Лабораторна робота № 5
z-ПЕРЕТВОРЕННЯ

Завдання: Реалізувати обернене *z*-перетворення на мові програмування C++. Застосувати програму для довільної послідовності.

Аналогові фільтри проектуються з використанням перетворення Лапласа. Рекурсивні *цифрові фільтри* (*ЦФ*) розробляються з використанням паралельної методики, яка називається *z-перетворенням*. Загальна стратегія цих двох перетворень та ж сама: зондування імпульсного відгуку синусоїдами і експонентами для знаходження системних полюсів і нулів. Перетворення Лапласа має справу з диференціальними рівняннями, *s*-областю та *s*-площиною. Відповідно, *z*-перетворення має справу з різницевими рівняннями, *z*-областю та *z*-площиною. Однак ці дві методики не є дзеркальним відображенням один одного; *s*-площина реалізується в прямокутній системі координат, *z*-площина використовує полярну форму. Рекурсивні цифрові фільтри часто створюються, починаючи з одного з класичних аналогових фільтрів таких, як Баттерворта, Чебишева або еліптичного. Ряд математичних перетворень потім використовуються для отримання бажаного цифрового фільтра. *z*-перетворення дає базові знання для створення ЦФ.

z-область

Для розуміння *z*-перетворення почнемо з опису перетворення Лапласа і покажемо, як його можна модифікувати в *z*-перетворення. Перетворення Лапласа визначається наступним співвідношенням між сигналом в часовій області і сигналом *s*-області:

$$X(s) = \int_{t=-\infty}^{\infty} x(t)e^{-st}dt$$

Тут $x(t)$ – сигнал в часовій області і $X(s)$ – сигнал *s*-області. Ця формула аналізує сигнал часової області в термінах синусних та косинусних хвиль, які мають експоненційно мінливу амплітуду. Реалізується це шляхом заміни комплексної змінної s на еквівалентний вираз $\sigma + j\omega$. Використовуючи такий альтернативний запис, перетворення Лапласа приводиться до вигляду:

$$X(\sigma,\omega) = \int_{t=-\infty}^{\infty} x(t)e^{-\sigma t}e^{-j\omega t}dt$$

Якщо ми маємо справу тільки з дійсним сигналом в часовій області (звичайний випадок), верхня і нижня половина *s*-площини є дзеркальними відображеннями одна одної, і вираз $e^{-j\omega t}$ спрощується до простого косинуса і синуса. Ця формула визначає кожне положення на *s*-площині двома параметрами σ і ω. Величина в кожному положенні є комплексне число, що складається з дійсної та уявної частини. Для знаходження дійсної частини сигнал часової області множиться на косинусну хвилю з частотою ω і амплітудою, яка експоненційо змінюється згідно параметру загасання σ. Реальна частина $X(\sigma, \omega)$ знаходиться інтегруванням отриманого сигналу.

Величина уявної частини $X(\sigma, \omega)$ знаходиться схожим способом, тільки використовується синус замість косинуса.

Перетворення Лапласа можна замінити на z-перетворення в три кроки. Перший крок очевидний: зміна неперервного сигналу в дискретний. Це виконується заміною змінної t на номер відліку n, і заміною інтегрування на сумування:

$$X(\sigma,\omega) = \sum_{n=-\infty}^{\infty} x[n] e^{-\sigma n} e^{-j\omega n}$$

Зазначимо, що в $X(\sigma, \omega)$ використовуються круглі дужки, так як цей сигнал неперервний, а не дискретний. Хоча тепер ми маємо справу з дискретним сигналом часовій області $x[n]$, параметри σ і ω ще мають неперервну величину. Наступний крок полягає в перезаписі експоненційного виразу. Експоненційний сигнал математично може бути представлений двома способами:

$$y[n] = e^{-\sigma n} \quad \text{або} \quad y[n] = r^{-n}$$

Як показано на рисунку, приведеному нижче, обидві ці формули дають експонентну криву. Перший вираз контролює загасання сигналу через параметр σ. Якщо σ додатній, сигнал зменшується за величиною зі зростанням номера відліку n. Відповідно крива буде збільшуватися, якщо σ від'ємний. Якщо $\sigma = 0$, сигнал буде мати постійну величину рівну одиниці.

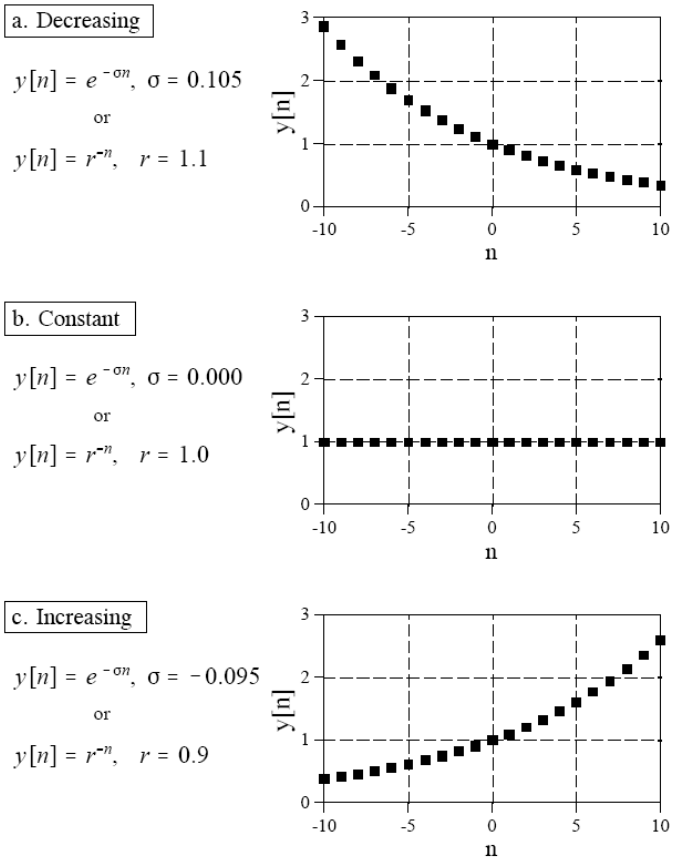

Рис. Експоненційні сигнали. Експоненти модна представити в двох різних математичних формах. Перетворення Лапласа використовує один спосіб, z-перетворення використовує інший спосіб.

