Erste Schritte mit PARI/GP

Lars Fischer

April 2007

Inhaltsverzeichnis

Einleitung

Auf der PARI/GP Homepage (`http://pari.math.u-bordeaux.fr/`) sind unter Documentation einige Dokumente verfügbar. Insbesondere das Tutorial und der ausführliche „User's Guide“ sind empfehlenswert. Als Gedächtnisstütze kann die „Reference Card“ dienen. Allerdings gehen diese Dokumente auf den vollen Funktionsumfang von PARI/GP ein. Das heißt, sie behandeln Objekte, die den Umfang einer elementaren Zahlentheorie Vorlesung sprengen würden. In dem vorliegenden Dokument möchte ich – nach einer kurzen Installations-Beschreibung– einige erste Schritte mit PARI/GP vorstellen.

1 Installationsanleitung

1.1 Unter Windows

1. Ladet euch die selbstinstallierende Binär-Distribution für Windows herunter. Das ist der vierte Link auf `http://pari.math.u-bordeaux.fr/download.html` (der PARI Download Seite). Ihr könnt zwischen der stabilen (stable) und der Entwicklerversion (unstable) wählen.
2. Ein Doppelklick auf die heruntergeladene Datei (Anfang April ist das die PARI-2-3-2.exe) startet die Installation.
3. Danach erscheint ein neues Icon auf eurem Desktop, mit diesem könnt ihr eine interaktive PARI-Sitzung starten.

1.2 Unter Linux

1.2.1 Mit der Paketverwaltung

PARI/GP ist z.B. unter Ubuntu in den universe Paketquellen enthalten. Dann reicht ein `sudo apt-get install pari-gp` für die Installation aus.

1.2.2 Aus den Quellen

Alternativ könnt ihr PARI auch aus dem Quellcode installieren. Damit habt ihr meistens eine etwas aktuellere Version zur Verfügung. Außerdem ist das gar nicht schwer:

1. Ladet die Source-Distribution der stable-version herunter (der erste Link auf `http://pari.math.u-bordeaux.fr/download.html` (der PARI Download Seite).
2. Diese Datei entpackt ihr in ein Verzeichnis.
3. Bevor ihr das Programm übersetzten könnt, müsst ihr die Entwicklerpakete eurer Distribution installieren. Unter Ubuntu geht das mit `sudo apt-get install build-essential`.
4. Unter Umständen müsst ihr noch die Entwicklerdateien von ncurses (unter Ubuntu: `sudo apt-get install libncurses5-dev`) und readline (unter Ubuntu: `sudo apt-get install libreadline5-dev`) installieren.
5. Dann ruft ihr, in dem Verzeichnis, welches ihr aus dem Archiv entpackt habt, den Befehl `./Configure` auf.
6. Wenn keine Fehlermeldung auftrat, könnt ihr nun `make all` und dann `sudo make install` aufrufen.

Unter Linux wird PARI/GP gestartet, indem ihr `gp` in einem Terminal aufruft.

2 Die Sprache

Wenn ihr PARI/GP installiert habt, so findet ihr unter Windows im Unterverzeichnis `doc`, also z.B. in `c:\Programme\Pari\doc`, die Dokumentation zu PARI/GP. Insbesondere im `users.pdf` stehen wertvolle Hinweise zu Benutzung. Hierbei ist das Kapitel 3.11 Programming in GP hervorzuheben. In dem Kapitel werden alle Kontrollstrukturen beschrieben, die ich im Folgenden, anhand von Beispielen, vorstellen werden. Ebenso kann ich das Kapitel 2 empfehlen, in diesem Kapitel werden u.a. alle Operatoren vorgestellt. In Kapitel 2.6 wird auf die Erstellung von benutzerdefinierten Funktionen und die Problematik von lokalen und globalen Variablen eingegangen.

