004
Б86

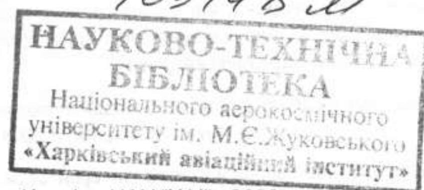
МИНИСТЕРСТВО НАУКИ И ОБРАЗОВАНИЯ УКРАИНЫ
Национальный аэрокосмический университет им. М. Е. Жуковского
«Харьковский авиационный институт»

К.А. Бохан, Г.А. Кучук

МЕТОДИ ЦИФРОВОЇ ОБРОБКИ СИГНАЛІВ

Програмна реалізація з використанням бібліотеки STL

Навчальний посібник


103448 м
НАУКОВО-ТЕХНІЧНА БІБЛІОТЕКА
Національного аерокосмічного університету ім. М.Є.Жуковського
«Харківський авіаційний інститут»

Харків НАУ "ХАІ" 2008

УДК 004.052:681.324

Бохан К.О., Кучук Г.А. Методи цифрової обробки сигналів. Програмна реалізація з використанням STL: Навч. посібник. – Х.: Нац. аерокосм. ун-т „Харк. авіац. ін-т", 2008. – 84 с. – Укр. мов.

Бохан К.А., Кучук Г.А. Методы цифровой обработки сигналов. Программная реализация с использованием STL: Учебное пособие. – Х.: Нац. аэрокосм. ун-т „Харьк. авиац. ин-т", 2008. – 84 с. – Укр. яз.

У навчальному посібнику наведені короткі теоретичні відомості про методи цифрової обробки сигналів та приклади їхньої програмної реалізації з використанням стандартної бібліотеки шаблонів (STL). Розглядаються методи визначення фізичних характеристик цифрових сигналів та дискретних імовірнісних джерел, властивості статистичних кодерів дискретних сигналів, дискретні ортогональні та вейвлет-перетворення сигналів, методи корекції візуальних характеристик та медіанної фільтрації цифрових зображень.

Посібник призначений для студентів спеціальностей «Комп'ютерні системи та мережі», «Спеціалізовані комп'ютерні системи», які навчаються по програмах підготовки бакалаврів, магістрів, а також для спеціалістів, які займаються розробкою програмного забезпечення для цифрової обробки сигналів.

В учебном пособии приведены короткие теоретические сведения о методах цифровой обработки сигналов и примеры их программной реализации с использованием стандартной библиотеки шаблонов (STL). Рассматриваются методы определения физических характеристик цифровых сигналов и дискретных вероятностных источников, свойства статистических кодеров дискретных сигналов, дискретные ортогональные и вейвлет-преобразования сигналов, методы коррекции визуальных характеристик и медианной фильтрации цифровых изображений.

Пособие предназначено для студентов специальностей «Компьютерные системы и сети», «Специализированные компьютерные системы», обучающихся по программам подготовки бакалавров, магистров, а также для специалистов, занимающихся разработкой программного обеспечения для цифровой обработки сигналов.

Іл. 88, табл. 3, бібліогр. 17 назв.

Рецензенти: ***професор*** кафедри автоматизації та комп'ютерних технологій Харківського національного технічного університету сільського господарства імені Петра Василенка, Заслужений винахідник України, Почесний радист СРСР, доктор технічних наук, професор КРАСНОБАЄВ Віктор Анатолійович;

доцент кафедри відео-, аудіо- и кінотехніки Національного технічного університету «ХПІ» кандидат фізико-математичних наук, старший науковий співробітник МОЖАЄВ Олександр Олександрович.

Затверджено до друку редакційно-видавничою комісією університету, протокол № 5 от 18.02.2008

© Національний аерокосмічний університет ім. М.Є. Жуковського "Харківський авіаційний інститут", 2008

ВСТУП

В навчальному посібнику наведені короткі теоретичні відомості про методи цифрової обробки сигналів та приклади їхньої програмної реалізації з використанням бібліотеки стандартних шаблонів (STL). Посібник складається з восьми розділів. У першому розділі наведені короткі відомості про бібліотеку STL, описана основні компоненти бібліотеки та розглянуто способи їх використання. Наступні розділи містять теоретичні відомості про відповідні методи цифрової обробки та приклади їх програмної реалізації з використанням бібліотеки STL. Зміст цих розділів наступний:

– в другому розділі розглядаються методи визначення фізичних характеристик цифрових сигналів;

– в третьому розділі розглядаються методи дискретних ортогональних перетворень сигналів в базисах Хаара, Уолша та дискретно-косинусне перетворення;

– у четвертому розділі приведені теоретичні основи вейвлет-перетворень та розглянуто метод швидкого вейвлет-перетворення в базисі Хаара;

– у п'ятому розділі розглянуто характеристики дискретних імовірнісних джерел та методи їх визначення;

– у шостому розділі розглядаються параметри статистичних кодерів дискретних сигналів, за допомогою яких можливо виконувати порівняння ефективності різноманітних статистичних кодерів;

– у сьомому розділі описані методи корекції візуальних характеристик цифрових зображень: лінійне контрастування, соляризація та препарування зображень;

– у восьмому розділі приведені теоретичні основи медіанної фільтрації цифрових зображень.

Для більш глибокого вивчення питань, які розглянуті у посібнику, рекомендується наступна література:

- для першого та другого розділів: [1, 2, 4, 11, 12];
- для третього розділу: [3, 9, 15, 16, 17];
- для четвертого та п'ятого розділів: [8, 10];
- для шостого та сьомого розділів: [5, 6, 7, 8, 13, 14].

1. ОСНОВИ ВИКОРИСТАННЯ СТАНДАРТНОЇ БІБЛІОТЕКИ ШАБЛОНІВ STL

1.1. Загальні відомості про бібліотеку STL

При створенні програмного забезпечення (ПЗ), що реалізує методи цифрової обробки сигналів, перед програмістом виникають наступні проблемні питання:

1) управління розподілом пам'яті;

2) ефективність програмного коду.

Перша проблема пов'язана з тим, що програмістові на етапі створення ПЗ невідомий об'єм даних, що обробляється. Це призводить до необхідності використання механізму динамічного розподілу пам'яті під час виконання програми.

Розглянемо приклад програмного коду на С++, що реалізує динамічний розподіл і звільнення пам'яті для двовимірного масиву даних довільної розмірності (рис. 1.1).

```
// створення динамічного масиву short
short** New2 (int m,  int n) // m × n – розмірність масиву даних
{ short**  data;
  data = new short*[m];     // крок 1 – виділення пам'яті для рядка.
 for (int j = 0; j < m; j++)
  data[j]= new short[n];  // крок 1 – виділення пам'яті для стовпців
  return data;
}

// видалення динамічного масиву
void Delete2 (short **temp, unsigned int m)
{  short **data = temp;
   for (unsigned i = 0; i < m;  i++)
          delete[] data[i];          // крок 1 – видалення стовпців
  delete[] data;    // крок 2 – видалення рядків
}
```

Рис. 1.1. Функції, що виконують динамічний розподіл і звільнення пам'яті для двовимірного масиву даних

Дані функції викликаються кожного разу при виділенні (звільненні) пам'яті. Окрім цього, при виділенні (звільненні) пам'яті необхідно перевіряти доступність необхідного об'єму пам'яті і контролювати всі можливі виняткові ситуації, пов'язані з помилками виділення (звільнення) пам'яті.

Розглянемо ситуацію, коли необхідно збільшити об'єм вже виділеної пам'яті. В цьому випадку поступають таким чином: виділяють нову ділянку динамічної пам'яті необхідного об'єму, копіюють дані з існуючої ділянки, яку потім звільняють. Як бачимо, необхідність включення в програмний код процедур управління динамічною пам'яттю призводить до збільшення трудомісткості створення подібного ПЗ, а також до зниження його надійності і ефективності.

Значно спростити створення ПЗ, яке реалізує методи цифрової обробки сигналів (ЦОС), а також підвищити його надійність і ефективність, дозволяє застосування стандартної бібліотеки шаблонів (STL), яка включена до стандарту C++. Стандартна бібліотека шаблонів надає набір добре сконструйованих і погоджено працюючих разом узагальнених компонентів C++. Особлива турбота була проявлена для забезпечення того, щоб всі шаблонні алгоритми працювали не тільки із структурами даних в бібліотеці, але також і з вбудованими структурами даних C++. Бібліотека дозволяє програмістам використовувати бібліотечні структури даних разом із своїми власними алгоритмами, а бібліотечні алгоритми – із своїми власними структурами даних. Добре визначені семантичні вимоги і вимоги складності гарантують, що компонент користувача працюватиме з бібліотекою і що він працюватиме ефективно. Ця гнучкість забезпечує широке застосування бібліотеки. При створенні STL багато зусиль було витрачено на те, щоб кожен шаблонний компонент в бібліотеці мав узагальнену реалізацію, ефективність виконання якої відрізняється від ефективності ручного програмного коду в межах декількох відсотків.

Бібліотека містить п'ять основних видів компонентів:

– алгоритм (*algorithm*): визначає обчислювальну процедуру;

– контейнер (*container*): управляє набором об'єктів в пам'яті;

– ітератор (*iterator*): забезпечує для алгоритму засіб доступу до змісту контейнера;

– функціональний об'єкт (*function object*): інкапсулює функцію в об'єкті для використання іншими компонентами;

– адаптер (*adaptor*): адаптує компонент для забезпечення різноманітних інтерфейсів.

Алгоритми – це рецепти для виконання певних завдань, таких, наприклад, як сортування масиву або пошук певного значення в списку.

Контейнер – це елемент даних, схожий на масив, який може містити декілька значень. Контейнери в STL є однорідними, тобто вони можуть містити значення тільки однакового типу. За допомогою шаблонів контейнерів можливо реалізувати масиви, списки, черги, дерева, хеш-таблиці та інші структури. При цьому реалізується автоматичне керування динамічною пам'яттю, доступ до елементів контейнера і безліч інших функцій.

Ітератори – це об'єкти, що дозволяють пересуватися по об'єкту-контейнеру так само, як звичайні покажчики дозволяють пересуватися по масиву; вони є узагальненням покажчиків.

Функціональні об'єкти – це об'єкти, що діють як функції; вони можуть бути об'єктами класу або покажчиками на функції (що включають ім'я функції, оскільки саме воно діє як покажчик).

1.2. Використання контейнерів. Контейнер vector

Бібліотека STL дозволяє побудувати безліч контейнерів, включаючи масиви, черги і списки, і виконувати різні операції з ними, зокрема пошук, сортування і заповнення випадковими значеннями.

Всі можливості масивів реалізуються за допомогою контейнера «vector», тому його і використовуватимемо для зберігання відліків дискретних сигналів. Розглянемо основні можливості цього контейнера. Вектор містить набір подібних значень, до яких можливий довільний доступ. Це означає, що можна використовувати індекс для безпосереднього доступу, наприклад, до десятого елементу масиву без необхідності доступу до дев'яти попередніх елементів. Таким чином, можна створити об'єкт класу vector, привласнити один такий об'єкт іншому і використовувати оператор [] для забезпечення доступу до елементів вектора. Шаблон класу vector в заголовному файлі vector (у ранніх версіях vector.h).

Для створення об'єкту шаблону vector застосовується звичайний запис vector<type>, що несе інформацію про те, який тип буде використаний. Крім того, шаблон класу vector динамічно розподіляє пам'ять, і тому при ініціалізації можна використовувати аргумент, що визначає, скільки елементів вектора потрібно створити (рис. 1.2).

Після створення векторного об'єкту перевантаження оператора [] дозволяє використовувати звичайну форму запису масиву для доступу до окремих елементів масиву (наприклад, одержання значення шостого елемента – x[5]), так і за допомогою ітераторів (посилань спеціального виду) (рис. 1.3).

```
#include vector
using namespace std;
vector<int> ratings(5); // вектор з п'яти значень типу int
vector<float> x(100); // вектор типу float
                        // з 0 – 100 елементами, що дорівнюють 0.
int n;
cin » n ;
vector<double> scores(n); // вектор з n значень типу double
```

Рис. 1.2. Створення об'єктів на основі шаблону **vector**

```
ratings[0] = 9;
for (int i = 0; i < n; i++)
cout « scores [i] « endl;
vector<float>::iterator ix = x.begin(); // оголошення ітератора ix
  // і ініціалізація адресою на перший елемент вектора x
float b = *(ix + 6); // одержання значення шостого елемента вектора x
ix = x.end()-1; // ініціалізація ix адресою на останній
                // елемент вектора x.
```

Рис. 1.3. Доступ до окремих елементів вектора

Всі контейнери в STL забезпечують певні базові методи, включаючи такі:

– size(), який повертає кількість елементів в контейнері;

– swap(), який обмінює зміст двох контейнерів;

– begin(), який повертає ітератор, що посилається на перший елемент контейнера;

– end(), який повертає ітератор, що посилається на елемент за межами контейнера.

1.3. Використання ітераторів та функцій контейнерів

Ітератор – це узагальнене поняття покажчика. Фактично він може бути і просто покажчиком, або ж об'єктом, схожим на покажчик, для якого визначені операції разіменування (operator*()) та інкременту (operator++()). Як буде видно далі, узагальнення покажчиків в ітератори дозволяє STL забезпечити уніфікований інтерфейс для різних класів контейнерів, включаючи і такі, для яких непридатні звичайні покажчики. Кожен клас-контейнер визначає відповідний ітератор. Ім'я типу для

такого ітератора утворене з оператора typedef необхідного класу, оголошеного як iterator. Наприклад, щоб оголосити ітератор для об'єкту vector спеціалізації double, необхідно виконати дії, що показані на рис. 1.4.

```
vector<double>::iterator pd; //  pd – це ітератор
vector<double> scores; // scores є об'єктом vector<double>
// ітератор pd можна використовувати в наступних цілях:
pd = scores.begin(); // указувати за допомогою pd на перший елемент
// scores
*pd = 22.3; // разіменування ітератора pd і присвоєння значення  першому
            // елементу
++pd; //зробити pd покажчиком на наступний елемент
```

Рис. 1.4. Оголошення ітератора для об'єкту класу **vector <double>**

Як видно з прикладу, ітератор поводиться як покажчик. Але, що означає термін „елемент" за межами контейнера? Суть цього терміну полягає в тому, що ітератор посилається на елемент, який є наступним за останнім елементом контейнера. Це подібно до нульового символу, який знаходиться за останнім реальним символом С-строки, з тією лише різницею, що нульовий символ є значенням елементу, а ітератор, що вказує на елемент за контейнером, є покажчиком (ітератором). Функція-елемент end() ідентифікує кінець контейнера. Якщо встановити ітератор на перший елемент контейнера, а потім збільшувати його значення, то рано чи пізно він досягне кінця контейнера, тобто весь вміст контейнера буде проглянутий. Таким чином, якщо об'єкти scores і pd визначені вище, то вивести їх зміст можна за допомогою програмного коду, наведеного на рис. 1.5.

```
for (pd = scores.begin() ; pd != scores.end(); pd++)
    cout « *pd  « endl;
```

Рис. 1.5. Вивід змісту контейнера

У бібліотеці STL визначено п'ять типів ітераторів і визначені алгоритми, для реалізації яких вони потрібні:

– ітератори введення і виводу;
– прямий ітератор;
– двосторонній ітератор;
– ітератор довільного доступу.

Ітератор введення використовується програмою для зчитування значень з контейнера і дозволяє получити доступ до всіх значень в контейнері. Зокрема, запит об'єкту по ітератору введення повинен дозволити програмі прочитати значення з контейнера, але не змінити його. Алгоритм, що базується на ітераторі введення, повинен бути однопрохідним, тобто це не вимагає значень ітератора, визначених при минулому проходженні контейнера, або попередніх його значень при поточному проходженні. Таким чином, ітератор введення є одностороннім: його значення можна збільшити, але не можна повернутися назад. Ітератор виводу схожий на ітератор введення з тією різницею, що разіменування ітератора з гарантією дозволить програмі змінити значення в контейнері, але не дозволить прочитати його. Таким чином, ітератор введення можна застосовувати для однопрохідних алгоритмів, що використовуються тільки для читання, а ітератор виводу – для однопрохідних алгоритмів, що використовуються тільки для запису.

Прямий ітератор, як і ітератори введення (виводу), використовується тільки з операцією ++ для навігації по контейнеру. Тому він може здійснювати тільки переміщення вперед по контейнеру на один елемент при кожному прирості. Проте, на відміну від ітераторів введення і виводу, він завжди проходить послідовність значень в одному і тому ж порядку. До того ж після приросту прямого ітератора, його можна разіменовувати за його попереднім значенням і отримати той же елемент, що і раніше. Ці властивості роблять можливим застосування багатопрохідних алгоритмів. Прямий ітератор може дозволити реалізувати два типи доступу: або читати і змінювати дані, або тільки читати їх. Двосторонній ітератор має всі властивості прямого ітератора, до яких додана можливість підтримки двох операторів декремента (префіксного і постфіксного), тобто він може проходити по контейнеру як вперед, так і назад. Ітератор довільного доступу має всі властивості двостороннього ітератора, а також операції (на зразок складання покажчиків), які підтримують довільний доступ і оператори порівняння, що використовуються для впорядкування елементів.

Бібліотека STL має декілька стандартних ітераторів. Щоб зрозуміти, навіщо потрібні подібні ітератори, необхідно ознайомитися з наступним матеріалом. Зокрема, є алгоритм copy(), призначений для копіювання даних з одного контейнера в інший. Цей алгоритм написаний з урахуванням термінології ітераторів, тому він може копіювати дані з контейнера одного типу в контейнер іншого типу, а також в масив або з нього, оскільки можна використовувати покажчики на масиви як ітератори. Перші два аргументи-ітератори у функції copy() задають діапазон значень, які потрібно скопіюва-

ти, а останній аргумент-ітератор указує, куди потрібно скопіювати перший елемент з діапазону. Перші два аргументи повинні бути ітераторами введення (як мінімум), а останній аргумент – ітератором виводу (як мінімум). Функція copy() перезаписує існуючі дані в контейнері призначення, причому розмір контейнера повинен бути достатнім, щоб вміщати елементи, які копіюються. Тому функцію copy() не можна використовувати для розміщення даних в порожньому векторі. Розглянемо приклад використання алгоритму copy() і двох стандартних ітераторів: ostream_iterator – ітератор виводу в потік; istream_iterator – ітератор читання з потоку.

Наступна функція виконує читання відліків сигналу з файлу, ім'я якого передається через аргумент nf, у вектор, на який посилається zns (приклад наведений на рис. 1.6).

```
void Load(char * nf, vector<unsigned char> & zns)
{ ifstream inp(nf, ios_base::binary); // створення потоку і скріплення
// його з файлом
 copy(istream_iterator<unsigned char,char>(inp),
istream_iterator<unsigned char,char>(), inserter(zns,zns.begin()));
// використання алгоритму copy для копіювання даних з потоку (файлу)
// до вектора zns
 inp.close(); // закриття потоку і відповідно файлу
}
```

Рис. 1.6. Читання відліків сигналу

Всі контейнери включають як загальні методи, однакові для всіх типів контейнерів, так і додаткові методи, що визначені тільки для певного типу контейнера. Наприклад, шаблон класу vector має додаткові методи, доступні лише для декількох контейнерів в STL (рис. 1.7). Один із зручних методів, який називається push_back(), додає елемент в кінець вектора. При додаванні елементу даний метод звертається до управління пам'яттю, щоб збільшити розмір вектора для розміщення елементів, що додаються. Це означає, що можна застосовувати код, наведений на рис. 1.8, у якому при кожній ітерації циклу до вектора scores додається один елемент. При написанні або запуску програми не потрібно запам'ятовувати номер елементу. Доти, поки у розпорядженні програми буде достатній об'єм пам'яті, вона зможе при необхідності розширювати вектор scores.

Метод erase() призначений для видалення заданого діапазону вектора. Він має два аргументи-ітератори, що визначають потрібний діапазон.

```
vector<float> x;
vector<float> y;
x.assign(y) // видалення всіх елементів вектора x і вставка значень
// елементів вектора y;
x.assign(10, 5.5) // видалення всіх елементів вектора x і вставка 10
// значень 5.5;
x.clear(y) // видалення всіх елементів вектора x;
x.erase(ix) // видалення елемента вектора x, на який указує ітератор ix;
x.erase(ix, ix+5) // видалення 5 елементів вектора x починаючи
// з елемента, на який указує ітератор ix;
x.insert(ix, y.begin(), y.end()) // вставка елементів вектора y у вектор x
// починаючи з позиції, на яку вказує ітератор ix (наприклад, вставка
// в кінець вектора x: x.insert(x.end(), y.begin(), y.end()));
x.pop_back() // видалення останнього елемента вектора x;
x.push_back(y[2]) // вставка в кінець вектора x значення третього
// елемента вектора y;
int size = x.size() // одержання кількості елементів, що зберігаються
// у векторі x.
```

Рис. 1.7. Опис деяких методів шаблону **vector**

```
vector<double> scores; //створити порожній вектор
double temp ;
while (cin » temp && temp >= 0)
scores.push_back(temp) ;
cout « "Ви ввели" « scores.size () « "scores.\n";
```

Рис. 1.8. Використання методу **push_back()**

Важливо чітко зрозуміти, як саме STL визначає діапазони, використовуючи два ітератори. Перший ітератор посилається на початок діапазону, а другий – на елемент, наступний за останнім елементом діапазону. Приклад наведений на рис. 1.9, де оператор видаляє перший і другий елементи, тобто ті, на які указують begin() і begin() + 1 (оскільки вектор забезпечує довільний доступ, такі операції, як begin() + 2, визначені для ітераторів класу vector). Якщо itl і it2 — два ітератори, використовується запис [pl, p2), що відображає те, що діапазон починається з pl і тягнеться далі, але не включає p2. Таким чином, діапазон [begin(), end()) містить в собі повний вміст колекції. Крім того, діапазон [pl, p2) вважається порожнім. Запис [) не відноситься до синтаксису C++, тому її немає в програмному коді, вона з'являється тільки в документації.

```
scores.erase(scores.begin(), scores.begin() + 2);
```

Рис. 1.9. Видалення елементів вектора

Метод insert() доповнює метод erase(). Він має три аргументи-ітератори. Перший задає позицію, в яку буде вставлений новий елемент. Другий і третій визначають діапазон, який буде вставлений. Цей діапазон, як правило, є частиною іншого об'єкту. Наприклад, в коді, наведеному на рис. 1.10, всі елементи (окрім першого) вектора new вставляються перед першим елементом вектора old.

```
vector<int> old;
vector<int> new;
...
old. insert(old.begin(), new.begin() + i, new.end());
```

Рис. 1.10. Зразок вставки елементів одного вектора в інший

У зв'язку з цим саме в такій ситуації зручно працювати з ітератором, що посилається на елемент за вектором, оскільки він дозволяє легко додати елементи в кінець вектора (рис. 1.11). Тут нові дані вставлені за елементом old.end(), тобто вони розміщені після останнього елементу вектора.

```
old.insert(old.end(), new.begin()+1, new.end());
```

Рис. 1.11. Вставка даних після останнього елементу вектора

1.4. Загальні функції бібліотеки STL. Функтори та алгоритми

Зазвичай при цифровій обробці сигналів виконується безліч дій з масивами – пошук в масивах, сортування, заповнення випадковим чином тощо. У STL представлений загальний підхід, що визначає для таких операцій функції, що не є елементами яких-небудь класів (шаблонів).

