Notiz für den entsprechenden Abschnitt in Stochastik- oder Fourier-Analysis Vorlesungen.

„FFT für W-theoretiker“

Heinrich v. Weizsäcker, Anhang: Jochen Voß

Gegeben seien zwei Wahrscheinlichkeitsverteilungen f, g über $\mathbb{Z}$ mit endlichem Träger. Wir wollen ihre Faltung

$$f * g(k) = \sum_{l \in \mathbb{Z}} f(l)\ g(k-l) \tag{1}$$

berechnen und zwar so, daß wir mit wenig Rechenoperationen auskommen. Die Bedingung $f, g \geq 0$, $\sum f(k) = 1$, $\sum g(l) = 1$ spielt dabei keine Rolle. Zunächst überlegen wir, wie oft man bei der natürlichen Vorgehensweise multiplizieren und addieren muß. Offenbar müssen alle Produkte $f(k)\ g(l)$ gebildet werden mit $f(k) \neq 0$ und $g(l) \neq 0$. Dies sind $r \cdot s$ Produkte, wenn der Träger von f r Elemente hat und der Träger von g s Elemente. Außerdem müssen diese Produkte noch geeignet aufsummiert werden. Wenn wir $n = \max(r, s)$ setzen und annehmen, daß r und s die gleiche Größenordnung haben, erhalten wir salopp gesprochen das Ergebnis, daß bei natürlicher Vorgehensweise der Aufwand der Faltung quadratisch mit dem Umfang der beiden Wahrscheinlichkeitsverteilungen wächst.

Ziel dieser Notiz ist, zu zeigen, wie man diesen Aufwand mit Hilfe der „FFT“ (Fast Fourier Transformation) angewendet auf die charakteristischen Funktionen auf die Größenordnung $O(n \log n)$ reduzieren kann, wobei die beteiligte Konstante in Sichtweite bleibt. Allerdings ist n hierbei die Ausdehnung (= größtes - kleinstes Element) des Trägers und nicht seine Mächtigkeit. Grundidee ist natürlich die für die charakteristischen Funktionen gültige Relation

$$\widehat{f * g}(t) = \hat{f}(t)\ \hat{g}(t)\ . \tag{2}$$

Außerdem geben wir im Anhang noch einige Hinweise zur effektiven Implementierung der FFT.

Im theoretischen Teil wollen wir zweierlei zeigen:

a) Die charakteristischen Funktionen brauchen nur an $2n$ (= Ausdehnung von $f * g$) Stellen berechnet zu werden. (Das ist verwandt mit dem sogenannten „Abtasttheorem“!)

b) Der Übergang zwischen einer Funktion h auf $\{1, \ldots, 2n\}$ und ihrer gemäß a) reduzierten Fouriertransformierten, der ja linear ist und daher durch eine invertierbare $2n \times 2n$ Matrix beschrieben werden kann, läßt sich aufgrund der speziellen Struktur dieser Matrix in beiden Richtungen mit $O(n \log n)$ Operationen durchführen.

Da die Berechnung der Produkte in (2) nur $2n$ Multiplikationen benötigt, steckt also der Hauptaufwand in der „Fouriertransformation“.

Zu a) 1. Durch Verschieben kann man sich tatsächlich auf den Fall beschränken, daß f und g außerhalb von $\{1, \ldots, n\}$ verschwinden: Sind nämlich $a+1$ und $b+1$ die Anfangspunkte der beiden Träger, und ist $f_a(x) = f(x-a)$ und $g_b(x) = g(x-b)$ so gilt mit $l = j + a$

$$f * g(k) = \sum_j f(j) g(k-j) = \sum_l f_a(l) g_b(k-l+b-a) = f_a * g_b(k+b-a).$$

Unter dieser Annahme verschwindet $f * g$ außerhalb von $\{1, \ldots, 2n\}$. Damit können wir uns auf die charakteristische Funktion von Verteilungen beschränken, die auf einem Intervall der

