Rozdział 5

Pętla o nieokreślonej ilości przebiegów

{chap:while}

5.1 Pożyczka

Poprzednio (str. 48) przeprowadzaliśmy symulację, jak będzie przebiegała spłata pożyczki, obliczaliśmy ile nam zostanie do spłacenia po określonym czasie. Jednak ta symulacja nie była odpowiedzią na pytanie, które jest najbardziej interesujące przy spłacie pożyczek. Zazwyczaj najbardziej nas interesuje, kiedy w końcu ją spłacimy.

Częściowo odpowiedź jest w poprzednim zadaniu. Jeśli umiemy obliczyć ile będziemy mieli długu po n latach, umiemy też obliczyć ile będziemy mieli długu po $n+1$ latach (a dług musi maleć, inaczej nigdy go nie spłacimy) to wystarczy znaleźć taką ilość lat aż dług stanie się zerowy. Wtedy będzie to oznaczało, że nic już nie jesteśmy winni bankowi czyli cała pożyczka została spłacona.

Należałoby to sformułować następująco: „powtarzaj obliczanie zadłużenia w kolejnych latach aż dług stanie się zerowy".

Jednak do tej pory nie mamy narzędzi programistycznych, aby takie zadanie rozwiązać. Wiemy jak powtarzać coś zadaną z góry ilość razy, ale w tym zadaniu to właśnie to ile razy trzeba powtórzyć (ile lat będziemy spłacali pożyczkę) jest treścią zadania.

Potrzebny jest nowy rodzaj pętli: „powtarzaj tak długo, aż coś się stanie". Jest to pętla o *nieokreślonej ilości przebiegów*.

Formalny zapis takiej pętli jest następujący:

while *warunek*
    ...
    ...
end%while

Jest on bardzo podobny do zapisu instrukcji warunkowej (zob. pkt. 3.2), można by wręcz powiedzieć, że jest łudząco podobny. Jednak sens tego zapisu jest zupełnie

inny. O ile w przypadku warunku `if` linie wewnątrz warunku będą *raz* wykonane jeśli warunek jest spełniony, o tyle w przypadku pętli `while` linie wewnątrz pętli *będą powtarzane* tak długo, jak długo warunek jest spełniony. Jeśli warunek nie jest spełniony na początku, to w obu przypadkach linie te zostaną pominięte przy wykonaniu i wykonane zostaną następne (o ile takie są). Jednak jeśli warunek przy pętli `while`, początkowo spełniony, nie stanie się fałszywy w rezultacie powtarzania, to pętla będzie wykonywała się w nieskończoność[1].

Warunek przy pętli `while` jest identycznie formułowany jak przy instrukcji warunkowej `if`.

Nasz program do obliczania jak długo będziemy spłacać pożyczkę, mógłby wyglądać następująco:

```
1 k=1000; % kwota pożyczki
2 p=8;     % stopa procentowa
3 r=200;  % rata roczna
4 while k>0
5   k=k*(1+p/100)-r;
6 end%while
```

Na początek należy skomentować dlaczego w programie jest warunek `k>0`, a nie jak należałoby się spodziewać `k~=0`. To jest efekt pewnej rozbieżności między rzeczywistością a naszym modelem. Jeśli, przykładowo, płacimy ratę roczną w wysokości 200 zł a w ostatnim roku zostało nam 60 zł zadłużenia, to nie wpłacamy całej raty 200 zł, tylko jej część czyli 140 zł i nasz stan zadłużenia wynosi zero. W naszym modelu, aby niepotrzebnie nie komplikować obliczeń zakładamy, że zawsze spłacamy pełną ratę czyli 200 zł. Tak więc w tym modelu po ostatnim roku nasze (modelowe) zadłużenie wyniosłoby -140 zł. Czyli kiedy zadłużenie osiągnie zero lub stanie się ujemne to skończyliśmy spłacać pożyczkę.

Program zadziała poprawnie, ale znowu nic z niego nie wynika. Brakuje nam najistotniejszej informacji: jak długo będzie trwało spłacanie pożyczki.

W poprzednich przykładach liczbę lat pokazywał nam licznik pętli. Instrukcja pętli o nieokreślonej liczbie przebiegów nie posiada licznika pętli. Jeśli jest nam potrzebny to musimy sami go stworzyć. Czyli przed pętlą zainicjować na wartość początkową i wewnątrz pętli zmieniać stosownie do naszych potrzeb. Tego właśnie elementu potrzebujemy, aby program do symulacji spłat pożyczki stał się użyteczny.

```
1 k=1000; % kwota pożyczki
2 p=8;     % stopa procentowa
3 r=200;  % rata roczna
4 t=0;     % licznik pętli - liczba lat
5 while k>0
6   k=k*(1+p/100)-r;
7   t=t+1;    % powiększamy licznik
```

[1] Trzeba wtedy przerwać działanie programu. W większości systemów można to uzyskać przez jednoczesne przyciśnięcie klawiszy `Ctrl` i `C`.

```
8 end%while
9 disp(t);
```

Teraz w zmiennej `t` przechowujemy liczbę lat. Na końcu wypisujemy tą wartość.

Przy okazji, warto pokazać, jak zrealizować pętlę o określonej ilości przebiegów przy pomocy instrukcji `while`. Odpowiedni szkic takiej konstrukcji wyglądałby następująco:

```
1 i=1;
2 while i<10
3    ....    % instrukcje do zrealizowania
4    i=i+1;  % powiększamy licznik pętli
5 end%while
```

Przewaga pętli z instrukcją `for` polega przede wszystkim na jasności zapisu. Widząc słowo kluczowe `for`, nie ma wątpliwości, jaka to jest pętla i widać też wyraźnie ile razy się wykona. W przypadku jej realizacji przez instrukcję `while` trzeba szukać, gdzie (i w jaki sposób) jest zmieniany licznik. Do tego dochodzi ryzyko, że w ferworze programowania zapomnimy w ogóle o zmianie licznika.

Tak więc pętla o określonej ilości przebiegów nie jest niezbędna. Jednak bardzo często potrzebujemy realizować pętle o z góry zadanej ilości przebiegów, więc taka pętla jest obecna w większości języków.

Zabezpieczenia. Na koniec warto zwrócić jeszcze uwagę na kwestię nieuchronnie związaną z pętlą o nieokreślonej ilości przebiegów, mianowicie na fakt, że jeśli nie mamy gwarancji, że warunek zakończenia pętli będzie kiedyś spełniony to taka pętla będzie działała w nieskończoność. Powody, dla których warunek zakończenia mógłby nie zostać nigdy spełniony są dwojakie. Może to być zwykła pomyłka, wystarczy np. wpisać "+" zamiast "-" w wyrażeniu, w naszym przykładzie dodawać ratę zamiast ją odejmować `k=k*(1+p/100)+r;` a wtedy `k` będzie rosnące i nigdy nie będzie spełniony warunek `k<=0`.

Drugą przyczyną takiego zachowania mogą być dane zadania, nie gwarantujące zbieżności. Jeśli tym zadaniu nie dobierzemy odpowiednio raty do kwoty i odsetek, to możemy nie spłacić nigdy pożyczki. Można to pokazać następująco: jeśli dopisywane odsetki $k \cdot \frac{p}{100}$ są równe racie r, to dług jest stały i nie rośnie, ani nie maleje (ale my ciągle spłacamy!). Jeśli rata jest większa niż dopisywane odsetki to dług, szybciej lub wolniej, maleje do zera[2]. Jeśli natomiast rata jest mniejsza od odsetek, to dług, znowu szybciej lub wolniej, rośnie. Tak więc w tym przypadku warunkiem zbieżności jest $r < k \cdot \frac{p}{100}$.

W ogólności nie zawsze łatwo jest znaleźć warunek zbieżności, a często nie można go w ogóle znaleźć. Istnieje dość prosta technika, która pozwala uniknąć

[2] Widać to wyraźnie w przypadku karty kredytowej, gdzie zazwyczaj bank domaga się spłaty tzw. kwoty minimalnej, tak dobranej aby pokrywała odsetki. W przeciwnym razie zadłużenie narastałoby lawinowo.

nieskończonej pętli. Wstawia się ograniczenie na liczbę przebiegów pętli. Czyli żąda się, aby licznik pętli nie przekroczył pewnej wartości. Wartość ograniczenia licznika nie wynika z samego problemu, a raczej ze znajomości kontekstu zadania.

W przypadku pożyczki, gdzie licznikiem pętli jest wartość `t` czyli liczba lat przez które spłacamy pożyczkę, sensowną wartością graniczną może być np. 100. Jeśli w ciągu 100 lat (wystarczyłoby 50 ale zadziała też 200) nie spłacimy pożyczki, to oznacza to, że w praktyce nie spłacimy jej nigdy.

W takim przypadku nasz program wyglądałby następująco:

```
1 k=1000;  % kwota pożyczki
2 p=8;     % stopa procentowa
3 r=200;   % rata roczna
4 t=0;     % licznik pętli - liczba lat
5 while (k > 0) & ( t < 100)
6   k=k*(1+p/100)-r;
7   t=t+1;    % powiększamy licznik
8 end%while
9 disp(t);
```

Oba warunki logiczne połączone są operatorem **i**, gdyż wystarczy aby jeden nie był spełniony.

Na początku oba warunki są spełnione, zrówno kwota zadłużenia `k` jest większa od zera jak i liczba lat `t` jest mniejsza od 100. Albo spłacimy pożyczkę w rozsądnym czasie czyli `k <= 0` albo, po 100 latach, `k > 0` ale `t >= 100`.

5.2 Suma liczb z klawiatury

Rozważmy następujące zadanie: mamy daną długą kolumnę liczb, dla których mamy obliczyć wartość średnią. Wartość średnia ciągu liczb a_1 ... a_n wyraża się wzorem:

$$\bar{a} = \frac{1}{n}\sum_{i=1}^{n} a_i$$

Chcemy napisać program, który ułatwi nam to zadanie, gdyż będziemy jedynie wpisywać kolejne liczby, a program wykona za nas żmudne rachunki. Pewnym kłopotem technicznym, przynajmniej w takim wariancie sformułowania, jest problem z określeniem, kiedy kończą się dane czyli brak naturalnego znacznika końca danych (skąd program ma wiedzieć, że już wpisaliśmy wszystkie liczby?). Aby nie komplikować zadania musimy się dodatkowo umówić, że po ostatniej liczbie będzie wartość 0 jako $n+1$ wartość, która sygnalizuje koniec danych. Tym samym zakładamy, że w danych nie może być wartości 0.

Zadanie wygląda dość prosto ale mamy dwa kłopoty. Jeden, prosty, to czytanie z klawiatury a drugi, raczej skomplikowany, to organizacja programu.

Wprowadzanie liczb. Do czytania z klawiatury OCTAVE udostępnia polecenie `input`. Używa się go w następujący sposób:
`x=input("Podaj wartość x=");`
po napotkaniu takiego polecenia OCTAVE wypisuje na ekranie komunikat: `"Podaj wartość x="` i czeka, aż wpiszemy wartość i naciśniemy klawisz Enter. Wtedy ta wartość jest wpisywana do zmiennej `x`. Argumentem polecenia `input` jest dowolny ciąg znaków, który jest wypisywany na ekranie.

Organizacja programu. Mając metodę wczytywania kolejnych liczb moglibyśmy algorytm rozwiązania sformułować następująco: „wczytuj kolejne liczby a_i, dodawaj je do sumy S, a po wczytaniu zera, podziel sumę S przez ilość liczb n".

O ile samo wnętrze pętli wygląda dość prosto

```
1 ...% tu wartości początkowe
2 ....... % tu początek pętli
3   x=input("Podaj wartość x=");
4   S=S+x;
5   n=n+1;
6 .......  % tu koniec pętli
7 S/n      % wynik
```

o tyle łatwo zauważyć problem ze sformułowaniem samej pętli.

Problem jest natury „co było pierwsze jajko czy kura?" czyli w przypadku pętli `while` najpierw jest sprawdzany warunek, w tym wypadku czy wartość `x` nie jest równa zero, a potem dopiero jest wczytywany `x`. Skąd program może wiedzieć, jaką liczbę za chwilę wpiszemy?

Istnieją dwie klasy zagadnień. Jedna dla której naturalne jest sformułowanie „jeśli zaszedł określony warunek, to powtarzaj zadane instrukcje" oraz druga dla której bardziej naturalne jest sformułowanie „powtórz określone instrukcje a potem sprawdź warunek".

