МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний авіаційний університет

Гаєв Є.О., Азарсков В.М.

Сучасне програмування

Навчальний посібник з дисциплін
„Програмування", „Алгоритмічні мови та програмування"

Частина 2

Складні типи даних та алгоритми, інтелектуальні програми

Київ 2016

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний авіаційний університет

Гаєв Є.О., Азарсков В.М.

Сучасне програмування

Навчальний посібник з дисциплін
„Програмування", „Алгоритмічні мови та програмування"

***Частина* 2 (*модулі 3, 4 і 5*)**

"Складні типи даних та алгоритми, інтелектуальні програми"

Київ 2016

УДК 004.94 (075.8)
ББК З 973.2-018.1я7
Г 134

Рецензенти: д.ф-м.н. проф. Глибовець М.М., декан факультету інформатики Нац. університету “Києво-Могилянська академія”; д.т.н. доцент Апостолюк В.О.

Затверджено на засіданні кафедри систем управління літальних апаратів Національного Авіаційного Університету *6 червня* 2016 року, протокол № 23.

Гаєв Є.О., Азарсков В.М

Г 134 ***Сучасне програмування.*** Навчальний посібник з дисциплін „Програмування”, „Алгоритмічні мови та програмування”. Частина 2 (модулі 3, 4 і 5) "Складні типи даних та алгоритми, інтелектуальні програми" – К.: НАУ, 2016. – 196 с.

ISBN 978-617-696-284-7

Ця друга частина підручника з програмування призначена для студентів першого року навчання та веде студентів далі, до оволодіння сучасним структурним програмуванням використовучи програмне і математичне середовище MATLAB. Висвітлено діалогові та GUI-програми, нові та складні типи даних (комірки, структури, зображення, звук, хендли тощо) задля реалізації складних алгоритмів і програм (анімації, перепорядкування числових та текстових масивів, рекурсії, фракталів, вейвлет-функції Дебєши тощо). Створені програми аналізуються “експериментально” та теоретично щодо їх поведінки із зростанням складності завдання. Студент отримує теоретичний і практичний багаж, достатній для власної складної і красивої курсової роботи (модуль 5). Контрольні питання і лабораторні завдання закріплюють практичні навички студентів.

Для студентів молодших курсів, що навчаються за спеціальністю „Системна інженерія” та іншими спеціальностями інформаційних технологій.

УДК 004.94 (075.8)
ББК З 973.2-018.1я7

ISBN 978-617-696-284-7

Короткий зміст

З М І С Т детальний

Передмова

Модуль 3

Другий рівень майстерності програміста

План третьої (заключної) частини (модулі 6, 7 і 8)

Передмова

Це є продовження нашого навчального посібника “Сучасне програмування”, призначена для другого семестру навчання студентів Національного авіаційного університету. Вона орієнтована передусім на тих студентів, що вчаться за спеціальністю “Системна інженерія”. У першій частині ми починали з елементів математики у MATLAB. Бо хіба можна уявити ***програмування*** – без математики, теоретичної основи комп’ютерних наук? З іншого погляду, як же сучасна математика та сучасне програмування – без MATLAB?

Там ми трохи випередили час та розповіли вам про застосування MATLAB до деяких задач математики, які для вас ще попереду в курсі вищої математики цього семестру. Це є такі питання:

- ♪ Диференціальне та інтегральні числення;
- ♪ Числові та функціональні ряди, зокрема ряди Тейлора і Фур’є.

У цій частині навчального посібника зосередимо більшу увагу на аспектах саме програмування. Та все ж радимо вам: занурюючись у більш складну університетську математику даного семестру та наступних, не забувайте отримані знання та навички з MATLAB! Окрім тем та теорем, що будете вивчати, повертайтеся до MATLAB, до наведених MATLAB-прикладів. Продумайте їх ще раз з урахуванням нового теоретичного матеріалу, застосовуйте MATLAB до нових задач! Не обмежуйтеся лише теорією та ручними викладками, що вимагатиме викладач. Робити їх – обов’язково! Та пробуйте виконати їх також і з MATLAB! Прийнамні графіки нададуть красоти та узагальненості вашим результатам. MATLAB лишається вам другом і помічником!

А у нашій дисципліні – **програмуванні** – ми підемо далі, до глибин цієї сучасної та захоплюючої науки. Вважаємо, що ви вже є програмістами першого рівня

майстерності. Ви можете вже ***алгоритмізувати*** нескладні математичні та (дехто) фізичні проблеми, застосувати у програмі не дуже складну алгебру логіки (оператори **&**, **|** та **~**), оператори управління обчислювальними процесами IF…ELSE, FOR, WHILE та SWITCH – CASE.. Та найбільш складне та цікаве з Науки Програмування ще попереду!

У цьому семестрі, з даним курсом ви навчитесь робити ваші програми більш інтелектуальними та привабливими. Програми стануть виглядати набагато сучаснішими, з красивими GUI (Graphical User Interface, інтерфейсом користувача).

Тож будьте готові до складнощів: з простих "одноходових" програм ми переходимо до програмування складних і дуже складних, що керують кількома іншими.

Подих захоплює?

Тоді у путь!

Та пам'ятайте: програми, що надаються нижче, можна скопіювати. Користі це на дасть. Краще так: зрозумійте завдання, та спробуйте самі зробити програму. Не вийшло – поді підгляньте. А вийде – ми поділяємо вашу радість. А якщо вона буде краще за нашу – так напишіть нам про це! Адреса тут: *Ye_Gayev@voliacable.com*.

Ще раз висловлюємо щиру вдячність фірмі MathWorks за гранти у вигляді постійно поновлюваних версій MATLAB, за промоцію попередніх книжок на їхньому сайті *http://www.mathworks.com/support/books/* та Web-семінари і консультації через MATLAB Central Newsreader.

Автори

Once again, we express our sincere gratitude to the firm MathWorks Inc. for their grants in the form of actual versions of the MATLAB constantly upgraded, for their promotion of previous books on their site *http://www.mathworks.com/support/books/* as well as for their Webinars and consultations via MATLAB Central Newsreader.

Authors

Модуль 3

Другий рівень майстерності програміста

5. Більше цікавих програм структурної парадигми

У першій частині навчального посібника ми вже створили кілька програм, а саме:

- *visualize* – програма-скрипт для співставлення графіків трьох функцій, розділ 3.1 з [1];
- *G*1, *G*2, *G*3 – візуалізація Фур'є наближень до заданої функції, все з більшою кількістю гармонік, 3.2 [1];
- *Step_pi* – будує графік періодичної сходинкової функції, 3.2.2;
- *MyNumDiff*, *MyNumInt* – будують і порівнюють графіки похідної (перша програма) та первісної функцій (друга) до деякої вихідної функції, та роблять це двома принципово різними шляхами, аналітичним та чисельним;
- *Welcome* – вітає користувача в залежності від часу доби;
- *MyDeterminant* – обчислює визначник матриці методом її еквівалентних перетворень (іншим методом – буде далі);
- *MyMin*, *MyMax* – знаходять найменший (найбільший) елемент вхідного чисельного вектора та його номер;
- *MyOrder* – повертає вектор з таких саме чисел, як і вхідний вектор, та, одначе, впорядкованих у відповідності до вказаного ключового слова '*increase*' або '*decrease*';
- *Helicopter* – скрипт, що обертає відрізок навколо центру;
- *MyCramer*, *MyGauss* – розв'язують СЛАР довільного порядку методом Крамера та (друга програма) методом Гауса еквівалентних перетворень вхідної матриці.

Різні програми, розв'язують доволі різні задачі. Та використовують вони лише чотири інструменти програмування – блоки *IF–END, FOR–END, WHILE–END SWITCH–CASE–END*. Ще багато чого можна зробити, спираючись лише на це. Надамо і інші приклади програм, бо

немає кращого способу навчитися програмуванню, як написати якнайбільше програм.

5.1. Розумні та ввічливі програми

У ході розвитку комп'ютерних наук програми ставали все більш "розумними" та "інтелектуальними". Це буде головним орієнтиром і у даній книжці. Частково можемо реалізувати це на прикладі програми *Helicopter*, 3.9 у [1].

5.1.1. Програма *Helicopter*. Нагадаємо, що це є програма анімації; яка обертає відрізок навколо його центру. Від нас залежить, якого коліру буде "*Helicopter*", буде він обертатися за годинниковою стрілкою (задамо параметр *Direction*=1) або проти неї (*Direction*=-1), буде швидко обертатися або повільно (параметр *Speed* змінюватимемо від 0 до 0.5). Додатково будемо впливати на ширину відрізку, що обертається. Всі ці параметри стануть формальними аргументами програми *Helicopt1.m*, яку надаємо нижче.

Лістінг програми *Helicopt1.m*

```
function Helicopt1(Nrotations,Color,Direction,Delay,Width,Help)
%Подібно до Helicopter, проте – m-function!
%Програма обертає відрізок кольором Color навколо центру
% проти (Direction=-1) чи за годин. стрілкою (Direction=1)
%визначену кількість разів Nrotations.
% Delay від 0.001 до 0.8 – швидкість обертання,
% Width від 1 до 20 – ширина пропелера.
% Якщо Help='Yes', надає допомогу (інформацію).
%-----------------------------------------
% Приклади запуску з командного рядка:
% Helicopt1(3, 'b', -1, .001,15, 'no')
% або
% Helicopt1(3,'b',-1,.001,15,'Yes')
%-----------------------------------------
%Copyright Ye.Gayev, January 2016

if Help ~='Y'
    figure(1)  %отримувати Figure завжди на екрані
    n=0; RotN=0;
    for Fi=-Direction*(0:pi/100:2*Nrotations*pi)
```

```
        n=n+1;
        x1=cos(Fi);      y1=sin(Fi);
        x2=-x1;          y2=-y1;
        Ox=0; Oy=0; %для «гвіздка»
        plot([x1, x2], [y1, y2],Color,'LineWidth',Width);
        title(['Обертів ', num2str(RotN),' з ', num2str(Nrotations)])
        axis([-1.2,1.2,  -1.2,1.2]),
        axis off  %прибрати неінформативні позначки осів
           if mod(n,200)==0
                RotN=RotN+1;
           end
        hold on, plot(Ox,Oy,'ko','MarkerSize',5), hold off
        pause(Delay)
    end
else
    disp(' '), disp('Як запускати з командного рядка:'),
    disp('============================================')
    disp('Nrots=3;C="m";Dir=-1;Delay=0.2;Width=5;Help="No";')
    disp(' Helicopt1(Nrots,C,Dir,Delay,Width,Help)')
    disp('де Nrots – скільки обертів, C – колір,')
    disp('Dir - напрямок,Delay від 0.001 до 0.8 швидкість,')
    disp('Width від 1 до 20 – ширина пропелера,')
    disp('та якщо Help="Yes" – дати допомогу!')
    disp('============================================')
end
```

Головні оператори програми достатньо пояснені за допомогою коментарів, так що обмежимось лише головним. Програма *Helicopt1* є розумною у кількох значеннях: 1. Вона або виконує завдання, або надає допомогу (якщо *Help*='*Yes*', в апострофах); 2. Завдання доволі різноманітні, що залежить від значень аргументів *Nrotations*, *Color*, *Direction*, *Delay* та *Width*. Спробуйте! 3. Заголовок фігури, який отримуємо, є інтерактивним, тобто кожного разу відображає значення *Nrotations* з завдання та степінь його виконання. Розберіться самі шляхом коментування "сумнівних" операторів, аби зрозуміти їх призначення!

Зверніть увагу: у новій програмі, що є *m*-функцією, немає жодного вихідного аргументу. Тому пишемо її назву відразу за ключевим словом . Порівняйте з випадками, коли

m-функція має один вихідний аргумент або більше одного, див. 5.2.2, 5.3.1 та 6.9.

Та у програмі є й недоліки. Головне з них – її "бюрократизм": якщо звертатися до неї "не по формі", з іншою кількістю аргументів, або випадково якесь значення поставлено не на ту позицію (наприклад, '*Yes*' не шостим аргументом) – працювати не захоче, або працюватиме неправильно! Порівняйте з програмою *plot*(): вона набагато "розумніше". Та незабаром, розділ 6.9, і ми навчимося!

5.1.2. Діалогова форма програми *Helicopter*. У недалекому майбутньому ми зможемо звертатися до програм людською мовою. Та поки що – у "письмовому вигляді" **-:)**. Команди, що використано, були пояснені у розділі 3.4.2:

Лістінг програми *HelicopterDialog.m*

```
%Програма аналогічна попередній,
%та виконує завдання у формі діалога.
%----------------------------------------
% Запуск з командного рядка:
% >> HelicopterDialog
% та відповідай на запитання!
%----------------------------------------
%Copyright Ye.Gayev, January 2016

Welcome  %Якщо ця програма працює
disp('~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~')
disp('Я можу запустити для Вас програму Helicopter,')
disp('якщо Ви відповісте на певні питання з мого боку.')
disp('Починаємо? Натисніть будь-яку клавішу!')
disp('==================================');
pause  % Чекає, що користувач торкне якусь клавішу

Nrotations=input('Скільки обертів хочете зробити? . . .
                                  Введіть якесь ціле=');
Color=input('Якого кольору має бути пропелер? . . .
            Введіть в апострофах r для червоного, . . .
            g для зеленого, тощо. ');
Width=input('Товщина пропелера (Ціле від 1 до 20)? ');
Direction=input('У якому напрямку? Введіть 1, якщо за . . .
                  годинниковою стрілкою, та -1 проти: ');
Delay=input('Нарешті, введіть, будь ласка, число . . .
          від 0.001 (швидке обертання) до 0.8 (повільне): ');
```

```
disp('Дякую! Тепер торкніть якусь клавішу, та . . .
                                        насолоджуйтеся.')
figure(1)  %аби Figure була на екрані завжди!
   n=0; RotN=0;
for Fi=-Direction*(0:pi/100:2*Nrotations*pi)
       n=n+1;
       x1=cos(Fi);      y1=sin(Fi);
       x2=-x1;           y2=-y1;
       Ox=0;Oy=0;
       plot([x1,x2], [y1,y2], Color,'LineWidth',Width);
       title(['Оберт ',num2str(RotN), ' з ', num2str(Nrotations)])
       axis([-1.2,1.2,  -1.2,1.2]),
       axis off     %убрати осі, які на разі зайві
            if  mod(n,200)==0
                 RotN=RotN+1;
            end
      hold on, plot(Ox,Oy,'ko','MarkerSize',5), hold off
      pause(Delay)
end
```

Така програма ніби розмовляє з користувачем, що надає їй ознаки штучного інтелекту. Як вона працює, зрозуміло з коментарів. Нагадаємо лише, що операція *mod*(*n*,200) повертає 0, якщо *n* кратне 200 та інше число у будь-якому іншому випадку. Водночас *n* збільшується на 1 при кожному проході через цикл. Такий алгоритм дозволяє рахувати оберти гелікоптера, див. 6.2.1.

5.1.3. Підсумки. Скрипт або –*m*-функція?

Тепер задамо таке питання: чим принципово відрізняються програми *Helicopt1.m* та *HelicopterDialog.m*, яка з них є скриптом, яка *m*-функцією? Ще питання: після її роботи, чи можна узнати значення внутрішньої змінної, припустимо *RotN*?

Важливо розуміти, що перша з них – *m*-функція, бо має вхідні аргументи та починається ключовим словом *function*. Тому значення *RotN* із зовнішнього MATLAB-середовища непомітне. Діалогова програма, у протилежність, не потрибує такого способу ввода вхідної інформації, є скриптом, и тому *RotN* видно із зовні.

Ця проста програма веде до низки більш цікавих програм. Обертання правильного багатокутника, див. Лаб. Роботу № , обертання зірки або багатолепесткової рози, рис.5.1 наприкінці розділу, що раніше створені нашими студентами. А в вас є хист до самостійного пошуку?

5.2. Обчислення функцій за їх рядами Тейлора

Звернемося до проблем обчислення значень функцій $\sin x$, $\cos x$, e^x, $\ln x$ тощо від аргументу x. Може, це нудно для когось, та принципово важливо програмістові!

Відомо, що комп'ютер у "початковому стані" здатен лише виконувати чотири арифметичні дії – складення, віднімання, множення та ділення чисел, попередньо перетворених у двійкове представлення. То як же він обчислює названі функції?

Допомогло відкриття Тейлора (див. 4.2 [1]), що будь-яку "хорошу" функцію можна представити у вигляді нескінченного ряду, де є як раз лише множення (точніше – піднесення до степеню). Наприклад, потрібно обчислити $y = \ln x$ для певного $x > 0$. Так прямо розклад логарифма у ряд Тейлора поблизу $x = 0$ не знайти. Беремо $y = \ln(x+1)$, запитуємо MATLAB:

>> *syms x*; *R*=*taylor*(*log*(*x*+1), '*Order*', 11); *pretty*(*R*)

та отримуємо відповідь

$$\frac{x^{10}}{10} + \frac{x^9}{9} + \frac{x^8}{8} + \frac{x^7}{7} + \frac{x^6}{6} + \frac{x^5}{5} + \frac{x^4}{4} + \frac{x^3}{3} + \frac{x^2}{2} + x$$

(запросили 11–1=10 доданків, та легко зрозуміти загальний вигляд). Такою формулою і користуються в комп'ютерних науках. Цей розклад для $\ln(x+1)$ залишимо для домашнього завдання, а надалі знов працюємо з обчисленням синуса (4.2) (завдання 1 Лаб. роботи № з [1] і 3 і 4 Лаб. роботи №8),

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \ldots + (-1)^n \frac{x^{2n-1}}{(2n-1)!}. \qquad (5.1)$$

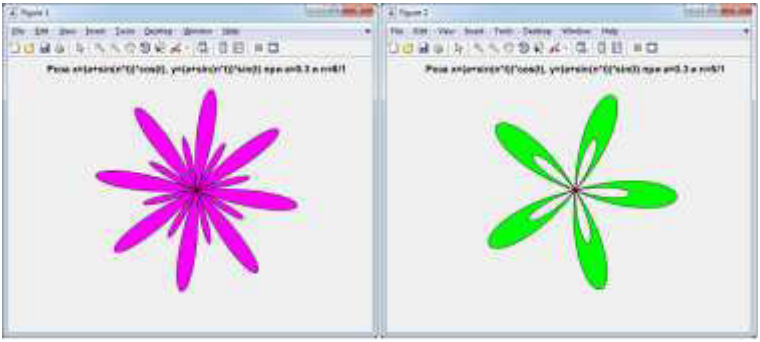

Рис.5.1. Програма обертання багатопелюсткової рози студентки Д.Малініної (2014)

5.2.1. Створимо таку програму *MySinN.m*, яка для заданого числа x видасть значення $\sin x$. Бачимо з (5.1), що додатково, окрім аргумента x, треба подати n, кількість доданків ряду Тейлора, що хочемо врахувати. Тобто, *MySinN* має бути *m*-функцією від двох аргументів x та n. Нижче пропонуємо програму у найпростішому варіанті.

Лістінг *MySinN.m*

```
function Sin =MySinN(x,n)
% Програма обчислює sin(x) за рядом Тейлора цієї функції
%              x^3    x^5    x^7                x^(2n+1)
% sin(x)= x- ---   + ----  -  ---- +...+(-1)^n --------
%              3!     5!     7!                 (2n+1)!
% (Аргумент беремо у градусах!)
%============================
% Приклади використання:
% 1) >> y=MySinN(30,10)
% 2) >> y=MySinN([30, 180+30],100)
% Copyright of Malinina, 15.01.2015
%============================

x=pi*x/180; %Обчислення Радіан за Градусами
F=1; %Початкове значення факторіалу
Sin =x;  %Початкове значення результата
for i=1:n
      F=F*2*i*(2*i+1); %Значення факторіалу
      Term=((-1)^i)*(x.^(2*i+1))/F; %Спільний член
      Sin = Sin + Term; %Накопичення результату
```

```
end
disp('Подивіться на різниіцю з точним значенням:')
Sin - sin(x)
```

Можна бачити, що авторка цієї програми Д.Малініна не полінувалася створити красиву хелпову частину. Це дійсно важливо для користувачів! Тепер власне про програму. Спочатку аргумент *x* у градусах перетворюється у радіани. Далі майбутнім значенням факторіала *F* та вихідного аргументу (результату) *sin* надається початкових значень 1 та *x* відповідно (зверніть увагу: це типовий прийом програміста!). Нарешті, виконується замовлена кількість циклів *n*, на кожному обчислюється *Term* (що є загальним членом згідно з (5.1)) разом з факторіалом *F* та результатом *sin* (таке ім'я захотілося авторці**-:)**. Легко вручну переконатися, що на кожному циклі ці величини набувають потрібних значень:

```
i=1   F=F*2*i*(2*i+1)=2*3   Term=-x^3/3!   sin=x-x^3/3!
i=2      F=3!*4*5=5!        Term=x^5/5!    sin=x-x^3/3!+ x^5/5!
i=3      F=5!*6*7=7!        Term=-x^7/7!   sin= sin - x^7/7!
                  і так далі.
```

Далі випробовуємо програму: *y=MySinN*(30,5) дає *y*=0.5000, так само як із більшим числом доданків *MySinN*(30,50). Більш того, щойно створена функція працює і для матриць:

```
>> y=MySinN([30  60  90; 30+360  60+360  90+360], 50)
    y =   0.5000    0.8660    1.0000
          0.5000    0.8660    1.0000
```

Це сталося тому, що піднесення у степінь ми використали із крапкою, *x*.^(2**i*+1), що автоматично враховує "матричну філософію" MATLAB. Ознакою правильності програми є також те, що так обчислена функція лишається періодичною з періодом T=360º.

Та піддамо нашу програму більш жорстким випробуванням. Візьмемо аргумент *n*=200 або більше:

```
>> y=MySinN( [30+360  60+360  90+360], 200)
    y =          NaN   NaN   NaN
```

Немає результату (Not a Number)! Однак, для менших значень *n* хоч якось спрацьовує:

```
>> y=MySinN([30+360  30+2*360  30+3*360],20)
y =1.0e+03 *(0.0005    0.0005    2.9347 )
```

Тобто, для кутів 390° та 750° результат привильний, а для 1110° (30 °+3*360°) – більше одиниці. (Останні два рядочки коду аналізують коректність результату). Чому така помилка? Як користуватися представленням (5.1), аби отримувати правильні результати для великих кутів? Це обговоримо далі, та у третій частині підручника.

Та головний недолік програми *MySinN* полягає в іншому: звідки обирати кількість доданків у ряді Тейлора *n*? Зрозуміло, що для малих аргументів, як *x=30°*, він може бути невеличким, а для аргументів типу *x=30°* + 5**360°*?

5.2.2. Виникає практична потреба обчислювати функцію $\sin x$, $\cos x$ або e^x тощо не за кількістю доданків у її розкладі у ряд Тейлора, а за досягненням певної точності ε. Є практичні задачі, де точність $\varepsilon = 0.01$ достатня. А є й такі, де потрібно $\varepsilon = 1 \cdot 10^{-7}$! Зрозуміло, що для малих значень аргумента x достатньо взяти малу кількість доданків ряду Тейлора n, а для великих x треба більше доданків у розкладі (5.1). А яке саме n, як визначити?

Зрозуміло, що на кожнім циклі вихідний аргумент змінюється на $R = \dfrac{x^{2n-1}}{(2n-1)!}$ (за модулем). Таким чином, слід використати цикл ***while***, перевіряючи умову $|R| < \varepsilon$. Це реалізовано у наступній програмі.

Лістінг *MySinEps.m*

```
function [Sin, i]=MySinEps(x,eps)
% Програма розрахунку sin(x) до досягнення точності eps.
% за допомогою рядів Тейлора
%              x^3    x^5    x^7              x^(2n+1)
% sin(x)= x- ---   + ----  -  ---- +...+(-1)^n --------
%              3!     5!     7!               (2n+1)!
% (Аргумент беремо у градусах!)
%==========================
```

```
%Приклад використання:
% example of use:
%  >> [Sin,i]=MySinEps(90,.001)
% i = 5 показує, скільки доданків було взято,
% аби досягти задану точність.
% Copyright of Malinina, 15.01.2015
x=pi*x/180;
Sin =0;
Term=1;
i=0;
while abs(Term)>eps  % повторюємо, якщо умова виконується
    Term=((-1)^i)*(x.^((2*i)+1)/factorial((2*i)+1));
    Sin = Sin +Term;
    i=i+1;     %  рахуємо кількість доданків
end
disp('Подивіться на різниіцю з точним значенням:')
Sin - sin(x)
```

Підкреслемо ще одну відмінність даної програми 5.2.2 від 5.2.1: у ній два вихідних аргументи, у той час як у попередній – один. Другий вихідний аргумент, *i*, вказує кількість доданків ряду Тейлора, що забезпечують замовлену точність. Висновок: коли вихідних аргументів більше одного, робимо один, але це вже буде *вектор аргументів*. Одначе, вихідні аргументи можуть бути й відсутніми – приклад надає розділ 5.1.1.

Алгоритм, що тут використано, пояснено блок-схемою на рис.5.2 наприкінці розділу. Вона за правилами [12-14] демонструє загальну структуру програми, хід обчислень за нею. Останній полягає у виконанні блоків 5 – 9 у циклі до досягнення умови $|R| \le \varepsilon$. Після цього програма виконує вивід обчислених значень *Sin* та *i*, і зупиняється (блоки 10,11).

У лабораторній роботі № 8 таку або аналогічну роботу радимо зробити самостійно!

5.3. Більш розумні *MyMin*, *MyOrder*

5.3.1. Блок *switch-case-end*. Програму *MyMin* у пп. 3.7.1 з [1] ми розробили для пошуку найменшого елемента у числовому векторі *x*. Та вона не відповідає “філософії MATLAB”, згідно якої не має бути різниці, чи є аргумент *x*

вектором чи двовимірним масивом (матрицею). Узагальнемо її таким чином, аби вона знаходила найменший елемент у кожному рядочку матриці, або у кожному стовпчику в залежності від того, як замовить користувач. Таким чином, програма має повернути вектор-стовпчик у першому випадку, та вектор-рядок у другому. Першим аргументом програми має бути матриця, що досліджується, а другим – *KeyWord*, що може приймати значення '*Row*' або '*Col*'. Створюючи нову програму вже можемо використовувати ті, яким ми MATLAB вже навчили; такими є, у даному випадку, *MyMin* і *MyMax*.

Лістінг *MyMinArray.m*

```
function [V,Indices]=MyMinArray(X,KeyWord)
%Шукає найменші числа у рядках матриці X,
% або у стовпчиках, залежно від значення
% KeyWord='row' або 'column', та повертає вектор з них,
% та вектор цілих Indices з їх положенням у матриці X.
%Приклади використання:
%>>x=[34 567 89;67 -9 0;21 37 50;85 19 47;12 57 64],[Y,ind]=MyMinArray(x,'row')
%   Y =    34    -9    21    19    12
% >> A=randi(20,  4,5),  [T,ind]=MyMinArray (A,'column')
%Copyright of D.Malinina, 09.02.2015
S=size(X); %видає вектор з кількістю рядочків та стовпців
switch KeyWord
    case 'row'
%         V=zeros(1,S(1));
        for i=1:S(1)
            [V(i), Indices(i)]=MyMin(X(i, : ));
        end
    case 'column'
%           V=zeros(1,S(2));
        for i=1:S(2)
            [V(i), Indices(i)]=MyMin(X( : ,i));
        end
    otherwise
        disp('Перевірте Ваше завдання!')
        disp('Окрім матриці')
        V=X, Indices=NaN;
        disp('Має бути або "row",  "column"')
end
```

Структура *SWITCH–CASE–CASE. . .–OTHERWISE–END* була пояснена у розділі 3.7.3, [1]. Першим аргументом даної

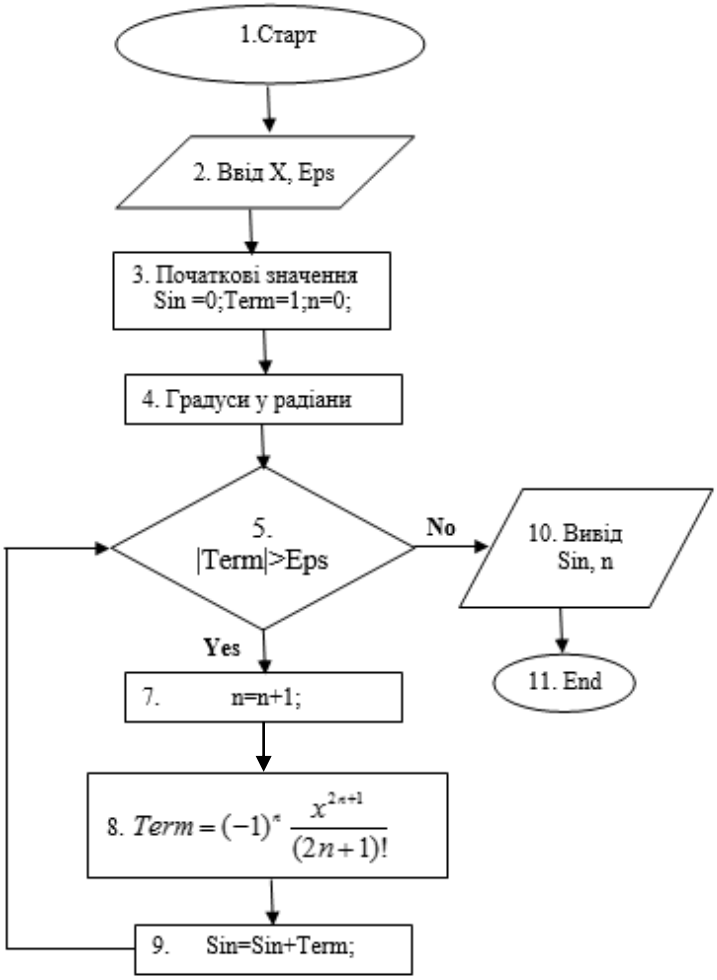

Рис.5.2. Блок схема програми *MySinEps.m* для обчислення сінуса з потрібною точністю за його рядом Тейлора

програми має бути числова матриця *X*; вона може бути згенерована таким чином *X*=*randi*[1](20,4,5).

