Berechnung von π auf einige tausend Dezimalstellen Genauigkeit

Matthias Köppe

Zusammenfassung

Approximationen für π berechnet man gewöhnlich durch Reihenentwicklungen des Arkustangens. Neben der Implementation eines solchen Verfahrens in der Programmiersprache Borland Pascal mit i386-Assemblercode für kritische innere Schleifen wird auch ein alternativer „Spigot"-Algorithmus von Rabinowitz implementiert, der ohne die sonst erforderte Arithmetik beliebig genauer Zahlen auskommt.

1 Berechnung durch Reihenentwicklungen des Arkustangens

1.1 Grundlagen

Die Reihenentwicklung für den Arkustangens lautet

$$\arctan x = x - \frac{x^3}{3} + \frac{x^5}{5} - + \cdots \tag{1}$$

Das Restglied $R_n(x)$ bei Berücksichtigung von n Summanden kann folgendermaßen abgeschätzt werden:

$$R_n(x) = (-1)^n \cdot \frac{x^{2n+1}}{2n+1} \cdot \frac{1}{1+\alpha x^2}, \qquad 0 < \alpha < 1 \tag{2}$$

Der Logarithmus des Reziproken liefert ungefähr die Zahl der gültigen Nachkommastellen in Abhängigkeit von der Zahl der Summanden n und dem Argument x:

$$\begin{aligned} N(n,x) &= \log\left|\frac{1}{R_n(x)}\right| \\ &= (2n+1)\log\frac{1}{x} + \log(2n+1) + \log(1+\alpha x^2) \end{aligned} \tag{3}$$

Eine Approximation für π läßt sich beispielsweise durch die Beziehung $\frac{\pi}{4} = \arctan 1$ erzielen. Nach Gleichung (3) benötigen wir für N gültige Nachkommastellen aber etwa 10^N Summanden, so daß mit dieser Entwicklung keine geeignete Genauigkeit erzielt werden kann.

Durch Wahl kleinerer Argumente x für den Arkustangens läßt sich ein günstigeres Verhalten der Restglieder erzielen. Wegen $N(n,x) > (2n+1)\log\frac{1}{x}$ genügen für N gültige Stellen weniger als $\frac{N}{2\log\frac{1}{x}}$ Summanden.

Wir wollen folgende Formeln von Machin (1706), Gauss und Störmer (1896) einsetzen:

$$\pi = 16 \arctan \frac{1}{5} - 4 \arctan \frac{1}{239} \tag{4}$$

$$\pi = 48 \arctan \frac{1}{18} + 32 \arctan \frac{1}{57} - 20 \arctan \frac{1}{239} \tag{5}$$

$$\pi = 23 \arctan \frac{1}{8} + 8 \arctan \frac{1}{57} + 4 \arctan \frac{1}{239} \tag{6}$$

Mittels Additionstheoremen lassen sich diese Formeln nachweisen. Es soll an dieser Stelle genügen, die Gleichung von Machin (4) zu überprüfen:

$$\begin{aligned}\pi &= 16 \arctan \frac{1}{5} - 4 \arctan \frac{1}{239} \\ &= 8 \arctan \frac{5}{12} - 4 \arctan \frac{1}{239} \\ &= 4 \arctan \frac{120}{119} - 4 \arctan \frac{1}{239} \\ &= 4 \arctan 1\end{aligned}$$

Die Qualität dieser Formeln kann nach dem jeweils größten Argument bewertet werden. Wir erhalten als benötigte Summandenanzahl n in Abhängigkeit von der Stellenzahl N:

$$\begin{aligned}n_{Machin}(N) < \frac{N}{2 \log 5} &\approx 0,72N \\ n_{Gauss}(N) < \frac{N}{2 \log 18} &\approx 0,40N \\ n_{Stoermer}(N) < \frac{N}{2 \log 8} &\approx 0,55N\end{aligned}$$

1.2 Komplexitätsbetrachtungen

Um π auf einige tausend Dezimalstellen Genauigkeit zu berechnen, muß eine Arithmetik beliebig genauer Zahlen anstatt gewöhnlicher Fließkomma-Arithmetik verwendet werden.