Przykładem pierwszej klasy jest kredyt bankowy, gdzie jeśli nasze zadłużenie nie jest zerowe to przez kolejny rok bank będzie dopisywał odsetki, a my spłacali ratę. Przykładem drugiej może być właśnie wczytywanie liczb. Najpierw trzeba wczytać liczbę, a dopiero potem można sprawdzić czy ta liczba jest taka czy inna (w tym konkretnym przypadku czy nie jest zerem).

Pętla `while` jest przykładem pętli o nieokreślonej ilości przebiegów ze sprawdzaniem warunku na początku. Taka pętla pasuje do zadania kredytu. Istnieje w OCTAVE, ale nie w Matlabie, pętla o nieokreślonej ilości przebiegów ze sprawdzaniem warunku na końcu. Pętlę taką zapisuje się jako:

```
do
      ...
      ...
until  warunek
```

Jej funkcjonowanie jest zupełnie analogiczne, poza tym, że *warunek* jest sprawdzany na końcu. W konsekwencji instrukcje wewnątrz pętli zawsze wykonają się przynajmniej raz, podczas kiedy dla pętli `while` mogą się nie wykonać ani razu.

Zadanie sumowania liczb z klawiatury bardzo łatwo jest napisać w OCTAVE, dysponując pętlą `do until`. Program wyglądałby następująco:

```
% ten program nie wykona się w  Matlabie
% tylko octave!!
S=0; % wartości początkowe
n=0;
do % tu początek pętli
  x=input("Podaj wartość x=");
  S=S+x;
  n=n+1;
until x != 0  % tu koniec pętli
S/(n-1)       % wynik
```

W programie, oprócz pętli `do until`, użyliśmy dla przykładu operatora „różny od” w postaci `!=`, gdyż OCTAVE akceptuje również taką formę.

Matlab nie posiada pętli `do until`, jak więc poradzić sobie z przypadkiem zagadnienia, kiedy najpierw trzeba coś wykonać, a dopiero później sprawdzić warunek[3]?

Są dwa sposoby, jeden prosty, ale dość prymitywny drugi nieco bardziej skomplikowany, za to elegancki.

Sposób pierwszy. Prosty sposób polega na przeniesieniu faktycznego sprawdzania warunku na koniec za pomocą dodatkowej zmiennej logicznej. Załóżmy, że nasza zmienna służąca „obejściu” nazywa się `koniec`.

```
koniec=0;
while 0==koniec
  .....
  ..... % instrukcje wewnątrz pętli
  .....
  if warunek
     koniec=1;
  end%if
end%while
```

Na początku ustawiliśmy zmienną `koniec` na wartość `0`. Formalny warunek przy pętli `while` sprawdza, czy `koniec` jest równa zero i musi być spełniony. Wtedy wykonuje się ciąg instrukcji pętli, a po nim (ale przed końcem pętli) jest instrukcja `if`, która sprawdza rzeczywisty warunek zadania. Jeśli jest on spełniony, to zmienna `koniec` jest ustawiana na jeden.

[3] Nie jest to tylko problem Matlaba. Sporo języków posiada tylko jeden wariant pętli o nieokreślonej ilości przebiegów.

W takim wariancie nasz program do obliczania średniej mógłby wyglądać następująco:

```
 1 S=0;
 2 n=0;

 3 koniec=0;
 4 while 0 == koniec
 5   x=input("Podaj wartość x=");
 6   S=S+x;
 7   n=n+1;
 8   if 0 == x
 9      koniec=1;
10   end%if
11 end%while
12 srednia=S/(n-1);
13 disp(srednia);
```

W programie tym warto zwrócić uwagę, że sumę liczb S dzielimy przez $n-1$ a nie przez n jak wynikałoby to ze wzoru na średnią. Jest to efekt tego, że znacznik końca danych (w postaci wartości zero) powiększa wartość n a przecież nie jest to wartość, która ma wpływać na średnią tylko znacznik.

Sposób drugi. Drugi sposób jest nie tyle skomplikowany, co wymaga pewnego wysiłku aby zrozumieć jak to działa.

Spróbujmy spojrzeć na różnice w działaniu pętli `while` (którą mamy dostępną w Matlabie) a pętlą DO UNTIL (która jest nam potrzebna w tym zadaniu a niedostępna). Tabela 5.1 (na str. 67) pokazuje, jak wygląda w obu przypadkach sekwencja sprawdzania warunku i wykonywania instrukcji. Porównując kolejność (która celowo w przypadku pętli DO UNTIL została przesunięta o jeden) widzimy, że działanie obu pętli różni się tym, że pętla `while` raz dodatkowo sprawdza warunek. Gdyby ten jeden raz warunek spełnić to dalej nie ma już różnicy. Z te-

`while`	DO UNTIL
warunek	
`instrukcje`	`instrukcje`
warunek	*warunek*
`instrukcje`	`instrukcje`
warunek	*warunek*
`instrukcje`	`instrukcje`
...	...

Tabela 5.1: Sekwencja wykonywania instrukcji i sprawdzania warunku dla pętli `while` i `do until`. {tab:do-while}

go pierwszego sprawdzenia wziął się nasz problem, że najpierw musimy wczytać liczbę (czyli wykonać instrukcje) a potem możemy sprawdzić co to za liczba (czyli

sprawdzić warunek). Ale gdyby tak ten jeden raz oszukać program? Sprawić, że za pierwszym razem warunek będzie spełniony w sposób sztuczny? Zastanówmy się, czy jest to w ogóle możliwe? Struktura naszego programu z użyciem pętli `while` musiałaby wyglądać następująco:

```
....% instrukcje początkowe
....% tu coś jest potrzebne
while x ~= 0
  x=input("Podaj wartość x=");
  S=S+x;
  n=n+1;
end%while
.... % instrukcje końcowe
```

Warunek `while x ~= 0` jest spełniony jeśli `x` jest różne od zera. A właściwie ile powinno wynosić `x` w tym miejscu? Przy pierwszym przejściu właściwie wartość `x` jest nieokreślona bo zmienna `x` jest przeznaczona na wartość liczby, którą za chwilę wczytamy. Czyli przed pierwszym wczytaniem może być tam cokolwiek. A jakie „cokolwiek” spełni nasz warunek przy pierwszym wejściu do pętli? Każda wartość z wyjątkiem zera. Tak więc, jeśli przed pętlą nadamy zmiennej `x` jakąś niezerową wartość, np. 1234, to przy pierwszym sprawdzeniu warunek będzie spełniony.

To „oszustwo” musi spełnić jeden warunek aby było skuteczne. Pierwszą instrukcją wewnątrz pętli musi być `x=input("Podaj wartość x=");` aby wartość `x` była wczytaną liczbą a nie fikcyjną wartością początkową.

Tak więc końcowy program mógłby wyglądać następująco:

```
S=0;
n=0;
x=pi; % fikcyjna wartość aby wejść do pętli
while x ~= 0
  x=input("Podaj wartość x=");
  S=S+x;
  n=n+1;
end%while
srednia=S/(n-1);
disp(srednia);
```

Warto w tym momencie zwrócić uwagę na pewną możliwą konwencję, którą opłaca się stosować. Jak powiedziano, w tym zadaniu wartość początkowa `x` może być dowolna, niezerowa. Pierwszą narzucającą się wartością jest 1. Jest to wartość poprawna, jednak jeśli ktoś będzie czytał nasz program (albo my sami po dłuższym czasie) a program nie będzie zawierał komentarzy, to będzie się zastanawiał jakie znaczenie ma ta jedynka. Szczególnie, że jedynka w przedostatniej linii (w mianowniku jest $n-1$) jest istotna. Pewnym ułatwieniem może być nadawanie wartości, które wyraźnie sugerują, że nie mają żadnego związku z zadaniem. Takimi wartościami mogą być 123456789 lub (jak w tym przypadku) wartość π.

Oprócz instrukcji `input` OCTAVE posiada też instrukcję `scanf` o nieco bardziej skomplikowanej składni ale pozwalającej uniknąć wpisywania znacznika końca danych.

Problem z zamianą pętli z warunkiem na końcu na pętlę z warunkiem na początku jest na tyle częsty, że wyjaśnienie go na tym prostym przykładzie było niezbędne.

5.3 Suma nieskończona

Paradoks Achillesa. W starożytnej Grecji ówcześni filozofowie szukali skomplikowanych odpowiedzi na proste pytania[4]. Jednym z takich zagadnień był paradoks Achillesa, opisany przez Zenona z Elei. Otóż jeśli wyobrazimy sobie wyścig szybkobiegacza jakim był Achilles z żółwiem, to wiadomo, że Achilles wygra. Ale jeśli dla wyrównania szans żółw wystartuje z pozycji w połowie drogi między linią startu a linią mety, to kto wtedy wygra? Wszyscy wiedzą, że i tak Achilles. Ale filozofowie rozumowali w taki sposób: aby Achilles dobiegł do tego miejsca, gdzie w chwili startu znajdował się żółw, potrzebuje trochę czasu. Ale w trakcie tego czasu żółw przesunie się (chyba słowo przebiegnie tu nie pasuje) pewien odcinek dalej. Tak więc, kiedy Achilles już dobiegnie do miejsca, w którym znajdował żółw w chwili startu, to dalej będzie ich dzielił pewien dystans. Żeby przebiec ten dystans Achilles potrzebuje pewnej ilości czasu. A w tym czasie żółw się znowu oddali o pewną odległość. Zawsze, kiedy Achilles dobiegnie do miejsca w którym znajdował się żółw, ten zdąży przejść trochę dalej. Czyli Achilles nigdy nie dogoni żółwia! Trapiło to filozofów, gdyż wiedzieli, że przecież Achilles przegoni żółwia a jednak logiczne rozumowanie mówiło co innego[5].

Napiszmy program, który symuluje wyścig Achillesa i żółwia. Załóżmy, że wyścig odbywa się na dystansie $1,0$ (jednej długości od startu do mety). Załóżmy[6], że Achilles przebiega ten dystans w $1,0$ jednostce czasu a żółw przebywa go w $100,0$ jednostkach czasu (w końcu to musi być żółw wyścigowy). Tak więc prędkość Achillesa wynosi $1,0$ jednostek prędkości a żółwia $\frac{1}{100}$ jednostek prędkości.

Jeżeli będziemy opisywali położenie obu zawodników na osi x, której początek ($x = 0$) jest na starcie a $x = 1$ jest na mecie, to w chwili startu ($t = 0$) Achilles jest na starcie $x_a(0) = 0$ a żółw jest w połowie drogi $x_z(0) = 0,5$. Prędkość Achillesa wynosi $v_a = 1$ a prędkość żółwia $v_z = \frac{1}{100}$. Położenie obu zawodników w dowolnej chwili t opisują równania:

$$x_a(t) = x_a(0) + v_a \cdot t$$

$$x_z(t) = x_z(0) + v_z \cdot t$$

Wykres położenia Achillesa i żółwia!

[4] I filozofom zostało tak do dzisiaj.

[5] Tak właśnie definiuje się *paradoks* – logiczne rozumowanie prowadzące do sprzeczności.

[6] Zmieniliśmy nieco dane w porównaniu z oryginalnym zadaniem.

W układzie (t, x) ta para równań przedstawia dwie proste. W zasadzie w tym momencie można by znaleźć punkt przecięcia obu prostych i rozwiązać zadanie. My jednak pójdziemy nieco okrężną drogą i postaramy się bezpośrednio odtworzyć rachunkowo rozumowanie Greków.

Zamiast ciągłego czasu t będziemy rozważali dyskretne chwile t_i, te w których Achilles dobiega do miejsca w którym znajdował się żółw w chwili t_{i-1}. Różnica odległości między żółwiem i Achillesem w chwili t_i wynosi

$$d(t_i) = d_i = x_z(t_i) - x_a(t_i)$$

Achilles przebiegnie dystans d_i w czasie $\Delta t_i = d_i/v_a$ i znajdzie się w punkcie $x_a(t_i + \Delta t_i) = x_a + d_i$ a po tym czasie żółw znajdzie się w punkcie

$$x_z(t_i + \Delta t_i) = x_z(t_i) + v_z \cdot \Delta t_i$$

Chwila, w której Achilles osiągnie następny punkt:

$$t_{i+1} = t_i + \Delta t_i$$

Pozostaje jeszcze kwestia jak długo mamy powtarzać te rachunki. Naturalną odpowiedzią wydaje się być: tak długo dokąd żółw wyprzedza Achillesa czyli $x_z > x_a$ lub $x_z - x_a > 0$.