Другий вираз використовує параметр *r*, який контролює загасання сигналу. Сигнал буде зменшуватися, якщо *r*> 1, і зменшуватися, якщо *r* <1. Сигнал буде мати постійну величину, коли *r* = 1. Ці дві формули відрізняються тільки способом вираження однієї і тієї ж величини. Один метод може переходити в інший, використовуючи такий вираз:

$$r^{-n} = [\mathrm{e}^{\ln(r)}]^{-n} = \mathrm{e}^{-n\ln(r)} = \mathrm{e}^{-\sigma n}$$

де: $\sigma = ln(r)$

Другий крок перетворення перетворення Лапласа в *z*-перетворення завершується використанням іншої експоненційної форми:

$$X(r,\omega) = \sum_{n=-\infty}^{\infty} x[n] r^{-n} e^{-j\omega}$$

Хоча це абсолютно правильний вираз для *z*-перетворення, він представлений не в найкомпактнішій формі для запису комплексного типу. Ця проблема долається в перетворенні Лапласа введенням нової комплексної змінної *s*, яка визначається, як $s = \sigma + j\omega$. Подібним чином ми будемо вводити нову змінну *z*:

$$z = r\, e^{j\omega}$$

Це визначення комплексної змінної *z*, як запис в полярній формі двох дійсних змінних *r* і ω. Третій крок в реалізації *z*-перетворення полягає в заміні *r* і ω на *z*. Така маніпуляція приводить нас до стандартної форми *z*-перетворення:

$$X(z) = \sum_{n=-\infty}^{\infty} x[n] z^{-n}$$

z-перетворення визначає співвідношення між сигналом часової області *x*[*n*] і сигналом *z*-області *X*(*z*).

Чому *z*-перетворення використовує r^n замість $e^{-\sigma n}$ і *z* замість *s*? Рекурсивні фільтри виконуються за допомогою набору рекурсивних коефіцієнтів. Аналізуючи ці системи в *z*-області, ми повинні конвертувати рекурсивні коефіцієнти в функцію перетворення *z*-області і навпаки. Визначення *z*-області таким чином (r^n і *z*) забезпечує найпростіший спосіб для перетворення між цими двома важливими представленнями. Дійсно, визначення *z*-області таким способом робить її тривіальною для переходу від одного представлення до іншого.

Нижчеприведений рисунок ілюструє різницю між *s*-площиною перетворення Лапласа і *z*-площиною *z*-перетворення. Розташування на *s*-площині визначається двома параметрами: σ – величиною експоненцій-

ного затухання уздовж горизонтальної осі і ω – величиною частоти уздовж вертикальної осі. Іншими словами, ці два дійсних параметра організовані у вигляді прямокутної системі координат. Ця геометрія виходить з визначення *s*, комплексної змінної, що представляє положення на *s*-площині зі співвідношення: $s = \sigma + j\omega$.

Для порівняння, *z*-область використовує змінні: *r* і ω, організовані у вигляді полярної системи координат. Відстань від початку координат *r* є величиною експоненціального затухання. Кут, виміряваний від додатної частини горизонтальної осі ω – це частота. Ця геометрія виходить з визначення $z = re^{-j\omega}$. Іншими словами, комплексна змінна, що представляє положення на *z*-площині, формується в полярній формі комбінацією двох дійсних параметрів.