Розглянемо три типові функції STL: for_each(), random_shuffle() і sort().

Функцію for_each() можна використовувати з будь-яким класом-контейнером. Вона має три аргументи. Перші два є ітераторами, що задають діапазон в контейнері, а останній є покажчиком на функцію (у

більш загальному сенсі останній аргумент – це об'єкт-функція). Функція for_each() виконує дію, на яку указує останній аргумент для кожного елементу контейнера в заданому діапазоні. Функція, на яку вказує третій аргумент, не повинна змінювати значення елементів контейнера. Функцію for_each() можна використовувати замість циклу for. Наприклад, код, наведений на рис. 1.12а можна замінити оператором рис. 1.12б. Таке застосування функції дозволить не "захаращувати" код явними змінними-ітераторами.

```
vector<Review>::iterator pr;
for (pr = books.begin() ; pr != books.end(); pr++)
        ShowReview(*pr);
```

а

```
for_each (books .begin (), books.end(), ShowReview);
```

б

Рис. 1.12. Приклад використання функції **for_each**()

Функція random_shuffle() включає два ітератори, що задають діапазон, у якому вона переставляє елементи у випадковому порядку. Наприклад, у виразі random_shuffle(books.begin(), books.end()) випадковим чином переставляються всі елементи у векторі books. На відміну від функції for_each, що працює зі всіма класами-контейнерами, дана функція вимагає, щоб клас-контейнер мав можливість довільного доступу до елементів, що якраз і дозволяє робити клас vector.

Функція sort() також вимагає того, щоб контейнер забезпечував довільний доступ. Вона має дві версії. Перша версія використовує два ітератори, що задають діапазон, і сортує його, застосовуючи оператор „<", визначений для елементів того типу, який збережений в контейнері. Наприклад, фрагмент програми, наведений на рис. 1.13, сортує зміст об'єкту coolstuff в зростаючому порядку, використовуючи оператор „<" для порівняння значень.

```
vector<int> coolstuff;
sort (coolstuff.begin(), coolstuff.end());
```

Рис. 1.13. Приклад використання функції **sort** () (перша версія)

Якщо елементи контейнера є об'єктами, заданими користувачем, тоді потрібна функція operator<(), визначена для роботи з такими об'єктами (в цілях використання функції sort()).

Нехай необхідне сортування в порядку убування або за порядком рейтингів, а не назв. Тоді можна скористатися другою формою функції sort(). Вона має три аргументи. Перші два аргументи є ітераторами, які задають діапазон. Останній, третій аргумент – це покажчик на функцію (у більш загальному сенсі – на функціональний об'єкт), яку необхідно використовувати замість operator<() для порівняння. Значення, що повертається, повинне бути конвертованим в тип bool, причому значення false значить, що два аргументи розташовані в невірному порядку. Приклад подібної функції наведений на рис. 1.14.

```
bool WorseThan(const Review & rl, const Review & r2)
{if (rl.rating < r2.rating)
        return true;
else return false;
}
.....
sort (books.begin, books.end(), WorseThan);
```

Рис. 1.14. Приклад використання функції **sort** () (друга версія)

Маючи функцію WorseThan(), можна сортувати вектор books, що складається з об'єктів Review, в порядку зростання рейтингів. Якщо два об'єкти мають однакові елементи-назви, функція operator<() сортує об'єкти, використовуючи елемент, що задає рейтинг.

Багато алгоритмів використовують функціональні об'єкти (функтори). Функтор – це будь-який об'єкт, який можна використовувати з оператором (). Існують такі види функторов: генератор (аргументи відсутні); унарна функція (функтор з одним аргументом); бінарна функція (функтор з двома аргументами). Приклад визначення унарного функтора наведений на рис. 1.15.

Крім функторів у бібліотеці STL визначено ряд дуже корисних шаблонів функцій – алгоритмів. Вказані алгоритми використовуються для обробки елементів контейнерів й реалізують такі процедури як пошук максимального значення серед елементів контейнеру, пошук мінімального значення серед елементів контейнеру, акумуляція елементів контейнеру або результатів обробки кожного елементу контейнера, трансформацію елементів контейнера й багато інших процедур, що пов'язані з обробкою множин даних.

Приклади використання алгоритмів наведено на рис. 1.16 – 1.18.

```
template <class Arg>  //шаблон класу унарної функції
class gist : private unary_function<Arg,void>
{ private:
  vector<Arg> & g; // посилання на вектор для зберігання результатів
  public:
  gist(vector<Arg>& gis, int max): g(gis) // конструктор унарної функції
    { g.clear();
      g.insert(g.begin(),(max+1),0);
    }
  void operator()(const Arg& x) // визначення оператора ()
      { g[x]+=1;
          } };
```

Рис. 1.15. Приклад визначення унарного функтора gist

```
  vector<float>::iterator max = max_element(x.begin(), x.end()); // повертає ітератор, що вказує на максимальний елемент вектора;
  float maxx = *max; // одержання максимального значення;
  vector<float>::iterator min = min_element(x.begin(), x.end()); // повертає ітератор, що вказує на мінімальний елемент вектора;
  float minx = *min; // одержання мінімального значення;
  float sum = accumulate (x.begin(), x.end(), 0); // алгоритм accumulate у даному випадку повертає суму елементів вектора x.
```

Рис. 1.16. Використання алгоритмів **accumulate, max_element, min_element**

```
float step2f (float & y, float x) // бінарна функція, що виконує
{y = y + x * x; // піднесення другого параметра x у квадрат і підсумовування
return y; }    // результату з першим параметром y  (акумулятором);
float sum=0; // акумулятор;
sum = accumulate (x.begin(), x.end(), sum, step2f); // спільне використання
// бінарної функції step2f і алгоритму accumulate дозволить одержати
// в змінній (акумуляторі) sum суму квадратів елементів вектора x;
```

Рис. 1.17. Приклад використання алгоритму *accumulate*

```
transform(x.begin(), x.end(), y.begin(), z.begin(), multiplies<float>());
// даний алгоритм формує вектор z, кожен елемент якого є добутком
// елементів векторів x і y, наприклад, z[5]=x[5]*y[5] (розміри векторів
// x і y повинні бути однаковими);
transform(x.begin(), x.end(), y.begin(), z.begin(), minus<float>()); // даний
// алгоритм формує вектор z, кожен елемент якого являє собою
// різницю елементів векторів x і y, наприклад, z[5]=x[5]–y[5].
```

Рис. 1.18. Приклад використання алгоритму *transform*

2. ФІЗИЧНІ ХАРАКТЕРИСТИКИ ЦИФРОВИХ СИГНАЛІВ

2.1. Визначення фізичних характеристик цифрових сигналів

Сигналом називають цілеспрямовані зміни фізичної величини, явища або процесу, викликані повідомленням, яке створюється джерелом інформації. За своєю фізичною природою сигнали – переносники інформації, які, як правило, є незалежними від джерела повідомлень. Тому з математичної точки зору можна вважати, що в процесі формування сигналів кожному повідомленню A_i ставиться у взаємно однозначну відповідність сигнал S_i, тобто множини A і S стають ізоморфними.

Всі фізичні процеси, що протікають в елементах і пристроях телекомунікаційних систем, залежно від особливостей їх математичного опису і аналізу можна поділити на два класи:

– детерміновані процеси;

– випадкові процеси.

Перші описуються і аналізуються за допомогою апарату класичної математики, а другі – за допомогою апарату теорії випадкових функцій.

Детермінованими слід вважати такі фізичні процеси, перебіг яких можна повністю передбачити, маючи в своєму розпорядженні дані про параметри процесу. Математично детермінований процес описується детермінованою функцією часу S(t), тобто такою функцією, що для будь-якого наперед заданого моменту часу t_i може бути однозначно визначене її значення $S(t_i)$. З інформаційної точки зору детермінованою функцією часу може бути описаний процес, відповідний відомому повідомленню, тобто можна сказати, що детермінованими функціями описуються неінформативні процеси.

Детерміновані процеси часто використовують при вивченні властивостей лінійних або нелінійних канальних пристроїв. Подаючи на вхід процес з відомими властивостями і аналізуючи характер зміни цих властивостей у вихідному процесі даного пристрою, можна отримати інформацію про характер перетворення, яке здійснюється канальним пристроєм, властивості якого вивчаються.

Таким чином, можна сказати, що в системах зв'язку детермінованими функціями часу описуються процеси, які виконують роль несучих коливань (процесів-переносників).

Випадковими або нерегулярними функціями математично описуються процеси, перебіг яких точно передбачити неможливо. З погляду одержувача інформації фізичні процеси, які переносять невідомі повідомлення, є випадковими процесами і тому можуть бути математично описані тільки випадковими функціями.

Взагалі, процеси, які відповідають реальним фізичним явищам, не можуть бути описані точними математичними співвідношеннями, якщо результат поведінки явища в майбутньому є невідомим. Результат будь-якого конкретного спостереження випадкового явища є лише одним з можливих результатів. І хоча він може бути точно описаним у вигляді аналітичного співвідношення або графіка, при спробі прогнозування процесу в майбутньому залишається невідомим, який з можливих результатів спостереження над процесом матиме місце. Функція часу, що описує конкретну поведінку випадкового явища, називається вибірковою функцією або реалізацією випадкового процесу. Всю множину можливих вибіркових функцій прийнято називати випадковою функцією $Y(t) = \{y_i(t)\}$.

Детерміновані процеси і вибіркові функції випадкових процесів можуть бути повністю описані будь-яким з відомих в класичній математиці способом: аналітично, графіком або таблицею. Для опису випадкових функцій жоден з цих методів є непридатним. Детальний опис випадкових процесів проводять за допомогою багатовимірних функцій розподілу щільності ймовірності їх значень.

Сигнал може бути описаний в умовних координатах, які будемо називати «стан S» та «час t». У теорії найбільш загальну класифікацію сигналів проводять у зв'язку з особливостями поведінки функцій S(t) в цих координатах, оскільки особливості математичного опису сигнальних функцій S(t) визначають і характер математичних моделей для дослідження інформаційних систем.

З цієї точки зору слід розрізняти чотири класи сигналів:

– *безперервні сигнали*, які описуються безперервними функціями безперервного аргументу S(t) (рис. 2.1, а);

– *дискретно-безперервні сигнали* ($S_j(t)$, $j = \overline{1, m}$), які описуються дискретними функціями безперервного аргументу (величина S у таких сигналів є дискретною, а аргумент t – безперервний (рис. 2.1, б); величина m визначає кількість дискретних станів або рівнів квантування величини S);

– *безперервно-дискретні сигнали*, які описуються безперервними функціями дискретного аргументу $k = \overline{1, n}$ (дискретні значення аргуме-

НАУКОВО-ТЕХНІЧНА БІБЛІОТЕКА Національного аерокосмічного університету ім. М.Є. Жуковського

нту t_k називають точками відліку, значення ж функції у відлікових точках можуть відрізнятися один від одного на нескінченно малу величину (рис. 2.1, в); величина n визначає кількість відліків аргументу t_k);

– *дискретні сигнали* ($S_j(t_k)$, $j=\overline{1,m}$), які описуються дискретними функціями дискретного аргументу $k=\overline{1,n}$ (рис. 2.1, г).

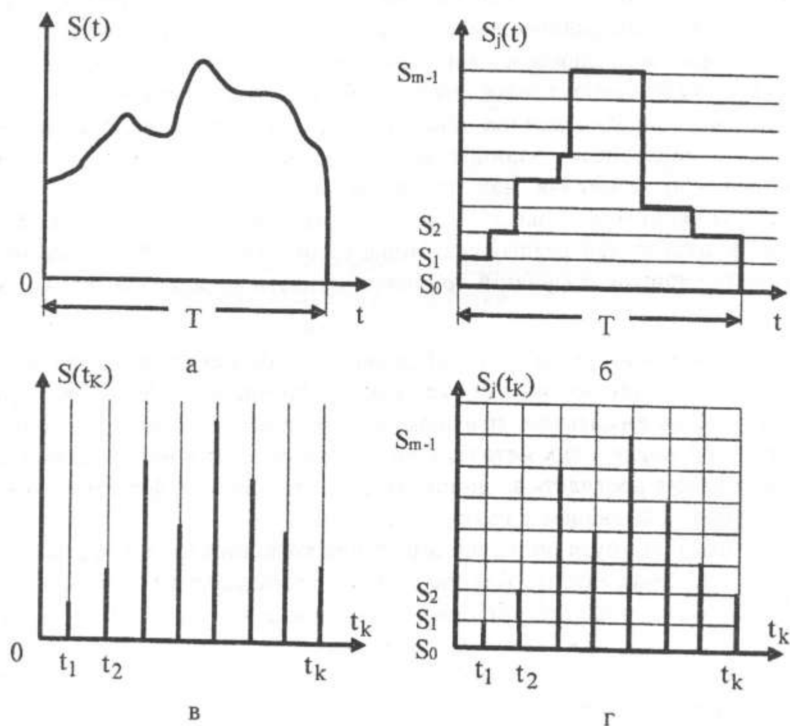


Рис. 2.1. Класи сигналів

Саме методи обробки дискретних сигналів є предметом вивчення теорії цифрової обробки сигналів.

Для виконання більшості процедур цифрової обробки сигналів (ЦОС) необхідно знати ряд фізичних характеристик сигналів, що обробляються. Найчастіше використовуються такі характеристики сигналів:

– ***динамічний діапазон*** сигналу (D_s), який визначається в одиницях фізичного виміру значень відліків сигналу, наприклад, вольтах або амперах:

$$D_s = s_{max} - s_{min}, \quad (2.1)$$

де s_{max} і s_{min} – відповідно максимальне і мінімальне значення відліків сигналу, що оброблюється; також для вимірювання динамічного діапазону використовують безрозмірні відносні величини

$$\hat{D}_c = \frac{s_{max} - s_{min}}{s_{min}}, \quad (2.2)$$

або логарифмічні одиниці

$$\tilde{D}_c = \lg \frac{s_{max} - s_{min}}{s_{min}}; \quad (2.3)$$

– ***енергія*** (E_s) і ***середня потужність*** (P_s) сигналу, які необхідні для побудови раціональної інформаційної системи з мінімальними енергетичними втратами:

$$E_s = \sum_{i=0}^{N-1} s_i^2; \quad (2.4)$$

$$P_s = \frac{1}{N} \sum_{i=0}^{N-1} s_i^2, \quad (2.5)$$

де N – кількість відліків дискретного сигналу; s_i – i-й відлік дискретного сигналу; для сумісного розгляду двох сигналів, що діють в технічній системі, вводять взаємну енергію та взаємну середню потужність сигналів:

$$E_{1,2} = \sum_{j=0}^{N-1} s_{1_j} s_{2_j}, \quad (2.6)$$

$$P_{1,2} = \frac{E_{1,2}}{(t_k - t_0)}; \quad (2.7)$$

– ***відстань між сигналами*** або ***середньоквадратичне відхилення*** ($d_{i,j}$), яке є кількісною оцінкою відмінності двох сигналів

$$d_{i,j} = \sqrt{\sum_{k=0}^{N-1} \left[s_{i_k} - s_{j_k} \right]^2} \quad (2.8)$$

(чим більше величина відстані, тим більша й різниця між сигналами);

– ***середнє значення відліків*** детермінованого сигналу або ***математичне сподівання*** ергодичного сигналу (m_s):

$$m_s = \frac{1}{N}\sum_{i=0}^{N-1} s_i\,; \tag{2.9}$$

– ***дисперсія значень відліків*** ергодичного сигналу (d_s), яка характеризує величину розсіяння значень процесу щодо його математичного сподівання; у зв'язку з тим, що дисперсія є квадратичною характеристикою, фізично її часто трактують як середню енергетичну характеристику ергодичного сигналу

$$d_s = \frac{1}{N}\sum_{i=0}^{N-1} (s_i - m_s)^2; \tag{2.10}$$

– ***функція автокореляції*** дискретного стаціонарного сигналу ($R_s(\tau)$), яка дозволяє оцінити взаємозв'язок відліків сигналу в його часових перетинах, що заходяться на відстані τ:

$$R_s(\tau) = \begin{cases} \dfrac{1}{N-\tau}\displaystyle\sum_{i=0}^{N-\tau-1} [s_{i+\tau} - m_s]\cdot[s_i - m_s], & \tau \ge 0; \\ R_s(-\tau), & \tau < 0; \end{cases} \tag{2.11}$$

– ***інтервал кореляції*** ($\tau_{кор}$), який характеризує граничний часовий зсув, при якому ще існують зв'язки між відліками стаціонарного сигналу (рис. 2.2) :

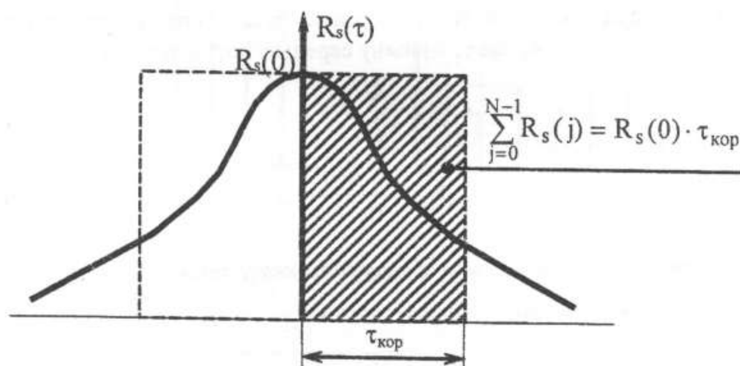


Рис. 2.2. Визначення інтервалу кореляції

$$\tau_{кор} = \sum_{j=0}^{N-1} R_s(j) \Big/ R_s(0). \tag{2.12}$$

2.2. Програмна реалізація функцій визначення фізичних характеристик цифрових сигналів

Далі наведені приклади програмної реалізації функцій, які виконують обчислення наступних фізичних характеристик цифрових сигналів: максимального значення відліків сигналу (рис. 2.3); мінімального значення відліків сигналу (рис. 2.4); енергії сигналу (рис. 2.5); середнього значення відліків сигналу (рис. 2.6); дисперсії значень відліків сигналу (рис. 2.7); формування відліків функції автокореляції та інтервалу кореляції (рис. 2.8). В наведених прикладах ss – посилання на об'єкт класу вектор, що містить відліки вихідного сигналу; mx – значення математичного сподівання; res – посилання на об'єкт класу вектор, до якого записані відліки функції автокореляції.

```
int MaxS(vector<unsigned char> & ss)
{ vector<unsigned char>::iterator max = max_element(ss.begin(), ss.end());
 return *max;
}
```

Рис. 2.3. Функція отримання максимального значення відліків сигналу

```
int MinS(vector<unsigned char> & ss)
{ vector<unsigned char>::iterator min = min_element(ss.begin(), ss.end());
 return *min;
}
```

Рис. 2.4. Функція отримання мінімального значення відліків сигналу

```
// Функція акумулятивного піднесення до степеня 2
unsigned int step2(unsigned int & y, unsigned int x)
{y = y + x * x;
 return y; }
//функція отримання енергії сигналу
unsigned int Es(vector<unsigned char> & ss)
{unsigned int sum=0;
 sum = accumulate (ss.begin(), ss.end(), sum, step2);
 return sum; }
```

Рис. 2.5. Функція обчислення енергії сигналу

```
float ms(vector<unsigned char> & ss)
{unsigned int sum=0;
 sum = accumulate (ss.begin(), ss.end(), 0);
 float mt = float(sum)/ss.size();
 return mt;}
```

Рис. 2.6. Функція отримання середнього значення відліків сигналу

```
float ds(vector<unsigned char> & ss, float mx)
{float sum=0;
 vector <float> tmp;
 vector <float> result(ss.size());
 tmp.assign(ss.size(), mx);
 transform(ss.begin(), ss.end(),tmp.begin(),result.begin(),
minus<float>());
 sum = accumulate (result.begin(), result.end(), sum, step2f);
 float dx = sum/ss.size();  return dx; }
```

Рис. 2.7. Функція отримання дисперсії відліків сигналу

```
int ak(vector<unsigned char> & ss, vector <float> & res)
{vector <int> tmp(ss.size());
 vector <float> fktmp;
 int sum=0;
 vector<unsigned char>  a(ss);
 vector<unsigned char>  b(ss);
 for(int t=0;t<ss.size();t++)
    {transform(a.begin(),a.end(),b.begin(),tmp.begin(), multiplies<int>());
     sum = accumulate (tmp.begin(), tmp.end(), 0);
     fktmp.push_back((float(sum)/float(ss.size())));
     a.erase(a.begin());
     b.pop_back();  sum = 0;
     tmp.assign(a.size(),0);   }
res.insert(res.end(),fktmp.rbegin(),fktmp.rend());
res.insert(res.end(),fktmp.begin()+1,fktmp.end());
float sm = accumulate (fktmp.begin(), fktmp.end(), 0);
return ceil(sm/fktmp[0]);  }
```

Рис. 2.8. Приклад обчислення відліків функції автокореляції та інтервалу кореляції

3. ДИСКРЕТНІ ОРТОГОНАЛЬНІ ПЕРЕТВОРЕННЯ ЦИФРОВИХ СИГНАЛІВ

3.1. Загальні відомості про ортогональні перетворення

Ортогональні перетворення (ОП) знаходять широке застосування в різних областях обробки сигналів, серед яких можна виділити такі, як аналіз, фільтрація, стиск, розпізнавання тощо.