{achilles.m} Znajdźmy, po jakim czasie Achilles dogoni żółwia

```
1 xa=0.0;
2 xz=0.5;
3 t=0.0;
4 while xz-xa > 0
5    d= xz -xa;
6    Dt=d/1;
7    xa=xa+1.0*Dt;
8    xz=xz+(1.0/100)*Dt;
9    disp([t, xz, xa, d, Dt]);
10   t=t+Dt;
11 end%while
12 disp(t);
```

Wyniki obliczane prze nasz program są pokazane w tabeli 5.2, gdzie pokazano dla kolejnych etapów (przebiegów pętli) i: czas t_i położenie Achillesa x_a, położenie żółwia x_z, odległość d oraz przyrost czasu Δt.

Jak wynika z tych rezultatów, kolejne dystanse maleją bardzo szybko a więc i kolejne czasy ich pokonania również gwałtownie maleją. Sumaryczny czas (t_i) rośnie coraz wolniej, tak że właściwie od pewnego momentu przestaje rosnąć.

Porównajmy wyniki, które uzyskaliśmy na drodze numerycznej z rozwiązaniem analitycznym. W kategoriach współczesnej fizyki ścisłe rozwiązanie zadania jest banalne[7]. Jeśli układ współrzędnych związemy nie z Ziemią ale umieścimy

[7] No, ale trzeba było tysięcy lat rozwoju nauki abyśmy nauczyli się rozwiązywać tak trudne dla starożytnych Greków zadania niemal w pamięci.

i	t	x_z	x_a	d	Δt
1	0.000000	0.505000	0.500000	0.5	0.5
2	0.500000	0.505050	0.505000	0.005	0.005
3	0.505000	0.505050	0.505050	5e-05	5e-05
4	0.505050	0.505051	0.505050	5e-07	5e-07
5	0.505050	0.505051	0.505051	5e-09	5e-09
6	0.505051	0.505051	0.505051	5e-11	5e-11
7	0.505051	0.505051	0.505051	5.00044e-13	5.00044e-13
8	0.505051	0.505051	0.505051	4.996e-15	4.996e-15

{tab:achilles} Tabela 5.2: Achilles i żółw

obserwatora na żółwiu[8], to z punktu widzenia obserwatora żółw stoi w miejscu natomiast Achilles będzie się zbliżał do niego z prędkością $1 - \frac{1}{100} = \frac{99}{100}$. Dzielący ich na początku dystans $0,5$ jednostek pokona więc w czasie

$$\frac{0,5}{1-\frac{1}{100}} = 0,505(05)$$

Suma szeregu geometrycznego wyraża się wzorem:

$$S = \sum_{i=1}^{\infty} aq^{n-1} = a + aq + aq^2 + aq^3 + \ldots = \frac{a}{1-q}$$

Jakkolwiek w tym momencie wydaje się być to zupełnie niejasne co wspólnego ma suma szeregu geometrycznego z naszym zadaniem, porównajmy czas w jakim Achilles dopędzi żółwia ze wzorem na sumę szeregu geometrycznego. Widzimy, że prawa strona wzoru odpowiada rozwiązaniu naszego zadania, jeśli wstawimy $a = 0,5$ i $q = 1/100$.

Skoro prawa strona wzoru na sumę szeregu geometrycznego odpowiada otrzymanemu przez nas wynikowi to nasz wynik możemy też otrzymać stosując lewą stronę czyli dodając kolejno $0,5\ 0,5/100\ 0,5/100^2$ itd.

Jak łatwo zauważyć ten ciąg wartości odpowada kolejnym Δt jak to widać w tabeli 5.2

Tak więc w rzeczywistości rozwiązaliśmy zadanie obliczenia sumy szeregu nieskończonego. W kategoriach matematyki możemy to zapisać:

$$t = \Delta t_1 + \Delta t_2 + \Delta t_3 + \ldots = \sum_{i=0}^{\infty} \Delta t_i$$

Dla Greków trudne do zrozumienia było, że nieskończona suma dodatnich liczb może dać skończoną wartość. Dopiero rozważania Newtona i Leibnitza na temat ciągłości uwolniły nas od tego paradoksu.

[8] Obserwator musi być idealny, czyli w tym przypadku nieważki i nie wywołujący oporu powietrza aby nie zakłócać ruchu żółwia.

Suma szeregu nieskończonego. Skoro, jak nam się wydaje, umiemy znajdować sumę nieskończonego szeregu to rozważmy szereg matematyczny:

$$S = 1 + \frac{1}{4} + \frac{1}{9} + \frac{1}{16} + \frac{1}{25} + \ldots = \sum_{n=1}^{\infty} \frac{1}{n^2}$$

W zmiennej `S` umieścimy sumę już dodanych wyrazów a na początku `S` musi wynosić zero. Wewnątrz pętli będziemy obliczali wartość kolejnego a_n i dodawali do sumy `S`.

```
1 S=0;
2 ....
3 while ...
4   an=1/(n*n);
5   S=S+an;
6   ....
7 end%while
```

Jednak wartość a_n zależy jawnie od n, tak więc musimy użyć licznika pętli w którym będziemy przechowywali wartość n. Na początku `n=1` a przy każdym przebiegu pętli musimy zwiększyć n o jeden.

```
1 S=0;
2 n=1;
3 ....
4 while ...
5   an=1/(n*n);
6   S=S+an;
7   n=n+1;
8 end%while
```

Tu dochodzimy do zasadniczej trudności, kiedy właściwie mamy przerwać sumowanie? Czyli jakie powinno być kryterium dla pętli `while`?

Pierwsze co się może nasunąć to sumować tak długo jak a_n jest większe od zera. Tylko, że z punktu widzenia matematyki a_n nigdy nie osiągnie zera! Numerycznie co prawda nie uda nam się uzyskać liczby mniejszej niż ok. 10^{-308} ale jeśli pierwszy wyraz a_1 jest 1 to $S_i > 1$ a dodawanie czegokolwiek mniejszego od zera maszynowego (por. str. 12) do `S` jest tylko stratą czasu.

W przypadku programu odtwarzającego wyścig żółwia z Achillesem ze str. 70 problem z kryterium zakończenia sumowania nie wystąpił, gdyż, mimo że sumowaliśmy Δt to kryterium zakończenia było wyrażone poprzez położenie x.

Dla abstrakcyjnego matematycznego szeregu nie mamy naturalnego kryterium zakończenia. Wiemy, że trzeba sumować pewną ilość wyrazów taką, aby z jednej strony wynik końcowy S nie był obarczony błędem a z drugiej strony aby nie dodawać wyrazów, które ze względu na arytmetykę nie mogą wpłynąć na S. Inaczej mówiąc sumujemy tak długo aż uznamy, że już wystarczy.

Pojawia się nieco sztuczna wartość *tolerancji*, czyli takiej wartości, że jeśli wyrazy a_n będą mniejsze od niej to przerywamy sumowanie.

Nie istnieje żadne matematyczne uzasadnienie takiego postępowania[9] ani oszacowanie jaka ma być wartość tolerancji. Należy ją dobrać tak aby wynik był dostatecznie dobry. W naszym przykładzie przyjmiemy jako wartość graniczną 10^{-7}.

W konsekwencji nasze kryterium zakończenia wyrażone przez tolerancję wyglądałoby następująco:

```
1 ...
2 tol=1e-7;
3 while an> tol
4   ...
5 end%while
```

Ponieważ kryterium jest zależne od `an`, przy pierwszym przebiegu trzeba je zainicjować na jakąkolwiek wartość gwarantującą wejście do pętli np. `an=1234567`.

Ostatecznie nasz program będzie wyglądał następująco.

```
1 S=0;
2 n=1;
3 an=1234567;
4 tol=1e-7;
5 while an> tol
6   an=1/(n*n);
7   S=S+an;
8   n=n+1;
9 end%while
10 disp(S);
```

Należy z całą mocą podkreślić, że łudząco podobne zadania: sumy szeregu skończonego:

$$S = \sum_{n=1}^{N} a_n$$

i sumy szeregu nieskończonego:

$$S = \sum_{n=1}^{\infty} a_n$$

są, mimo niemal identycznego zapisu zupełnie odmienne zarówno z punktu widzenia matematyki jak i z punktu widzenia progamowania.

W przypadku programu, pierwsze zadanie narzuca pętlę o określonej ilości przebiegów (`for`) podczas, kiedy drugie narzuca pętlę o nieokreślonej ilości przebiegów (`while`).

[9] Dociekliwi mogliby argumentować, że suma skończonej ilości wyrazów szeregu i suma pozostałych muszą być skończone ale dalej nie rozstrzyga to ile wyrazów trzeba wysumować aby dostać dobre oszacowanie wyniku.

Szereg harmoniczny. Gdybyśmy spróbowali obliczyć sumę szeregu:

$$S = \sum_{n=1}^{\infty} \frac{1}{n}$$

to nasz program wymagałby jedynie małej modyfikacji, uwzględniającej inny przepis na wyraz a_n, który w tym wypadku wynosi

$$a_n = \frac{1}{n}$$

```
 1 S=0;
 2 n=1;
 3 an=12345678;
 4 tol=1e-7;
 5 while an> tol
 6   an=1/n;
 7   S=S+an;
 8   n=n+1;
 9 end%while
10 disp(S);
```

Jak łatwo się przekonać, program obliczy wartość sumy takiego szeregu i wypisze (dla tolerancji jak w przykładzie) wartość: $12,090$.

W tym momencie narzuca się pytanie jak to możliwe, przecież szereg harmoniczny jest rozbieżny, czyli jego suma zmierza do nieskończoności?

Dochodzimy do dość istotnej kwestii. To, co nazywamy programem do obliczania sumy szeregu nieskończonego precyzyjniej należałoby nazwać programem szacującym przyliżenie granicy takiego szeregu, *przy milczącym założeniu, że szereg ten jest w ogóle zbieżny*.

Z tego wynikałoby, że powinniśmy najpierw udowodnić, że szereg jest zbieżny a dopiero potem można próbować liczyć oszacowanie granicy. Tak jednak nie robimy. Dowód zbieżności umiemy przeprowadzić dla stosunkowo prostych szeregów. Często używamy komputera wtedy, kiedy szereg jest tak skomplikowany, że jedyne co jesteśmy w stanie zrobić to liczyć (zdarza się, że wielkim wysiłkiem) jego wyrazy. Czasami zamiast dowodu zbieżności posiłkujemy się interpretacją fizyczną. W przykładzie bezpośredniej symulacji zadania z Achillesem, nie wnikając w subtelności matematyczne obliczyliśmy granicę, gdyż wiemy, że Achilles musi prześcignąć żółwia. Podobnie, gdyby ugięcie mostu wyrażało się przez jakiś szereg, to z góry wiadomo, że ugięcie będzie skończone[10]. Ugięcie może być bardzo duże, może być większe niż dopuszczamy ale skończone, czyli taki szereg musi być zbieżny. W przypadkach, kiedy nie mamy ani wsparcia ze strony matematyki (w postaci dowodu zbieżności) ani fizyki (nie wiemy, czy na pewno musi być

[10] Na ogół nasze modele nie potrafią odtworzyć bezpośrednio zerwania się mostu. Modele zdolne taką sytuację odtworzyć to zupełnie inna klasa.

zbieżny) istnieją pewne techniki, które lepiej czy gorzej podpowiadają nam czy przypadkiem szereg nie jest rozbieżny. W praktyce najczęściej zapominamy o tych subtelnościach i po prostu liczymy granicę szeregu mając albo jakieś przesłanki albo nadzieję, że to jest szereg zbieżny. Należy jednak zdawać sobie sprawę, że w pewnych przypadkach możemy wpaść w pułapkę.

Szereg dany rekurencyjnie. Rozważmy szereg matematyczny:

$$S = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots = \sum_{n=0}^{\infty} \frac{1}{2^n}$$

W zasadzie można by znowu przepisać poprzedni program, wstawiając jedynie inne wyrażenie na obliczanie a_n:

```
an=1/(2^n);
```

Program obliczający ten szereg napisany tak jak poprzednie działałby poprawnie i dawałby właściwe wyniki.