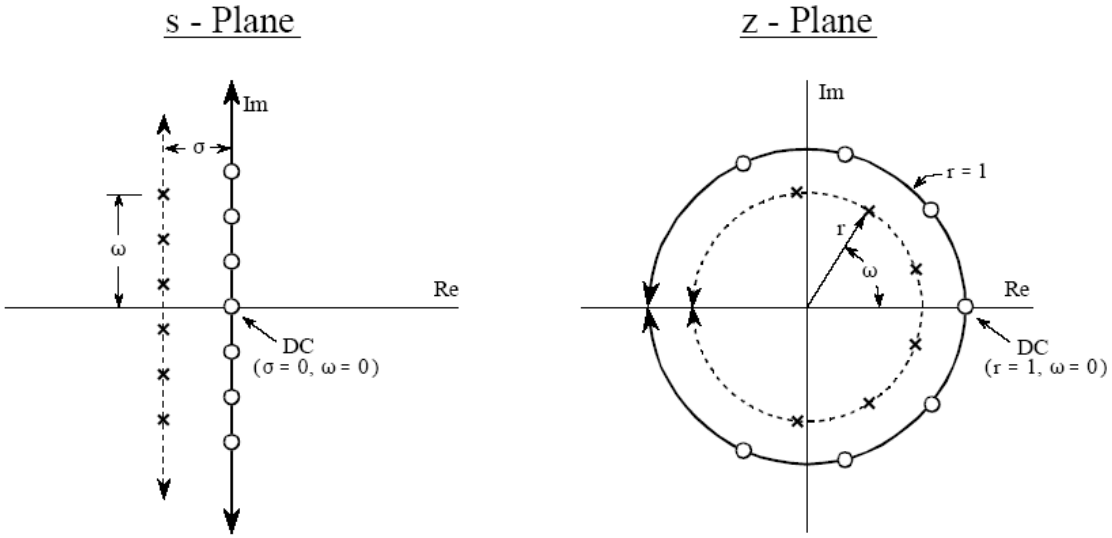

Рис. Співвідношення між *s*-площиною і *z*-площиною. *s*-площина є єпрямокутною системою координат з σ, що виражає відстань по дійсній (горизонтальній) осі, і з ω, що виражає відстань по уявній (вертикальній) осі. *z*-площина представлена в полярній формі за допомогою парметра *r*, який представляє відстань від початку координат, і параметра ω, який представляє кут, виміряний від додатної частини горизонтальної осі. Вертикальні лінії на *s*-площині (які показані на цьому прикладі для умовних знаків $\times$ і $\circ$) відповідають колам на *z*-площині.

Це призводить до того, що вертикальні лінії на *s*-площині відповідають колам на *z*-площині. Наприклад, *s*-площину на рисунку показує розташування $\times$ і $\circ$ (вертикальні лінії). Еквівалентні значення $\times$ і $\circ$ на *z*-площині лежать на колах з центром в початку координат. Це можна зрозуміти з представленого раніше співвідношення: $\sigma = \ln(r)$. Наприклад, вертикальна вісь на *s*-площині ($\sigma = 0$) відповідає на *z*-площині одиничному колу ($r = 1$). Вертикальні лінії лівої половини s-площини відповідають на *z*-площині колам, які містяться всередині одиничного кола. Аналогічним чином, вертикальні лінії в правій половині *s*-площини відповідають на

z-площині колам, які знаходяться зовні одиничного кола. Іншими словами, ліва і права сторони *s*-площини відповідають внутрішній і зовнішній сторонам одиничного кола. Наприклад, система з неперервними сигналами нестабільна, коли ○ виявляються в правій половині *s*-площини. Подібним чином система з дискретними сигналами нестабільна, коли ○ знаходяться зовні одиничного кола на *z*-площині. Коли сигнал в часовій області повністю дійсний (найбільш типовий випадок), верхня і нижня половина *z*-площини є дзеркальним відображенням одна одної, так як і в *s*-області.

Зверніть особливу увагу на те, як частотна змінна ω використовується в цих двох перетвореннях. *Неперервний* синусоїдальний сигнал може мати будь-яку частоту, від певного постійного значення до нескінченності. Це означає, що на *s*-площині ω може змінюватися від мінус нескінченності до плюс нескінченності. *Дискретний* сигнал може мати частоту тільки від певного сталого значення до половини частоти дискретизації. Тобто, частота може змінюватися від 0 до 0,5, коли вона виражається в долях частоти дискретизації, або від 0 до π, коли вона виражається як кругова частота ($\omega = 2\pi f$). Це співпадає з геометрією *z*-площині, коли ми інтерпретуємо ω, як кут, виражений в радіанах. Тобто, додатні частоти відповідають куту від 0 до π радіан, а від'ємні частоти відповідають куту від 0 до -π радіан. Оскільки *z*-площина виражає частоту іншим способом, ніж *s*-площина, то деякі автори публікацій використовують різні символи для їх розрізнення. Зазвичай використовують:

Ω для представлення частоти в *z*-області

ω для частоти в s-області.

Тут ми використовувати ω в обох випадках, але ви можете зустріти схоже позначення і в інших матеріалах по ЦОС.

На s-площині величини, які лежать на вертикальній осі, рівні частотному відгуку системи. Тобто, перетворення Лапласа, обчислене для σ = 0, аналогічне перетворенню Фур'є. Таким же чином, частотний відгук в *z*-області знаходиться на одиничному колі. Це можна побачити, обчисливши *z*-перетворення для $r = 1$, що спрощує формулу до дискретного перетворення Фур'є (ДПФ). У цьому випадку нульова частота буде в одиниці по горизонтальній осі *z*-площині. Додатні частоти розташовуються проти годинникової стрілки від нульової частоти, займаючи верхнє півколо. Аналогічним чином, від'ємні частоти йдуть за годинниковою стрілкою від нульової частоти, формуючи нижнє півколо. Додатні і від'ємні частоти спектра зустрічаються в спільній точці, $\omega = \pi$ і $\omega = -\pi$. Ця кругова геометрія також відповідає частотному спектру періодичного дискретного сигналу. Тобто, коли кут перевищує π, це ситуація аналогічна як і від 0 до π. Коли ви проходите по колу декілька разів, то будете бачите ту саму картинку знову і знову.

Перехід від неперервних систем до дискретних

Більшість типових методів переходу з *s*-області в *z*-область – це *білінійне перетворення*. Це математичний спосіб конформного відображення (conformal mapping), при якому комплексна площина алгебраїчно змінюється або перетворюється в іншу комплексну площину. Білінійне перетворення змінює $H(s)$ в $H[z]$ наступною підстановкою:

$$s \to \frac{2(1-z^{-1})}{T(1+z^{-1})}$$

Записуємо формулу для $H(s)$, потім замінюємо кожне s на вище наведений вираз. У більшості випадків використовується

$$T = 2\tan(1/2) = 1{,}093.$$

При цьому частотний рівень *s*-області від 0 до π радіан/секунду перетвориться в частотний рівень *z*-області від 0 до нескінченності радіан. Не вдаючись у подробиці, зазначимо, що білінійне перетворення володіє бажаною властивістю зміни *s*-області в *z*-область, при якому вертикальні лінії відображаються в коло.