Обчислювальний процес працює з нею в залежності від значення другого аргументу *KeyWord*. Якщо *KeyWord*='row', то у результуючий вектор *X* попадають найменші числа з рядків матриці *X*(*i*, **:**); якщо ж *KeyWord*='column' (ці слова мають бути в апострофах, текстовим типом даних, див. 6.2), то програми видає найменші із стовпчиків X(**:** ,*i*). Одночасно знаходимо індекс даного значення та заносимо його у вектор *Indices*. Це, здається, легко зрозуміти з лістінгу програми. Якщо *KeyWord* іншим, користувачеві видається порада перевірити завдання. Та навіть якщо програма йде останнім шляхом, то все одне слід надати якихось значень вихідним аргументам. Обрано таке рішення: *V*=*X*, *Indices*=*NaN*. Аналогічно працювала програма *MyOrder*, розділ 3.7.2

Алгоритм пояснено блок-схемою рис.5.3. Спочатку, від блоку 1 до 4, алгоритм має лінійний характер. Блок 4 перевіряє, чому дорівнює *KeyWord*. Якщо *KeyWord*='row', то рухаємось правою гілкою, виконуючи блоки 5 – 8. Тут також блоки 6 та 7 виконуються у циклі, поки не переберемо усі рядки. Для зображення цього ми використали певні стандарти для блок-схем [12-14]. Окрім цього, дана блок-схема не показує подробиці роботи блоку 7, а зображує його узагальнено, як підпрограму *MyMinM*, також за стандартом [12-14]. Інколи додатково роблять “виноску”, де дають текстове посилання на сторінки або рисунки, де робота цієї підпрограми пояснюється.

Якщо ж *KeyWord*='column', то рухаємося лівою гілкою; там дії аналогічні, але найменші числа шукаються вже за стовпчиками (окрема підпрограма 11). Якщо ж *KeyWord* має якесь інше значення, то програма іде за середньою гілкою, де передбачено надати користувачеві пораду щодо правильного формулювання завдання.

Програма має очевидний недолік: навіть якщо дамо їй розумне завдання

[1]Якщо незрозуміло, дивіться >>*help randi* у командному рядку.

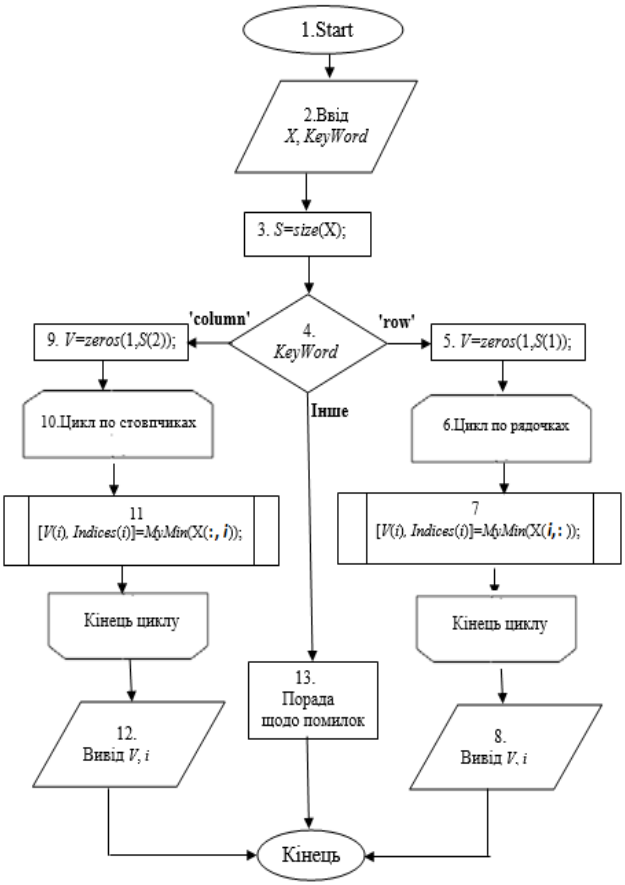

Рис. 5.2. Блок-схема пошуку найменших чисел у рядочках матриці *X* (права гілка), або у стовпчиках *X* (ліва гілка) програма *MyMinArray.m*

>> *A*=*randi*(20, 4,5), [*T,ind*]=*MyMinArray*(*A*,'Column') ('Column' в апострофах з великої літери, або 'CoLuMn' і таке інше), отримаємо відповідь

Перевірте Ваше завдання!
Окрім матриці
V = ...
Indices = NaN
Має бути або “row”, або “column”

бо 'column' не дорівнює 'Column'. Як виправити програму, зробити її ще трохи розумніше – розглянемо у 6.2 і 6.9.

Важливе спостереження: зверніть увагу, що у редакторі *m*-файлів рядки коду *V*(*i*)=*MyMin*(. .); завжди підкреслені рожевим. Тому, це не помилка, лише попередження "*The variable ... appears to change size every loop iteration...*". Чому це? Це натякує на те, що збереження цієї змінної потрибує не *статичного блоку пам'яті*, як зазвичай, а *динамічної пам'яті*, розмір якої змінюється у ході виконання програми. Пояснення буде далі, а зараз лише рецепт: "розкоментувати" попередні рядочки типу *V*=*zeros*(1,*S*(2)). Тепер збереження масиву *V* потребуватиме не динамічної, а статичної пам'яті, ось MATLAB і "задоволений". Перевірте!

Для подальшого будемо вважати, що ви створили й аналогічну програму *MyMaxArray.m*. Таким чином, цеглина на цеглину, створюють у комп'ютерній науці програми все складніші й складніші[2].

5.3.2. Ще один параметр – ключове слово. У розділі 3.7.2 запропоновано програму, що пересортировує вхідний вектор у відповідності до ключового слова *"Increase" або "Decrease"*. Вона не є достатньо універсальною. Перевірте: програма добре працює для векторів-рядків вимірності 1∗*N* та не працює для векторів-стовпчиків *N*∗1. Причина – утворення вектора меншої вимірності *X*1=[*X*1(1 : *I*-1), *X*1(*I*+1 : *end*)], коли з нього треба викинути знайдений найменший або найбільший елемент. Аби виправити такий недолік, пропонуємо зробити модіфіковану версію такої програми, що використовує принципово інший алгоритм: якщо на черговому циклі знайшли найменший елемент (аналогічно, для найбільшого), то замінюємо його на таке число, що напевно більше (менше) за усіх елементів, що лишилися в *X*1, і тому у наступному циклі ніяк не може виявитися найменшим (найбільшим). Таким числом буде 10^10 або *NaN*

[2]Природно, що програми *sin*( ), *cos*( ), *min*( ), *sort*( ) та інші в MATLAB вже існують. Та нам важливо зараз зрозуміти, як вони працюють, засвоїти відповідні алгоритми.

(відповідно, –10^10 або –*NaN*). Надаємо модифікацію програми .

Лістінг *MyOrderM.m*

```
function Y=MyOrderM(X, Order)
%Така ж програма як "MyOrder.m", лише змінено алгоритм,
% завдяки чому працює як з векторами 1*N, так і з N*1.
% Повертає вектор Y з елементами вхідного вектору X
%у порядку зростання, якщо Order є 'Increase',
% та у порядку спадання, якщо Order є 'Decrease'.
% Приклад використання:
% >> X=randi(20,10,1)-10,  Y=MyOrderM(X, 'increase').
% Використовує MyMax і MyMin, розроблені раніше.
% Copyright Ye.Gayev, Febr. 2016

L=length(X);  X1=X;
Y=X;  %Початкове значення результату
switch Order
    case {'Increase', 'increase'} %у порядку збільшення
        for i=1 : L
            [Y(i), I]= MyMin(X1);
            X1(I)=NaN; %елемент № I вже не буде найменшим
        end
    case {'Decrease', 'decrease'} %у порядку зменшення
        for i=1 : L
            [Y(i), I]= MyMax(X1);
            X1(I)=-NaN;%елемент № I вже не буде найбільшим
        end
    otherwise
        disp(' ')  % задля того, щоб пропустити рядочок
    disp('Другий параметр має бути "Increase" або "Decrease".')
    disp('Перевірте Ваше завдання і спробуйте ще раз!')
    disp(' ')
end
```

Зверніть увагу, що запис програми “сходинками” значно полегшує її сприйняття у читанні! Легко бачити, де блок починається, яким ***end*** закінчується.

5.3.3. Три вхідних аргументи. Тепер зробимо більш загальну програму як до останньої, так і до двох попередніх *MyMinArray* і *MyMaxArray*. Це буде програма *MyOrderArray* з трьома вхідними аргументами – матрицею *X* та двома

ключовим словами; перше може бути *Order*, що дорівнює *"Increase" або "Decrease"*, а друге – *KeyWord* як раніше, *"Row" або "Column"*. Результатом (вихідним аргументом) буде матриця тієї ж вимірності, у якій рядки (або стовпчики) вміщують такі само числа, та впорядковані у відповідності до аргументу *Order*, тобто або зростають, або зменшуються. Використовуючи вже розроблені програми, маємо таку:

Лістінг *MyOrderArray.m*

```
function Y=MyOrderArray(X, Order, RowColumn)
%Повертає такі саме числа з масиву X, але упорядковані
%за зростанням або падінням в залежності від
% Order='Increase' або Order='Decrease',
% по рядках, якщо RowColumn='Row',
% та по стовбчиках, якщо RowColumn='Column'.
%Приклад використання:
% >> X=randi(20,7,5)-10,  Y=MyOrderArray(X, 'Decrease','row')
% Copyright Ye.Gayev, Febr. 2016

Size=size(X);
Nrows=Size(1); Ncols=Size(2);
  Y=X;
switch Order
    case 'Increase'
        switch RowColumn
            case 'row'
                for i=1:Nrows
                    Y(i, : )=MyOrderM(X(i, : ), 'Increase') ;
                end
            case 'column'
                for i=1:Ncols
                    Y( : ,i)=MyOrderM(X(:, i ),'Increase');
                end
          otherwise
              disp('Щось не так з Вашим завданням!')
              disp('Третій аргумент має бути або "row" ')
              disp(', або "column" в апострофах.')
              disp('Перевірте, та спробуйте знов!')
      end
    case 'Decrease'
        switch RowColumn
            case 'row'
```

```
                for i=1:Nrows
                    Y(i, : )=MyOrderM(X(i, : ), 'Decrease') ;
                end
            case 'column'
                for i=1:Ncols
                    Y( : ,i)=MyOrderM(X( : ,i),'Decrease');
                end
            otherwise
                disp('Щось не так з Вашим завданням!')
                disp('Третій аргумент має бути або "row" , ')
                disp('або "column" в апострофах.')
                disp('Перевірте, та спробуйте знов!')
        end
    otherwise
        disp('Щось не так з Вашим завданням!')
        disp('Другий аргумент має бути або "Increase", ')
        disp(' або "Decrease" в апострофах.')
        disp('Перевірте, та спробуйте знов!')
end
```

Це є доволі розумна програма: деякі її користувачі можуть здивуватися, що вона навіть підказує, коригує неправильно поставлене завдання. Бачимо, що у ній дві вкладені структури *switch–case– . . –otherwise–end*, та в останній ще блок *for–end*. Ось як вона працює з командного рядка:

```
>> X=randi(20,7,5)-10,  Y=MyOrderArray(X, 'Decrease','row')
X =    7    1    7   -9    4
       9   10   -7    7   -6
      -7   10   -1    9    5
       9   -6    9    4   -9
       3   10    6    6   -4
      -8   10   10    5   -9
      -4    0    4   -2   -8

Y =    7    7    4    1   -9
      10    9    7   -6   -7
      10    9    5   -1   -7
       9    9    4   -6   -9
      10    6    6    3   -4
      10   10    5   -8   -9
       4    0   -2   -4   -8
```

Перша команда закінчується комою, і тому видає *випадковий* масив 7*5 цілих чисел між -10 та 10. Програма *Y=MyOrderArray*(*X*, 'Decrease','row') повертає масив *Y* з тих саме чисел, які по рядках упорядковані за зменшенням. Програма легко сприймається завдяки її записом "сходинками". Виглядає так, що у ній лише 42 рядки (коментарі не враховано). Та насправді, у кожному циклі кожного випадку використовується ще програма (точніше, кажуть *підпрограма*) *MyOrderM*. І якщо врахувати і ці коди, програма разом містить 120 рядків коду. Вже доволі велика й складна!

5.4. Робота із звуком та зображеннями в MATLAB

Досі ми працювали із звуком пасивно, могли додати до програми той чи інший звук, що вже є у MATLAB,

```
>> load handel; sound(y)
>> load laughter; sound(y)
```

та інші, що зберігаються у директорії
C:\Program Files\MATLAB\R2012a\toolbox\matlab\audiovideo\
з розширенням ***.mat***. Задля більшого задоволення від програмування, для збільшення можливостей ваших програм доцільно вивчити більше як про звук, так і про зображення.

MATLAB-команда *sound* розуміє лише *mat*-файли, тобто числові дані, спеціальним чином конвертовані із стандартних ***.wav*** або ***.mp***3 звукових файлів. Допустимо, ваша улюблена музика *Music.mp*3 зберігається саме у такому файлі (важливо: всі назви мають буть латиною без пропусків!). Якщо, крім того, він знаходиться в робочій директорії MATLAB, де містяться всі ваші програми, то виконати цю музику можна таким чином:

```
>> [y,Fs]=audioread('Music.mp3'); %конвертація до mat-формату
>> sound(y,Fs)  %виконання музики
```

MATLAB-розробник повідомив, однак, що команда *sound* має бути видалена з наступних релізів. Вони пропонують нові команди, засновані вже на об'єктно-орієнтованій парадигмі:

>> *MusicStuct*=*audioplayer*(*y*,*Fs*); %конвертація до структури
>> *play*(*MusicStuct*) %виконання музики

Якщо ж музику розміщено у директорії MUSIC (це зручніше), то треба вказувати *повний шлях* до неї на зразок

>> *MusicPlace*='c:\Users\. . . :\MUSIC\ *Music.mp*3';
>> [*y*,*Fs*]=*audioread*(*MusicPlace*); %конвертація до mat-формату
>> *MusicStuct*=*audioplayer*(*y*,*Fs*); %конвертація до структури
>> *play*(*MusicStuct*) %виконання музики

Використання звуку, взагалі кажучи, не входить до стандартного навчального курсу. Та якщо комусь це цікаво – розберіться самі! Це може бути темою курсової роботи. Надаємо приклад такої у розділі 10.2.2.

Також якість ваших програм можна суттєво поліпшити, якщо використовувати “мистецькі” властивості MATLAB. Будь-яке зображення *MyPict.jpg* (або в інших графічних форматах) розміщуємо у робочій директорії MATLAB; тепер:

>>*MyPictMat*=*imread*('MyPict.jpg'); %конвертація до mat-формату
>>*image*(*MyPictMat*) %зображення у графічному вікні

Тут є великий простір для фантазії!

5.5. Про парадигми програмування

Персонаж Жана-Батіста Мольєра, такий собі "Міщанин-шляхтич" Журден, пізнав одного разу, що він не такий-сякий, а вже 40 років розмовляє прозою! От і нам прийшов час узнати, що ми поки що користуємося лише однією з кількох методологій (парадигм) програмування [7], а саме – ***структурним програмуванням***, [8]. Його ж називають ще процедурним програмуванням, [10]. Дана парадигма[3] полягає у тому, що створюються багато програм для доволі простих завдань, потім вони використовуються у розробках більш складних у якості підпрограм, ці програми також використовують як підпрограми у ще складніших, і так далі аж до таких програм, як Microsoft Word, як Microsoft Windows, MATLAB тощо.

[3] https://uk.wikipedia.org/wiki/Парадигма_(філософія)

Аби таке стало можливим, треба строго, однозначно викликати підпрограми за їх *сігнатурою*, тобто з урахуванням кількості вхідних та вихідних аргументів, та ще й змісту аргументів (де числа, де текстовий аргумент).

Декому це здастся занадто бюрократичним. Та якщо це дуже не подобається – знайдемо шляхи зменшення бюрократичності! Та це буде далі, у розділі 6.9. А поки що жодного відступу від того стандарту, за яким програму створено!

Аби краще відчути у подальших задачах сутність саме ***структурного програмування***, корисно узнати, які ще існують парадигми програмування [8]. MATLAB здатен ще на ***об'єктно-орієнтоване програмування*** (ООП) [11]. У третій частині підручника будемо вивчати мову Java, що користується виключно ООП-парадигмою.

Для закріплення нового матеріалу даного розділу пропонуються Лабораторна робота № 7 і Лабораторна робота № 8.

5.6. КОНТРОЛЬНІ ПИТАННЯ ДО РОЗДІЛУ 5

1. Скільки вхідних та вихідних аргументів можуть мати програма-скрипт і *m*-функція? Які формати запису використовують для *сігнатур* цих програм (тобто у їх заголовках[4])?
2. Які MATLAB-команди слід використовувати для створення диалогових програм? Діалогові програми є скриптами чи *m*-функціями?

[4] Пояснимо ще раз, що *сигнатура* – це як виглядає виклик програми, наприклад *plot*(*x*) (в аргументах один вектор), або *plot*(*x*,*y*, 'r') (два вектори та колір), тощо.

3. Що таке *формальні аргументи* та *фактичні аргументи* програми? Різниця між ними?
4. Що станеться, якщо до програми з, припустимо, двома вхідними аргументами звернемось з одним або трьома фактичними аргументами?
5. Використовуємо програму 1, яка, у свою чергу, викликає програму 2 (підпрограму). Остання створює масив, припустимо, *P*. Чи "помітен" він з програми 1, чи може бути використаний? А якщо "дуже хочемо" бачити?
6. Якими засобами програмісти надають рис "інтелекту", "розумності" та "ввічливості" своїм програмам?
7. У якому сенсі програма *Helicopt1* є "розумною" та "ввічливою"?
8. Які головні ідеї лежать в основі програм анімації, як *Helicopt1*?
9. У чому сутність ряда Тейлора? Як отримати ряд Тейлора для, скажімо $x \sin x$ або $\sin x \big/ x$?
10. Як забезпечити потрібну точність в обчисленні функції за її рядом Тейлора? Як взагалі пояснити термін, *точність*?
11. Поясніть, за яким алгоритмом працює програма пошуку найбільшого елемента числового вектора *MyMax*? Прохання надати її блок-схему.
12. Як навчити попередню програму "розуміти" масиви (програма *MyMaxArray*)?
13. Поясніть, за яким алгоритмом працює програма пошуку найбільшого елемента числового вектору *MyOrder*?
14. Поясніть, за яким алгоритмом працює програма сортування числового масива *MyOrder*, надайте блок-схему алгоритму.
15. Які стандартні блоки (елементи) використовують для зображення блок-схем алгоритмів?
16. Як зображають на блок-схемах початок та закінчення алгоритму?
17. Як відрізняються лінійні та розгалужені блок-схеми? Чому виникають розгалуження у блок-схемах алгоритмів?

18. Як зображають на блок-схемах ввід початкової інформації та вивід результатів? Якими, у якій формі можуть бути ци результати?
19. Як зображають логічні дії на блок-схемах? Скільки може бути "вхідних стрілочок" до блоку логіки, скільки "стрілочок" може виходити? Що надписують?
20. Як на блок-схемах зображають циклічне виконання певних дій? Які два варіанти існують?
21. Які ви знаєте парадигми програмування, у чому вони полягають?
22. Яку роль грає оператор *global*?

5.7. ЗАДАЧІ ДО РОЗДІЛУ 5

1. Маючи вже досвід з обертанням многокутника, зробіть програму обертання „трипелюсткової рози" $x = \sin(3t)\cos(t)$, $y = \sin(3t)\sin(t)$, $t \in [0,2\pi]$ (Рис.5.31).

2. Знаємо, що задати колір через *Text* можемо лише вісьми варіантів *Text*=['w','k','b','g','r','y','c','m']. Однак, колір можна ще задавати числовим вектором *Color*=[*r b g*], де додатні числа *r*, *b*, *g* не більше 1. Пропонуємо створити програму (назвемо її *MyColor*3), що генерує 18 кольорів у вигляді таких числових векторів. В розділі 7.1.3 вона стане до нагоди. **Пропонуємо алгоритм дій:** беремо число $a = 1/3$ та утворюємо всі можливі розміщення a, $2a$ та $3a$ на трьох позиціях такого вектора.

3. Розробіть програму *Palette*(*n*), що за заданим *n* видає вектори *Color*=[*r*, *b*, *g*] довжиною 3 із значеннями $r,b,g = 0, 1/n, 2/n, \ldots, n/n$. Приклад: *Palette*(2) видає лише вектори (кольори) [0 0 0], [0 0 ½], [0 ½ ½], [1 ½ 1] і т.д. – скількі кольорів у *Palette*(2)? У довільній *Palette*(*n*)?

4. MATLAB-програму, що обертає „трипелюсткову розу" узагальніть таким чином, аби були можливості вибору: за годинникової стрілкою чи проти, швидко чи повільно, рожевого або іншого кольору, рис.5.31.

5✍**.** Узагальніть попередню програму, аби вона обертала на екрані *N*-пелюсткову розу. Увага: парні *N* вимагають певних хитрощів!

6✍. Розробіть програму, що створює на екрані пульсуючий еліпс, або пульсуюче сонечко з „променями”.

7✍. Розробіть діалогову програму, яка запитує користувача про ціле число *N*, створює на дисплеї правильний *N*–кутник того чи іншого кольору та обертає його у замовленому напрямку. У другій частині підручника „одягнемо” цю задачу у графічний інтерфейс користувача (GUI).

Сподіваємось, Ви розв’язали більшість задач.
Вітаємо! Можете рухатись далі.

6. До розумних комп'ютерів: Складні типи даних

Концепції "***тип даних***" спочатку не існувало у комп'ютерній науці. Таке поняття виникло у ході бурхливого розвитку цієї науки внаслідок узагальнення та розмаїття результатів, що було отримано. Не випадково, що класичні книги Вірта [18] та Ахо [19] базуються саме на цьому понятті! Та особливого значення це поняття набуло з появою об'єктно-орієнтованого програмування, про що мова у наступній частині підручника.

Що ж таке "***тип даних***"? З цим ми пов'язуємо певний визначений набор характеристик (*полів*, розділ 6.5), спосіб їх збереження, як цілісність, у пам'яті комп'ютера та дозволених дій над даним об'єктом[5] (*методів*). Так, з поняттям "*шахова гра*" пов'язуємо наявність дошки 8-на-8, визначеного набору фігур, правил ходів кожної фігури та гри загалом.

Подивимось, як це реалізовано у вже знайомих нам поняттях. Кажемо, що змінна така-то належить до певного ***типу даних***, якщо знаємо (*i*) з яких інших даних вона складається (як правило, більш простих), та (*ii*) строго визначені дії, дозволені над даними такого типу. Розглянемо приклади.

6.1. Які типи даних вже знаємо?

6.1.1. Базовий для MATLAB тип даних – масив. Розглянемо спочатку тип даних ***числовий масив***. Що це таке? Дамо *рекурсивне* визначення:

i. Окреме число – масив. Для чисел визначені аріфметичні операції +, –, *, / (первинні);

ii. Набір чисел (числовий вектор) – масив. Такий набір пов'язуємо з певним іменем через квадратні дужки, $x=[x_1, x_2, \ldots, x_n]$ або $y=[y_1; y_2; \ldots; y_n]$;

[5] https://uk.wikipedia.org/wiki/Тип_даних, https://ru.wikipedia.org/wiki/Тип_данных

iii. Прямокутна таблиця чисел – масив. Такий масив вводимо у пам'ять комп'ютера так, як описано у розділі 3.1. Над числовими масивами визначені як первинні операції, так і подальші **.***, **./**, **^**, **.^**, *length*( ), *size*( ) та інші, операції доступу до певних елементів *x*(*i*,*j*) або частин масиву *x*(2:3:*end*, **:**) тощо.

Комплексні числа $a+b\boldsymbol{i}$ – окремий тип даних; йому притаманні ще деякі операції (модуль числа $|x|=\sqrt{a^2+b^2}$).

6.1.2. Текстовий тип даних – це будь-яка стрічка з літер, номерів, символів, як наприклад

x='мій АдРес Name5@i.ua', *y*='3.1415'

(апострофи позначають, що це саме *текст*). Найпростіші дії над ними – див. 1.7.2. Вони суттєво відрізняються від дій над числовими масивами, чи не так?

Прямокутна таблиця з таких елементів – вже *текстовий масив* (у Java його називають *String*[ ]); приклад – задача 16 лабораторної роботи №1. Взагалі, "філософія MATLAB" полягає у тому, що якщо утворено якийсь новий тип даних, то можемо працювати і з масивом таких даних, і з масивом масивів, і так далі до розумної глибини.

6.1.3. У розділі 1.7.2 ми визначили також *символьний тип даних* – це будь-який текст, що можна інтерпретувати як правильний математичний вираз. Зрозуміло, що у пам'яті комп'ютера він зберігається як текст, з поміткою "*syms*", і щойно комп'ютер до нього звертається, так відразу "розуміє", що від такого "тексту" можна отримати певні числові значення, виконувати операції *diff*( ), *int*( ), *ezplot*() тощо.

6.1.4. У розділі 3.3.1 використали термін *Умова*, не пояснивши його. Так от, *Умова* – це вираз, який приймає значення типу ***boolean*** (читають *бу'лиан*), тобто такий, що є або ***true*** (істина), або ***false*** (хибність). З кількох *Умов* можна утворювати складні *Умови* за допомогою операцій &, |, ~, &&, ||. Часто кажуть, що *true* є 1 (один), та *false* є 0 (нуль). Та арифметичні операції над таким типом даних у більшості алгоритмічних мов заборонені. Окрім MATLAB (див. 3.3.2, с.

131 [1])! Програму *TryTest*2, що використовує *boolean* тип даних, див. в 6.8.

Раніше використовували умови на зразок A>B, A==B, тепер познайомимось із більш складними умовами. Так, команда *BoolenValue*=*is**numeric***(*A*) надає *BoolenValue* значень *true* (1) або *false* (0) залежно від того, є *A* числовим масивом чи ні. Аналогічно, *is**integer***(*A*) видає *true*, якщо *A* є масивом з цілих (тобто типу ***uint*8**), та *false* в інших випадках.

Далі познайомимось з іншими важливими типами даних, що використовуються не лише у MATLAB, а й в інших високорівневих мовах програмування та у комп'ютерних науках взагалі.

6.2. Більше – про текстові дані

У лабораторній роботі №1, завдання 16 ми лише "гралися" з текстовим типом даних. Побачили, що вони є навіть "попередниками" символьних даних, символьної математики. Використовували їх для організації "діалогу" користувача з комп'ютером, а також у блоках *switch–case–end*. Це дуже важливий тип даних! Мова Java навіть починає з нього, з відомої програми "*Hello, World*!".

Не дивно, що у MATLAB є багато команд для роботи з текстом:

strcat, *horzcat*, *vertcat*, *cat*

та інші. Вони роблять так звану конкатенацію (об'єднання) текстових рядків у єдиний рядок, у стовпчик, у масив (див. *help*!). Нам достатньо робити це найпростішим чином за допомогою квадратних дужок:

```
>> x='What  to say?';   y='Nothin to say';
>> S_row=[x, y];        S_col=[x; y];
```

Остання операція здійсненна лише за умови *length*(*x*)=*length*(*y*). Зрозуміло, що текстова строка у MATLAB – вектор з кількох символів; з неї можемо відокремити будь-яке слово чи літеру,

```
>> Word1=x(1 : 4);
```

А от поєднати (конкатенувати) текст і числа неможливо, спробуйте:

>> *x*1=[*x*, 25, *y*],

Отримаєте якусь несинитницю. Аби це стало можливим, використайте попереднє *перетворення числа у текст*

>> *x*1=[*x*, ***num2str***(25), *y*];

(перетворення одного типу даних в інший робиться часто, і тому цій операції надали окрему назву *casting*, *кастінг*; буває, що при цьому частина інформації губиться). Інформація не губиться, якщо число *n* перетворюється в еквівалентний текст '*n*'. Для цього в MATLAB створено команди *str*2*num*( ) та *num*2*str*( ). Покажемо їх застосування.