Zur Berechung der Arkustangensreihe für ein genaues Argument sind offenbar viele – nämlich $O(N)$ – Multiplikationen nötig. Eine naive Implementation der Multiplikation hat eine Zeitkomplexität von $O(N^2)$. Mit der Schnellen Fourier-Transformation (FFT) läßt sich die Zeitkomplexität zwar auf $O(N \log N)$ reduzieren, aber die Implementation ist recht aufwendig.

In den verwendeten Formeln muß die Arkustangensreihe allerdings nur mit Argumenten berechnet werden, die Stammbrüche mit kleinen Nennern sind. Dadurch können wir den Algorithmus so formulieren, daß keine Multiplikation zweier genauer Zahlen erforderlich ist. – Wir benötigen damit nur noch die folgenden Operationen:

- Addition und Subtraktion zweier genauer Zahlen
- Multiplikation einer genauen Zahl mit einer kleinen ganzen Zahl
- Division einer genauen Zahl durch eine kleine ganze Zahl

Alle diese Operationen lassen sich ohne Speicherplatz und mit einer Zeitkomplexität von $O(N)$ durchführen.

Die Berechnung des Arkustangens durch Summation von $n(N) = O(N)$ Gliedern mit Stellenzahl N hat insgesamt also eine Zeitkomplexität von $O(N^2)$.

Wenn intern in einem anderen Zahlensystem als dem Dezimalsystem gearbeitet wird, ist zur Ausgabe von genauen Zahlen eine Konvertierung erforderlich. Diese erreicht man durch mehrfache Multiplikationen mit kleinen ganzen Zehnerpotenzen. Insgesamt hat die Konvertierung eine Zeitkomplexität von $O(N^2)$.

1.3 Implementationsdetails

1.3.1 Zahlensystem

Die obigen Komplexitätsbetrachtungen haben gezeigt, daß es für die Geschwindigkeit der Approximation insbesondere darauf ankommt, die verwendete Stellenzahl zu reduzieren, ohne die Genauigkeit zu senken. Dies erreicht man durch einen Übergang von der Basis 10 zu einer möglichst hohen, aber prozessorunterstützten Basis.

Durch die Verwendung von 32-bit-Prozessoren und direkter Codierung der primitiven Operationen im i386-Assembler kann als interne Basis 2^{32} gewählt werden. Bei Additionen genauer Zahlen senken wir die Zahl der Additionszyklen um den Faktor 10 gegenüber der Addition der Zahlen im Dezimalsystem.

1.3.2 Interne Repräsentation genauer Zahlen

Der abstrakte Datentyp `THuge` verkörpert eine Zahl einstellbarer Genauigkeit. Er ist intern als offenes dynamisches Array von 32-bit-Worten (PASCAL-Typ `LongInt`) realisiert.

Wir benötigen nur relativ kleine, nichtnegative Zahlen. Deshalb stellen wir die Vorkommastellen durch ein einzelnes 32-bit-Wort dar und definieren die Zahlen als vorzeichenlos. Das Vorkommawort wird am höchsten Index des Arrays gespeichert; ihm gehen die Nachkommaworte voraus.

Es werden zwei Konstruktoren bereitgestellt, die `THuge`-Zahlen dynamisch erzeugen:

- `NewHuge` erzeugt eine Zahl und weist ihr einen ganzzahligen Wert zu
- `CopyHuge` erzeugt eine Zahl und überträgt den Wert einer anderen Zahl in sie

Der Destruktor `FreeHuge` zerstört eine Zahl.