Ортогональні перетворення використовуються для обробки зображень, аудіо- і мовних сигналів, сейсмічної інформації та інших видів сигналів. Як приклади можна привести метод стиску зображень JPEG і метод стиску аудіоінформаціїї MP3, у яких використовується дискретно-косинусне перетворення (ДКП). У загальному виді структурна схема процедури обробки сигналу з використанням ортогонального перетворення представлена на рис. 3.1.

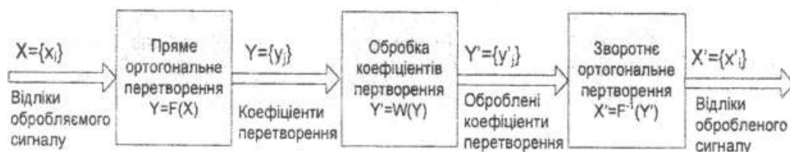


Рис. 3.1. Структурна схема обробки сигналу в спектральній області

Ортогональні перетворення призначені для переведення вихідного сигналу з просторово-часової області в спектрально-частотну область. При цьому вихідний сигнал (функція часу) може бути представлений через ортонормовану множину спектральних функцій у вигляді ряду. Представлення сигналів у вигляді множини спектральних функцій дозволяє проводити їх спектральний аналіз, виконувати згортки складних сигналів, проводити обробку в спектральній області з декорельованими спектральними елементами сигналу тощо.

Ортогональні перетворення (ОП) складаються з прямого і зворотного перетворення. ОП можуть бути реалізовані: безпосередньо по формулах прямого та зворотного перетворення, за допомогою матричних операцій та швидких алгоритмів. У загальному вигляді дискретне ортогональне перетворення має вигляд:

$$Y(k) = <1/N> \sum_{m=0}^{N-1} X(m)W(k,m),\ k = \overline{0, N-1}; \qquad (3.1)$$

$$X(m) = \sum_{k=0}^{N-1} Y(k)W(k,m),\ m = \overline{0, N-1}, \qquad (3.2)$$

де X(m) – m-й відлік вихідного дискретного сигналу; Y(k) – k-й коефіцієнт перетворення; W(k, m) – m-й відлік k-ї базисної функції ортогонального перетворення; N – кількість відліків у вихідному сигналі або в блоці вихідного сигналу при поблочній обробці; $< 1/N >$ – коефіцієнт нормування, який у деяких ортогональних перетвореннях може бути відсутнім.

Вирази (3.1) і (3.2) визначають відповідно пряме і зворотне ортогональне перетворення.

Базисні функції W (k, m) ортогонального перетворення повинні відповідати вимогам попарної ортогональності й ортонормування. Для дискретних сигналів вказана вимога має вигляд

$$\sum_{m=0}^{N-1} W(k,m)W(l,m) = \begin{cases} 1, \text{при } k = l; \\ 0, \text{при } k \neq l. \end{cases}$$

При цьому вся множина попарно ортогональних і ортонормованих функцій утворить ортогональний базис.

На даний момент існує велика кількість різних ортогональних перетворень, що відрізняються системами базисних функцій. У цифровій обробці сигналів найбільше поширення одержали такі ортогональні перетворення:

- перетворення Фур'є;
- перетворення Уолша;
- дискретно-косинусне перетворення;
- перетворення Хаара.

Розглянемо вищеперераховані ортогональні перетворення.

Якщо {X(m)} – це послідовність X(m), $m = \overline{0, N-1}$ – кількість відліків дискретного сигналу, то відповідно пряме та зворотне дискретне перетворення Фур'є (ДПФ) цієї послідовності визначається як

$$Y(k) = \frac{1}{N} \sum_{m=0}^{N-1} X(m)W^{km},\quad k = \overline{0, N-1}; \qquad (3.3)$$

$$X(m) = \sum_{k=0}^{N-1} Y(k)W^{-km},\quad \overline{m = 0, N-1}, \qquad (3.4)$$

де $W = e^{-i2\pi/N}$ – ортогональні експоненціальні функції; $i = \sqrt{-1}$.

Дискретно-косинусне перетворення визначається виразами:

– пряме перетворення:

$$Y(0)=\frac{1}{\sqrt{N}}\sum_{m=0}^{N-1}X(m)\,;\;\; Y(k)=\sqrt{\frac{2}{N}}\sum_{m=0}^{N-1}X(m)\cos\frac{(2m+1)k\pi}{2N},\quad k=\overline{1,N-1}\,;$$

– зворотне перетворення:

$$X(m)=\frac{1}{\sqrt{N}}Y(0)+\sqrt{\frac{2}{N}}\sum_{k=0}^{N-1}Y(k)\cos\frac{(2m+1)k\pi}{2N},\quad m=\overline{1,N-1}\,.$$

Перетворення Уолша (за Адамаром) можна записати у наступному вигляді:

– пряме перетворення:

$$Y(k)=\frac{1}{N}\sum_{m=0}^{N-1}X(m)(-1)^{<m,k>},\quad k=\overline{0,N-1}\,;$$

– зворотне перетворення:

$$X(m)=\sum_{k=0}^{N-1}Y(k)(-1)^{<m,k>},\quad m=\overline{0,N-1}\,,$$

де $<m,k>=\sum_{s=0}^{n-1}k_s m_s$; $n=\log_2 N$; k_s та m_s є коефіцієнтами двійкового представлення k і m відповідно, тобто для кожного десяткового числа

$$k=k_{n-1}2^{n-1}+k_{n+2}2^{n-2}+\ldots+k_1 2^1+u_0 2^0\,,$$

де $k_\upsilon=0$ або 1 ($\upsilon=\overline{0,n-1}$).

Таким же чином кожне десяткове число m, $0\le m\le N-1$ можна представити у вигляді

$$m=m_{n-1}2^{n-1}+m_{n-2}2^{n-2}+\ldots+m_1 2^1+m_0 2^0\,,$$

де $m_x=0$ або 1 ($x=\overline{0,n-1}$).

Пряме та зворотне перетворення Хаара відповідно записуються в наступному вигляді:

$$Y(k)=\frac{1}{N}\sum_{m=0}^{N-1}X(m)H(k,m),\quad k=\overline{0,N-1}\,;$$

$$X(m)=\sum_{k=0}^{N-1}Y(k)H(k,m),\quad m=\overline{0,N-1}\,,$$

де дискретні відліки базисних функцій визначаються виразами:

$$t=m/(N-1);$$

$$H(k,t)=\begin{cases}1,\ \text{при } t\in[0,1),\ k=0;\\ 2^{t/2},\ \text{при } \frac{v-1}{2^r}\le t<\frac{v-\frac{1}{2}}{2^r},\ r=\lfloor\log_2 k\rfloor,\ v=k-(2^r-1);\\ -2^{t/2},\ \text{при } \frac{v-\frac{1}{2}}{2^r}\le t<\frac{v}{2^r},\ r=\lfloor\log_2 k\rfloor,\ v=k-(2^r-1);\\ 0,\ \text{для інших } t\in[0,1),\end{cases}$$

де $\lfloor x\rfloor$ – операція округлення до меншого цілого.

3.2. Матрична форма представлення дискретних ортогональних перетворень

Відліки базисних функцій зручно представляти у вигляді матриці

$$W(n)=\left[w_{i,j}\right]\quad (n=\log_2 N),$$

де i-й рядок є вектором відліків i-ої базисної функції. У цьому випадку вираз для прямого (3.3) і зворотного (3.4) перетворень зручно записувати в матричному вигляді таким чином:

$$Y(n)=<\frac{1}{N}>\cdot W(n)\otimes X(n)\ ; \tag{3.5}$$

$$X(n)=W^T(n)\otimes Y(n), \tag{3.6}$$

де W(n) – матриця відліків базисних функцій ортогонального перетворення розмірності N × N; X(n) – вектор з N відліків вихідного сигналу або блоку сигналу при поблочній обробці; Y(n) – вектор з N коефіцієнтів перетворення; ⊗ – операція множення матриць; $<\frac{1}{N}>$ – коефіцієнт нормування, який в деяких ОП може бути відсутнім (наприклад, у ДКП).

Матриці дискретних відліків перших восьми базисних функцій даних перетворень мають такий вигляд:

– перетворення Уолша

$$W_{\text{Уолш}}(3)=\begin{bmatrix}1&1&1&1&1&1&1&1\\1&1&1&1&-1&-1&-1&-1\\1&1&-1&-1&-1&-1&1&1\\1&1&-1&-1&1&1&-1&-1\\1&-1&-1&1&1&-1&-1&1\\1&-1&-1&1&-1&1&1&-1\\1&-1&1&-1&-1&1&-1&1\\1&-1&1&-1&1&-1&1&-1\end{bmatrix};$$

– дискретно-косинусне перетворення

$$W_{ДКП}(3) =$$

$$= \begin{bmatrix} 0,353553 & 0,353553 & 0,353553 & 0,353553 & 0,353553 & 0,353553 & 0,353553 & 0,353553 \\ 0,490393 & 0,415818 & 0,277992 & 0,097887 & -0,097106 & -0,277329 & -0,415375 & -0,490246 \\ 0,461978 & 0,191618 & -0,190882 & -0,461673 & -0,462282 & -0,192353 & 0,190145 & 0,461366 \\ 0,414818 & -0,097106 & -0,490246 & -0,278653 & 0,276667 & 0,490710 & 0,099448 & -0,414486 \\ 0,353694 & -0,,53131 & -0,354256 & 0,352567 & 0,354819 & -0,352001 & -0,355378 & 0,351435 \\ 0,277992 & -0,490246 & 0,096324 & 0,416700 & -0,414486 & -0,100228 & 0,491013 & -0,274673 \\ 0,191618 & -0,462282 & 0,461366 & -0,189409 & -0,193822 & 0,463187 & -0,460440 & 0,187195 \\ 0,097887 & -0,278653 & 0,416700 & -0,490862 & 0,489771 & -0,413593 & 0,274008 & -0,092414 \end{bmatrix};$$

– перетворення Хаара

$$W_{Хаар}(3) = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ \sqrt{2} & \sqrt{2} & -\sqrt{2} & -\sqrt{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \sqrt{2} & \sqrt{2} & -\sqrt{2} & -\sqrt{2} \\ 2 & -2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & -2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & -2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & -2 \end{bmatrix}.$$

На практиці для спрощення процедури перетворення виконують розбиття сигналу, що обробляється на блоки розмірності N ($N = 2^n$, де $n = 1, 2, 3, \ldots$), кожний з яких обробляється окремо. Якщо кількість відліків сигналу не кратне N, то такий сигнал подовжується, наприклад, додаванням нульових відліків для забезпечення кратності N.

3.3. Швидкі алгоритми дискретних ортогональних перетворень

Реалізація ОП на основі розглянутих вище виразів характеризується високою надмірністю арифметичних операцій, що призводить до значних часових витрат при виконанні ортогональних перетворень сигналів. До надмірних арифметичних операцій можна віднести:

- операції, що дублюються при обчисленні різноманітних коефіцієнтів перетворення;

- операції множення на 0 та 1;
- операції сумування значень, що дорівнюють 0.

У процентному відношенні надмірні арифметичні операції можуть складати 80% всіх арифметичних операцій, необхідних для виконання ОП. Виключити виконання надмірних арифметичних операцій при ОП дозволяє застосування швидких алгоритмів ОП. Для різноманітних ОП створено багато швидких алгоритмів, найпростішими з яких є алгоритми типу Кулі-Тьюки. Найбільш наглядно структура швидких алгоритмів пояснюється за допомогою графів типу «метелик», приклад якого для алгоритму швидкого перетворення Фур'є наведено на рис. 3.2.

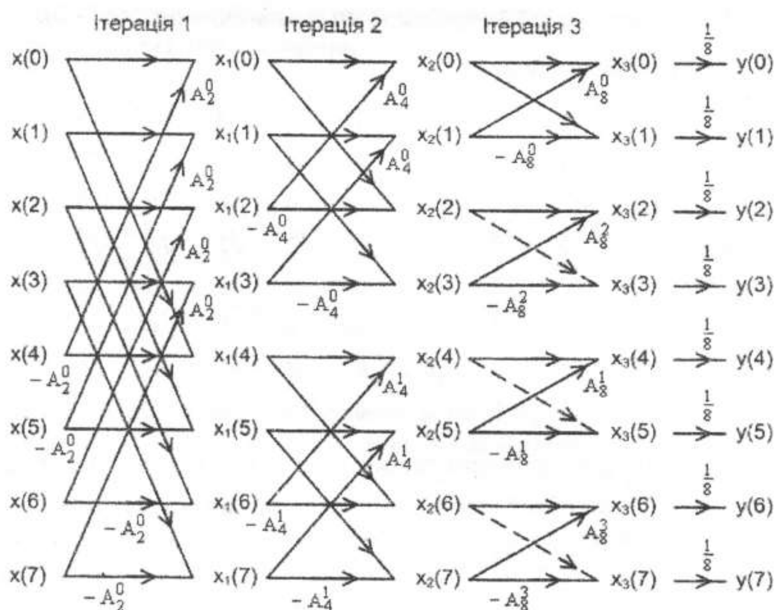


Рис. 3.2. Граф прямого швидкого перетворення Фур'є при N = 8

На графі (рис. 3.2) точки з'єднання кінців стрілок вказують на підсумовування операндів, що стоять в точках початку стрілок. Якщо біля стрілки вказано число, то перед підсумовуванням операнд, що стоїть на початку даної стрілки, помножується на це число, при цьому

$$A_s^v = (A_{2^r})^v = (W^{N/2^r})^v,$$

де $W = e^{-i2\pi/N}$, наприклад, $x_1(0) = x(0) + A_2^0 \cdot x(4)$.

Зворотне швидке перетворення Фур'є теж виконується у відповідності з графом на рис.3.2, якщо його розглядати у протилежному напрямку – з правого кінця (всі стрілки графа змінюють напрямок на протилежний).

Число арифметичних операцій (тобто комплексних множень з подальшим складанням або відніманням) у випадку використання формул (3.3) та (3.4) складає $2N^2$. Такий метод обчислення коефіцієнтів ДПФ називатимемо «прямим методом». Використання швидкого перетворення Фур'є приводить до $\log_2 N$ ітерацій, а загальне число необхідних арифметичних операцій стане рівним приблизно $N\log_2 N$.

На рис. 3.3 наведено граф прямого швидкого перетворення Уолша (впорядкованість по Адамару) для випадку, коли N = 8.

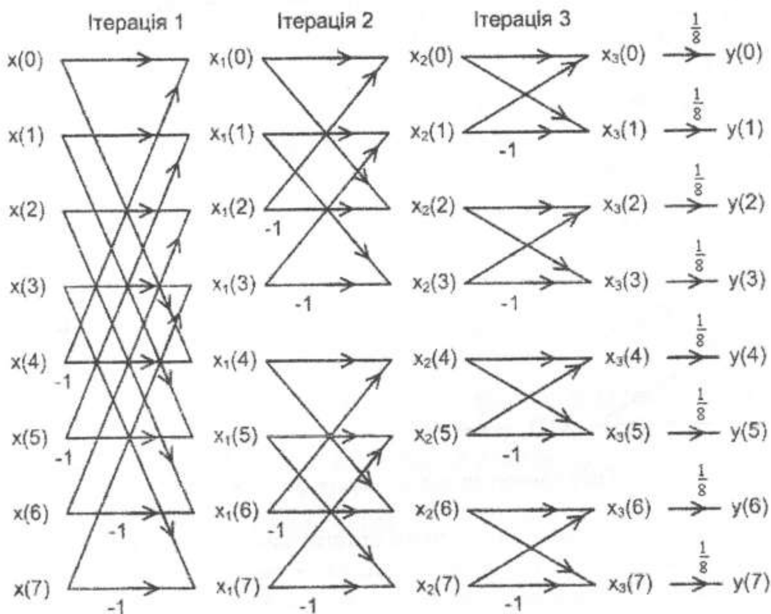


Рис. 3.3. Граф прямого швидкого перетворення Уолша при N = 8

З графа швидкого перетворення Уолша (рис. 3.3) витікає, що для його здійснення, за винятком нормування за допомогою множника 1/8, потрібні тільки операції складання і віднімання. Число складань і віднімань, необхідне для обчислення коефіцієнтів перетворення Уолша, дорівнює $N\log_2 N$. Зворотне швидке перетворення Уолша виконується у відповідності з графом на рис.3.3, якщо його розглядати у протилежному напрямку – з правого кінця (всі стрілки графа змінюють напрямок на протилежний).

На рис. 3.4 наведено граф прямого швидкого перетворення Хаара, запропонований Ендрюсом, для випадку, коли N = 8. Для виконання перетворення Хаара у відповідності з наведеним графом необхідно $2 \cdot (N - 1)$ операцій складання/віднімання і N операцій множення. Зворотне швидке перетворення Хаара виконується у відповідності з графом на рис. 3.4, якщо його розглядати у протилежному напрямку – з правого кінця (всі стрілки графа змінюють напрямок на протилежний).

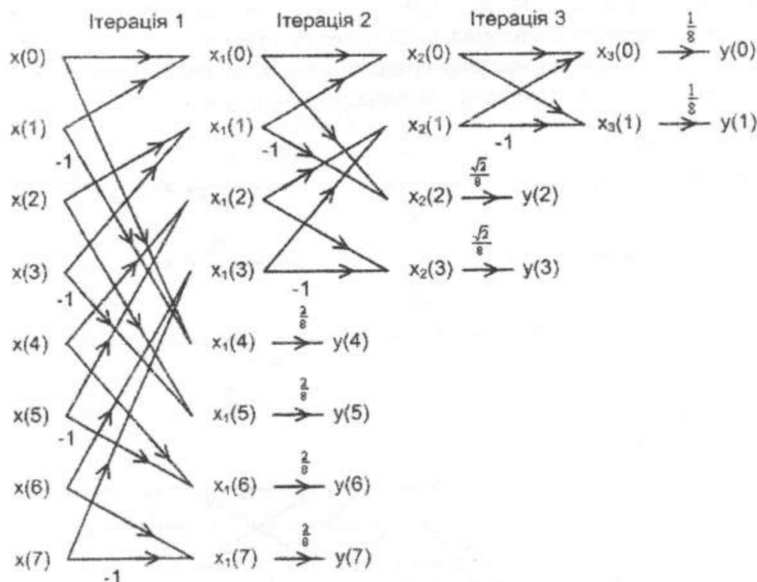


Рис. 3.4. Граф прямого швидкого перетворення Хаара при N = 8

У процесі виконання дискретного ортогонального перетворення виникають погрішності, які пов'язані з помилками округлення при виконанні дійсних операцій. Зазначені помилки приводять до того, що сигнал, отриманий у результаті зворотного ОП, не точно відповідає вихідному сигналу. Величину погрішності, внесеної ОП, можна оцінити за допомогою показника середньоквадратичного відхилення, яке визначається виразом

$$\sigma = \sqrt{\sum_{i=0}^{N-1} (\hat{x}_i - x_i)^2 \Big/ N}\,, \tag{3.7}$$

де N – кількість відліків у сигналі; x_i – i-й відлік вихідного сигналу; $\hat{x}_i$ – i-й відлік відновленого сигналу.

3.4. Програмна реалізація ортогональних перетворень дискретних сигналів

У даному підрозділі наведено приклади програмних процедур, які реалізують пряме та зворотне ортогональне перетворення відліків дискретного сигналу з поблоковою обробкою (N = 8) у відповідності до виразів (3.5) та (3.6). Крім того, приведені приклади програмної реалізації процедури обчислення помилки перетворення (середньоквадратичного відхилення) та часу виконання перетворення. Приклади реалізовані на мові C++ з використання бібліотеки STL.

3.4.1. Функція, яка реалізує ОП відліків дискретних сигналів

1. Вихідні дані функції наведені на рис. 3.5.

```
//матриця відліків ДКП
float mDkp[8][8] = {{0.353553, 0.353553, 0.353553, 0.353553, 0.353553,
0.353553, 0.353553, 0.353553},
{0.490393, 0.415818, 0.277992, 0.097887, -0.097106, -0.277329, -0.415375,
-0.490246},
{0.461978, 0.191618, -0.190882, -0.461673, -0.462282, -0.192353, 0.190145,
0.461366},
{0.414818, -0.097106, -0.490246, -0.278653, 0.276667, 0.490710, 0.099448,
-0.414486},
{0.353694, -0.353131, -0.354256, 0.352567, 0.354819, -0.352001, -0.355378,
0.351435},
{0.277992, -0.490246, 0.096324, 0.416700, -0.414486, -0.100228, 0.491013,
-0.274673},
{0.191618, -0.462282, 0.461366, -0.189409, -0.193822, 0.463187, -0.460440,
0.187195},
{0.097887, -0.278653, 0.416700, -0.490862, 0.489771, -0.413593, 0.274008,
-0.092414}};
//вектор, в якому зберігаються відліки сигналу
vector<unsigned char> signal;
//вектор, в якому зберігаються значення коефіцієнтів перетворення
vector<float> trans;
//вектор, в якому зберігаються відліки відновленого сигналу
vector<int> vost;
```

Рис. 3.5. Вихідні дані функції, яка реалізує ОП

2. Опис аргументів функції:

basa – вказівник на двовимірний масив, що містить відліки базисних функцій:

inp – вектор, що містить відліки вихідного сигналові:

out – вектор, до якого записуються розраховані коефіцієнти перетворення.

