Materiały do ćwiczeń z programowania w Fortranie 90

Przykładowe programy

Andrzej Eilmes
Zakład Metod Obliczeniowych Chemii UJ
Kraków 2006, 2007

Poniższe przykłady programów stanowią uzupełnienie do „Materiałów do ćwiczeń z programowania w Fortranie 90”, które Czytelnik powinien poznać wcześniej. Pojawiające się w tekście odsyłacze do numerów rozdziałów dotyczą „Materiałów...” (jeśli nie podano inaczej).

Przykład 1.

Znajdowanie pierwiastków równania kwadratowego.

Liczba pierwiastków (i wzory na ich obliczanie) zależą od znaku wyróżnika trójmianu. Do zaprogramowania schematu podejmowania decyzji w programie wykorzystujemy strukturę zagnieżdżonych instrukcji blokowych `if` przedstawioną w rozdziale 3.3. W przypadku pierwiastków zespolonych możemy policzyć osobno część rzeczywistą i urojoną rozwiązania. Ponieważ są to liczby rzeczywiste, do przechowania ich wartości wystarczy zmienna typu `real`.

Program może mieć postać:

```
!    znajdowanie pierwiastkow rownania kwadratowego

program p_rown_kw

  implicit none
  real a,b,c,delta,x1,x2,xre,xim

  write(*,*) 'podaj wspolczynniki a,b,c w rownaniu'
  write(*,*) ' ax^2 + bx + c = 0'
  read(*,*) a,b,c

  delta = b*b - 4*a*c
  if (delta>0) then
     x1 = (-b+sqrt(delta))/(2*a)
     x2 = (-b-sqrt(delta))/(2*a)
     write(*,*) 'rownanie ma dwa pierwiastki rzeczywiste'
     write(*,*) 'x1 = ',x1
     write(*,*) 'x2 = ',x2
  else
     if (delta<0) then
        xre = -b/(2*a)
        xim = sqrt(-delta)/(2*a)
        write(*,*) 'rownanie ma dwa pierwiastki zespolone'
        write(*,*) 'x1 = ',xre,' + i*',xim
        write(*,*) 'x2 = ',xre,' - i*',xim
     else
        x1 = -b/(2*a)
        write(*,*) 'rownanie ma jeden podwojny pierwiastek'
        write(*,*) 'x = ',x1
     end if
  end if

end program p_rown_kw
```

Program możemy zmodyfikować tak, aby pierwiastki zespolone przechować w zmiennych typu `complex`:

```
!    znajdowanie pierwiastkow rownania kwadratowego

program p_rown_kw

  implicit none
  real a,b,c,delta,x1,x2
  complex z1,z2

  write(*,*) 'podaj wspolczynniki a,b,c w rownaniu'
  write(*,*) ' ax^2 + bx + c = 0'
  read(*,*) a,b,c

  delta = b*b - 4*a*c
  if (delta>0) then
     x1 = (-b+sqrt(delta))/(2*a)
     x2 = (-b-sqrt(delta))/(2*a)
     write(*,*) 'rownanie ma dwa pierwiastki rzeczywiste'
     write(*,*) 'x1 = ',x1
     write(*,*) 'x2 = ',x2
  else
     if (delta<0) then
        z1 = (-b+sqrt(cmplx(delta)))/(2*a)
        z2 = (-b-sqrt(cmplx(delta)))/(2*a)
        write(*,*) 'rownanie ma dwa pierwiastki zespolone'
        write(*,*) 'x1 = ',z1
        write(*,*) 'x2 = ',z2
     else
        x1 = -b/(2*a)
        write(*,*) 'rownanie ma jeden podwojny pierwiastek'
        write(*,*) 'x = ',x1
     end if
  end if

end program p_rown_kw
```

Zwróćmy uwagę na wykorzystanie funkcji `cmplx` do skonwertowania wartości zmiennej `delta` na liczbę zespoloną przy wywołaniu funkcji `sqrt`. Dzięki temu użyta zostanie funkcja działająca na argumencie zespolonym i zwracająca wynik zespolony.

Uwaga: Program jest tylko przykładem zaprogramowania znanych z algebry wzorów. Nie zawsze jest to rozwiązanie optymalne ze względu na dokładność rozwiązań; tym zagadnieniem nie będziemy się tu zajmować.

Przykład 2.

Znajdowanie największego nietrywialnego podzielnika liczby naturalnej.

Największy podzielnik liczby znajdziemy sprawdzając podzielność danej liczby *n* kolejno przez liczby mniejsze (tzn. *n*–1, *n*–2, ..., 3, 2). Pierwsza liczba, która da resztę z dzielenia równą zero jest szukanym podzielnikiem (i możemy wtedy zakończyć przeszukiwanie). Jeśli natomiast żadna liczba do 2 włącznie nie podzieli bez reszty liczby badanej, możemy stwierdzić, że badana liczba jest pierwsza.

Do sprawdzenia podzielności liczby *n* przez *k* wykorzystamy funkcję `mod`. Wartością funkcji `mod(n,k)` jest reszta z dzielenia *n* przez *k* (czyli jeśli `mod(n,k)` jest równe 0, to *k* jest podzielnikiem *n*).

Przykładowy program:

```
!    znajdowanie najwiekszego podzielnika liczby naturalnej

program max_p
  implicit none
  integer n,i

  write(*,*) 'podaj liczbe naturalna'
  read(*,*) n

  do i = n-1,2,-1
     if (mod(n,i)==0) then
        write(*,*) 'najwiekszy nietrywialny podzielnik = ',i
        stop
     end if
  end do

  write(*,*) 'podana liczba jest pierwsza'

end program max_p
```

Przykład 2a.

Rozkład liczby naturalnej na czynniki.

Jest to problem zbliżony do poprzedniego. Zauważmy jednak, że po znalezieniu podzielnika badanej liczby powinniśmy badać podzielność ilorazu (zaczynając od ostatnio znalezionego podzielnika), np. po stwierdzeniu, że 36 dzieli się przez 2, badamy, czy iloraz 36/2 czyli 18 dzieli się przez 2, następnie badamy podzielność ilorazu 18/2 = 9 przez 2 a potem przez 3, itd.

Przykładowy program ma postać:

```
!   rozklad liczby na czynniki

program rozklad

  implicit none
  integer n,k,i
  logical podz

  write(*,*) 'podaj liczbe naturalna'
  read(*,*) n

  podz=.false.

  do i=2,n-1
     do while (mod(n,i)==0)
        if (.not.podz) then
           write(*,*) 'rozklad podanej liczby na czynniki:'
           podz=.true.
        end if
        write(*,*) i
        n=n/i
     end do
  end do

  if (.not.podz) write(*,*) 'podana liczba jest pierwsza'

end program rozklad
```

Iloraz badanej liczby `n` przez ostatnio znaleziony podzielnik `i` przechowujemy w zmiennej `n`. Należy zwrócić uwagę, że zmiana wartości `n` wewnątrz pętli `do` nie ma wpływu na liczbę jej przebiegów (zabroniona jest tylko modyfikacja zmiennej sterującej, czyli `i`).
Pętlę `do while` wykorzystujemy do sprawdzania podzielności przez aktualnie badany podzielnik (czyli `i`), tak długo, jak długo liczba się przez niego dzieli.

Zmienna logiczna `podz` służy do przechowania informacji, czy znaleźliśmy choć jeden podzielnik (przyjmuje wtedy wartość `.true.`). Przy znalezieniu pierwszego podzielnika (i tylko wtedy) wypisujemy tekst ‘rozkład podanej liczby na czynniki’. Jeśli po sprawdzeniu wszystkich liczb od 2 do n–1 zmienna `podz` ma wartość `.false.`, to znaczy, że nie znaleziono żadnego podzielnika, czyli badana liczba jest pierwsza.