To, co chcemy pokazać na przykładzie tego programu, to tzw. optymalizacja programu, czyli taka zmiana programu aby działał szybciej. Jest to dość kontrowersyjne zagadnienie, gdyż w programowaniu uważa się, że „przedwczesna optymalizacja jest źródłem wszelkiego zła". Obecnie uważamy, że najważniejszą rzeczą jest poprawność programu (brak błędów) a czytelny ale na ogół nieoptymalny zapis bardzo ułatwia uniknięcie pomyłek. Tak więc, należy raczej pisać przejrzyste programy i unikać optymalizacji dokąd nie okaże się, że dany fragment istotnie spowalnia działanie programu. W szczególności, praktycznie wszystkie analizowane tutaj przykłady są bardzo proste i wykonują się błyskawicznie. Różnica czasu wykonania wersji „szybszej" i „wolniejszej" będzie niezauważalna, więc poprawianie szybkości jest tutaj jedynie ćwiczeniem. Tym niemniej warto pokazać na czym to polega i że program po optymalizacji może wyglądać zupełnie inaczej.

We wzorze na n-ty wyraz ciągu pojawia się wyrażenie 2^n. Gdyby wykonywać potęgowanie 2^n poprzez mnożenie 2 przez siebie $n-1$ razy (w Rozdz. 7 pokażemy jak to można zrobić znacznie szybciej), to przy obliczaniu wyrazu a_2 musimy wykonać 1 mnożenie, dla wyrazu a_3 2 mnożenia a dla a_n $n-1$ mnożeń. Pamiętając zadanie Gaussa 4.2 łatwo obliczyć, że aby obliczyć sumę pierwszych n (pominąwszy zerowy i pierwszy) wyrazów na obliczanie jedynie znaków poszczególnych składników należy wykonać $\frac{n(n-1)}{2}$ mnożeń. Przykładowo, przy sumowaniu tysiąca składników[11] ilość mnożeń poświęconych na obliczanie mianowników jest rzędu pół miliona. Bez względu na to czy to jest dużo czy mało, to wszystko są operacje niepotrzebne.

Tymczasem, warto zauważyć, że wyraz ciągu a_n można wyrazić rekurencyjnie:

$$a_{n+1} = \frac{1}{2} a_n \qquad a_0 = 1$$

[11] Wyrazy tego konkretnego szeregu maleją błyskawicznie, więc na pewno nie ma potrzeby obliczania tysiąca składników ale nie każdy szereg tak szybko się zbiega.

Oznacza to, że zamiast potęgowania możemy dzielić wyraz przez 2.

Nieco zmienia to postać programu:

```
1 S=0;
2 an=1;
3 tol=1e-7;
4 while an> tol
5   S=S+an;
6   an=an/2;
7 end%while
8 disp(S);
```

W przypadku użycia wzoru rekurencyjnego wartość początkowa nie jest wartością fikcyjną i musi być równa a_0 czyli 1.

Warto też zauważyć, że w tym wypadku nie potrzeba używać licznika pętli bo a_n, przy wykorzystaniu rekurencji, nie zależy jawnie od n.

Szereg znakozmienny [12]**.** Rozważmy szereg:

$$S = \sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{2n-1} = \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \dots$$

Jest to szereg dosyć podobny do harmonicznego[13], z tą różnicą, że w szeregu harmonicznym wszystkie wyrazy dodajemy a tutaj co drugi odejmujemy. Skutek tej niewielkiej zmiany jest dość zasadniczy, nasz szereg jest zbieżny.

Równie interesująca jest jego granica, gdyż zmierza on do $\pi/4$. Jeśli więc wynik pomnożymy przez cztery, to powinniśmy otrzymać przybliżenie liczby π.

W zasadzie moglibyśmy napisać program obliczający sumę tego szeregu, który byłby niemal kopią ogólngo programu, z odpowiednio zmodyfikowaną linią gdzie oblicza się wyraz a_n:

```
1 S=0;
2 an=98765;
3 tol=1e-5;
4 n=1;
5 while abs(an) > tol
6    an=(-1)^(n+1)/(2*n-1);
7    S=S+an;
8    n=n+1;
9 end%while
```

[12] Szereg znakozmienny to każdy szereg z wyrazami różnych znaków. Tutaj, dla zwartości zapisu będziemy nazywali szeregiem znakozmiennym nasz konkretny przykład.

[13] To podobieństwo nie jest tak bardzo oczywiste bo tutaj mamy tylko wyrazy nieparzyste. Jednak jeśli myślowo rozdzielimy szereg harmoniczny na dwa szeregi: z nieparzystymi i z parzystymi wyrazami, to gdyby oba były zbieżne, to ich suma byłaby skończona. Uzasadnienie, że $\sum \frac{1}{2n} = \frac{1}{2}\sum \frac{1}{n}$ nie wymaga komentarza. Natomiast skoro $\frac{1}{2n-1} > \frac{1}{2n}$ więc na mocy kryterium porównawczego suma szeregu wyrazów nieparzystych zmierza do nieskończoności. A więc oba są rozbieżne.

```
10 disp(4*S);
```

Podstawowa i bardzo ważna różnica w stosunku do poprzednich programów jest taka, że przy kryterium zakończenia iteracji pojawia się wartość bezwzględna. Jest ona tutaj niezbędna, gdyż w przeciwieństwie do poprzednich przykładów, gdzie zawsze wyrazy były dodatnie, tutaj wyrazy są raz dodatnie a raz ujemne. Bez wartości bezwzględnej już drugi wyraz spełniałby warunek `an < tol` bo liczba ujemna jest mniejsza od dowolnej dodatniej, więc zakończylibyśmy sumowanie na pierwszym wyrazie i otrzymany wynik 4 miałby niewiele wspólnego z π.

Z tego wynikałoby, że jeśli jest potrzeba, to należy umieszczać wartość bezwzględną w warunku. Raczej odwrotnie. Należy z zasady ją tam zawsze umieszczać, za wyjątkiem przypadków, kiedy jej tam być nie może.

i	4S	π-4S
1	4.000000	0.858407
2	2.666667	-0.474926
3	3.466667	0.325074
4	2.895238	-0.246355
5	3.339683	0.19809
6	2.976046	-0.165546
7	3.283738	0.142146
8	3.017072	-0.124521
9	3.252366	0.110773
10	3.041840	-0.099753
...	...	...
100	3.131593	-0.00999975
...	...	...
1000	3.140593	-0.001
...	...	...
10000	3.141493	-0.0001
...	...	...
20000	3.141543	-5e-05
...	...	...
30000	3.141559	-3.33333e-05
...	...	...
40000	3.141568	-2.5e-05
...	...	...
49999	3.141613	2.00004e-05
50000	3.141573	-2e-05
50001	3.141613	1.99996e-05

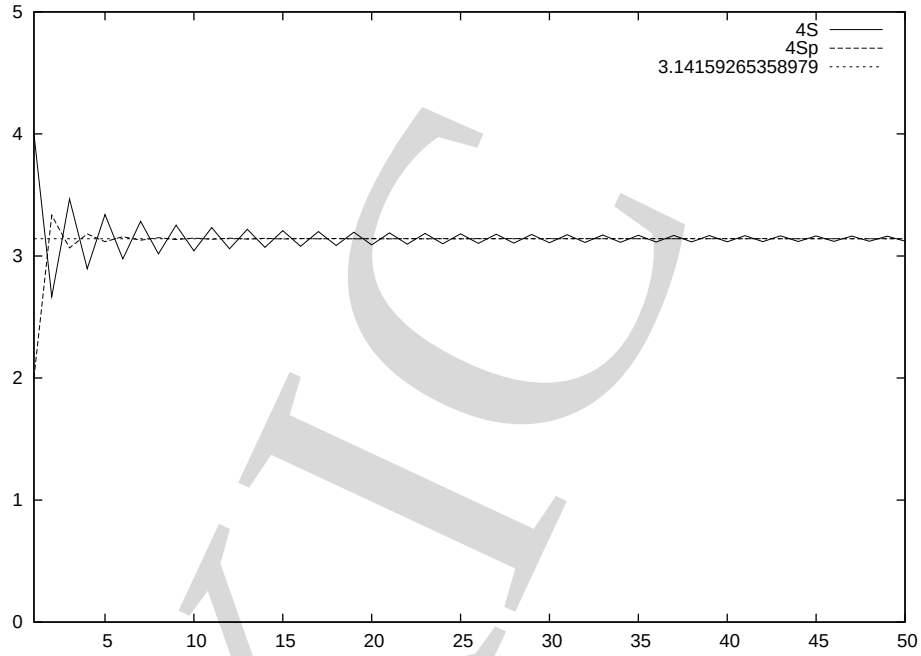
Tabela 5.3: Zbieżność szeregu znakozmiennego. {tab:znak}

W tabeli 5.3 pokazano wyniki działania programu dla wybranych kroków. Program kończy działanie na 50001 kroku. Jak widać wartości bardzo wolno zmierzają

do π. Jest to przykład, że nie wszystkie szeregi dają dobre oszacowanie po niewielkiej liczbie wyrazów. Tutaj, nawet po 50 tysiącach wyrazów możemy jedynie stwierdzić, że rozwinięcie dziesiętne liczby π jest między 3,1415 a 3,1416 czyli mamy dokładne jedynie cztery cyfry znaczące.

Jest to przykład szeregu wolno zbieżnego.

Warte jest też uwagi, że oszacowanie $4S$ jest raz z nadmiarem a raz niedomiarem. Jest to dokładniej pokazane na Rys. 5.1, gdzie linią ciągłą połączono punk-



{fig:conv}

Rysunek 5.1: Wykres zbieżności szeregu.

ty odpowiadające kolejnym wartościom $4S_n$. Linią składającą się z samych kropek pokazano stałą wartość rozwinięcia dziesiętnego π, która powinna być linią do której zmierza rozwiązanie. Jak widać łamana pokazująca kolejne przybliżenia podlega wygasającym oscylacjom.

To sugeruje, że szybszą zbieżność otrzymalibyśmy biorąc, jako oszacowanie π średnią arytmetyczną z dwu ostatnich wartości $4S$. Taka łamana odpowiadająca kolejnym średnim jest pokazana na Rys. 5.1 linią przerywaną. Jak widać zmierza ona znacznie szybciej do stałej π.

Przy okazji tego programu, jako że tutaj trzeba policzyć tysiące wyrazów szeregu, większy sens mają rozważania dotyczące optymalizacji. We wzorze na n-ty wyraz ciągu pojawia się wyrażenie $(-1)^{(n+1)}$. Jest to matematyczny sposób zapisu zmiany znaku. Z punktu widzenia matematyki jest to sposób elegancki ale w programowaniu może być kosztowny. Gdyby wykonywać potęgowanie x^a poprzez mnożenie x przez siebie $a-1$ razy (w Rozdz. 7 pokażemy jak to można zrobić znacznie szybciej), to przy obliczaniu znaku a_1 musimy wykonać 1 mnożenie, dla

znaku a_2 2 mnożenia a dla znaku a_n n mnożeń. Pamiętając zadanie Gaussa 4.2 łatwo obliczyć, że aby obliczyć sumę pierwszych n wyrazów na obliczanie jedynie znaków poszczególnych składników należy wykonać $\frac{n(n-1)}{2}$ mnożeń. Przykładowo, przy sumowaniu 50 tysięcy składników ilość mnożeń poświęconych na znaki jest rzędu 250 milionów.

Znowu, bez względu na to czy to jest dużo czy mało, to wszystko są operacje niepotrzebne. Potęgowanie jest tutaj tylko po to, aby uzyskać zwarty zapis matematyczny. Potrzebujemy jedynie zagwarantować aby przy wyrazach nieparzystych znak był dodatni a przy parzystych był ujemny.

Do sprawdzenia czy liczba jest parzysta czy też nieparzysta można użyć dzielenia *modulo*. Dzielenie *modulo* zwraca resztę z całkowitego dzielenia jednej liczby przez drugą. W OCTAVE dzielenie *modulo* jest realizowane przez funkcję `mod`, która ma dwa argumenty. Przykładowo `mod(p,2)` zwróci `0` jeśli wartość zmiennej `p` jest podzielna przez 2 (parzysta) albo 1 jeśli jest niepodzielna czyli nieparzysta.

W naszym przypadku moglibyśmy wykorzystać tę funkcję do obliczania wyrazu `an`

```
1   ....
2   an=1/n;
3   if 1==mod(n+1,2)
4      an=-an;
5   end%if
6   ....
```

Sposób jest dość skuteczny, chociaż wcale może nie być jasne, dlaczego użycie funkcji `mod()` ma być lepsze niż potęgowanie.

