SAGE-Tutorium 06 im SoSe 2009

Lars Fischer*

03.06.2009

Inhaltsverzeichnis

*WWW: `http://w3.countnumber.de/fischer`, EMail: vorname.nachname (bei der) uni-siegen.de

1 Wiederholung

- Polynome und Polynomringe
- Quadratische Reziprozität

2 Nachtrag zu Dingen, die ich gesehen habe

2.1 Variablen

Ich habe folgendes gesehen:

```
x=var("x")
x=0
```

Dazu ist zu sagen:

- SAGE/Python sind dynamische Programmiersprachen. Variablen werden durch Zuweisungen erstellt. Wenn es `x` vorher nicht gab, dann existiert `x` nach der Ausführung von `x=0`.
- `x=var("x")` ist keine Variablendefinition im Sinne von C, Java oder Pascal. Es sorgt nur dafür, das `x` vom Typ Symbolischer Ausdruck ist. Ausdrücke mit `x`, z.B. `5*x`, sind dann ebenfalls vom Typ Symbolischer Ausdruck.
- Symbolische Ausdrücke sind langsam und sollten deswegen vermieden werden.
- Oben wird `x` zuerst als Symbolische Variable angelegt und direkt danach als Ganze Zahl (durch das `=0`), damit wird die Wirkung der ersten Anweisung sofort aufgehoben.

  ```
  x=var("x")
  type(x)
  x=0
  type(x)
  ```

2.2 Funktionen

Es reicht wirklich eine Funktion **einmal** anzulegen. Man muss sie nicht in jedem Schleifendurchlauf erneut anlegen:

```
for j in range (1,1001):
    x=var('x') # in jedem Schleifendurchlauf
    a=var('a') # dto.
    x=0        # hier wird die var Anweisung aufgehoben
    a=j        # dto.
    # Nun wird in jedem Schleifendurchlauf die Funktion erneut
    # definiert. Das ist ca. 999-Mal überflüssig.
    def divisible(number, divisor=1):
        if number%divisor == 0:
            return divisor
        if number%divisor>0:
            return 0
   # .....
```

3 Optimierung und Beschleunigung von Python/SAGE

Wenn eine Funktion oder ein Algorithmus zu langsam ist, dann haben wir mehrere Möglichkeiten:

1. Wir finden ein anderes Verfahren oder einen anderen Algorithmus.
2. Unter Umständen können wir aber auch unser langsames Verfahren an einigen Stellen entscheidend beschleunigen. Dazu müssen wir die Code-Abschnitte finden, in denen die meiste Zeit verschwendet wird. Es lohnt sich meistens nur, die innersten Blöcke von Schleifen zu optimieren. Diese werden am häufigsten durchlaufen und eine Optimierung an dieser Stelle bringt am meisten.

Wir betrachten wieder das Beispiel aus der vorletzten Woche, das in vielerlei Hinsicht verbesserungswürdig ist (siehe Epilog). Aber nehmen wir für den Augenblick an, dass wir keine bessere Implementierung finden können, um an diesem Beispiel das Vorgehen und die Möglichkeiten zu untersuchen.

```
def filterPrimes(N=10000):
    Zahlen = range(N)
    Kopie = Zahlen[:]

    i=0
```

```
    while i < len(Zahlen):
        z= Zahlen[i]
        if not is_prime(z):
            del Kopie[ Kopie.index( z ) ]
        i+=1

    return Kopie

print filterPrimes(100)



%time
print len( filterPrimes( 10000 ) )
```

3.1 Cython

Da SAGE/Python eine interpretierte Sprache ist, liegt hier unser erster Verdacht, warum das so lange dauert. Die Tatsache, dass SAGE/Python interpretierte Sprachen sind, hat für den Entwicklungsprozess viele Vorteile. Ein gravierender Nachteil ist, dass interpretierte Sprachen prinzipiell langsamer sind als kompilierte Sprachen. (Hinter den Kulissen wird dem Entwickler viel Arbeit abgenommen.)

Fürs Number-Crunching ist man hingegen auf schnellen Code angewiesen. Das ist ebenfalls mit SAGE möglich. Und zwar können wir unseren Algorithmus ersteinmal in SAGE entwickeln und testen. Wenn wir (wie hier) feststellen, dass es zu langsam ist, so können wir immer noch zu einer kompilierten Variante unseres Algorithmus übergehen.