Przykład 3.

Znajdowanie najmniejszej wspólnej wielokrotności dwu liczb.

Algorytm znalezienia NWW liczb *n* i *m* jest następujący: sprawdzamy podzielność *n* przez *m*, następnie *n*+*n* przez *m*, *n*+*n*+*n* przez *m*, itd. Pierwsza z badanych wielokrotności *n*, która podzieli się bez reszty, jest szukaną NWW. Nietrudno zauważyć, że szybciej znajdziemy rozwiązanie, jeśli testujemy podzielność wielokrotności liczby większej przez liczbę mniejszą. Do wyboru mniejszej i większej liczby z pary wykorzystać możemy funkcje `min` i `max` (patrz Uzupełnienie A).

Program:

```
!     znajdowanie najmniejszej wspolnej wielokrotnosci dwu liczb

program nww
  implicit none
  integer n,m,lm,lw,k

  write(*,*) 'podaj dwie liczby naturalne'
  read(*,*) n,m

  lm = min(n,m)
  lw = max(n,m)

  k = lw
  do
     if (mod(k,lm)==0) then
        write(*,*) 'NWW dla podanych liczb = ',k
        exit
     end if
     k = k + lw
  end do

end program nww
```

Przykład 4.

Znajdowanie najmniejszego (największego) elementu w podanym ciągu liczb oraz liczby elementów ciągu mniejszych od zera.

Wczytajmy ciąg liczb z pliku zewnętrznego i przechowajmy go w jednowymiarowej tablicy. Pierwsza liczba we wczytywanym pliku podaje, ile jest elementów ciągu (przechowujemy tę wartość w zmiennej `n`. Znalezienie najmniejszego (największego) elementu w ciągu realizujemy przeglądając tablicę i porównując jej elementy z zapamiętaną wartością najmniejszego (największego) dotąd znalezionego. Jeśli trafimy na mniejszy (większy) zapamiętujemy nową wartość. Oczywiście na początku przeglądania tablicy znalezione minimum (maksimum) to pierwszy element. Zliczanie elementów spełniających pewien warunek (np. < 0), realizujemy sprawdzając ten warunek kolejno dla wszystkich elementów i zwiększając w razie potrzeby licznik znalezionych elementów, dla których warunek jest spełniony (w przykładowym programie licznikiem jest zmienna `nm0`).

Program:

```
!  znajdowanie najmniejszej i najwiekszej liczby w ciagu
!  oraz liczby elementow ciagu mniejszych od zera

program minmax

  implicit none
  real, allocatable :: x(:)
  real xmin,xmax
  integer i,n,nm0

!     wczytanie danych

  open(11,file='p4.dat')
  read(11,*) n
  if (n<1) then
     write(*,*) 'za malo danych'
     stop
  end if
  allocate(x(n))
  do i=1,n
     read(11,*) x(i)
  end do
  close(11)


!    znajdowanie najmniejszego i najwiekszego elementu ciagu


  xmin = x(1)
  xmax = x(1)

  do i = 2,n
     if (x(i)>xmax) xmax = x(i)
```

```
      if (x(i)<xmin) xmin = x(i)
   end do

   write(*,*) 'najmniejsza liczba = ',xmin
   write(*,*) 'najwieksza liczba = ',xmax

!     znajdowanie liczby elementow ciagu mniejszych od zera

   nm0 = 0

   do i = 1,n
      if (x(i)<0) nm0 = nm0 + 1
   end do

   write(*,*) 'liczba elementow ujemnych = ',nm0
   deallocate(x)

end program minmax
```

Można zauważyć, że w opisanym przykładzie nie musieliśmy zapamiętywać wczytanych elementów w tablicy. Zadanie można zrealizować sprawdzając odpowiednie warunki dla ostatnio wczytanego elementu ciągu (później jego wartość nie jest już potrzebna).
Zmodyfikowany program:

```
!  znajdowanie najmniejszej i najwiekszej liczby w ciagu
!  oraz liczby elementow ciagu mniejszych od zera

program minmax
   implicit none
   real x,xmin,xmax
   integer nm0,io

   nm0 = 0
   open(11,file='p4a.dat')
   read(11,*,iostat=io) x
   if (io/=0) then
      write(*,*) 'za malo danych'
      stop
   end if
   xmin = x
   xmax = x
   if (x<0) nm0 = nm0 + 1

   do
      read(11,*,iostat=io) x
      if (io/=0) then
         exit
      end if
      if (x>xmax) xmax = x
      if (x<xmin) xmin = x
```

```
      if (x<0) nm0 = nm0 + 1
   end do
   close(11)

   write(*,*) 'najmniejsza liczba = ',xmin
   write(*,*) 'najwieksza liczba = ',xmax
   write(*,*) 'liczba elementow ujemnych = ',nm0

end program minmax
```

W tym przypadku plik z danymi nie zawiera na początku wartości określającej liczbę elementów ciągu. Przy czytaniu danych wykorzystaliśmy specyfikator `iostat` do określenia zmiennej (w naszym przypadku `io`) przechowującej kod błędu (patrz rozdział 6.1). Jeśli wczytanie kolejnej wartości się nie powiodło (bo w pliku nie ma więcej danych), zmienna ta przyjmuje wartość różną od zera, co jest dla nas sygnałem zakończenia wczytywania .

Przykład 5.

Sortowanie jednowymiarowej tablicy.

Prosty algorytm (tzw. proste wybieranie) uporządkowania w kolejności rosnącej elementów przechowywanych w jednowymiarowej tablicy jest następujący:
- pierwszy element tablicy porównujemy po kolei ze wszystkimi po nim następującymi i, jeśli są w nieodpowiedniej kolejności, zamieniamy je miejscami
- po tym etapie mamy pewność, że pierwszy element tablicy jest już na swoim miejscu (pozostałe niekoniecznie)
- powtarzamy zatem procedurę porównując drugi element tablicy z wszystkimi następnymi, po tym przeglądnięciu tablicy na pewno mamy elementy 1 i 2 na właściwych pozycjach
- procedurę kontynuujemy do chwili, aż przedostatni element trafi na właściwe miejsce (wtedy ostatni też już musi być na właściwej pozycji).

Algorytm ten zastosujmy do uporządkowania wczytanego z pliku ciągu liczb rzeczywistych:

```
!  sortowanie jednowymiarowej tablicy
program sort
  implicit none
  real tmp
  real, allocatable :: x(:)
  integer i,j,n
  character(30) nazwa

!     wczytanie danych

  write(*,*) 'podaj nazwe pliku danych'
  read(*,'(a)') nazwa
  open(11,file=nazwa)
  read(11,*) n
  allocate(x(n))
  do i=1,n
     read(11,*) x(i)
  end do
  close(11)

!    sortowanie

  do i = 1,n-1
     do j = i+1,n
        if (x(i)>x(j)) then
           tmp  = x(j)
           x(j) = x(i)
           x(i) = tmp
        end if
     end do
  end do

!     wypisanie wyniku

  write(*,*) 'posortowana tablica:'
  do i = 1,n
     write(*,*) x(i)
  end do

  deallocate(x)
  end program sort
```

W ten sam sposób posortować możemy teksty w porządku leksykograficznym (zależnym od przyjętego uporządkowania znaków). Wystarczy w tym celu zmienić typ tablicy `x` (oraz zmiennej `tmp`) na znakowy (w poniższym przykładzie elementy tablicy to teksty o długości do 60 znaków).

```
!  sortowanie jednowymiarowej tablicy

program sort
  implicit none
  character(60), allocatable :: x(:)
  character(60) tmp
  integer i,j,n
  character(30) nazwa

!     wczytanie danych

  write(*,*) 'podaj nazwe pliku danych'
  read(*,'(a)') nazwa
  open(11,file=nazwa)
  read(11,*) n
  allocate(x(n))
  do i=1,n
     read(11,'(a)') x(i)
  end do
  close(11)


!    sortowanie

  do i = 1,n-1
     do j = i+1,n
        if (x(i)>x(j)) then
           tmp  = x(j)
           x(j) = x(i)
           x(i) = tmp
        end if
     end do
  end do

!     wypisanie wyniku

  write(*,*) 'posortowana tablica:'
  do i = 1,n
     write(*,'(a)') x(i)
  end do
  deallocate(x)

end program sort
```

Program ten możemy też użyć do posortowania liczb (bo zapis liczby to pewien tekst), wynik może być jednak inny niż przy sortowaniu wg wartości (np. sprawdź, jaka będzie kolejność liczb 1, 2, 10). Warto też sprawdzić przez posortowanie odpowiedniego zestawu tekstów jaka jest kolejność uporządkowania liter (dużych i małych) oraz cyfr.