1.3.3 Arithmetische Operationen

Die elementaren Operationen sind direkt im i386-Assembler implementiert worden, um die inneren Schleifen zu verkürzen und Berechnungen mit 32-bit-Zahlen zu erlauben.

- `AddHuge` addiert zwei Zahlen und speichert das Ergebnis in der ersten Zahl.
- `SubHuge` subtrahiert die zweite Zahl von der ersten und speichert das Ergebnis in der ersten Zahl.

- `TimesHuge` multipliziert eine Zahl mit einem `LongInt`.
- `DivHuge` dividiert eine Zahl durch ein `LongInt`.

1.3.4 Konvertierung in das Dezimalsystem

Eine `THuge`-Zahl kann im Dezimalsystem gedruckt werden, indem zunächst die Vorkommastellen ausgegeben werden und dann sukzessive die Restzahl mit einer Zehnerpotenz multipliziert wird, um die Nachkommastellen nach vorn zu bringen.

Die größte Zehnerpotenz, die in ein 32-bit-Wort paßt, ist 10^9. Nach jeder Multiplikation können also neun Dezimalstellen gedruckt werden.

Die Konvertierung und Ausgabe findet in der Prozedur `PrintHuge` statt.

1.3.5 Algorithmus zur Arkustangens-Berechnung

Die Funktion `ArcTanHuge` berechnet den Ausdruck $\mathit{Factor} \cdot \arctan \frac{1}{\mathit{Arg}}$ durch Berücksichtigung von $2 \cdot \mathit{Count} + 1$ Summanden.

1. Berechne Arg^2 und Arg^4.
2. Initialisiere Schleifeninvarianten:
   - $\mathit{ArgPower} \longleftarrow \frac{\mathit{Factor}}{\mathit{Arg}}$
   - $\mathit{Result} \longleftarrow \mathit{ArgPower}$
   - $\mathit{Denominator} \longleftarrow 1$
3. Iteriere *Count* mal:
   - $\mathit{Denominator} \longleftarrow \mathit{Denominator} + 2$
   - $X \longleftarrow \frac{\mathit{ArgPower}}{\mathit{Denominator} \cdot \mathit{Arg}^2}$
   - $\mathit{Result} \longleftarrow \mathit{Result} - X$
   - $\mathit{ArgPower} \longleftarrow \frac{\mathit{ArgPower}}{\mathit{Arg}^4}$
   - $\mathit{Denominator} \longleftarrow \mathit{Denominator} + 2$
   - $X \longleftarrow \frac{\mathit{ArgPower}}{\mathit{Denominator}}$
   - $\mathit{Result} \longleftarrow \mathit{Result} + X$