Unter SAGE müssen wir dazu nicht einmal alles neuschreiben, denn zu einer SAGE-Installation gehört bereits Cython (für eine Python-Umgebung kann man Cython nachinstallieren, deswegen gilt das hier Gesagte auch für Python).

In einer ersten Näherung ist die Sprache Cython »Python mit der Möglichkeit Typen festzulegen«.

Desweiteren erstellt Cython automatisch eine C-Version unserer Funktion und fügt genug Schnittstellen-Code hinzu, so dass wir die kompilierte C-Version unserer Funktion benutzen können. (Unbedingt einen Blick auf die spyx-Datei und vor allem auch eine Blick auf die c-Datei werfen! Beide könnt Ihr nach der Auswertung der Cython-Zellen als Link anklicken.)

Damit eignet sich Cython hervorragend, um einzelne Python-Funktionen in eine kompilierte (und damit schnellere) Bibliothek auszulagern.

```
%cython

from sage.all import is_prime

def filterPrimesCythonV1(N=10000):
    Zahlen = range(N)
    Kopie = Zahlen[:]

    i=0
    while i < len(Zahlen):
        z= Zahlen[i]
        if not is_prime(z):
            del Kopie[ Kopie.index( z ) ]
        i+=1

    return Kopie


%time
print "Python:"
print len( filterPrimes( 10000 ) )



%time
print "CythonV1:"
print len( filterPrimesCythonV1( 10000 ) )
```

Wir haben im Prinzip nur `%cython` in die erste Zeile der Zelle geschrieben, der Rest passiert hinter den Kulissen. Wir können sogar noch etwas Geschwindigkeit gewinnen, indem wir die Variablen mit ihren Typen deklarieren.

```
%cython

from sage.all import is_prime

def filterPrimesCythonV2(int N=10000):
    cdef int i=0, z
    cdef list Zahlen, Kopie
    Zahlen = range(N)
```

```
    Kopie = Zahlen[:]

    i=0
    while i < len(Zahlen):
        z= Zahlen[i]
        if not is_prime(z):
            del Kopie[ Kopie.index( z ) ]
        i+=1

    return Kopie


%time
print "CythonV2:"
print len( filterPrimesCythonV2( 10000 ) )
```

Fazit: Wie wir gleich sehen werden, hatte Cython überhaupt keine großen Chancen mehr Geschwindigkeit herauszuholen.

3.2 Profiling und Optimierung

Nachdem selbst mit Cython keine wirkliche Geschwindigkeitsteigerung möglich ist, sollte langsam aber sicher der Verdacht aufkeimen, das die Implementierung Mist ist.

Am Anfang aller Optimierungsbemühungen sollte man sich einen Überblick verschaffen, an welcher Stelle es sich überhaupt lohnt. Wir haben oben den zweiten Schritt vor dem ersten gemacht und ohne eingehende Analyse einfach losgelegt.

»Premature optimization is the root of all evil.« – Donald Knuth

Wie in der Einleitung erwähnt, lohnen sich Optimierungen nur an der innersten Stellen, die am häufigsten durchlaufen wird.

Die innerste Stelle unserer Funktion ist
`if not is_prime(z): del Kopie[ Kopie.index( z ) ]`.
Dabei sind `is_prime, del` und `index` Funktionen die zum Kern von SAGE/Python gehören und bereits über eine schnelle Implementierung in C verfügen. Sowohl die SAGE/Python als auch die Cython Varianten rufen diese auf. Durch den Übergang zu Cython haben wir nur das Drumherum um diese innerste Stelle beschleunigt.

Wir verschaffen uns nun Gewissheit, indem wir unsere Funktion mit einem Profiler analysieren:

```
import profile
profile.run("filterPrimes(100)")
#profile.run("filterPrimes(1000)")
#profile.run("filterPrimes(10000)")
```

Je größer `N` ist, umso mehr Zeit wird in der `index` Funktion vergeudet. (Gar nicht mal in `is_prime`!) Wir sollten eine Implementierung anstreben, die uns dieses Index-Sucherei erspart.