6.2.1. Інтерактивні рядки, тобто такі, що змінюються у ході діалога користувача з програмою або у ході її виконання. У програмах *Helicopt1* та *HelicopterDialog* (див. 5.1.1, 5.1.2) можна спостерігати, що заголовок графічного вікна рахує кількість обертів. Це реалізовано наступним алгоритмом.

Виконуючи цикл, програма кожного разу збільшує змінну циклу *Fi* на *pi*/100; кожний оберт гелікоптера – це 2**pi*, тобто відбувається кожні (2**pi*)/(*pi*/100)=200 циклів. Таким чином, змінна *n*=*n*+1 нібито слідкує за обертами: коли *n*=200 – відбувся один оберт, *n*=400 – другий оберт і т.д. Тепер величина *mod*(*n*,200) дає решту від целочисленного ділення *n* на 200, тобто *mod*(1,200) або *mod*(201,200), або *mod*(401,200) дають 1; *mod*(2,200) або *mod*(202,200), або *mod*(402,200) дають 2, і т.д., а у випадках *mod*(200,200), *mod*(400,200), *mod*(800,200) і т.д. періодично отримуємо 0. Це використовуємо як ознаку чергового оберту гелікоптеру.

Як тільки *n* стає кратно 200, змінну *RotN* , що зовні циклу дорівнювала 0, збільшуємо на 1. Маємо фрагмент коду з програми 5.1.1:

```
if  mod(n,200)==0
      RotN=RotN+1;
end
```

тобто *RotN* рахує оберти. Далі використовуємо цю змінну у команді

title(['Оберт ',*num2str*(*RotN*), ' з ', *num2str*(*Nrotations*)])

Команда *title* тут застосовується до текстового рядка у квадратних дужках. В останньому частини 'Оберт' та ' з ' – незмінні, а складова *num2str*(*RotN*) змінюється з кожним новим значенням *RotN*. Ось і маємо інтерактивність! Інший приклад створення інтерактивності див. 7.1.4.

Аналогічно до команд наприкінці розділу 6.1.4, команди *ischar*( ) та *isletter*( ) видають *true*, якщо аргументом є літера, та *false* для інших аргументів.

6.2.2. Вдосконалення програм сортування. Ми помітили раніше (розділ 5.3), що програми *MyOrderM* та *MyOrderArray* получилися не дуже зручними, бо правильну відповідь вони дають лише для аргументів '*increase*' та '*row*', а неправильну для аргументів '*inCreAse*' та '*Row*' тощо. Це легко виправити.

На множині текстових даних діють MATLAB-команди *lower* та *upper*, що текстовий аргумент перетворюють на такий з, лише, маленькими (великими) літерами. Ось приклад виправленого коду:

```
switch lower(KeyWord)
        case 'row'
                ......
        case 'column'
                ......
end
```

Він зробить наші програми більш толерантними до принад користувача.

Можна було б зробити ще багато корисних програм над текстовим типом даних, та от проблема – слідкувати весь час за потрібною кількістю літер та пропусків. Виникає бажання пошукати більш зручний тип даних!

6.2.3. Яка літера більша, 'A' > 'B' чи 'A' < 'B' ? Нібито дивне питання. Та спробуємо у командному рядочку. Маємо:

```
>> 'A' > 'B',
     ans =    0
```

```
>> 'A' < 'B',
        ans =     1
```

Легко зрозуміти, що відповідь “ans=0” означає насправді “ans=***false***”, а “ans=1” – “ans=***true***”, тобто перша нерівність непідтвержена, а друга – так.

Пригадаємо, що у пам’яті комп’ютера будь-яка інформація зберігається у вигляді числових кодів, літери 'A', 'B' та інші символи у тому числі. Подивимось, чому вони дорівнюють:

```
>> double(['A','B', 'a','b', 'є','ї'])
ans =        65        66        97        98      1108      1111
```

(тобто код української '**є**' дорівнює 1108, і таке інше). Тепер, мабуть, зрозуміло, з яких причин, дійсно, 'A'<'B'. Використаємо наше “відкриття” для упорядкування текстового вектора.

6.3. Новий тип даних *cell* (комірка)

cell, *комірка* – такий тип даних, що “не звертає уваги” на вміст окремих елементів. Він вводиться аналогічно до масиву (тобто лишається прямокутною таблицею), та, на відміну, використовує фігурні дужки. Приклади:

```
>>Example1= {'Всі знають, що pi='; 3.1415; ' і часто це використовують'}
                Example1 =     'Всі знають, що pi='
                                    [3.1415]
                                ' і часто це використовують '
```

тобто в одному стовбчику розмістилися два текстових рядки

```
  >> Example1{1}
      ans =   Всі знають, що pi=
i
  >> Example1{3}
        ans =    і часто це використовують
```

і одне число *Example1*{2}=3.1415. Зверніть увагу, що довжини *length*(*Example*1{1}) і *length*(*Example*1{3}) різні!

```
  >> Example2= {'Word 1', [1.2  -3; -4  5.2]; [1; 2; 3; 4], 'Слово 2'}
      Example2 =    'Word 1'        [2x2 double]
                   [4x1 double]     'Слово 2'
```

У цій комірці окремі текстові дані розміщені разом з матрицями:

```
>> Example2{1, 2}
                ans =   1.2000    -3.0000
                       -4.0000     5.2000
>> Example2{2, 1}
                ans =    1
                         2
                         3
                         4
```

Доступ до кожного елемента – через фігурні дужки { }! Якщо ж використовувати круглі дужки, отримуємо лише загальну інформацію:

```
>> Example2(2,1)
                ans =    [4x1 double]
```

(лише інформацію про "контейнер" [4x1 *double*]). Якщо елементом комірки є масив, як у нашому випадку *Example*2{1, 2}, то його окремих елементів можна отримати через круглі дужки (вони мають бути останніми):

*Example*2{1, 2}(1, 1:*end*).

Зрозуміло, що такий тип даних винайдено для того, аби створювати складні бази даних. Далі розглянемо це на прикладі програми сортування слів.

Над цим новим типом даних працюють ще такі команди:

length(*Example*2), *size*(*Example*2), *iscell*(*Example*2)

Наступна команда інформує про зміст комірки

```
>> celldisp(Example2)
Example2{1,1} = Word
          Example2{2,1} =    1
                             2
                             3
                             4
Example2{1,2} =   1.2000  -3.0000
                 -4.0000   5.2000
Example2{2,2} =Слово 2
```

А ця команда дає певне візуальне зображення:

cellplot(*Example*2)

(рис. 6.1*а*.). Більш того, *cell* може в себе включати іншу *cell*! У наступному прикладі замість *Example*2{2,2} вставлена комірка *Example*2,

>>*Example*3= {'Word 1', [1.2 -3; -4 5.2]; [1; 2; 3; 4], ***Example2***}

Команда *celldisp* детально покаже зміст нової комірки *Example*3, а команда *cellplot* надасть її візуальне зображення, рис. 6.1*б*.

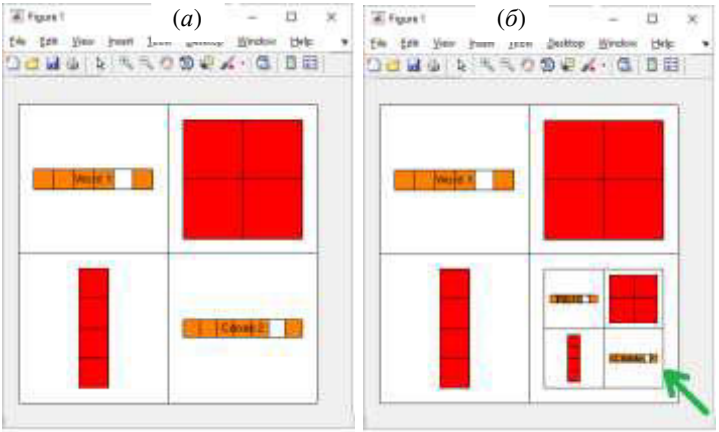

Рис. 6.1. Візуальне представлення комірок *Example*2 (*а*) та *Example*3 (*б*) командою ***cellplot***.

6.3.1. Програма сортування слів використовує комірки (cell) та корисна сама по собі. Завдання ставиться таким чином: зробити програму *MyWordOrder*, що отримує вектор із слів (допустимо, імена студентів групи), а видає тіж ж слова, упорядковані або за алфавітом, або протилежним чином, як то "замовляє" ключове слово.

План дій: аналогічно до програми *MyOrder* для чисел, зробимо кілька допоміжних програм, на підставі яких розв'яжемо потім поставлене завдання. Це такі програми:

1. Програма *MinWord*, що приймає вектор-рядок або вектор-стовпчик *List* зі слів, та видає *найменше* з них, тобто таке, яке починається з *найменшої літери*, та його номер у

List. Пропонуємо зробити таку програму самостійно, і лише потім подивитися у Додатку 1. Ось зразок, як вона працює:

```
>> L={'Sasha','Masha','Alex','Петя','Anna','Boris','Євгенія','Andrey'};
>> [x, I]=MinWord(L)
                x =Alex,   I =    3
```

Можна також брати стовпчик *L*={'Sasha'**;** . . . **;**'Andrey'}. Зверніть увагу: програма працює також із слов'янськими літерами, та вважає їх "більшими" за латинські. При цьому українські літери "менші" за російські☺.

Аналогічно – потрібна програма *MaxWord*; вважаємо, що ви її також зробили і можете використовувати надалі.

2. Після цього робимо програму *MyWordOrder*(*X*,*WordOrder*), що повертає ті ж самі слова за алфавітом, якщо *WordOrder='alphabetic'*, та у протилежному порядку, якщо *WordOrder='nonalphabetic'*. Надаємо:

Лістінг *MyWordOrder.m*

```
function V=MyWordOrder(List, Order)
%Програма працює із коміркою слів List,
% та повертає комірку V з тими ж словами, але упорядкованими
%за алфавітом, якщо Order='alphabetic',
% та у протилежному порядку, якщо Order='nonalphabetic'.
% В інших випадках просить уточнити задачу
%та видає List незмінним.
% Приклад використання:
%>>x={'Sasha','Иммануил','Masha','Alex','Петя','Anna','Аня','Boris','Rita'};
% >>y=MyWordOrder(x,'alphabetic')
% >>y=MyWordOrder(x,'Nonalphabet')
%-----------------------------------------------
%% Copiright Katerina Khavrai 3.03.16
Order=upper(Order); Order=Order(1);%враховуєму 1шу літеру!
L=length(List); V=List; V1=List;
MxWord='яяя';
MnWord='@@';
switch Order
    case 'A'
        for i=1:L
            [x1,I1]=MinWord(V1);
            V1{I1}=MxWord;
            V{i}=x1;
        end
```

```
    case 'N'
       for i=1:L
          [x1,I1]=MaxWord(V1);
          V{i}=x1;
          V1{I1}=MnWord;
       end
   otherwise
          disp('Перевірте Ваше завдання!')
          disp('Параметр Order має бути)
          disp(' або alphabetic, або nonalphabetic у лапках.')
end
```

Пояснимо алгоритм, який тут використано. Визначили кількість слів *L* в комірці *List*, створили комірки тимчасову *V*1=*List* та для майбутнього результату *V*. У циклах *L* разів переглядаємо комірку *V*1 та обираємо найменше слово, якщо *Order*='alphabetic' (найбільше, якщо *Order*='nonalphabetic') та записуємо його черговим у відповідь *V*; аби при наступному перегляді *V*1 не обрати вже знайдене найменше (найбільше) слово, замінюємо його у цій комірці на *MxWord* – таке, що не за яких обставин не буде найменшим (*MnWord*, найбільшим). Такими обрано слова *MxWord*='яяя' та *MnWord*='@@', та можна якісь інші. У *Help*-частині надано приклад використання програми. Особливість програми у тому, що завдання ('alphabetic' або 'nonalphabetic') можна скоротити аж до однієї літери за рахунок використання команди ***upper*** (або ***lower***).

Тут можна використати й інший алгоритм: знайшовши “найменше” або “найбільше” слово на кожному циклі, можна не замінювати його на якесь інше, а просто викреслити з комірки *V*1,

*V*1=**[***V*1(1:*I*1-1), *V1*(*I*1+1:*end*)**]**;

Однак, такий алгоритм праіцює для комірок-рядків, та має бути іншим для комірок-стовпчиків.

У програми є недолік: якщо два слова мають однакову першу літеру, то друга ніяк не врахована у відповіді! Це можна виправити. Крім того, для програм такої складності доцільно використовувати прості GUI, про які буде мова в розділі 8.3.

6.4. База даних з *cell*

Бази даних (БД), або, точніше, *системи керування базами даних* (СКБД), такі, як **Oracle**, **MySQL** тощо – велике досягнення сьогоднішньої комп'ютерної науки. Тут, у MATLAB, спробуємо відтворити деякі з притаманних ним рис.

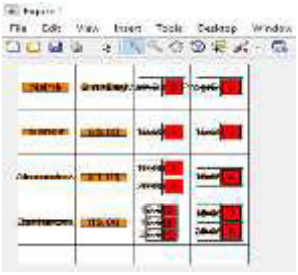

Рис. 6.2 . Візуальне представлення бази даних *MyGroup*, р. 6.6.

6.4.1. БД *МояГрупа*. Розглянемо задачу: створити у MATLAB колекцію даних про вашу студентську групу – з листом студентів, датами їх народження, з їх оцінками з головних дисциплін. Нехай ця колекція називається *MyGroup*; це буде комірка (cell), де, як ми вже знаємо, можна зберігати різноманітні дані. Єдине, за чим треба слідкувати – щоб вона була прямокутною.

Утворимо комірку 10*4 та введемо перший рядок, який буде заголовком, лише форматом наступних рядків:

```
>>MyGroup=cell(10,4)
>>MyGroup(1,:)={'Name','BirthDay',{'MathDate',0},{'ProgrDate',0}}
```

(тобто спочатку буде прізвище студента 'Name', його день народження, а далі – комірка з датами його відповідей з математики та масив отриманих оцінок 'MathDate',[0]; таке саме – щодо програмування 'ProgrDate',[0]). Ось як ввести дані про двох перших студентів:

```
>>I=2;MyGroup(I,:)={'Ivanov','20.05', {'10.09',5}, {'15/09',3}}
>>I=I+1;MyGroup(I,:)={'Alexandrov','21.01',{'10.09',4;'20.09',3},{'15/09',4}}
```

Останнє означає: студент Alexandrov народився 21-го січня, з математики має оцінки 4 за 10.09 та 3 за 20.09. Аналогічно щодо програмування. Так само додаєм до бази наступного студента

```
>>I=I+1;MyGroup(I,:)={'Zakharova','15.06',{'10.09',5;'15.09',4;'25.09',4},
                                          {'15/09',3;'25.09', 5}}
```

і так далі – утворюємо БД для групи. Команда *cellplot* засвідчує візуально, що відповідні відомості "лежать" на її полицях.

6.4.2. БД *МояБібліотека.* Ще один приклад – проста база даних та програма роботи з нею, створеною студенткою Д.Малініною. Спочатку створимо саму БД про назви книжок, їх авторів та рік створення:

```
>> Name={'Siddhartha'; '451 Fahrenheit'; 'The Man,who laughs'};
>> Author={'Hermann Hesse'; 'Ray Bradbury'; 'Victor Hugo'};
>> Year={'1922'; '1953'; '1869'};
>> DashaLibrary={Name, Author, Year}
```

Результуюча комірка у кожному з трьох рядочків містить вказану інформацію. Тепер вміщуємо програму роботи з БД:

Лістінг *MyLibrary.m*

```
function MyLibrary
% Ця маленька програма працює з коміркою "DashaLibrary".
% Після запуску
% >> MyLibrary
% маємо обрати одну з опцій, аби отримати інформацію.
% Copyright of Malinina, 17.03.2015
load DashaLibrary    % БД має бути збережена у пам'яті!
Prompt={'Аби отримати ще інформацію з Library,';. . .
        'прохання запустити знов:'; . . .
        '       MyLibrary   '};
choice=menu({'WELCOME!'; 'Choose any options '}, . . .
                    'Books','Authors','Years of publishing');
if choice==1
    Books=DashaLibrary{1};
    table(Books)
    disp(Prompt)
elseif choice==2
    Authors=DashaLibrary{2};
    table(Authors)
    disp(Prompt)
else
    Years=DashaLibrary{3};
    table(Years)
    disp(Prompt)
end
```

Рис. 6.3. GUI-програма *MyLibrary* управління БД, розділ 8.

Будьте уважні: труднощі роботи з базами даних на основі комірок пов'язані з адресацією до її елементів. Це складна річ! Якщо вже мова про бази даних, то треба навчитися їх зберігати і зчитувати при кожному новому запуску MATLAB. Про це буде мова у розділі 6.8. Доцільно вже зараз використати також прості елементи GUI, про які йдеться у розділі 8, див. рис. 6.3.

6.5. Новий тип даних структура (*struct*)

Тип даних ***struct*** (*структура*) MATLAB також позичена з інших сучасних мов. Це такий формат запису та збереження даних у пам'яті обчислювального середовища, коли найрізноманітніші дані зберігаються як окремі ***поля*** якоїсь змінної, і ці поля відокремлюються крапкою від імені цієї змінної: *Sol.x*, *Sol.Matrix*, *Sol.MyCell* тощо. Ось як їх можна ввести:

>>*Sol.x*='a Sentence'; *Sol.Matrix*=[1 2 3; 4 5 6]; *Sol.Cell*=*Example*3;

Отримати щойно введені дані можна, вказавши ім'я структури та, за крапкою, поле:

```
>> Sol.Matrix
ans =     1    2    3
          4    5    6
```

Якщо, припустимо, загрузили[6] якусь структуру з пам'яті, та забули її поля – узнати можна командою

```
>> fieldnames(Sol)
ans =    'x'
         'Matrix'
         'Cell'
```

[6] Читання та збереження даних – див. 6.7.

MATLAB широко використовує структури для збереження результатів складних математичних задач. Припустимо, треба розв'язати систему алгебраїчних рівнянь

$$\begin{cases} x^2 + xy + y = 3 \\ x^2 - 4y + 3 = 0 \end{cases}.$$

Це можна зробити так[7]:

```
>> SOL=solve('x^2 + x*y + y -3',  'x^2 - 4*y + 3')
   SOL =    x: [2x1 sym]
            y: [2x1 sym]
```

Характер відповіді показує, что результатом є структура. Тепер легко отримати значення її полів (власне розв'язок):

```
>> SOL.x
        ans =    1
                -3
```

Аналогічно, для вектора *SOL.y* можна отримати кожну з компонетів *SOL.y*(2), тощо.

```
>> SOL.y
```

Надамо ще використання MATLAB для розв'язування системи диференціальних рівнянь (на другому курсі буде до нагоди). Уявимо, треба розв'язати систему звичайних диференціальних рівнянь

$$\begin{cases} f'(t) = f + g \\ g'(t) = -f + g \end{cases}.$$

Виконуємо команди:

```
>> syms f(t)  g(t)
>> Sol=dsolve( diff(f)= =f+g, diff(g)= = -f+g )
        Sol =    g: [1x1 sym]
                 f: [1x1 sym]
```

З структури, що обчислена[8], маємо розв'язок:

```
>> f=Sol.f
    f =  C2*exp(t)*cos(t) + C1*exp(t)*sin(t)
```

---

[7] А можна – викликавши Symbolic Toolbox >>*syms x* [1].

[8] Тут *dsolve*( ) – MATLAB-програма для розв'язку системи ЗДР, що записані у дужках у певному форматі; *diff*(*f*) – відповідник до $f'(t)$

```
>> g=Sol.g
    g =  C1*exp(t)*cos(t) - C2*exp(t)*sin(t)
```

Він залежить від двох сталих *C*1 та *C*2.

Головне призначення структур у комп'ютерних науках – створення різноманітних *баз даних*. Хоча для цього вже є спеціалізовані програми[9] (Бази Даних, системи управління БД), доцільно навчитися їх створювати і в MATLAB.

6.6. База даних із *struct*

Наведемо приклад використання структури для утворення БД "Фрукти". Утворюємо БД:

```
>>i=1;Fruits(i).Name='Apricot';Fruits(i).Country='Armenia';. . .
                                                    Fruits(i).Price=10;
>>i=i+1;Fruits(i).Name='Apple';Fruits(i).Country='Ukraine'; . . .
                                                     Fruits(i).Price=5;
>>i=i+1;Fruits(i).Name='Strowberr';Fruits(i).Country='Serbia'; . . .
                                                    Fruits(i).Price=40;
>>i=i+1;Fruits(i).Name='Lemon';Fruits(i).Country='Georgia';
                                                    Fruits(i).Price=30;
```

Створено 4 "шафи" з полями:

```
>> Fruits
    Fruits =    1x4 struct array with fields:
      Name
      Country
      Price
```

Для такого типу даних можна визначити розмірність

```
>> size(Fruits)
        ans =     1     4
```

Можна узнати, що лежить у "шафі" на якійсь поличці:

```
>> Fruits(1:end).Name
            ans =Apricot
            ans =Apple
            ans =Strowberry
            ans =Lemon
```

та з якої країни

```
>> Fruits(1:end).Country
```

[9] https://uk.wikipedia.org/wiki/База_даних

ans =Armenia
ans =Ukraine
ans =Serbia
ans =Georgia

Доцільно зберігати БД, що утворено.

Виникає питання: студенти мають вивчати дані тих типів, що запропоновані відомими авторами, чи мають право створювати власні? Безумовно, мають право! Пропонуємо задачі 2 – 4 розділу 6.10.

6.7. Збереження даних

Робота з БД обов'язково має бути пов'язана із збереженням БД, що утворена, із її загрузкою заново після нового запуску комп'ютера, з розширенням БД або вилученням одного чи більше її компонентів, знову з її збереженням, і так далі. Познайомимо читачів з командами ***save*** та ***load***.

Припустимо, що створивши БД *MyGroup* у розділі 6.4.1, ми її зберігли командою

>> ***save*** *MyGroup* (6.1)

Тоді, після нового запуску MATLAB маємо спочатку загрузити цю БД у вигляді комірки

>> *load MyGroup* (6.2)

і лише після цього можемо продовжувати роботу з нею, наприклад – її розширювати:

>>*n*=5; *MyGroup*(*n*,:)={'Azarov','12.11', {'10.09',2}, {'15/09',44}};
>>*n*=*n*+1; MyGroup(*n*,:)={'Komarova','10.01', {'10.09',4}, {'15/09',5}};

і так далі. Завершивши роботу – зберігаємо її. Можно також зберігти БД у певний бінарний файл:

>> ***save***('MyGroupFile','MyGroup') (6.3)

Утворився файл "*MyGroupFile*.***mat***", що зберігає комірку *MyGroup*. Якщо через деякий час вона стане потрібною, то

>> ***load***('MyGroupFile') (6.4)

і комірка *MyGroup* з'явиться для подальшої роботи з нею. Таким чином, маємо дві форми збереження даних: (6.1) та (6.3) з відповідними для них загрузками (6.2) та (6.4).

Аналогічно – робота з БД у вигляді структур.

```
>> L=length(Fruits)
>>L=L+1;Fruits(L).Name='Orang';Fruits(L).Country='Urugw';Fruits(L).Price=23;
```

Ось головне, що дозволить вам успішно виконати Лабораторну роботу № 9.

6.8. Обробка помилок; *try-catch-end* блок

Два наступних підрозділи при першому читанні підручника можна пропустити. Цей матеріал трохи "випереджає події", та стане до нагоди у подальших розділах та програмах.

Коли ми говоримо про мови високого рівня, то тут йдеться насправді не лише про мову, а й також про транслятор або компілятор, які програму "перекладають" комп'ютеру, або "середовищу", як у випадку з MATLAB. Останні мають бути ще "розумніше", аніж програми, бо мають додатково обслуговувати програміста. Підказувати йому про помилки (ви це бачили у MATLAB), якось себе "розумно поводити" у випадках, коли такі помилки все ж-таки виникають, створювати інші зручності.

Якщо у роботі програми виникають помилки, то найкраще – їх передбачити і врахувати, знайти й виправити. Наприклад, якщо виникає ділення на нуль в операції *X/Y*, то у програмі слід було передбачити, що означає *Y*=0 та яку інформацію надати користувачеві у такому випадку. Та буває, що програмісту немає часу з цим працювати і він бажає відкласти на потім аналіз цього випадку. І буває ще складніше, коли проблема виникає не з вини програміста, а з дій користувача. Такі випадки усунути неможливо, та програміст все одне має таке передбачати! Ось про ці "неусувні" помилки і йдеться у даному розділі.

6.8.1. Уявіть собі, навіть коли все гаразд – MATLAB напоготові "ловити" (*catch*) можливі помилки. Цьому слугує

системна змінна *lasterror*. Зверніть увагу, MATLAB не питає, чому вона дорівнює:

```
>> Question1=lasterror
Question1 =
            message: '  '
            identifier: '  '
            stack: [0x1 struct]
```

Виявляється, що *Question*1 – структура з полями (розділ 6.5). Та поки що всі її поля, наприклад *Question*1.*message*, пусті. Навмисно зробимо якусь помилку:

```
>> x=2/0 % ділення на нуль
        x =    Inf
>> Question2=lasterror
Question2 =
    message: 'Error: Expression or statement is incomplete or incorrect.'
    identifier: 'MATLAB:m_unexpected_sep'
      stack: [0x1 struct]
```

Тепер бачимо, що *Question*2.*message* та *Question*2.*identifier* отримали якісь текстові значення. Останні несуть певну інформацію про помилку, що виникла.

6.8.2. Припустимо, ми передбачаємо як правильні, так і неправильні дії користувача. Якщо все робиться правильно і помилки не виникають, такі оператори ми залишимо у блоці ***try***; якщо ж виникає помилка – тоді відповідні дії доручаються операторам з блоку ***catch***. Ось як це має виглядати у програмі:

```
try
    Оператори нормальної роботи;
catch
    Оператори на випадок аварії у попередньому блоці;
end
```

Наведемо приклад такої програми:

Лістінг "*TryTest*1.m"

```
function TryTest1(Arg)
%Приклад TRY-END оператора
%Copyright Ye.Gayev, June 2016.
try
```

```
    ezplot(Arg)
    %Все ОК, якщо текст Arg – правильна функція від 'x' в апострофах
catch
    disp(' "Arg" має бути правильною функцією у лапках!')
    title('Не можу побудувати! Введіть правильно.')
   F=input('Введіть функцію від x у лапках, F= ');
   figure(1),   ezplot(F)
end
```

Якщо все гаразд, наприклад,

```
>> TryTest1('sin(x)/x')
```

маємо графік функції $F = \frac{\sin(x)}{x}$ у діапазоні $x \in [-2\pi, 2\pi]$.

Якщо ж аргумент з помилкою, як тут

```
>> TryTest1('sin(x)x')
```

то програма відмовляється працювати. Хоча помилки червоним не видає, як зазвичай, та "каже": *"Arg" має бути правильною функцією у лапках!.* І пропонує її ввести наново.

Програма стала працювати "розумніше". Та може бути ще кращою! Її недоліком є те, що у випадку, коли користувач і вдруге ввів текст з помилкою, або забув про апострофи – програма "не витримує" та "лається" червоним… Покращити можна за допомогою встроєного блока *try - end*.

6.8.3. Як і раніше, дозволяються, і є корисними, встроєні блоки *try-catch-end*.

Лістінг *TryTest2.m*

```
function TryTest2(Arg)
%Приклад TRY-END оператора із внутрішнім блоком
%та оператором continue
%Copyright Ye.Gayev, June 2016.
Condition=true;
%%  I=0;
%%while Condition && (I<=3)
%%    I=I+1;
while Condition
   try
        ezplot(Arg)
      %Все ОК, якщо текст Arg – правильна функція від 'x'
```

```
        Condition=false;
    catch
        try
            disp(' "Arg" має бути правильною функцією у
лапках!')
            title('Не можу побудувати! Введіть правильно.')
            F=input('Введіть функцію від x, F= ');
            figure(1),   ezplot(F)
            Condition=false;
        catch
            continue %повторення циклу!
        end %кінець внутрішнього блоку try-catch
    end %кінець зовнішнього блоку try-catch
end

%% if (I>3)
%%    error('Скільки можна повторювати?')
%% end
```

Така програма має “нескінченне терпіння”, бо пропонує виправити текст вводу до нескінченності. Якщо ж закоментувати той оператор ***while***, що існує, та розкоментувати рядочки з %%, то програмі “вистачить терпіння” лише до ***I*** =4.

Звернемо увагу також на те, що в останньому прикладі вперше використано корисний оператор ***continue***. Дослідіть його самостійно!