Nach diesem Hinweis auf die Dokumentation wollen wir PARI/GP starten: Nachdem ihr PARI/GP gestartet habt, könnt ihr an der Eingabeaufforderung (das Fragezeichen) eure Befehle eingeben.

2.1 Bildschirmausgabe

Mit dem `print` Befehl wird etwas auf den Bildschirm geschrieben, z.B. `print("Hallo Welt")`. Nach der Ausgabe erfolgt automatisch ein Zeilenumbruch. Der Befehl `print1` unterlässt diesen automatischen Zeilenumbruch. Z.B.:

```
? print("Hallo");print(" Welt")
Hallo
 Welt
? print1("Hallo");print(" Welt")
Hallo Welt
```

Es können auch mehrere Dinge mit einem print-Befehl ausgegeben werden, dazu werden diese Dinge mit Komma hintereinander geschrieben. PARI/GP fügt keine Leerzeichen ein, diese müssen selber hinzugefügt werden. Zeichenketten erfordern die doppelten Anführungszeichen, Zahlen können direkt angegeben werden.

```
? print("Ding 1 ", 2," Drei");
Ding 1 2 Drei
?
```

2.2 Variablen und Konstanten

Variablen wird ein Wert zugewiesen, danach existieren sie. Es ist zu beachten, dass es ganze Zahlen und Fließkommazahlen gibt. Variablennamen sind case-sensitive, d.h. die Groß- und Kleinschreibung macht einen Unterschied.

```
? a=2
```

```
%7 = 2
? b=2.0
%8 = 2.0000000000000000000000000000
? 3*a
%9 = 6
? 3*b
%10 = 6.0000000000000000000000000000
? a/5
%11 = 2/5
? b/5
%12 = 0.4000000000000000000000000000
? A=7
%13 = 7
? print (a," ", A)
2 7
```

Die wichtigsten vordefinierten Konstanten sind `I` und `Pi`:

```
print(I,"  ",I^2,"  ",Pi)
I  -1  3.1415926535897932384626433832
```

Die üblichen Operatoren sind vorhanden: `+,-,*,/,%`, wobei `%` der Rest bei der Division ist.

```
? 17%5
%22 = 2
? 17/5
%23 = 17/5
? 17/5.0
%24 = 3.4000000000000000000000000000
```

Mit `Mod(x,p)` wird eine Restklasse modulo p erstellt. PARI/GP unterstützt das Rechnen mit den Restlassen direkt.

```
? a=Mod(5,13);
? print(a," ", 2*a," ", 3*a," ",4*a)
Mod(5, 13) Mod(10, 13) Mod(2, 13) Mod(7, 13)
```

2.3 Kontrollstrukturen

Es gibt unter PARI/GP die gängigsten Kontrollstrukturen für if-Abfragen und Schleifen. Etwas ungewöhnlich ist die kompakte Syntax, die ohne viele Schlüsselworte auskommt. Es gibt z.B. kein `else` Schlüsselwort. Stattdessen ist das `if` eine Funktion, die drei

Parameter hat. Eine Bedingung, eine erste Anweisungsfolge, die abgearbeitet wird, falls die Bedingung erfüllt ist, sowie eine zweite Anweisungsfolge, die abgearbeitet wird, falls die Bedingung nicht erfüllt ist. Die zweite Anweisungsfolge ist optional. Hier sind ein paar Beispiele:

```
? a=1
%25 = 1
? if(a==1, print("a ist eins"))
a ist eins
? if(a==1, print("a ist eins"), print("a ist was anderes"))
a ist eins
? a=2
%26 = 2
? if(a==1, print("a ist eins"), print("a ist was anderes"))
a ist was anderes
```