3.3 Epilog

Warum löschen wir die zusammengesetzten Zahlen? Wir können sie ebensogut markieren und hinterher die übrigen Zahlen einsammeln.

```
# keine Cython-Magie, diese Version gilt es immer noch zu schlagen:
def filterPrimesV2(N= 10000):
    Zahlen = [z for z in range(N) ]
    for i in range( len( Zahlen ) ):
        if not is_prime( Zahlen[i]):
            Zahlen[i] = False
    return [z for z in Zahlen if z]

# das ist nur unwesentlich besser:
def filterPrimesV3(N= 10000):
    return [z for z in range(N) if is_prime(z) ]

# und eigentlich gibt es bereits eine Funktion, die die Primzahlen auflistet:
def filterPrimesV4(N=10000):
    return prime_range(N)


%time

print "Optimierte Python Version:"
print len(filterPrimesV2(10000))
#print len(filterPrimesV3(10000))
#print len(filterPrimesV4(10000))
```

3.4 Vorgehen bei Optimierung

1. Am Anfang sollte ein korrektes Verfahren stehen.
2. Durch Tests sollte sichergestellte sein, dass das Verfahren wirklich korrekt ist. (Es hat keine Sinn eine Version zu optimieren, die sich fehlerbedingt noch verändert.)
3. Falls das Verfahren zu langsam ist: mit einem Profiler oder auf eine andere Art die Stelle bestimmen, die die meiste Zeit benötigt.
4. Dort ändern.
5. Zurück zu 2.

Optimierungen an den inneren Stellen bringen den meisten Gewinn. Wenn es, nach allen Optimierungen, immer noch zu langsam ist, könnte man Cython probieren.

Bevor man viel Zeit mit Optimierungen verschwendet: Kann man das Problem auf eine komplett andere Weise lösen?

3.5 Zwei Beispiele, wo Cython mehr Chancen hat

3.5.1 Beispiel Summierung

```
def py_sum(N):
    s=0 # probieren: int( 0 )
    for i in range(N):
       s+=i

    return s



%time
py_sum(10^6)
```

Nun in Cython. (Beachtet, dass die Variablen als long long definiert sind. Auf meinem 32Bit-Rechner ist das notwendig, mit einem einfachen long ist das Ergebnis zu falsch.)

```
%cython

def cy_sum(long N):
    cdef long long s = 0
    cdef long long i
```

```
    for i from 0<= i < N:
       s+=i

    return s


%time

cy_sum(10**6)
```

3.5.2 Beispiel Probedivision

```
def ProbeDivision_is_prime( n ):
    "Testet durch Probedivision bis Wurzel n, ob n prim ist."

    if n % 2 == 0: return False
    d= 3 # nur noch ungerade, d+=2 testen
    w= sqrt(n)
    while d <= w:
        if n%d == 0: return False
        d+=2

    return True


%time

for n in range(50):
    ProbeDivision_is_prime(n)
print "fertig"


%cython
# es ist nur obige Zeile geaendert worden
def ProbeDivision_is_prime_CythonV1( n ):
    "Testet durch Probedivision bis Wurzel n, ob n prim ist"

    if n % 2 == 0: return False
    d= 3 # nur noch ungerade, d+=2 testen
    w= sqrt(n)
```

```
    while d <= w:
        if n%d == 0: return False
        d+=2

    return True


%time

for n in range(300000):
    ProbeDivision_is_prime_CythonV1(n)

print "fertig"



%cython

# jetzt wurden noch Typen angegeben:
def ProbeDivision_is_prime_CythonV2( int n ):
    "Testet durch Probedivision bis Wurzel n, ob n prim ist"
    cdef int d
    cdef int w= int(sqrt(n))+1

    if n % 2 == 0: return False
    d= 3 # nur noch ungerade, d+=2 testen
    while d < w:
        if n%d == 0: return False
        d+=2
    return True



%time

for n in range(300000):
    ProbeDivision_is_prime_CythonV2(n)

print "fertig"
```

Wenn Ihr SAGE von der Kommandozeile benutzt: `attach file.spyx`. Bei der Erweiterung spyx geht SAGE von einer Cython-Datei aus.

3.6 Vergleich ähnlicher Tools

Unter `http://www.scipy.org/PerformancePython` findet Ihr einen Vergleich verschiedener Tools, die ebenfalls, hinter den Kulissen, kompilierten Code erzeugen. Der Abschnitt über Pyrex entspricht Cython. (Cython ist der Nachfolger von Pyrex.) In Sage ist weave verfügbar via

```
import scipy
scipy.weave?
```

4 SAGE direkt nutzen (ohne Notebook)

Wenn die SAGE-Programme immer länger werden (z.B. die Projekte), dann ist das Notebook unter Umständen nicht mehr die richtige Umgebung. Ein Editor mit Syntax-Färbung für Python erleichtert die Eingabe ungemein. Außerdem braucht Ihr dann nicht mehr unseren Server zu benutzen, der anscheinend nicht zu gebrauchen ist.