6.9. Поліморфізм програм; аргументи *VarArgIn* і *VarArgOut*

Ви звернули увагу, що MATLAB часто-густо робив те, що не дозволено нам, “простим програмістам”? Припустимо, MATLAB майже одне й те саме завдання виконати кількома способами:

```
>> x=0:.1:2; y=sqrt(x);
>> plot(y) % Перший спосіб
>> plot(x,y) % Другий спосіб
>> plot(x,y,'s:') % Третій спосіб
>> hP=plot(x,y,'s:');  % Четвертий спосіб
>> plot(x,y,'s:r','MarkerSize', 10)  % П’ятий спосіб
```

```
>> plot(x,y,'s:r', x,y.^2, 'go--')  % Шостий спосіб
>> plot(x,y,'s:r', x,y.^2, 'go--', x,sin(y), 'm*-')  % Сьомий спосіб
```

тобто *сігнатура*[10] виклику методу може включати різну кількість вхідних і вихідних аргументів, навіть їх порядок у списку.

Погодьтеся, це зручно для користувачів. Та звідки ж такі "інтелектуальні здібності" в MATLAB-програмах? Точніше: як це *компілятор* або *інтерпретатор* здогадуються, що саме хоче від них користувач, програміст?

В MATLAB-середовищі це може бути зроблено двома способами. Другий спосіб будемо вивчати у наступному підручнику з мови Java; парадигма об'єктно-орієнтованого програмування, що у ній використовується, з самого початку заклала таку вимогу *гнучкості*, *поліморфізму*[11]. MATLAB останніх версій позичив цю якість з найсучасніших мов програмування.

Тут розглянемо лише те, як MATLAB врахувує це у процедурній парадигмі програмування. Для цього слова *varargin*, *varargout*, *nargin*, *nargout* оголошуються ключовими, тобто такими, що мають особливе значення. (Їх зміст стане зрозуміліше, і запам'ятати їх легше, якщо записати їх *VarArgIn*, *VarArgOut* – variable arguments internal, variable arguments out, тобто змінна кількість внутрішніх та зовнішніх аргументів, та *NargIn*, *NargOut* – кількість аргументів *N* внутрішніх та *N* зовнішніх). І MATLAB-функція має спочатку визначити ці параметри, та прийняти ті чи інші рішення в залежності від них. Покажемо це на прикладі.

Раніше ми розробили кілька модифікацій програми сортування, а саме *MyOrder*, *MyOrderArray*, *MyOrderM* та *MyWordOrder*. Спробуємо на разі зробити єдину програму

---

[10] "В языке программирования Java *сигнатуру метода* составляет его имя и последовательность типов параметров; тип значения в сигнатуре не участвует", [https://ru.wikipedia.org/wiki/API].

[11] Цей термін складається з двох грецьких слів: *полі* (багато) і *морфос* (форма) – тобто такий, що *існує у багатьох формах*.

MyCleverOrder, яка буде “сама вирішувати”, що саме їх робити. Загальний план такий: якщо запустити *MyCleverOrder* без аргументів взагалі – вона надрукує Help (коротку інструкцію з користування нею); у загальному випадку кількох аргументів програма має спочатку шукати серед аргументів “предмет роботи” – числовий масив або комірку із слів. Якщо такого аргумента немає – видає пораду користувачеві і зупиняється; якщо знайшла “предмет роботи” – шукає далі, що саме з ним робити. Перш за все, одне з завдань “Increase” або “Decrease” для числового масива, “Alphabetic” або “Non-alphabetic” у випадку текстової комірки. Не знайшла – приймає за умовчанням “Decrease” або “Alphabetic”. Далі слід шукати серед аргументів ключові слова “Rows” або “Columns”: якщо знайшли – таке буде завдання; не знайшли – буде “Rows” за умовчанням. Коли ж із завданням визначилися повністю (*X* перший аргумент,) – викликаємо програму *MyOrderArray* для числового масива; а програму *MyWordOrder* треба узагальнити на випадок третього аргументу “Rows” або “Columns”.

Як бачимо, доволі складна і розвинена логіка роботи програми. На рис. 6.4 надано її блок-схему.

6.10. КОНТРОЛЬНІ ПИТАННЯ ДО РОЗДІЛУ 6

1. Що ми маємо на увазі, коли кажемо про *тип даних*? Обгрунтуйте це на прикладах кількох типів даних, які ви знаєте.
2. У якому сенсі матриці (масиви) є основним типом даних для MATLAB?
3. У розділі 1.7.7 першої частини ми вели мову про многочлени. Охарактеризуйте многочлени саме як тип даних. Які ознаки слід приймати до уваги?
4. Що ви можете сказати про символьну математику, символьний тип даних?
5. Які MATLAB-команди визначені для текстового типу даних?

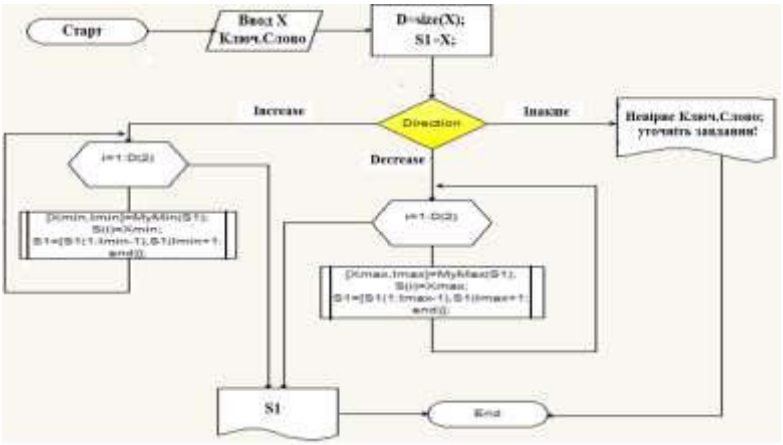

Рис. 6.4. Блок-схема “розумного” сортування, яке залежить від ключового слова *Increase* або *Decrease*

6. Які дії визначені для типу даних ***boolean***? Чому дорівнює *true*+2, *false*+2, 2**true* в MATLAB та в інших алгоритмічних мовах? Чому дорівнює (*true* & *false*), *true*|*false*, *true* & *true*?
7. Як в MATLAB організувати інтерактивні вивод та надпис на рисунку?
8. Як конвертувати великі літери у малі, і навпаки?
9. Що таке ***cell***, як використовувати такий тип даних?
10. Що можете сказати про ***struct***? Як його використовувати?
11. Як узнати, що вміщує *cell* або *struct*?
12. Чому саме останні типи даних використано у програмі сортування слів *MyWordOrder*, як працює алгоритм сортування?
13. Які стандартні блоки (елементи) використовують для зображення блок-схем алгоритмів?
14. Як зображають на блок-схемах початок та закінчення алгоритму?
15. Як відрізняються лінійні та розгалужені блок-схеми? Чому виникають розгалуження у блок-схемах алгоритмів?

16. Як зображають на блок-схемах ввід початкової інформації та вивід результатів? Якими, у якій формі можуть бути ци результати?
17. Як зображають логічні дії на блок-схемах? Скільки може бути “вхідних стрілочок” до блоку логіки, скільки “стрілочок” може виходити? Що надписують?
18. Як на блок-схемах зображають циклічне виконання певних дій? Які два варіанти існують?
19. Як на блок-схемах зображають підпрограми, тобто такі частини алгоритму, що самі потрибують доволі складної блок-схеми?
20. Які ви знаєте парадигми програмування, у чому вони полягають?

Якщо не змогли відповісти на більшість питань – радимо проробити все з початку

Закріпленню матеріалу даного розділу сприяє також Лабораторна робота № 9.

6.11. ЗАДАЧІ ДО РОЗДІЛУ 6

1. Створити у MATLAB новий тип даних, який назвемо *множина* (*Menge*[12] [20]). Це є будь-який вектор, що складається з визначеної кількості елементів (або чисел, або текстових), що знаходяться усі разом у будь-якому порядку. Припустимо: *CityMenge*={'NewYork', 'Kyiv', 'Paris', 'Madrid', 'Berlin'}. Створити операції (програми), що визначені для такого типу даних. Зразки: *доступ до елементу City*3=*CityMenge*{3}; *розмір множини L*=*length*(*Menge*). Більш конкретно: *Shift*1(*Menge*) зміщує всі елементи на 1 вперед, тобто другий у ньому стає першим, третій – другим, і т.д., нарешті на місце останнього стає перший; *Shift*2(*Menge*) робить те саме у

[12]https://de.wikipedia.org/wiki/Menge_(Datenstruktur)– використовуємо німецьку назву таких даних замість англійської ***set***, бо останнє позначає Java команду “встановити”.

зворотньому напрямку, тобто на місце першого стає останній, другого – перший, і т.д.

2. Для новоствореного типу даних запрограмувати операції *Permutation* (повертає вектор тих саме елементів, що розміщені випадково, кожний елемент зустрічається один раз) та *Combination* (вектор з тих елементів, випадково розміщених, кожний елемент зустрічається випадково від 0 разів, до *k* разів).

3. Бажано також створити операцію (команду) *isMenge*(*M*) на зразок *isnumeric*( ), *iscell*( ) та *isstruct*() MATLAB, що повертає *true* (1), якщо *Menge* є множина, та *false* (0) якщо не є такою. Можливо, зроблене в 1 – 2 порибує модифікації.

4. Розробіть програми для "таємних перемовин", тобто: ви надсилаєте другові лист, який "кодуєте". Друг, знаючи принцип кодування, може його "розкодувати" та прочитати. Пропонуємо простий *алгоритм кодування*: програма читає ваш текст, кожний символ перетворює у числовий код та додає, припустимо, 1. Отримане число перетворюється у символ. Результатом і є "закодований лист".

5. Зробіть попередню програму "більш секретною". Задля цього стала, що додається, може кожного разу змінюватись. Проблема тепер у тому, аби її якимось чином повідомляти разом з листом, аби його можна було розкодувати.

Сподіваємось, Ви розв'язали більшість задач.
Вітаємо! Можете рухатись далі.

7. До розумних комп'ютерів: Складні алгоритми

Майже немає різниці, якою алгоритмічною мовою розмовляти з комп'ютером. Головне – ті ***алгоритми***, на які маємо його оргінізувати, тобто послідовність дій та можливих логічних переходів. Ми вивчили вже доволі багато алгоритмів: алгоритми Крамера та Гауса для СЛАР, обчислення визначника методом еквівалентних перетворень (методом Гауса), алгоритм пошуку найменшого (найбільшого) елемента числового масиву (або такого "із слів"), бульбашковий алгоритм пересортування числового масиву (такого ж "із слів"), алгоритми чисельного диференціювання та інтегрування. Тут продовжимо знайомство з алгоритмами, з самими відомими і фундаментально-важливими з них [15–17].

7.1. Ітерації. Програма Galaxy

Iteration (*ітерація*) у буквальному перекладі з англійської означає *повторення*. Ми можемо повторювати якусь низку операцій визначену кількість (за допомогою структури *for–end*) або невизначену кількість разів (*while–end*). Здавалося б, простий алгоритм, рис. 7.1. Та за ним – величезна купа застосувань, що відрізняються математичним та фізичним змістом!

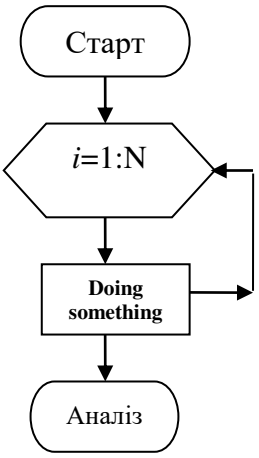

Рис. 7.1. Загальна блок-схема ітерацій

7.1.1. Команда *fzero*( ) означає *знайти нуль*. Її вже вивчали раніше в [1]; використовувати можна у такому загальному форматі:

>>[*x*, *fval*, *exitflag*, *output*]=
fzero('*cos*(*x*)-*x*/5',10)

де в апострофах – конкретне рівняння, що підлягає розв'язку. Та яку роль грає x_0=10, "*початкове наближення*"? Структура *output* повідомляє, зокрема, скільки *iterations*, повторень, було зроблено. Самостійно дослідіть, як кількість ітерацій залежить від x_0!

7.1.2. Метод ітерацій. Спробуємо самі навчитися розв'язувати рівняння методом ітерацій. Якщо дано рівняння виду $F(x)-x=0$, то перепишемо його в еквівалентному вигляді

$$x=F(x). \tag{7.1}$$

За його розв'язок (корінь) вважають таке число x_*, що, якщо його підставити у (7.1), отримаємо тотожність $x_*=F(x_*)$. Складність у тому, як його знайти. Беремо навмання якесь $x=x_0$ та підставляємо його у праву частину (7.1), маємо $x_1=F(x_0)$; та навряд чи вгадали корінь, і тому $x_1\neq x_0$. Тим не менше обчислюємо наступне значення $x_2=F(x_1)$; скоріше за все, знову, $x_2\neq x_1$, і тому x_1 також не може бути коренем (7.1). Так повторюємо багато разів, *ітеруємо*: $x_n=F(x_{n-1})$, $n=3,4,5,....$ Математики стверджують таке:

Теорема 1: *Якщо* послідовність ітерацій $x_1,x_2,x_3,...,x_n$ *збігається* (тобто існує $\lim\limits_{n\to\infty}x_n$), то вона збігається саме до кореня x_*, тобто $\lim\limits_{n\to\infty}x_n=x_*$.

Довести це зовсім легко, та обмежимося лише демонстрацією на прикладах. Ітеруючи наведене раніше рівняння $x=\dfrac{\cos x}{5}$, маємо:

```
>> x0=20; x1=cos(x0)/5, x0=x1;
                x1 =   0.0816
>> x1=cos(x0)/5, x0=x1;
                x1 =   0.1993
>> x1=cos(x0)/5, x0=x1;
                x1 =   0.1960                    (7.2)
```

```
>> x1=cos(x0)/5, x0=x1;
                x1 =   0.1962
>> x1=cos(x0)/5, x0=x1;
                x1 =   0.1962 .
```

Бачимо, що чергові значеня $x1$ спочатку "коливаються", та після четвертої ітерації результати більше не змінюються. Тобто знайдено розв'язок x_*=0.1962! Спробуйте інше початкове значення $x0$: послідовність ітерацій так само швидко "збігається" до x_*=0.1962.

Наступний експеримент проведіть з рівнянням

$$5\cos x - x = 0\,. \tag{7.3}$$

Це буде випадок, коли ітерації **не збігаються**! Від чого залежить збіжність? Математика дає відповідь:

Теорема 2: Якщо "поблизу кореня" x_* похідна правої частини за абсолютним значенням менше за 1, $|F'(x_*)|<1$, то ітерації неодмінно збігаються, можна шукати корінь методом ітерацій! (У разі першого рівняння легко бачити, що дана умова виконана).

Лабораторна робота 10 надає вам можливість дослідити метод ітерацій для розв'язання рівнянь. Для кращого її виконання та розв'язання – деякі додаткові відомості.

Чи може метод ітерацій застосовувати до рівнянь іншого вигляду, аніж (7.1)? Так, для будь-яких рівнянь! Бо будь-яке рівняння, хоча б й загального вигляду $F(x)=0$, можна записати й так[13]: $x = x - k\cdot F(x)$, тобто привести його до виду (7.1). Ітерації будуть збігатися, якщо похідна правої частини $\Phi'(x) = |1 - k\cdot F'(x)| < 1$. Тобто, оцінюємо $|F'(x)|$ та обираємо додатнє число $k < 1/|F'(x)|$. Існують й інші перетворення, що можуть будь-яку праву частину перетворити на "збіжну".

[13] Це – **еквівалентне** рівняння, якщо $k \neq 0$. Можете пояснити, що таке "*еквівалентне*"?

Порівнювати числа (7.2) не зручно. Та можливі дві графічні форми подання результатів ітерування. Розглянемо, наприклад, ітерування рівняння (7.3) за алгоритмом

```
>> i=1; x0=10; x(i)=x0; x(i+1)=5*cos(x0); x0=x(i+1);
>> i=i+1;  x(i+1)=5*cos(x0);  x0=x(i+1);
>> . . . . . .
```

Зробимо таке N=50 разів, а краще – запрограмувати у якійсь програмі *Iterations.m*, де N можна задавати різним. Її результати можно подати у формі $x = \Phi(i)$ (номер ітерації по осі абсцис) або $x_{i+1} = \Phi(x_i)$ (наступна ітерація у залежності від попередньої):

```
>> figure(1), plot(x,'o-'),
>> xlabel('Номер ітерації'), ylabel('Ітероване значення')
>> figure(2),  plot(x(1 : end-1),  x(2 : end),'o-')
>> xlabel('Попередня ітерація'), ylabel('Наступна ітерація')
```

Останні чотирі команди утворюють два рисунка 7.2 з результатами. На першому з них по осі ординат відкладено номер ітерації, на другому – значення попередньої ітерації. Можна зробити висновок, що збіжності немає, а послідовні ітерації поводять себе нерегулярно, хаотично. Зовсім інакше виглядають результати ітерування у разі збіжності цього процесу. Зробіть такий графічний аналіз самостійно!

7.1.3. Дослідження "хаосу"; програма *Galaxy*. Що ж відбувається, коли ітерації не збігаються? Уявіть собі, що тут існують випадки, коли ми наближуємось до найсучаснішої науки, яка досліджує виникнення хаосу[14], [15,19]. І тут виявляється, що навіть у хаосі присутні певні закономірності!

Розглянемо просте квадратичне рівняння

$$x^2 - x + C = 0 \tag{7.4}$$

та ітераційну схему для нього

$$x_{i+1} = x_i^2 + C\,. \tag{7.5}$$

[14] Точніше, "детермінованого хаосу"; відчуваєте протиріччя між визначеннями *детермінований* і *хаос*?

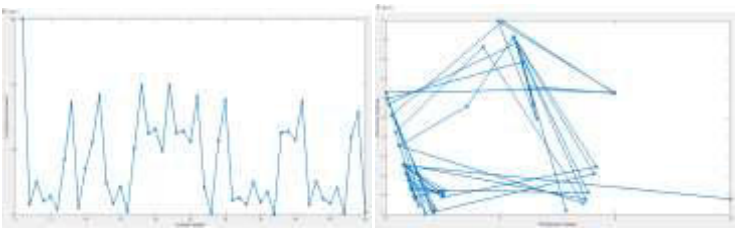

Рис. 7.2. Два різних геометричних представлення результатів ітерування рівняння (7.3). Хаос, немає порядку?

Шкільних знань достатньо, аби зрозуміти, що рівняння (7.4) за умови $D=\frac{1}{4}-C<0$ (тобто $C>\frac{1}{4}$) має лише комплексні корені. Саме вони нам потрібні у цьому дослідженні; більш того, вкористовуємо лише комплексні C, тому і корені (7.4) мають вигляд[15] $x_1=\mathrm{Re}+i\,\mathrm{Im}$ та $x_2=\mathrm{Re}-i\,\mathrm{Im}$ (спряжені комплексні числа). Будемо робити ітерації (7.5), починаючи з деякого довільного x_0, та слідкувати, як вони себе поводять. У тому числі, чи наближуються вони до коренів рівняння (7.4). Ітерації на зразок (7.2) легко (самостійно!) запрограмувати у вигляді діалогової програми, що спочатку запитує про бажані комплексні значення C та x_0 (зокрема, можна тримати останнє постійним $x_0=0$). Назвемо програму *Galaxy* (назва зрозуміла з подальшого). Послідовні ітерації за (7.5) будуємо у ній командою *plot*(*x*,'or'). Пояснемо, що комплексну точку *x*=Re+***i***Im MATLAB будує на площині як звичайну з координатами *x*=Re, *y*=Im. Пропонуємо, аби *Galaxy* утворювала такий зручний діалог у командному вікні (введені дані – жирним):

```
>> Galaxy
   'Привіт!'
   'Я покажу різноманітні фігури, що є ітераціями '
   '       X(i+1)=X(i)^2 +C.       '
   'Треба буде ввести комплексне C та початкове X0.'
```

[15] Нагадаємо, що Re називається дійсною, а Im – уявною частиною числа x.

'Приклади для X0=0:'
'C=-0.05-0.6i -- 3-промінева галактика проти годинника'
'C=0.36+0.35i -- 5 галактик що вибухають'
'C=-.3905-.5868i -- якийсь лист '
'C=.31+.40i -- 9-промінева галактика'
'C= -.37-.59i -- 8-промінева галактика '
Введіть C= -0.05-0.6i
Корені є $x1$=1.1967+0.43059***i***, $x2$=-0.19671-0.43059***i***
Введіть x0: 0

Результати експериментування з такими ітераціями показані на рис. 7.3. Бачимо, що точки, які вважали хаотичними, у випадках (А), (Б) і (В) лягають чомусь не довільно, а на гілки певних кривих, що нагадують галактику[16]. Закрутка "променів галактики" може бути за годинниковою стрілкою (Б), або проти неї (В). Червоні "сонечки" показують корені рівняння (7.4). У кожному з цих випадків чергові точки все ближче, нескінченно близько, наближаються до одного з коренів (збігаються до нього). Спробуйте "лупою" розглянути "центри галактик" – побачите такі само "галактики", що "уходять у мікросвіт". Таку властивість трохи далі, в 7.2, назвемо *фрактальною*. У випадку (Г) отримуємо 5 "галактик" (як і раніше, значення C вказано у підписі), у випадку (Е) – дві "галактики". У випадку (Д) точки випадають лише на показаний контур, що трохи нагадує лист рослини.

У випадках "галактик" (А), (Б) і (В) послідовність точок, як кажуть математики, збігається, причому – до кореня, що відповідає теоремі 2 (одного з двох, лівого). Спробуємо нові експерименти: будемо брати початкове наближення близьким до правого кореня. Нехай, для початку, $X0$ дорівнює кореню x_* з високою точністю. Теоретично, тоді $x_{i+1} = x_*^2 + C = x_*$, бо це корінь. Тобто, всі ітерації попадають у цей саме корінь. Та насправді маємо трохи інше.

[16] Враження посилиться, якщо точки ("зірки") зробити білими, а фон ("небо") чорним(**:-** Як це зробити – допомога у 8.2.

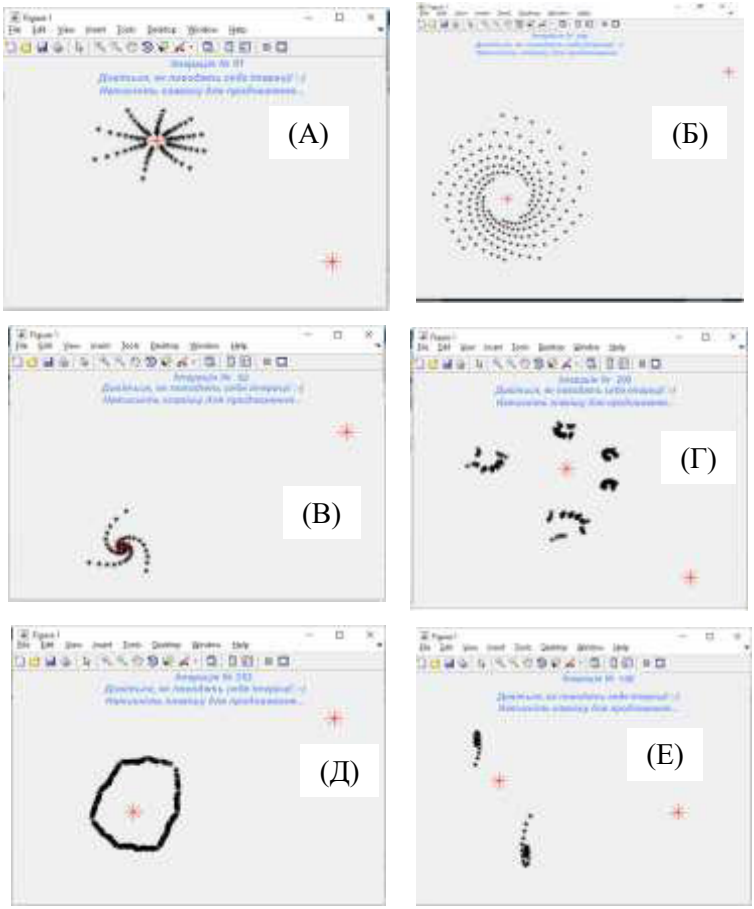

Рис. 7.3. Різні випадки поведінки ітераційної послідовності (7.5): (А) 9-промінева Галактика, C=.31+.40$\boldsymbol{i}$; (Б) 8-промінева, за годинником, C=.37-.59$\boldsymbol{i}$; (В) 3-промінева проти годинника, C=0.05-0.6$\boldsymbol{i}$; (Г) п'ять Галактик, C=0.36+0.35$\boldsymbol{i}$; (Д) "Лист", C=.3905-.5868$\boldsymbol{i}$; (Е) дві Галактики, C=-.8+.1$\boldsymbol{i}$.

Оберем C=-0.05-0.6i, випадок (В), та, на відміну, задамо початкове значення дуже близьким до правого кореня $X0$=1.197+0.4306$\boldsymbol{i}$ (у діалозі передбачено вивід значень обох). Результати показано на рис. 7.4: бачимо, що точки спочатку

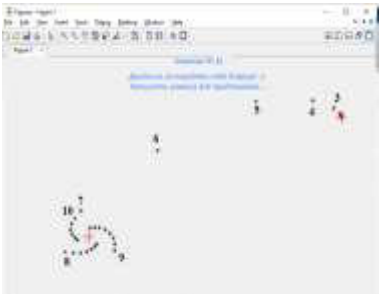

Рис. 7.4. Ітераційна послідовність у випадку (В) при $X0$, близьким до правого “сонечка”

близькі до правого “сонечка”, та покидають його і – диво! – “притягуються” до лівого; правий, здається, навіть “відштовхує” чергові значення! Та увага: такий результат дуже “нестійкий”: невеличке загрублення $X0$ (наприклад =1.2+0.431***i***) – і точки відлітають на нескінченність, збіжність процесу відсутня.

Чому точки-ітерації на рис. 7.3 випадають лише на ці лінії? Точно це ще невідомо нікому, так само як ще ніхто не може дати точний прогноз погоди, пояснити причини фінансових катастроф тощо. Більше з цього приводу можна прочитати в [15,19].

Відібрані найбільш красиві та дивні випадки. Та найтиповіша поведінка послідовності – розбіжність точок. Так, у випадку (Г) після певної ітерації точка стає більше одиниці за модулем, після чого наступні ітерації швидко “уходять на нескінченність”, що нагадує вибух галактики. Аналогічно, якщо значення С у випадку (Д) трохи збільшити, також настає “вибух”.

Тим не менше, ми виявили певні закономірності там, де спочатку очикували випадковість, хаос. Пропонуємо піти далі та дослідити питання: у випадках, де точки падають лише на N певних променів, чи випадково вони попадають на той, чи інший? Виявилося, що у багатьох випадках (візьмемо для конкретики N=5) точки № 1,6,11,… падають на один промінь, точки № 2,7,12,…, та лише ці, що дають остачу 2 при діленні на N, на інший, і так далі з кожним промінем. Довести це дозволяє програма MyColor3(i), що генерує певний колір №***i*** з набору фіксованих кольорів (задача 2 в 5.6). Радимо зробити таку програму, провести таке дослідження!

На підставі цього матеріалу ви вже можете самостійно будувати більш складні фрактали з [19].

7.1.4. Програма *SpiderWeb* (“павутина”). Красива програма, що ми позичили з книжки [15] університета Кембріджа, дозволяє ще глибше проникнути у світ ітерацій та поліпшити нашу майстерність у комп’ютерній науці. Завдання для програміста звучить таким чином: задля реалізації ітераційного процесу (7.1), іншого погляду на нього, будуємо графіки функцій $y = F(x)$ та $y = x$ (діагональ); зрозуміло, що їх перетин – корінь рівняння (7.1); на тому ж графіку будуємо точку початкового наближення $(x_0,0)$; перша ітерація дає $x_1 = F(x_0)$, можемо провести вертикальний відрізок з $(x_0,0)$ до (x_0, x_1), тобто – до кривої; друга ітерація дає точку (x_1, x_2) – проводимо горизонтальну лінію до неї від (x_0, x_1), тобто до діагоналі; третя ітерація дає змогу перейти від діагоналі до точки (x_1, x_2) – вертикаль до $y = F(x)$, і так далі, рис. 7.5. На верший погляд завдання здається дивним і незрозумілим, та практика доводить: студентам подобається!

Яка лінія (павутина) получиться в решті решт? Відповідь залежить від функції $F(x)$ та (у меншому степені) від початкового значення x_0. Три можливих випадки показано на рис. 7.6: у випадку (Б) ітерації збігаються до кореня рівняння (7.1); у випадку (А) “кружляння павутини” переходить до якоїсь орбіти навколо кореня, яка далі майже незмінна; у випадку (В) таких орбіт декілька (аналогія із “квантовими станами” електрона?). На рисунках подана також всі вихідні дані – $F(x)$ та x_0.