Istnieje jeszcze prostszy sposób na uwzględnienie znaku. Ciąg zdefiniowany rekurencyjnie:

$$z_{n+1} = -1 \cdot z_n \qquad z_1 = 1$$

generuje wartości:

$$1 \quad -1 \quad 1 \quad -1 \quad 1 \quad -1 \quad \ldots$$

a wtedy wyrazy naszego ciągu można zapisać:

$$a_n = \frac{z_n}{2n-1}$$

Korzystając z ciągu z_n nasz program wyglądałby następująco:

```
1 S=0;
2 an=pi;
3 tol=1e-5;
4 n=1;
5 znak=1;
6 while abs(an) > tol
7    an=znak/(2*n-1);
8    S=S+an;
```

```
 9    n=n+1;
10    znak= -znak;
11 end%while
```

5.4 Granica ciągu

Rozważmy podobne zagadnienie do sumy szeregu nieskończonego w postaci granicy ciągu.

Podobieństwo obu zagadnień wynika z definicji sumy szeregu nieskończonego. Definiuje się ciąg sum częściowych

$$S_n = \sum_{i=p}^{n} a_i$$

czyli skończoną sumę wszystkich wyrazów, od początkowego p (najczęściej od 0 lub 1) do zadanego n:

$$S_1 = a_1$$

$$S_2 = S_1 + a_2$$

$$S_3 = S_2 + a_3$$

$$S_n = S_{n-1} + a_n$$

Sumą szeregu nieskończonego S nazywamy granicę ciągu sum częściowych S_n, czyli

$$S = \lim_{n\to\infty} S_n$$

Ponieważ już wiemy, jak znajdować sumę szeregu nieskończonego S, więc przy okazji rozwiązywaliśmy zadanie znajdowania granicy ciągu S_n.

W przypadku szeregu rozumowaliśmy w ten sposób: jeśli pewien wyraz a_n jest już dostatecznie mały to ten wyraz i wszystkie następne nie wpłyną znacząco na wartość sumy, więc możemy przerwać dodawanie.

To samo ale w kategoriach ciągu sum częściowych S_n należałoby sformułować: jeśli różnica między kolejnymi wyrazami ciągu (sum częściowych) S_n i S_{n-1} jest dostatecznie mała to nie opłaca się obliczać poprawek wyrazu, w przybliżeniu jesteśmy w granicy.

Jeśli dla zadanego ciągu c_n zdefiniujemy:

$$a_1 = c_1 - 0$$

$$a_2 = c_2 - c_1$$

$$a_3 = c_3 - c_2$$

itd., czyli ogólnie:

$$a_n = c_n - c_{n-1}$$

to ciąg sum częściowych S_i szeregu $\sum a_n$ będzie tożsamościowo równy ciągowi c_n.

Czyli zamiast rozwiązywać zadanie granicy ciągu moglibyśmy szukać sumy odpowiednio spreparowanego szeregu.

W tym ujęciu zadanie sumy szeregu niewiele różni się od zadania granicy ciągu. Jednak w praktycznej implementacji jest istotna zmiana. Różnica między kolejnymi wyrazami ciągu (sum częściowych) S_n i S_{n-1} jest dana jawnie jako a_n, więc sprawdzamy czy a_n (dokładniej $|a_n|$) jest dostatecznie małe. W przypadku granicy ciągu różnicę kolejnych wyrazów $|c_n - c_{n-1}|$ musimy sobie sami obliczyć. I nie byłoby to trudne zadanie, gdyby nie fakt, że w jednym przebiegu pętli obliczamy tylko jedną wartość c_n. Dodatkowo, ponieważ musimy obliczać różnicę wyrazów nie możemy skorzystać z uproszczenia, które stosowaliśmy poprzednio (np. przy szeregu danym rekurencyjnie), gdzie używaliśmy jednej zmiennej do przechowywania zarówno poprzedniego jak i następnego wyrazu. Tu na etapie obliczenia różnicy musimy mieć obie wielkości a więc musimy je przechowywać osobno.

Najprostszym rozwiązaniem wydaje się obliczanie w jednym przebiegu pętli dwu wyrazów: c_n i c_{n-1}. Jest to rozwiązanie atrakcyjne ale wymaga podwójnej ilości obliczeń. Nie ma to żadnego znaczenia w prostych przypadkach (a takie tutaj analizujemy), jednak kiedy wyjdziemy poza proste przykłady akademickie i zaczniemy szukać granicy ciągu, którego jeden wyraz może wymagać godzinnych obliczeń to jest to rozwiązanie nie do przyjęcia.

W pierwszym przebiegu musimy obliczać wyrazy a_2 i a_1, w drugim a_3 i a_2, w trzecim a_4 i a_3. Widać, że w kolejnym przebiegu obliczamy wyraz poprzedni, który poprzednio obliczaliśmy jako bieżący.

Sposobem, który pozwala uniknąć podwójnego obliczania wyrazów jest zapamiętanie obliczonego wyrazu i w następnym przebiegu użycie go jako wyrazu poprzedniego. Nie kosztuje to żadnych obliczeń a jedynie dodatkową zmienną, w której przechowujemy poprzedni wyraz ciągu.

Rozważmy zadanie na przykładzie ciągu:

$$c_n = \left(1 + \frac{1}{n}\right)^n$$

Oznaczenia są sprawą drugorzędną ale przejrzysty system oznaczeń ułatwia zrozumienie zadania. Oznaczmy więc przez `cn` bieżący (n-ty) wyraz ciągu, przez `cp` poprzedni ($n-1$) wyraz. Różnicę bieżącego `cn` i poprzedniego `cp` oznaczymy przez `r`.

Przy tych oznaczeniach koncepcja budowy programu wyglądałaby następująco:

```
1  ....
2  while abs(r) > tol
3     cn=(1+1/n)^n;
4     r= cn - cp;
5     n=n+1;
6     .....
```

```
7   end%while
```

W tej wersji koncepcja jest jasna, obliczmy wyraz bieżący, obliczamy różnicę i powiększamy `n` aby `cn` zmieniało się. Natomiast widać, że zmienna `cp` nie zmienia swojej wartości, taka jak była na początku (tu nie jest to pokazane) taka pozostaje. W konsekwencji, w zmiennej `cp` przechowujemy pierwszy wyraz a zmienna `r` zawiera różnicę między bieżącym i **pierwszym** wyrazem ciągu.

Aby tego uniknąć trzeba aktualizować zmienną `cp`. Ta aktualizacja musi nastąpić **po** obliczeniu różnicy. Tak więc nasz szkic wyglądałby następująco:

```
1   ....
2   while abs(r) > tol
3       cn=(1+1/n)^n;
4       r= cn - cp;
5       cp=cn;
6       n=n+1;
7   end%while
```

Gdyby pomiędzy obliczeniem `cn` a aktualizacją `cp` nie było linii `r=cn-cp` a więc nasz fragment wyglądałby:

```
1       cn=(1+1/n)^n;
2       %  gdyby tu nie było linii
3       cp=cn;
```

to moglibyśmy nie używać w ogóle zmiennej `cn` gdyż efekt byłby identyczny jak linia:
`cp=(1+1/n)^n;`
Obecność **między** tymi dwoma liniami wyrażenia `r=cn-cp` zmusza nas do użycia dwu zmiennych i określonej kolejności obliczania.

Pozostaje nam dopisać początek programu. Na początku musi znaleźć się wartość początkowa `n`[14], inicjalizacja wartości różnicy `r` na fikcyjną wartość, która zagwarantuje wejście do pętli oraz zainicjowanie wartości `cp`. To ostatnie sprawia pewne kłopoty.

Najbardziej naturalne wydaje się obliczenie wartości c_1, wpisanie jej jako `cp` i rozpoczęcie obliczania od $n = 2$.

```
1   ....
2   cp=2; % wartość wyrazu ciągu dla n=1
3   n=2;  % zaczynamy od następnego wyrazu
4   while ....
5       .....
6   end%while
```

[14] W przypadku ciągu – inaczej niż w przypadku szeregu, gdzie musimy zacząć od dolnej granicy sumowania – nie musimy zaczynać od 1. Równie dobrze można zacząć obliczenia od wyrazu c_{1000}.

Nie jest to jednak rozwiązanie pozbawione wad. Często mylimy się przy obliczaniu czegoś w pamięci. Bardziej pewne wydawałoby się rozwiązanie tego w postaci:

```
1   r=1234;
2   n=1;
3   cp=(1+1/n)^n;
4   n=n+1;
5   while abs(r)>1e-7
6     cn=(1+1/n)^n;
7     r=cn-cp;
8     cp=cn;
9     n=n+1;
10  end%while
11  disp(cn);
```

Skutek takiej budowy jest równoważny ale nie mamy szansy pomylić się przy obliczaniu `cp`.

Jednak i takie rozwiązanie nie jest idealne. W programie mamy dwie linie, które zawierają (dokładniej: powinny zawierać) identyczne wyrażenie, na początku przypisanie wartości `cp` i wewnątrz pętli przypisanie `cn`. Zazwyczaj staramy się unikać powtarzania identycznych fragmentów kodu bo jeśli zechcemy zmodyfikować program aby obliczał granicę innego ciągu to łatwo zapomnieć o tym, że trzeba zmiany wprowadzić w obu miejscach. Warto też zwrócić uwagę, że rozważamy tu najprostsze ciągi, których wyraz możemy obliczyć w jednej linijce. W praktycznych zagadnieniach obliczenie wyrazu może wymagać wielu linii a wtedy trzeba je wpisywać podwójnie.

Inną drogą do rozwiązania jest zauważenie, że nawet gdybyśmy wpisali błędną wartość początkową `cp` to ma to wpływ na pierwsze dwa przebiegi pętli. Przy pierwszym wartość `r` będzie błędna i może wpłynąć to na drugi przebieg, gdyż przy braku szczęścia możemy nie spełnić warunku wejścia do drugiej pętli. Wiedząc o tym, możemy nadać zmiennej `cp` jakąkolwiek (fikcyjną) wartość ale zmusić program aby wykonał co najmniej dwie iteracje:

```
1   r=1234;
2   n=1;
3   cp=45679;
4   while (abs(r)>1e-7) | ( n<3 )
5     cn=(1+1/n)^n;
6     r=cn-cp;
7     cp=cn;
8     n=n+1;
9   end%while
10  disp(cn);
```

W sumie mamy trzy możliwości rozwiązania początku ale żadna nie jest idealna.

Na koniec pokażemy możliwość innej organizacji tego samego programu, pozostawiając to do samodzielnej analizy.

```
r=1234;
n=1;
cn=(1+1/n)^n;
while abs(r)>1e-7
  cp=cn;
  n=n+1;
  cn=(1+1/n)^n;
  r=cn-cp;
end%while
disp(cn);
```

5.4.1 Pierwiastek kwadratowy.

{sec:sqrt}

Ciąg dany rekurencyjnie:

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{c}{x_n}\right)$$

Do czego zmierza ten ciąg? Odpowiedź na to pytanie jest dwuczęściowa, najpierw należałoby udowodnić, że jest on zbieżny a potem pokazać jaka jest granica tego ciągu. Część pierwszą pominiemy, stwierdzając, że można udowodnić zbieżność. Skupimy się na części drugiej. Aby znaleźć granicę tego ciągu zauważmy, że dla $n \to \infty$ wyraz x_n zmierza do pewnej granicy g czyli $x_n \to g$. A do czego zmierza x_{n+1}? Też musi zmierzać do g. W takim razie musi zachodzić:

$$g = \frac{1}{2}\left(g + \frac{c}{g}\right)$$

Mnożąc obie strony przez $2g$ dostajemy: $2g^2 = g^2 + c$. Przenosząc g^2 na lewą stronę mamy: $g^2 = c$ a więc

$$g = \sqrt{c}$$

```
format long;
c=2;
x=pi;
xp=0;
while abs(x-xp) > 1e-6
  xp=x;
  x=(x+c/x)/2;
end%while
disp(x);
```

Tabela 5.4 pokazuje przebieg obliczeń programu.