4.1 Vorgehen

- Eine SAGE-Datei `example.sage` in einem Editor schreiben, ein Editor mit Python-Unterstützung ist ungemein hilfreich.
- sage eine einem Terminal starten
- dort `attach example.sage`
- Nach jeder Änderung, im Editor speichern, dann am SAGE-Prompt eine leere Zeile eingeben (ENTER drücken).

Vorteile

- Ihr müsst nicht das Notebook (und damit unseren Server) benutzen.
- Ihr startet Euer eigenes SAGE.
- Syntax-Hervorhebungen in dem Editor (falls dieser Python unterstützt).

Nachteile

- Ihr benutzt nicht das Notebook.
- Ihr müsst euer eigenes SAGE starten. Dazu habt Ihr verschieden Möglichkeiten:
  1. Lokale Installation auf Eurem eigenen Computer.

2. Zugriff per SSH auf euren CIP-Pool Zugang. Unter Windows gibt es dazu PuTTY (`http://www.chiark.greenend.org.uk/~sgtatham/putty/`).
3. Ihr arbeitet direkt im CIP-Pool.

Probieren, ob 3D-Grafik von der Konsole aus funktioniert. So wird das Java-Browser-Plugin umgangen, man braucht nur ein installiertes JRE (von SUN).

5 Fragen zur Vorlesung?

5.1 Bestimmung einer Lösung x zu $x^2 \equiv a \pmod{p}$

Angenommen es gelte $\left(\frac{a}{p}\right) = 1$. Wie findet man ein x mit $x^2 \equiv a \pmod{p}$?

Fallunterscheidung:

1. Sei $p \equiv 3 \pmod{4}$.

   Dann gibt es eine einfache Antwort. Es ist nämlich $b := a^{(p+1)/4}$ eine Wurzel von a, da dann $b^2 = a^{(p+1)/2} = a^{(p-1)/2+1} = a^{(p-1)/2}a = a$. (Im Fall, dass a ein quadratischer Rest ist, gilt $a^{(p-1)/2} = 1$.)

2. Sei $p \equiv 1 \pmod{4}$.

   Dann gibt es keinen deterministischen Algorithmus. Allerdings gibt es ein randomisiertes Verfahren zur Bestimmung einer Wurzel. (Das findet man z.B. in dem Buch »Elementary Number Theory: Primes, Congruences and Secrets« von William Stein.)

Ein Algorithmus für beide Fälle:

```
def findSqrt(a , p):
    """Bestimmt ein b mit $b^2 \equiv a \pmod{p}.

    INPUT:
        a -- ganze Zahl mit legendre_symbol(a,p) == 1
        p -- ungerade Primzahl
    OUTPUT:
        b - Restklasse mod p mit b^2 == Mod(a,p)

    EXAMPLES:
        sage: L= [ (3,13), (5,389), (17,103)]
        sage: for a,p in L: print findSqrt(a,p)^2 == a
        True
        True
```

```
        True
    """

    if legendre_symbol(a,p) == -1:
        raise ValueError, "%d ist kein quadratischer Rest mod %d"%(a,p)

    if Mod(p,4) == 3:
        return Mod(a,p)^((p+1)/4)
    else:
        R= IntegerModRing(p)
        P.<x>=PolynomialRing( R )
        I.<alpha> = P.quotient( x^2 - a)
        # I= ((Z/pZ)[x])/(x^2 - a)
        # alpha erfuellt die Relation alpha^2 = a, bzw. alpha^2-a = 0

        while True:
            z = R.random_element() # hier sind wir nicht mehr deterministisch
            w = (1+ z*alpha)^((p-1)/2)
            (u, v) = (w[0], w[1])
            if v != 0: break

        if (-u/v)^2      == a: b=  -u/v
        if ((1-u)/v)^2   == a: b= (1-u)/v
        if ((-1-u)/v)^2  == a: b= (-1-u)/v

        return Mod(b,p)
```

5.2 Jacobi-Symbol, bzw. verallgemeinertes Legendre-Symbol

Rechenregeln fuer das Jacobi-Symbol

Es seien Q, Q' ungerade und positiv, dann haben wir für das Jacobi-Symbol:

- $\left(\frac{P}{Q}\right)\left(\frac{P}{Q'}\right) = \left(\frac{P}{QQ'}\right)$
- $\left(\frac{P}{Q}\right)\left(\frac{P'}{Q}\right) = \left(\frac{PP'}{Q}\right)$
- falls P, Q teilerfremd sind: $\left(\frac{P^2}{Q}\right) = \left(\frac{P}{Q^2}\right) = 1$
- falls PP', QQ' teilerfremd sind: $\left(\frac{P'P^2}{Q'Q^2}\right) = \left(\frac{P'}{Q'}\right)$
- $P' \equiv P \pmod{Q} \Rightarrow \left(\frac{P'}{Q}\right) = \left(\frac{P}{Q}\right)$

Für bestimmte Wert von P und ungerades positive Q haben wir noch:

- $\left(\frac{1}{Q}\right) = 1$
- $\left(\frac{0}{Q}\right) = 0$
- $\left(\frac{-1}{Q}\right) = (-1)^{(Q-1)/2}$
- $\left(\frac{2}{Q}\right) = (-1)^{(Q^2-1)/8}$

Hier kann man quadratische Reziprozität ausnutzen, sobald a ungerade und positiv ist. (Beim Legendre-Symbol muss a eine ungerade Primzahl sein, bevor man quadratische Reziprozität ausnutzen kann.)

Das quadratische Reziprozitäts-Gesetz für das Jacobi-Symbol lautet: Es seien P, Q verschiedene ungerade positive teilerfremde Zahlen, dann gilt:

$$\left(\frac{P}{Q}\right) = (-1)^{(P-1)(Q-1)/4} \left(\frac{Q}{P}\right)$$

Das Jacobi-Symbol kann 1 sein, obwohl der Zähler kein quadratischer Rest ist (die Funktion `kronecker_symbol` berechnet das Jacobi-Symbol):

```
print legendre_symbol(8,3)
print legendre_symbol(8,5)
print kronecker_symbol(8,15) #
```

```
for r in IntegerModRing(15):
    #print r^2
    if r^2 == 8: print r
```

Bei der Definition des Jacobi-Symbols war es wichtiger, die quadratischer Reziprozität zu haben, als dass es nur für quadratische Reste 1 ist.

Wäre das Jacobi-Symbol so definiert worden, dass es nur für quadratische Reste 1 ist, so gäbe es kein Reziprozitätsgesetzt. ($P = 5, Q = 9$ sind dafür ein Beispiele.)

6 Vergabe der Projekte

7 Aufgaben

Aufgabe 1: Schaut Euch nochmal Eure bisherigen Programme an. Versucht ein Programm, das zu langsam war, zu optimieren.

Greift erst ganz zum Schluss zu Cython. Vergleicht die Laufzeiten der zu den verschiedenen Optimierungsschritten. (Hinweise:

- Cython ist ein amerikanisches Programm. Deswegen kommt es natürlich nicht mit Umlauten zurecht, auch nicht mit Umlauten in den Kommentaren.
- Unter Umständen müsst ihr Eure Programme mit `from sage.all import FUNKTION` oder ähnlichen Anweisungen ergänzen.

)

Aufgabe 2: Schreibt eine Funktion

```
def myJacobi(a,q):
 """Berechnet das Jacobi-Symbol a über q.

 INPUT:
     a,q -- Integer und q ungerade
 OUTPUT:
     1,-1,0

 EXAMPLES:
     sage: myJacobi(2, 15)
     1
     sage: myJacobi(105, 317)
     1
 """
```

welche den Wert des Jacobi-Symbols $\left(\frac{a}{q}\right)$ berechnet. Implementiert die Funktion, ohne die SAGE-Funktionen `kronecker_symbol`, bzw. `factor` zu benutzen.

Aufgabe 3: Setzt Euch mit Euren Projekten auseinander und überlegt Euch für nächste Woche, ob etwas unklar ist oder Schwierigkeiten bereitet.

8 Nächstes Mal

- Dateien lesen und schreiben
- Ausnahmebehandlung
- Doc-Testing

9 Quellcode

Das gesamte Worksheet ist als Text-Datei in dem PDF eingebettet.

- Im Acrobat-Reader lässt es sich unter dem Büroklammer-Symbol in der linken Leiste herunterladen.
- Okular zeigt es im File-Menu als Embeded Files an.
- Unter Linux kann man die Text-Datei auch mit
  pdftk Tutorium06.pdf unpack_files
  aus dem PDF herauslösen.

Anschließend lässt sich die Text-Datei mit der Upload-Funktion des SAGE-Notebooks hochladen.