Form $\{1, \cdots, m\}$ konzentriert sind.
2. Nach Definition der charakteristischen Funktion ist

$$\hat{f}(t) = \sum_{k=1}^{m} f(k) e^{ikt}. \tag{3}$$

Wir behaupten, daß f durch die Werte von $\hat{f}$ auf der Menge $\{\frac{1}{m}2\pi, \ldots, \frac{m-1}{m}2\pi, 2\pi\}$ bestimmt ist. Es gilt nämlich für $j = 1, \ldots, m$ die Umkehrformel

$$f(j) = \frac{1}{m} \sum_{l=1}^{m} \hat{f}(\frac{2\pi l}{m}) e^{-ij\frac{2\pi l}{m}}. \tag{4}$$

Eine andere Möglichkeit dies auszusprechen, ist folgende: Betrachte die komplexe $m \times m$-Matrix

$$\mathsf{F}_m = \left(\frac{1}{\sqrt{m}} e^{-\frac{kl}{m}2\pi i} \right)_{1 \le k,l \le m}. \tag{5}$$

Dies ist die Matrix der diskreten Fouriertransformation. Dann gilt die offenbar zu (4) äquivalente Aussage

Lemma 1 *Die Matrix F_m ist unitär, dh. es gilt*

$$\mathsf{F}_m^{-1} = \left(\frac{1}{\sqrt{m}} e^{\frac{kl}{m}2\pi i} \right)_{1 \le k,l \le m}. \tag{6}$$

Beweis: Zum Beweis setzen wir $\gamma(r) = e^{i\ r\frac{2\pi}{m}}$. Wesentlich ist nur folgendes

$$\begin{cases} \gamma(r+s) = \gamma(r)\ \gamma(s) \\ \gamma(0) = \gamma(m) = 1 \\ \gamma(r) \neq 1 \text{ für } r \neq 0 \ mod\ m \end{cases}$$

Wir wollen zeigen, daß $\frac{1}{m}\sum_{l=1}^{m} \gamma(l(k-j)) = \delta_{kj}$. Die Behauptung für $k = j$ ist klar wegen $\gamma(l0) = \gamma(0) = 1$ für $l = 1, \ldots, m$. Für die Aussage im Fall $k \neq j$ zeigen wir $\sum_{l=1}^{m} \gamma(l\ r) = 0$ für $r \neq 0$ mod m. Es ist

$$\gamma(r) \sum_{l=1}^{m} \gamma(l\ r) = \sum_{l=1}^{m} \gamma(r)\gamma(l\ r) = \sum_{l=1}^{m} \gamma((l+1)r) = \sum_{l=1}^{m} \gamma(l\ r)$$

wegen $\gamma((m+1)r) = \gamma(m\ r)\ \gamma(r) = \gamma(r)$. Wegen $\gamma(r) \neq 1$ kann nur $\sum_{l=1}^{m} \gamma(l\ r) = 0$ sein. (Anschaulich: die Punkte $\gamma(l\ r),\ l = 1, \ldots, \frac{m}{ggT(r,m)}$ sind die Eckpunkte eines regelmäßigen, dem Einheitskreis einbeschriebenen Polygons). *q.e.d.*

b) Wir wollen jetzt zeigen, daß diese Matrizen, obwohl sie voll besetzt sind, die Berechnung der Bildvektoren $\mathsf{F}_m z$ bzw. $\mathsf{F}_m^{-1} z$ in $O(m \log m)$ Schritten zulassen. Zur Vereinfachung betrachten wir nur Zweierpotenzen $m = 2^s$.

Satz 1 *Für jedes $s \in \mathbb{N}_0$ und jedes $z \in \mathbb{C}^{2^s}$ lassen sich die Vektoren $\sqrt{2^s}\mathsf{F}_{2^s} z$ und $\sqrt{2^s}\mathsf{F}_{2^s}^{-1} z$ mit $2s\,2^s$ arithmetischen Operationen berechnen. Zum Beispiel ergibt sich für die Matrizen $\tilde{\mathsf{F}}_m = \sqrt{m}\mathsf{F}_m$ und jedes $z \in \mathbb{C}^{2^{s+1}}$ der Vektor $\hat{z} = \tilde{\mathsf{F}}_{2^{s+1}} z$ rekursiv durch die Beziehungen*

$$\begin{pmatrix} \hat{z}_1 \\ \vdots \\ \hat{z}_{2^s} \end{pmatrix} = \hat{x} + \Delta \hat{y} \tag{7}$$

$$\begin{pmatrix} \hat{z}_{2^s+1} \\ \vdots \\ \hat{z}_{2^{s+1}} \end{pmatrix} = \hat{x} - \Delta\hat{y} \tag{8}$$