Основні властивості z-перетворення

Послідовність	z-перетворення	Область збіжності
$x[n]$	$X(z)$	$R_{x-} < \lvert z\rvert < R_{x+}$
$y[n]$	$Y(z)$	$R_{y-} < \lvert z\rvert < R_{y+}$
$ax[n]+by[n]$	$aX(z)+bY(z)$	$\max[R_{x-},R_{y-}] < \lvert z\rvert < \min[R_{x+},R_{y+}]$
$x[n+n_0]$	$z^{n_0}X(z)$	$R_{x-} < \lvert z\rvert < R_{x+}$
$a^n x[n]$	$X(a^{-1}z)$	$\lvert a\rvert R_{x-} < \lvert z\rvert < \lvert a\rvert R_{x+}$
$nx[n]$	$-z\frac{dX(z)}{dz}$	$R_{x-} < \lvert z\rvert < R_{x+}$
$x^*[n]$	$X^*(z^*)$	$R_{x-} < \lvert z\rvert < R_{x+}$
$x[-n]$	$X(1/z)$	$1/R_{x+} < \lvert z\rvert < 1/R_{x-}$
$\mathrm{Re}\{x[n]\}$	$\frac{1}{2}\left[X(z)+X^*(z^*)\right]$	$R_{x-} < \lvert z\rvert < R_{x+}$
$\mathrm{Im}\{x[n]\}$	$\frac{1}{2i}\left[X(z)-X^*(z^*)\right]$	$R_{x-} < \lvert z\rvert < R_{x+}$
$x[n]\otimes y[n]$	$X(z)Y(z)$	$\max[R_{x-},R_{y-}] < \lvert z\rvert < \min[R_{x+},R_{y+}]$
$x[n]\cdot y[n]$	$\frac{1}{2\pi i}\oint_C X(v)Y(z/v)v^{-1}dv$	$R_{x-}R_{y-} < \lvert z\rvert < R_{x+}R_{y+}$

Програма на C для оцінки оберненого z-перетворення та перетворення послідовної структури в паралельну

Обчислення оберненого z-перетворення за допомогою методу розкладання в степеневий ряд або на елементарні дроби можна оформити у вигляді програми на мові C. Ця програма також використовується для перетворення передавальної функції $H(r)$ системи дискретного часу з послідовної структурри в паралельну. Оскільки зазначена програма досить велика, для ефективності було сформовано два програмних модуля `izt.c` і `ltilib.c`, що зберігаються в окремих файлах, які можна скомпілювати окремо, а потім з'єднати:

`izt.c` — програма обчислення оберненого z-перетворення через розкладання в степеневий ряд або на елементарні дроби, і перетворення передавальної функції $H(r)$, представленої у вигляді послідовної структури, в еквівалентну передавальну функцію, що має паралельну структуру, через розкладання на елементарні дроби;

`ltilib.c` — бібліотека функцій, в тому числі функцій **`power_series`** і **`partial_fraction`**.