Etwas verwirrend ist die Verwendung von Komma und Semikolon. Wenn man mehrere Anweisungen hintereinander und auf einmal ausführen möchte, so werden diese Anweisungen jeweils durch ein Semikolon voneinander abgetrennt. Andererseits werden bei dem `if` die verschiedenen Parameter durch Komma voneinander getrennt. Die Parameter Nummer Zwei und Drei sind Anweisungsfolgen. Wenn sie mehr als eine einzelne Anweisung lang sind, so enthalten sie Semikolons. Ein Beispiel:

```
? a=36
%31 = 36
{
   if( isprime(a),           \\ Bedingung
       print("a ist prim"),  \\ erste Anweisungsfolge, nur eine Anweisung
       print("a ist keine Primzahl");    \\ zweite Anweisungsfolge,
       print("Die Faktorisierung ist:"); \\ drei Anweisungen, getrennt
       factor(a);                        \\ durch ;
     )
}
a ist keine Primzahl
Die Faktorisierung ist:
%35 =
[2 2]
[3 2]
```

In dem vorigen Beispiel wurden Zeilenumbrüche eingefügt. PARI/GP erfordert dann die äußeren geschweiften Klammern. Zum besseren Verständnis wurden die Zeilen zusätzlich mit Kommentaren versehen.

Die nächste wichtige Kontrollstruktur ist die `for` Schleife:

```
? for(i=1,5,if(i%2==0,print("Die Zahl ist gerade."),print(i," ist ungerade.")))
1 ist ungerade.
Die Zahl ist gerade.
3 ist ungerade.
Die Zahl ist gerade.
5 ist ungerade.
```

Während der Schleife nimmt `i` nacheinander die Werte von 1 bis 5 an. Ähnlich funktioniert die `forprime` und die `fordiv` Schleife, die über die Primzahlen zwischen den angegebenen Zahlen bzw. die Teiler der angegebenen Zahl läuft.

```
forprime(p=1,10,print1(p,", "))
2, 3, 5, 7,
fordiv(28,k,print1(k,", "))
1, 2, 4, 7, 14, 28,
```

Neben der `for` Schleife stehen die `while` und `until` Schleife zur Verfügung. Diese führen eine Anweisung aus, solange wie, bzw. bis eine Bedingung erfüllt ist.

```
? x=20
%42 = 20
? while(!isprime(x), print(x);x=x+1)
20
21
22
? x=x+1
%43 = 24
? until(isprime(x), print(x);x=x+1)
24
25
26
27
28
```

In der obigen `while` Schleife wird die Bedingung `isprime(x)` durch das vorangestellte `!` negiert. Das Ausrufezeichen ist das logische Nicht. Ein nachgestelltes `!` ist hingegen das Zeichen zur Berechnung der Fakultät:

```
? 5!
%44 = 120
```

Es stehen verschiedene Funktionen zur Summenbildung zur Verfügung: `sum` berechnet die Summe eines Ausdrucks, während `sumdiv` die Summe der Teiler einer Zahl berechnet:

```
? sum(i=1,100,i)
%40 = 5050
? sumdiv(28,k,k)
%41 = 56
```

2.4 Funktionen

Ihr könnt unter PARI/GP auch eigenen Funktionen definieren. Dazu gebt ihr einen Namen und eine Liste der Parameter an. Innerhalb der Funktion könnt mittels `return` die Funktion verlassen und einen Wert zurückgeben.

```
test(a,b)=
{
  print("a= ",a,"; b= ",b);
  return(a+b);
}
? test(2,3)
a= 2; b= 3
%48 = 5
```

Hier wurde eine Funktion erstellt, die zwei Parameter entgegennimmt, etwas auf den Bildschirm schreibt und dann die Summe zurückgibt.

```
test(a,b)=
{
  print(a," ",b);
  c="nutzlos";
  return([a,b]);
}
? print(c)
c
? d=test(2,3)
2 3
%1 = [2, 3]
? c
%2 = "nutzlos"
? d[2]
%3 = 3
```

Dieses Beispiel sieht ganz ähnlich aus, wie das vorige. Es gibt aber zwei Unterschiede. Zum einen gibt die Funktion einen Vektor zurück. Damit hat man die Möglichkeit mehr als einen einzelnen Wert als Funktionsergebnis zurückzugeben. Zum anderen wird in

der Funktion eine weitere Variable `c` eingeführt. Diese Variable ist global! Sie existiert nach dem Aufruf von `test` in dem globalen Kontext. Vor dem Aufruf von `test` ist sie undefiniert (`print(c)` liefert nur `c`, keinen Wert). Nach dem Aufruf von `test` hat `c` plötzlich einen Wert.