wobei

$$\hat{x} = \tilde{\mathsf{F}}_{2^s} \begin{pmatrix} z_2 \\ z_4 \\ \vdots \\ z_{2^{s+1}} \end{pmatrix}, \ \hat{y} = \tilde{\mathsf{F}}_{2^s} \begin{pmatrix} z_1 \\ z_3 \\ \vdots \\ z_{2^{s+1}-1} \end{pmatrix} \tag{9}$$

und

$$\Delta = \begin{pmatrix} e^{\frac{2\pi i}{2^{s+1}}1} & & O \\ & \ddots & \\ O & & e^{\frac{2\pi i}{2^{s+1}}2^s} \end{pmatrix} \tag{10}$$

Beweis: Wir verifizieren (7) - (10) und zeigen mit Induktion, daß für jedes $z \in \mathbb{C}^{2^s}$ der Vektor $\tilde{\mathsf{F}}_{2^s} z$ mit $2s2^s$ Operationen berechnet werden kann.
Induktionsbeginn. Im Fall $s = 0$ ist $\mathsf{F}_{2^0}^{-1} = \mathsf{F}_{2^0} = (1)$. Man braucht nichts zu tun.
Induktionsschritt von s nach $s+1$. Wir bezeichnen mit $\gamma_s(j)$ die Zahl $e^{\frac{j}{2^s}2\pi i}$. Sei $z \in \mathbb{C}^{2^{s+1}}$ gegeben. Dann ist

$$\hat{z}_k = \sum_{l=1}^{2^{s+1}} \gamma_{s+1}(-k\ l)\ z_l = \sum_{l=1}^{2^s} \gamma_{s+1}(-k\ 2l)\ z_{2l} + \sum_{l=1}^{2^s} \gamma_{s+1}(-k(2l-1))\ z_{2l-1}.$$

Offenbar ist $\gamma_{s+1}(2j) = \gamma_s(j)$. Mit der Multiplikativität von γ_{s+1} läßt sich dies daher auch so schreiben

$$\hat{z}_k = \sum_{l=1}^{2^s} \gamma_s(-k\ l)\ z_{2l} + \gamma_{s+1}(k) \sum_{l=1}^{2^s} \gamma_s(-k\ l) z_{2l-1} \tag{11}$$

Für $1 \leq k \leq 2^s$ genügt uns dies. Für $2^s+1 \leq k \leq 2^{s+1}$ verwenden wir noch, daß γ_s periodisch ist mit Periode 2^s sowie $\gamma_{s+1}(k) = -\gamma_{s+1}(k-2^s)$ und können daher (11) schreiben als

$$\hat{z}_k = \sum_{l=1}^{2^s} \gamma_s(-(k-2^s)l)\ z_{2l} - \gamma_{s+1}(k-2^s) \sum_{l=1}^{2^s} \gamma_s(-(k-2^s)l)\ z_{2l-1} \tag{12}$$

Die Gleichungen (11) und (12) sind aber nichts anderes als die Beziehungen (7)-(10).
Nach Induktionsvoraussetzung benötigt die Berechnung der Vektoren $\hat{x}$ und $\hat{y}$ jeweils $2s2^s$ Operationen. Die beiden Multiplikationen mit der Diagonalmatrix Δ benötigen jeweils 2^s Multiplikationen. Schließlich sind in (7) und (8) noch jeweils 2^s Additionen bzw. Subtraktionen. Also erhält man $2 \cdot 2s2^s + 2 \cdot 2^s + 2 \cdot 2^s = 2(s+1)2^{s+1}$ Operationen zur Berechnung von $\hat{z}$. Für die Berechnung von $\sqrt{2^s}\mathsf{F}_{2^s}^{-1} z$ ersetzt man im ganzen Argument alle Einheitswurzeln durch ihre komplex konjugierten. *q.e.d.*

Für die Berechnung von $\mathsf{F}_{2^{s+1}} z$ dividiert man ganz zum Schluß noch durch $\sqrt{2^{s+1}}$.