Przykład 6.

Mnożenie macierzy.

Element z *i*-tego wiersza i *j*-tej kolumny macierzy **C** będącej iloczynem macierzy **A** i **B** (**C** = **AB**) zadany jest wzorem:

$$c_{ij} = \sum_{k=1}^{ika} a_{ik} b_{kj}$$

gdzie *ika* to liczba kolumn macierzy **A**. Ponieważ musimy obliczyć wszystkie elementy macierzy **C** (tzn. dla *i* = 1 ... *iwa* i *j* = 1 ... *ikb*), do wykonania mnożenia potrzebujemy trzech zagnieżdżonych pętli.

Macierze w programie reprezentowane są przez dwuwymiarowe tablice.
Niezbędne dane program wczytuje z pliku ‘p6.dat’, zawierającego w pierwszym wierszu wymiary (liczbę wierszy i liczbę kolumn) macierzy **A**, w drugim wierszu wymiary macierzy **B**, następnie zapisaną wierszami macierz **A** i analogicznie macierz **B**. Po wczytaniu wymiarów macierzy należy sprawdzić, czy mnożenie jest wykonalne (czyli, czy *ika* = *iwb*).

Program:

```
!  mnozenie macierzy

program mnozmac
  implicit none

  real, allocatable :: a(:,:),b(:,:),c(:,:)
  integer iwa,iwb,ika,ikb
  integer i,j,k

!   wczytanie danych

  open(11,file='p6.dat')
  read(11,*) iwa,ika
  read(11,*) iwb,ikb

  if (ika/=iwb) then
     write(*,*) 'tych macierzy nie da sie pomnozyc'
     stop
  end if

  allocate(a(iwa,ika),b(iwb,ikb),c(iwa,ikb))

  do i = 1,iwa
     read(11,*) (a(i,j),j=1,ika)
  end do

  do i = 1,iwb
     read(11,*) (b(i,j),j=1,ikb)
  end do

  close(11)

!   mnozenie
```

```
  do i = 1,iwa
     do j = 1,ikb
        c(i,j) = 0.
        do k = 1,ika
           c(i,j) = c(i,j) + a(i,k)*b(k,j)
        end do
     end do
  end do

!  wypisanie wynikow

  write(*,*) ' macierz A:'

  do i = 1,iwa
     write(*,*) (a(i,j),j=1,ika)
  end do

  write(*,*)

  write(*,*) ' macierz B:'

  do i = 1,iwb
     write(*,*) (b(i,j),j=1,ikb)
  end do

  write(*,*)

  write(*,*) ' iloczyn A*B:'

  do i = 1,iwa
     write(*,*) (c(i,j),j=1,ikb)
  end do

  deallocate(a,b,c)

end program mnozmac
```

W powyższym programie zaprogramowaliśmy obliczanie wyniku mnożenia macierzy w sposób jawny, bezpośrednio programując odpowiednie wyrażenie. W Fortranie 90 można jednak skorzystać z funkcji `matmul` zwracającej jako wynik iloczyn podanych macierzy (patrz Uzupełnienie A):

```
!  mnozenie macierzy

program mnozmac
  implicit none

  real, allocatable :: a(:,:),b(:,:),c(:,:)
  integer iwa,iwb,ika,ikb
  integer i,j

!   wczytanie danych

  open(11,file='p6.dat')
  read(11,*) iwa,ika
  read(11,*) iwb,ikb

  if (ika/=iwb) then
     write(*,*) 'tych macierzy nie da sie pomnozyc'
```

```
      stop
    end if

    allocate(a(iwa,ika),b(iwb,ikb),c(iwa,ikb))

    do i = 1,iwa
       read(11,*) (a(i,j),j=1,ika)
    end do

    do i = 1,iwb
       read(11,*) (b(i,j),j=1,ikb)
    end do

    close(11)

!   mnozenie

    c=matmul(a,b)


!  wypisanie wynikow

    write(*,*) ' macierz A:'

    do i = 1,iwa
       write(*,*) (a(i,j),j=1,ika)
    end do

    write(*,*)

    write(*,*) ' macierz B:'

    do i = 1,iwb
       write(*,*) (b(i,j),j=1,ikb)
    end do

    write(*,*)

    write(*,*) ' iloczyn A*B:'

    do i = 1,iwa
       write(*,*) (c(i,j),j=1,ikb)
    end do

    deallocate(a,b,c)

end program mnozmac
```

Przykład 7.

Obliczanie iloczynu skalarnego dwu wektorów.

Iloczyn skalarny wektorów **a** = (a_1, ..., a_N) i **b** = (b_1, ..., b_N) w przestrzeni N – wymiarowej zadany jest wzorem

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{N} a_i b_i$$

W pierwszym przykładowym programie obliczanie iloczynu skalarnego zrealizujemy przy pomocy funkcji rzeczywistej `skalarny`, której argumentami będą dwie jednowymiarowe tablice rzeczywiste przechowujące mnożone wektory.

Z definicji długości wektora

$$\|\mathbf{a}\| = \sqrt{\sum_{i=1}^{N} a_i^2}$$

widać, że możemy ją policzyć jako pierwiastek z iloczynu skalarnego wektora samego z sobą. Naszą funkcję użyjemy zatem także do obliczenia długości wektorów.

Program:

```
! przyklad obliczania iloczynu skalarnego

program ilskal
  implicit none

  real, allocatable :: x(:),y(:)
  real dlx,dly,sxy
  integer i,n
  character(30) nazwa

!     wczytanie danych

  write(*,*) 'podaj nazwe pliku danych'
  read(*,'(a)') nazwa
  open(11,file=nazwa)
  read(11,*) n
  allocate(x(n),y(n))
  do i=1,n
     read(11,*) x(i),y(i)
  end do
  close(11)

!    wywolania funkcji skalarny

  dlx = sqrt(skalarny(x,x))
  dly = sqrt(skalarny(y,y))
  sxy = skalarny(x,y)

  deallocate(x,y)

!    wypisanie wynikow

  write(*,*) 'dlugosc wektora x = ',dlx
  write(*,*) 'dlugosc wektora y = ',dly
  write(*,*) 'iloczyn skalarny wektorow x i y = ',sxy
```

```
contains

!   definicja iloczynu skalarnego

  real function skalarny(x,y)

    integer i
    real x(:),y(:)

    skalarny = 0.
    do i = 1,size(x)
       skalarny = skalarny + x(i)*y(i)
    end do

  end function skalarny

end program ilskal
```

Deklaracje tablic w funkcji skalarny zapewniają, iż tablica będzie miała tyle elementów, ile zostanie przesłane jako parametr aktualny przy wywołaniu funkcji. Ile tych elementów jest, dowiadujemy się badając rozmiar tablicy `x` przy pomocy funkcji `size` (oczywiście rozmiar tablicy `y` ma być taki sam). Zwróćmy uwagę, iż w funkcji `skalarny` dostępna jest zmienna `n` z programu głównego (poprzez *host association*), zatem moglibyśmy używać jej wartości do określenia liczby elementów tablicy zamiast `size(x)`. Spowodowałoby to jednak, iż napisana przez nas funkcja jest mniej uniwersalna (co stałoby się, gdyby użyć jej w programie, w którym wymiar przestrzeni przechowano w zmiennej o innej nazwie, np. `m` ?).