1.4 PASCAL-Programm

```
program Pi;

{ Berechnet Pi durch Arcustangensreihen.
}

{ THuge-Zahlen, vorzeichenlose Festkommazahlen.
  Layout: <HugeSize> DWORDs.
  Das letzte DWORD enth"alt die Vorkommastellen.
}
type
  PHuge = ^THuge;
```

```
  THuge = array[0..0] of LongInt;

function NewHuge(Value: LongInt; HugeSize: Integer): PHuge;
{ Erzeugt eine Zahl und weist ihr einen ganzzahligen Wert zu }
var
  P: PHuge;
begin
  GetMem(P, HugeSize * SizeOf(LongInt));
  FillChar(P^, HugeSize * SizeOf(LongInt), 0);
  P^[HugeSize - 1] := Value;
  NewHuge := P
end;

procedure FreeHuge(Huge: PHuge; HugeSize: Integer);
{ Entfernt eine Zahl }
begin
  FreeMem(Huge, HugeSize * SizeOf(LongInt))
end;

function CopyHuge(Huge: PHuge; HugeSize: Integer): PHuge;
{ Erzeugt eine Zahl und weist ihr eine Kopie des Arguments zu }
var
  P: PHuge;
begin
  GetMem(P, HugeSize * SizeOf(LongInt));
  Move(Huge^, P^, HugeSize * SizeOf(LongInt));
  CopyHuge := P
end;

{$L huge.obj (huge.asm)  Elementare Operationen }

procedure AddHuge(A, B: PHuge;
  HugeSize: Integer); near; external;
procedure SubHuge(A, B: PHuge;
  HugeSize: Integer); near; external;
procedure TimesHuge(A: PHuge; Int: LongInt;
  HugeSize: Integer); near; external;
procedure DivHuge(A: PHuge; Int: LongInt;
  HugeSize: Integer); near; external;

const
{ Gr"o"ste ganze Zahl im Wertebereich eines Doppelwortes }
  LongUnit = 1000000000;
  LongExp = 9;

function CodedDigits(HugeSize: Integer): Integer;
{ Berechnet die Anzahl der in einer THuge-Zahl
  codierten dezimalen Nachkommastellen. }
begin
  CodedDigits := Trunc(
```

```
    (HugeSize - 1) * (SizeOf(LongInt) * 8) * ln(2) / ln(10));
end;

function NeededToCodeDigits(Count: Integer): Integer;
{ Berechnet die HugeSize, die zur Codierung von Count
  dezimalen Nachkommastellen erfordert wird. }
begin
  NeededToCodeDigits := Trunc(
    Count * (ln(10) / ln(2)) / (SizeOf(LongInt) * 8)) + 3;
end;

procedure PrintLong(var F: Text; Long: LongInt);
{ Druckt einen Doppelwortwert als Dezimalzahl mit
  9 Stellen und evtl. f"uhrenden Nullen. Es wird
  angenommen, da"s Long < LongUnits gilt. }
var
  Buf: string[LongExp];
  i: Integer;
begin
  Str(Long:LongExp, Buf);
  i := 1;
  while Buf[i] = ' ' do
  begin
    Buf[i] := '0';
    Inc(i)
  end;
  Write(F, Buf)
end;

procedure PrintHuge(var F: Text; Huge: PHuge;
  HugeSize: Integer);
{ Druckt eine Zahl. }
var
  Copy: PHuge;
  i: Integer;
begin
  Copy := CopyHuge(Huge, HugeSize);
  Write(F, Copy^[HugeSize - 1], '.');
  Copy^[HugeSize - 1] := 0;
  TimesHuge(Copy, LongUnit, HugeSize);
  For i := 1 to (CodedDigits(HugeSize) div LongExp) do
  begin
    PrintLong(F, Copy^[HugeSize - 1]);
    Copy^[HugeSize - 1] := 0;
    TimesHuge(Copy, LongUnit, HugeSize);
  end;
  FreeHuge(Copy, HugeSize)
end;

function ArcTanHuge(Arg: LongInt; Factor: LongInt;
```

```
  Count: Integer; HugeSize: Integer): PHuge;
{ Berechnet Factor*arctan(1/Arg).
}
var
  i: Integer;
  Result, ArgPower, X: PHuge;
  Denominator, ArgSquare, ArgQuad: LongInt;
begin
  If Count = 0
  then Count := Trunc(
    CodedDigits(HugeSize) / ((2 / ln(10)) * ln(Arg)));
  ArgSquare := Sqr(Arg);
  ArgQuad := Sqr(ArgSquare);
  { Aktuelle Argumentpotenz anlegen }
  ArgPower := NewHuge(Factor, HugeSize);
  DivHuge(ArgPower, Arg, HugeSize);
  { Ergebnispuffer anlegen, erstes Glied speichern }
  Denominator := 1;
  Result := CopyHuge(ArgPower, HugeSize);
  { Iterationsschleife }
  For i := 1 to Count do
  begin
    { Negatives Glied }
    Inc(Denominator, 2);
    X := CopyHuge(ArgPower, HugeSize);
    DivHuge(X, Denominator * ArgSquare, HugeSize);
    SubHuge(Result, X, HugeSize);
    FreeHuge(X, HugeSize);
    { Positives Glied }
    DivHuge(ArgPower, ArgQuad, HugeSize);
    Inc(Denominator, 2);
    X := CopyHuge(ArgPower, HugeSize);
    DivHuge(X, Denominator, HugeSize);
    AddHuge(Result, X, HugeSize);
    FreeHuge(X, HugeSize);
  end;
  { Zwischenwerte entfernen }
  FreeHuge(ArgPower, HugeSize);
  { Ergebnis liefern }
  ArcTanHuge := Result
end;

function PiByMachin(Count: Integer; HugeSize: Integer): PHuge;
{ pi = 16 arctan(1/5) - 4 arctan(1/239) }
var
  A, B: PHuge;
begin
  A := ArcTanHuge(5, 16, Count, HugeSize);
  B := ArcTanHuge(239, 4, Count, HugeSize);
  SubHuge(A, B, HugeSize);
```

```
  FreeHuge(B, HugeSize);
  PiByMachin := A
end;

function PiByGauss(Count: Integer; HugeSize: Integer): PHuge;
{ pi = 48 arctan(1/18) + 32 arctan(1/57) - 20 arctan(1/239) }
var
  A, B: PHuge;
begin
  A := ArcTanHuge(18, 48, Count, HugeSize);
  B := ArcTanHuge(57, 32, Count, HugeSize);
  AddHuge(A, B, HugeSize);
  FreeHuge(B, HugeSize);
  B := ArcTanHuge(239, 20, Count, HugeSize);
  SubHuge(A, B, HugeSize);
  FreeHuge(B, HugeSize);
  PiByGauss := A
end;

function PiByStoermer(Count: Integer; HugeSize: Integer): PHuge;
{ pi = 24 arctan(1/8) + 8 arctan(1/57) + 4 arctan(1/239) }
var
  A, B: PHuge;
begin
  A := ArcTanHuge(8, 24, Count, HugeSize);
  B := ArcTanHuge(57, 8, Count, HugeSize);
  AddHuge(A, B, HugeSize);
  FreeHuge(B, HugeSize);
  B := ArcTanHuge(239, 4, Count, HugeSize);
  AddHuge(A, B, HugeSize);
  FreeHuge(B, HugeSize);
  PiByStoermer := A
end;

function PiBySpigot(Count: Integer; HugeSize: Integer): PHuge;
var
  sum, x: PHuge;
  i: Integer;
begin
  sum := NewHuge(2, HugeSize);
  x := NewHuge(2, HugeSize);
  For i := 1 to Count do
  begin
    TimesHuge(x, i, HugeSize);
    DivHuge(x, 2 * i + 1, HugeSize);
    AddHuge(Sum, x, HugeSize)
  end;
  FreeHuge(x, HugeSize);
  PiBySpigot := sum
end;
```

```
{ Main
}

var
  TheHugeSize: Integer;
  Digits, Iterations: Integer;
  Method: string;
  E: Integer;
  X: PHuge;

procedure Syntax;
begin
  WriteLn('Berechnung von Pi durch Arcustangensreihen.');
  WriteLn('Argumente: <Stellenzahl> <Iterationen> ',
    '[Machin|Gauss|Stoermer]');
  Halt(1);
end;

begin
  Val(ParamStr(1), Digits, E); If E <> 0 then Syntax;
  Val(ParamStr(2), Iterations, E); If E <> 0 then Syntax;
  TheHugeSize := NeededToCodeDigits(Digits);
  Method := ParamStr(3);
  case Upcase(Method[1]) of
    'M': X := PiByMachin(Iterations, TheHugeSize);
    'G': X := PiByGauss(Iterations, TheHugeSize);
    'S': X := PiByStoermer(Iterations, TheHugeSize);
    'X': X := PiBySpigot(Iterations, TheHugeSize);
    else Syntax;
  end;
  PrintHuge(Output, X, TheHugeSize); WriteLn;
  FreeHuge(X, TheHugeSize);
end.
```