Folgerung 1 *Zwei W-Vektoren der Ausdehnung 2^s können mit $(12(s+1)+8)2^s$ arithmetischen Operationen gefaltet werden.*

Beweis: OBdA seien f, g auf $\{1, \ldots, 2^s\}$ konzentriert. Dann benötigt die Berechnung von $\hat{f}$ und $\hat{g}$ über $\{\frac{1}{2^{s+1}}2\pi, \ldots, 2\pi\}$ jeweils $2(s+1)2^{s+1}+2^{s+1}$. Die Berechnung von $\hat{f}\cdot\hat{g}$ benötigt 2^{s+1} und die Berechnung von $f*g$ mit Hilfe von $\mathsf{F}_{2^{s+1}}^{-1}$ nochmal $2(s+1)2^{s+1}$. Das ergibt zusammen $2(s+1)2^{s+1} + 2^{s+1} + 2(s+1)2^{s+1} + 2^{s+1} + 2^{s+1} + 2(s+1)2^{s+1} + 2^{s+1} = (12(s+1)+8)2^s$. *q.e.d.*

Wir erhalten folgende Tabelle

s	$12(s+1)+2$	2^s	Produkt	$(2^s)^2$
6	92	64	$>$	
7	104	128	$<$	

Wenn wir grob $(2^s)^2$ Operationen für den naiven Zugang von (1) veranschlagen, scheint sich der „Umweg" über die FFT für Vektoren von Zweierpotenzordnung ab der Ausdehnung $2^7 = 128$ zu lohnen. Folgendes Bild zeigt die gemessenen Rechendauern für das im Anhang beschriebene Programm. Wie man erkennt, sind für Vektoren dieser Länge beide Verfahren

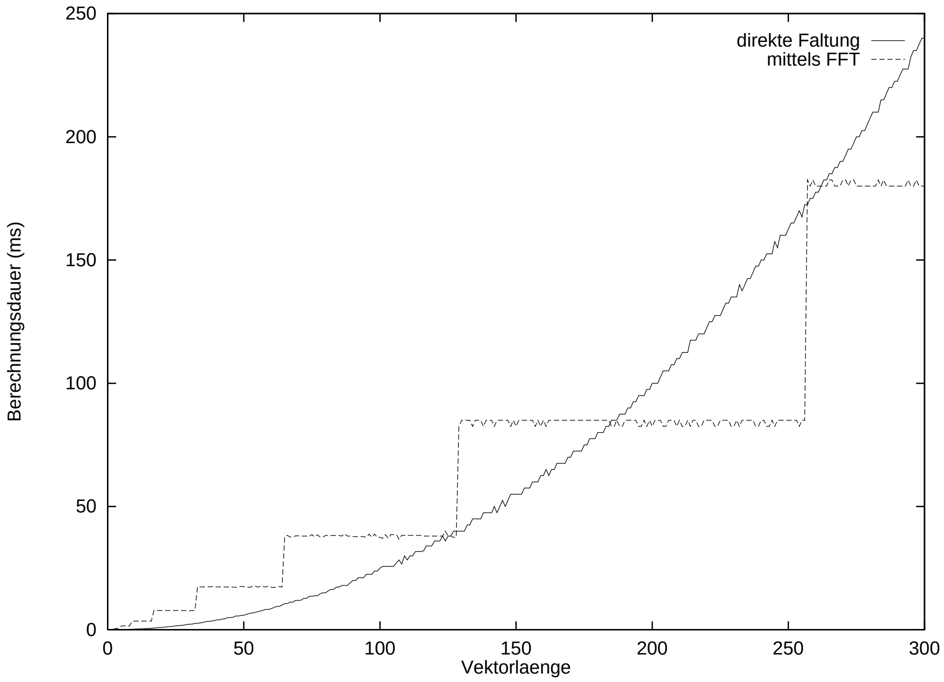

Abbildung 1: Vergleich der mittleren Rechenzeiten (Graphische Darstellung der Ausgabe des Programms `times` aus dem Anhang).

etwa gleich schnell. Dies ist vermutlich auf den zusätzlichen Verwaltungsaufwand des Programms zurückzuführen. Die Beschränkung auf Vektoren von Zweierpotenzordnung führt dazu, daß de facto erst für Vektoren der Länge ≥ 200 die FFT-Faltung effizienter wird.