3. Функція ортогонального перетворення наведена на рис. 3.6.

```
void fOP(float * basa, vector<unsigned char> & inp, vector<float> & out)
{ vector<float>::iterator bs = basa; // ітератор, який вказує на елементи
                                      // масиву відліків базисних функцій
 vector<double> tmp(8,0); // вектор, до якого записуються
                          // розраховані коефіцієнти перетворення
                          //  поточного блоку відліків довжини 8
 vector<double> block8(8,0); // вектор, до якого записуються відліки
                             // поточного блоку довжини 8
 vector<double> ishod; // вектор, до якого записуються відліки
                       // вихідного сигналу з перетворенням
                       // типу unsined char до double
 ishod.assign(inp.begin(),inp.end()); // копіювання з перетворенням типу
                                      // відліків вихідного сигналу
                                      // до вектора ishod
vector<double> fun(8,0); // вектор, до якого записуються відліки
                         // поточної базисної функції
int kol_block = ceil(float(ishod.size())/8); // розрахунок кількості блоків
                                             // довжини 8
 int ras = (kol_block*8)-ishod.size(); // кількість відліків, якими необхідно
                                       // доповнити вихідний сигнал
                                       // для забезпечення кратності 8
 if(ras>0) ishod.insert(ishod.end(),ras,0); // доповнення вихідного сигналу
                                            // нульовими відліками
 vector<double>::iterator tinp = ishod.begin(); // ітератор, який вказує
                                                // на елементи
                                                // вихідного сигналу
 for(int i=0; i<kol_block; i++) // цикл обробки блоків сигналу довжини 8
  {block8.assign(tinp,tinp+8); // копіювання відліків поточного
                               // блоку до вектора block8
```

Рис. 3.6. Функція ортогонального перетворення

```
bs = basa; // ініціалізація ітератора на перший елемент масиву
        // відліків базисних функцій
for(int j=0;j<8;j++) // цикл ОП поточного блоку довжини 8
  { fun.assign(bs, bs+8); // копіювання відліків j-ї базисної
                                    // функції до вектора fun
    transform(block8.begin(), block8.end(), fun.begin(),tmp.begin(),
    multiplies<double>()); // поелементне перемноження елементів
            // поточного блоку  відліків вихідного сигналу
            // з відліками j-ї базисної функції
    double sum = 0; // об'явлення акумулятора та його ініціалізація
    sum = accumulate (tmp.begin(), tmp.end(), sum);
          // сумування поелементних добутків,
          // отриманих у попередньому операторі -
          // формування коефіцієнта перетворення
    out.push_back(ceil(sum)); // запис отриманого коефіцієнта до
                              // вектора коефіцієнтів перетворення
     tmp.assign(8,0); // очищення вектора добутків
     bs += 8; // перехід до наступної базисної функції
   }
 tinp += 8; // перехід до наступного блоку відліків
              // вихідного сигналу довжини 8
 }
}
```

Рис. 3.6. Функція ортогонального перетворення (закінчення)

Дану функцію можна використовувати й для виконання зворотного ОП. Для цього необхідно:

– як перший аргумент передати покажчик на транспоновану матрицю відліків базисних функцій;

– змінити тип другого аргументу на *vector<int> &* або *vector<float> &* (у залежності від типу вектора, який зберігає коефіцієнти перетворення) і передавати як другий аргумент посилання на вектор коефіцієнтів перетворення;

– змінити тип третього аргументу на *vector<int> &* або *vector<unsigned char> &* (у залежності від типу вектора, який зберігає відліки відновленого сигналу) і передавати як третій аргумент посилання на вектор відліків відновленого сигналу.

Реалізувати пряме і зворотне перетворення за допомогою однієї функції можна використовуючи механізм перевантаження функцій.

3.4.2. Функція, яка реалізує обчислення помилки перетворення (середньоквадратичного відхилення)

1. Опис аргументів функції:
ish – вектор, який містить відліки вихідного сигналу;
vos – вектор, який містить відліки відновленого сигналу.

2. Функція обчислення СКВ наведена на рис. 3.7.

```
float SCO(vector<unsigned char> ish, vector<int> vos)
{ vector<int> tmp;
  tmp.assign(ish.begin(),ish.end());
  vector<int> tmp2(ish.size(),0);
  int sum = 0;
  transform(tmp.begin(), tmp.end(), vos.begin(),tmp2.begin(),
minus<int>());
  sum = accumulate (tmp2.begin(), tmp2.end(), sum, step2);
  float sc = double(sum)/tmp2.size();
  return sqrt(sc);
}
```

Рис. 3.7. Функція обчислення СКВ

3.4.3. Функція визначення часу виконання перетворень

Програмно визначити час виконання ОП з достатньо високою точністю можливо за допомогою мультимедійних функцій WIN API *timeGetTime()*. Дана функція повертає системний час у мілісекундах. Для її використання необхідно підключити заголовний файл *mmsystem.h*. На рис. 3.8 наведений приклад використання функції *timeGetTime()*.

```
#include <mmsystem.h>
...
unsigned int time1pr = timeGetTime(); // отримання системного часу
fOP((float *)mDkp, signal, trans); // виконання ОП
unsigned int time2pr = timeGetTime(); // отримання системного часу
unsigned int time = time2pr – time2pr;
                        // визначення часу, витраченого на виконання
                        // функції fOP((float *)mDkp, signal, trans)
```

Рис. 3.8. Приклад використання функції *timeGetTime()*

4. ДИСКРЕТНІ ВЕЙВЛЕТ-ПЕРЕТВОРЕННЯ ЦИФРОВИХ СИГНАЛІВ

4.1. Загальні відомості про дискретні вейвлет-перетворення

Вейвлет-перетворення надають можливість проведення аналізу сигналів як у часовій, так і в спектральній областях, що визначається хорошію локалізацією базисних функцій вейвлет-перетворень у зазначених областях. Наприклад, базисні функції перетворення Фур'є локалізовані тільки в частотній області, що значно утруднює аналіз локальних особливостей сигналів у часовій області. На основі вейвлет-перетворень вже створена велика кількість різних методів обробки сигналів в області стиску (JPEG 2000), фільтрації, розпізнавання образів та ін.

Під вейвлетами розуміють функції, зсуви та розтягування яких утворюють базис багатьох важливих просторів, в тому числі й в $L^2(R)$, до якого належить більшість сигналів, що розглядаються теорією цифрової обробки сигналів. Вейвлет-функції є компактними як в часовій так й у частотній областях. Вейвлети можуть бути ортогональними, напівортогональними, біортогональними. Вони також можуть бути симетричними, асиметричними та несиметричними. Розрізняють вейвлети з компактною областю визначення та без неї. Вони характеризуються ступенем гладкості. Для більшості практичних задач необхідно, щоб вейвлети були ортогональними, симетричними (асиметричними) та з компактною областю визначення. Єдиним вейвлетом, що відповідає цим вимогам, є вейвлет Хаара. Але вейвлети Хаара характеризуються низьким ступенем гладкості, що обмежує їх використання в деяких практичних задачах.

В даному посібнику основи вейвлет-перетворень розглядаються на прикладі вейвлетів Хаара.

Вейвлет-базиси визначається двома функціями:

$$\phi_{j,k}(t) = 2^{j/2}\,\phi(2^j t - k); \tag{4.1}$$

$$\psi_{j,k}(t) = 2^{j/2}\,\psi(2^j t - k), \tag{4.2}$$

де змінні $j, k \in Z$; j – рівень розкладання або масштаб; k – зсув базисної функції.

Наведені функції називаються відповідно скейлинг-функцією і вейвлет-функцією. Система, яка побудована із зміщених і відмасштабова-

них копій зазначених функцій, є ортонормованим базисом $L^2(R)$. В загальному випадку теорія вейвлет-перетворень нерозривно пов'язана з теорією кратномасштабного аналізу.

Позначимо через V^0 простір всіх функцій, які є постійними на інтервалі [0, 1). Тоді V^0 – векторний простір функцій, тобто якщо ми складемо дві постійні функції, то сума буде постійною функцією і також належатиме V^0 і якщо ми помножимо постійну функцію на число (скаляр), то результат буде постійною функцією і належатиме V^0. Базисна масштабуюча функція ϕ належить V^0. Фактично, кожен елемент простору V^0 може бути отриманий множенням ϕ на відповідну константу. Таким чином $\{\phi\}$ складає (досить тривіальний) базис простору V^0.

Розглянемо декілька складніший простір функцій. Нехай V^1 – простір кусково-постійних функцій, що є константами на інтервалах [0, 1/2) і [1/2, 1). V^1 – це теж векторний простір функцій. Масштабуючі функції $\phi_{1,0}(t)$ і $\phi_{1,1}(t)$ також є елементами простору V^1. Решта всіх елементів простору V^1 може бути представлена лінійною комбінацією функцій $\phi_{1,0}(t)$ і $\phi_{1,1}(t)$. Можна показати, що функції $\{\phi_{1,0}(t), \phi_{1,1}(t)\}$ утворюють базис простору V^1. Відмітимо також, що функція, яка постійна на інтервалі [0,1), є постійною і на кожному з інтервалів [0, 1/2) і [1/2, 1), тому кожен елемент V^0 є елементом V^1, тобто $V^0 \subset V^1$. Продовжуючи далі таким чином, визначимо V^2 як простір кусково-постійних функцій на інтервалах [0, 1/4),...,[3/4, 1), і V^n – як простір кусково-постійних функцій на рівновіддалених інтервалах завдовжки $1/2^n$. Кожен простір V^n – це векторний простір, масштабуючі функції якого $\{\phi_{n,j}(t) \quad j=0,...,2^{n-1}\}$ утворюють в ньому базис. Крім того, простори V^n послідовно вкладені один в одного, тобто

$$V^0 \subset V^1 \subset ... \subset V^n \subset V^{n+1} \subset ...$$

Тепер ми визначимо скалярний добуток елементів простору V^n:

$$<f,g>=\int_0^1 f(t)g(t)dt .$$

Векторний простір з введеним скалярним добутком називається евклідовим простором. Дві функції у цьому просторі називаються ортогональними щодо скалярного добутку, якщо $<f,g>=0$.

Ортогональність функцій є корисною з декількох причин.

Перш за все, відмітимо що, $<\phi_{1,0},\phi_{1,1}>=0$, тому $\{\phi_{1,0}(t),\ \phi_{1,1}(t)\}$ утворюють ортогональний базис для V^1. Фактично для кожного j і $k \neq \ell$, ми маємо $<\phi_{j,k},\phi_{j,\ell}>=0$, так що $\{\phi_{j,k},\ \ k=0,...,2^j-1\}$ – це множина взаємно ортогональних базисних векторів в V^j. Відмітимо також, що $<\psi_{j,k},\psi_{j,\ell}>=0$ при $k \neq \ell$.

По-друге, для даного евклідова простору U, що належить більшому евклідову простору S, ми можемо говорити про множину векторів в S, ортогональних всім векторам в U. Ця множина називається ортогональним доповненням простору U в просторі S і позначається $U^{\perp}$. Неважко перевірити, що вона також є векторним простором (а також і евклідовим простором). Розглянемо такий простір

$$W^j \equiv \{h \in V^{j+1} :< h, f >= 0 \text{ для всіх } f \in V^j\}.$$

Простір W^j визначений як ортогональне доповнення простору V^j в просторі V^{j+1}. Ми вже знайомі з деякими елементами W^j. Розглянемо $\psi_{j,k}$. Відмітимо, що довжини інтервалів, на яких $\psi_{j,k}$ постійна, складають половину довжини інтервалів, на яких елементи простору V^j є константами. Іншими словами, $\psi_{j,k} \in V^{j+1}$ при кожному k. Більш того, легко переконатися в тому, що $<\psi_{j,k},f>=0$ для кожного $f \in V^j$ і, таким чином, $\psi_{j,k} \in W^j$ при кожному k і кожному j. Розглянемо, наприклад $f \equiv f_0\phi_{1,0} + f_1\phi_{1,1} \in V^1$, де f_0, f_1 є скалярними константами. Тоді

$$<\psi_{1,0},f>= f_0 <\psi_{1,0}\phi_{1,0}> + f_1 <\psi_{1,1}\phi_{1,1}>= 0+0=0,$$

тому що $<\psi_{1,0}\phi_{1,0}>= \int_0^{1/4}(1)(1)dt + \int_{1/4}^{1/2}(-1)(1)dt = 0$ і $<\psi_{1,0}\phi_{1,0}>= \int_{1/2}^{1}(0)(1)dt = 0$.

Таким чином, $\psi_{1,0} \in W^1$. Аналогічно доводиться, що $\psi_{1,1} \in W^1$ і взагалі, $\psi_{j,k} \in W^j$.

Яку розмірність має простір W^j? Вочевидь, вона не більша, ніж розмірність V^{j+1}, оскільки $W^j \subset V^{j+1}$. Таким чином, розмірність W^j не може бути більше, ніж 2^{k+1} (ми говоримо тут про розмірність векторного простору, тобто про кількість елементів в базисі). Оскільки

$\left\{\psi_{j,k} : k = \overline{0, 2^j - 1}\right\}$ – це множина із 2^j взаємно ортогональних і, отже, незалежних векторів, то розмірність W^j дорівнює принаймні 2^j. Але це і найбільша можлива розмірність, оскільки в V^{j+1} існує інша множина 2^j взаємно ортогональних векторів, а саме $\left\{\phi_{j,k} : k = \overline{0, 2^j - 1}\right\}$, і кожний з цих векторів також ортогональний кожному з векторів $\psi_{j,k}$. Тоді будь-який вектор з W^j, який не може бути виражений через вектори $\psi_{j,k}$, може бути вираженим через вектори $\phi_{j,k}$, а це неможливо.

Ми прийшли до наступного висновку: розмірність простору W^j дорівнює 2^j, а і $\left\{\psi_{j,k} : k = \overline{0, 2^j - 1}\right\}$ є базисом в W^j. Іншими словами, вейвлет-функції утворюють базис простору, ортогонального доповнення V^j в V^{j+1}. Побічний результат викладеного вище є в тому, що знайдено альтернативний базис для простору вищого порядку V^{j+1}. Ми вже знаємо, що одним базисом V^{j+1} є $\{\left\{\phi_{j+1,k} : \overline{k = 0, 2^{j+1} - 1}\right\}\}$. Альтернативним базисом є $\{\varphi_{j,0}, \varphi_{j,1}, ..., \varphi_{j,2^j-1}, \psi_{j,0}, \psi_{j,1}, ..., \psi_{j,2^j-1}\}$. Тоді можна задати один крок процесу вейвлет-перетворення, а саме крок, що полягає в переході від розрізнення V^{j+1} до нижчого розрізнення, як представлення елементу $g_{j+1} \in V^{j+1}$. Припустимо, що $g_{j+1} \in V^{j+1}$ спочатку був виражений як

$$g_{j+1} = a_{j+1,0}\varphi_{j+1,0} + ... + a_{j+1,2^j-1}\varphi_{j+1,2^j-1}.$$

Тоді елемент g_{j+1} може бути розкладений, наприклад, таким чином:

$$g_{j+1} = a_{j,0}\varphi_{j,0} + ... + a_{j,2^j-1}\varphi_{j,2^j-1} + d_{j,0}\psi_{j,0} + ... + d_{j,2^j-1}\psi_{j,2^j-1},$$

а коефіцієнти $\{d_{j,0}, ..., d_{j,2^j-1}\}$ стають частиною вейвлет-перетворення. На наступному кроці цього процесу ми б отримали вираз

$$g_j = a_{j,0}\varphi_{j,0} + ... + a_{j,2^j-1}\varphi_{j,2^j-1} \in V^j$$

У цілому, багаторівневе вейвлет-перетворення можна представити у вигляді наступної схеми

$$\begin{array}{ccccccc} V^0 & \to & V^1 & \to & V^2 & \to \dots \to & V^j \\ & \searrow & & \searrow & & \searrow & \\ & & W^1 & & W^2 & & W^j \end{array}$$

Так само говорять, що функції $\varphi_{j,k}$ і вейвлети $\psi_{j,k}$ є відповідно низькочастотним і високочастотним фільтром.

Вигляд базисних функцій визначається такими рекурентними виразами:

$$\varphi(t) = \sqrt{2}\sum_{k=0}^{2M-1} h_k \varphi(2t-k)\,; \tag{4.3}$$

$$\psi(t) = \sqrt{2}\sum_{k=0}^{2M-1} g_k \varphi(2t-k)\,; \tag{4.4}$$

$$g_k = (-1)^k h_{2M-k-1}, \tag{4.5}$$

де $M \in Z$; h_k і g_k – деякі послідовності, що визначають вид базисних функцій.

У випадку, коли M = 1 і всі $h_k = 1/\sqrt{2}$, то відповідно до (3.3) – (3.5) одержимо скейлинг- і вейвлет-функції ($\varphi(t)$, $\psi(t)$) Хаара (рис. 4.1).

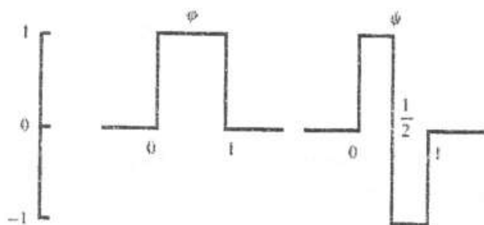


Рис. 4.1. Скейлинг-функція $\phi(t)$ та «материнський вейвлет» Хаара $\psi(t)$

Припустимо, що є послідовність, що складається з 2^n точок для деякого цілого $n > 0$. Можна ототожнити її з наступною функцією з V^0:

$$f(t) = x_1\varphi_{0,0}(t) + \dots + x_{2^n}\varphi_{0,2^n-1}(t). \tag{4.6}$$

Першим кроком обчислення вейвлет-перетворення послідовності $\{x_1, x_2, \dots, x_{2^n}\}$ буде розкладання f(t) за альтернативним базисом простору V^0, половину якого складають вейвлети:

$$\begin{aligned} f(t) = c_{1,0}\varphi_{1,0}(t) + \dots + c_{1,2^{n-1}-1}\varphi_{1,2^{n-1}-1}(t) + \\ + d_{1,0}\psi_{1,0}(t) + \dots + d_{1,2^{n-1}-1}\psi_{1,2^{n-1}-1}(t). \end{aligned} \tag{4.7}$$

Коефіцієнти $\{d_{1,0},...,d_{1,2^{n-1}-1}\}$ при базисних вейвлет-функціях складають половину коефіцієнтів вейвлет-перетворення, тому ми збережемо ці значення. Наступним кроком процесу перетворення є застосування такого ж базисного перетворення до інших членів рівності (4.7):

$$g_1(t) = c_{1,0}\varphi_{1,0}(t) + ... + c_{1,2^{n-1}-1}\varphi_{1,2^{n-1}-1}(t). \qquad (4.8)$$

Таким чином, g_1 – це елемент V^1 і тому може бути розкладений за альтернативним базисом, що складається з масштабуючих функцій $\varphi_{2,k}$ і вейвлетів $\psi_{2,k}$.

Перш ніж продовжити цей процес, поставимо таке питання: як отримати коефіцієнти рівності (4.8) з коефіцієнтів рівності (4.7)? Для цього ми використовуємо ортогональність. Нагадаємо, що кожна функція $\varphi_{1,k}$ ортогональна кожній функції $\varphi_{1,k}$ так само як і всім, і, аналогічно, кожен вейвлет $\psi_{1,k}$ ортогональний іншим вейвлетам $\psi_{1,k}$ і всім масштабуючим функціям $\varphi_{1,k}$. Нагадаємо також, що кожна функція $\varphi_{1,k}$ і кожен вейвлет $\psi_{1,k}$ є нормованими через тотожність (4.1) і (4.2). Щоб скористатися цією ортогональністю і нормованістю, помножимо обидві частини (4.8) на $\varphi_{1,k}(t)$ і проінтегруємо по t від 0 до 1. В результаті отримаємо

$$\int_0^1 f(t)\varphi_{1,k}(t)dt = c_{1,k}. \qquad (4.9)$$

Через ортогональність у правій частині (4.9) залишається тільки один член, а нормування призводить до відсутності коефіцієнта при $c_{1,k}$. Тепер підставимо праву частину рівності (4.7) замість $f(t)$ в (4.9). Наприклад, при $k = 0$ ліва частина рівності (4.9) буде такою:

$$\int_0^{1/2^n} x_1\sqrt{2^n}\sqrt{2^{n-1}}dt + \int_{1/2^n}^{2/2^n} x_2\sqrt{2^n}\sqrt{2^{n-1}}dt = (x_1 + x_2)\left(1/\sqrt{2}\right)2^n\left(1/\sqrt{2}^n\right) = \qquad (4.10)$$

$$= (x_1 + x_2)/\sqrt{2}.$$

Комбінуючи (3.9) і (3.10) при $k = 0$, отримаємо

$$c_{1,0} = (x_1 + x_2)/\sqrt{2}. \qquad (4.11)$$

Квадратний корінь у знаменнику коефіцієнта (4.11) з'являється за рахунок нормування. Якби використовувались ненормовані базисні функції, то було б набуто двоточкового середнього значення.

Решта коефіцієнтів $c_{1,k}$, $k = 1,...,2^{n-1} - 1$ обчислюється аналогічно.

Таким чином

$$c_{1,k} = \frac{x_{2k} + x_{2k+1}}{\sqrt{2}}, \quad k = 0,...,2^{n-1} - 1. \tag{4.12}$$

Аналогічно, використовуючи властивості ортогональності і нормованості функцій, можна обчислити коефіцієнти $d_{1,k}$ за наступною формулою:

$$d_{1,k} = \frac{x_{2k} - x_{2k+1}}{\sqrt{2}}, \quad k = 0,...,2^{n-1} - 1. \tag{4.13}$$

Аналогічно можна отримати формули для любого кроку (рівня) j вейвлет-перетворення

$$c_{j+1,k} = \frac{c_{j,2k} + c_{j,2k+1}}{\sqrt{2}}; \tag{4.14}$$

$$d_{j+1,k} = \frac{c_{j,2k} - c_{j,2k+1}}{\sqrt{2}}, \tag{4.15}$$

де j – рівень (крок) вейвлет-перетворення; $c_{j+1,k}$ – k-й коефіцієнт перетворення в базисі $\varphi_{j,k}(t)$; $d_{j+1,k}$ – k-й коефіцієнт перетворення в базисі $\psi_{j,k}(t)$.