Das kann zu unerwarteten Seiteneffekten führen, wenn die Variable `c` vorher einen Wert gehabt hätte, dann wäre er nach dem Aufruf von `test` überschrieben worden. Deswegen gehört es auch unter PARI/GP zum guten Stil lokale Variablen (das sind solche die nur innerhalb der Funktion gültig sein sollen) zu benutzen. Als Faustregel kann sogar gelten, dass jede Variable, die eine Funktion nutzt lokal und nur in Ausnahmefällen global ist. Dazu dient die Funktion `local`. Hier noch mal unser Beispiel, diesmal mit lokaler Variable `c`. Vor der erneuten Definition von `test` wird die globale Variable `c` gelöscht. Nach dem Aufruf von `test` ist sie immer noch undefiniert.

```
kill(c)
? test(a,b)=
{
  local(c);

  print(a," ",b);
  c="nutzlos";
  return([a,b]);
}
? print(c)
c
```

Um den Unterschied zwischen lokalen und globalen Variablen genauer zu untersuchen betrachten wir das Beispiel `mysigma()` von dem ersten Übungsblatt in zwei Versionen. Die ersten Version ist korrekt und benutzt eine lokale Variable `pf`. Die zweite Varianten benutzt keine lokale Variable, das führt zu ungewollten Seiteneffekten:

```
mysigma(n)=
{
    local(pf, ergebnis); \\ pf ist lokal

    if(n==1,
        print1("-> mysigma( n=", n, ") ");
        return(1);
    );

    pf = factor(n); \\ Primfaktoren von n

    print1("-> mysigma( n=", n, ", pf_vorher =", pf );
    ergebnis = mysigma(n/(pf[1,1]^pf[1,2] ))*(pf[1,2] + 1);
    print1(", pf_nachher =", pf" ) ");
    return(ergebnis);
}


mysigma_Kaputt(n)=
{
    local(ergebnis); \\ pf ist hier global!

    if(n==1,
        print1("-> mysigma( n=", n, ") ");
        return(1);
    );

    pf = factor(n); \\ Primfaktoren von n

    print1("-> mysigma( n=", n, ", pf_vorher =", pf );
    ergebnis = mysigma_Kaputt(n/(pf[1,1]^pf[1,2] ))*(pf[1,2] + 1);
    print1(", pf_nachher =", pf" ) ");
    \\ der vorige aufruf von mysigma_Kaputt() ändert pf, deswegen
    \\ wird mit dem falschen (pf[1,2] +1)
    \\ multipliziert und die Berechnung verfälscht
    return(ergebnis);
}

testWert=12;
erg1=mysigma(testWert);
print(erg1);
print(" ...  und nun die kaputte Variante:");
erg2=mysigma_Kaputt(testWert);
print(erg2);
```

Die Ausgabe (ich habe sie von Hand eingerückt) verrät, was schief geht:

```
-> mysigma( n=12, pf_vorher=[2, 2; 3, 1]
  -> mysigma( n=3, pf_vorher=Mat([3, 1])
    -> mysigma( n=1) ,
  pf_nachher=Mat([3, 1]) ) ,
pf_nachher=[2, 2; 3, 1] )
6
 ...  und nun die kaputte Variante:
-> mysigma( n=12, pf_vorher=[2, 2; 3, 1]
  -> mysigma( n=3, pf_vorher=Mat([3, 1])
    -> mysigma( n=1) ,
  pf_nachher=Mat([3, 1]) ) ,
pf_nachher=Mat([3, 1]) )
4
```

Der Fehler wird deutlich, wenn man die beiden letzten Zeilen der Ausgaben vergleicht. Die kaputte Variante verändert das `pf`. In der kaputten Varianten existiert ja nur ein einziges globales `pf`, welches bei jedem Aufruf verändert wird. In der anderen Variante hat jede Instanz von `mysigma()` ihr eigenes individuelles (lokales) `pf`, welches durch den rekursiven Aufruf nicht verändert wird.