W powyższym programie dla przykładu zaprogramowaliśmy własną funkcję obliczającą iloczyn skalarny. W praktyce nie jest to potrzebne, ponieważ funkcja taka w Fortranie 90 istnieje i ma nazwę `dot_product` (Uzupełnienie A). Zmodyfikowany program wygląda następująco:

```
! przyklad obliczania iloczynu skalarnego

program ilskal
  implicit none

  real, allocatable :: x(:),y(:)
  real dlx,dly,sxy
  integer i,n
  character(30) nazwa

!     wczytanie danych

  write(*,*) 'podaj nazwe pliku danych'
  read(*,'(a)') nazwa
  open(11,file=nazwa)
  read(11,*) n
  allocate(x(n),y(n))
  do i=1,n
     read(11,*) x(i),y(i)
  end do
  close(11)


!    wywolania funkcji obliczajacej iloczyn skalarny
```

```
  dlx = sqrt(dot_product(x,x))
  dly = sqrt(dot_product(y,y))
  sxy = dot_product(x,y)

  deallocate(x,y)

!    wypisanie wynikow

  write(*,*) 'dlugosc wektora x = ',dlx
  write(*,*) 'dlugosc wektora y = ',dly
  write(*,*) 'iloczyn skalarny wektorow x i y = ',sxy


end program ilskal
```

Przykład 8.

Obliczanie iloczynu wektorowego w przestrzeniu trójwymiarowej.

Do obliczenia iloczynu wykorzystamy podprogram, mimo iż wywołany będzie on tylko raz (uzyskamy jednak bardziej przejrzysty program). Przy okazji zobaczymy przykład programowania funkcji, której wynikiem jest tablica.

Ponieważ dla uproszczenia umówiliśmy się, że obliczamy iloczyn w przestrzeni trójwymiarowej, wszystkie wektory występujące w programie przechowujemy w tablicach jednowymiarowych o trzech elementach. Składowe wektora wynikowego wyrażają się prosto poprzez składowe mnożonych wektorów.

Program:

```
!  obliczanie iloczynu wektorowego w przestrzeni trojwymiarowej

program il_wektorowy
  implicit none

  real x(3),y(3),z(3)
  integer i

  write(*,*) 'podaj skladowe wektora x'
  read(*,*) (x(i),i=1,3)
  write(*,*) 'podaj skladowe wektora y'
  read(*,*) (y(i),i=1,3)

  z=ilwek(x,y)

  write(*,*) 'iloczyn wektorowy x i y wynosi:'
  write(*,*) (z(i),i=1,3)

contains

  function ilwek(a,b)

    real a(3),b(3),ilwek(3)

    ilwek(1) = a(2)*b(3) - a(3)*b(2)
    ilwek(2) = a(3)*b(1) - a(1)*b(3)
    ilwek(3) = a(1)*b(2) - a(2)*b(1)

  end function ilwek

end program il_wektorowy
```

Zarówno w programie głównym jak i w podprogramie użyliśmy statycznych deklaracji rozmiarów tablic, ponieważ były one z góry określone (zaprogramowany wzór stosuje się dla przestrzeniu trójwymiarowej).

Przykład 9.

Rozwiązywanie układu 3 równań liniowych z 3 niewiadomymi z zastosowaniem wzorów Cramera.

Nasz układ równań ma postać:

$$a_{11}x + a_{12}y + a_{13}z = b_1$$
$$a_{21}x + a_{22}y + a_{23}z = b_2$$
$$a_{31}x + a_{32}y + a_{33}z = b_3$$

Wartości niewiadomych *x*,*y* i *z* znajdujemy ze wzorów:

$$x = \frac{W_x}{W}, \qquad y = \frac{W_y}{W}, \qquad z = \frac{W_z}{W},$$

gdzie wyznacznik główny układu *W* to wyznacznik macierzy współczynników *a* (musi być różny od zera) a wyznaczniki W_x, W_y, W_z otrzymujemy zastępując w macierzy współczynników *a* odpowiednio 1, 2 lub 3 kolumnę kolumną prawych stron równań *b*.

W naszym programie musimy zatem policzyć cztery wyznaczniki; wykorzystamy do tego celu funkcję `det`, argumentem której jest macierz 3×3. Oprócz tego przydatna będzie pomocnicza procedura `copy`, która w pomocniczej tablicy `tmp` umieszcza macierz współczynników `a` i w miejsce jej kolumny o numerze `kol` umieszcza kolumnę zadaną przez prawe strony równań `b`.

Po obliczeniu głównego wyznacznika układu powinniśmy sprawdzić, czy jest on różny od zera. Ponieważ jest to liczba rzeczywista, przyrównanie jej do zera niekoniecznie da pożądany efekt (patrz rozdział 3.2). Zamiast tego sprawdzamy, czy wartość bezwzględna wyznacznika jest mniejsza od dostatecznie małej wartości progowej `eps`. Wartość tę zadamy przy pomocy stałej (z atrybutem `parameter)`, aby w razie potrzeby zmiany łatwo było ją odszukać.

Program:

```
!   rozwiazywanie ukladu 3 rownan liniowych z 3 niewiadomymi metoda
!   wyznacznikow

program wyznaczniki
  implicit none

  real, parameter :: eps=1.e-8
  real a(3,3),b(3),tmp(3,3)
  real w,wx,wy,wz,x,y,z
  integer i,j
  character(30) nazwa


!      wczytujemy dane

  write(*,*) 'podaj nazwe pliku danych'
```

```
  read(*,'(a)') nazwa
  open(11,file=nazwa)
  do i = 1,3
     read(11,*) (a(i,j),j=1,3),b(i)
  end do

!      obliczamy wyznacznik glowny

  w = det(a)

  if (abs(w)<eps) then
     write(*,*) 'wyznacznik glowny = 0'
     stop
  end if

!      obliczamy wyznaczniki wx, wy, wz

  call copy(a,b,tmp,1)
  wx = det(tmp)

  call copy(a,b,tmp,2)
  wy = det(tmp)

  call copy(a,b,tmp,3)
  wz = det(tmp)

!     obliczamy wartosci zmiennych

  x = wx/w
  y = wy/w
  z = wz/w

  write(*,*) 'rozwiazanie ukladu'
  write(*,*) 'x = ',x
  write(*,*) 'y = ',y
  write(*,*) 'z = ',z

contains

  subroutine copy(a,b,tmp,kol)

    real a(3,3),b(3),tmp(3,3)
    integer i,j,kol

    do i=1,3
       do j=1,3
          tmp(i,j) = a(i,j)
       end do
    end do

    do i=1,3
       tmp(i,kol)=b(i)
    end do

  end subroutine copy


  function det(a)

    real a(3,3),det
```

```
    det =  a(1,1)*a(2,2)*a(3,3) + a(2,1)*a(3,2)*a(1,3) &
           + a(3,1)*a(1,2)*a(2,3) - a(1,3)*a(2,2)*a(3,1) &
           - a(2,3)*a(3,2)*a(1,1) - a(3,3)*a(1,2)*a(2,1)

  end function det

end program wyznaczniki
```

Atrybut `parameter` użyty w odniesieniu do wartości `eps`, oznacza, iż jest to stała, której wartość nadano już w deklaracji. Wartości tej nie można już w programie zmienić.