Формули (4.14) та (4.15) і є прямим вейвлет-перетворенням в базисі Хаара.

Аналогічним чином можливо отримати формули для зворотного вейвлет-перетворення Хаара:

$$c_{j,2k} = \frac{c_{j+1,k} + d_{j+1,k}}{\sqrt{2}}; \tag{4.16}$$

$$c_{j,2k+1} = \frac{c_{j+1,k} - d_{j+1,k}}{\sqrt{2}}. \tag{4.17}$$

Розглянемо порядок обчислення коефіцієнтів вейвлет-перетворення на прикладі вейвлетів Хаара. На рис. 4.2 наведена схема прямого m-етапного вейвлет-перетворення Хаара. На даному рисунку безперервними стрілками показано обчислення коефіцієнтів вейвлет-перетворення за допомогою виразу (4.14), а переривчастими стрілками – за допомогою виразу (4.15).

Схема зворотного перетворення наведена на рис. 4.3. Безперервними стрілками на даному рисунку вказано застосування виразу (4.16), а переривчастими стрілками – виразу (4.17).

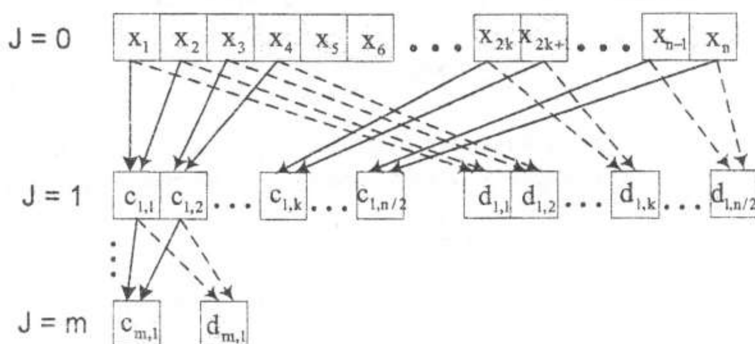


Рис. 4.2. Схема прямого m-етапного вейвлет-перетворення Хаара

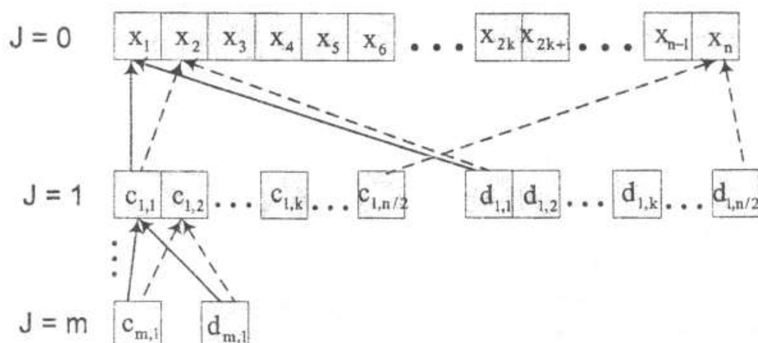


Рис. 4.3. Схема зворотного m-етапного вейвлет-перетворення Хаара

4.2. Програмна реалізація дискретного вейвлет-перетворення у базисі Хаара

У даному підрозділі наведено приклади процедур (функцій), що є програмною реалізацією прямого та зворотного вейвлет-перетворення у базисі Хаара. Приведені приклади програмних процедур (функцій) реалізовані на мові програмування C++ з використанням STL.

4.2.1. Визначення функції, що реалізує пряме вейвлет-перетворення

У даному прикладі вирази (4.14) і (4.15) реалізуються у вигляді шаблону класу унарної функції (рис. 4.4).

```
//шаблон класу унарної функції для прямого перетворення
template <class Arg>
class out_times_x :  private unary_function<Arg,void>
{ private:
  Arg x1; // поле, в якому зберігають відліки з непарними номерами у
          // послідовності, що обробляється
  bool parit; // логічна зміна, яка вказує, чи знаходиться аргумент
              //функції у  вихідній послідовності
              // на парному місці (false – номер аргументу у
              // послідовності непарний, true – номер парний)
  vector<Arg>& h; // посилання на вектор, до якого записуються
                  // високочастотні  елементи трансформанти
   vector<Arg>& l; // посилання на вектор, до якого записуються
                   // низькочастотні елементи трансформанти
  public:
// конструктор класу унарної функціїї
  out_times_x(vector<Arg>& outh, vector<Arg>& outl): h(outh), l(outl)
   { x1=0;
     parit = false; // номер першого елемента оброблюємого вектору є
                    // непарним
   }
 void operator()(const Arg& x) // визначення оператора ()
  { Arg tmph, tmpl;
    if(parit)  // якщо x має парний номер у вхідній послідовності
     { tmph = (x1 - x)/sq2; // то виконуємо розрахунок коефіцієнтів
       tmpl = (x1 + x)/sq2; // вейвлет-перетворення
       h.push_back(tmph); // отримані коефіцієнти записуємо в кінець
       l.push_back(tmpl); // векторів transh та transl
       parit = false; // наступний аргумент функції має непарний номер
     }
    else  // інакше (якщо x має непарний номер у вхідній послідовності)
     { x1 = x; // записуємо  значення до поля x1
       parit = true; // наступний аргумент функції має парний номер
     }
  }
}
```

Рис. 4.4. Шаблон класу унарної функції,
що реалізує пряме вейвлет-перетворення

Даний шаблон у функції прямого вейвлет-перетворення (рис. 4.5) використовується в STL-алгоритмі *for_each*.

Опис аргументів функції прямого вейвлет-перетворення:
inp – посилання на вектор, який містить відліки сигналу, що обробляється;
outh – посилання на вектор, до якого записуються коефіцієнти $d_{j,k}$;
outl – посилання на вектор, до якого записуються коефіцієнти $c_{j,k}$.

```
void fWP(vector<unsigned char> & inp, vector<float> & outh,
vector<float> & outl)
{ if((inp.size()%2)!=0) inp.push_back(0); // якщо вхідний вектор відліків
// має непарну довжину,то дописати в кінець один нульовий відлік
   out_times_x<float> w(outh, outl); // створення екземпляру класу
                              // унарної функції для прямого перетворення w
   for_each(inp.begin(), inp.end(), w); // для кожного елемента вихідного
                                    // вектора inp виконати унарну операцію w
}
```

Рис. 4.5. Визначення функції прямого вейвлет-перетворення

4.2.2. Визначення функції, що реалізує зворотне вейвлет-перетворення

У даному прикладі (рис. 4.6) вирази (4.16) і (4.17) також реалізуються у вигляді шаблона класу унарної функції.

```
template <class Arg, class Arg2>  //шаблон класу унарної функції для
                                  // зворотного перетворення
class out_times_s :  private unary_function<Arg,void >
 { private:
   vector<Arg2>& s; // посилання на вектор, до якого записуються
                      // відновлені відліки сигналу
  vector<Arg>& h; // посилання на вектор, що містить
                    // високочастотні коефіцієнти перетворення
  vector<Arg>::iterator ih; // ітератор (вказівник) на елементи вектора h
  public:
    // конструктор
```

Рис. 4.6. Шаблон класу унарної функції,
що реалізує зворотне вейвлет-перетворення

```
out_times_s(vector<Arg>& outh, vector<Arg2>& outs): h(outh), s(outs)
{ ih = h.begin(); // встановлення ітератора на перший елемент вектора h
}
void operator()(const Arg& x) //  визначення оператора ()
{ Arg2 tmpx1, tmpx2;
   tmpx1 = ceil((x + (*ih))/sq2); // розрахунок відліків сигналу X(2*i)
   tmpx2 = ceil((x - (*ih))/sq2); // та  X(2*i+1)
   s.push_back(tmpx1); // запис відновлених відліків  X(2*i)
   s.push_back(tmpx2); // та  X(2*i+1)
   ih++; // встановлення ітератора на наступний елемент вектора h
  } }
```

Рис. 4.6. Шаблон класу унарної функції, що реалізує зворотне вейвлет-перетворення (закінчення)

Даний шаблон у функції зворотного вейвлет-перетворення (рис. 4.7.) використовується в STL-алгоритмі *for_each*.

```
void fWB(vector<int> & out, vector<float> & inpl, vector<float> & inph)
{ out_times_s<float, int> w(inph, out); // створення екземпляру класу
                                        // унарної функції
                                        // для зворотного перетворення w
 for_each(inpl.begin(), inpl.end(), w); // для кожного елемента векторів,
                        // що містять коефіцієнти перетворення
                        // (inpl та inph) виконати унарну функцію w
}
```

Рис. 4.7. Визначення функції зворотного вейвлет-перетворення

5. ДИСКРЕТНІ ЙМОВІРНІСНІ ДЖЕРЕЛА

5.1. Загальні відомості про дискретні ймовірності джерела

Для стиску дискретних сигналів широко застосовуються статистичні методи стиску (СМС), що усувають статистичну надмірність сигналів. При цьому дискретний сигнал представляється як випадкова послідовність символів деякого алфавіту A, породжена джерелом S. Статистичні (інформаційні) властивості випадкової послідовності цілком визначаються властивостями джерела S, що породжує дану послідовність. Як правило, ці властивості одержують на основі аналізу досить тривалої послідовності символів, породженої джерелом S. Для цього досить визначити тип цього джерела та імовірності появи символів його алфавіту. Розрізняють три основних типи джерела:

– джерело Хартлі, що породжує рівноймовірнісні і незалежні символи алфавіту;

– джерело Бернуллі, що породжує незалежні символи алфавіту джерела з різною ймовірністю;

– джерело Маркова, що породжує випадкові послідовності символів, у яких ймовірність появи символів різна і залежить від символів, породжених раніше.

Основною інформаційною характеристикою джерел є ентропія, яка визначає інформаційну ємність алфавіту джерела і є мірою невизначеності породження елемента алфавіту даного джерела. Ентропію для зазначених типів джерел одержують за допомогою наступних виразів:

– джерело Хартлі:

$$H_0 = -\log_2 p(A) = \log_2 N\,, \tag{5.1}$$

де H_0 – ентропія джерела Хартлі; $p(A)$ – ймовірність символів алфавіту $A = \{a_i\}$, $i = \overline{0, N-1}$, причому для джерела Хартлі ймовірності $p(a_i) = 1/N$, $i = \overline{0, N-1}$; N – потужність алфавіту A;

– джерело Бернуллі:

$$H_1 = -\sum_{i=0}^{N-1} p(a_i)\log_2 p(a_i)\,, \tag{5.2}$$

де H_1 – ентропія джерела Бернуллі; $p(a_i)$ – ймовірність символів алфавіту $A = \{a_i\}$, $i = \overline{0, N-1}$;

– *джерело Маркова r-го порядку*:

$$H_r = -\sum_{i_1=0}^{N-1} \sum_{i_2=0}^{N-1} \dots \sum_{i_{r+1}=0}^{N-1} p(a_{i_1} \dots a_{i_{r+1}}) \log_2 p(a_{i_{r+1}} / a_{i_1} \dots a_{i_r}), \quad (5.3)$$

де H_r – ентропія джерела Маркова r-го порядку, при цьому

$$H_0 \geq H_1 \geq H_2 \geq \dots \geq H_r. \quad (5.4)$$

Розглянемо властивості ентропії і інформації.

Твердження 1. Нехай $A = \{A_1, A_2, \dots, A_k\}$ – алфавіт, тоді

$$H(A) \leq \log k.$$

Доказ. Розглянемо функцію $f(x) = -x \log x$. Функція $f(x)$ опукла вгору при $x > 0$, оскільки $f''(x) = -1/(x \ln 2) < 0$.

Тоді для функції f виконується нерівність Єнсена:

$$\text{якщо } \sum_{i=1}^{k} \alpha_i = 1 \text{ і } \alpha_i > 0, \text{ то } \sum_{i=1}^{k} \alpha_i f(x_i) \leq f\left(\sum_{i=1}^{k} \alpha_i x_i\right).$$

Нехай $\alpha_i = 1/k$, а $x_i = p(A_i)$. Тоді

$$H(A) = k \sum_{i=1}^{k} \frac{1}{k} f(p(A_i)) \leq kf\left(\sum_{i=1}^{k} \frac{p(A_i)}{k}\right) = -k\left(\sum_{i=1}^{k} \frac{p(A_i)}{k}\right) \log\left(\sum_{i=1}^{k} \frac{p(A_i)}{k}\right) = \log k.$$

Твердження доведене.

Нехай $A = \{A_1, A_2, \dots, A_k\}$ і $B = \{B_1, B_2, \dots, B_n\}$ – розбиття. Ентропією A при заданому B називається $H(A \setminus B) = \sum_j p(B_j) H(A \setminus B_j)$. Відмітимо, що $H(A \setminus A) = 0$. Через AB позначатимемо розбиття, що складається із всіх подій вигляду $A_i B_j$, де $i = \overline{1,k}, j = \overline{1,n}$.

Твердження 2. Нехай $A = \{A_1, A_2, \dots, A_k\}$ і $B = \{B_1, B_2, \dots, B_n\}$ – розбиття, тоді $H(AB) = H(B) + H(A \setminus B)$.

Доказ.

$$H(AB) = -\sum_{i,j} p(A_i B_j) \log p(A_i B_j) = -\sum_{i,j} p(A_i B_j)\left(\log p(A_i \setminus B_j) + \log p(B_j)\right) =$$

$$= -\sum_j \left(p(B_j) \sum_i p(A_i \setminus B_j) \log p(A_i \setminus B_j) \right) - \sum_j \left(\log p(B_j) \sum_i p(A_i B_j) \right) =$$

$$= H(A \setminus B) + H(B).$$

Остання рівність виходить з формули повної ймовірності

$$\sum_i p(A_iB_j) = \sum_i p(A_i)p(B_j \setminus A_i) = p(B_j)$$

і визначення $H(A \setminus B)$. Твердження доведене.

На множині розбиття простору Ω визначимо частковий порядок $B \succ A$ (B інформативніше за A) якщо $BA = B$.

Твердження 3. Хай A, B, C – розбиття, $B \succ C$. Тоді $H(A \setminus B) < H(A \setminus C)$.

Доказ. За умовою твердження BC=B, тоді $C_mB_j = B_j$ або $C_mB_j = 0_j$. Оскільки множини C_m не перетинаються, то для кожного B_j знайдеться єдине C_m таке, що $B_j = C_mB_j$. Множину чисел j, для яких $B_j = C_mB_j$, позначимо через j(m). Введемо позначення $B_m^j = B_j$, якщо $j \in j(m)$.

Тоді $C_m = \sum_{j(m)} B_j^m$ і справедливі наступні співвідношення:

$$-\sum_{j(m)} p(A_iB_j^m)\log p(A_i \setminus B_j^m) = -p(C_m)\sum_{j(m)} \frac{p(B_j^m)}{p(C_m)} p(A_i \setminus B_j^m)\log p(A_i \setminus B_j^m) \le$$

$$\le -\left(\sum_{j(m)} p(B_j^m)p(A_i \setminus B_j^m)\right)\log\left(\sum_{j(m)} p(B_j^m)p(A_i \setminus B_j^m) \Big/ p(C_m)\right) \le$$

$$\le -p(C_mA_i)\log p(A_i \setminus C_m),$$

де перша нерівність виходить з нерівності Йенсена для функції –xlogx, друге – із формули повної ймовірності. Тоді

$$H(A \setminus B) = -\sum_i \sum_m \sum_{j(m)} p(A_iB_j^m)\log p(A_i \setminus B_j^m) \le$$

$$\le -\sum_i \sum_m p(C_mA_i)\log p(A_i \setminus C_m) = H(A \setminus C).$$

Твердження доведене.

Наслідок 1. $H(A \setminus B) < H(A)$.

Доказ. Розглянемо $C = \{\Omega\}$. Неважко переконатися, що $H(A \setminus C) = H(A)$ і для всього розбиття B виконано $B \succ C$. Тоді нерівність $H(A \setminus B) < H(A)$ виходить з попереднього твердження.

Інформацією розбиття A щодо розбиття B називається

$$I(A, B) = H(A) - H(A \setminus B).$$

Відмітимо, що $I(A, A) = H(A)$.

Наслідок 2. Якщо $B \succ C$, то $I\{A,B\} > I\{A,C\}$.

Доказ наслідку виходить з визначення величини I(A, B) і твердження 3.

Ентропія дозволяє визначити як інформаційну ємність повідомлень, сформованих джерелом так і надмірність кодування цих повідомлень. Наприклад, якщо потужність алфавіту A дискретного сигналу дорівнює 256 і для збереження кожного відліку приділяється 8 біт, то надмірність такого представлення дискретного сигналу для джерела Бернуллі визначається так:

– в абсолютних величинах:

$$R_{10} = H_0 - H_1 ; \tag{5.5}$$

– у відносних величинах:

$$\hat{R}_{10} = \frac{H_0 - H_1}{H_0} = 1 - \frac{H_1}{H_0} . \tag{5.6}$$

Для довільного джерела порядку r ці вирази приймуть вид:

$$R_{r0} = H_0 - H_r ; \tag{5.7}$$

$$\hat{R}_{r0} = \frac{H_0 - H_r}{H_0} = 1 - \frac{H_r}{H_0} . \tag{5.8}$$

5.2. Програмна реалізація процедур обчислення характеристик ймовірнісних джерел

У даному підрозділі на рис. 5.1 – 5.5 наведені приклади програмної реалізації процедур (функцій), що виконують обчислення характеристик дискретних ймовірних джерел: гістограми алфавіту джерела; визначення ентропії джерела; визначення надмірності представлення (кодування) символів алфавіту ймовірного джерела.

```
template <class Arg>
class frec :  private unary_function<Arg, float>
{ private:
   int cnt; // загальна кількість відліків
  public:
  frec(int c): cnt(c) {}; // конструктор класу унарної функції
  // визначення оператора (), аргумент x – значення лічильника
  float operator()(const Arg & x)
  { return x/float(cnt); // отримати частоту
   }
};
```

Рис. 5.3. Шаблон класу унарної функції, яка виконує поділ символьних лічильників на загальну кількість відліків сигналу для отримання частот символів

```
void Gist(vector<unsigned char> & sign, vector<float> & gst)
{ vector<unsigned int> tmp; // лічильники символів алфавіту
// визначення потужності алфавіту
 vector<unsigned char>::iterator max = max_element(sign.begin(),
sign.end());
// екземпляр унарної функції, яка виконує підрахунок кількості
// входжень символу до сигналу, що аналізується
 gist<unsigned int> L(tmp,(*max));
 for_each(sign.begin(),sign.end(),L); // для відліків сигналу підрахувати
// кількість кожного символу алфавіту
 gst.assign(tmp.size(),0.00); // ініціалізація гістограми
// екземпляр класу унарної функції, яка виконує поділ символьних
// лічильників на загальну кількість відліків
// сигналу для отримання частот символів
 frec<unsigned int> F(sign.size());
 transform(tmp.begin(), tmp.end(),gst.begin(), F); // розрахунок
// гістограми
}
```

Рис. 5.1. Визначення функції, яка формує гістограму алфавіту джерела: *sign* – посилання на вектор, що містить відліки сигналу; *gst* – посилання на вектор, до якого записуються відліки гістограми

```
template <class Arg>
class gist : private unary_function<Arg,void>
{ private:
  vector<Arg> & g; // посилання на вектор із значеннями лічильників
  public:
    gist(vector<Arg> & gis, int max): g(gis) // конструктор класу
                                              // унарної функції
    { g.clear();
      g.insert(g.begin(),(max+1),0);
    }
// визначення оператора () унарної функції,
// x – значення поточного відліку сигналу
  void operator()(const Arg& x)
      { g[x]+=1; // для символу x збільшити лічильник на 1
      } };
```

Рис. 5.2. Шаблон класу унарної функції, яка виконує підрахунок кількості входжень символу до сигналу, що аналізується

```
template <class Arg>
 class xlog2x :  private unary_function<Arg,void>
 { private:
   vector<Arg>& logx; // посилання на вектор,
                          // до якого записуються значення f(x)
  public:
   xlog2x(vector<Arg>& lgx): logx(lgx) // конструктор класу
                                          // унарної функції
    { logx.clear();
    }
  void operator()(const Arg& x) // визначення оператора ()
      { double res;
        res = (-1) * x * Log2(x);
        logx.push_back(res);
      }
 };
```

Рис. 5.4. Шаблон класу унарної функції f(x) = x·log(x)

```
double Entropy(vector<float> & gst)
{ vector<double> lg; // вектор для зберігання значень -x*log(x)
 vector<double> tmp;
 tmp.assign(gst.begin(),gst.end());
 xlog2x<double> LOG(lg); // екземпляр функції -x*log(x)
// вилучення частот, значення яких дорівнює 0
 vector<double>::iterator p0 = remove (tmp.begin(), tmp.end(), 0);
 tmp.erase(p0,tmp.end());
// розрахунок для кожного x значень функції f(x) = -x*log(x)
 for_each(tmp.begin(),tmp.end(),LOG);
 double sum=0;
// розрахунок суми значень функції Σf(x) = -Σx*log(x)
 sum = accumulate (lg.begin(), lg.end(), sum);
 return sum;
}
```

Рис. 5.5. Визначення функції, що обчислює ентропію

6. СТАТИСТИЧНЕ КОДУВАННЯ (СТИСНЕННЯ) ДИСКРЕТНИХ СИГНАЛІВ

6.1. Загальні теоретичні відомості про статистичні кодери

Для цифрового представлення відліків дискретного сигналу використовуються різні двійкові коди. Простішим з них є код фіксованої довжини, який формується в результаті рівномірного квантування дискретних відліків на певну кількість рівнів (N). В цьому випадку двійковим кодом відліку дискретного сигналу є двійкове представлення значення рівня квантування найближчого до амплітуди відліку. Таке кодування характеризується високою надмірністю, оскільки не враховує статистичні властивості джерела дискретного сигналу. Іншими словами, всі рівні квантування рівноймовірнісні, що відповідає джерелу Хартлі. Декодуємість такого коду забезпечується його рівномірністю. Іншим різновидом двійкових кодів є нерівномірні префіксні коди, що враховують статистичні властивості джерела сигналу. При цьому рівням квантування, імовірність появи яких в дискретному сигналі вище, привласнюється код меншої довжини. Таке кодування називають статистичним кодуванням. Властивість префіксності даних кодів забезпечує їх декодуємість.

