$q^2 \not\equiv 4a + c^2 \pmod{p}$$

gilt.

(1) Falls nein, ist die Periodenlänge nach dem obigen Satz auf gar keinen Fall maximal und wir verwerfen die Parameter und beenden den Algorithmus.

(2) Falls ja, gehen wir zu Schritt 2.

2. Schritt: Setze:

$$\tau_0 = 0,$$

$$\tau_1 = 1.$$

3. Schritt: Bestimme das kleinste $n \in \mathbb{N}$, das die folgende Eigenschaft besitzt:

Für τ_n , welches durch die Rekursion

$$\tau_i \equiv c \cdot \tau_{i-1} + a \cdot \tau_{i-2} \pmod{p} \quad , \quad 0 \leq \tau_i < p \quad , \quad (2 \leq i \leq n)$$

erzeugt wurde, gilt: $\tau_n = 0$.

(Hierbei genügt es nach dem obigen Satz, für die Teiler n von $p + 1$ zu überprüfen, ob $\tau_n = 0$ gilt. Ihr braucht das aber nicht zu beachten, sondern könnt das für alle $n \in \mathbb{N}$ überprüfen, dann ist es einfacher in Excel umzusetzen.)

4. Schritt: Die Periodenlänge λ berechnet sich zu $\lambda = n - 1$.

5. Schritt: Wenn $\lambda = p$ gilt, dann ist die Periodenlänge maximal und wir haben geeignete Parameter gefunden. Andernfalls verwerfen wir die Parameter und beenden den Algorithmus.

Zusatzaufgabe:

Gib dir mal eine beliebige Primzahl p fest vor. „Programmiere“ nun in Excel den obigen Algorithmus so nach, dass man für beliebige a und c mit $0 \leq a, c < p$ überprüfen kann, ob der Inverse Kongruenz-Generator für die betreffenden Parameter eine maximale Periodenlänge besitzt.

Dieser Algorithmus ist praktisch, da man mit wenig Rechenaufwand schnell testen kann, ob die Periodenlänge maximal ist, ohne den Algorithmus des Inversen Kongruenz-Generators komplett durchlaufen lassen zu müssen. Dies ist für kleine p irrelevant. Unter Umständen kann dies aber bei der Suche nach geeigneten Parametern im Falle sehr großer Primzahlmoduln einen signifikanten Zeitgewinn bringen.

Eine wichtige positive Eigenschaft des Linearen Kongruenz-Generators hätten wir damit schon einmal gerettet. Denn wir wissen genau, wie wir garantieren können, dass die Periodenlänge maximal ist. Okay, es wäre schön, wenn wir auch (neben dem obigen Satz) weitergehende theoretische Kriterien dafür hätten. Solche gibt es auch, aber diese sind an dieser Stelle zu kompliziert. Sie erfordern Erkenntnisse der Höheren Mathematik, die wir uns hier in der Kürze der Zeit leider nicht erarbeiten können. Wir begnügen uns damit, dass wir experimentell mit Hilfe des obigen Algorithmus herausfinden können, wie wir die Parameter für eine maximale Periodenlänge wählen können.

Jetzt gebe ich noch einen Satz an, der den Vorteil des Inversen Kongruenz-Generators heraushebt und damit die erste der obigen beiden Fragen beantwortet. Erinnern wir uns daran: Die Schwäche beim Linearen Kongruenz-Generator bestand darin, dass in der zweidimensionalen Darstellung von Paaren aufeinanderfolgender Zufallszahlen alle Punkte auf parallelen Geraden in der Ebene lagen und entsprechend in der dreidimensionalen Darstellung von Tripeln aufeinanderfolgender Zufallszahlen alle Punkte auf parallelen Ebenen im Raum lagen. Für den Inversen Kongruenz-Generator besteht diese Schwäche nicht. Im Gegenteil, die Punkte meiden sogar diese Strukturen, so wie es auch echte Zufallszahlen tun würden!

Satz 3.2 (Eichenauer-Herrmann, 1991) *Gegeben sei ein Inverser Kongruenz-Generator mit maximaler Periodenlänge. Für ein beliebiges $k \geq 2$ betrachten wir k aufeinanderfolgende Zufallszahlen dieses Generators und fassen diese als Punkte im \mathbb{R}^k auf. Dann enthält keine Hyperebene im \mathbb{R}^k mehr als k Punkte.*

Sprich: In der zweidimensionalen Darstellung enthält keine Gerade mehr als zwei Punkte, in der dreidimensionalen Darstellung enthält keine Ebene mehr als drei Punkte, etc.

Toll! Das ist ein Verhalten, was man sich von Pseudo-Zufallszahlen wünscht!

Aufgabe 5:

Erzeuge mit dem bereits programmierten Inversen Kongruenz-Generator 2500 Zufallszahlen mit den Parametern $a = 66$, $p = 2027$ und $c = 1$. Lasse dir die Zufallszahlen als Paare in der Ebene darstellen, wie wir das beim Linearen Kongruenz-Generator gemacht haben. Was fällt dir auf? Bestätigen sich die obigen Ergebnisse auch optisch?