Anhang: Implementation der FFT

Dieser Anhang enthält eine Implementation des oben beschriebenen Verfahrens zur Faltung zweier Vektoren mittels FFT. Der erste Teil implementiert die Hin- und Rücktransformation der FFT (Dateien `fft.h` und `fft.c`). Den zweiten Teil bildet ein kurzes Programm (Datei `times.c`), das mittels dieser Routinen verschiedene Vektoren miteinander faltet und die benötigte Rechenzeit mißt. Zum Vergleich wird auch die zur direkten Berechnung der Faltung benötigte Zeit gemessen.

Die Funktionen wurden in der Programmiersprache C unter Unix implementiert und verwenden Spracherweiterungen des GNU C-Compilers wie z.B. die Datentypen für komplexe Zahlen. Wie in der Programmiersprache C üblich, werden im Programm alle Arrays mit Null beginnend indiziert, d.h. das $f(1)$ von oben findet sich hier unter der Bezeichnung `f[0]`.

Zur FFT: Die folgenden beiden Dateien stellen drei Funktionen zur Verfügung.

- `void fft (number *data, unsigned int anz)` Erwartet als Eingabe einen Vektor `data` der Länge `anz`. Diese Länge muß eine Zweierpotenz sein. Die Funktion ersetzt den Vektor durch seine (diskrete) Fouriertransformierte.

- `void reverse_fft (number *data, unsigned int anz)` Dito, aber auf den Vektor `data` wird die (diskrete) Fourier-Rücktransformation angewendet.

- `unsigned int next_power_of_two (unsigned int x)` Diese Funktion berechnet aus einer Zahl `x` die nächstgrößere Zweierpotenz. Aus 8 wird z.B. 8 und aus 9 wird 16.

`fft` und `reverse_fft` rufen intern die Funktion `do_fft` auf. Diese Funktion implementiert das oben beschriebene Verfahren, indem sie die Komponenten des Eingabevektors in die zwei Teilvektoren aus Formel (9) aufteilt und sich mit diesen Vektoren als Argument selbst rekursiv aufruft.[1]

Die Deklaration der Funktionen (in `fft.h`) ist

```
/* fft.h - Declarations for the file "fft.c".  */

/** Define the data types.  **/
typedef  __complex__ long double  number;
typedef  long double  skalar;

/** Declare the functions.  **/
extern  unsigned int  next_power_of_two (unsigned int x);

extern  void  fft (number *data, unsigned int anz);
extern  void  reverse_fft (number *data, unsigned int anz);
```

Die Implementation erfolgt in der Datei `fft.c`.

```
/* fft.c - Implement the Fast Fourier
 *         Transformation.  */

#include <math.h>
#include "fft.h"

/* Pi as a 'long double' number */
#define PI_L 3.1415926535897932385L


unsigned int
next_power_of_two (unsigned int x)
/* Returns the smallest power of two,
 * which is greater than or equal to 'x'.
 * This is a possible length of an input
 * vector for the function 'fft'.  */
{
  int  result = 1;

  x -= 1;
  while (x>0)  x >>= 1, result <<=1;
  return  result;
}

static number
complex_exp (skalar phi)
/* Calculate 'exp(i*phi)'.  */
{
  return  cosl(phi) + (0+1i)*sinl(phi);
}

static void
do_fft (number *data, unsigned int size,
        int reverse_flag)
```