1.5 i386-Assembler-Modul

```
;
; huge numbers

        ideal
        locals  @@
        public  AddHuge, SubHuge, TimesHuge, DivHuge

segment code byte public
        assume  cs: code
        p386

proc    AddHuge near
```

```
        arg     HugeSize: word, B: dword, \
                A: dword = ArgSize
        push    bp
        mov     bp, sp

        mov     cx, [HugeSize]
        push    ds
        lds     si, [B]                 ; source
        les     di, [A]                 ; dest
        clc                             ; clear carry
@@1:
        mov     eax, [si]
        adc     [es:di], eax
        pushf
        add     si, 4
        add     di, 4
        popf
        loop    @@1

        pop     ds
        pop     bp
        ret     ArgSize
endp

proc    SubHuge near
        arg     HugeSize: word, B: dword, \
                A: dword = ArgSize
        push    bp
        mov     bp, sp

        mov     cx, [HugeSize]
        push    ds
        lds     si, [B]                 ; source
        les     di, [A]                 ; dest
        clc                             ; clear carry
@@1:
        mov     eax, [si]
        sbb     [es:di], eax
        pushf
        add     si, 4
        add     di, 4
        popf
        loop    @@1

        pop     ds
        pop     bp
        ret     ArgSize
endp

proc    TimesHuge near
```

```
        arg     HugeSize: word, I: dword, \
                A: dword = ArgSize
        push    bp
        mov     bp, sp

        mov     cx, [HugeSize]
        mov     ebx, [I]
        les     di, [A]                 ; dest
        xor     esi, esi                ; clear carry
@@1:
        mov     eax, [es:di]
        mul     ebx
        add     eax, esi
        adc     edx, 0
        mov     [es:di], eax
        mov     esi, edx
        add     di, 4
        loop    @@1

        pop     bp
        ret     ArgSize
endp

proc    DivHuge near
        arg     HugeSize: word, I: dword, \
                A: dword = ArgSize
        push    bp
        mov     bp, sp

        mov     cx, [HugeSize]
        mov     ax, cx
        dec     ax
        shl     ax, 2
        mov     ebx, [I]
        les     di, [A]                 ; dest
        add     di, ax
        xor     edx, edx                ; clear carry
@@1:
        mov     eax, [es:di]
        div     ebx
        mov     [es:di], eax
        sub     di, 4
        loop    @@1

        pop     bp
        ret     ArgSize
endp

ends
end
```