Przykład 10.

Obliczanie funkcji e^x poprzez jej rozwinięcie w szereg.

Odpowiednie rozwinięcie w szereg na postać:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!} + \ldots = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

Należy zauważyć, że *i*-ty składnik tej sumy otrzymujemy wymnażając składnik poprzedni, tzn. (*i*-1)-szy przez *x/i*.

W naszym programie możemy oczywiście uwzględnić jedynie skończoną liczbę członów rozwinięcia szeregu.

Dla sprawdzenia zbieżności naszego rozwinięcia w zależności od *n* i *x*, program oprócz sumy szeregu wypisuje wartość obliczoną z użyciem funkcji `exp(x)`.

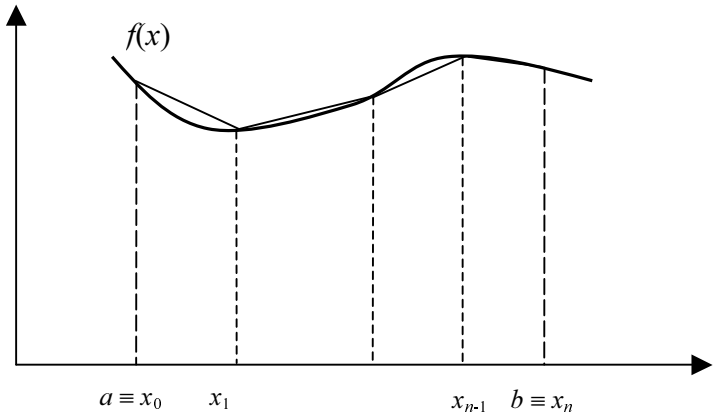
Program:

```
!   obliczanie funkcji exp jako sumy szeregu

program suma_sz
  implicit none
  real suma,skladnik,x
  integer i,n

  write(*,*) 'podaj x'
  read(*,*) x
  write(*,*) 'podaj liczbe czlonow szeregu'
  read(*,*) n

  suma = 1.
  skladnik = 1.

  do i = 1,n
     skladnik = skladnik*x/i
     suma = suma + skladnik
  end do

  write(*,*) 'exp(x) z szeregu = ',suma
  write(*,*) 'exp(x) z funkcji = ',exp(x)

end program suma_sz
```

Przykład 11.

Całkowanie metodą trapezów.

W celu przybliżonego obliczenia całki oznaczonej z funkcji $f(x)$ w granicach od a do b wykres funkcji przybliżamy łamaną dzieląc przedział $<a,b>$ na części. Pole figury ograniczonej łamaną znajdujemy jako sumę pól utworzonych w ten sposób trapezów; pole to jest przybliżeniem szukanej całki.



Zakładając, że odcinek $<a,b>$ podzieliliśmy na n równych części (każda o długości $h = (b-a)/n$), oznaczmy $a \equiv x_0$, $b \equiv x_n$. Pole trapezu znajdującego się pomiędzy x_i a x_{i+1} wynosi (trapez stoi na „boku", zatem jego podstawy to wartości funkcji w punktach x_i i x_{i+1}):

$$h\frac{f(x_i)+f(x_{i+1})}{2}.$$

Zauważmy, że podczas sumowania pól wszystkich trapezów wartości funkcji w punktach x_i dla i = 1,... n-1 będziemy liczyć dwukrotnie. Znajdujemy zatem następujący wzór na pole figury ograniczonej łamaną (przybliżenie naszej całki):

$$h\left[\frac{1}{2}\big(f(a)+f(b)\big)+\sum_{i=1}^{n-1}f(x_i)\right]$$

Uzbrojeni w tę wiedzę przystąpmy do napisania programu, który policzy całkę oznaczoną z funkcji sin(x)cox(x).

W programie użyliśmy funkcji o nazwie `f`, definiującej całkowaną funkcję. Dzięki temu instrukcje w programie głównym są bardziej przejrzyste a do zmiany całkowanej funkcji wystarczy jedynie modyfikacja podprogramu `f`.

Program:

```
!  calkowanie metoda trapezow

program trapez
  implicit none
  real a,b,pi,h,x,s,calka
  integer i,n
```

```
  write(*,*) 'podaj granice przedzialu calkowania (w stopniach)'
  read(*,*) a,b
  write(*,*) 'podaj ilosc podzialow przedzialu'
  read(*,*) n

  pi = 4.*atan(1.0e0)
  a = pi*a/180
  b = pi*b/180
  h = (b-a)/n

  s = 0.

  do i = 1,n-1
     x = a + i*h
     s = s + f(x)
  enddo
  s = s + (f(a)+f(b))/2
  calka = h*s

  write(*,*) 'wartosc calki: ', calka

contains

  real function f(x)
    real x

    f = sin(x)*cos(x)

  end function f

end program trapez
```

Przykład 12.

Odwracanie macierzy.

W rozdziale 11.3 omówiliśmy wykorzystanie procedury `la_gesv` z biblioteki Lapack95 do rozwiązania układu równań liniowych. Zauważyliśmy tam, że obliczenie macierzy odwrotnej wymaga rozwiązania *n* układów równań o współczynnikach zadanych przez odwracaną macierz. Prawe strony tych równań tworzą macierz jednostkową. Nasz program będzie zatem prostą modyfikacją programu z rozdz. 11.3: musimy jedynie zmienić nieco wczytywanie danych – wczytujemy tylko macierz **A**, natomiast tworzymy macierz **B**. Odpowiednio modyfikujemy także wypisywanie wyniku.

Przy kompilacji programu musimy pamiętać o dołączeniu odpowiednich bibliotek (patrz rozdz. 11.3).

Program:

```
! wyznaczanie macierzy odwrotnej

program macodwr
  use la_precision
  use f95_lapack
  implicit none

  integer info,i
  real(dp),allocatable ::  a(:,:),b(:,:)
  integer, allocatable :: ipiv(:)

  call czytaj1()

  call la_gesv(a,b,ipiv,info)

  if (info==0) then
     call wypisz1()
  else
     write(*,*) 'problemy'
  end if
  deallocate(a,b,ipiv)

contains

  subroutine czytaj1()
    integer i,j,n

    open(11,file='p12.dat')
    read(11,*) n
    allocate(a(n,n),b(n,n),ipiv(n))
    do i=1,n
       read(11,*) (a(i,j),j=1,n)
    end do
    close(11)
    do i=1,n
       do j=1,n
          b(i,j) = 0.
       end do
       b(i,i) = 1.
    end do
  end subroutine czytaj1
```

```
  subroutine wypisz1()
    integer i,j

    open(11,file='p12.res')
    do i=1,size(b,1)
       write(11,*) (b(i,j),j=1,size(b,2))
    end do

  end subroutine wypisz1

end program macodwr
```

Przykład 13.

Obliczenia metodą Hückela dla polienów cyklicznych.

Analizując opis metody Hückela (np. A. Gołębiewski, *Elementy mechaniki i chemii kwantowej*, rozdz.15.2) oraz metody rozwiązywania zagadnienia własnego w chemii kwantowej (*ibidem*, uzupełnienie F) stwierdzamy, że znalezienie energii i postaci orbitali molekularnych dla cyklicznego polienu o n atomach węgla sprowadza się do diagonalizacji macierzy postaci:

$$\begin{bmatrix} \alpha_0 & \beta_0 & 0 & 0 & \cdots & 0 & \beta_0 \\ \beta_0 & \alpha_0 & \beta_0 & 0 & \cdots & \cdots & 0 \\ 0 & \beta_0 & \alpha_0 & \beta_0 & 0 & \cdots & 0 \\ \vdots & 0 & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & 0 & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & \ddots & \ddots & \ddots & \beta_0 \\ \beta_0 & 0 & \cdots & \cdots & 0 & \beta_0 & \alpha_0 \end{bmatrix}$$

Macierz ta ma wymiar $n \times n$, jej elementy diagonalne mają wartość α_0 zaś element pozadiagonalny z i-tego wiersza i j-kolumny jest równy β_0, gdy atomy i i j sąsiadują ze sobą; w przeciwnym wypadku element ten jest równy 0. Ponieważ w cząsteczce atom i-ty sąsiaduje z atomami i-1 oraz i+1, więc w większości wierszy macierzy elementy β_0 sąsiadują z diagonalą (z wyjątkiem pierwszego i ostatniego wiersza, ponieważ atom 1 jest połączony z n-tym). Wartości α_0 i β_0 są parametrami empirycznymi.