Програма 1. `izt.c`

```
/*--------------*/
/*                                    */
/*  програма для:
(1) обчислення оберненого z-перетворення за допомогою статичних рядів
(2)  перетворення  передавальної  функції  H(z)  в  каскадну  форму  до
еквівалентної передавальної функції   */
/*    вхідний файл:   coeff.dat       */
/*    вихідний файл:  xdata.dat       */
/*--------------*/
#include   <stdio.h>
#include   <math.h>
#include   <dos.h>

#define    size  512
#define pi 3.141592654
#define    maxbits    30

typedef    struct     {
     double     real;
     double     imag;
     double     modulus;
     double     angle;
     }complex;
complex    poly_polar(double P2[], complex, int);
void pfraction();
complex polar_pole();
complex    cmul1(complex, complex);
complex    cdiv1(complex, complex);
complex    cadd1(complex, complex, int);
complex Dz(complex,int);
complex Dzz(complex, complex, complex);
complex    fixnp(complex);
void  pole_array(int);
void  poly_product(double C[], double D1[], double D2[], int);
void  poly_product1(double P1[], double AB[], int);
void  read_coeffs();
```

```
void print_pfcoeffs();
void printpar();
void cascade_parallel();
void zero_arrays();
double     cabs1(complex);
void izt_output();
void ncoeff(int);
double     A[size], B[size], Ni[size], Di[size];
double     ak[size], bk[size], h[size];
void power_series();
int   signx(double);
int   M, N, N1, M1, nstage, iopt,iir;
long npt;
complex    pk[10], ck[10], p[20];
float l1n, l2n, fmax;
double     B0;
FILE *in, *out, *fopen();/* global function + variable declarations */

/*------------------*/
main()
{
extern     double     A[size], B[size], Ni[size], Di[size];
extern     int   M, N, N1, M1, nstage, iopt, iir;
extern     long npt;
extern     FILE *in, *out, *fopen();

     iir=0;
     read_coeffs();              /*go read the system coeffs */
     poly_product1(Di,B,nstage);      /*form B(z) */
     poly_product1(Ni,A,nstage);      /*form A(z) */
     M1=M; N1=N;
     if(B[M]==0){                /*system of odd order*/
          M1=M-1;
          N1=M1;
     }
     printf("select desired operation\n");
     printf("0  for power series method of IZT\n");
     printf("1  for partial fraction coeffs estimation\n");
     printf("2  for cascade to parallel conversion\n");
     scanf("%d",&iopt);
     switch(iopt){
          case 0:
               printf("enter number of data points required\n");
               scanf("%ld",&npt);
               power_series();
               izt_output();
               break;
          case 1:
               pfraction();
               print_pfcoeffs();
               break;
          case 2:
               pfraction();          /* compute PF coeffs */
               cascade_parallel();        /* compute parallel coeffs for
H(z)*/
               print_pfcoeffs();     /* print PF coeffs */
               printpar();           /* print parallel coeffs for H(z) */
               break;
          default:   break;
     }
     exit(0);
}
/*--------------*/
```

```
/*function to evaluate the partial fraction coefficients   */

void pfraction()
{
     int  i;
     complex    nz[10], dz[10];
     extern     double    A[size], B[size];
     extern     complex pk[10], ck[10];
     extern     double B0;
     extern     int M, N1, M1, nstage;


     B0=A[M1]/B[M1];                 /* compute constant coeff */

          polar_pole();              /*   find   positions   of   poles
p1,p2,p3*/
          for(i=1; i<=M1; ++i){
               nz[i]=poly_polar(A,pk[i],M1);   /*evaluate N(z) at p1 */
               dz[i]=Dz(pk[i],i);              /*evaluate  D'(z)  at  p1
*/
               ck[i]=cdiv1(nz[i],dz[i]);       /*obtain      c1      =
N(z)/D'(z), z=p1 */
          }

}

/*---------------*/
/* function to evaluate the denominator polynomial D'(z)  */

complex    Dz(complex px, int npfc)
{
int  i,j;
complex    dk[10], dz;
extern     complex p[20];

pole_array(npfc);
                    /*evaluate Dk(z)=z(z-p[2])...(z-p[M]), z=px */
     if(M1==2){
          dz=Dzz(px,px,p[1]);
     }
     if(M1>2){
          j=1;
          dk[1]=Dzz(px,px,p[1]);
          for(i=2; i<(M1-1); ++i){
               dk[i]=Dzz(dk[i-1],px,p[i]);
               j=i;
          }
          dz=Dzz(dk[j],px,p[j+1]);
     }
     return(dz);
}

/*------------*/
/* function to evaluate factors of the form z(z-p) */

complex    Dzz(complex pa, complex pb, complex pc)
{
     complex    t1, t2, dz;
     t1=cmul1(pa,pb);
     t2=cmul1(pa,pc);
     dz=cadd1(t1,t2,0);
     return(dz);
}
```

```
/*-------------*/
/* function to re-order the poles */

void pole_array(npfc)
{
      int  i;

      switch(npfc){
      case 1:    /*pole array for c1*/
            for(i=1; i <=9; ++i){
                  p[i]=pk[i+1];
            }
            break;
      case 2:          /*pole array for c2 */
            p[1]=pk[1];
            for(i=2; i<=9; ++i){
                  p[i]=pk[i+1];
            }
            break;
      case 3:               /*pole array for c3 */
            p[1]=pk[1];
            p[2]=pk[2];
            for(i=3; i<=9; ++i){
                  p[i]=pk[i+1];
            }
            break;
      case 4:
            p[1]=pk[1];
            p[2]=pk[2];
            p[3]=pk[3];
            for(i=4; i<=9; ++i){
                  p[i]=pk[i+1];
            }
            break;
      case 5:

            for(i=1; i<=4; ++i){
                  p[i]=pk[i];
            }
            for(i=5; i<=9; ++i){
                  p[i]=pk[i+1];
            }
            break;

      case 6:

            for(i=1; i<=5; ++i){
                  p[i]=pk[i];
            }
            for(i=6; i<=9; ++i){
                  p[i]=pk[i+1];
            }
            break;

      case 7:

            for(i=1; i<=6; ++i){
                  p[i]=pk[i];
            }
            for(i=7; i<=9; ++i){
                  p[i]=pk[i+1];
            }
```

```
          break;
     case 8:

          for(i=1; i<=7; ++i){
               p[i]=pk[i];
          }
          p[8]=pk[9];
          p[9]=pk[10];
          break;
     case 9:
          for(i=1; i<=8; ++i){
               p[i]=pk[i];
          }
          p[9]=pk[10];
          break;

     case 10:
          for(i=1; i<=9; ++i){
               p[i]=pk[i];
          }

          break;
     }
}

/*------------*/
/*function to print the partial fraction coefficients */


void print_pfcoeffs()
{
int  k;
long i;
extern    int  M, M1;
extern    complex   ck[10], pk[10];
extern    double B0;

   printf("poles of the z-transform\n");
   printf("\n");
   printf("pk   \treal   \t\timag   \t\tmag   \t\tphase\n");
   for(k=1; k <= M1; ++k){
     pk[k].angle=pk[k].angle*180/pi;
     i=(long)(pk[k].angle/360);
     pk[k].angle=pk[k].angle - i*360;
     if(pk[k].angle < -180)
          pk[k].angle=pk[k].angle+360;
     if(pk[k].angle > 180)
          pk[k].angle=pk[k].angle - 360;
     printf("%d      \t%f         \t%f         \t%f         \t%f\n",k,
pk[k].real,pk[k].imag,pk[k].modulus,pk[k].angle);
   }
   printf("\n");
   printf("\n");
   printf("partial fraction coeffs \n");
   printf("\n");
   printf("B0 = %f \n",B0);
   printf("\n");
   printf("Ck   \treal   \t\timag   \t\tmag   \t\tphase\n");
   for(k=1; k <= M1; ++k){
     ck[k].angle=ck[k].angle*180/pi;
     i=(long)(ck[k].angle/360);
     ck[k].angle=ck[k].angle - i*360;
     if(ck[k].angle < -180)
```

```
          ck[k].angle=ck[k].angle+360;
     if(ck[k].angle > 180)
          ck[k].angle=ck[k].angle - 360;
     printf("%d      \t%f         \t%f         \t%f         \t%f\n",k,
ck[k].real,ck[k].imag,ck[k].modulus,ck[k].angle);
   }
   printf("press enter to continue \n");
   getch();
}
/*------------------*/
/* Function to compute coefficients for parallel realization   */

void cascade_parallel()
{
     int  i;
     extern     double     ak[size], bk[size];
     extern     complex pk[10], ck[10];
     extern     double B0;
     extern     int  M;

          for(i=0; i <M; ++i){
               ncoeff(i);
               /* compute coeffs for 2nd order sections*/
          }

}
/*-------------*/

void printpar()
{
     int  i, j;
     extern     double ak[size];
     extern     int  M;

     printf("stage   \tNi(z)\n");
     printf("\n");
     for(i=0; i <= M/2; ++i){
          j=2*i;
          printf("%d   \t%f   \t%f\n",i,ak[j], ak[j+1]);
     }
     j=0;
     printf("\n");
     printf("\n");
     printf("stage   \tDi(z)\n");
     printf("\n");
     for(i=0; i <= M/2; ++i){
          printf("%d \t%f \t%f \t%f\n",i,Di[j],Di[j+1],Di[j+2]);
          j=j+3;
     }

     printf("press enter to continue \n");
     getch();

 }
/*-----------------*/
/* function to compute numerator coeffs of second order sections from
partial fraction coeffs
*/

void ncoeff(int i)
 {
     int  j;
     complex    temp1, temp2, temp3;
```

```
      extern     double ak[size];
      extern     complex pk[10], ck[10];

      j=2*i;
      temp1=cadd1(ck[j+1], ck[j+2],1);
      ak[j]=temp1.real;
      temp1=cmul1(ck[j+1],pk[j+2]);
      temp2=cmul1(ck[j+2],pk[j+1]);
      temp3=cadd1(temp1,temp2,1);
      ak[j+1]=-temp3.real;
}

/*----------------*/
void  izt_output()
{
      long  i;
      extern     long npt;
      extern     double h[size];

      if((out=fopen("xdata.dat","w"))==NULL){
            printf("cannot open file xdata.dat\n");
            exit(1);
      }
      for(i=0; i < npt; ++i){
            fprintf(out,"%15e\n",h[i]);
      }
      fclose(out);
}
```