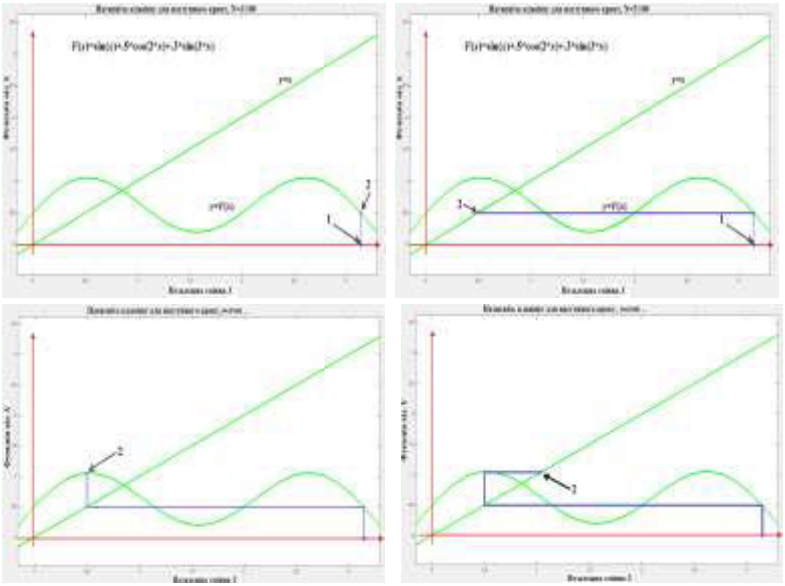

Рис. 7.5. Початковий, перший, другий і третій послідовні кроки “павучка” 2, що робить *SpiderWeb*; 1 – його початкове положення.

Залишаємо програму для самостійної розробки. Лише декілька порад програмістові: 1. Зробіть осі Ox та Oy окремим кольором (червоним наприклад), та ще стрілочки на кінцях; 2. Лінії $y = F(x)$ та $y = x$ мають бути також кольоровими, відмінними від майбутніх “павутинок”; рівняння та початкове значення доцільно запросити у діалозі; 3. На кожному етапі побудови “павутини” доцільно в інтерактивному заголовку повідомляти користувача про номер ітерації та пропонувати “*Для наступного кроку натисніть будь-яку клавішу*”.

Все це зробити легко, та воно поліпшить сприйняття програми користувачем. Наступне зробити складніше: 4. Аби графік не “плигав”, доцільно відразу визначити його майбутній розмір. Радимо: перш ніж демонструвати “павутину”, зробіть всі потрібні ітерації “мовчки”, сказавши

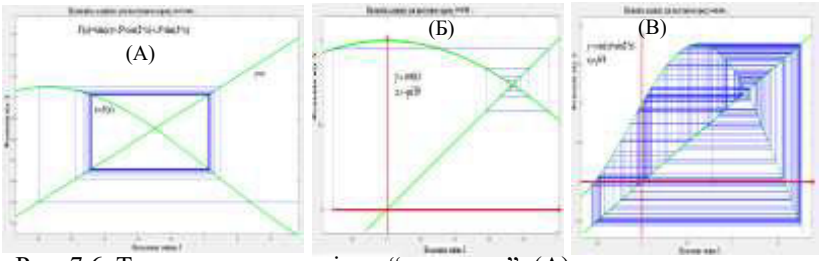

Рис. 7.6. Три випадки поведінки “павутини”: (А) стягується до постійної орбіти; (Б) збігається до кореня; (В) кілька “квантових” орбіт.

користувачеві, бо йде час, “*Прохання почекати поки я думаю*[17]”; отримавши всі ітерації, визначіть найбільше та найменше значення – звідси й розміри рисунка; 5. Коли зроблено багато ліній, важко відрізнити саме останню. Розумно її кінець позначити “павучком”; тобто якимось значком. Якщо це буде стрілочка – вона має бути то справа наліво, то знизу уверх, то зліва направо, то зверху униз. Реалізувати таку ідею – через логіку *if-elseif-else*.

Таким чином, програма *SpiderWeb* робить вас вже просунутим програмістом! Демонстраційну версію програми, зразок, можна запросити в авторів за адресою Ye_Gayev@voliacable.com.

7.2. Програма побудови фракталу *SnowFlake*

А таку складну програму у нашому підручнику ми ще не будували! Вона складна у тому відношенні, що треба зробити кілька простих програм, а потім ще одну, програму-менеджер, яка буде ними керувати стільки разів, скільки їй наказано. Одночасно вона знайомить вас з новим об’єктом сучасної науки – фракталами [15,16].

7.2.1. Постановка завдання

[17] Замість цього можна використати GUI ***waitbar***, розділ 8.3.

Треба побудувати на екрані комп'ютера ***фрактал Сніжинка***, (*фрактал Коха*) [15,16], який утворюється наступним чином, рис. 7.7:

А. Спочатку будуємо просто трикутник; він може буде довільним, з вершинами *A*, *B*, *C*, та красивіше робити його правильним, з рівними сторонами одиничної довжини (рис. 7.7*а*). Це – фрактал *SnowFlake* першого рівня.

Б. Далі кожну з сторін ділимо на три рівні частини, рис. 7.7*б*.

В. А тепер для середніх відрізків $[A_1,A_2]$, $[B_1,B_2]$ $[C_1,C_2]$ них шукаємо вершини *D*, що знов утворюють з ними правильний трикутник (рис. 7.7*в*). Зрозуміло, що ці трикутники менші за розміром.

Г. Щойно отримані вершини з'єднуємо прямими лініями, а попередні лінії вбираємо. Всі ці етапи показано на рис. 7.7. На цьому закінчено фрактал *SnowFlake* другого рівня.

Д. Зупиняємось, розмірковуємо, та знов повторюємо кроки Б – Г, вже із збільшеною кількістю відрізків, для замкненого многокутника $[A,A_1,D_1,A_2,B,B_1,D_2,B_2,C,C_1,D_3, C_2,A]$.

Друге повторення – на рис. 7.7*в*, рис. 7.8, 7.9. Так робимо кілька разів, поки наш комп'ютер буде ***спроможним*** (див. 9.8) робити таке.

Складність програми *SnowFlake*, що створює такі фігури полягає у тому, що програміст має визначити частини потрібної роботи, створити допоміжні програми (підпрограми, або процедури), перевірити їх, і вже потім робити програму-менеджер, що їх викликає потрібну кількість разів. Власне, так і треба розв'язувати кожну складну задачу у вашому житті [21]!

Виділимо три таких задачі: 1. Побудувати правильний трикутник ABC; 2. Для заданих точок A і B знайти і побудувати такі A_1 та A_2, що ділять цей відрізок у співвідношенні 1:3 та 2:3; 3. Для заданих точок A і B знайти і побудувати таку D, що утворює разом із ними правильний трикутник ADB. Розглянимо їх почергово.

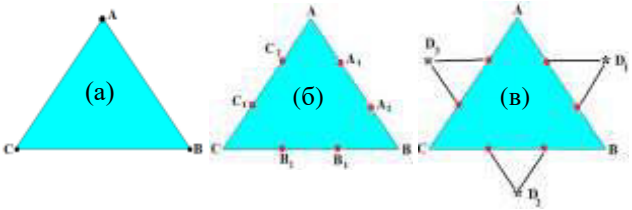

Рис. 7.7. Перші етапи побудови Сніжинки Коха

7.2.2. Перша програма, *SeedingTriangle*. Це можна зробити багатьма способами. Беремо такий: розмістимо правильний трикутник так, аби його центр був у початку координат та одна з вершин (скажімо, А) – на осі О*y*. Тоді координати вершин будуть: ***A***(0,1), ***B***(*cos*(30°),-*sin*(30°)) та ***C***(-*cos*(30°),-*sin*(30°)). Звідси наступна MATLAB-програма:

```
function [A,B,C]=SeedingTriangle
%Підпрограма для SnowFlake.m.
%За даними точками (векторами) знаходить
%точки A,B та C, що утворюють правильний трикутник.
%Приклад використання:
%  [A,B,C]=SeedingTriangle
%Ye.Gayev, 2015
A=[0,1];
B=[cosd(30), -sind(30)];
C=[-cosd(30), -sind(30)];
%to plot the Triangle
figure(1)  %Побудова трикутника:
plot([A(1),B(1),C(1), A(1)], [A(2),B(2),C(2), A(2)], 'c')
axis square, axis equal
hold on
```

7.2.3. Друга програма, *Trisection*. Ставимо цю допоміжну задачу таким чином: на вході дві будь-які точки *A* і *B* (кожна з них – вектор з двох чисел, координат *x* та *y*); на виході треба отримати дві інші точки, назвемо їх A_1 і A_2, що ділять відрізок *AB* у відношеннях 1:3 (перша) та 2:3 друга. Це – одна з початкових задач аналітичної геометрії [21, інші підручники], лише треба її “оживити” шляхом графічного зображення. Пригадуємо потрібні формули:

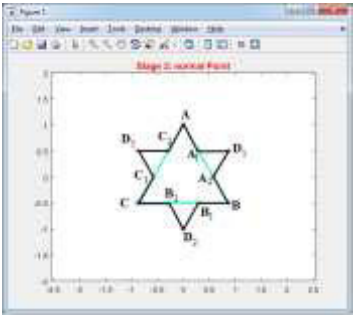

Рис. 7.8. Перша фігура за новими точками

$$x_1 = x_1 + \frac{1}{3}(x_2 - x_1),$$

$$y_1 = y_1 + \frac{1}{3}(y_2 - y_1),$$

$$x_2 = x_1 + \frac{2}{3}(x_2 - x_1),$$

$$y_2 = y_1 + \frac{2}{3}(y_2 - y_1).$$

На їх підставі виникає така проста програма:

```
function
[A1,A2]=Trisection(A, B)
        %Підпрограма для SnowFlake.m.
        %Знайти 2 точки A1(x1,y1) та A2(x2,y2),
        %Ye.Gayev, renewed for 2015
        Ax=A(1);   Ay=A(2);
        Bx=B(1);   By=B(2);
        x1=Ax+(Bx-Ax)/3;      y1=Ay+(By-Ay)/3;
        x2=Ax+2*(Bx-Ax)/3;  y2=Ay+2*(By-Ay)/3;
        A1=[x1, y1];
        A2=[x2, y2];
plot([A(1),B(1)], [A(2),B(2)], 'bp-'), %Відрізок AB
hold on
plot([A1(1),A2(1)], [A1(2),A2(2)], 'go') %Будуємо точки A1, A2
title('Стадія1:Trisection','Color', 'r')
%pause
```

7.2.4. Третя програма, *NormalPoint* є складною. Можна запропонувати кілька алгоритмів пошуку точок *D*, що разом з *A* та *B* утворюють правильний трикутник. Найпростішим, здається, є такий: $D(x, y)$ належить колу радіусом $R=|AB|$ з центром в точці *A*, тобто задовільняє рівнянню $(x-x_A)^2+(y-y_A)^2=R^2$. З іншого боку, вона лежить також на колі з центром в точці B, тобто задовільняє $(x-x_B)^2+(y-y_B)^2=R^2$. Таку систему з двох рівнянь від

двох невідомих *x* і *y* не складно розв'язати кожному школяру. Ліньки думати? Доручимо це MATLAB:

```
>> syms Ax Ay Bx By x y R
>> Sol=solve((x-Ax)^2+(y-Ay)^2-R,  (x-Bx)^2+(y-By)^2-R)
        Sol =    x: [2x1 sym]
                 y: [2x1 sym]
```

Така відповідь нас не лякає, знаємо, що *Sol* – структура, *struct* (див. 6.6) з полями *x* та *y*. У полях, як видно, зберігаються два рядочки даних типу *sym*, тобто – математичні формули. Чому їх дві? Бо згадані вище кола перетинаються у двох точках, система має 2 розв'язки. Отримати перші формули для координат точки *D* можемо так:

```
>> Dx=Sol.x(1)
>> Dy=Sol.y(1)
```

Відповідні вирази тут не наводимо, бо вони занадто громіздки. Та формули можуть бути спрощені таким чином:

```
>> Dx= simplify(Sol.x(1))                                    (7.6)
>> Dy= simplify(Sol.y(1))
```

Такий результат вже можна використовувати – скопіювати та підставити у програму, надану далі. (Не забувайте, проте, що таких точок дві!). Та цікаво, як наша формула має виглядати "людськими" очами. Пригадаємо ще одну команду, результат якої надамо повністю:

```
>> pretty(Dx)
          Ax     Bx     Ay #1        By #1
          ---- + --- + -------    -  -------    ,
           2      2        2            2
```

де доданок #1 подається ще однією формулою

```
          /            2       2          2       2                    \
          |        Ax  - 2 Ax Bx + Ay  - 2 Ay By + Bx  + By  - 4 R |
  #1=sqrt| - ------------------------------------------------------  --- |
          |            2          2          2       2                  |
          \        Ax  - 2 Ax Bx + Ay  - 2 Ay By + Bx  + By             /
```

Використаємо ці результати у наступній програмі:

```
function D=NormalPoint(A, B)
 %Підпрограма для SnowFlake.m.
```

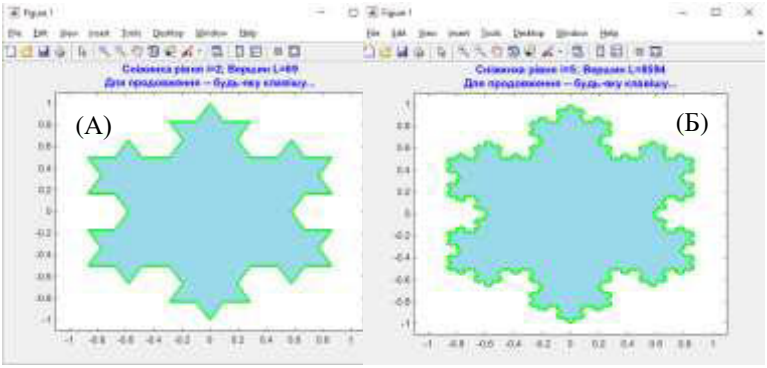

Рис. 7.9. Сніжинки другого (А) та п’ятого рівнів (Б). Зверніть увагу, як швидко нарощується кількість вершин многокутника!

```
%Знайти точку D(x,y), що утворює правильний трикутник
% до двох вихідних A(Ax,Ay) та B(Bx,By).
% Ye.Gayev, renewed for 2015
Ax=A(1); Ay=A(2);
Bx=B(1); By=B(2);
R=(Ax-Bx)^2+(Ay-By)^2; % квадрат відстані між A і B
D(1)= . . . ;  %зкопіювати (7.6)
D(2)= . . . ;  %зкопіювати (7.6)
%для тестування програми:
plot([A(1), B(1)], [A(2), B(2)], 'o-g'), hold on
plot(D(1), D(2), 'rp')
axis square, axis equal
title('Задача 3: Правильний трикутник:')
```

Тепер перевіримо всі три підпрограми.

7.2.5. Ручна побудова фракталу

Виходимо з того, що ви перевірили кожну з програм 7.2.2 –7.2.4 окремо. А тепер виконаємо таку послідовність дій у командному вікні (коментарі пояснюють, що саме і для чого робиться):

```
>> % Отримуємо 3 перші вершини, маємо початковий трикутник:
>> [A, B, C]=SeedingTriangle;
>> % отримуємо 3 додаткові точки на [A,B]:
>> [A1, A2]=Trisection(A, B);  D1=NormalPoint(A1, A2);
>> % отримуємо 3 додаткові точки вже на [B, C]:
>> [B1, B2]=Trisection(B, C);  D2=NormalPoint(B1, B2);
```

```
>> % отримуємо ще 3 точки вже на [C, A]:
>> [C1, C2]=Trisection(C, A);  D3=NormalPoint(C1, C2);
```

(Позначення зрозумілі: точки, наприклад, $B1$ і $B2$ йдуть за відповідною точкою B; нормальні точки D нумеруємо послідовно; ці назви точок далі не використовуються, бо інакше може літер не вистачити). Рис. 7.7 демонструє всі етапи побудови. Утворюємо повний перелік вершин

$$Vertices=[A; A1; D1; A2; B; B1; D2; B2; C; C1; D3; C2; A]$$

та з'єднуємо їх новою лінією (рис. 7.8). Замість трьох на початку тепер маємо 13 вершин. Їх x–координати отримуємо як *Vertices*(:,1), та y–координати як *Vertices*(:,2).

Аби йти далі, команди *Trisection* та *NormalPoint* слід застосувати послідовно до відрізків [A, $A1$], [$A1$, $D1$], [$D1$, $A2$], [$A2$, B] і так далі 13 разів. Однак робити це вручну – занадто довго. Та ручне випробування, що тут виконане, допомагає побачити вже загальний алгоритм.

7.2.6. Остаточна програма (програма-менеджер). Тепер зробимо програму-менеджер *SnowFlake*, що буде таке робити потрібну кількість разів. Ця сама кількість, n, – її вхідний аргумент.

Лістінг програми *SnowFlake.m*

```
function SnowFlake(n)
%Програма-менеджер, що будує
%Сніжинку Коха рівня n за допомогою
%програм 'SeedingTriangle.m', 'Trisection.m'
%та 'SeedingTriangle.m'.
%Приклад запуску:
%>> SnowFlake(4)
%Copyright Ye.Gayev, March 2015.

%   Етап 1:
[A,B,C]=SeedingTriangle;%Початковий трикутник
title({'Початковий трикутник.';'Розпочинай! '})
pause  %Зупинка

%    Етапи 2 та 3
Vertices=[A;B;C;A];%Початкові 3 вершини
for i=1:n
```

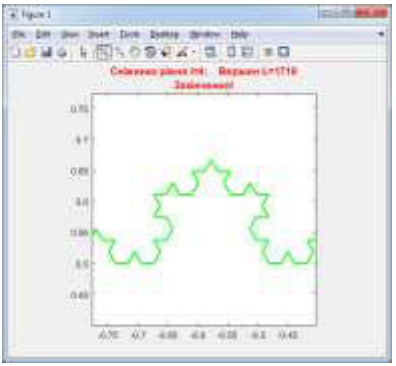

Рис. 7.10. Фрагмент Сніжинки “у малому”: самоподібність.

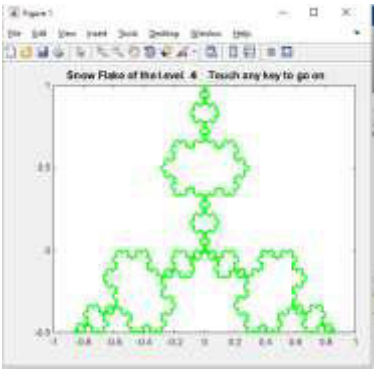

Рис. 7.11. “Альтернативна” Сніжинка: у чому проблема?

```
 %   Етап 2:
 L=length(Vertices(:,1)); %Кількість вершин?
 close(1) %знищуємо попередній полігон
 %Будуємо наступний полігон:
 plot(Vertices(:,1),Vertices(:,2),'g')
 axis([-1.1 1.1, -1.1 1.1])
title({['Сніжинка рівня i=',num2str(i-1),...
'; Вершин L=',num2str(L-1)]; ...
'Для продовження -- будь-яку клавішу...'},...
         'Color','b')
 pause

 %   Етап 3: точки A1, A2 та D для відрізків:
 VerticesTMP=[]; %для майбутніх нових точок:
 for j=1:L-1
     A=Vertices(j,:);% 2 послідовні точки
     B=Vertices(j+1,:);
     [A1, A2]=Trisection(A, B);
     D=NormalPoint(A1, A2);
     % Накопичуємо нові вершини:
     VerticesTMP=[VerticesTMP;A;A1;D;A2;B];
    % pause(.2) % Пояснення у тексті!
 end
 Vertices=VerticesTMP;
end
```

```
%Завдання виконано, завершуємо:
L=length(Vertices(:,1));
close(1)
plot(Vertices(:,1),Vertices(:,2),'g','LineW',2)
axis square, axis equal
axis([-1.1 1.1, -1.1 1.1])
title({['Сніжинка рівня i=',num2str(i),...
    ';    Вершин L=',num2str(L-1)];
     'Закінченно!'},'Color','r')
```

У програмі, як бачимо, задану кількість разів n обходимо всі сторони полігона (внутрішній цикл), і новоотримані вершини накопичуємо у матриці *VerticesTMP*. Коли ж переходимо до зовнішнього циклу – передаємо матрицю до основного переліку вершин, *Vertices=VerticesTMP*. Та перш ніж знову розпочати перегляд всіх новоотриманих вершин – обновлюємо тимчасовий масив, *VerticesTMP*=**[ ]**.

Вигляд програми трохи зміниться, якщо розкоментувати рядок `% pause(.2)` – тоді на екрані побачимо поступову побудову всіх нових вершин програмами *Trisection* та *NormalPoint*.

Пограйтеся програмою *SnowFlake* для різних n, та зверніть увагу на обставини, що є типовими для фракталів:

1. як зростає час її роботи із ростом n; при великих n програма так "замислюється", що дочекатися відповіді неможливо **(:-**. Час, який програмі потрібен, обговорюється у розділі 8.
2. як зростає поступово кількість вершин полігону та кількість його сторін. Чи можете отримати формулу їх залежності від n?
3. інструментом "Лупа" подивіться на якусь частинку границі Сніжинки: нагадує "сама себе" при меншій n?
4. "у житті" таку ізрізану форму мають кордони Норвегії – відомі фіорди. Питання: чому дорівнює довжина границь Сніжинки? Зрозуміло, що $\lim_{n\to\infty} L = \infty$, необмежено зростає. Між тим за своєю площею фігура лишається обмеженою. Не дивно? Більше про властивості цього й інших фракталів можна прочитати в [11]. Звикайте: незвичні геометричні об'єкти, фрактали,

та пов'язані з ними об'єкти (вейвлети, зокрема) починають входити у наше життя! Приклад – відомий формат ***.jpg*** для зображень.

7.2.7. Попередження. Інколи виникають зовсім не такі "сніжинки", а такі, як показано на рис. 7.11. Особливо тоді, коли *SeedingTriangle* беруть не правильний, а довільний. У чому тут причина? Пригадаємо, що на етапі 7.2.4 алгоритму ми отримали дві точки *D*, та чомусь обрали одну з них. Спробуйте поміняти, у підпрограмі *NormalPoint* взяти формули для іншої точки. Причина у тому, що *D* попадає у середину фігури, що будується. Інколи, однак, заміна D_1 на D_2 не допомагає. У такими випадку алгоритм слід доповнити перевіркою за якимось критерієм на кожному кроці, яка з двох лежить у середині, яка – ззовні, та обирати останню.

7.3. Рекурсія.

Рекурсія – чудовий інструмент програміста! Рекурсивна програма –це така програма, що викликає (використовує) сама себе. Та перш аніж показати її застосування у комп'ютерних науках та пов'язані з нею проблеми, необхідно торкнутися загально-культурного, філософського розуміння цього поняття та його історії.

7.3.1. Історія і філософія


Аристотель

Чи ви зимислювалися над тим, "*Як ми думаємо*"? Людство розмірковує над цим питанням багато сторіч. Перші закони мислення були зформульовані Аристотелем більш ніж 2 000 років тому. Його логіка повністю заперечувала так званий "*логічний порочний круг*" [22]. Приклад такої логічної помилки надав польський письменник-фантаст і філософ Станіслав Лем. Головний герой його «Зоряних щоденників» астронавт Ійон Тихий шукає в «Космічній енциклопедії» інформацію про ***сепульки***:. Він читає таке:

«СЕПУЛЬКИ — *важливий елемент цивілізації ардритів з планети Ентеропія*. *Див*. **СЕПУЛЬКАРІЇ**».

Він послідував наданим вказівкам і прочитав:

«СЕПУЛЬКАРІЇ — *пристрої для* ***сепуляції*** (*див.*)».

Астронавт пошукав «Сепуляція»; там значилось:

«СЕПУЛЯЦІЯ — заняття ардритів з планети Ентеропія. Див. **СЕПУЛЬКИ**».

Ви зрозуміли, що таке *сепулька*? Навряд, бо невідоме поняття визначається … через нього самого. На відміну від цього, всі математичні науки будуються дуже строго: основні поняття та їх відношення (аксиоми) приймаються як очевидні, і з них "виводяться" всі інші властивості та факти предметної сфери, теореми. Кожне твердження базується на кількох інших "нижнього рівня", кожне з них – на інших, і так далі "униз", аж поки дойдемо до "очевидних" аксіом.

Пересічна людина також замислювалася над деякими дивними проблемами, наприклад, "Чи може Змій з'їсти сам себе"? Зображення рис. 7.12*а* зустрічається у багатьох народів і отримало назву *Uroboros*. Нидерландский график Морис Ешер намалював "*Руку, що відтворює сама себе*", рис. 7.12*б*. Барон Мюнхаузен, як відомо, витягнув сам себе за волосся з ями…

Такий спосіб мислення отримав назву "*логічний порочний круг*"; зрозуміло, чому він заборонений в науці! Здавалося б, якщо "логічний круг" використовувати не можна, його слід уникати й у комп'ютерній науці. Та тут діє Закон: "Якщо дуже хочеться – так можно". За певних умов, однак! І тут нам час до програмування…

7.3.2. Рекурсія та метод математичної індукції в математиці – важливий спосіб доведення теорем, що стосуються нескінченних послідовностей тверджень T_n, що залежать від номера n: якщо (*i*) T_1 правильно, та (*ii*) з правильности T_{n-1} витікає й правильність T_n, то тоді твердження є вірним у загальному випадку[18], для будь-якого номера n.

7.3.3. Найпростіша рекурсія: програма *MyFactorial*

[18] https://uk.wikipedia.org/wiki/Математична_індукція; дивіться приклади!

Рис. 7.12. (*а*) “Уроборос”: що станеться із Змієм, що їсть сам себе? (б) Моріс Ешер “Рука, що створює сама себе”

Рекурсія у програмування – це коли програма звертається сама до себе. Покажемо, як рекурсія дозволяє будувати обчислювальні алгоритми. Природно, що в MATLAB існує програма для обчислення факторіалу *factorial*, де

$$factorial(n) = n! \overset{Df}{=} 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n .$$

Тут треба додати, що вважається, “за визначенням”, $0! \overset{Df}{=} 1$. Маємо навчитися створювати аналогічну функцію.

Наступна програма обчислює цю важливу математичну функцію, спираючись на *алгоритм рекурсії*:

Лістінг *MyFactorial2.m*

```
function Fact=MyFactorial2(n)
%Обчислення факторіалу цілих n методом рекурсії.
%Ye.Gayev, 2015
if n<0
    disp('Факторіал не існує для від’ємних чисел!')
    return
elseif n==0                                    (б)
    Fact=1; %за визначенням для n=0
else
      Fact=MyFactorial2(n-1)*n ;
      %^Увага, програма звертається до себе!
end
```

Як працює така програма? Отримавши якесь ціле *n*, програма множить це *n* на “саму себе”, але від аргументу, що на 1 менше, *n*-1. Цей *MyFactorial*2(*n*-1) також зводиться до тієї ж функції *MyFactorial*2(*n*-2), і так далі, аж доки отримаємо *MyFactorial*2(*0*). Остання функція вже має визначене значення 1. Тепер програма *MyFactorial*2 повертається на крок назад та обчислює *MyFactorial*2(2)=$1\cdot 2$. Знов на крок вище – маємо *MyFactorial*2(3)=$2\cdot 3=6$; знов на крок назад – *MyFactorial*2(4)=$6\cdot 4$, і так далі до *MyFactorial*2(*n*). Маємо нарешті остаточний результат.

Теорія, що вівчає рекурсивні функції (і програми) стверджує:

Теорема. *Будь-яку рекурсивну програму можна перетворити на таку, що використовує лише цикли*.

Нескладно побачити, що надану програму дійсно можна побудувати за допомогою циклу *for–end*:

Лістінг *MyFactorial*1.*m*

```
function Fact=MyFactorial1(n)
%Обчислення факторіалу циклом.
%Ye.Gayev, 2015
if n<0
    disp('Факторіал не існує для відємних чисел!')
    return
elseif n==0
    Fact=1; %за визначенням для n=0
else
    Fact=1; %початкове значення
    for i=1:n
        Fact=Fact*i;  %накопичення доданків
    end
end
```

На відміну від попередньої програми, тут величина *Fact* формується поступово: спочатку 1 множиться на 2, потім на 3, і т.д. до *n*. Поки що не видно, що рекурсивна побудова комп’ютерних програм часто приводить до значно простіших алгоритмів. Перш, ніж це показати, розглянемо ще кілька задач.