W zasadzie moglibyśmy poszukać w literaturze wartości elementów α_0 i β_0 i użyć w obliczeniach, wcale jednak nie rozjaśniłoby to otrzymanego schematu poziomów energetycznych.

Zwróćmy natomiast uwagę na fakt, iż energia określona jest z dokładnością do stałej addytywnej. Dodanie do elementów diagonalnych powyższej macierzy jakiejś stałej wartości przesunie energie wszystkich poziomów o tę wartość. Możemy zatem umówić się, że α_0 przyjmujemy za zero energii (kładąc 0 jako wartość tego elementu macierzowego). Z kolei możemy przyjąć, że energie wyrażamy w jednostkach β_0 (czyli $\beta_0 = 1$). Wtedy nasza macierz ma postać:

$$\begin{bmatrix} 0 & 1 & 0 & \cdots & 0 & 1 \\ 1 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 1 & \ddots & 1 & 0 & \vdots \\ \vdots & \ddots & \ddots & & \ddots & 0 \\ 0 & & \ddots & \ddots & \ddots & 1 \\ 1 & 0 & \cdots & 0 & 1 & 0 \end{bmatrix}$$

Pamiętać tylko musimy, że jeśli energię orbitalu obliczymy jako E, to po powrocie do oryginalnych jednostek jest to $\alpha_0 + E \cdot \beta_0$.

Po skonstruowaniu odpowiedniej macierzy musimy ją zdiagonalizować. Wykorzystamy w tym celu podprogram `la_syev` z biblioteki Lapack91. Na podstawie podręcznika albo kodu podprogramu ustalamy jego argumenty:

```
SUBROUTINE LA_SYEV( A, W, JOBZ, UPLO, INFO)
```

`a` – dwuwymiarowa tablica rzeczywista zawierająca diagonalizowaną macierz; jeśli `jobz` jest równe '`V`', to po wyjściu z podprogramu zawiera wektory własne macierzy (kolumnami)
`w` – jednowymiarowa tablica rzeczywista o długości `n`; po zakończeniu pracy procedury zawiera wartości własne macierzy
`jobz` – zmienna znakowa (`character`) określająca zadanie dla podprogramu (gdy jest równa '`N`' obliczane są tylko wartości własne, gdy jest równa '`V`' – wartości i wektory własne).
`uplo` – zmienna znakowa podająca, czy w tablicy `a` zapisany jest górny ('`U`'), czy dolny ('`L`') trójkąt diagonalizowanej macierzy
`info` – zmienna całkowita; po wyjściu z podprogramu jest równa 0, jeśli diagonalizacja przebiegła pomyślnie

Pozostaje nam wobec tego skonstruować macierz hamiltonianu (jak widać wystarczy w tablicy zapisać tylko jej górny albo dolny (tak my zrobimy) trójkąt (czyli element nad lub pod główną przekątną wraz z nią). Po zawołaniu procedury diagonalizującej sprawdzamy, czy zmienna `info` jest równa 0 i jeśli tak, to możemy wypisać wynik. Zwróć uwagę, iż w przykładowym programie odpowiednikiem tablicy `a` jest tablica `h` a odpowiednikiem tablicy `w` jest tablica `e`).

Oto program (przy kompilacji należy dołączyć biblioteki i podać lokalizację modułów – patrz r. 11.3):

```
!   obliczenia metoda Hueckla dla polienow cyklicznych

program hueckel
  use la_precision
  use f95_lapack
  implicit none

  real(dp), allocatable :: h(:,:),e(:)
  integer info,i,j,n
  character jobz,uplo

! konstrukcja macierzy hamiltonianu (dolny trojkat)

  write(*,*) 'podaj liczbe atomow wegla'
  read(*,*) n

  allocate(h(n,n),e(n))

  h = 0.0

  do i=2,n
     h(i,i-1) = 1.0
  end do
  h(n,1) =  1.0

!    wywolanie procedury diagonalizaujacej
```

```
  jobz = 'V'
  uplo = 'L'

  call la_syev(h,e,jobz,uplo,info)

!  sprawdzamy, czy diagonalizacja sie powiodla
!  i wypisujemy wynik

  if (info/=0) then
     write(*,*) 'problemy'
  else
     write(*,*) ' '
     write(*,*) 'energie orbitali i wspolczynniki kombinacji'
     write(*,*) ' '
     do i = 1,n
        write(*,'(''E = alpha0'',sp,f10.4,''*beta0'')') e(i)
        write(*,'(10f10.5)') (h(j,i),j=1,n)
        write(*,*) ' '
     end do
  end if

  deallocate(h,e)

end program hueckel
```

Użycie odpowiednich modułów Lapacka95 osiągamy przy pomocy instrukcji `use` (patrz 11.3).

Deskryptor `sp` w specyfikacji formatu w instrukcji wypisującej energie oznacza żądanie poprzedzania liczby dodatniej znakiem +.

Przy porównaniu rozwiązań otrzymanych z użyciem powyższego programu z podręcznikiem należy uwzględnić, że dowolna kombinacja liniowa funkcji odpowiadających stanom zdegenerowanym jest też poprawnym rozwiązaniem. Stąd współczynniki w orbitalach dla benzenu mogą nie pokrywać się z rozwiązaniem danym wzorami 15.58-15.61 z cytowanego podręcznika A. Gołębiewskiego.

Przykład 14.

Błądzenie przypadkowe w jednym wymiarze.

Rozważmy pijanego osobnika rozpoczynającego ruch wzdłuż krawężnika od latarni (położenie latarni przyjmujemy za początek układu współrzędnych). Ze względu na stan upojenia alkoholowego krok do przodu i do tyłu jest jednakowo prawdopodobny. Należy znaleźć średnią odległość osobnika od latarni po wykonaniu *n* kroków.

Do symulacji przypadkowego ruchu wykorzystamy procedurę `random_number` generującą liczby pseudolosowe o rozkładzie równomiernym w przedziale (0,1) (patrz r. 11.1). W zależności od wartości zwróconej przez tę procedurę zwiększymy (krok do przodu) lub zmniejszymy (krok wstecz) zmienną przechowującą aktualną odległość pijaka od latarni. Długość kroku przyjmujemy za jednostkę długości, czyli zmiana wartości zmiennej to +1 lub –1. Wyboru między tymi wartościami możemy dokonać odejmując 0.5 od wartości liczby pseudolosowej i badając znak wyniku (wartości dodatnie i ujemne będą jednakowo prawdopodobne). Czynności te musimy powtórzyć `nkrok` razy, gdzie `nkrok` to liczba kroków wykonana przez delikwenta. Jest to treść najbardziej wewnętrznej pętli w programie.

Chcąc znaleźć średnie przesunięcie osobnika po `nkrok` krokach symulację powinniśmy powtórzyć i policzyć średnią z uzyskanych wyników. Czynności te realizuje druga pętla (ze zmienną sterującą `i`) – zwróć uwagę, że uśredniamy wartość bezwzględną przesunięcia.