Програма2. `ltilib.c`

```
/*----------------------------  */
/*                              */
/*  Бібліотека основних функцій */
/*                              */
/*----------------------------  */

#include   <stdio.h>
#include   <math.h>
#include   <dos.h>

#define    size  512
#define pi 3.141592654
#define    maxbits    30

typedef    struct     {
      double     real;
      double     imag;
      double     modulus;
      double     angle;
      }complex;

complex    poly_polar(double P2[], complex, int);
complex polar_pole();
complex    cmul1(complex, complex);
complex    cdiv1(complex, complex);
complex    cadd1(complex, complex, int);
complex    fixnp(complex);
void  poly_product(double C[], double D1[], double D2[], int);
void  poly_product1(double P1[], double AB[], int);
```

```
void read_coeffs();
double     cabs1(complex);
void zero_arrays();
void power_series();
int  signx(double);
/*---------------------------*/
/* function to read coefficient values  of IIR or FIR systems*/

void read_coeffs()
{
     int  i,j;
     extern     int M,N,nstage, iir;
     extern  double A[size], B[size], Ni[size], Di[size];
     extern     FILE *in, *out, *fopen();


     if((in=fopen("coeff.dat","r"))==NULL){
          printf("cannot open file coeff.dat\n");
          exit(1);
     }

     zero_arrays();  /*initialise all global arrays */

     if(iir==1){
          printf("specify type of system \n");
          printf("0  for IIR system \n");
          printf("1  for FIR system \n");
          scanf("%d",&iir);
     }
     switch(iir){

     case 0:    /* read IIR filter coefficients for H(z) in cascade*/
          fscanf(in,"%d", &nstage);
          j=0;
          for(i=0; i<nstage; ++i){
               fscanf(in,"%lf %lf %lf", &Di[j], &Di[j+1], &Di[j+2]);
               fscanf(in,"%lf %lf %lf", &Ni[j], &Ni[j+1], &Ni[j+2]);
               j=j+3;
          }
          M=nstage*2; N=M;
          break;
     case 1:    /* read FIR coefficients */
          fscanf(in,"%d",&N);
          for(i=0; i <N; ++i){
               fscanf(in,"%15e",&A[i]);
          }
          B[0]=1.0; M=N;
          break;
     default:
          printf("invalid option selected \n");
          printf("press enter to continue \n");
          getch();
          break;
     }
     fclose(in);
}
/*---------------------------*/
/*function performs complex addition */

complex    cadd1(complex pa, complex pb, int nadd)
{
     long  k;
     complex np;
```

```
      np.real=0; np.imag=0;
      if(nadd==0){
            np.real=pa.real - pb.real;
            np.imag=pa.imag - pb.imag;
      }
      if(nadd==1){
            np.real=pa.real + pb.real;
            np.imag=pa.imag + pb.imag;
      }
      np.modulus=sqrt(np.real*np.real + np.imag*np.imag);
      np=fixnp(np);
      return(np);
}
/*-------------------------*/
/* Function to adjust angles of complex numbers to their correct values */

complex    fixnp(complex pa)
{
      long  k;
      complex np;

      np=pa;
      if(fabs(pa.real) < 0.0000000008){
            np.real=0;
      }
      if(fabs(pa.imag) < 0.0000000008){
            np.imag=0;
      }
      if(np.real==0 && np.imag==0){
            np.angle=0;
            return(np);
      }
      if(np.real==0 && np.imag!=0){
            if(np.imag > 0)
                  np.angle=pi/2;
            if(np.imag < 0)
                  np.angle=1.5*pi;;
            return(np);
      }
      if(np.real!=0 && np.imag==0){
            if(np.real>0)
                  np.angle=0;
            if(np.real<0)
                  np.angle=pi;
            return(np);
      }
      np.angle=atan(np.imag/np.real);
      if(np.real < 0)
            np.angle=np.angle+pi;
      k=(long)(np.angle/2*pi);
      np.angle=np.angle - k*2*pi;
      return(np);
}
/*-------------------------------*/
/* function to multiply two complex numbers */
complex    cmul1(complex pa, complex pb)
{
      complex np;
      np.real=0; np.imag=0;
      np.modulus=pa.modulus*pb.modulus;
      np.real=np.modulus*cos(pa.angle + pb.angle);
      np.imag=np.modulus*sin(pa.angle + pb.angle);
      np=fixnp(np);
```

```
     return(np);
}

/*-----------------------------*
/* function to divide two complex numbers  */
complex cdiv1(complex pa, complex pb)
{
     complex  np;
     np.real=0; np.imag=0;
     np.modulus=pa.modulus/pb.modulus;
     np.real=np.modulus*cos(pa.angle - pb.angle);
     np.imag=np.modulus*sin(pa.angle - pb.angle);
     np=fixnp(np);
     return(np);
}

/*---------------------------------*/

/*Routine to compute and express the pole of a second order section
in polar and rectangular forms. The denominator polynomial has the form

     D(z) = 1 + b1z^-1 + b2z^-2. The pole is represented as:

*/
complex    polar_pole()
{
     int   i,k, j=0;
     complex    px, ptemp;
     double     temp;
     extern     int nstage;
     extern     double Di[size];
     extern     complex pk[10];

     for(i=0; i<nstage; ++i){
          temp=Di[1+j]*Di[1+j] - 4*Di[2+j];
          if(temp >=0){               /*poles are real */
               px.imag=0; ptemp.imag=0;
               if(Di[2+j]==0){       /*simple poles */
                    px.real=-Di[1+j];
                    ptemp.real=0;
               }
               else{
                    px.real=(-Di[1+j] + sqrt(temp))/2;
                    ptemp.real=(-Di[1+j] - sqrt(temp))/2;
               }
               px.modulus=fabs(px.real);
               ptemp.modulus=fabs(ptemp.real);
               px.angle=0; ptemp.angle=0;
               if(px.real<0)
                    px.angle=acos(-1);
               if(ptemp.real<0)
                    ptemp.angle=acos(-1);
          }
          else{                       /* complex conjugate poles */