2.5 Rekursion

Die Funktion `mysigma` aus dem vorigen Abschnitt ist ein Beispiel für eine rekursive Funktion. Solche Funktionen zeichnen sich dadurch aus, dass sie sich selbst aufrufen.

Beispiele für rekursive Definitionen oder Algorithmen sind:

- Fakultät: $n! = n(n-1)!, \quad 1! = 1$
- Fibonacci-Zahlen: $fib(n) = fib(n-1) + fib(n-2), \quad fib(0) = 0, fib(1) = 1$
- quicksort, bzw. die ganze Klasse der `divide & conquer` Algorithmen
- Fraktale, wie der Pythagoras-Baum oder die Koch-Kurve

Eine rekursive Funktion braucht eine Abbruchsbedingung und man sollte sich vorher überlegen, ob diese Abbruchbedingung auch eintritt. Ansonsten ruft sich die Funktion ewig weiter auf. Unter Umständen wird sie irgendwann von PARI/GP abgebrochen.

Rekursive Funktionen sind immer da angebracht, wo eine Problem oder eine Datenstruktur rekursiv definiert ist. Doch oft sind iterative Algorithmen oft schneller, da durch jeden Funktionsaufruf etwas Overhead entsteht.

3 Hilfe

PARI/GP verfügt über eine eingebaute Hilfefunktion. Wenn ihr wissen wollt, was eine bestimmte Funktion tut, so könnt ihr einfach ein Fragezeichen davorschreiben und ihr erhaltet eine Beschreibung der Funktion und ihrer Parameter:

```
?for
for(X=a,b,seq): the sequence is evaluated, X going from a up to b.
```

Es gibt noch eine ausführlichere Online Hilfe: wenn ihr zwei Fragezeichen voranstellt(`??for`), wird der entsprechende Abschnitt aus der PDF-Dokumentation angezeigt. Unter Linux geht das ohne Probleme. In der README.win32 steht drin, dass es unter manchen Windows-Systemen nicht geht. Bei mir z.B. geht es nicht.

Eine weitere nützliche Funktion ist die kontext-sensitive Ergänzung von Ausdrücken, also ihr schreibt `for` hin und drückt die Tast TAB. Dann versucht PARI/GP das Symbol zu kompletieren. Wenn es nur eine möglich Ergänzung gibt, wird diese hingeschrieben. Gibt es mehrere, so werden diese unter Linux ebenfalls angezeigt. Das geht unter Windows (bei mir) nicht, dort passiert leider gar nicht. Was aber unter Windows und Linux geht ist folgendes: Drückt ihr Alt und ? (also ALT, Shift und ß), so wird eine Liste der möglichen Ergänzungen angezeigt:

```
? for<hier ALT-? drücken>
for           forell       forstep      forvec
fordiv        forprime     forsubgroup
? for
```

Wenn ihr ein einzelnes Fragezeichen eingebt, so wird euch eine Liste mit Kategorien von 0 bis 12 angezeigt und mit `?0`, `?1`, bis `?12` könnt ihr euch die Stichworte in diesen Kategorien anschauen. Z.B. zeigt `?4` die Funktionen aus der Kategorie Zahlentheorie an:

```
?4

addprimes      bestappr       bezout         bezoutres      bigomega
binomial       chinese        content        contfrac       contfracpnqn
core           coredisc       dirdiv         direuler       dirmul
divisors       eulerphi       factor         factorback     factorcantor
factorff       factorial      factorint      factormod      ffinit
fibonacci      gcd            hilbert        isfundamental  ispower
isprime        ispseudoprime  issquare       issquarefree   kronecker
lcm            moebius        nextprime      numbpart       numdiv
omega          precprime      prime          primepi        primes
qfbclassno     qfbcompraw     qfbhclassno    qfbnucomp      qfbnupow
qfbpowraw      qfbprimeform   qfbred         qfbsolve       quadclassunit
```

```
quaddisc      quadgen       quadhilbert   quadpoly      quadray
quadregulator quadunit      removeprimes  sigma         sqrtint
zncoppersmith znlog         znorder       znprimroot    znstar
```