2 Der Spigot-Algorithmus von Rabinowitz

2.1 Grundlagen

Der „Spigot"-Algorithmus für die Stellen π wurde 1995 von Stanley Rabinowitz und Stan Wagon vorgestellt. Er basiert auf der folgenden Reihendarstellung für $\frac{\pi}{2}$:

$$\frac{\pi}{2} = 1 + \frac{1}{3} + \frac{1 \cdot 2}{3 \cdot 5} + \frac{1 \cdot 2 \cdot 3}{3 \cdot 5 \cdot 7} + \ldots \tag{7}$$

Durch Klammerung wie im Horner-Schema zur Berechnung von Polynomen erhält man die Darstellung

$$\pi = 2 + \frac{1}{3}(2 + \frac{2}{5}(2 + \frac{3}{7}(2 + \ldots))). \tag{8}$$

Bezüglich der „gemischten Basis" $(\frac{1}{3}, \frac{2}{5}, \frac{3}{7}, \ldots)$ kann π also als $2,222\ldots$ dargestellt werden.

Der Algorithmus von Rabinowitz nimmt lediglich die Konvertierung zwischen dem Zahlensystem mit der gemischten Basis und dem Dezimalsystem vor. In jedem Schritt des Algorithmus wird dabei eine Dezimalstelle von π ermittelt. Unter bestimmten Bedingungen müssen einige Ziffern nachgebessert werden.