               px.modulus=sqrt(fabs(Di[2+j]));
               px.angle=acos(-Di[1+j]/(2*px.modulus));
               px.real=px.modulus*cos(px.angle);
               px.imag=px.modulus*sin(px.angle);
               ptemp.modulus=px.modulus;
               ptemp.angle=-px.angle;
               ptemp.real=px.real;
               ptemp.imag=-px.imag;
          }
```

```
            k=2*i+1;
            pk[k]=px;
            pk[k+1]=ptemp;
            j=j+3;
      }
}
/*--------------------------
Routine to evaluate a given polynomial, of order n, at z=pi. The result
is returned as a complex number in polar and rectangular forms
*/

complex    poly_polar(double P2[], complex pj, int n)
{
      complex    np;
      int  i, k;

      i=0; np.real=0; np.imag=0;

      for(i=0; i<n; ++i){
            k=n-i;
            np.real=np.real + P2[i]*pow(pj.modulus,k)*cos(k*pj.angle);
            np.imag=np.imag + P2[i]*pow(pj.modulus,k)*sin(k*pj.angle);
      }
      np.real=np.real+P2[n];
      np.modulus=sqrt(np.real*np.real+np.imag*np.imag);
      np.angle=atan(np.imag/np.real);
      if(np.real < 0)
            np.angle=np.angle+pi;
      return     (np);
}

/*--------------------------
Function to compute the absolute magnitude of a complex number
*/

double     cabs1(complex z)
{
      double temp;
      temp=z.real*z.real + z.imag*z.imag;
      temp=sqrt(temp);
      return(temp);
}

/*------------------------------ */
/* Function to produce a polynomial in direct form */

void poly_product1(double P1[], double AB[], int nsect)
{

      double  C[size], D1[size], D2[size];
      int  i, NN;

            if(nsect<2){
                  for(i=0; i< 3; ++i){
                        AB[i]=P1[i];
                  }

            return;
            }
            D1[0]=P1[0];    /* retrieve F1(z) */
            D1[1]=P1[1];
            D1[2]=P1[2];
```

```
          D2[0]=P1[3];     /* retrieve F2(z) */
          D2[1]=P1[4];
          D2[2]=P1[5];
          NN=2;
          poly_product(C, D1, D2, NN);     /* compute F1(z)F2(z) */


     if(nsect>2){

          D1[0]=C[0];      /* retrieve F1(z)F2(z) */
          D1[1]=C[1];
          D1[2]=C[2];
          D1[3]=C[3];
          D1[4]=C[4];
          D2[0]=P1[6];     /* retrieve F3(z) */
          D2[1]=P1[7];
          D2[2]=P1[8];
          D2[3]=0.0;
          D2[4]=0.0;
          NN=4;
          poly_product(C,  D1,  D2,  NN);  /*  compute  the  product
F1(z)F2(z)F3(z) */

     }
     if(nsect>3){

          D1[0]=C[0];      /* retrieve F1(z)F2(z)F3(z) */
          D1[1]=C[1];
          D1[2]=C[2];
          D1[3]=C[3];
          D1[4]=C[4];
          D1[5]=C[5];
          D1[6]=C[6];

          D2[0]=P1[9];     /* retrieve F4(z) */
          D2[1]=P1[10];
          D2[2]=P1[11];
          D2[3]=0.0;
          D2[4]=0.0;
          D2[5]=0.0;
          D2[6]=0.0;

          NN=6;
          poly_product(C, D1, D2, NN);     /*                    compute
[F1(z)F2(z)][F2(z)F4(z)] */

     }
     if(nsect>4){

          D1[0]=C[0];      /* retrieve F1(z)F2(z)F3(z)F4(z) */
          D1[1]=C[1];
          D1[2]=C[2];
          D1[3]=C[3];
          D1[4]=C[4];
          D1[5]=C[5];
          D1[6]=C[6];
          D1[7]=C[7];
          D1[8]=C[8];

          D2[0]=P1[9];     /* retrieve F5(z) */
          D2[1]=P1[10];
          D2[2]=P1[11];
          D2[3]=0.0;
```

```
          D2[4]=0.0;
          D2[5]=0.0;
          D2[6]=0.0;
          D2[7]=0.0;
          D2[8]=0.0;

          NN=8;
          poly_product(C, D1, D2, NN);    /*                    compute
[F1(z)F2(z)][F2(z)F4(z)F5(z)] */

          }
               /* save results */

          for(i=0; i<(2*nsect+1); ++i){
               AB[i]=C[i];
          }

}

/*-----------------------------
Function to compute the product of two polynomials
*/

void poly_product(double C[], double D1[], double D2[], int NN)
{
     int  i, j, k, j1;
     double     sum1=0.0, sum2=0.0;

     for(i=0; i<size; ++i){     /* initialize the result array */
          C[i]=0.0;
     }
     for(i=0; i < (NN+1); ++i){

          j=NN-i;
          j1=i+NN;
          for(k=0; k < (j+1); ++k){
               sum1=sum1+D1[k]*D2[j-k];
               sum2=sum2+D1[i+k]*D2[NN-k];
          }
          C[j]=sum1;
          C[j1]=sum2;
          sum1=0.0; sum2=0.0;
     }
}
/*-----------------------------*/
/* Function to zero global arrays used as inputs */

void zero_arrays()
{

     long i;
     extern     double     A[size], B[size], Ni[size], Di[size];
     extern     double     ak[size], bk[size], h[size];

     for(i=0; i <size; ++i){
          A[i]=0.0;
          B[i]=0.0;
          Ni[i]=0;
          Di[i]=0;
          ak[i]=0;
          bk[i]=0;
          h[i]=0;
     }
```

```
}
/*---------------------------------*/
/* Function to compute the izt using the power series method */

void power_series()
{
      int   i, k;
      long  n;
      double     sum1, sum2, sum3=0;
      double temp;
      extern     double A[size], B[size], h[size];
      extern     float l1n, l2n;
      extern     long npt;
      extern     int M;

      /*compute h[n] recursively */

      h[0]=A[0]/B[0];
      sum2=h[0];
      for(n=1; n<npt; ++n){
            sum1=0.0;
            k=n;
            if(n>M)
                  k=M;
            for(i=1; i<=k; ++i){
                  sum1=sum1+h[n-i]*B[i];
            }
            h[n]=(A[n] - sum1)/B[0];
            temp=signx(h[n])*h[n];
            sum2=sum2+temp;
            sum3=sum3+temp*temp;
      }
      l1n=sum2;
      l2n=sqrt(sum3);
}
/*-------------------------*/
int   signx(double x)
{
      int   temp;
      if(x < 0)
            temp=-1;
      else
            temp=1;
      return(temp);
}
```