Дамо визначення вказаних термінів. Нехай E = {0, 1}. Позначимо через $E^* = \bigcup_{i=1}^{\infty} E^i$ множину всіх скінчених наборів 0 і 1. Нехай $A\{a_1, a_2, \ldots, a_i, \ldots\}$ – деякий алфавіт.

Кодуванням (кодом) називається ін'єктивне відображення $f: A \to E^*$. Значення $f(a_i)$ називається кодом букви, або кодовим словом. Через $|f(a_i)|$ позначимо довжину кодового слова.

Таке кодування ще називають посимвольним кодуванням.

Кодування f називається таким, що дешифрується, якщо довільна послідовність кодових слів $f(a_{i_1}), f(a_{i_2})$,..., $f(a_{i_n})$, записаних злито, однозначно поділяється на кодові слова.

Кодування f називається префіксним, якщо ніяке кодове слово $f(a_i)$ не є префіксом (початком) іншого кодового слова $f(a_j)$.

Наприклад, код, що містить два кодових слова 0 і 10, є префіксним; код, що містить два кодових слова 0 і 01, є таким, що дешифрується, але не префіксним; а код, що має кодові слова 0, 1 і 10, не є ні префіксним,

ні таким, що дешифрується. Неважко помітити, що кожен префіксний код є таким, що дешифрується, а зворотне невірно.

Розглянемо параметри статистичних кодерів, які визначають їх ефективність.

1. ***Вартість кодування*** f джерела S з алфавітом, де k – потужність алфавіту (кількість рівнів квантування) і заданою імовірністю появи символів алфавіту $p(a_i)$ визначається виразом

$$C(f,S) = -\sum_{i=0}^{k-1} p(a_i)|f(a_i)|, \qquad (6.1)$$

де $|f(a_i)|$ – довжина коду символу a_i.

Вартість кодування $C(f,S)$ показує скільки біт в середньому витрачається на кодування одного символу алфавіту A при використанні кодування f.

2. ***Надмірність кодування*** f джерела S з заданим алфавітом $A = \{a_0, a_1, ..., a_{k-1}\}$ визначається виразом

$$R(f,S) = C(f,S) - H(S), \qquad (6.2)$$

де $H(S)$ – ентропія джерела S з алфавітом $A = \{a_0, a_1, ..., a_{k-1}\}$.

Надмірність кодування $R(f,S)$ показує на скільки біт середня довжина кодів символів алфавіту A при використанні кодування f перевищує середню інформаційну місткість символів, що формуються джерелом S.

3. ***Коефіцієнт стиснення*** відліків дискретного сигналу визначається наступним виразом:

$$K = \frac{V_i}{V_s} = \frac{N \cdot \log_2 k}{N \cdot C(f,S)}, \qquad (6.3)$$

де V_i – об'єм початкової послідовності дискретних відліків в бітах (при кодуванні рівномірним кодом); V_s – об'єм стислої послідовності дискретних відліків в бітах (після застосування статистичного коду); N – кількість дискретних відліків в сигналі; k – кількість рівнів квантування дискретного сигналу.

Серед різних методів статистичного кодування можна виділити клас оптимальних статистичних кодів $\{f_0\}$.

Префіксне кодування f_0 називається оптимальним для джерела S, якщо для кожного префіксного кодування f джерела S виконується нерівність

$$R(f_0, S) \le R(f, S).$$

Для кожного джерела існує оптимальний код, оскільки множина префіксних кодів джерела з надмірністю, меншою або рівною 1, не порожня і скінчена. Одне джерело може мати декілька оптимальних кодів з різними наборами довжин кодових слів.

Вся теорія статистичного кодування базується на теоремі Шенона, яка для джерела Бернуллі (при посимвольному кодуванні) звучить наступним чином:

1) для довільного джерела S і префіксного коду f надмірність кодування ненегативна, тобто $R(f,S) > 0$;

2) для кожного джерела S знайдеться префіксний код f з надмірністю, що не перевищує одиниці, тобто $R(f,S) < 1$.

Доказ цієї теореми можна знайти у рекомендованій літературі.

Введемо поняття ***блокового кодування***. Нехай $A = \{a_1, a_2, \dots, a_k\}$ і $E = \{0,1\}$. Тоді $A^* = \bigcup_{i=1}^{\infty} A^i$ – множина кінцевих послідовностей букв алфавіту A, а E^* – множина скінчених двійкових послідовностей.

Кодування f називається m-блоковим, якщо $f : A^m \to E^*$. В цьому випадку кодом слова довільної довжини є конкатенація кодів блоків, що складають дане слово. Якщо кодоване слово не розділяється на ціле число блоків довжиною m, то його потрібно доповнити довільним чином і в кінці коду слова приписати число доданих букв.

Посимвольні коди $f : A \to E^*$, що дешифруються, є окремим випадком кодування, визначеного вище, оскільки можливе довизначення

$$f(a_{i_1}, a_{i_2} \dots a_{i_m}) = f(a_{i_1}) f(a_{i_2}) \dots f(a_{i_m}).$$

Блокове кодування f називається префіксним, якщо для будь-яких двох слів $x, y \in A^n$ однакової довжини f(x) не є префіксом f(y).

Для блокового кодування f вартістю кодування стаціонарного джерела S з алфавітом A є $C(f,s) = \lim\sup_{n \to \infty} C_n(f,S)$, де

$$C_n(f,S) = \frac{1}{n} \sum_{x \in A^n} p(x) |f(x)$$

Зокрема, для n-блокового кодування f маємо $C(f,S) = C_n(f,S)$.

Теорема Шенона прийме наступний вигляд:

1) Для кожного стаціонарного джерела S з алфавітом A і довільного префіксного коду f надмірність кодування не є негативною, тобто $R(f,S) \geq 0$.

2) Для кожного стаціонарного джерела S з алфавітом A знайдеться блокове префіксне кодування з скільки завгодно малою надмірністю.

Вище було розглянуто кодування, яке блоки з однаковою кількістю букв відображає в кодові слова різної довжини. Також існує кодування, що ставить у відповідність словам різної довжини кодові слова однакової довжини. Таке кодування називають кодуванням типа VB на відміну від блокового кодування, що має тип BV. Можна також розглядати кодування типу VV, яке слова змінної довжини відображає в слова змінної довжини. Вказані види кодування більш детально розглянуті у рекомендованій літературі [8, 10].

У сучасних методах стиснення сигналів найширше використовуються такі оптимальні коди як арифметичне кодування і код Хафмана. Основною відмінністю даних кодів один від одного є те, що код Хафмана визначає процедуру побудови оптимального коду для кожного символу джерела S з алфавітом A, що володіє наступною властивістю

$$|f(a_i)| = -\lceil \log_2 p(a_i) \rceil, \tag{6.4}$$

де $p(a_i)$ – ймовірність формування джерелом S символу a_i з алфавіту $A = \{a_0, a_1, ..., a_{k-1}\}$; $\lceil \bullet \rceil$ – операція округлення до найближчого цілого вгору; k – потужність алфавіту A.

Визначимо процедуру стиснення Хафмана джерела S з алфавітом

$$A = \{a_1, a_2, ..., a_k\},$$

яка полягає в злитті двох найменше ймовірних букв.

1. Перенумеруємо букви a_i так, щоб $p(a_1) \geq p(a_2) \geq ... \geq p(a_k)$.

2. Нехай $A' = \{a'_1, a'_2, ..., a'_{k-1}, ...\}$. Визначимо джерело S' з алфавітом A' і ймовірністю букв $p(a'_i) > p(a_i)$ при $i < к-1$ і $p(a'_{k-1}) = p(a_{k-1}) + p(a_k)$. Введемо позначення $S^{(i)} = \left(S^{(i-1)}\right)'$.

За визначенням процедури стиснення алфавіт джерела $S^{(i)}$ містить $к - i$ букв. Побудуємо код Хафмана h за індукцією. Алфавіт джерела $S^{(k-2)}$ складається з двох букв ${a_1}^{(k-2)}$ і ${a_2}^{(k-2)}$. Нехай

$$h^{(k-2)}({a_1}^{(k-2)}) = 0 \text{ ; } h^{(k-2)}({a_2}^{(k-2)}) = 1,$$

а $h^{(i+1)}$ – префіксний код джерела $S^{(i+1)}$ – вже відомий. Оскільки $S^{(i+1)} = \left(S^{(i)}\right)'$ і за побудовою знайдеться буква $a_j^{i+1} \in A^{(i+1)}$ така, що $p(a_j^{i+1}) = p(a_{k-i-1}^i) + p(a_{k-i}^i)$, можна визначити:

$$h^{(i)}(a_{k-i-1}^{i}) = h^{(i+1)}(a_{j}^{i+1})0\ ;$$

$$h^{(i)}(a_{k-i}^{i}) = h^{(i+1)}(a_{j}^{i+1})1\,.$$

Кодові слова для решти букв, ймовірність яких не змінилася, залишимо тими ж самими, тобто $h^{(i)}(a_m^i) = h^{(i+1)}(a_m^{i+1})$. Ясно, що з префіксності коду $h^{(i+1)}$ слідує префіксність коду h^i. Отриманий за індукцією код $h = h^{(0)}$ – код Хафмана для джерела S.

Побудуємо, наприклад, код Хафмана для джерела з ймовірністю появи букв $p(a_5) = p(a_6) = 0{,}1$. Процес побудови коду зображений у вигляді дерева на рис. 6.1. Зліва від вершини вказана буква, що їй відповідає, а справа – ймовірність букви. Кодові слова $f(a_1) = 00$, $f(a_2) = 10$, $f(a_3) = 010$, $f(a_4) = 011$, $f(a_5) = 110$, $f(a_6) = 111$ є кодами листових вершин дерева, що підтверджує префіксність коду Хафмана.

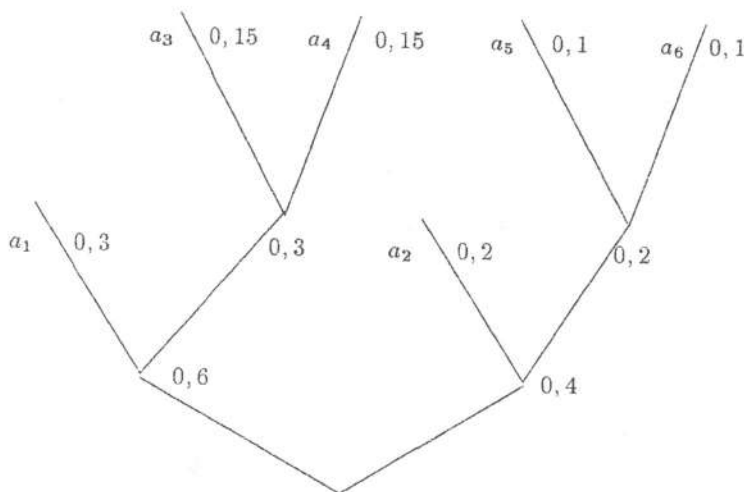


Рис. 6.1. Дерево коду Хафмана

У той же час метод арифметичного кодування є значно складнішим і визначає процедуру побудови одного оптимального коду для всієї послідовності відліків дискретного сигналу. Даний код має наступну властивість:

$$|f(Z)| = \left\lceil \log_2 \prod_{j=0}^{N-1} p(a_{i_j}) \right\rceil = \left\lceil \sum_{j=0}^{N-1} \log_2 p(a_{i_j}) \right\rceil, \quad i = \overline{0, k-1}, \qquad (6.5)$$

де Z – кодована послідовність відліків дискретного сигналу довжини N:

$$Z = \left\{ a_{i_0}, a_{i_1}, ..., a_{i_j}, ..., a_{i_{N-1}} \right\},$$

a_{i_j} – j-й відлік з рівнем квантування a_i $i = \overline{0, k-1}$; $p(a_{i_j})$ – ймовірність того, що j-й відлік кодованої послідовності Z прийме значення a_i з алфавіту $A = \{a_0, a_1, ..., a_i, ..., a_{k-1}\}$.

Тому що опис методу арифметичного кодування потребує багато місця, у даному посібнику він не наводиться. Його можна знайти у рекомендованій літературі [8, 10].

6.2. Програмна реалізація функцій оцінки ефективності статистичних кодерів

У даному підрозділі наведено приклади програмної реалізації процедур (функцій), за допомогою яких можливо визначити ефективність статистичних кодерів для заданих сигналів (рис. 6.2 – 6.11).

```
template <class Arg>
 class gist :  private unary_function<Arg,void>
 {  private:
    vector<Arg> & g; // вектор, до якого записуються значення
                     // лічильників
    public:
    gist(vector<Arg>& gis, int max): g(gis) // конструктор класу
                                             // унарної функції
    { g.clear();
      g.insert(g.begin(),(max+1),0);
      }
  void operator()(const Arg& x) // визначення оператора ()
      { g[x]+=1; // для символа x збільшити лічильник на 1
      }
 };
```

Рис. 6.2. Шаблон класу унарної функції лічильника

```
template <class Arg>
 class frec :  private unary_function<Arg,float>
 { private:
    int cnt; // загальна кількість символів
   public:
    frec(int c): cnt(c) // конструктор класу унарної функції
   {  };
   float operator()(const Arg& x) // визначення оператора ()
  { return x / float(cnt); // отримати частоту
     }
};
```

Рис. 6.3. Шаблон класу унарної функції поділу лічильників на загальну кількість елементів

```
void Gist(TFirst & sign, vector<float> & gst)
{ vector<unsigned int> tmp; // лічильники символів алфавіту
  TFirst::iterator max = max_element(sign.begin(), sign.end());
                          // визначення потужності алфавіту
  gist<unsigned int> L(tmp,(*max)); // створення екземпляру класу
                                     // унарної функції лічильника

  for_each(sign.begin(),sign.end(),L); // для відліків сигналу підрахувати
                                        // кількість кожного символу
                                        // алфавіту
  gst.assign(tmp.size(),0.00); // ініціалізація гістограми
  frec<unsigned int> F(sign.size()); // екземпляр класу унарної функції
  transform(tmp.begin(), tmp.end(),gst.begin(), F); // розрахунок
                                                    //  гістограми
}
```

Рис. 6.4. Визначення функції, яка формує гістограму алфавіту джерела: *sign* – посилання на вектор, що містить відліки сигналу; *gst* – посилання на вектор, до якого з аписуються відліки гістограми

```
template <class Arg>
 class xlog2x :  private unary_function<Arg,void>
 {  private:
    vector<Arg>& logx; // посилання на вектор,
                           // до якого записуються f(x)
    public:
    xlog2x(vector<Arg>& lgx): logx(lgx) // конструктор класу
                                        // унарної функції
     { logx.clear();
     }
  void operator()(const Arg& x) // визначення оператора ()
          { double res;
       res = (-1) * x * Log2(x);
       logx.push_back(res);
      }
};
```

Рис. 6.5. Шаблон класу унарної функції f(x)=x·log(x)

```
double Entropy(vector<float> & gst)
{ vector<double> lg; // вектор для зберігання значень -x*log(x)
 vector<double> tmp;
 tmp.assign(gst.begin(),gst.end());
 xlog2x<double> LOG(lg); // екземпляр функції -x*log(x)
 vector<double>::iterator p0 = remove (tmp.begin(), tmp.end(), 0);
                              // вилучення
 tmp.erase(p0,tmp.end()); // частот значення яких дорівнює 0
 for_each(tmp.begin(),tmp.end(),LOG); // розрахунок
                                     // для кожного f(x) = -x*log(x)
 double sum=0;
 sum = accumulate (lg.begin(), lg.end(), sum); // розрахунок
                                             // Сум(-x*log(x))
 return sum;
};
```

Рис. 6.6. Визначення функції, яка обчислює ентропію

```
template <class Arg>
 class log2x :  private unary_function<Arg,void>
 { private:
  vector<Arg>& logx; // посилання на вектор, до якого записуються f(x)
  public:
log2x(vector<Arg>& lgx): logx(lgx) // конструктор класу унарної функції
    { logx.clear();
    }
 void operator()(const double & x) // визначення оператора ()
      { Arg res;
       if(x!=0)
        { res = Ceil(-Log2(x));
         logx.push_back(res);
        }
       else {logx.push_back(0);}
     };
};
```

Рис. 6.7. Шаблон класу унарної функції f(x) = – log(x)

```
template <class Arg>
 class SetKod :  private unary_function<Arg,void>
 {  private:
    vector<Arg>& vkod; // посилання на вектор, до якого записуються
                       // довжини кодів для кожного відліку
    vector<Arg>& vkod_alf; // посилання на вектор, до якого записані
                           // довжини кодів символів алфавіту
public:
    SetKod(vector<Arg>& vkodx, vector<Arg>& cnt_bit): vkod(vkodx),
vkod_alf(cnt_bit) // конструктор класу унарної функції
     { vkod.clear();
     }
 void operator()(Arg & x) // визначення оператора ()
      { vkod.push_back(vkod_alf[x]);
      };
};
```

Рис. 6.8. Шаблон класу унарної функції заповнення вектора довжин кодів відліків

```
Param LKodH(TFirst & sign, vector<float> & gst, vector<int>& cnt)
{ Param res;
 log2x<int> LOG(cnt); // екземпляр функції -[log(x)]
 for_each(gst.begin(),gst.end(),LOG); // розрахунок
                                       // для кожного f(x) = –[log(x)]
 vector<int> vkodx; // вектор для зберігання довжин кодів алфавіту
 SetKod<int> SetLKod(vkodx,cnt);
 for_each(sign.begin(),sign.end(),SetLKod); // формування послідовності
                                       // довжин кодів відліків сигналу
 int LCod = 0; // об'єм закодованої послідовності у бітах
 LCod = accumulate (vkodx.begin(), vkodx.end(), 0); // підрахунок об'єму
 res.Lcod_sign = LCod;
 vector<float> fcnt; // тимчасовий вектор
 fcnt.assign(cnt.begin(),cnt.end()); // копіювання
 vector<float> res_mult(fcnt.size(),0); // вектор значень p(ai)*log2|f(ai)|
// визначення p(ai)*log2|f(ai)|
 transform(fcnt.begin(),    fcnt.end(),    gst.begin(),    res_mult.begin(),
multiplies<float>());
 float st = 0;
 st = accumulate (res_mult.begin(), res_mult.end(), st); // визначення
                                                          // вартості С
res.C = st;
return res;
};
```

Рис. 6.9. Визначення функції, яка обчислює параметри кода Хаффмана:
sign – посилання на вектор, що містить відліки сигналу;
gst – посилання на вектор, до якого записуються відліки гістограми;
cnt – посилання на вектор, до якого записуються
довжини кодів Хафмана символів алфавіту;
Param – структура, що містить параметри коду

```
class mult_prob: private unary_function<int,void>
{vector<float>& gist; // посилання на гістограму частот
 double & sumlg; // акумулятор
 public:
```

Рис. 6.10. Шаблон класу унарної функції $\sum_j \log_2 p(a_{i_j})$

```
  mult_prob(vector<float>& gstx, double & sm): gist(gstx), sumlg(sm)
                                                        // конструктор
  { };
  void operator()(int x) // визначення оператора ()
  { float t = gist[x]; // визначення частоти
                     // для значення поточного відліку
    double lgx = -Log2(t); // двійковий логарифм
    sumlg = sumlg + lgx; // накопичення
  };
};
```

Рис. 6.10. Шаблон класу унарної функції $\sum_{j} \log_2 p(a_{i_j})$ (закінчення)

```
Param LKodA(TFirst & sign, vector<float> & gst)
{ Param res;
 double smx = 0;
 mult_prob mltp(gst,smx); // екземпляр функції
for_each(sign.begin(),sign.end(),mltp); // визначення об'єму закодованої
                                         // послідовності
 res.Lcod_sign = Ceil(smx);
 res.C = (float)res.Lcod_sign/(float)sign.size(); // вартість кодування
 return res; }
```

Рис. 6.11. Визначення функції, яка обчислює параметри арифметичного коду

7. МЕТОДИ КОРЕКЦІЇ ВІЗУАЛЬНИХ ХАРАКТЕРИСТИК ЦИФРОВИХ ЗОБРАЖЕНЬ

7.1. Загальні відомості про методи корекції візуальних характеристик цифрових зображень

При обробці і візуалізації цифрових зображень часто виникає необхідність додати цифровому зображенню такі якості, завдяки яким його сприйняття людиною було б по можливості комфортним. Часто буває корисним підкреслити, підсилити якісь риси, особливості, нюанси картини, що спостерігається, з метою поліпшення її суб'єктивного сприйняття. Такі методи, як правило, змінюють ті або інші візуальні характеристики цифрових зображень. Особливістю даних методів є те, що для них відсутні строгі математичні критерії оптимальності. Їх заміняють якісні представлення про доцільність тієї або іншої обробки, що спираються на суб'єктивні оцінки результатів. Прикладом реалізації таких методів можуть служити набори фільтрів, які використовуються в таких програмних пакетах як Photoshop, Photo Paint та ін.

У даному розділі розглядається кілька методів корекції візуальних характеристик цифрових зображень, заснованих на поелементній обробці, у яких результат обробки в будь-якій точці кадру залежить тільки від значення вхідного зображення в цієї ж точці. Вочевидь, що перевагами таких процедур є їх гранична простота. Разом з тим, багато які з них призводять до очевидного суб'єктивного поліпшення візуальної якості. Крім цього, дуже часто поелементна обробка застосовується як заключний етап при рішенні більш складної задачі цифрової обробки зображення.