W pierwszej kolumnie podany jest krok `i`, w drugiej pokazano kolejne przybliżenia wartości pierwiastka. W trzeciej pokazano kwadrat kolejnych przybliżeń, które jak widać szybko się poprawiają aby w piątym kroku dać błąd na tyle mały, że kwadrat wychodzi niemal dokładnie 2,0. W ostatniej kolumnie pokazano różnicę między przybliżeniem a wartością ścisłą.

i	x_i	$(x_i)^2$	$x_i - \sqrt{2}$
1	1.8891062130	3.5687222839	4.748927e-01
2	1.4739039673	2.1723929048	5.969040e-02
3	1.4154222383	2.0034201126	1.208676e-03
4	1.4142140784	2.0000014597	5.16064e-07
5	1.4142135624	2.0000000000	9.39249e-14

Tabela 5.4: Obliczanie $\sqrt{2}$. {tab:sqrt}

W pięciu krokach dotarliśmy do bardzo dokładnego oszacowania $\sqrt{2}$. To sformułowanie daje bardzo szybką zbieżność do rozwiązania dokładnego[15]. W szczególności widać to po ostatniej kolumnie, gdzie jest błąd rozwiązania. Widać, że niemal dokładnie błąd w następnym kroku jest szacowany poprzez kwadrat błędu w poprzednim. Mówimy w takim wypadku o *kwadratowym tempie zbieżności algorytmu*. Jest to jedna z najlepszych szybkości zbieżności algorytmów, które możemy osiągnąć[16].

Oprócz tego, że bardzo szybko dostajemy dobry wynik, sformułowanie to ma dość wyjątkową a bardzo pożądaną w praktyce cechę. Niezależnie od tego jak złe będzie początkowe przybliżenie x_0 (ale $x_0 > 0$) to i tak dojdziemy do właściwego rozwiązania, co najwyżej będzie to wymagało więcej iteracji. Można się o tym przekonać zmieniając w programie początkową wartość `x=1.0e12;`.

Jest to bardzo ważna cecha algorytmu i jest to związane z pojęciem *stabilności algorytmu*. Jest to dosyć trudne pojęcie i nie będziemy go tutaj analizować. Jedynie zasygnalizujemy istotną konsekwencję. Jeśli nasz ciąg jest zbieżny niezależnie od wartości początkowej to wszelkie błędy zaokrągleń, które nieuchronnie pojawiają się na etapach pośrednich można traktować jak zaburzenie początkowego przybliżenia, więc nie wpłyną na wynik końcowy. W odróżnieniu od sumy szeregu, gdzie wcale nie mamy gwarancji, że zaokrąglenia pojawiające się przy obliczaniu poszczególnych wyrazów nie skumulują się i nie dadzą granicy nieco różnej od wartości do której zmierza szereg.

W ramach podsumowania zamieszczamy ten program zamknięty w funkcję:

```
1 function y=pierw(c)
2   x=c;
3   xp=0;
4   while abs(x-xp) > 1e-6
```

[15] Dokładniej do dostatecznie dobrego przybliżenia rozwiązania ścisłego. Rozwinięcie dziesiętne $\sqrt{2}$ jest nieskończone i nieokresowe, więc nie da się go uzyskać znajdując skończoną liczbę cyfr.

[16] Wyjątkowo rzadko ale zdarzają się algorytmy o sześciennym tempie zbieżności.

```
5     xp=x;
6     x=(x+c/x)/2;
7   end%while
8   y=x;
9 end%function

10 pierw(4)
11 pierw(2)
12 pierw(1e-12)
```

Jako ostatnie w programie umieściliśmy wywołanie `pierw(1e-12)` i w tym przypadku otrzymamy wynik nieprawidłowy, gdyż $\sqrt{10^{-12}} = 10^{-6}$. Jest to wbrew pozorom przykład czysto praktyczny, gdyż takie rzędy wielkości pojawiają się przy obliczeniach dla odległości mierzonych w μm czyli 10^{-6}m.

Pokazuje to jednocześnie, że dobór wartości tolerancji, tak aby funkcja zawsze dobrze działała jest zadaniem bardzo trudnym.

5.5 Pierwiastek sześcienny.

Jako pewną ilustrację praktycznego zastosowania zadania obliczania granicy ciągu pokażemy przybliżoną metodę znajdowania pierwiastka trzeciego stopnia. Jest to uproszczenie pewnej ogólnej metody numerycznej zwanej bisekcją.

Tym razem, zamiast sformułowania matematycznego odwołamy się do tzw. zdrowego rozsądku.

Jeśli nie umiemy wprost obliczyć pierwiastka[17], to zawsze możemy sprawdzić czy dana liczba jest pierwiastkiem przez podniesienie do trzeciej potęgi.

Jeśli podniesiemy jakąś liczbę (> 1) do trzeciej potęgi i wyjdzie za dużo to wiemy, że pierwiastkiem musi być mniejsza liczba. Jeśli wyjdzie za mało to pierwiastkiem musi być liczba większa.

To spostrzeżenie jest podstawą sposobu rozwiązywania, który większość ludzi stosuje w praktyce. Polega on na systematycznym zgadywaniu rozwiązania, dodając kolejną cyfrę znaczącą do już znalezionego przybliżenia.

Załóżmy, że chcemy znaleźć $\sqrt[3]{3}$. Sprawdzamy, że poszukiwany pierwiastek jest większy od 1 (bo $1^3 < 3$) i mniejszy niż 2 (bo $2^3 = 8 > 3$). Zgadujemy drugą cyfrę, sprawdźmy 1.5, obliczamy $1.5^3 = 3.3750 > 3$. Z kolei $1.4^3 = 2.7440 < 3$. Tak więc szukany pierwiastek musi być między 1.4 a 1.5 a dokładniej pierwsze dwie cyfry to 1.4. W takim razie sprawdzamy wartość 1.45, $1.45^3 = 3.0486$ a więc za dużo. Natomiast $1.44^3 = 2.9860 < 3$ czyli za mało. Pierwsze trzy cyfry znaczące pierwiastka to 1.44

Kolejno 1.445 daje $1.445^3 = 3.0172$, wartość 1.444 daje $1.444^3 = 3.0109$ a więc też za dużo. Również 1.443 jest zbyt duże bo $1.443^3 = 3.0047$. Natomiast

[17] W rzeczywistości mamy konstruktywną metodę obliczania $\sqrt[3]{x}$ ale ten algorytm jest idealny do przypadków, kiedy nie umiemy rozwiązać zadania wprost.

1.442 jest za mało bo $1.442^3 = 2.9984$. Nasz pierwiastek leży w przedziale między 1.442 a 1.443.

W ten sposób w każdym kroku wyznaczamy kolejną cyfrę rozwinięcia dziesiętnego. Oznacza to też, że skracamy przedział, w którym musi leżeć poszukiwany pierwiastek dziesięciokrotnie. Na początku przedział był długości 1 bo pierwiastek musiał być wewnątrz przedziału $(1\ 2)$. Potem przedział był długości 0,1 bo przedział wynosił $(1,4\ 1,5)$. W kolejnym kroku przedział miał długość 0,01 bo wynosił $(1,44\ 1,45)$ itd.

Zawężanie przedziału od dziesięciokrotnie krótszego wywodzi się z mnożenia pisemnego, gdzie w każdym kroku mamy identyczną – i powoli wzrastającą – liczbę cyfr znaczących do mnożenia. Ceną za to ułatwienie jest potrzeba zgadywania kolejnej cyfry znaczącej, jednej z dziesięciu. Gdybyśmy nie zdawali się na szczęście i oszacowanie, to właściwie należałoby sprawdzać każdą możliwą cyfrę, czyli obliczać np. trzecią potęgę każdej[18] wartości: 1,40, 1,41, 1,42 ... 1,49.

Gdybyśmy zamiast mnożyć pisemnie używali kalkulatora, kiedy ilość cyfr znaczących do mnożenia nie odgrywa istotnej roli, moglibyśmy zorganizować to inaczej. Zamiast dzielić przedział na dziesięć części moglibyśmy dzielić przedział na dwie części i w każdym kroku wykonywać tylko jedno mnożenie. Zacznijmy znowu od przedziału $(1\ 2)$. W połowie długości jest $1,5$. Trzecia potęga to $1,5^3 = 3,3750$ czyli za dużo. Tak więc nasz pierwiastek leży w przedziale $(1\ 1,5)$. Weźmy znowu środek przedziału: $(1+1,5)/2 = 1,25$. Dla $1,25$ mamy $1,25^3 = 1,9531$ a więc za mało. Nasz pierwiastek leży w przedziale $(1,25\ 1,5)$. Wartość w środku przedziału to $(1,25+1,5)/2 = 1,3750$. Dla tej wartości mamy $1,3750^3 = 2,5996$ a więc też za mało. Poszukiwany pierwiastek musi leżeć wewnątrz $(1,375\ 1,5)$. Kolejny krok to wartość $1,4375$ i sprawdzenie, że $1,4375^3 = 2,9705$ a więc za mało. Pierwiastek leży w przedziale $(1,4375\ 1,5)$. W kolejnym kroku otrzymamy przedział $(1,4375\ 1,4688)$ itd.

Powtarzając to postępowanie, będziemy systematycznie zawężali przedział poszukiwań, w którym musi leżeć poszukiwane rozwiązanie. Po wystarczającej liczbie podziałów otrzymamy przedział na tyle mały, że oszacowanie jest dostatecznie dokładne do naszych celów, co oznacza, że nasza poszukiwana wartość jest większa niż początek przedziału i mniejsza niż koniec przedziału.

Podstawową wadą tej metody – przynajmniej w obliczeniach ręcznych – jest trudniejsza kontrola dokładności. Jeśli liczymy kolejne cyfry znaczące to widzimy od razu z jaką dokładnością uzyskaliśmy przybliżenie. W przypadku dzielenia na pół nie widać tak wyraźnie jaka jest dokładność rozwiązania na danym kroku. Aby mieć oszacowanie dokładności wystarczy obliczyć długość przedziału Δl, gdyż wtedy możemy powiedzieć, że nasze przybliżenie to punkt w środku przedziału $\pm\Delta l/2$.

To co jest wadą w obliczeniach na kartce okazuje się być zaletą w przypadku obliczeń przy użyciu komputera. Komputer zawsze używa tej samej liczby cyfr

[18] W zasadzie nie dla dziesięciu a dla dziewięciu bo możemy pomijać cyfrę 0 gdyż dla niej już obliczyliść trzecią potęgę.

znaczących (ok. 16-17) i śledzenie, która jest już dokładna, a która jeszcze nie, nie jest takie proste do zorganizowania. Natomiast obliczenie długości przedziału to jedno odejmowanie.

Algorytm:

Podsumowując, do wyliczenia wartości $\sqrt[3]{p}$ korzystamy z funkcji odwrotnej. Funkcją odwrotną do $x = \sqrt[3]{p}$ jest funkcja $p = x^3$. Mamy dane $p > 1$ i szukamy x. Wiemy, że x^3 jest funkcją rosnącą więc dla liczb mniejszych od x zachodzi $x^3 < p$ a dla liczb większych od x zachodzi $x^3 > p$.

Na początku musimy znaleźć przedział $[x_d\ x_g]$ taki, że $x_d^3 < p$ oraz $x_g^3 > p$.

Mając przedział $[x_d\ x_g]$ obliczamy środek przedziału x_s. Jeśli $x_s^3 < p$ to poszukiwany pierwiastek x leży w przedziale $[x_s\ x_g]$ a jeśli $x_s^3 > p$ to poszukiwany pierwiastek leży w przedziale $[x_d\ x_s]$. Czyli jeśli $x_s^3 < p$ to x_s staje się nową wartością x_g. W przeciwnym przypadku x_s staje się nową wartością x_d.

Operację powtarzamy tak długo aż $x_g - x_d$ stanie się dostatecznie małe.

Rozważmy program na przykładzie $\sqrt[3]{\pi}$. Szukamy takiej liczby x, że podniesiona do trzeciej potęgi daje π. Załóżmy, że poszukujemy pierwiastka x w przedziale od 1 do π. Jest to przedział który spełnia nasze wymagania, gdyż początek przedziału spełnia warunek $1^3 < \pi$ i koniec przedziału spełnia warunek $\pi^3 > \pi$. Wybór przedziału jest dość arbitralny[19], więc moglibyśmy wybrać przedział [1, 1000].

{pierw3.m}

```
1   p=pi;
2   xd=1;
3   xg=p;
4   while abs(xg-xd) > 1e-7
5     x=(xd+xg)/2;
6     y=x^3;
7     if y > p
8        xg=x;
9     else
10       xd=x;
11    endif
12  end%while
13  disp(x);
```

Ciąg obliczeń pokazuje tabela 5.5, gdzie w pierwszej kolumnie jest numer kroku, otrzymane przybliżenie, wartość funkcji (x^3) dla tego przybliżenia oraz oszacowanie przedziału po danym kroku.