Метод статичного ряду

Обернене z-перетворення $x(n)$ обчислюється рекурсивним способом за допомогою функції `power_series()` (див. програму), в основі якої лежить наступне рівняння:

$$x(n) = \left[b_n - \sum_{i=1}^{n} x(n-i)a_i \right] \Big/ a_0, \quad n = 1, 2, \ldots,$$

де

$$x(0) = b_0/a_0.$$

Для того, щоб можна було скористатися програмою для обчислення оберненого z-перетворення методом розкладання в степеневий ряд, z-образ

повинен бути представлений або безпосередньо, або у вигляді послідовної структури:

$$X(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \ldots + b_N z^N}{a_0 + a_1 z^{-1} + a_2 z^{-2} + \ldots + a_M z^M}$$

безпосередній вигляд

$$X(z) = \prod_{k=i}^{K} X_i(z)$$

послідовна структура

де $X_i(z)$ – ланка другого порядку, якої задається як

$$X_i(z) = \frac{b_{0i} + b_{1i} z^{-1} + b_{2i} z^{-2}}{1 + a_{1i} z^{-1} + a_{2i} z^{-2}}.$$

Для роботи програми потрібно створити файл вхідних даних під назвою `coeff.dat`. У цьому файлі записується кількість каскадів *K* (для безпосереднього представлення *K* = 1) і коефіцієнти чисельника і знаменника z-образу. Користуватися таким вхідним файлом дуже зручно, оскільки це позбавляє від необхідності вводити коефіцієнти вручну, а при цьому можна допустити помилку. Більш того, це дозволяє використовувати результати одного дослідження в якості вхідних даних для іншої програми.