7.3.4. Обчислення чисел Фібоначчі: *MyFibonacci*

Італійський математик XIII сторіччя Leonardo Pisano, названий Fibonacci, запропонував першу у світі математичну модель з екології, так звану задачу про розмноження кролів[19]. Згідно з нею, популяція кролів зростає як чисельна послідовність

1, 1, 2, 3, 5, 8, 13, 21, ...,

кожне число дорівнює сумі двох попередніх, тобто за рекурсивною формулою

$$F(1)=1,\ F(2)=1,\ F(n)=F(n-1)+F(n-2). \qquad (7.7)$$

Обчислювати $F(n)$ можна за такою рекурсивною програмою:

```
function Y=My2Fibonacci(N)
%знаходження чисел Fibonacci за їх номером N
% з використанням РЕКУРСІЇ.
if N==1
    Y=1;
elseif N==2
    Y=1;
else
    Y=My2Fibonacci(N-1)+My2Fibonacci(N-2);
end
```

Здається, що її робота за формулою (7.7) є цілком очевидною. Надамо також “переробку” такої програми без рекурсії, за допомогою циклів.

```
function Y=My1Fibonacci(N)
%знаходження чисел Fibonacci за їх номером
%починаючи з двох перших
% і без використання РЕКУРСІЇ
N1=1;
if N==1
    Y=1;
elseif N==2
    Y=1;
else
```

[19] Див. https://uk.wikipedia.org/wiki/Фібоначчі

```
    Y0=1; Y1=1; %початкові значення Y0, минуле, та Y1, позаминуле
    while N1<N
        Y=Y1+Y0;
        N1=N1+1;
        Y0=Y1; Y1=Y; % минуле значення Y0, позаминуле Y1
    end
end
```

Таким чином, сутність рекурсивної програми, скажімо *RecursProg*, полягає у тому, що вона звертається сама до себе. Та цього ще недостатньо! Розглянемо таку повчальну програму

```
function RecursProg
%Найпростіше звертання програми до себе
RecursProg % Виклик самої себе!
```

Запустимо її з командного рядка та й отримаємо:

```
>> RecursProg

Maximum recursion limit of 500 reached. . . exceeding your available
    stack space can crash MATLAB and/or your computer.
```

Помилка цієї програми полягає у тому, що вона звертається багато разів до себе – а далі? До яких меж, до якої *глибини*? Адже нарешті програма має дістатися якогось "берега" та піти назад. "*Глибина рекурсії*", до якої у MATLAB дозволено "заглиблюватися", складає 500. Після цього може статися аварія з вашим комп'ютером! Та не лякайтеся: машина зупиняється та попереджає вас, як вище. (Для порівняння: Java, яку вивчатимо далі, дозволяє глибину рекурсії лише 12).

7.3.5. Блок-схеми рекурсивних програм

Яким чином інтерпретатор MATLAB "розуміє" та виконує рекурсивні програми? Пояснемо на прикладі *MyFactorial*(4): отримавши таке завдання від програми, інтерпретатор створює блок пам'яті (стек), куди записує, що треба 4 помножити на *MyFactorial*(3); для останнього також створює блок пам'яті "3 *помножити на MyFactorial*(2)"; знов створює стек "2 на *MyFactorial*(1); ще один стек "1 *помножити на MyFactorial*(0)". Останнє значення йому відомо, тому алгоритм "повертається" і почергово передає у

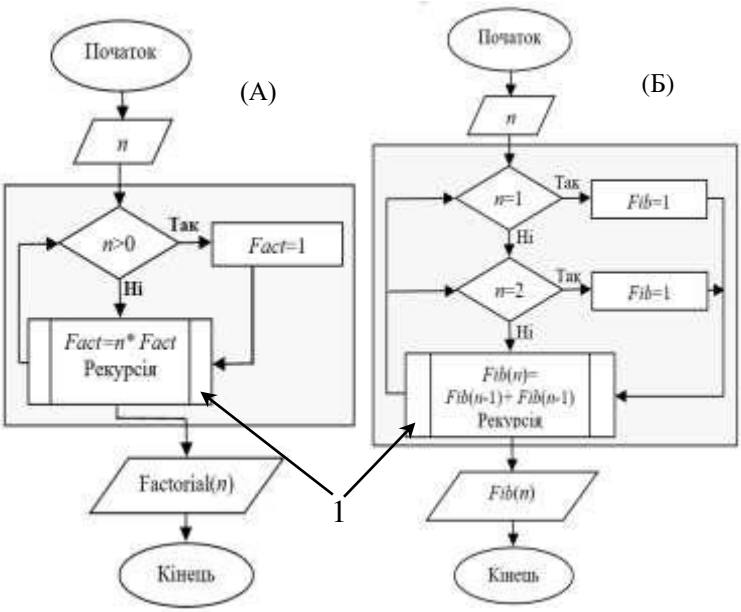

Рис. 7.13. Блок-схеми програм (А) *factorial*; (Б) *fibonacci*

попередні стеки чергові значення. Дійшов до початку – результат готовий.

Ми бачили, що блок-схеми суттєво пояснюють роботу алгоритмів. Як же показати, що "програма звертається сама до себе"? У комп'ютерній науці домовлено показувати підпрограми, які програма викликає, блоком 1, рис. 7.13. У даному випадку домовлено додатково писати "Рекурсія". Певну частину блоків окремлюють у додаткову рамку, рис. 7.13.

7.4. Рекурсія: Обчислення визначника

Досі рекурсивні програми були так само прості, як і альтернативні. Дамо складніший приклад.

У першій частині підручника було розроблено програму *MyDet*1, що обчислює визначник методом Гаусса (Додаток 6 в [1]). Тепер зробимо рекурсивну програму *MyDet*2. Для цього потрібен якийсь метод, що обчислення

визначника n-го порядку зводить до такого для $(n\text{-}1)$-порядку. Можемо спиратися на

Теорема про розклад визначника за елементами рядка

$$|A| = a_{11}A_{11} - a_{12}A_{12} + \ldots + (-1)^{n+i} a_{1n}A_{1n} \qquad (7.8\text{р})$$

або стовпчика

$$|A| = a_{11}A_{11} - a_{21}A_{21} + \ldots + (-1)^{n+1} a_{n1}A_{1n} \qquad (7.8\text{с})$$

(номер радка/стовпчика взято i=1).

У теоремі $A_{ij} = (-1)^{i+j} M_{ij}$ – алгебраїчне доповнення елемента a_{ij} вихідної матриці A, що за вказаною формулою пов'язано з мінором цього елемента M_{ij}; а останній – це визначник матриці, що утворюється з A після викреслення i-го рядка та j-го стовпчика. Розмірність визначників M_{ij} на 1 менше за визначник $|A|$!

Понижуючи розмірність потрібних визначників, прийдемо до визначника першого порядку. А вже тут – за визначенням:

$$|a_{ij}| \overset{Df}{=} a_{ij}. \qquad (7.9)$$

Корисно цю теорему "випробувати руками" перш ніж програмувати. У випадку n=2 маємо:

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11} \cdot |a_{22}| - a_{12} \cdot |a_{21}| = a_{11}a_{22} - a_{12}a_{21},$$

тобто отримали відомий результат. Для n=3 маємо:

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} = a_{11} \begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} - a_{12} \begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{vmatrix} + a_{13} \begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{33} \end{vmatrix} =$$

$$= a_{11}(a_{22}a_{33} - a_{23}a_{32}) - a_{12}(a_{21}a_{33} - a_{23}a_{31}) + a_{13}(a_{21}a_{33} - a_{22}a_{31}),$$

тобто теорему використано двічі.

Для створення рекурсивної програми для довільної розмірності n єдина складність виникає у створенні матриць розмірності n-1, від яких обчислюємо мінори. Тут треба пригадати головні властивості MATrix LABoratory у маніпуляції з матрицями.

Нехай A – квадратна матриця розмірності n; отримаємо з неї матрицю для мінора першого елемента в рядку a_{11}:

M=A(2:*end*, 2:*end*) %матриця (n-1) на (n-1)

Аналогічно для останнього елемента рядка a_{1n}:

M=A(2:*end*, 1:*end*-1). %матриця (n-1) на (n-1)

Складніше діяти для будь-якого іншого елемента рядка a_{1i}:

$M1$=A(2:*end*, 1: ***i***-1) %матриця (n-1) на (i-1),
$M2$=A(2:*end*, ***i***+1 : *end*) %матриця (n-1) на (n-i).

Тепер з цих допоміжних матриць утворюємо результуючу

M=[$M1$, $M2$] %матриця (n-1) на (n-1).

Від неї вже і треба обчислювати визначники – мінор та алгебраїчне дополнення. Тепер програма буде зрозумілою:

Лістінг *MyDet2.m*

```
function D=MyDet2(A)
% обчислення визначника матриці A методом рекурсії.
% Приклад використання:
% >> A=rand(10,10),  D=MyDet2(A)
% D=MyDet(A)%
% Порівняйте з det(A)!
%% Coright Katerina Khavrai 4.03.2015
L=size(A); L1=L(1);L2=L(2);
if L1~=L2   % Якщо матриця не квадратна:
    disp('Матриця не є квадратною!')
    disp('Зкоректуйте завдання!')
     D=NaN;  % Надати значення D !
    return  % Закінчити роботу у такому випадку
end

% Обчислення у випадку квадратної матриці:
if L2==1
    D=A(1,1);
```

```
else
    sign=1; D=0;
    for i=1:L2
        if i==1
            M=A(2:end, 2:end),;
        elseif i==L2
            M=A(2:end, 1:end-1),;
        else
            M1=A(2:end, 1: i-1),; %матриця (n-1) на (i-1),
            M2=A(2:end,  i+1:end),; %матриця (n-1) на (n-i).
            M=[M1, M2],;  %матриця (n-1) на (n-1).
        end
        D=D+sign*MyDet2(M)*A(1,i);
        sign=sign*(-1);  %чередування знаку доданків
    end
end
```

Це функція *MyDet2* рекурсивна, бо вона звертається до *MyDet2*, сама до себе. Ось її випробування:

```
>> format long
>> A=rand(7,7);  D=MyDet2(A)
D =  -0.056009310573719
>> det(A)
ans =  -0.056009310573719
```

З високою точністю визначник співпадає із звичайним обчисленням.

7.5. Рекурсія: Вейвлет-функція Добєши

Для усіх наведених раніше випадках ми мали альтернативні програми, що не використовували рекурсії. Розглянемо зараз задачу, де без рекурсії обійтися дуже складно!

7.5.1. Функція Добєши. Побудуємо функцію Добєши[20] $\varphi(r)$, яку положено в основу популярного ***.jpg***-формата для зображень [23]. Задається вона на інтервалі $0 \le r \le 3$ аргумента r:

[20] https://ru.wikipedia.org/wiki/Добеши_Ингрид

→для певних r ii значення задаються явно

$\varphi(r)=0$ якщо $r\le 0$, $\varphi(1)=2h_0$, $\varphi(2)=2h_3$, $\varphi(r)=0$ якщо $r\ge 3$;

→для всіх інших – рекурсивно, тобто

$$\varphi(r)=h_0\cdot\varphi(2r)+h_1\cdot\varphi(2r-1)+h_2\cdot\varphi(2r-2)+h_3\cdot\varphi(2r-3), \qquad (7.10)$$

де коефіцієнти є

$$h_0=\frac{1+\sqrt{3}}{4},\ h_1=\frac{3+\sqrt{3}}{4},\ h_2=\frac{3-\sqrt{3}}{4}\ \text{та}\ h_3=\frac{1-\sqrt{3}}{4}.$$

Припускаємо, студентам це не дуже зрозуміло. Тому спробуємо, як завжди, спочатку "попрацювати руками". Якщо побудувати графік функції Добеши лише за даними у точках 0, 1, 2 та 3 отримаємо графік 1 рис. 7.14. Розрахуємо її більш точно, взявши до уваги проміжні точки $r=\frac{1}{2}$, $\frac{3}{2}$ та $\frac{5}{2}$.

Враховуючи (7.10) маємо для першої та другої:

$$\varphi(\tfrac{1}{2})=h_0\cdot\varphi(1)+h_1\cdot\varphi(0)+h_2\cdot\varphi(-1)+h_3\cdot\varphi(-2)=h_0\cdot 2h_0+0+0+0=2h_0^2,$$

$$\varphi(\tfrac{3}{2})=h_0\varphi(3)+h_1\varphi(2)+h_2\varphi(1)+h_3\varphi(0)=0+2h_1h_3+2h_2h_0+0=2(h_1h_3+h_2h_0)$$

і аналогічно $\varphi(\frac{5}{2})$. Остаточні чисельні значення нам не важливі – цю роботу нехай робить комп'ютер. Побудуємо графік тепер – маємо криву 2 на рис. 7.14. Сподіваємось, метод роботи з рекурсивно заданой функцією (7.10) зрозумілий: обчислюємо її лише для так званих ***бінарних*** значень аргументу, тобто таких, що можно отримати з дроби виду $r=\frac{p}{2^k}$, де k – будь-яке ціле, а чисельник – $p=0,1,2,\ldots,3\cdot 2^k$. Тепер можна писати програму MyDaubechies обчислення $\varphi(r)$ для бінарних r:

Лістінг *MyDaubechies.m*

```
function Dr=MyDaubechies(r)
%Обчислення функції Добеші для бінарного r
% Φ(r)=0, якщо r<=0;
% Φ(1)=2h0,  Φ(2)=2h3;
% Φ(r)=0, якщо r>=3;
% Φ(r)=h0*Φ(2r) + h1*Φ(2r-1) + h2*Φ(2r-2) + h3*Φ(2r-3).
```

```
% [http://sernam.ru/book_sel.php?id=74]
%-------------------------------------------------
% Ye.Gayev copyright, March 2016.
h0=(1+sqrt(3))/4;   h1=(3+sqrt(3))/4;
h2=(3-sqrt(3))/4;   h3=(1-sqrt(3))/4;
if r<=0
    Dr=0;
elseif r>=3
    Dr=0;
elseif r==1
    Dr=2*h0;
elseif r==2
    Dr=2*h3;
else
    Dr=h0*MyDaubechies(2*r)+...
       h1*MyDaubechies(2*r-1)+...
       h2*MyDaubechies(2*r-2)+...
       h3*MyDaubechies(2*r-3);
end
```

Робота програми здається очевидною. Перевіримо її. По перше, програма працює для будь-якого цілого,

```
>> y=MyDaubechies(1)
    y =    1.3660.
```

По-друге, для раціональних чисел типу $r = \frac{p}{2}$

```
>> y=MyDaubechies(5/2)
    y =    0.0670.
```

По-третє, для $r = \frac{p}{2^k}$

```
>> y=MyDaubechies(7/4)
y =   -0.0915
>> y=MyDaubechies(37/16)
y =   -0.0043,
```

та не працює для інших аргументів, *y=MyDaubechies*(*pi*/2), *y=MyDaubechies*(7/3) тощо. Причина: аргументи не є бінарними!

Можемо, таким чином, обчислювати функцію Добеші для бінарних аргументів $r = \frac{p}{2^k}$, де k – будь-яке ціле. Аби тепер побудувати її графік для обраног k, поділимо область $0 \le r \le 3$ на $3 \cdot 2^k$ частин з кроком $dr = \frac{1}{2^k}$, і обчислимо значення функції в кожному з цих аргументів. За ними можемо побудувати графік функції. Доручимо це програмі

Лістінг *MyDaubechiesPlot*

```
function MyDaubechiesPlot(n, Color)
%Будує графік функції Добєши кольором Color
%для точок від r=0 до r=3 з бінарним кроком dr=1/2^n,
% де n – порядок функції.
% Ye.Gayev copyright, March 2016.
dR=1/2^n;
r=0:dR:3;    N=length(r);
for i=1:N
      %використання попередньої програми:
      F(i)=MyDaubechies(r(i));
end
plot(r, F, Color)
```

Функція Добєши, в ідеалі, відповідає $k = \infty$. Та такого ми отримати не можемо. На рис. 7.14,Б побудовано її графік для $k = 10$. Збільшуйте фрагменти кривої, побачите "диво": на кривій маса зламів. І тим не менше, ця функція неперервна та неперервно диференційовна!

7.5.2. Вейвлет-функція Добєши. Коли вже маємо функцію Добєши, можна визначити й наступну функцію

$$\psi(r) = -h_0 \cdot \varphi(2r-1) + h_1 \cdot \varphi(2r) - h_2 \cdot \varphi(2r+1) + h_3 \cdot \varphi(2r+2), \quad (7.11)$$

яку називають вейвлет-функцією. Її графік також нескладно побудувати з використанням рекурсії. Зробіть це самостійно! Тук важливо, що $\psi(r)$ обчислюється не саме через себе, а викликом $\varphi(r)$, яка вже обчислюється рекурсивно. Кажуть: $\psi(r)$ використовує *непряму рекурсію*.

Для закріплення останнього матеріалу пропонується лабораторна робота № 11.

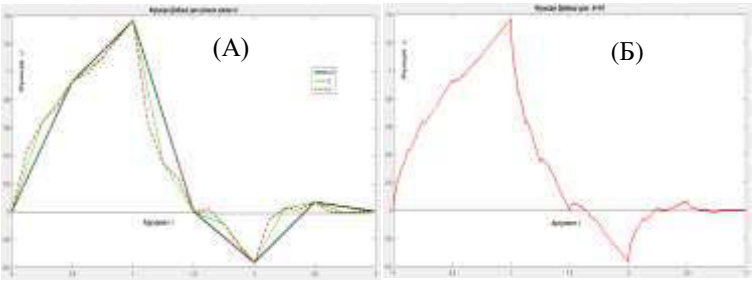

Рис. 7.14. Функція Добєши (А) для маленьких k=1,2 та 3; (Б) для k=10.

7.6. КОНТРОЛЬНІ ПИТАННЯ ДО РОЗДІЛУ 7

1. Що таке *ітерація* – як цей термін перекладається, що означає? Яке ваше враження: чи складна ця операція, які знання потрибує, чи в складних задачах застосовується?
2. Як працює MATLAB-команда *fzero*, яку роль грають її аргументи? Оберіть рівняння для дослідження з Додатку 2 посібника [1].
3. За яких умов метод ітерацій може знаходити корені рівнянь? У яких змінних доцільно проводити графічний аналіз ітераційного процесу?
4. Організуйте ітерування рівняння, обраного у 2, та знайдіть його розв'язок. Дослідіть, як впливає початкове значення ітерації. Графічний аналіз проведіть обома методами, що показано на рис. 7.2.
5. Яке визначення ви надали би терміну *хаос*? У співставленні з терміном *детермінований процес*?
6. Проведіть дослідження, як початкове значення $\boldsymbol{x0}$ впливає на результати роботи програми *Galaxy*.
7. Оберіть ті випадки роботи програми *Galaxy*, коли ітераційні точки утворюють n-променеву "галактику", та дослідіть, що вони поводять себе з періодом n. (Порада: використати програми *MyColor*3, *Palette* розділу 5.7).

7.7. ЗАДАЧІ ДО РОЗДІЛУ 7

1. У випадку (А) рис. 7.3 перевірте, що на 1й, 2й, …, 9й промені падають лише ті точки, що генеруються програмою Galaxy, номер яких діляться на 9 з остачею відповідно 0, 1, …,8. Порада: застосувати програму *Palette*, 5.7.
2. У випадку (Б) рис. 7.3 перевірте, що на 1й, 2й, …, 8й промені падають лише ті точки програми Galaxy, номер яких діляться на 8 з остачею відповідно 0, 1, …,7.
3. Покажіть: якщо програмою *Palette* генерувати лише три кольори, то на три промені галактики випадку (В) рис. 7.3 попадуть точки лише одного кольору.
4. Знайдіть формулу, як зростає кількість сторін фракталу *SnowFlake* разом із його рівнем (складність) *n*.
5. Зробіть програму побудови трикутника Серпінського[21], рис. 7.15, ліворуч, за алгоритмом: на кожному етапі (включаючи початковий, №0) будується трикутник; кожна його сторона ділиться навпіл і вирізається середній трикутник; три крайові трикутники піддаються аналогічній процедурі, і так – до певного рівня *n*.
6. Зробіть програму побудови килима Серпінського[22], рис. 7.15, праворуч, за алгоритмом: на кожному етапі кожна сторона квадрата ділиться на 3 рівні частини, та середній квадрат вирізається; така ж процедура повторюється з кожним з восьми квадратів, що лишилися.
7. Побудуйте графік вейвлет-функції Добєши $\psi(r)$.

Сподіваємось, Ви розв'язали більшість задач.
Вітаємо! Можете рухатись далі.

[21] https://uk.wikipedia.org/wiki/Трикутник_Серпінського
[22] https://uk.wikipedia.org/wiki/Килим_Серпінського

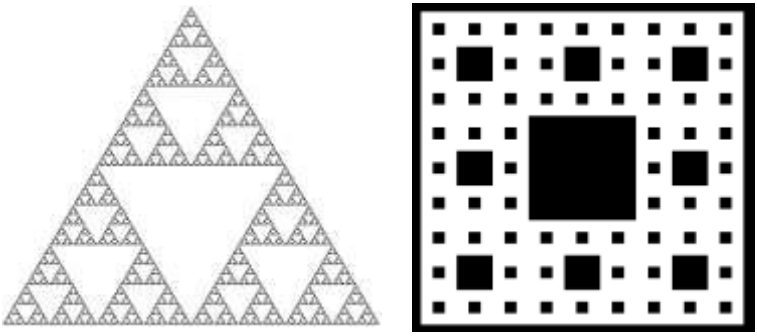

Рис. 7.15. Фрактали Серпінського: трикутник (ліворуч)
і килим (праворуч). Самі можете запрограмувати?

Модуль 4

Третій рівень майстерності програміста

8. До розумних комп'ютерів: Графічні інтерфейси (GUI)

8.1. До розумних і чудових – крок за кроком

Коли інженери і математики лише створювали перші комп'ютери, вони "турбувалися лише за себе", за свої завдання: проводити складні розрахунки, переважно для військової галузі. Тому екран тогочасної електронної обчислювальної машини (ЕОМ) виглядав чорним, як показано на рис. 8.1.*а*. Ввод завдання до ЕОМ, та часто-густо й вивід інформації відбувався з/на паперову перфоленту, рис. 8.1,*б*. Вивчити сотню-дві команд "спілкування" з такою ЕОМ математикам було не складно.

Та з того часу, як комп'ютер почали використовувати гуманітарії, офісні працівники й навіть двірники (хоча б у вигляді електронної книги або мобільного телефону), вимоги до способу комунікації з ЕОМ значно посилилися. Тепер вони такі, аби цим приладом та програмами могли користуватися навіть "чайники".

Ми вже багато чому навчилися у цьому курсі – розв'язувати математичні задачі з MATLAB та створювати власні комп'ютерні програми. Останні виглядають, однак, якось несучасно… Як же створювати такі „красиві" програми, як Word, як Photoshop, та такі інші?

Давайте обговоримо, чого ми хочемо від наших програм, як вони мають виглядати? До висновку нам не дійти, якщо не обернутися назад: який шлях пройшло „комп'ютерне людство", аби створити саме такі програми.

Як людина „спілкувалася" з комп'ютером на початку комп'ютерної ери? Замість виразу на зразок „$a + b$" ми „казали" комп'ютерові якось так

0101101 001 1101001

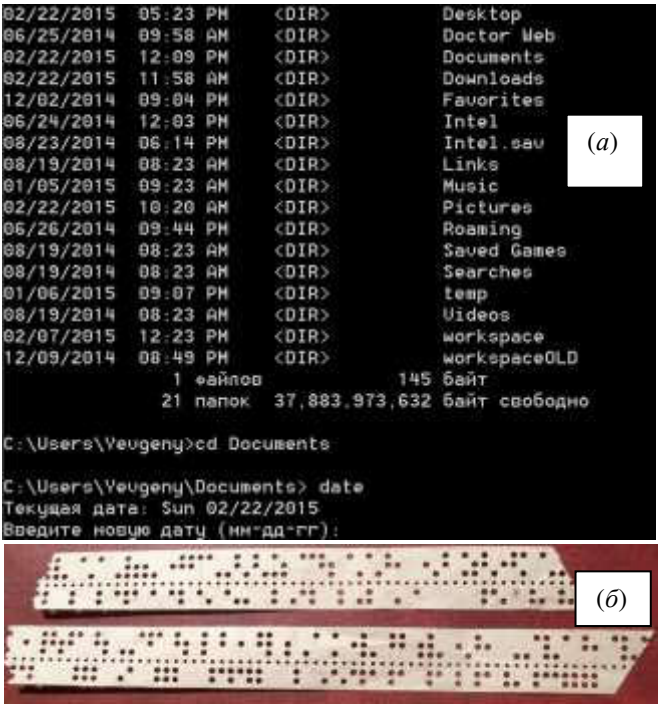

Рис. 8.1. (*а*) Вигляд екрану до ери Windows; (*б*) Зразок перфокарти, за допомогою якої інформація вводилася у комп'ютер. Вивід здійснювався на екран та на таку ж перфоленту.

що означало приблизно „*візьми число з ячейки №* $\mathbf{1}*2^0+\mathbf{0}*2^1+\mathbf{1}*2^2+\mathbf{1}*2^3+\mathbf{0}*2^4+\mathbf{1}*2^5=45$ *та додай* (*операція номер 001*) *до нього число з ячейки №* $\mathbf{1}*2^0+\mathbf{0}*2^1+\mathbf{0}*2^2+\mathbf{1}*2^3+\mathbf{0}*2^4+\mathbf{1}*2^5+\mathbf{1}*2^6=105$". Далі мала б бути наступна команда номер *100* „*запиши результат у ячейку пам'яті № ...*". А повідомити комп'ютер про це ми мали б за допомогою *перфокарти* або *перфоленти* із дірочками (рис. 8.1,*б*). „Зручно" і „наочно", чи не так **-:**)? А на рис. 8.2 бачимо одну з перших обчислювальних машин, яка мала

Рис. 8.2. "МІР-2" – це електронна обчислювальна машина, що у 1968 р. була створена в Інституті кібернетики НАН України. Дивіться подробиці в Internet-музеї [4]

назву МІР-2 (**М**ашина **І**нженерних **Р**озрахунків[23]). Можемо пишатися, що така машина народилась у нашій країні!

З того часу багато зусиль було покладено саме на проблему „спілкування" людини з комп'ютером. Мова спілкування досягла високого рівня, вона сьогодні майже звична людська мова, хіба що кількість слів і висловів обмежена:

„***for . . . end***", тобто „*робити . . . таку-то кількість разів*",

або

„***if . . . else . . . end***",

тобто „*якщо . . ., то робити . . .* , *а інакше робити . . .*".

Такі речення вводимо з клавіатури і бачимо їх на екрані. *Компілятор* або *інтерпретатор*[24] перекладає їх на внутрішню мову комп'ютера. Екран і клавіатура дозволили спілкуватися з комп'ютером більш „інтелектуально". Наприклад, у формі письмового діалогу програма може повідомити і запитати на екрані:

„***Зроблено . . . ітерацій.***

[23]Більше тут: https://uk.wikipedia.org/wiki/МИР_(ЕОМ)
[24]https://uk.wikipedia.org/wiki/Компілятор
https://uk.wikipedia.org/wiki/Інтерпретатор

Продовжувати – введи 1, зупинитись – введи 0".

Такі діалогові програми обговорювали у першій частині [1] та у [2].

З часів перших ЕОМ були винайдені більш зручні *DOS-оболонки*[25] Norton Commander, Far, Total Commander та подібні, що використовуються до цього часу. Однак більшість з нас використовує сьогодні винахід фірми Microsoft – операційні системи Windows, де певні команди комп'ютеру сховані (викликаються) за тими чи іншими графічними іконками: натискуємо – команда виконується. Згодом комп'ютер буде навчено розуміти нашу людську мову. Але сьогодні стандартом визнано саме „графічний" спосіб спілкування з ним, який науково називається *Graphical User Interface*, *графічний інтерфейс користувача*, або скорочено GUI. Саме GUI-оболонку навчимося „одягати" на наші MATLAB-програми, які вже розроблені.

Про історію винаходу GUI та інші подробиці можна прочитати в [5–7]. Доцільно обрати якусь Вашу улюблену програму та дослідити – які в неї є елементи GUI. Візьмемо, наприклад, текстовий редактор Word. На рис. 8.3 і 8.4 показано графічні зображення деяких з них:

1. „*Зберегти*" та „*Друкувати*" – кнопки, що натискаються (*pushbutton*), рис. 8.3 ліворуч. Зверніть увагу ще на один винахід, що став стандартом інтерфейсу: наводимо мишу – має з'явитися підказка!

2. Кнопка, що відкриває нове меню, рис. 8.3 праворуч. Такі кнопки називають *popuptenu*: натискаємо – „випадає" меню додаткового вибору.

3. На рис. 8.4 показано меню принтера. Маленькі круглі віконця 3 ліворуч, де може або стояти чорний „кружечок", або ні, називають *radiobutton*. Маленькі квадратні віконці (4) праворуч, де може бути „галочка" – теж елементи управління, так звані *checkbox*.