Widać też (i można to udowodnić), że ciąg przybliżeń x_n zmierza do wartości dokładnej x.

Mamy do czynienia z ciągiem i zadaniem poszukiwania granicy tego ciągu. Co prawda ciąg ten nie bardzo daje się opisać w kategoriach wzorów matematycznych ale nie jest to istotne, gdyż jest zupełnie jasne jak te kolejne wyrazy ciągu obliczać.

[19] W naszej uproszczonej wersji funkcja musi być monotoniczna w przedziale.

i	x_i	x_i^3	$x >$	$x <$
0			1.0000	3.1416
1	2.0708	8.8800	1.0000	2.0708
2	1.5354	3.6196	1.0000	1.5354
3	1.2677	2.0373	1.2677	1.5354
4	1.4015	2.7531	1.4015	1.5354
5	1.4685	3.1666	1.4015	1.4685
6	1.4350	2.9551	1.4350	1.4685
7	1.4517	3.0596	1.4517	1.4685
8	1.4601	3.1128	1.4601	1.4685
9	1.4643	3.1397	1.4643	1.4685
10	1.4664	3.1531	1.4643	1.4664
11	1.4653	3.1464	1.4643	1.4653
12	1.4648	3.1430	1.4643	1.4648
13	1.4646	3.1413	1.4646	1.4648
14	1.4647	3.1422	1.4646	1.4647
15	1.4646	3.1418	1.4646	1.4646

{tab:pierw3}

Tabela 5.5: Ciąg przybliżeń $\sqrt[3]{\pi}$

Zbieżność. Równie dobrze moglibyśmy tego algorytm użyć do poszukiwania pierwiastka kwadratowego. W takim przypadku wystarczyłoby zmienić w programie `y=x^3` na `y=x*x`.

Warto porównać zbieżność takiego algorytmu z programem opisanym na str. 84.

Moglibyśmy rozumować w ten sposób: na pewnym etapie znaleźliśmy przedział długości d taki, że poszukiwana wartość musi się znajdować wewnątrz przedziału. W takim razie, nasze przybliżenie to środek przedziału x_s a dokładność[20] jest $\pm d/2$. W kolejnym kroku znajdziemy nowe przybliżenie (nowy przedział) czyli otrzymamy inna wartość x_s ale dokładność będzie dwukrotnie większa, gdyż długość przedziału zmniejszyła się do połowy.

Jeśli początkowy przedział przyjmiemy długości 1 (czyli [1,2]) to otrzymamy ciąg oszacowań błędów:

$$\frac{1}{2} \quad \frac{1}{4} \quad \frac{1}{8} \quad \frac{1}{16} \quad \frac{1}{32} \quad \frac{1}{64} \quad \frac{1}{128} \quad \ldots$$

Jest to zbieżność jest liniowa, gdyż błąd w kroku $n+1$–szym daje się wyrazić jako

$$\Delta_{n+1} = c_1 \Delta_n$$

W tym wypadku stała c_1 jest równa $\frac{1}{2}$.

Zbieżność byłaby kwadratowa gdyby błąd można było opisać wyrażeniem:

$$\Delta_{n+1} = c_2 (\Delta_n)^2$$

[20] To jest inna miara błędu, nie jak do tej pory różnica między wartością otrzymaną a dokładną ale maksymalny błąd przybliżenia.

Przykładowo, gdyby c_2 było równe 1 to zbieżność kwadratowa generowałaby ciąg oszacowań:

$$\frac{1}{4} \quad \frac{1}{16} \quad \frac{1}{256} \quad \frac{1}{65536} \quad \ldots$$

Widać wyraźnie, że algorytm o kwadratowej zbieżności błyskawicznie zmierza do rozwiązania. Potwierdzają to tez masze przykłady. Algorytm ze str. 84 po pięciu krokach daje 14 cyfr znaczących a algorytm o zbieżności liniowej (dla $\sqrt[3]{\pi}$) jak widać w tabeli 5.5, po 14 krokach daje jedynie 5 cyfr znaczących.

Dlatego bardzo lubimy algorytmy o zbieżności kwadratowej. Jednak nie zawsze takimi dysponujemy (a dokładniej rzadko takie mamy), wiec często musimy używać algorytmów wolniej zbieżnych.

Warto też zwrócić uwagę, ze algorytm liniowy często jesteśmy w stanie sformułować bez większego wysiłku, więc w sytuacjach, kiedy czas obliczeń nie jest najważniejszy[21] stosujemy z powodzeniem te „wolne" algorytmy.

5.6 Zmiana układu pozycyjnego

Pętla o nieokreślonej ilości przebiegów nie jest używana tylko do zadania granicy ciągu czy sumy szeregu. Spotyka się ją w wielu innych zadaniach. Jako przykład zagadnienia informatycznego pokażemy zagadnienie przeliczania liczby na inny układ pozycyjny. Mamy mianowicie zamienić liczbę z postaci dziesiętnej na postać dwójkową (binarną).

Zapis dziesiętny liczby jest zapisem pozycyjnym, czyli każda cyfra powinna być mnożona przez odpowiednią potęgę podstawy układu (10 w przypadku układu dziesiętnego).

$$127_{10} = 1 \cdot 10^2 + 2 \cdot 10^1 + 7 \cdot 10^0$$

Analogicznie, w każdym innym układzie podobnie rozumie się zapis pozycyjny cyfr:

$$261_8 = 2 \cdot 8^2 + 6 \cdot 8^1 + 1 \cdot 8^0$$

$$1011_2 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$$

Aby znaleźć reprezentację liczby dziesiętnej (będziemy przykładowo używali 127) można rozumować następująco: w układzie dwójkowym liczba ta będzie reprezentowana przez ciąg cyfr (zer lub jedynek) $b_n..b_3b_2b_1b_0$. Nie wiemy z góry ile tych cyfr będzie ale musi zachodzić:

$$127 = b_n \cdot 2^n + b_{n-1} \cdot 2^{n-1} + \ldots + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0$$

Warto zauważyć, że liczba składająca się z cyfr $b_n..b_3b_2b_10$ (cyfra zero na końcu) jest zawsze podzielna przez podstawę układu, w przypadku układu dwójkowego przez 2. Z tego wynika, że jeśli liczba dziesiętna, którą przeliczamy na układ

[21] Również wtedy, kiedy to nie ta część algorytmu decyduje o czasie obliczeń.

dwójkowy jest podzielna przez dwa to wtedy (i tylko wtedy) ostatnia cyfra reprezentacji dwójkowej b_0 jest równa zero.

Skoro nasza liczba (127) nie jest podzielna przez 2 to cyfra b_0 musi być równa 1. Wtedy mamy:

$$127 = b_n \cdot 2^n + b_{n-1} \cdot 2^{n-1} + \ldots + b_2 \cdot 2^2 + b_1 \cdot 2^1 + 1$$

a po przeniesieniu 1 na lewą stronę musi zachodzić:

$$126 = b_n \cdot 2^n + b_{n-1} \cdot 2^{n-1} + \ldots + b_2 \cdot 2^2 + b_1 \cdot 2^1$$

Jeśli podzielimy teraz obie strony przez 2 (a muszą być podzielne) to otrzymamy:

$$63 = b_n \cdot 2^{n-1} + b_{n-1} \cdot 2^{n-2} + \ldots + b_2 \cdot 2^1 + b_1 \cdot 2^0$$

i dochodzimy do zagadnienia rozkładu liczby 63 na postać dwójkową.

Skoro 63 jest nieparzyste to ostatnia cyfra rozwinięcia dwójkowego (teraz b_1) musi być 1. Znowu przenosząc 1 na lewą stronę mamy:

$$62 = b_n \cdot 2^{n-1} + b_{n-1} \cdot 2^{n-2} + \ldots + b_2 \cdot 2^1$$

Dzielimy obie strony przez 2 i otrzymujemy:

$$31 = b_n \cdot 2^{n-2} + b_{n-1} \cdot 2^{n-3} + \ldots + b_2 \cdot 2^0$$

Dojdziemy w końcu do

$$1 = b_n$$

co kończy obliczenia.

Można to ująć krótko (i niezależnie od układu na którą przeliczamy):

1. oblicz resztę z dzielenia zadanej liczby przez podstawę układu
2. otrzymana reszta jest kolejną (od prawej) cyfrą rozwinięcia
3. od liczby odejmij resztę i wynik podziel przez podstawę układu, to będzie nowa wartość liczby
4. jeśli (nowa) wartość liczby jest większa od zera to wróć do pkt. 1

W zasadzie zarys programu wynika bezpośrednio z opisanego algorytmu za wyjątkiem reprezentacji wyniku. OCTAVE używa liczb w reprezentacji dziesiętnej[22] i trudno zmusić ją do operacji na liczbach binarnych.

Tak więc możemy uzyskać binarny zapis liczby jako ciąg znaków (tekst) typu `"10101"`, na którym można jedynie przeprowadzać operacje właściwe dla tekstów bądź spróbować oszukiwać. Na początek zajmiemy się ta pierwszą możliwością. `{dec2bin.m}`

[22] Dokładniej to OCTAVE komunikuje się z użytkownikiem w układzie dziesiętnym a wewnętrznie obliczenia prowadzi w binarnym.

```
liczba=127;
wynik="";
while liczba > 0
  reszta=mod(liczba,2);  % reszta z dzielenia
  cyfra=num2str(reszta);
  wynik=strcat(cyfra,wynik);
  liczba=(liczba-reszta)/2;
end%while
disp(wynik);
```

Można też spróbować pewnego oszustwa. Obliczamy kolejne cyfry w układzie dwójkowym ale w OCTAVE wpisujemy je w układzie dziesiętnym. Wynik wygląda jak liczba dwójkowa ale operacja `11+10` da `21` zamiast `101`.

{dec2bin-num.m}

```
liczba=127;
wynik=0;
pozycja=1;
while liczba > 0
  reszta=mod(liczba,2);  % reszta z dzielenia
  cyfra=num2str(reszta);
  wynik=cyfra*pozycja+wynik;
  pozycja=10*pozycja;
  liczba=(liczba-reszta)/2;
end%while
disp(wynik);
```

5.7 Szeregi funkcyjne

Wiele bardzo ważnych, z punktu widzenia praktyki, funkcji jest definiowanych poprzez tzw. szeregi funkcyjne. Są to szeregi nieskończone, gdzie wyrażenie na n-ty wyraz jest również funkcją zmiennej x.

Jednym z najbardziej popularnych jest szereg Taylora, gdzie, jeśli znamy wartość funkcji i jej pochodnych w pewnym punkcie a możemy wyliczyć wartości funkcji dla dowolnego x:

$$f(x) = f(a) + \frac{x-a}{1!} f^{(1)}(a) + \frac{(x-a)^2}{2!} f^{(2)}(a) + \ldots + \frac{(x-a)^n}{n!} f^{(n)}(a) + \ldots$$

oraz jego wersja dla $a = 0$ czyli szereg Maclaurina

$$f(x) = f(0) + \frac{x}{1!} f^{(1)}(0) + \frac{x^2}{2!} f^{(2)}(0) + \ldots + \frac{x^n}{n!} f^{(n)}(0) + \ldots$$

Jest też wiele innych szeregów, choćby szereg Fouriera.

Z punktu widzenia matematyki analiza szeregów funkcyjnych jest dalece trudniejsza od analizy szeregów liczbowych. Z punktu widzenia programowania oba

zadania właściwie niczym się nie różnią. W szeregu funkcyjnym występuje dodatkowa **stała**[23] wartość wpisana do zmiennej x.

{sec:exp}

Funkcja wykładnicza e^x Korzystając z rozwinięcia funkcji e^x w szereg Maclaurina oraz z faktu, że dowolna pochodna e^x jest również równa e^x możemy napisać:

$$e^x = e^0 + \frac{x}{1!}e^0 + \frac{x^2}{2!}e^0 + \frac{x^3}{3!}e^0 + \dots$$

skoro zaś $e^0 = 1$ więc mamy:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

co można zapisać w postaci:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

Jest to niemal identyczne do zadania sumy szeregu, które rozpatrywaliśmy wcześniej za wyjątkiem tego, że wyraz a_i szeregu jest zależny od x, czyli mamy $a_i(x)$.