Суть поелементної обробки зображень зводиться до наступного. Нехай, $y_{i,j} = x_{i,j}$ – значення яскравості вихідного і отриманого після обробки зображень відповідно в точці кадру, що має декартові координати i (номер рядка) і j (номер стовпця). Поелементна обробка означає, що існує функціональна однозначна залежність між цими яскравостями

$$y_{i,j} = f_{i,j}(x_{i,j}),$$

що дозволяє по значенню вихідного відліку визначити значення відліку після обробки. У загальному випадку, як це враховано в даному виразі, вигляд або параметри функції, що описує обробку, залежать від поточних координат. При цьому обробка є *неоднорідною.*

Проте в більшості практичних процедур використовується *однорідна* поелементна обробка. В цьому випадку індекси i та j можуть бути відсутніми, а залежність між яскравостями вихідного і обробленого зображень описується функцією

$$y = f(x),$$

однаковою для всіх точок кадру.

Розглянемо математичний опис деяких методів зміни візуальних характеристик цифрових зображень.

Лінійне контрастування зображення. Задача контрастування пов'язана з поліпшенням узгодження динамічного діапазону зображення і пристрою візуалізації. Якщо для цифрового представлення кожного відліку зображення виділяється 1 байт (8 біт), то вхідний або вихідний сигнали можуть приймати одне з 256 значень. Зазвичай як робочий використовується діапазон 0 ... 255; при цьому значення 0 при візуалізації відповідає рівневі чорного, а значення 255 – рівневі білого. Припустимо, що мінімальна і максимальна яскравості вихідного зображення дорівнюють x_{min} і x_{max} відповідно. Якщо ці параметри або один з них істотно відрізняються від граничних значень яскравого діапазону, то візуалізована картина виглядає як ненасичена, незручна, стомлююча при спостереженні. При лінійному контрастуванні використовується лінійне поелементне перетворення вигляду

$$y = a \cdot x + b, \tag{7.1}$$

де a і b визначаються бажаними значеннями мінімальної y_{min} і максимальної y_{max} вихідної яскравості.

Вирішивши систему рівнянь:

$$\begin{cases} y_{min} = a \cdot x_{min} + b; \\ y_{max} = a \cdot x_{max} + b \end{cases}$$

щодо параметрів перетворення a і b, неважко привести (6.1) до вигляду

$$y_{i,j} = \frac{x_{i,j} - x_{min}}{x_{max} - x_{min}}(y_{max} - y_{min}) + y_{min}, \tag{7.2}$$

де $x_{i,j}$ – i,j-відлік вихідного зображення, значення якого лежить у вихідному діапазоні $[x_{max}, x_{min}]$; $y_{i,j}$ – i,j-відлік обробленого зображення, значення якого лежить у заданому діапазоні $[y_{max}, y_{min}]$.

Даний вираз визначає значення відліків цифрового зображення після обробки.

Соляризація зображення. Для даного методу значення відліків обробленого зображення визначаються за допомогою наступного виразу:

$$y_{i,j} = k \cdot x_{i,j} \cdot (x_{max} - x_{i,j}),$$

де x_{max} – максимальне значення відліків вихідного сигналу; k – константа, яка дозволяє керувати динамічним діапазоном перетвореного зображення.

Функція, що описує дане перетворення, є квадратичною параболою, її графік для $k = 1$ наведений на рис. 7.1. При $y_{max} = x_{max}$ динамічні діапазони вихідного та обробленого зображень збігаються, що може бути досягнуте при $k = 4/x_{max}$.

Як випливає з рис. 7.1, зміст соляризації полягає в тому, що ділянки вихідного зображення, які мають рівень білого або близький до нього рівень яскравості, після обробки мають рівень чорного. При цьому зберігають рівень чорного і ділянки, що мають його на вихідному зображенні. Рівень же білого на виході здобувають ділянки, що мають на вході середній рівень яскравості (рівень сірого).

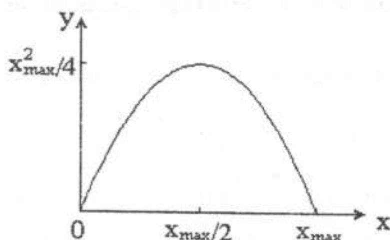


Рис. 7.1. Функція, яка описує соляризацію

Препарування зображення. Препарування – це цілий клас поелементних перетворень зображень. Характеристики процедур препарування, які застосовуються на практиці, приведені на рис. 7.2. Зупинимося на описі деяких з них.

Перетворення з граничною характеристикою (рис. 7.2, а) перетворює напівтонове зображення, що містить усі рівні яскравості, у бінарне, точки якого мають яскравості $y = 0$ або $y = y_{max}$. Така операція, яка називається іноді бінаризацією або бінарним квантуванням, може бути корисною, коли для спостерігача важливі обриси об'єктів, що присутні на зображенні, а деталі, що утримуються у середині об'єктів або у середині фону, не є суттєвими.

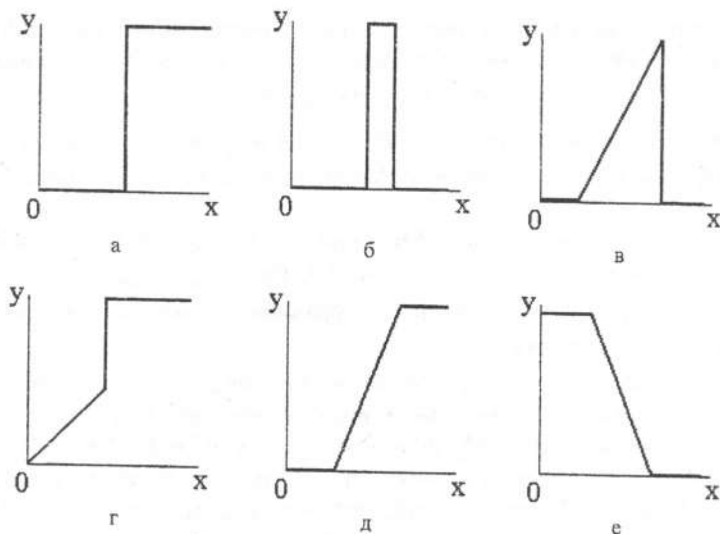


Рис. 7.2. Приклади перетворень,
які використовуються при препаруванні

Математично така процедура (рис. 7.2, а) описується таким виразом:

$$y_{i,j} = \begin{cases} y_{max}, \text{ якщо } x_{i,j} > x_0; \\ y_{min}, \text{ якщо } x_{i,j} \le x_0, \end{cases}$$

де $y_{i,j}$ – відлік обробленого зображення; y_{max} – максимальне значення відліків обробленого зображення; y_{min} – мінімальне значення відліків обробленого зображення; $x_{i,j}$ – відлік вихідного зображення; x_0 – обраний поріг, що визначає перепад положення на рис. 7.2, а.

Для більш складного перетворення, яке задається кривою на рис. 7.2, д, вираз буде мати такий вигляд:

$$y_{i,j} = \begin{cases} y_{min}, \text{ якщо } x_{i,j} \le x_1; \\ (x_{i,j} - x_1)(y_{max} - y_{min})/(x_2 - x_1) + y_{min}, \text{ якщо } x_1 < x_{i,j} < x_2; \\ y_{max}, \text{ якщо } x_{i,j} \ge x_2, \end{cases}$$

де x_1 – перший поріг, що визначає початок лінійного збільшення $y_{i,j}$ (рис. 7.2, д); x_2 – другий поріг, що визначає кінець лінійного збільшення $y_{i,j}$ (рис. 7.2, д).

На рис. 7.3 наведений приклад використання бінарізації зображення.

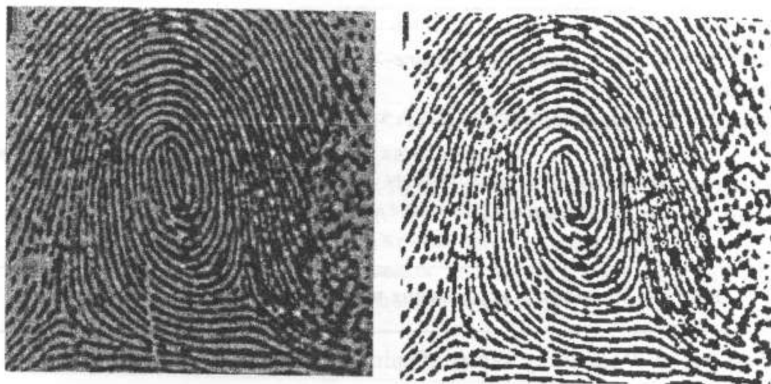


Рис. 7.3. Приклад використання бінарізації зображення

7.2. Програмна реалізація методів корекції візуальних характеристик цифрових зображень

У даному підрозділі на рис. 7.4 – 7.14 наведено приклади програмної реалізації таких процедур (функцій):

– контрастування зображення з використанням STL (рис. 7.6 – 7.8);

– соляризації зображення з використанням STL (рис. 7.9 – 7.11);

– препарування зображення з використанням STL (рис. 7.12 – 7.14);

– додаткових функцій, що призначені для отримання значень відліків зображення та відображення зображення (рис. 7.4 – 7.5).

```
typedef vector<unsigned char> TCol; // визначення типу, призначеного
                                    // для зберігання відліків рядків зображення
typedef vector<TCol> TImg; // визначення типу, призначеного
                           // для зберігання відліків зображення
...
TImg img_x;      // визначення двовимірного масиву
                 // для збереження відліків зображення
TCol tmp_col; // вектор для збереження відліків рядків зображення
unsigned char * pnt_l; // покажчик на рядок відліків зображення
Image1->Picture->LoadFromFile(OpenPictureDialog1->FileName);
                                    // завантаження відліків зображення з файлу
```

Рис. 7.4. Процедура копіювання відліків зображення з компонента TImage у динамічний масив типу TImg

```
unsigned int  H = Image1->Picture->Height; // одержання розмірів
unsigned int  W = Image1->Picture->Width; // зображення
for (int i=0;i<H; i++)
{ pnt_l = (unsigned char *)Image1->Picture->Bitmap->ScanLine[i];
                           // одержання адрес рядків відліків зображення
  tmp_col.assign(pnt_l, (pnt_l+W)); // копіювання відліків і-го рядка
  img_x.push_back(tmp_col);         // у масив img_x
}
```

Рис. 7.4. Процедура копіювання відліків зображення (закінчення)

```
TImg img_y; // масив, у якому зберігаються відліки зображення
for(int i=0; i<H; i++)
copy(img_y[i].begin(),img_y[i].end(),
     (unsigned char *)Image1->Picture->Bitmap->ScanLine[i]);
                        // копіювання відліків і-го рядка масиву img_y
                        // у і-й рядок компонентові Image1 (тип TImage)
Image1->Refresh(); // відновлення зображення
```

Рис. 7.5. Процедура відображення зображення, відліки якого зберігаються в динамічному масиві img_y

```
class fcontr : private unary_function<unsigned char,void>
{ private:
  TCol & col_o; // посилання на вектор, призначений для збереження
                //відліків рядка обробленого зображення
  unsigned char yma, ymi, xma, xmi; // змінні, призначені
// для збереження відповідно максимального і мінімального значень
// відліків обробленого зображення (yma, ymi), а також максимального
// і мінімального значень відліків вихідного зображення (xma, xmi)
 public:
fcontr(TCol& c_im, unsigned char ymax, unsigned char ymin,
       unsigned char xmax,
       unsigned char xmin): col_o(c_im) // конструктор унарної функції
{ col_o.clear();
  yma = ymax;
  ymi = ymin;
```

Рис. 7.6. Визначення класу унарної функції, яка розраховує значення поточного відліку при контрастуванні

```
  xma = xmax;
  xmi = xmin;
 }
void operator()(const unsigned char & x) // визначення оператора ()
{ unsigned char tmp;
  tmp = ceil((x - xmi)*((float)(yma - ymi)/(float)(xma - xmi)) + ymi);
                        // розрахунок значення поточного відліку
                        // рядка зображення, що обробляється
  col_o.push_back(tmp); // додавання поточного відліку в кінець
                        // рядка, що обробляється
};
};
```

Рис. 7.6. Визначення класу унарної функції, яка розраховує значення поточного відліку при контрастуванні (закінчення)

```
void Contrast(TImg & inp, TImg & out, unsigned char mxy,
              unsigned char miy)
{ TCol c_tmp; // вектор для збереження відліків
              // рядка зображення, що обробляється
  TCol::iterator max_c; // ітератор (покажчик) на максимальне значення
                        // відліку в рядку, що обробляється
  TCol::iterator min_c; // ітератор (покажчик) на мінімальне значення
                        // відліку в рядку, що обробляється
  unsigned char maxx, minx, maxy, miny;
// нижче проводиться перевірка на правильність завдання динамічного
// діапазону: максимальне значення з { mxy, miy } привласнюється
// maxy, мінімальне значення – miny
  if(mxy>miy)
   {maxy = mxy;
    miny = miy;
   }
  else
   {maxy = miy;
    miny = mxy;
   };
```

Рис. 7.7. Визначення функції, що виконує контрастування зображення, відліки якого зберігаються в масиві inp, результат контрастування записується в масив out, mxy і miy задають відповідно максимальне і мінімальне значення відліків обробленого зображення

```
out.clear(); // очищення масиву, призначеного для збереження відліків
// обробленого зображення, нижче виконується пошук мінімального і
// максимального значення відліків у вихідному зображенні
 for (int i=0;i<H; i++) // переглядаємо кожен рядок
                          // вихідного зображення
   {max_c = max_element(inp[i].begin(), inp[i].end());
                                         // знаходимо максимальне
    min_c = min_element(inp[i].begin(), inp[i].end());
                               // і мінімальне значення в поточному рядку
    if((*max_c)>maxx) maxx = *max_c; // запам'ятовуємо нове
                                          // максимальне значення
    if((*min_c)<minx) minx = *min_c; // запам'ятовуємо нове
                                        // мінімальне значення
 };
 fcontr fun(c_tmp,maxy, miny, maxx, minx); // створення екземпляра
     // унарної функції класу fcontr, виконуємо контрастування
     // вихідного зображення, результат записуємо в масив out
 for (int i=0;i<H; i++)
    {for_each(inp[i].begin(), inp[i].end(),fun);
// застосовуємо унарну функцію fun класу fcontr
// для кожного відліку і-го рядка вихідного зображення,
// результат обробки рядка записується у вектор c_tmp
     out.push_back(c_tmp); // додаємо в оброблене зображення
                              // новий рядок
     c_tmp.clear(); // очищаємо вектор для запису нових значень
    }
}
```

Рис. 7.7. Визначення функції, що виконує контрастування зображення (закінчення)

```
void __fastcall TForm1:: BitBtn3Click(TObject *Sender)
{Contrast(img_x, img_y, UpDown4->Position, UpDown3->Position);
// виконання контрастування
// відображення обробленого зображення
 for(int i=0; i<H; i++)
 copy(img_y[i].begin(),img_y[i].end(),
      (unsigned char *)Image1->Picture->Bitmap->ScanLine[i]);
 Image1->Refresh();
}
```

Рис. 7.8. Оброблювач для кнопки – запуск процедури контрастування

```
class fsolar : private unary_function<unsigned char,void>
{ private:
 TCol & col_o; // посилання на вектор, призначений для збереження
               // відліків рядка обробленого зображення
 unsigned char xma; // змінна, призначена для збереження
                    // максимального значення відліків
                    // вихідного зображення
 float kx; // коефіцієнт масштабування
 public:
 fsolar(TCol& c_im, unsigned char xmax, float k): col_o(c_im)
                                        // конструктор унарної функції
 { col_o.clear();
  xma = xmax;
  kx = k;
 }
 void operator()(const unsigned char & x) // визначення оператора (),
                                  // x –поточний відлік, що обробляється
 { unsigned char tmp;
  tmp = ceil(kx*x*(xma-x)); //розрахунок нового значення поточного відліку
   col_o.push_back(tmp);   // запис отриманого значення в кінець
                           // рядка, що обробляється
 };
};
```

Рис. 7.9. Визначення класу унарної функції, що розраховує значення поточного відліку при соляризації

```
void Solar(TImg & inp, TImg & out, float kxx)
{ TCol c_tmp;
 TCol::iterator max_c;
 unsigned char maxx;
 out.clear();
// визначення максимального значення відліків вихідного зображення
 for (int i=0;i<H; i++)
   { max_c = max_element(inp[i].begin(), inp[i].end());
```

Рис. 7.10. Визначення функції, що виконує соляризацію зображення, відліки якого зберігаються в масиві inp, результат соляризації записується у масив out, kxx задає коефіцієнт масштабування динамічного діапазону для обробленого зображення

```
    if((*max_c)>maxx) maxx = *max_c;
  }
 fsolar fun(c_tmp,maxx,kxx); // створення екземпляра унарної функції
                             // класу fsolar
// виконуємо порядково соляризацію вихідного зображення,
// результат записуємо в масив out
 for (int i=0;i<H; i++)
   { for_each(inp[i].begin(), inp[i].end(),fun);
     out.push_back(c_tmp);
     c_tmp.clear();
   }
}
```

Рис. 7.10. Визначення функції, що виконує соляризацію зображення (закінчення)

```
void __fastcall TForm1::BitBtn2Click(TObject *Sender)
{ float ks = 1/(float)UpDown1->Position; // розрахунок значення
                       // коефіцієнта масштабування динамічного діапазону
  Solar(img_x, img_y, ks);
  for(int i=0; i<H; i++)
    copy(img_y[i].begin(),img_y[i].end(),
    (unsigned char *)Image1->Picture->Bitmap->ScanLine[i]);
  Image1->Refresh();
}
```

Рис. 7.11. Оброблювач для кнопки «запуск процедури соляризації»

```
class fprep : private unary_function<unsigned char,void>
{ private:
  TCol & col_o;
  unsigned char mg; // перемінна, призначена для збереження
                    // значення порога бінаризації
  public:
  fprep(TCol& c_im, unsigned char mega): col_o(c_im)
                                    // конструктор унарної функції
{col_o.clear();
mg = mega;
 }
```

Рис. 7.12. Визначення класу унарної функції, що розраховує значення поточного відліку при препаруванні зображення

```
void operator()(const unsigned char & x) // визначення оператора (),
                                         // x – поточний оброблюваний відлік
  { unsigned char tmp;
    tmp = x>mg?255:0;
    col_o.push_back(tmp);
  };
};
```

Рис. 7.12. Визначення класу унарної функції, що розраховує значення поточного відліку при препаруванні зображення (закінчення)

```
void Prepar(TImg & inp, TImg & out, unsigned char mgx)
{ TCol c_tmp;
  out.clear();
  fprep fun(c_tmp,mgx);
  for (int i=0;i<H; i++)
     {for_each(inp[i].begin(), inp[i].end(),fun);
      out.push_back(c_tmp);
      c_tmp.clear();
     }
}
```

Рис. 7.13. Визначення функції, що виконує препарування (бінаризацію) зображення, відліки якого зберігаються в масиві inp, результат соляризації записується в масив out, mgx задає поріг бінаризації

```
void __fastcall TForm1::BitBtn4Click(TObject *Sender)
{ Prepar(img_x, img_y, UpDown2->Position);
  for(int i=0; i<H; i++)
     copy(img_y[i].begin(),img_y[i].end(),
     (unsigned char *)Image1->Picture->Bitmap->ScanLine[i]);
  Image1->Refresh();
}
```

Рис. 7.14. Оброблювач для кнопки «запуск процедури препарування»

8. МЕДІАННА ФІЛЬТРАЦІЯ ЦИФРОВИХ ЗОБРАЖЕНЬ

8.1. Загальні відомості про медіанну фільтрацію цифрових зображень

Зазвичай зображення, сформовані різними інформаційними системами, спотворюються дією перешкод. Це затрудняє як їх візуальний аналіз людиною-оператором, так і автоматичну обробку в ЕОМ. При рішенні деяких задач обробки зображень в ролі перешкод можуть виступати ті або інші компоненти самого зображення. Наприклад, при аналізі космічного знімка земної поверхні може стояти завдання визначення меж між її окремими ділянками – лісом і полем, водою і сушею тощо. Виходячи з цього окремі деталі зображення всередині областей, що розділяються, є перешкодою.

Ослаблення дії перешкод досягається фільтрацією. При фільтрації яскравість (сигнал) кожної точки початкового зображення, спотвореного перешкодою, замінюється деяким іншим значенням яскравості, яке визначається в найменшій мірі спотвореним перешкодою. Що може послужити основою для таких рішень? Зображення є двовимірною функцією просторових координат, яка змінюється за цими координатами повільніше (іноді значно повільніше), ніж перешкода, що також є двовимірною функцією. Це дозволяє при оцінці корисного сигналу в кожній точці кадру прийняти до уваги деяку множину сусідніх точок, скориставшись певною схожістю сигналу в цих крапках. У інших випадках, навпаки, ознакою корисного сигналу є різкі перепади яскравості. Проте, як правило, частота цих перепадів відносно невелика, так що на значних проміжках між ними сигнал або є постійним, або змінюється поволі. І в цьому випадку властивості сигналу виявляються при спостереженні його не тільки в локальній точці, але і при аналізі її околу. Відмітимо, що поняття околу є достатньо умовним. Воно може бути утворено лише найближчими по кадру сусідами, але можуть бути околи, що містять достатньо багато і досить сильно видалені точки кадру. У цьому останньому випадку, звичайно, ступінь впливу далеких і близьких точок на рішення, що приймаються фільтром в даній точці кадру, буде абсолютно різним.

Таким чином, ідеологія фільтрації ґрунтується на раціональному використанні даних як з робочої точки, так і з її околиці. У цьому виявляється істотна відмінність фільтрації від розглянутих вище поелементних процедур: фільтрація не може бути поелементною процедурою обробки зображень.