[25] https://uk.wikipedia.org/wiki/DOS,
https://uk.wikipedia.org/wiki/Файловий_менеджер

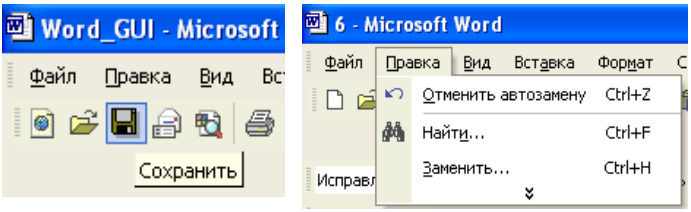

Рис. 8.3. Приклади елементів GUI у програмі Word

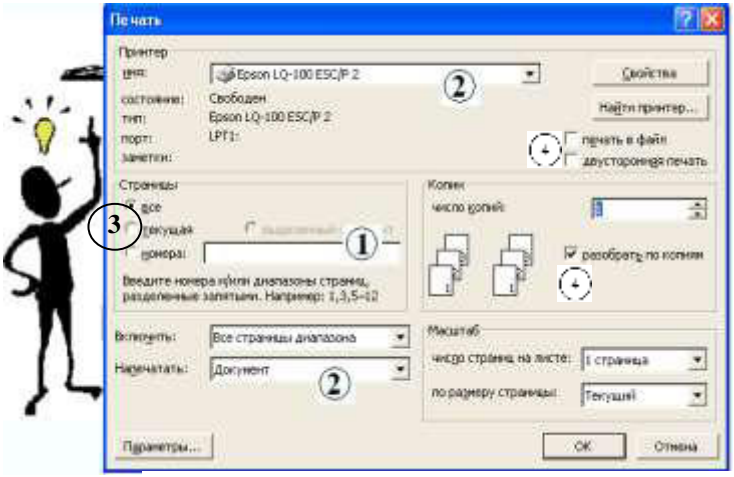

Рис. 8.4. GUI прінтера програми Word

4. Вікна типу 1 у меню принтера (рис. 8.4) слугують для вводу текстової чи числової інформації і називаються *Edit*-елемент, а вікна 2 – *listbox*-елементи, для вибору з певного переліку можливостей.

MATLAB також має колекцію елементів управління, з яких можна складати власні графічні програми. Точніше – графічні оболонки для Ваших програм. Тут познайомимось з тими програмними засобами, що надають нашим програмам рис “інтелекту”. Найпростіший шлях – діалогові програми через командний рядок. Такі програми були створені у розділі 4.1 та 4.2 (програми *limit2.m*, *MyTaylor*), у розділах

5.1.2 та 7.1.3 (програми *Helicopter*, *Galaxy* та *SpiderWeb*). Та все ж вони далекі від сучасного рівня!

8.2. Управління графікою; новий тип даних –хендли

Ми бачили, що сучасний тип програми, GUI, – це управління програмами "через картинки". З картинок і почнемо. Побудуємо якусь картинку, так наприклад,

>> *x*=-2**pi* : *pi*/100 : 2**pi*; *y*=*sin*(*x*)./*x*; *Hp*1=*plot*(*x*,*y*)

або так[26]

>> *Hp*2=*ezplot*('*sin*(*x*)/*x*') .

Отримаємо графік відомої функції, як раніше. Та зараз справа не у цьому: зверніть увагу, що *Hp*1 або *Hp*2 отримали значення "*Hp*2=*Line with properties*: . . ." (задля цього ми не поставили ";" наприкінці). Що це? Для чого?

Ще більше "*properties*" (властивостей) отримаємо командою

>> ***get***(*Hp*2),

і головні з "*властивостей*" такі: '*Color*', '*LineStyle*', '*LineWidth*', '*Marker*', '*MarkerSize*', '*MarkerFaceColor*', '*Visible*' (пишуться в апострофах!). *Hp*1 або *Hp*2 – це handl'и (хендли), у перекладі "*ручка*" до "*об'єкта*", створеного командою[27]. Цей об'єкт зараз – щойно побудована лінія, графік; саме до неї відносяться названі властивості. Переконаємося, що ця "*ручка*" (а надалі будемо казати *хендл*, або *вказівник* – ще один, новий ***тип даних***) дійсно дає можливість впливати на властивості об'єкта, лінії у даному випадку:

>> ***set***(*Hp*2, '*Color*', [.5 .1 .5], '*LineWidth*', 6) – (8.1)

отримали цю криву іншого кольору й іншої товщини! Експерименти з іншими властивостями зробіть самі.

На часі зформулювати загальне правило для нових команд ***get*** і ***set***: вони отримують (встановлюють) ті чи інші

---

[26] В ранніх версіях MATLAB команда *ezplot* хендла не мала.

[27] Термін "*об'єкт*" пишемо у лапках, бо він вимагає визначення. Тут достатньо розуміти його інтуїтивно.

значення для властивостей (*properties*) об'єкта з вказаним хендлом. Останній обов'язково вказується першим аргументом. У разі використання лише одного аргументу, команда ***get***(*Hp*2) дасть перелік всіх властивостей об'єкта та їх (раніше встановленні) значення; команда ***set***(*Hp*2) дасть не лише перелік властивостей, а також всі можливі значення для них (наприклад: *LineStyle*: {'-' '--' ':' '-.' 'none'} – можливі стилі для кривої). Аби надати об'єкту певної властивості, треба використати, після хендлу, ще кілька пар аргументів – властивість та її значення, як у (8.1).

Важливо додати до (8.1): колір кривої можна задавати не лише у формі *set*(*Hp*2,'*Color*','***r***'), а також за допомогою числового вектора *set*(*Hp*2,'*Color*', **[.5 .1 .5]**). У такому разі використовується схема RBG, **[*r*, *b*, *g*]**, де у квадратних дужках – позитивні числа менше 1, доля *red*, *blue* та *green* у кольорі.

Можуть здивувати такі властивості, як "*Children*" та "*Parent*" (діти, батьки). Та трохи здорового глузду: "*Parent*" для кривої є осі фігури та сама фігура. Остання, скоріше за все, має номер 1 – отже, її хендл є один. Отримаємо властивості рисунка:

```
>> get(1)   .
```

Їх майже 60! Одна властивість нам знайома, '*Color*'. Командою *set*(1, '*Color*', '***y***') робимо жовтим всю рамку навколо осей графіка. (А як пофарбувати в середині? Дивіться далі). Найчастіше використовуються ще такі властивості вікон *Figure*: '*MenuBar*', '*Pointer*', '*Position*', '*Resize*' та '*Visible*'. Командами на зразок ***set***(1,'*Pointer*') довідайтесь їх можливості та отримайте задоволення, пограйтеся з ними!

Якщо на екрані у вас кілька фігур Figure, кликніть мишкою на одній з них – вона стане *current* (виділеною). Командою *gcf* (**g**et **c**urrent **f**igure) можна узнати її номер, її хендл. Тепер саме вона підкоряється вашій волі, і ви можете надавати їй різноманітних властивостей командами ***get*** і ***set***!

На фігурі *Figure* розміщуються осі, *axis*; вони можуть бути невидимими (після команди *axis off*), та вони є! Саме на осях "тримаються" криві, графіки, текст тощо. Саме вони є *Parent* для графіків рисунка. Доступ до них можна отримать через хендл *gca* (**g**et **c**urrent **a**xis). Проведіть дослідження, як рекомендовано в задачах 8.7–8.8, на які властивості осей ви можете впливати.

У цьому посібнику ми широко використовували графіку, картинки для дослідження математичних питань. А тепер графічні об'єкти будемо використовувати як засоби управління програмами, для створення графічних інтерфейсів, **G**raphical **U**ser **I**nterfaces, GUI.

8.3. Готові GUI-елементи

Вже діалогові програми, зроблені раніше, справляли враження розумних. Тепер навчимося робити ще й по-сучасному красиві програми. MATLAB має велику кількість вже підготовлених команд, за допомогою яких можна прикрасити будь-яку з ваших програм. Ось майже повний перелік:

uisetcolor, uiputfile, fwrite,
save, saveas, errordlg,
warndlg, helpdlg, msgbox, (8.1)
questdlg, inputdlg, uigetfile,
uiputfile, waitbar .

Найважливіші ми тут обговоримо; решту пропонуємо дослідити самостійно.

8.3.1. Гра з командою *menu*. Почнемо ще з однієї команди MATLAB, що дозволяє нам створювати різноманітні графічні оболонки для діалогу, з команди ***menu***. Надамо такий приклад:

```
>> MyChoice=menu('Яка програма краще?', . . .
              'MATLAB', 'MathCAD', 'Maple', 'Matematica', 'Excel')
```

Така команда створює красиве меню (рис. 8.5), яке дозволяє обрати один з кількох запропонованих елементів. Заголовок меню утворюється першим аргументом команди (цей надпис

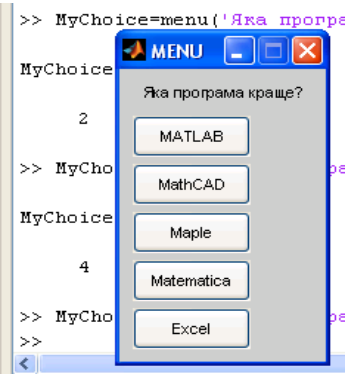

Рис. 8.5. Меню вибору.

слід брати у лапки, як і всі наступні). Наступні аргументи команди *menu* утворюють перелік кнопок (елементів вибору). Кожну з кнопок можна натискувати. У результаті змінна *MyChoice* набуває того чи іншого цілого значення від 1 до 5. В залежності від значення, командами логіки можна передбачити ті чи інші дії у програмі. Сподіваємось, це зрозуміло. Зараз головну увагу приділимо можливостям керувати цим об'єктом.

Можна його „зберегти" задля подальших експериментів. Для цього введіть у командному рядку комбінацію клавіш <Ctrl–C>. Створене меню стає „відокремленим" від командного рядочка. Так можна створити кілька меню одночасно. Всі вони „симулюють" можливість якихось дій, кнопки зберігають „пружні" властивості, їх можна натискувати – нічого, однак, не стається: знищити їх неможливо, змінити розмір неможливо. Та все ж їх хендли існують, і дають можливість ними керувати!

Кликнемо мишкою по одному з цих „мертвих" меню – воно стає активним. Тепер команда *get*(***gcf***) повідомить про величезний перелік властимостей, якимо можна управляти, а команди

```
>> set(gcf,'Color', 'y')
>> set(gcf,'Resize', 'on')
>> set(gcf,'Name', 'Прохання відповісти:')
```

змінять колір вікна на жовтий, дозволять змінювати розмір та положення вікна меню, навіть намалювати новий заголовок 'Прохання відповісти:' замість "MENU". Аби узнати більше про цей GUI-елемент, запитайте допомогу з командного рядка *help menu*, або у Help-браузері за ключовими словами "menu" або "Uicontrol Properties".

8.3.2. Три діалогових вікна *-dlg*; команда *msgbox*. Продовжимо наші експерименти-дослідження з трьома командами *errordlg*, *warndlg*, та *helpdlg* зі списку (8.1). Вони мають схожий "мінімальний" формат

```
Hdlg=errordlg('Dialogue Text','Title Text'),
  Hdlg=warndlg('Dialogue Text','Title Text'),                (8.2)
Hdlg=helpdlg('Dialogue Text','Title Text').
```

Рис. 8.6 демонструє результати запуску цих команд (з командного вікна або з "середини" якоїсь іншої програми. Графічні вікна відрізняються іконкою – значками помилки, попередження, просто інформації про щось. Натискання на ОК дозволяє програмі працювати далі. Легко зрозуміти, що перший аргумент в (8.2) – це текст повідомлення, 'Dialogue Text', а другий аргумент 'Title Text' – заголовок вікна. Хендл *Hdlg* може бути використаний для управління властивостями вікна, наприклад командою ***set***(*Hdlg*,'Color', 'm') безпосередньо за (8.2).

Команда *msgbox* нібито узагальнює попередні діалогові команди. Окрім формату (8.2) на зразок

```
>>Hdlg=msgbox({'Повідомлення str1', …
               'Повідомлення str2',…
               'Повідомлення str3'},…
                    'Заголовок');
              set(Hdlg,'Resize','on')
```

(з Повідомленням у три рядки та вікна із розміром, що змінюється), тут є можливість або застосувати ті самі іконки *Error*, *Warning* або *Help*, або будь-яку власну. Це розширює можливості програміста. Для створення власних ('custom') іконок є дві можливості. Перша – застосувати 3-вимірний вектор кольорів, наприклад

```
>>Icon(:,:,1)=[0 1;0 1];Icon(:,:,2)=[.8 1;0 .3];Icon(:,:,3) = [.3 0; .9 .5];
>> h=msgbox('Повідомлення','Заголовок','custom',Icon);
```

(результат показано на рис. 8.7*а*) Друга можливість – створити невеличку картину (в Paint, наприклад *Home.bmp*),

```
>>cdata=imread('c:\Users\Home\Documents\Home.bmp');
```

Рис. 8.6. Три форми діалогу: помилка, попередження, питання

Рис. 8.7. Вікна ***msgbox*** із власною іконкою.

```
>>  ^Перетворили в mat-формат, вказали шлях
>> msgbox('Повідомлення', 'Заголовок', 'custom',cdata);
```

(результат на рис. 8.7*б*). Більше інформації – в Help.

8.3.3. Команда *questdlg* має зовсім інший формат:

Answer= *questdlg*('Питання','Заголовок','str1','str2','str3', 'str2').

Її результат показаний рис. 8.8. З нього зрозуміла роль першого та другого аргументів. Команду корисно використовувати, коли треба обрати з трьох або двох варіантів; значення варіанта (кнопки, що натиснуто) отримує змінна *Answer*; наприклад *Answer*='str2'. Далі слід використати блок *switch–case–end*, аби його врахувати. Останній аргумент має обов'язково повторювати один з попередніх – відповідна кнопка буде виділена (друга на рис. 8.8). У такому разі *Answer* не є хендлом, і тому не може бути використаний для управління вікном.

8.3.4. Команда *inputdlg* використовується для ввода даних через графічне вікно:

```
>> answer=inputdlg('Введіть потрібне значення:')
        answer =     '120'
```

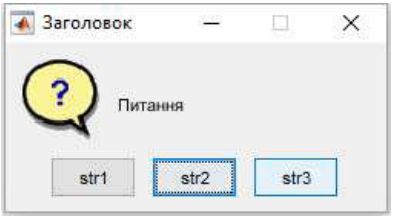

Рис. 8.8. Вибір з трьох варіантів

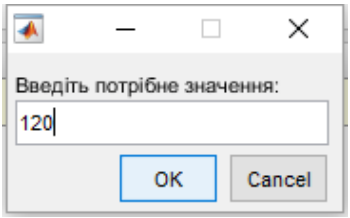

Рис. 8.9. Вікно ***inputdlg***.

Робота команди, роль аргументів зрозуміла з рис. 8.9. Зрозуміло також, що якщо потрібно ввести текстові дані – працюємо з цією командою безпосередньо. Якщо ж потрібно число, як на рисунку, то треба ще застосувати його перетворення у текст, *Num=str2double*(*answer*).

8.3.5. Команди *uisetcolor*, *uigetfile*, *uiputfile*. Ці команди видають вікна, відомі вам з інших програм Microsoft Windows і Microsoft Office. Бачимо:

>> *Color=**uisetcolor***;

(рис. 8.10). Відповіддю буде колір у вигляді вектора, як *Color*= 0.9922 0.9176 0.7961 (система кольорів RGB).

Команда ***uigetfile*** дозволяє відкрити той чи інший файл:

>> [*FileName*, *PathName*] = ***uigetfile***
*FileName=Galaxy*4.*m*
PathName=C:*Users**Home**Documents**MATLAB*\

(рис. 8.11А). При цьому у додатковому вікні праворуч можно цей файл читати, якщо він текстовий, як *m*-файли. Відповіддю будуть ім'я прочитаного файла та шлях до нього. Аналогічно, команда ***uiputfile*** дозволяє якийсь підготовлений об'єкт зберігти у файлі, рис. 8.11Б. Важливо, що обидві команди допускають "фільтри" в числі аргументів – це дозволяє бачити не усі файли, а з певним розширенням; див. ***help*** *uiputfile*. Наприклад, ви створюєте бази даних (див. розділи 6.4.1, 6.4.2, 6.6) і хочете записувати їх лише у файли з вашим власним розширенням ***.****MyB* (лише з трьох літер!).

Для розробки складних програм мужуть бути корисними й інші готові програми MATLAB – *uisetfont, fwrite* – розберіться з ними самостійно.

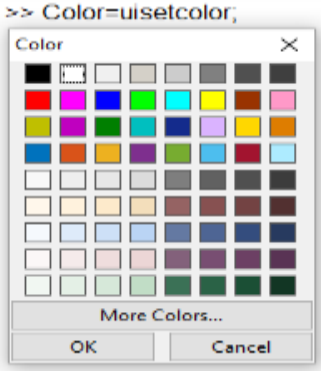

Рис. 8.10. Команда uisetcolor

8.4. Граємось з *uicontrol*; програмування подій

MATLAB-колекцію елементів управління (їх називають *controls*), з яких можна складати власні графічні програми, показано на рис. 8.12. Не треба їх вчити напам'ять. Давайте „пограємось" ними один за одним. Цьому присвячено наступні підрозділи. Як і раніше, це буде не проста гра. Вона відкриє нам нові команди MATLAB, нові властивості його об'єктів і нові типи даних.

8.4.1. Гра з командою *uicontrol*. Починаємо з навчальної гри з командою *uicontrol* (можна здогадуватись, що назва цієї команди складена зі слів ***control***, управління, і ***ui***, від ***U***ser ***I***nterface). Її виконання у командному вікні MATLAB

```
>> Hbut=uicontrol                                    (8.3)
```

створить пусте графічне вікно. Та найважливішим є те, що зліва внизу на рисунку є елемент управління – за умовчанням, стилю *pushbutton* (кнопка для натискання)! Її можна *push* – натискувати, вона кожного разу піддається назад, немов на пружині; її номер 1 в колекції рис. 8.12. Одночасно команда (8.3) видає деякі властивості хендла *Hbut* до цієї кнопки (бо крапки з комою немає). Командою *get*(*Hbut*) або *get*(*gco*) отримаємо всі властивості повністю. У цьому переліку багато таких, що вже знайомі, або про призначення яких легко здогадатися. Тут і починаємо нашу "гру". Виконаємо, наприклад, команди

```
>>set(Hbut, 'BackgroundColor', 'y')
```

Рис. 8.11. Команди (*а*) ***uigetfile*** та (*б*) ***uiputfile***

>>*set*(*Hbut*, 'ForegroundColor', 'b', 'String', 'Start!')

Перша з них розфарбовує кнопку у жовтий колір, а друга – створює на ній надпис **Start**! синього кольору. Командою

>> *set*(*Hbut*, 'FontSize', 12, 'FontWeight', 'bold')

робимо надпис більш виразним. Тут треба враховувати, що ще більший 'FontSize' може потребувати також зміни властивості 'Position'. Наприклад,

>> *set*(*Hbut*, 'Position', [20 20 100 30])

(вікно стало більшим). У числовому векторі перша пара чисел – *x*- та *y*-координати нижнього лівого кута GUI-елементу, друга пара – верхнього правого кута. Можна перевірити *Units*, одиниці вимірювання, у яких відкладаються названі розміри: за умовчанням це *pixels*.

Тепер – більш принципова властивість *Style*. Командою *set*(*Hbut*, 'Style') узнаємо повний перелік можливостей:

pushbutton | *togglebutton* | *radiobutton* | *checkbox* | *edit* | *text* | *slider* | *frame* | *listbox* | *popupmenu* (8.4)

За умовчанням це була *pushbutton* (кнопка, яку можна натискувати), а тепер маємо можливість змінити такий GUI-елемент на інший, наприклад

>> *set*(*Hbut*,'Style','radiobutton'),

або, краще, з самого початку замовити потрібний *Style*:

>> *Hbut*=*uicontrol*('Style','radiobutton')

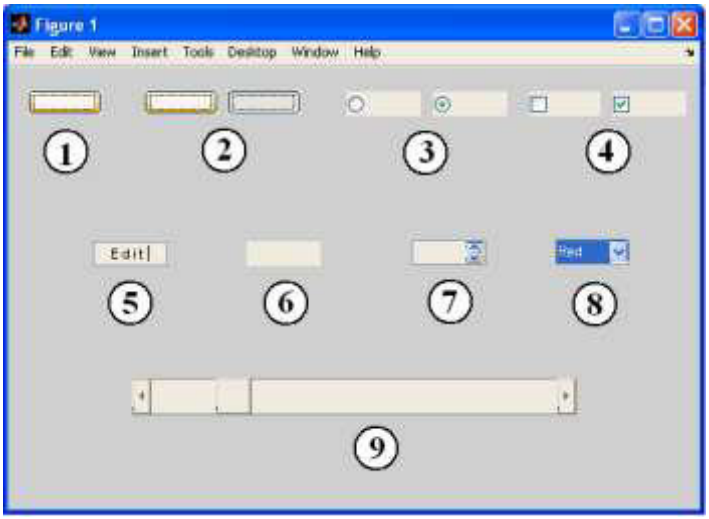

Рис. 8.12. Різновиди елементів GUI, які складають колекцію MATLAB: 1. *pushbutton*; 2. *togglebutton*; 3. *radiobutton*; 4. *checkbox*; 5. *edit*; 6. *StaticText* – незмінний надпис; 7. *listbox*; 8 *popupmenu* – меню, що випадає; 9 *slider*.

Курсором миші можна поставити у *radiobutton* чорний кружочок, можна убрати. Цей GUI-елемент позначено номером 3 на рис. 8.12. Аналогічним чином можна створити *чек-бокс* (номер 4 на рис. 8.12). У чек-боксі можна ставити „галочку", а можна її убирати. Нічого іншого не трапиться... Поки ще!

Проведемо ще один експеримент:

```
>>Hbut=uicontrol('Style','slider','BackgroundColor','y',. . .
                                        'Position', [60 30 460 30])
```

(саме таке значення для Position підібрано емпірично). Трохи, додатково, пофарбуємо:

```
>> set(Hbut, 'BackgroundColor', 'y')   .
```

Для такого GUI-елементу стилю ***slider*** є актуальними ще такі властивості: *Max* та *Min* (за умовчанням дорівнюють 1 та 0 відповідно), *SliderStep* (за умовчанням $\frac{1}{100}$). Можемо встановлювати й інші якості GUI-елементів черех їх хендл,

Hbut наразі. Інші стилі GUI-елементів з списку (8.4) пропонуємо дослідити самостійно.

Властивість *String* , яка робить надпис на GUI-елементі, треба використовувати особливим чином, коли GUI-елемент має стиль *listbox* або *popupmenu*! Адже треба створити не просто надпис, а кілька рядків (припустимо, 5) з надписами *Надпис*1, *Надпис*2, . . ., *Надпис*5 на них. Усі ці надписи, що разом задають значення для *String*, треба записати як один, у лапках, однак із розділювачем | між ними. Ось так:

. . . **,** ‘ *String* **’, ‘***Надпис1* | *Надпис2* | . . . | *Надпис5* **’,** . . .

Не всі властивості GUI-елементів так легко використовувати. Радимо провести власні експерименти з цими командами та відобразити їх у лабораторній роботі № 12. А от властивість ***Visible***, сподіваємось, вам сподобається... Спробуйте!

8.4.2. Програмування подій. Прийшов час *to make our GUI alive* (оживити наше GUI)! За таке відповідає властивість ***Callback***. Проілюструємо її використання на прикладі меню, що дозволяє обрати ту чи іншу функцію та побудувати її графік.

Створимо картинку з самого початку:

```
>>figure, Hbut=uicontrol('Style','popup', ...
                         'String', 'Обери функцію:', ...
                                   'Position',[20 20, 110 320])
```

Властивість *String* є обов’язковою, якщо йдеться про GUI-елемент стилю *popup* (меню, що випадає); значення властивості *Position* підібрано таким чином, аби дана та подальші надписи були весь час видними. Тепер трохи підкоригуємо властивість *String*:

```
>>set(Hbut,'String', 'Обери функцію: |sin(x)|sin(x)/x|x^2 |abs(x)' )
```

Можливість вибору стала більшою! І тепер починаємо головне: властивість ***Callback*** пов’язуємо з іменем функції, яку хочемо виконати. Наприклад:

```
>> set(Hbut, 'Callback', 'ChosenFunction(Hbut)' )          (8.5)
```

Тепер треба створити *m*-файл цієї функції *ChosenFunction.m*, що робимо в редакторі *m*-функцій:

Лістінг *ChosenFunction.m*

```
function ChosenFunction(H)
%Функція для завдання Callback,
%що будує 3 графіки за вибором.
%Copyright Ye.Gayev, Apr. 2016.
val = get(H,'Value');
if val == 1%^бо val =1 відповідає тексту, Заголовку
     %картинку перетворюємо у mat–файл cdt:
    cdt=imread('c:\Users\Home\Documents\Home.png');
    msgbox({'Треба обрати одну з кількох функцій', . . .
                '     та повести подальший діалог!'}, . . .
                   'Помилка!', 'custom', cdt);
    return %Закінчуємо роботу у цьому випадку
elseif val==2
     x=GettingData(H); %Підпрограма нижче!
     F=sin(x),; Name='sin(x)';
elseif val == 3
     x=GettingData(H); %Спільна підпрограма нижче!
     F=sin(x)./x;  Name='sin(x)/x';
elseif val == 4
     x=GettingData(H); %Спільна підпрограма нижче!
     F=x.^2;  Name='x^2';
elseif val == 5
     x=GettingData(H); %Спільна підпрограма нижче!
     F=abs(x);  Name='|x|';
end
plot(x,F,'-o'); %<Будуємо графік у визначеному диапазоні x
title(['Замовлено графік функції F=', Name])
xlabel('x, аргумент'), ylabel('y, функція')
 %кілька рядків пропуску аби легше читати

function x=GettingData(H)
%Підпрограма до головної програми зверху, у тому ж файлі
Min=inputdlg('З якого X починати графік:');
Max=inputdlg('Яким X закінчити:');
Min=str2double(Min); Max=str2double(Max);
N=inputdlg('На скільки частин ділити інтервал:');
N=str2double(N); %Бо інакше N – текст
Dx=(Max-Min)/N; %Крок побудови графіку
x=Min : Dx : Max; %Аргумент для побудови графіку
```

На рис. 8.13 показано фрагмент працюючої програми: рисунок з GUI-елементом у вигляді меню, що випадає, та графік функції, побудованої за попереднім діалогом. Принцип роботи, таким чином, наступний: обраний рядок *PopUp*-меню генерирує той чи інший номер від 1 до 5. Він передається у функцію, що указана у *Callback*. В залежності від нього будується графік тієї чи іншої фунції.

Зауваження: Тут можлива проблема для обчислення "*F=sin*(*x*)*./x*". Та зараз відволікатися на її подолання не будемо.

8.5. Універсальний GUI-конструктор *guide*

Тепер ми багато знаємо про „цеглинки", з яких можна складати GUI-програми. Отже, читачу, ви готові братися за першу таку СПРАВЖНЮ програму? Починаємо!

Насправді, все складне ви вже вивчили. Лишається навчитися "огортати" програми у GUI-одяг. Оберемо вже створену "звичайну" програму (тобто, що запускають з командного рядочка) ***helicopter*** та обернемо її у GUI-одяг. Новій програмі надамо імя ***helicopter_GUI*** (одне слово!). Доцільно слідувати наступним крокам.

8.5.1. **Планування GUI-програми.** Розпочати слід з складення загального плану програми: програма *helicopter_GUI*

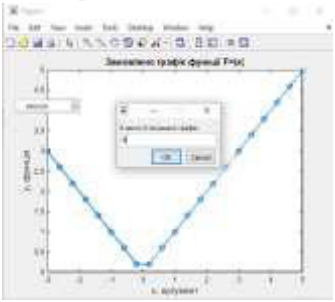

Рис 8.13. Остаточна програма з використанням властивості *Callback.*

(*i*) має обертати відрізок за годинниковою стрілкою або проти – для цього підходить *radiobutton*;

(*ii*) швидко чи повільно – можна обирати через *slider*;

(*iii*) того чи іншого кольору – обирати через *listbox* або *popupmenu*;

(*iv*) тонкий відрізок чи товстий – товщину можна ввести у вікно *EditText*.

(*v*) Нарешті, запускати через

PushUp-кнопку.
(*vi*) Та ще потрібні надписи над GUI-елементами, для заголовока програми (*StaticText*).
Зрозуміло тепер, які елементи треба передбачити.

8.5.2. **Дизайн GUI-програми.** У сьогоднішньому "серійному виробництві" програм мало хто використовує лише "цеглини", про які була мова раніше. "Будинок росте" швидче й дешевше, якщо використовувати певні блоки. Так само і в нас: редактор GUI-інтерфейсу ***guide*** дозволяє складати Graphical User Interface певними блоками. Познайомтеся:

>> guide

(*гайд* у перекладі з англійської означає *путівник*). У вікні, що з'явиться, треба обрати закладку[28] „Create New GUI" з опцією „Blank GUI (Default)" та натиснути кнопку OK. З'явиться вже сам GUI-редактор. Його вікно виглядає так, як показано на рис. 8.14. Зверніть увагу на помітку 1: заголовок нашої майбутньої GUI-програми – ***untitled*** (не назване). На верхній горизонтальній панелі розташовані певні кнопки меню 2 – з ними познайомимось пізніше. На вертикальній панелі 3, ліворуч – вибор певних GUI-кнопок з колекції (8.4). Про їхнє призначення можна здогадатись, розглядаючи смислові малюнки на них, або наводячи на них указник миші, бо з'являється підказка. Усі їх назви нам вже знайомі, окрім трьох останніх. Але ж ті нам поки що не потрібні. Наводячі мишу на кут 4 вікна GUI-редактора можемо змінити розмір майбутньої GUI-програми. Робимо це відразу, враховуючи кількість потрібних GUI-елементів, або пізніше, якщо їм стане тісно.