Tym niemniej będziemy traktowali to zadanie podobnie jak poprzednio. Mając na uwadze fakt, że $n!$ jest funkcją bardzo szybko rosnącą nie należy bezpośrednio obliczać wyrazu a_n. Raczej skorzystamy z rekurencji i przekształcimy wzór na a_n do postaci: $a_{n+1} = f(a_n)$ co w naszym przypadku daje:

$$a_{n+1} = \frac{x}{n} \cdot a_n$$

z warunkiem $a_0 = 1$.

W zasadzie wzór jest ważny dla dowolnego x. Jednak ze względów, o których później, będziemy zakładali, że nasze x jest nie większe od 3 (dokładniej $x < e$).

{myexp.m}

```
1 function y=myexp(x)
2   an=1;
3   n=0;
4   s=1;
5   while abs(an) > 1e-7
6      n=n+1;
7      an=x/n;
8      s=s+an;
9   end%while
10  y=s;
11 end%function
12 myexp(0)
13 myexp(1)
14 myexp(-1)
```

[23] W rzeczywistości jest to zmienna ale w trakcie obliczania sumy jest ona niezmienna czyli tak jakby była stałą.

{sec:sin}

Funkcja $sin(x)$**.** Rozwińmy funkcję $\sin(x)$ w szereg Maclaurina w otoczeniu zera.

$$\sin(x) = \sin(0) + \frac{x}{1!}\cdot\sin'(0) + \frac{x^2}{2!}\cdot\sin''(0) + \frac{x^3}{3!}\cdot\sin'''(0) + \frac{x^4}{4!}\cdot\sin^{IV}(0) + \ldots$$

Ponieważ wszystkie parzyste pochodne funkcji $\sin(x)$ dają $\pm\sin(x)$, który w punkcie $x = 0$ znika, więc wszystkie parzyste człony rozwinięcia znikają.

$$\sin(x) = \frac{x}{1!}\cdot\sin'(0) + \frac{x^3}{3!}\cdot\sin'''(0) + \frac{x^5}{5!}\cdot\sin^{V}(0) + \frac{x^7}{7!}\cdot\sin^{VII}(0) + \ldots$$

Z kolei wszystkie nieparzyste pochodne $\sin(x)$ dają $\pm\cos(x)$. Funkcja $\cos(x)$ w punkcie $x = 0$ jest równa 1. Człony zawierające nieparzyste pochodne $\sin(x)$ dają na zmianę +1 i -1, więc szereg można zapisać:

$$\sin(x) = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

Widać też możliwość rekurencyjnego sformułowania nieparzystych wyrazów a_n szeregu:

$$a_{n+2} = -a_n\frac{x^2}{(n+1)(n+2)}$$
$$a_1 = x$$

{mysin.m}

```
 1 function S=mysin(x)
 2   an=x;
 3   S=0;
 4   n=1;
 5   while abs(an)>1e-9
 6     S=S+an;
 7     an= -an*x*x/((n+1)*(n+2));
 8     n=n+2;
 9   end%while
10 end%function
11 mysin(pi/2)
12 mysin(pi)
13 mysin(3*pi/2)
14 mysin(2*pi)
```

Po uruchomieniu tego programu otrzymamy jako wyniki `1.0000`, `-5.2892e-10`, `-1.00000`, `6.4945e-10`.

Wyniki wyglądają znakomicie, dla $\pi/2$ i $3\pi/2$ otrzymaliśmy $1,0$ a dla π i 2π wartości mniejsze niż 10^{-9} czyli błąd jest poniżej narzuconej tolerancji[24].

[24]Tolerancję narzucamy na wartość wyrazu a nie na wartość sumy, więc to, że również cała suma jest z podobnym błędem jest bardzo dobrym wynikiem.

Jednak gdybyśmy obliczyli wartość `mysin(11*pi)` to uzyskalibyśmy wynik `-0.008077`. Jest to ewidentnie wynik błędny, gdyż 11π to $5 \cdot 2\pi + \pi$. Ze względu na okresowość funkcji $\sin(x)$ dowolna wielokrotność 2π daje (powinna dawać) zero a jak widzieliśmy dla π nasza funkcja działała prawidłowo.

Nie jest to błąd programu czy sformułowania. Jest to znowu skutek skończonej reprezentacji liczb. Dla stosunkowo dużej wartości x ($11\pi \approx 34$) kilka pierwszych wyrazów jest dość duże co do bezwzględnej wartości a dopiero potem gwałtownie maleją[25]. Przy czym te duże wyrazy są odejmowane. Można prosto pokazać, że największe błędy związane z zaokrąglaniem są popełniane przy odejmowaniu. Tak więc w tym przypadku jest to efekt, który zawsze pojawi się przy odejmowaniu wyrazów podobnej wielkości. Analogiczny efekt można obserwować przy obliczaniu – naszą funkcją e^x (por. str. 93) – dla dużych ujemnych wartości x (np. -20).

Można w tym momencie stwierdzić, że komputer jest mało wiarygodnym narzędziem, gdyż nie potrafi prawidłowo obliczyć $\sin(11\pi)$ i można przestać go używać.

Można też przyjąć do wiadomości, że ma swoje słabe strony i, wiedząc kiedy one się ujawniają, tak organizować obliczenia aby w jak najmniejszym stopniu wpływało to na wynik. W następnych rozdziałach pokażemy jak poprzez proste zmiany w organizacji programu można zmodyfikować naszą funkcję aby dawała dobre wyniki dla dowolnie dużych wartości x.

5.8 Znaczenie w inżynierii

Nie od rzeczy będzie wspomnieć, że dwa zadania, granica ciągu i suma szeregu, pokazane w tym rozdziale są jednymi z najważniejszych w inżynierii.

Dla większości zadań inżynierskich nie mamy danego rozwiązania ścisłego. W zupełności, do celów praktycznych, wystarczają nam przybliżenia.

Większość metod poszukiwania rozwiązania opiera się na jednym z dwu, niezbyt różniących się schematów. Albo mamy dane pewne przybliżenie początkowe i umiemy znaleźć poprawkę, która da lepsze przybliżenie, czyli w kategoriach abstrakcyjnych mamy rozwiązanie S_0 oraz umiemy znaleźć poprawkę ΔS_1. Po uwzględnieniu poprawki mamy lepsze rozwiązanie $S_1 = S_0 + \Delta S_1$. Teraz znowu możemy znaleźć poprawkę ΔS_2 a w rezultacie uzyskujemy kolejne przybliżenie $S_2 = S_1 + \Delta S_2$. Jeśli to zapiszemy w postaci jednego równania to mamy:

$$S = S_0 + \Delta S_1 + \Delta S_2 + \Delta S_3 + \Delta S_4 + \ldots$$

Łatwo w tym rozpoznać sumę nieskończonego szeregu. I łatwo też uświadomić sobie, że tak jak w praktyce nie poprawiamy w nieskończoność, tak i w zadaniu, które umownie nazywamy obliczeniem szeregu nieskończonego urywamy sumowanie po skończonej liczbie składników, kiedy uznamy, że wynik nas satysfakcjonuje.

[25] Gdyż funkcja $n!$, która jest w mianowniku rośnie znacznie szybciej niż x^2 która jest w liczniku.

Zupełnie analogicznie wygląda drugi schemat. Mamy dane jakieś przybliżenie g_0 i potrafimy go poprawić otrzymując lepsze przybliżenie g_1 itd. Łatwo w tym rozpoznać granicę ciągu przybliżeń rozwiązania:

$$g = \lim_{n\to\infty} g_n$$

Znowu, niepraktyczne byłoby szukanie w nieskończoność. Szukamy, aż znajdziemy dostatecznie dobre (cokolwiek miałoby to znaczyć) przybliżenie. Czyli tak jak w programie, urywamy po pewnej liczbie wyrazów.

W zasadzie różnica pomiędzy oboma schematami sprowadza się do tego, czy łatwiej jest znaleźć poprawkę czy łatwiej jest znaleźć wprost lepsze przybliżenie.

W kategoriach matematyki jest to trudno rozróżnialne ale należy podkreślić, że te schematy postepowania są daleko ogólniejsze. Niekoniecznie poszukiwany wynik musi być liczbą czy jakąkolwiek wielkością policzalną. Projekt skomplikowanego obiektu, dużego mostu czy samolotu, też powstaje w podobny sposób. Formułujemy pewną koncepcję wstępną. Poprawiamy ją. Powstaje pierwsza wersja projektu. Kolejne zmiany i kolejne wersje aż dojdziemy do wniosku, że dalej się nie da poprawić lub nie ma to sensu (np. ekonomicznego). O ile część poprawek na pewno jest policzalna (np. nośność) o tyle niektóre, takie jak estetyka, ekologia itp., nie dają się w żaden sposób przeliczyć.

O ile zadań nie dających się wyrazić w liczbach (czy innych kategoriach mierzalnych) nie da się oprogramować, o tyle te schematy pojawiają się wyjątkowo często we wszelkich mierzalnych zagadnieniach inżynierii. Umiejętność ich rozpoznania i oprogramowania może decydować o jakości wyniku. Inżynier, który będzie w stanie zobaczyć taki schemat w zadaniu do rozwiązania a następnie go oprogramować[26] będzie w stanie efektywnie znaleźć rozwiązanie najlepsze z możliwych. Inżynier, który nie zauważy schematu będzie skazany na metodę prób i błędów. Inżynier, który zauważy schemat ale nie będzie w stanie go sformułować w kategoriach algorytmu ugrzęźnie w rachunkach.

{z1}

5.9 Ćwiczenia

1. Użytkownik wpisuje z klawiatury kolejne liczby różne od zera. Wpisanie wartości zero oznacza koniec danych. Napisać program, który znajdzie największą z wpisanych przez użytkownika liczb.

2. Dla zadania jak zad. 1 napisać program, który oblicza średnią liczb z klawiatury.

3. Dla zadania jak zad. 1 napisać program, który oblicza średnią **kwadratów** liczb z klawiatury.

[26] Niekoniecznie osobiście, równie ważna jest umiejętność wytłumaczenia problemu programiście, który o naprężeniach wie tyle co o krasnoludkach, natomiast co to jest pętla o nieokreślonej liczbie przebiegów to wie już od przedszkola.

4. Dla zadania jak zad. 1 uzasadnić, że w ramach dotychczasowej wiedzy o OCTAVE nie da się napisać programy znajdującego medianę.

5. Znaleźć który, największy wyraz ciągu Fibonacciego, jest mniejszy od 1000. $F_0 = 1$, $F_1 = 1$, $F_{n+1} = F_n + F_{n-1}$.

6. Napisać program, który oblicza (szacuje) wartość zera maszynowego.

7. Napisać funkcję, która zwraca pierwszą cyfrę znacząca liczby. Pomóc może wykorzystanie funkcji `mod()`.

8. Napisać funkcję `fpow`, która oblicza wartość x^n dla ułamkowej potęgi $n \in (-1, 1)$. Warto wykorzystać rozwinięcie funkcji potęgowej w szereg Taylora w otoczeniu 1.

9. Napisać funkcje: `dec2oct` przeliczającą z układu dziesiętnego na ósemkowy i `dec2hex` odpowiednio na szestnastkowy.

10. Napisać program, który obliczy granicę ciągu $\lim_{n\to\infty}(1 + 2/n)^n$

11. Napisać program, który znajduję sumę szergu znakozmiennego, korzystając ze średniej kolejnych wyrazów, jak na rys. 5.1.

12. Dostosować program do obliczania pierwiastka trzeciego stopnia ze str. 88 dla liczb $p < 1$. Na tej podstawie napisać wersję uniwersalną działającą zarówno dla $p < 1$ jak i dla $p > 1$.

13. Porównać tempo zbieżności przy obliczaniu e dla ciągu $\lim_{n\to\infty}(1 + 1/n)^n$ i szeregu Maclaurina ($x = 1$).

14. Napisać program, który oblicza różnicę dwu kątów. Sprawdzić dla π i 5π.

15. Napisać program obliczający rozwinięcie π analogicznie jak program ze str. 88, korzystając z $\sin(\pi) = 0$. Przyjąć przedział początkowy [3,4]. Wykorzystać, że tym przedziale dla $x < \pi$ wartość $\sin(x) > 0$ a dla $x > \pi$ wartość $\sin(x) < 0$.

16. Napisać program obliczający NWD, czyli Największy Wspólny Dzielnik, korzystając z algorytmu Euklidesa.

17. Sporządzić tablicę wartości funkcji `mysin` (str. 94) od 1π do 100π co $\pi/2$.