4 Skripte

Wenn eure Berechnungen länger werden und ihr sie nicht jedes Mal neu eingeben wollt, könnt ihr sie in Dateien speichern. Diese Skript-Dateien könnt ihr dann wieder in PARI/GP laden und ausführen lassen.

Dazu legt ihr eine Datei an und schreibt einfach eine Reihe von PARI/GP Anweisungen in diese Datei.

Wir greifen hier nochmal das obige Beispiel auf. Diesmal werden die Zeilen aber nicht interaktiv eingegeben, sondern in einer Datei test.gp gespeichert. Hier ist der Inhalt der Datei:

```
{ \\ nötig wegen Zeilenumbrüchen beim if

  print("Ein kleines Skript.");
  a=15;
  if(isprime(a),
     print("a ist eine Primzahl"),
     print("a ist keine Primzahl");
     print("Die Faktorisierung ist:");
     factor(a);
  );
}
```

Nun könnt ihr die Dateien `test.gp` in der Pari-Sitzung einlesen. Dazu gibt es die Funktion `read` sowie etwas kürzer `\r`. Beide lesen die Datei ein und führen die Anweisungen in ihr aus.

```
? read("test.gp")
Ein kleines Skript.
a ist keine Primzahl
Die Faktorisierung ist:
%7 =
[3 1]

[5 1]

? \r test.gp
Ein kleines Skript.
```

```
a ist keine Primzahl
Die Faktorisierung ist:
```

Wie man sieht, sind beide Varianten nicht ganz gleich. Die Funktion `read` gibt einen Wert zurück: das Ergebnis der letzten Berechnung, welches hier die Faktorisierung von `a` ist. `\r` liefert keine Ergebnis zurück. In dem Skript hätte `print(factor(a));` stehen müssen.

Es gibt noch eine andere Möglichkeit: ihr müsst eine Eingabeaufforderung öffnen, dort wechselt ihr in das Verzeichnis, wo die Datei gespeichert ist und ruft dann PARI/GP in der folgenden Weise auf:

```
gp -f -q < test.gp

Ein kleines Skript.
a ist keine Primzahl
Die Faktorisierung ist:
```

Die erste Zeile ist der Aufruf von PARI/GP. Dazu muss gp in eurem Pfad enthalten sein. (Wer mit Eingabeaufforderung, Pfad, Verzeichniswechsel etc. nichts anfangen kann, liest bitte im nächsten Abschnitt weiter).

Als Editor mit dem ihr die Skripte erstellt eignet sich jeder Editor (nur nicht WORD). Wenn ihr mit einem Programmiereditor vertraut seid, könnt ihr den verwenden. Leider gibt es keinen speziellen Editor für Pari (es gibt für Emacs einen Pari-Mode).

5 SCITE und ein pari-Modus

SCITE ist ein flexibler und plattformunabhängiger Programmiereditor. Er zeichnet sich durch eine Vielzahl unterstützter Programmiersprachen aus. Ich habe einen Pari-Modus zusammengestellt. Wenn ihr diesen installiert, dann

- wird euer Programm auf Knopfdruck ausgeführt
- es gibt farblicher Hervorhebungen für die verschiedenen Syntax-Elemente von PARI/GP
- Autovervollständigung

Nähere Informationen findet ihr auf der Übungs-Homepage.