Der Algorithmus hat eine Zeitkomplexität von $O(N^2)$ und ist sehr leicht zu implementieren. Für eine Genauigkeit von N Dezimalstellen muß allerdings eine Kette von $[\frac{10}{3}N]$ „Ziffern" verwendet werden, die in jedem Schritt des Algorithmus (das heißt: für jede Dezimalstelle) sequentiell verarbeitet werden. Deshalb läßt sich dieses Verfahren weniger effizient realisieren als eine gewöhnliche Reihenentwicklung.

2.2 PASCAL-Programm

```
program SpigotPi;
{ A Spigot Algorithm for the Digits of Pi }

const
  { Maximale Stellenzahl }
  Max = 8000;

{ Der Zwischenspeicher f"ur die noch nicht g"ultigen Stellen }
var
  NumberOfDigits: Integer;
  DigitBuffer: array[0..Max] of Char; { "uberdimensioniert }
  NextDigit: PChar;

procedure FlushDigits;
begin
  NextDigit[0] := #0;
  Write(DigitBuffer);
  NextDigit := @DigitBuffer
end;

procedure IncreaseDigits;
```

```
var
  P: PChar;
begin
  P := @DigitBuffer;
  while P <> NextDigit do
  begin
    If P[0] = '9'
    then P[0] := '0'
    else Inc(P[0]);
    Inc(P)
  end
end;

procedure AddDigit(Digit: Integer);
begin
  NextDigit[0] := Chr(Digit + Ord('0'));
  Inc(NextDigit)
end;

{ Die Ziffernkette }

var
  TheChain: array[0..(10 * Max) div 3] of Integer;
  NumberOfColumns: Integer;

procedure SetupChain;
var
  i: Integer;
begin
  For i := 0 to NumberOfColumns do
    TheChain[i] := 2
end;

function IterateChain: Integer;
var
  column, B, remainder: Integer;
  x, carry, quotient: LongInt;
begin
  column := NumberOfColumns;
  carry := 0;
  while column > 0 do
  begin
    x := carry + LongInt(TheChain[column]) * 10;
    B := 2 * column + 1;
    quotient := x div B;
    remainder := x mod B;
    carry := quotient * column;
    TheChain[column] := remainder;
    Dec(column)
  end;
```

```
  x := LongInt(TheChain[0]) * 10 + carry;
  TheChain[0] := x mod 10;
  IterateChain := x div 10
end;

var
  i, E, Digit: Integer;

procedure Syntax;
begin
  WriteLn('Berechnung von Pi durch einen Spigot-Algorithmus.');
  WriteLn('Argument: <Stellenzahl>');
  Halt(1);
end;

begin
  If ParamCount <> 1 then Syntax;
  Val(ParamStr(1), NumberOfDigits, E);
  If E <> 0 then Syntax;
  If NumberOfDigits > Max then NumberOfDigits := Max;
  NumberOfColumns := (10 * LongInt(NumberOfDigits)) div 3;
  SetupChain;
  NextDigit := @DigitBuffer;
  For i := 1 to NumberOfDigits do
  begin
    Digit := IterateChain;
    case Digit of
      9:
        AddDigit(9);
      10:
        begin
          IncreaseDigits;
          FlushDigits;
          AddDigit(0)
        end;
      else begin
        FlushDigits;
        AddDigit(Digit)
      end
    end
  end;
  FlushDigits;
  WriteLn
end.
```

Literatur

[1] Gellert, W. (Herausgeber): *Mathematik*. Bibliographisches Institut, Leipzig, 13. Auflage, 1986.

[2] Sedgewick, Robert: *Algorithmen*. Addison-Wesley Publishing Company, Bonn–München–Reading, Mass., 1991.

[3] Stewart, Ian: *Mathematische Unterhaltungen*. Spektrum der Wissenschaft, Seiten 10–14, Dezember 1995.