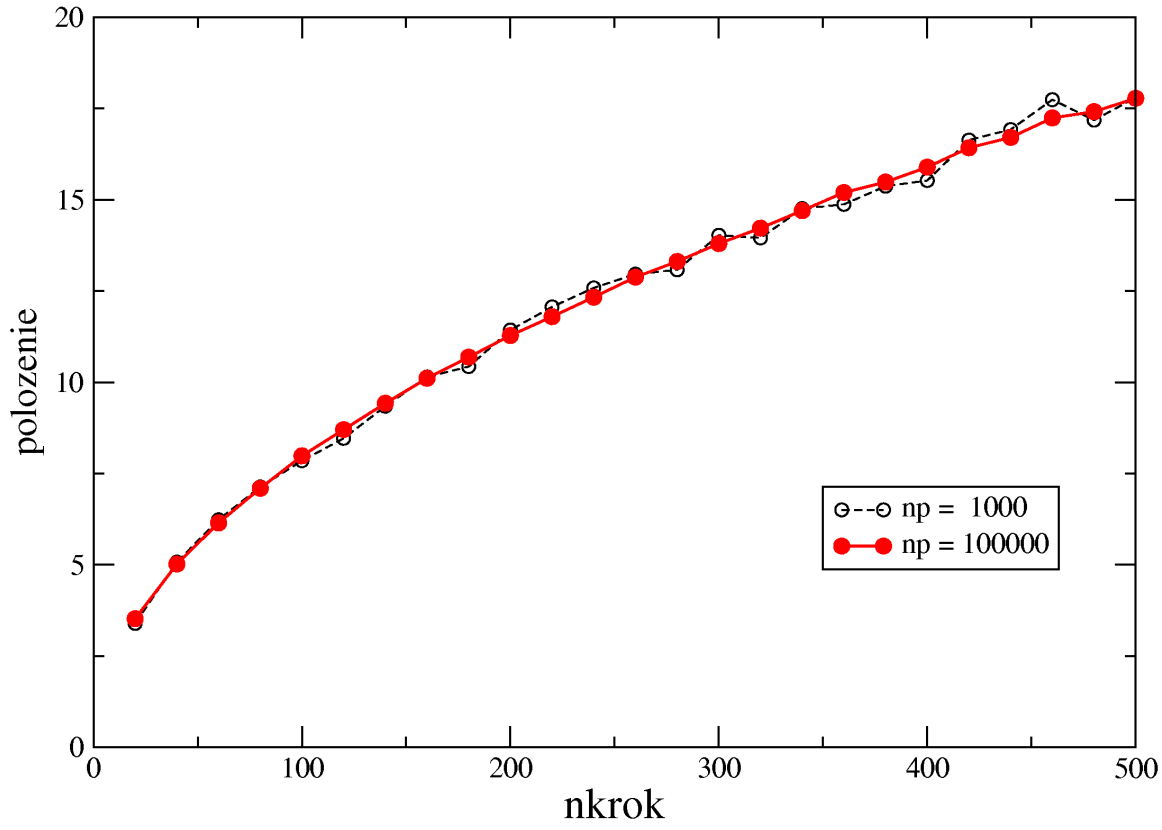
Ponieważ chcielibyśmy zbadać zależność przesunięcia pijaka od liczby wykonanych kroków, wyposażamy program w najbardziej zewnętrzną pętlę zmieniającą wartość `nkrok` w zadanych granicach. Pozostaje wczytać dane, zainicjować generator liczb procedurą `init_random_seed` (patrz 11.1) i wypisać wyniki.

Program (należy pamiętać o skompilowaniu go razem z plikiem initrs.f90 – patrz r. 11.1)

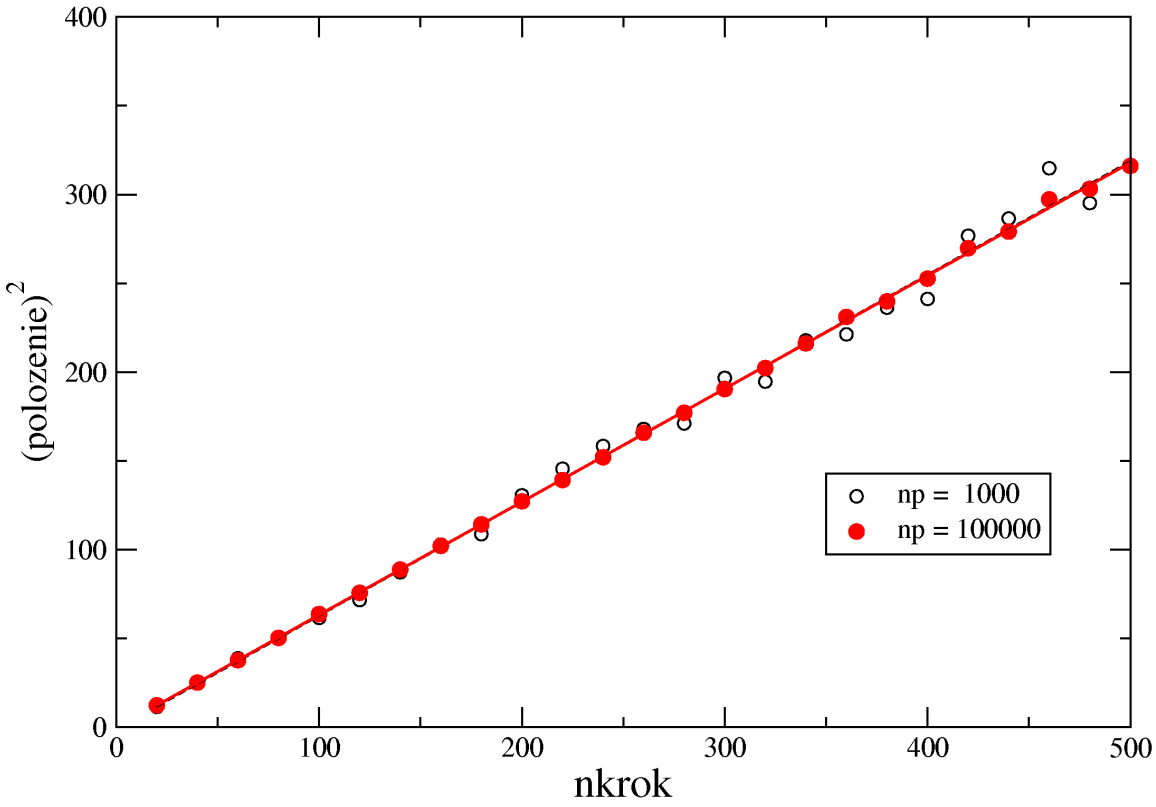
```
!       bladzenie przypadkowe w 1 wymiarze

program JasPijaczek
  use initrs
  implicit none
  real x
  integer np,nk1,nk2,ns,nkrok,i,j,ip,suma

  write(*,*) 'podaj zakres krokow (w.pocz, w.konc, przyrost)'
  read(*,*) nk1,nk2,ns
  write(*,*) 'podaj liczbe powtorzen'
  read(*,*) np

! inicjalizacja generatora liczb pseoudolosowych

  call init_random_seed()

  open(12,file='pijak.dat')

  do nkrok = nk1,nk2,ns
     write(*,*) 'nkrok = ',nkrok
     suma = 0
     do i = 1,np
        ip = 0
```

```
         do j = 1,nkrok
            call random_number(x)
            x = x - 0.5
            if (x<0) then
               ip = ip - 1
            else
               ip = ip + 1
            end if
         end do
         suma = suma + abs(ip)
      end do
      write(12,*) nkrok,suma/real(np)
      write(*,*) nkrok,suma/real(np)
   end do

   close(12)

end program JasPijaczek
```

Zobaczmy na przykładowe wyniki otrzymane dla 1000 i 100000 powtórzeń przy liczbie kroków zmieniającej się od 20 do 500:



Widać, że przy zwiększeniu liczby powtórzeń statystyczne fluktuacje maleją. Kształt krzywych coś nam już może sugerować; aby się upewnić, narysujmy wykres kwadratu średniego przesunięcia jako funkcję liczby kroków:



Linie proste to linie najlepszego dopasowania do punktów.