Завдання полягає в тому, щоб знайти таку раціональну обчислювальну процедуру, яка дозволяла б досягати якнайкращих результатів. Загальноприйняте при рішенні цієї задачі спиратися на використання імо-

вірнісних моделей зображення і перешкоди, а також на застосування статистичних критеріїв оптимальності. Причини цього зрозумілі – це випадковий характер як інформаційного сигналу, так і перешкоди і це прагнення отримати мінімальну *в середньому* відмінність результату обробки від ідеального сигналу. Різноманіття методів і алгоритмів пов'язане з великою різноманітністю сюжетів, які доводиться описувати різними математичними моделями. Крім того, застосовуються різні критерії оптимальності, що також веде до різноманітності методів фільтрації. Нарешті, навіть при збігу моделей і критеріїв дуже часто із-за математичних труднощів не вдається знайти оптимальну процедуру. Складність знаходження точних рішень породжує різні варіанти наближених методів і процедур.

Надалі дотримуватимемося прийнятою при цифровій обробці зображень декартової системи координат з початком в лівому верхньому кутку кадру і з позитивними направленнями з цієї точки вниз та праворуч.

На рис. 8.1 показані приклади околів різних типів, зображені у вигляді сукупностей точок.

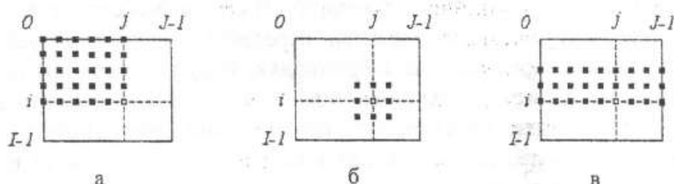


Рис. 8.1. Приклади околів різних видів

Центром околів, робочою точкою, в якій здійснюється обробка, є точка з координатами i, j (на рис. 8.1 не зачорнена). Залежно від типу околу розрізняють *каузальну, некаузальну і напівкаузальну* фільтрацію зображень.

Поняття каузальної (причинно-наслідкової залежності) пов'язують із співвідношенням координат поточної точки i, j та точок, що входять в окіл. Якщо *обидві координати* (номер рядка і номер стовпця) всіх точок околу не перевищують відповідних координат поточної точки, то окіл і обробка, що її використовує, називаються каузальними. Приклад такого околу наведений на рис. 8.1, а. Деякі точки околу, наведеного на рис. 8.1, б, задовольняють принципу каузальності. Разом з тим, тут є і такі точки, *обидві координати* яких перевищують відповідні координати робочої точки. Фільтрація, що спирається на використання околів з поєднанням таких властивостей, називається некаузальною.

Околу, показаному на рис. 8.1, в, відповідає напівказуальна фільтрація. Одна з координат *всіх* точок околу (в даному прикладі номер рядка)

не перевищує відповідної координати робочої точки. Друга ж координата (в прикладі номер стовпця) у деяких точок також не перевищує відповідної координати робочої точки. Проте серед точок околу є і такі, у яких ця друга координата перевищує відповідну координату робочої точки.

Всі лінійні алгоритми фільтрації призводять до згладжування різких перепадів яскравості зображень, що пройшли обробку. Цей недолік є особливо істотним, якщо споживачем інформації є людина, і принципово не може бути виключений в рамках лінійної обробки. Річ у тому, що лінійні процедури є оптимальними при гаусівському розподілі сигналів перешкод. Реальні зображення, строго кажучи, не підкоряються даному розподілу. Причому, одна з основних причин цього полягає в наявності у зображень різноманітних меж, перепадів яскравості, переходів від однієї текстури до іншої тощо. Тому після локального гаусівського опису в межах обмежених ділянок, багато реальних зображень в зв'язку з цим погано представляються як глобальні гаусівські об'єкти. Саме це і служить причиною поганої передачі меж при лінійній фільтрації.

Друга особливість лінійної фільтрації – її оптимальність, як тільки що згадувалося, при гаусівському характері перешкод. Зазвичай цій умові відповідають шумові перешкоди на зображеннях, тому при їх придушенні лінійні алгоритми мають високі показники. Проте, часто доводиться мати справу із зображеннями, спотвореними перешкодами інших типів.

Одним з широко поширених видів перешкод, що діють на цифрові зображення, є імпульсні перешкоди. При дії імпульсної перешкоди на зображенні спостерігаються білі або (і) чорні точки, хаотично розкидані по кадру (рис. 8.2). Застосування лінійної фільтрації в цьому випадку неефективно – кожний з вхідних імпульсів (по суті – дельта-функція) дає відгук у вигляді імпульсної характеристики фільтру, а їх сукупність сприяє розповсюдженню перешкоди на всю площу кадру.

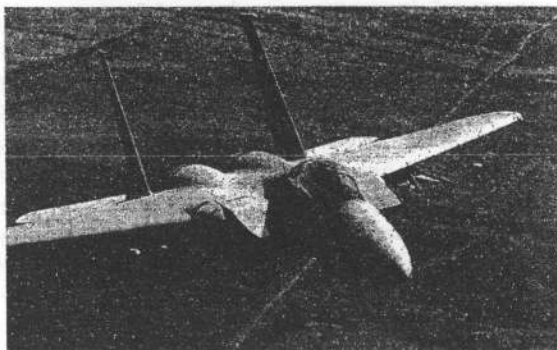


Рис. 8.2. Приклад дії імпульсної перешкоди на цифрове зображення

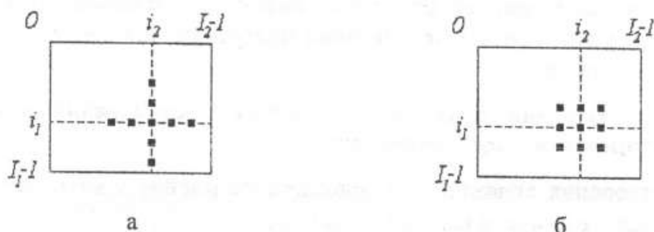
Вдалим рішенням вищеперерахованих проблем є застосування *медіанної фільтрації,* запропонованої Дж. Тьюки у 1971 році для аналізу економічних процесів.

При застосуванні медіанного фільтру (МФ) відбувається послідовна обробка кожної точки кадру, внаслідок чого утворюється послідовність оцінок. У ідейному відношенні обробка в різних точках незалежна (цим МФ схожий на масковий фільтр), але в цілях її прискорення доцільно алгоритмічно на кожному кроці використовувати раніше виконані обчислення.

При медіанній фільтрації використовується двовимірне вікно (апертура фільтру), що звичайно має центральну симетрію, при цьому його центр розташовується в поточній точці фільтрації. На рис. 8.3 показані два приклади найбільш часто вживаємих варіантів вікон у вигляді хреста та у вигляді квадрата. Розміри апертури належать до параметрів, що оптимізуються в процесі аналізу ефективності алгоритму. Відліки зображення, що опинилися в межах вікна, утворюють *робочу вибірку* поточного кроку.

Двовимірний характер вікна дозволяє виконувати, по суті, двовимірну фільтрацію, оскільки для утворення оцінки використовуються дані як з поточних рядка і стовпця, так і з сусідніх.

Позначимо робочу вибірку у вигляді одновимірного масиву $Y=\{y_1, y_2, ..., y_n\}$; число його елементів дорівнює розміру вікна, а їх розташування довільно. Звичайно застосовують вікна з непарним числом точок (це автоматично забезпечується при центральній симетрії апертури і при входженні самої центральної точки в її склад). Якщо впорядкувати послідовність $\{y_i, i=\overline{1,n}\}$ за збільшенням, то її *медіаною* буде той елемент вибірки, який займає центральне положення в цій впорядкованій послідовності. Одержане таким чином число і є продуктом фільтрації для поточної точки кадру.



а б

Рис. 8.3. Приклади вікон при медіанній фільтрації

Зрозуміло, що результат такої обробки насправді не залежить від того, в якій послідовності представлені елементи зображення в робочій вибірці Y. Формальне позначення описаної процедури має вигляд

$$x^* = \mathrm{med}(y_1, y_2, ..., y_n). \quad (8.1)$$

Розглянемо приклад. Припустимо, що вибірка має такий вигляд: $Y = \{136, 110, 99, 45, 250, 55, 158, 104, 75\}$, а елемент 250, розташований в її центрі, відповідає поточній точці фільтрації (i_1, i_2) (рис. 8.3). Велике значення яскравості в цій точці кадру може бути результатом дії імпульсної (точкової) перешкоди. Впорядкована за збільшенням вибірка має при цьому вигляд {45, 55, 75, 99, 104, 110, 136, 158, 250}, отже, відповідно до процедури (8.1), одержуємо $x^* = \mathrm{med}(y_1, y_2, ..., y_9) = 104$. Бачимо, що вплив сусідів на результат фільтрації в поточній точці призвів до ігнорування імпульсного викиду яскравості, що слід розглядати як ефект фільтрації. Якщо імпульсна перешкода не є точковою, а покриває деяку локальну область, то вона також може бути придушена. Це відбудеться, якщо розмір цієї локальної області буде менший, ніж половина розміру апертури МФ. Тому для придушення імпульсних перешкод, що вражають локальні ділянки зображення, слід збільшувати розміри апертури МФ. З виразу (8.1) витікає, що дія медіанного фільтру полягає в ігноруванні екстремальних значень вхідної вибірки – як позитивних, так і негативних викидів. Такий принцип придушення перешкоди може бути застосований і для ослаблення шуму на зображенні. Проте дослідження придушення шуму за допомогою медіанної фільтрації показує, що її ефективність при рішенні цієї задачі нижче, ніж у лінійної фільтрації.

8.2. Програмна реалізація методу медіанної фільтрації зображень

У даному підрозділі наведено приклади програмної реалізації процедур (функцій) медіанної фільтрації цифрових зображень. Наведені процедури (функції) реалізовані на мові програмування C++ з використанням бібліотеки STL.

8.2.1. Створення динамічного масиву і копіювання в нього відліків зображення, що обробляється

Для створення динамічного двовимірного масиву з використанням STL необхідно визначити наступні типи даних:

typedef vector<unsigned char> *TCol;* – вектор, призначений для зберігання відліків рядків зображення;

typedef vector<*TCol****>*** *TImg;* – вектор, що містить елементи типу *TCol* (рядки зображення) або іншими словами – двовимірний масив відліків зображення.

Процедура копіювання відліків зображення з компоненту TImage в динамічний масив типу TImg наведена на рис. 8.4.

```
TImg img_x; // визначення двовимірного масиву
            // для зберігання відліків зображення
TCol tmp_col; // вектор для зберігання відліків рядків зображення
unsigned char * pnt_l; // покажчик на рядок відліків зображення
Image1->Picture->LoadFromFile(OpenPictureDialog1->FileName);
                        // завантаження відліків зображення з файлу
unsigned int  H = Image1->Picture->Height; // отримання розмірів
unsigned int  W = Image1->Picture->Width; // зображення
for (int i=0;i<H; i++)
{ pnt_l = (unsigned char *)Image1->Picture->Bitmap->ScanLine[i];
                        // отримання адресів рядків відліків зображення
tmp_col.assign(pnt_l, (pnt_l+W)); // копіювання відліків і-го рядка
img_x.push_back(tmp_col);       // у масив img_x
}
```

Рис. 8.4. Процедура копіювання відліків

8.2.2. Процедура відображення зображення, відліки якого зберігаються в динамічному масиві img_y

Дана процедура наведена на рис. 8.5.

```
TImg img_y; // масив, в якому зберігаються відліки зображення
for(int i=0; i<H; i++)
copy(img_y[i].begin(),img_y[i].end(),
     (unsigned char *)Image1->Picture->Bitmap->ScanLine[i]);
        // копіювання відліків і-го рядка масиву img_y у і-й рядок
       // компоненту Image1 (тип TImage)
Image1->Refresh(); // оновлення зображення
```

Рис. 8.5. Процедура відображення зображення

8.2.3. Процедура обчислення медіани фільтру з використанням STL

Функція, яка отримує значення медіани у апертурі фільтру, має такі вхідні параметри:

_img – посилання на масив відліків початкового зображення;

_sz – розмір апертури фільтру (завжди непарне);

_type – визначає вид апертури, вибраний користувачем (у прикладі може приймати два значення:

MRect – назва апертури, що наведена на рис. 8.3, б;

MPlus – назва апертури, що наведена на рис. 8.3, а;

_x і *_y* – координати відліку, що обробляється.

Текст процедури наведений на рис. 8.6.

```
unsigned char GetMediana(TImg & _img, unsigned char _sz, TTypeApert
_type, unsigned int _x, unsigned int _y)
{int dxy = _sz>>1; // кількість відліків праворуч, ліворуч, вище і нижче
          // від відліку, що обробляється і які потрапляють в апертуру
          // розрахунок меж апертури з урахуванням меж зображення
 int y_start = ((signed)_y-dxy)<0?0:(signed)_y-dxy;
                    // верхня межа поточного положення апертури
 int y_end = ((signed)_y+dxy)<_img.size()?(_y+dxy+1):_img.size();
                    // нижня межа поточного положення апертури
 int x_start = ((signed)_x-dxy)>0?(signed)_x-dxy:0;
                    // права межа поточного положення апертури
int x_end = ((signed)_x+dxy)<_img[0].size()?(_x+dxy+1):_img[0].size();
                 // нижня межа поточного положення апертури
vector<unsigned char> wind; // масив для зберігання відліків,
                              // що потрапили у вікно апертури
vector<unsigned char>::iterator it_med=wind.begin();
                              // ітератор на медіану в апертурі фільтру
 switch(_type) // вибір виду апертури фільтру
   {case MRect: // вид апертури, представлений на рисл. 7.3, б
          for (int i=y_start;i<y_end;i++)
           for (int j=x_start;j<x_end;j++)
              wind.push_back(_img[i][j]); // заповнення масиву wind
               // значеннями відліків, які потрапили у вікно апертури
          break;
   case  MPlus: // вид апертури, представлений на рис. 7.3, а
          for (int i=y_start;i<y_end;i++)
            wind.push_back(_img[i][_x]);
          for (int j=x_start;j<x_end;j++)
          // заповнення масиву wind значеннями відліків,
          // які потрапили у вікно апертури
```

Рис. 8.6. Процедура обчислення медіани фільтру

```
            wind.push_back(_img[_y][j]);
        break;
  };
 sort(wind.begin(),wind.end()); // сортування елементів апертури
 it_med = wind.begin() + (wind.size()>>1);
        // встановити ітератор на середину відсортованої апертури
return *it_med; // одержати і повернути з функції значення медіани
}
```

Рис. 8.6. Процедура обчислення медіани фільтру (закінчення)

8.2.4. Процедура медіанної фільтрації зображення

Процедура наведена на рис. 8.7.

```
// Визначення обробника події натиснення кнопки «Фільтрація»
void __fastcall TForm1::BitBtn2Click(TObject *Sender)
{unsigned int sz_mf = StrToInt(Edit1->Text);
      // рахування розміру апертури, заданого користувачем
// цикл обробки кожного відліку початкового зображення img_x,
// H – висота зображення, W – ширина зображення,
// img_y[i][j] – масив, призначений для зберігання відліків
// обробленого зображення, apertura – змінна,
// в яку записується вид апертури, вибраний користувачем
 for (int i=0; i<H; i++)
  for (int j=0; j<W; j++)
   img_y[i][j] = GetMediana(img_x, sz_mf, apertura, j, i);
              // отримання медіани відповідно до виразу (7.1)
// відображення обробленого зображення
 for(int i=0; i<H; i++)
    смітmю(img_y[i].begin(),img_y[i].end(),(unsigned char *)
  Image1->Picture->Bitmap->ScanLine[i]);
 Image1->Refresh();
}
```

Рис. 8.7. Процедура медіанної фільтрації зображення

СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

1. Ахмед Н., Рао К.Р. Ортогональные преобразования при обработке цифровых сигналов: Пер. с англ. – М.: Связь, 1980. – 248 с.
2. Бондарев В.Н., Трестер Г., Чернега В.С. Цифровая обработка сигналов: методы и средства. Уч. пос. 2-е изд. – Х.: Конус, 2001. – 398 с.
3. Воробъев В.И., Грибунин В.Г. Теория и практика вейвлет-преобразования. – С.-Пб.: ВУС, 1999. – 204 с.
4. Гольденберг Л.М., Матюшкин Б.Д., Поляк М.Н. Цифровая обработка сигналов: Справочник – М.: Радио и Связь, 1985. – 312 с.
5. Горбунов Б.А., Дементьев В.Н., Пяткин В.П. Распознавание изображений в дистанционном зондировании // Автоматизированная обработка изображений природных комплексов Сибири. – Новосибирск: Наука, 1988. – 138 с.
6. Грузман И.С., Киричук В.С., Косых В.П., Перетягин Г.И., Спектор А.А. Цифровая обработка изображений в информационных системах: Уч. пос. – Новосибисрк: НГТУ, 2000. – 168 с.
7. Зубарев Ю. В,. Дворкович В. П. Цифровая обработка телевизионных и компьютерных изображений. – М.: МЦНТИ, 1997. – 212 с.
8. Методы передачи изображений. Сокращение избыточности / Под ред. У.К. Прэтта. – М.: Радио и связь, 1983. – 264 с.
9. Петухов А.В. Введение в теорию базисов всплесков. – С.-Пб.: СПбГТУ, 1999. – 132 с.
10. Потапов В.Н. Теория информации. Кодирование дискретных вероятностных источников. – Новосибирск: НГУ, 1999. – 70 с.
11. Прата С. Язык программирования С++. Лекции и упражнения / Пер. с англ. – К.: ДиаСофт, 2001. – 656 с.
12. Применение цифровой обработки сигналов / Под ред. Э. Оппенгейма. – М.: Мир, 1980. – 550 с.
13. Прэтт У. Цифровая обработка изображений. Кн.2. – М.: Мир, 1982.– 318 с.
14. Стрюк А.Ю., Бохан К.А. Цветовые модели в системах сжатия видеоданных // Радиоэлектроника и информатика. – 2002. – № 1 (18).
15. Уэлстид С. Фракталы и вейвлеты для сжатия изображений в действии. – М.: Триумф, 2003. – 320 с.
16. Daubechies I. Ten Lectures on Wavelets. – Philadelphia: SIAM, 1992. – 124 c.
17. Mallat S.G., Multiresolution Approximations and Wavelet of orthonormal Bases of L2(R) // Transactions of the American Mathematical Society. – 1989. – V. 315, № 1. – P. 69 – 87.

ЗМІСТ

Вступ 3
1. Основи використання стандартної бібліотеки шаблонів........................ 4
   1.1. Загальні відомості про бібліотеку STL... 4
   1.2. Використання контейнерів. Контейнер vector................................ 6
   1.3. Використання ітераторів та функцій контейнерів............................ 7
   1.4. Загальні функції бібліотеки STL. Функтори та алгоритми..... 12
2. Фізичні характеристики цифрових сигналів 16
   2.1. Визначення фізичних характеристик цифрових сигналів....... 16
   2.2. Програмна реалізація функцій визначення фізичних характеристик цифрових сигналів... 21
3. Дискретні ортогональні перетворення цифрових сигналів 23
   3.1. Загальні відомості про ортогональні перетворення.................. 23
   3.2. Матрична форма представлення дискретних ортогональних перетворень.. 26
   3.3. Швидкі алгоритми дискретних ортогональних перетворень.... 27
   3.4. Програмна реалізація ортогональних перетворень дискретних сигналів.. 31
4. Дискретні вейвлет-перетворення цифрових сигналів 35
   4.1. Загальні відомості про дискретні вейвлет-перетворення......... 35
   4.2. Програмна реалізація дискретного вейвлет-перетворення у базисі Хаара 42
5. Дискретні ймовірністні джерела.. 46
   5.1. Загальні відомості про дискретні ймовірності джерела........... 46
   5.2. Програмна реалізація процедур обчислення характеристик ймовірнісних джерел 49
6. Статистичне кодування (стиснення) дискретних сигналів.................. 52
   6.1. Загальні теоретичні відомості про статистичні кодери........... 52
   6.2. Програмна реалізація функцій оцінки ефективності статистичних кодерів.. 57
7. Методи корекції візуальних характеристик цифрових зображень 63
   7.1. Загальні відомості про методи корекції візуальних характеристик цифрових зображень.. 63
   7.2. Програмна реалізація методів корекції візуальних характеристик цифрових зображень 67

8. Медіанна фільтрація цифрових зображень 74
8.1. Загальні відомості про медіанну фільтрацію цифрових зображень.. 74
8.2. Програмна реалізація методу медіанної фільтрації зображення . 78
Список рекомендованої літератури 82

Навчальне видання

Бохан Костянтин Олександрович
Кучук Георгій Анатолійович

ЦИФРОВА ОБРОБКА СИГНАЛІВ

Програмна реалізація з використанням бібліотеки STL

Навчальний посібник
для студентів комп'ютерних спеціальностей
вищих навчальних закладів

Відповідальний за випуск К.О. Бохан

Оригінал-макет виготовлений на кафедрі комп'ютерних систем і мереж
Національного аерокосмічного університету ім. М.Є. Жуковського
"Харківський авіаційний інститут"

Дод. план 2008.
Підписано до друку 21.02.2008
Редактор І.А. Лебедеєа Коректор В.В. Кірвас
Формат 60×84 1/16. Бум. офс. № 2. Друк різограф
Ум. друк. арк. 10,25. Вид. арк. 10,75. Наклад. 150 прим. Зам. 221-08. Ціна договірна

Національний аерокосмічний університет ім. М.Є. Жуковського
"Харківський авіаційний інститут"
Україна, 61070, Харків–70, вул. Чкалова, 17
e-mail: lesch@xai.edu.ua

Віддруковано у друкарні ФОП «АЗАМАЄВА В.П.»
61111, Харків – 111, вул. Познанська, 6, тел. 362-01-52
Свідоцтво про внесення суб'єкта видавничої справи до державного реєстру видавців, виготівників і розповсюджувачів видавничої продукції ХК № 134 від 23.02.05 р.