[1] Das üblicherweise in der Literatur vorgeschlagenen Verfahren (z.B. bei J. Werner, *Numerische Mathematik*. Vieweg, 1992) unterscheidet sich leicht von dem hier beschriebenen. Dort wird durch Indexberechnungen das explizite Anlegen der Teilvektoren überflüssig gemacht, und so der Speicherbedarf der Funktion reduziert. Allerdings konnte ich bei Zeitmessungen keinerlei Geschwindigkeitsvorteile dieses Verfahrens feststellen.

```
/* Perform a fast (forward or reverse)
 * Fourier transformation on the vector
 * ‘data’.  The result is stored in place
 * of the input data.  ‘size’ is the
 * length of the vector ‘data’.  If
 * ‘reverse_flag’ is non-null, a reverse
 * Fourier transformation is done.  */
{
  unsigned int  m = size/2;
  number  x[m], y[m];
  unsigned int  j, k;
  number  q, q0;

  if (size>1) {
    for (j=1,k=0; j<size; j+=2)
      x[k++] = data[j];
    for (j=0,k=0; j<size; j+=2)
      y[k++] = data[j];
    do_fft (x, m, reverse_flag);
    do_fft (y, m, reverse_flag);

    if (reverse_flag) {
      q0 = complex_exp (2*PI_L/size);
    } else {
      q0 = complex_exp (-2*PI_L/size);
    }
    q = q0;
    for (j=0; j<m; ++j) {
      data [j] = x[j] + q*y[j];
      data [j+m] = x[j] - q*y[j];
      q *= q0;
    }
  }
}

void
fft (number *data, unsigned int size)
/* Perform a Fast Fourier Transformation
 * on the vector ‘data’.  The result is
 * stored in place of the input data.
 * ‘size’ is the length of the vector
 * ‘data’ and has to be a power of two. */
{
  unsigned int  j;
  number  q;

  do_fft (data, size, 0);
  q = sqrtl (1.0/size);
  for (j=0; j<size; ++j)  data[j] *= q;
}

void
reverse_fft (number *data,
             unsigned int size)
/* Perform a fast reverse Fourier
 * Transformation on the vector ‘data’.
 * The result is stored in place of the
 * input data.  ‘size’ is the length of
 * the vector ‘data’ and has to be a power
 * of two.  */
{
  unsigned int  j;
  number  q;

  do_fft (data, size, 1);
  q = sqrtl (1.0/size);
  for (j=0; j<size; ++j)  data[j] *= q;
}
```

Zur Faltung: Das folgende Programm implementiert zwei Methoden, um die Faltung zweier Vektoren zu berechnen. Zur Vereinfachung müssen beide Vektoren dieselbe Größe haben. Die Funktion `method_a` implementiert direkt den Ansatz aus Formel (1). Die Funktion `method_b` verwendet das Verfahren aus Satz 1 bzw. Folgerung 1. Ist die Länge der Vektoren hier keine Zweierpotenz, so füllt die Funktion die Vektoren mit Nullen auf, bis eine geeignete Länge erreicht ist. Ist der Träger von $\hat{f}$ dann in $\{1, ..., 2^s\}$ enthalten, so gilt nach Gleichung (3)

$$\hat{f}(2\pi \frac{l}{2^{s+1}}) = \sum_{k=1}^{2^s} f(k) e^{2\pi i \frac{lk}{2^{s+1}}} = \sum_{k=1}^{2^{s+1}} f(k) e^{2\pi i \frac{lk}{2^{s+1}}},$$

dh. man kann den Vektor weiter bis zur Länge 2^{s+1} mit Nullen auffüllen und erhält aus diesem neuen Vektor durch Anwendung der 2^{s+1}-komponentigen FFT die benötigten Werte $\hat{f}(\frac{1}{2^{s+1}}2\pi), \hat{f}(\frac{2}{2^{s+1}}2\pi), \ldots, \hat{f}(2\pi)$.
Die Funktion `doit` wendet nacheinander beide Verfahren auf Vektoren der Länge `m` an und mißt die dazu benötigten Zeiten. Das Hauptprogramm `main` erzeugt zwei Vektoren mit „zufälligen“ Einträgen und ruft die Funktion `doit` nacheinander für $\mathtt{m} = 1, 2, \ldots, 300$ mit diesen Vektoren als Argument auf.

```
/* times.c - Geschwindigkeitsvergleich */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#include "fft.h"

static void
method_a (unsigned int m, const number *f,
          const number *g, number *result)
/* Methode a: "Direkte" Berechnung der
 * Faltung.  ‘f’ und ‘g’ sind Vektoren der
 * Laenge ‘m’.  Die Funktion legt das
 * Ergebnis in ‘result’ ab; dies sollte
 * ein Vektor der Groesse ‘2*m’ sein.  */
{
  unsigned int  k, l;

  for (k=0; k<m; ++k) {
    number  zwerg = 0;
    for (l=0; l<k; ++l)
      zwerg += f[l] * g[k-l-1];
    result [k] = zwerg;
  }
  for (k=m; k<2*m; ++k) {
    number  zwerg = 0;
    for (l=k-m; l<m; ++l)
      zwerg += f[l] * g[k-l-1];
    result [k] = zwerg;
  }
}