Wyraźnie widać, że średnie przesunięcie jest proporcjonalne do pierwiastka kwadratowego z liczby wykonanych kroków:

$$przesunięcie \sim \sqrt{nkrok}$$

Przykład 15.

Symulacja rozdziału na kolumnie chromatograficznej.

Symulację przeprowadzimy w sposób następujący. Załóżmy, że proces przemieszczania się substancji wzdłuż kolumny możemy potraktować jako serię następujących po sobie aktów, w których substancja z pewnym prawdopodobieństwem albo przemieszcza się wzdłuż kolumny o jednostkową odległość, albo pozostaje w miejscu.

W programie będziemy badać zachowanie się 3 substancji a, b i c różniących się prawdopodobieństwem przemieszczenia się wzdłuż kolumny. W trzech jednowymiarowych tablicach (`a`, `b`, `c`)będziemy przechowywać aktualne położenia cząstek. Ustalenie, czy cząstka przemieści się o jednostkową odległość sprowadza się do wylosowania liczby z przedziału (0,1) (procedura `random_number` – patrz rozdz. 11.1) i porównania jej z zadanym prawdopodobieństwem przemieszczenia cząstki. Po dokonaniu ustalonej liczby losowań dla wszystkich cząstek zliczamy cząstki zajmujące określone położenia. Wyniki tego zliczania przechowywane są w tablicach `na`, `nb`, `nc` (możliwe jest też, że sprawdzana cząstka opuściła już kolumnę, wtedy nie zwiększamy żadnego elementu tablicy).

Przy deklarowaniu tablic użyjemy parametru `dlugosc` określającego długość kolumny.

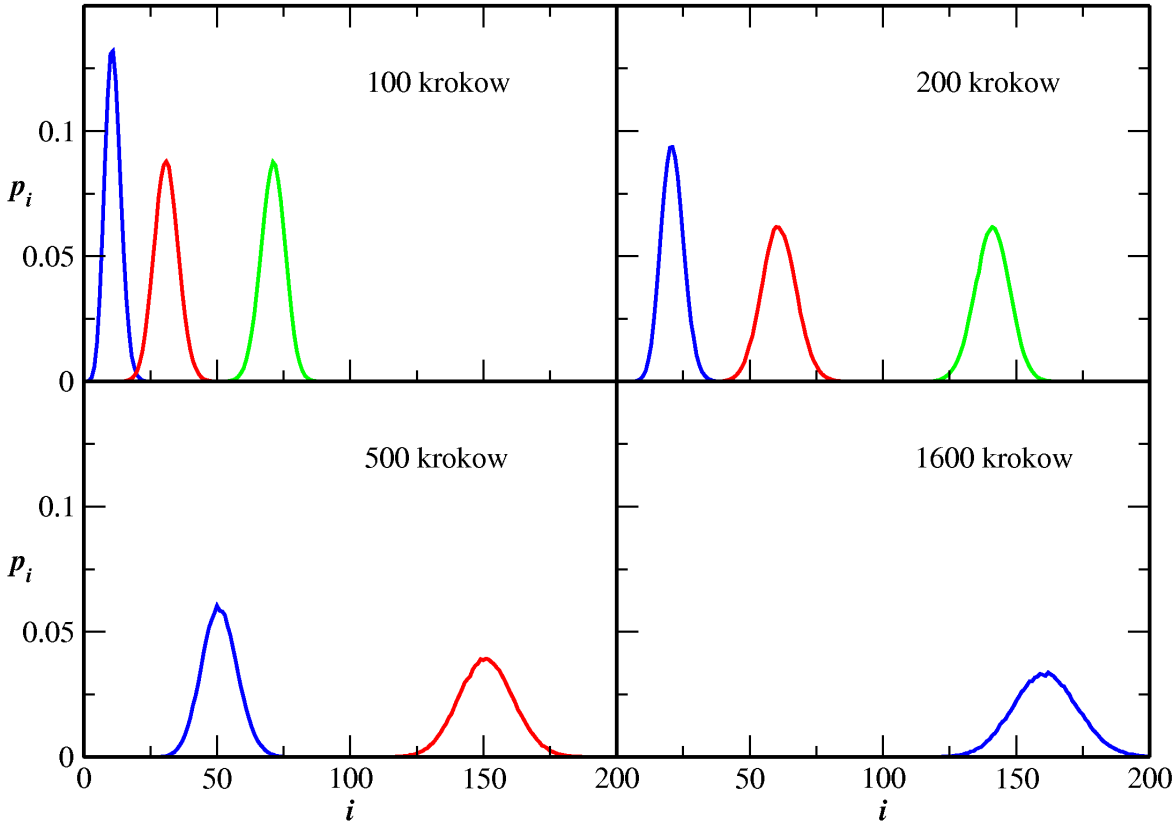
Wynikiem działania programu będzie plik zawierający w kolejnych kolumnach położenie w kolumnie i liczbę cząstek każdego rodzaju znajdujących się na tym odcinku kolumny.

Program (przy kompilacji należy dołączyć `initrs.f90` – p. 11.1)

```
!       symulacja rozdzialu na kolumnie chromatograficznej

program kolumna
  use initrs
  implicit none

  integer, parameter :: dlugosc=200
  integer, allocatable :: a(:),b(:),c(:)
  integer na(dlugosc),nb(dlugosc),nc(dlugosc)
  integer i,j,n,ncz
  real x(3),pa,pb,pc

  write(*,*) 'podaj prawdopodobienstwa pa,pb,pc'
  read(*,*) pa,pb,pc
  write(*,*) 'podaj liczbe czastek'
  read(*,*) ncz
  write(*,*) 'podaj liczbe krokow'
  read(*,*) n

  allocate(a(ncz),b(ncz),c(ncz))

  call init_random_seed()

!     startowe polozenia czastek

  do i = 1,ncz
     a(i) = 1
     b(i) = 1
```

```
      c(i) = 1
   end do

!     transport czastek wzdluz kolumny

   do i = 1,n
      do j = 1,ncz
         call random_number(x)
         if (x(1).le.pa) a(j) = a(j) + 1
         if (x(2).le.pb) b(j) = b(j) + 1
         if (x(3).le.pc) c(j) = c(j) + 1
      end do
   end do

!  zliczenie czastek zajmujacych okreslone polozenie

   do i = 1, dlugosc
      na(i) = 0
      nb(i) = 0
      nc(i) = 0
   end do

   do i = 1,ncz
      if (a(i)<=dlugosc) na(a(i)) = na(a(i)) + 1
      if (b(i)<=dlugosc) nb(b(i)) = nb(b(i)) + 1
      if (c(i)<=dlugosc) nc(c(i)) = nc(c(i)) + 1
   enddo

   open(13,file='kolumna.dat')
   do i = 1,dlugosc
      write(13,*) i,na(i)/real(ncz),nb(i)/real(ncz),nc(i)/real(ncz)
   end do

   deallocate(a,b,c)
   close(13)


end program kolumna
```

Poniżej przedstawiono wyniki symulacji dla 200000 cząstek substancji o następujących prawdopodobieństwach przemieszczenia: $p_a = 0.1$ (niebieska), $p_b = 0.3$ (czerwony), $p_c = 0.7$ (zielony) po 100, 200, 500 i 1600 krokach.



Najszybciej przemieszcza się substancja o największym p. Widać rozdział substancji i rozmywanie się pasm w trakcie procesu.