```

```
static void
method_b (unsigned int m, const number *f,
          const number *g, number *result)
/* Methode b: Berechnet erst die
 * (`next_power_of_two(2*m)'-komponentige)
 * Fouriertransformierte von `data',
 * quadriert das Ergebnis und transformiert
 * es zurueck.  Das Ergebnis ist ein Vektor
 * der Laenge `next_power_of_two(2*m)' und
 * wird in `result' abgelegt.  */
{
  int  size = next_power_of_two(2*m);
  number  buf_f[size], buf_g[size];
  skalar  q = sqrtl (size);
  int  j;

  /* Kopiere die Eingabevektoren und
   * fuelle mit Nullen auf.  */
  memcpy (buf_f, f, m*sizeof(number));
  memcpy (buf_g, g, m*sizeof(number));
  for (j=m; j<size; ++j) {
    buf_f[j] = 0;  buf_g[j] = 0;
  }

  fft (buf_f, size);
  fft (buf_g, size);
  for (j=0; j<size; ++j)
    result[j] = q * buf_f[j] * buf_g[j];
  reverse_fft (result, size);
}

static void
doit (unsigned int m, const number *f,
      const number *g)
/* Misst die Ausfuehrungszeit beider
 * Methoden.  */
{
  int  size = next_power_of_two (2*m);
  number  result_a[2*m], result_b[size];
  clock_t  t1, t2, t3;
  double  time_a, time_b, base;
  skalar  delta;
  unsigned int  j, count;

  j = next_power_of_two (m);

  /* "Willkuerlich" festgelegte Anzahl
   * von Testlaeufen.  */
  count = (int)(80000.0 / (m*m));
  if (count>100)  count = 100;
  if (count<4)  count = 4;

  t1 = clock ();
  for (j=0; j<count; ++j)
    method_a (m, f, g, result_a);
  t2 = clock ();
  for (j=0; j<count; ++j)
    method_b (m, f, g, result_b);
  t3 = clock ();
  base = CLOCKS_PER_SEC*count;
  time_a = (t2-t1) / base;
  time_b = (t3-t2) / base;

  delta = 0;
  for (j=0; j<2*m; ++j) {
    number  x = result_b[j] - result_a[j];
    skalar  tmp = sqrtl (__real__(x * ~x));
    if (tmp>delta)  delta = tmp;
  }
  /* Die restlichen Komponenten sollten
   * wieder Null sein.  */
  for (j=m*2; j<size; ++j) {
    number  x = result_b[j];
    skalar  tmp = (__real__(x * ~x));
    if (tmp>delta)  delta = tmp;
  }

  printf ("%3u %6.1f %6.1f %12Lg %d\n",
          m, 1000*time_a, 1000*time_b,
          delta, count);
}

int
main ()
{
  unsigned int  j, m;
  number  f[300], g[300];

  /* Zufaellige Vektoren als Eingabe.  */
  for (j=0; j<300; ++j) {
    f[j] = (double)rand() / RAND_MAX;
    g[j] = (double)rand() / RAND_MAX;
  }

  printf ("  m  time_a time_b"
          "      Fehler Anzahl\n");
  for (m=2; m<=300; ++m)  doit (m, f, g);

  return  0;
}
```

Man erhält folgendes Ergebnis (vgl. Abb. 1). Dabei ist `m` die Länge des Vektors und `time_a` sowie `time_b` sind die mittleren Rechenzeiten für Methode A bzw. Methode B in Millisekunden (wie sie sich auf einem Computer mit 80486 Prozessor unter dem Betriebssystem Linux ergaben). Die Berechnung wurde `Anzahl`-mal durchgeführt, um die kurzen Zeiten genauer messen zu können. Die Spalte `Fehler` gibt die Maximumnormen der Differenz beider Ergebnisse an.

```
  m  time_a time_b      Fehler Anzahl
  2    0.0    0.2  4.07339e-20 100
  3    0.1    0.6  1.80188e-19 100
  4    0.1    0.6   2.9277e-19 100
  5    0.1    1.5  2.17553e-19 100
  6    0.1    1.6  2.97125e-19 100
  7    0.1    1.5  5.10761e-19 100
...
296  235.0  180.0  4.30875e-16 4
297  235.0  182.5  4.23835e-16 4
298  237.5  180.0  4.31066e-16 4
299  240.0  180.0  4.23798e-16 4
300  240.0  180.0  4.23678e-16 4
```