Головне **робоче поле** 5 для дизайну GUI-програми розділено на клітинки. Це зручно для розташування GUI-елементів один відносно другого.

З колекції GUI-елементів 3 обираємо GUI-елемент "*Static Text*", натискуємо його та „розтягуємо" на робочому полі. Розтягнули на усю ширину поля 5, від лівого краю до

[28]Порівняйте з 8.5.5.

правого – так буде красивше. Майбутній GUI-елемент майже непомітний, хіба що закриває клітинки робочого поля та має надпис “Static text”.

Так само відшуковуємо серед GUI-елементів „*Radio Button*” та розташовуємо її, *button* (*кнопку*), на робочому полі. Ми обрали положення праворуч уверху. GUI-елемент весь час відмічено маркерами: якщо „потягнути” за маркер, можна змінювати розмір та положення GUI-елементу, що має надпис „Radio Button”.

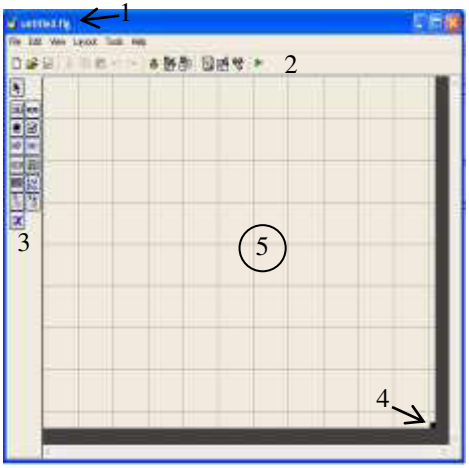

Рис. 8.14. Робоче вікно GUI-середовища.

Зупинимось покищо із створюванням нових GUI-елементів, попрацюємо з цими двома. Зробіть подвійний щиглик на якомусь з них – на елементі “*Static Text*”, наприклад. (Можна піти й іншим шляхом – натиснути на іконку  з меню інструментів 2). З’явиться ***Property Inspector*** (*Інспектор властивостей*) для даного GUI-елементу, рис. 8.15. У ньому – довжелезний перелік властивостей даного елементу, багато з яких вже відомі. Так, ті самі, що видавали у командне вікно команди *set* і *get*. Клікніть мишею на властивості *BackgroundColor* наверху вікна та оберіть колір, який подобається. Ми обрали синій. Кнопка OK, і маємо синю „шапку” у нашому GUI-вікні. Шукаємо властивість String, де за умовчанням стоїть “Static Text”. Змінюємо текст на такий: „Введіть потрібні параметри на натисніть <Обертати!>” – клацніть мишею десь у іншому місці Інспектора, і цей надпис з’явиться у шапці програми. Тобто розміщуємо тут головну інформацію для користувача. Тепер властивостями *FontSize*

(розмір шрифту), *FontWeight* та *ForegroundColor* (колір надпису) зробіть цей надпис красивішим, жовтого кольору. До важливої властивості треба буде ***Tag*** ще повернутися.

Переходимо на GUI-елемент „*Radio Button*". Інспектор властивостей переключається на нього і дозволяє так само обрати кольори для усієї панелі радіо-кнопки та для надпису; тут, однак, обираємо *String*="За годинником", як було запроектовано у 8.5.1. Ще буде потрібним і ***Tag*** для цього GUI-елементу.

Рухаємось далі. На робочому полі розташовуємо місце для виводу майбутнього графіка (гвинта, що обертається) – GUI-елемент *Axies*. На новому білому (за умовчанням) полі з діагоналями, що перетинаються, написано за умовчанням „Axes1" (*Oci*1, бо у якійсь іншій GUI-програмі Ви, може, захочете ще й другі, й треті Осі мати). Клацаємо мишею на осях – і вже Інспектор властивостей готовий до послуг саме для цього GUI-елементу. Ми, однак, нічого не змінили.

Тепер обираємо *Pop-Up menu* (**4**) для вибору кольору гвинта. В Інспекторі властивостей для цього GUI-елемента головним є замість String="Pop-up Menu" (за умовчанням) задати таблицю вибору:

Оберіть колір:
red
yellow
green
blue
magenta (8.6)

Ми написали англійські назви кольорів (як їх надає команда *help plot*), це трохи спрощує майбутнє програмування. Але ж можна запрограмувати й для назв українською...

Перший рядок „Оберіть колір:" з'явився у вікні цього GUI-елемента. А де ж інші рядочки? Вони з'являться після „оживлення" нашої GUI-програми, див. Етап 2. Також ми знов надали значень**:** *BackgroundColor* жовтий та *ForegroundColor* червоний кольори.

Повзун „*Slider*" (9) з естетичних міркувань краще розмістити під вікном для графіків *Axes*, та нехай ще його

довжина співпадає з ним. Інспектор властивостей для цього GUI-елемента має тіж самі назви. На цей раз найбільш важливими з них є властивости *Max* та *Min*. Потім, коли *Slider* запроцює, він буде видавати *Value* (*Значення*) між *Min* та *Max*, тобто $Min \le Value \le Max$. Ці властивості *Max* та *Min* можна залишити незмінними.

Однак створені нами GUI-елементи не працюють. Чому? Тому, що ми знаходимось ще у GUI-редакторі, де весь час зберігається можливість іхньої зміни (редагування). Зробимо зупинку та перевіремо, як працює GUI-програма на цьому етапі. Серед іконок горизонтального меню 2 обираємо зелену стрілку ***Run*** (побігли) – нам пропонується зберігти роботу з розширенням ***.fig***. Зберігаємо нашу роботу з іменем *helicopter_GUI*. Додатково виникає редактор *m*-файлів з програмою *helicopter_GUI**.m*** – нею будемо займатись пізніше. А також виникає „жива" графічна оболонка – у ній усі GUI-елементи працюють, рис. 8.16. Все гаразд з програмою, чи треба щось змінити? Ви задоволені естетичним виглядом програми? Якщо так, рухаємось далі.

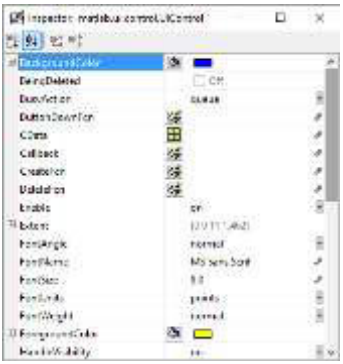

Рис. 8.15. Inspector Properties для одного з GUI-елементів

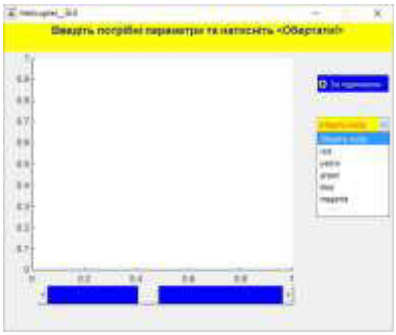

Рис. 8.16. Напівготова GUI-програма

8.5.3. **Редагування GUI-програми.** Наша програма напівготова. Продовжуємо роботу над нею у середовищі *guide*. Додаємо вікно стилю EditText, куди будемо вводити ціле число, ширину "гвинта". Із властивістю String інспектора можна обійтися одним з двох способів: або задати

її як ‘*Введіть ціле*’ замість ‘*Edit Text*’, або ввести конкретне ціле ‘за умовчанням’, наприклад ‘10’ (далі врахувати, що це все ж-таки текст!). Робимо надпис зверху “Ширина гвинта” за допомогою GUI-елемента стилю StaticText. Так само нижче слайдера надписуємо “Швидкість обертання”. Лишається зробити кнопку з надписом “Start!” (зрозуміло, вона є стилю *PushButton*); колір та розмір шрифту підбирайте самі з естетичних міркувань. Аби усі GUI-елементи розташовувалися красиво один до другого, слід використовувати іконку меню  (Align Objects, вирівняти об’єкти).

Збереження роботи (або через ту саму зелену стрілочку, або натисканням іконки “Дискета”) додає потрібні зміни у файли *helicopter_GUI**.m*** *та helicopter_GUI**.fig***. Тепер *guide*-середовище можна закрити, а “оживити” нову GUI-програму з командного рядка

```
>> Helicopter_GUI
```

Етап дизайну закінчився. У щойно створеної красивої графічної програми „все рухається”, та… нічого не стається! Це тому, що MATLAB ще не знає, чого ми хочемо з цією програмою, ще не запрограмували ніяких дій.

8.5.4. **Програмуємо GUI-елементи**. Розглянемо файл *helicopter_GUI.m*, що з’явився у редакторі *m*-файлів. У програмі більше за 150 рядків. Проаналізуємо його.

1. Програма починається з рядка

```
function varargout=Helicopter_GUI(varargin)
```

Це зрозуміло**:** якийсь вхідний аргумент `varargin` пов’язано з вихідним аргументом `varargout` через ім’я щойно створеної нами програми ***`Helicopter_GUI`***. (Вказані імена аргументів[29] складаються з трьох англійських слів – `Variable Argument`, *змінний аргумент*, та `In` (*вхідний*), чи `Out` (*вихідний*). Вони грають особливу роль у MATLAB-програмуванні. Про це можна прочитати в [5]. Нам же ці подробиці поки що не потрібні).

[29]точніше, довільна кількість аргументів, див. *help varargin*

Наступні рядочків 20 – це коментарі англійською мовою. Вони призначені для нас, створювачів програми, та нічого не скажуть майбутньому юзеру. Тому їх доцільно замінити на свій коментар, на зразок

```
% Це графічна програма,
% яка обертає "гвинт" навколо центру.
%Напрямок обертання, колір та ширина "гвинта",
% швидкість обертання
%мають бути обрані у відповідних вікнах.
%Обравши ці дані, натисніть кнопку "ОБЕРТАТИ!"
%Запускати з командного рядка:
% >> Helicopter_GUI%
%Copyright Гаєв Є.О., лютий 2008.
```

Текст буде видано користувачеві, якщо він запитає в MATLAB допомогу

>> help Helicopter_GUI

Здається, що такий коментар надасть йому вичерпну інформацію, як з програмою працювати. Завжди супроводжуйте ваші програми грамотним та вичерпним коментарем!

2. Далі іде порція програмного кода, що починається з коментаря

```
% Begin initialization code - DO NOT EDIT
```

та закінчується коментарем

```
% End initialization code - DO NOT EDIT
```

Дослухаємось поради „**DO NOT EDIT**”, тобто „**НЕ РЕДАГУВАТИ**” порцію коду між коментарями.

Ще далі автоматично розміщено декілька підпрограм, що починаються ключовим словом ***function***. Частину з них теж не радимо змінювати. Йдеться про функції *Helicopter_GUI_* з закінченнями *_OpeningFcn* та *_OutputFcn*. Їх теж не чипаємо.

А от частину – необхідно. Усі вони пов’язані із створеними нами GUI-елементами, і таке знання допомагає їх знайти. Переглянемо назви наступних функцій. В них фігурують *radiobutton1*, *popupmenu1*, *slider1*, *edit1*, та *pushbutton1* – зрозуміло, що назви згенеровано з урахуванням обраного нами стилю GUI-елемента. Якби

radiobutton, для прикладу, використовувалась б двічі чи тричі, їх функції були б названі `radiobutton2` та `radiobutton3`. Більшості GUI-елементів відповідають дві функції, з закінченнями `_CreateFcn` або `_Callback`. Так от, останні нам і потрібні! Кілька GUI-елементів мають лише одну `Callback`-функцію (*radiobutton* та *pushbutton*). GUI-елемент стилю *axes* її взагалі не мають.

Ви ще не забули, за що який GUI-елемент відповідає? Пропонуємо навести тут порядок. Запускаємо *guide*-середовище знову, та цього разу через закладку *Open Existing GUI* обираємо нашу програму. Знов бачимо середовище розробки з макетом програми, знов переглядаємо Property Instector для кожного із створених GUI-елементів. Та цього разу працюємо з властивістю *Tag* (*ярлик*, "*кликуха*" у перекладі). Її призначення зрозуміло з перекладу. Надаємо "кликуху" кожному з елементів таким чином, аби відобразити як його стиль (походження), так і призначення. Пропонуємо такі значення властивості *Tag* кожного з GUI-елементів у порядку їх створення: *radioDirection*, *popupColor*, *editWidth*, *sliderSpeed*, *pushStart*, *axesGraphWindow*. Зберігаємо зміни. Бачимо, що вони враховані в *m*-файлі *helicopter_GUI.m*. Тепер функції `_CreateFcn` та `_Callback` мають приставки, що відображають їх призначення, наприклад `pushStart_Callback`.

3. З останньої й почнемо. "Start!" – головна кнопка програми, із нею ми пов'язуємо виконання певних дій, а саме: обертання відрізку ("гвинта") у певному напрямку, заданого кольору, ширини, із обраною швидкістю, як це було у програмах *Helicopter*, розділи 5.1.1 та 5.1.2. Тому, не звертаємо уваги на коментарі цієї підфункції (лише цієї!) та вміщуємо за ними безпосередньо наступний код, позичений у 5.1.2:

```
function pushStart_Callback(~, ~, ~)
    %global Direction  Color  Width  Delay
    Direction=1;%Всі ці значення
    Color='r';     %надалі закоментуємо
    Width=10;
```

```
    Delay=.1;
for Fi=-Direction*(0:pi/100:2*100*pi)
      x1=cos(Fi);      y1=sin(Fi);
      x2=-x1;          y2=-y1;
      Ox=0;  Oy=0;%Для побудови центра обертання, “гвіздка”
      plot([x1, x2], [y1, y2], Color, 'LineWidth', Width);
      axis([-1.2,1.2,  -1.2,1.2]),
      axis off     %Убрати осі, які на разі зайві
     hold on, plot(Ox,Oy, 'ko', 'MarkerSize',5), hold off
     pause(Delay) %Delay впливає на швидкість “обертання”
end
```

Головна частина коду між ***for*** та ***end*** зрозуміла на підставі підрозділів 5.1.1 – 5.1.2. Та потрібно надати якісь дані для *Direction*, *Color*, *Width*, *Delay*, аби програма ***Helicopter_GUI*** почала працювати. Для її зупинки ми змушені переривати її силою (поки що!), <***Ctr–C***> **(:-**

Та бачимо, що поки що програма не звертає уваги на інші елементи управління…

4. Тепер треба „прокласти комунікації” між GUI-елементами, і це робимо програмуванням функцій, що задають *Direction*, *Color*, *Width*, *Delay*. У попередньому коді закоментуємо рядки, що їх задають, та розкоментуємо рядок

```
global Direction  Color  Width  Delay
```

тобто надаємо наказ “бери ці дані зі спільної області пам’яті ***global*** ”. Тепер працюємо з кожною функцією, що відповідає за вказані величини.

За напрямок обертання відповідальною є функція radioDirection_Callback. Тут стає до нагоди автоматично згенерована підказка (hint) з коментаря

```
% Hint: get(hObject,'Value') returns toggle state of radioDirection
```

На її підставі пишемо функцію у такому вигляді:

```
function radioDirection_Callback(hObject, ~, ~)
% hObject    handle to radioDirection (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% Hint: get(hObject,'Value') returns toggle state of radioDirection
global Direction
Direction=1;
```

```
Dir=get(hObject, 'Value')   % ;
if Dir==0
   Direction=-1;
end
```

Стається наступне: команда *get* визначає значення, що ми передаємо через GUI-елемент, і надає його змінній *Dir*; якщо воно дорівнює нулю (крапка не поставлена), то *Direction* надається значення -1; якщо *Dir* інше (а може бути лише 1 або 0), то "за умовчанням" *Direction*=1. Визначене значення має бути занесене до спільної області пам'яті. На цьому робота функції закінчується. (Поведінку *Dir* будете бачити у командному вікні; після відлагодження функції за командою *get* треба поставити крапку з комою, про що підказує коментар).

Аналогічно кодуємо роботу функції, що відповідає за обрання кольору:

```
function popupColor_Callback(hObject, ~, ~)
% hObject    handle to popupColor (see GCBO)
% eventdata  reserved - to be defined in a future
version of MATLAB
% handles    structure with handles and user data
(see GUIDATA)
% Hints: contents = cellstr(get(hObject,'String'))
% returns popupColor contents as cell array,
% contents{get(hObject,'Value')}
% returns selected item from popupColor
global Color
contents = cellstr(get(hObject, 'String'));
Color= contents{get(hObject, 'Value')};
```

Тут відбувається таке: *подія*[30] (обрання кольору через GUI-елемент) викликає передачу у змінну *contents* все те, що ви написали для властивості String (перша команда *get*); ця змінна – комірка, у змінну *Color* з неї передається той конкретний рядочок (назва кольору), що обрано; вона засилається у спільну область пам'яті *global*. З останньої її

[30]Створювачі компіляторів назвали це *подія-орієнтоване* програмування.

отримує підпрограма *pushStart_Callback*, що й виконує потрібні дії.

Слід, однак, мати на увазі таке: якщо у переліку кольорів (8.6) запишемо літери не англійською мовою, то слід додати "перекодування" на зразок

```
if Color=='Червоний'
   Color='r';
elseif Color=='Зелений'
   Color='g';
elseif . . .
   . . . . .
end
```

За "ширину гвинта" відповідає функція *editWidth_Callback*. Тут записуємо такий, вже майже зрозумілий код:

```
   function editWidth_Callback(hObject, ~, ~)
   % hObject  handle to editWidth (see GCBO)
   % eventdata reserved - to be defined in a future version
of MATLAB
   % handles structure with handles and user data (see
GUIDATA)
   % Hints: get(hObject,'String') returns contents of
editWidth as text
   %str2double(get(hObject,'String')) returns contents of
editWidth as a double
   global  Width
   Width=str2double(get(hObject,'String'))
```

Команда *str2double* потрібна для того, аби текстову величину *get*(*hObject*,'String') у число.

Лишається задіяти слайдер. Як вже відомо, на швидкість обертання впливає як крок анімації у рядку *Fi=-Direction** (0:***pi***/**100**:2*100**pi*) підпрограми *pushStart_Callback*, так і аргумент *Delay* команди *pause*. Підемо останнім шляхом. Відповідний код має вигляд

```
function sliderSpeed_Callback(hObject, ~, ~)
% hObject    handle to sliderSpeed (see GCBO)
%eventdata reserved to be defined in
%a future version of MATLAB
%handles structure with handles and user data (see GUIDATA)
%Hints: get(hObject,'Value') returns position of slider
```

```
%get(hObject,'Min') and get(hObject,'Max')
%to determine range of slider
global   Delay
k=1;
Delay= k*get(hObject,'Value')    ;
```

На вашому комп'ютері швидкість обертання може стати занадто повільною або занадто швидкою. Це можна врегулювати коефіцієнтом, наприклад *k*=0.5. Через *get* ви можете отримати також найменше та найбільше значення, встановленні слайдеру Інспектором Властивостей, або самі їх змінити. Тому через змінну *Delay* ми маємо ліворуч найшвидче обертання, праворуч – найповільніше. Недолік такого підходу можна виправити, додавши код із перетворенням

```
k=1;     Delay=1/(k+Delay)    ;
```

Тоді для *k*=1 маємо 1< *Delay* <0.5; для *k*=10 маємо 10< *Delay* <1. Найкраще для вашого комп'ютера знаходимо шляхом "випробувань та помилок". Можна пошукати й кращого перетворення, якщо розумієтесь в математиці(**:-**.

Програма, в принципі, закінчена. Як вона виглядає в майже остаточному вигляді показано на рис. 8.17*а*, де стрілочками умовно показано обертання "гвинта". Подобається?

8.6. Випробування, налагодження, поліпшення GUI

Творчій студент завжди критикує: "а якщо інакше? Змінити..." – і це добре! Тим більше, що недоліки нашої програми лежать на поверхні. Дійсно, у ній ми не маємо можливості зупинити обертання. Натиснемо на хрестик праворуч, "закрити програму". Що ж стається насправді? GUI зникає, та гвинт продовжує обертатися уже у звічайному вікні! Як таке виправити? Лишається один, "насильницький", спосіб – натиснути комбінацію <Ctrl–C>…

Ось чому треба обговорити можливості поліпшення наших GUI-програм, їх випробування та налагодження.

Розглянемо, як можна додати ще кнопку "Stop!" та, головне, запрограмувати її. Спочатку через *guide* знову

відкриваємо “креслення” нашої розробки через закладку *Open Existing GUI*. Додаємо до GUI ще одну *pushbutton* “Stop!” та розміщуємо відносно попередньої. За допомогою Property Instector надаємо їй таких властивостей, аби у решті решт отримати приблизно таке, як показано на рис. 8.17*б*. Треба зберегти новий дизайн. Та спочатку важлива порада.

Порада програмісту: коли зробили суттєві виправлення у програмі, що працює, не поспішайте її зберігати, і тим “знищувати” попередній текст програми! Часто виявляється, що нова програма не працює, а попередня, де все було гаразд, вже не існує **(:-**… Виправлену програму краще “*Save as*… ”, “зберігти як” модифікацію, у нашому випадку, наприклад, як *helicopter_GUI*_**v2**. Тоді утворяться два нових файли *helicopter_GUI*_v2.***m*** та *helicopter_GUI*_v2.***fig***, де у текст першого буде автоматично додано лише одну нову функцію *pushStop_Callback*. Там напочатку лише коментарі. Який код там має бути?

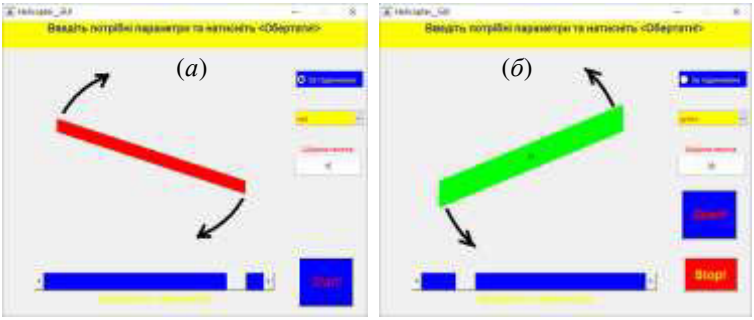

Рис. 8.17. Робоча програма *Helicopter_GUI* (*а*) та її модифікація *Helicopter_GUI*_v2 (*б*) з кнопкою “Stop!”.

Думаємо: чому “гвинт продовжує обертатися у звічайному вікні” після закриття програми? Очевидно, що продовжується виконання циклу, аж поки змінна циклу *Fi* не зробить всі призначені їй кроки 0 : *pi*/100 : 2*100, тобто 100 повних обертів. Пропонуємо такий алгоритм: нехай кожного проходження циклу перевіряється деяка змінна, скажімо, *STOP*. І коли вона дорівнюватиме 1 виконуємо команду ***brake***, тобто “перервати роботу!”. Значення змінної треба

пов’язати з новою кнопкою. Такий алгоритм реалізовано у наступній редакції підфункцій:

```
% --- Executes on button press in pushStart.
function pushStart_Callback(~, ~, ~)
global Direction  Color  Width  Delay   STOP
 STOP=0;
for Fi=-Direction*(0:pi/100:2*100*pi)
      if STOP==1
            break
      end
         x1=cos(Fi);       y1=sin(Fi);
         x2=-x1;              y2=-y1;
         plot([x1,x2], [y1,y2], Color,
                              'LineWidth',Width);
         axis([-1.2,1.2,  -1.2,1.2]),
         axis off     %убрати оці
         hold on,
         plot(0,0,'ko','MarkerSize',5),
         hold off
        pause(Delay)
end

% --- Executes on button press in pushStop.
function pushStop_Callback(~, ~, ~)
global STOP
STOP=1;
```

Лабораторна робота № 12 надасть вам можливість все, що розказано вище, проробити самостійно, отримати багатий досвід розробки графічного інтерфейсу для будь-яких ваших задач.

8.7. КОНТРОЛЬНІ ПИТАННЯ ДО РОЗДІЛУ 8

1. Дослідіть властивості фігур '*Color*', '*MenuBar*', '*Pointer*', '*Position*', '*Resize*' та '*Visible*'; продумайте, як можете використовувати їх на практиці!
2. Як можно звертатися до осей фігур?

Як можна використовувати їх властивості '*Color*', '*XColor*', '*YColor*', '*ZColor*'?

3. Що можуть встановлювати наступні властивості для вісі O*x*, O*y* графіків, а саме '*FontAngle*', '*FontName*', '*FontSize*', '*FontWeight*'?
4. Чого ми досягаємо командами *set*(***gca***,'*GridLineStyle*', ':'), *set*(***gca***,'*LineWidth*', 5), *get*(***gca***,'*Position*'), *get*(***gca***,'*XLabel*'), *get*(***gca***,'*XTick*')?
5. На екрані – кілька графічних вікон. Якою командою можна узнати, яке з них є "актуальним"?
6. Які відповіді ви очикуєте від команд ***gcf*** і *get*(***gcf***)?

Якщо не змогли відповісти на більшість питань – радимо проробити все з початку

8.8. ЗАДАЧІ ДО РОЗДІЛУ 8

1. Проведіть дослідження, як впливає властивість '*Color*' на фігуру Figure, на осі останньої, на побудовану криву.
2. Яку роль грає властивість '*MenuBar*' для фігури Figure? Аналогічно, властивості '*Position*', '*Resize*' та '*Visible*'?
3. Створіть довільну Figure з графіком та дослідіть властивість '*Pointer*'. Як можна використовувати це на практиці?
4. Дослідіть, на які властивості осей ви можете впливати командами *gca*, *get*(*gca*) та *set*(*gca*).
5. Зробіть GUI-програму, що обертає многокутник обраної кількості сторін, у замовленому напрямку, того чи іншого кольору, швидко чи то повільно.

6. Зробіть GUI-програму, що обертає „трипелюсткову розу" $x = sin(3t)cos(t)$, $y = sin(3t)sin(t)$, $t \in [0,2\pi]$ за годинникової стрілкою чи проти, того чі іншого кольору. (Можна сказати і так: програму розділу 5.7, задача 5, „огорніть" у GUI-оболонку).

7. „Огорніть" у GUI-оболонку програму, про яку йшлося у розділі 5.7, задача 6 (пульсуючий еліпс, або пульсуюче сонечко з „променями").

9. Складність алгоритмів та програм

Прийшов час, шановні програмісти, критично подивитися на ті програми, що ми навчилися робити. “Складність” алгоритмів і програм – це дуже важливо. Та як це розуміти, “***складність***”?

Це можна оцінювати принаймні з двох точок зору: **яку пам’ять потребує** та чи інша програма, чи не занадто велику? **Як швидко працює** програма, чи не занадто повільно? Якщо складність подвоїти, як збільшиться час роботи програми?

Колись потреба алгоритму у великій пам’яті для збереження проміжних даних була ледь не визначальною. Та ті часи пішли у минуле; саме сьогоднішня мініатюризація елементів пам’яті дозволила нам мати фантастичні комп’ютерні ігри, що зовсім не бережуть пам’ять! Та й швидкодія сьогоднішніх процесорів виросла у тисячі разів. Та все ж вона є ще недостатньою, не дозволяє, наприклад, робити надійні прогнози погоди…

Ось чому саме цей вимір ***ефективності*** наших програм, тобто ***як швидко вони працюють***, розглянемо у даному розділі. Раніше ми присвятили даній темі короткий розділ 3.4 в [2], та тепер розглянемо тему детально.

Як швидко працює програма? *Як її порівняти з іншими програмами за даним критерієм*? Це будемо досліджувати.

9.1. Вимірювання часу роботи програми

Припустимо, наша програма *Prog.m*, яку будемо досліджувати, працює з аргументом *Arg*, “складність” якого можна характеризувати цілим числом *n*. Наприклад, якщо *Arg* є квадратна матриця, то *n* – її розмір. Так само, якщо *Arg* – система лінійних рівнянь (СЛАР, розділ 2 в [1]), або визначник матриці. Якщо *Prog.m* має справу з сортуванням числового або текстового вектора, то *n* – довжина вектора. Нарешті, для фракталу *SnowFlake* або функції Добєши *n* – це рівень, який хочемо досягти. Як складність *n* впливає на час виконання?

Покажемо, як скласти таблицю вимірів часу $T=T(n)$, що потрібен для виконання завдань програмою $Prog(Arg(n))$ для аргументів Arg різної складності n. Принципово, для виконання з командного рядка MATLAB це виглядає таким чином:

```
>> n=1; inc=1; tic ; Prog(Arg(n)) ; T(n)=toc ;          (9.1)
>> n= n+inc;   tic ; Prog(Arg(n)) ; T(n)=toc ;
```

Тобто використовуємо MATLAB-команду ***tic*** для запуску виміру часу, та ***toc*** для зупинки "секундоміра" та запису результату. Такі виміри робимо кілька разів (можна не з інкрементом *inc*=1, а з *inc*=10, 100, та навіть, інколи, *inc*=1000). Припустимо, зробили N вимірів . Будуємо тепер графік:

```
>> plot(1:inc:N, T, 'g*-')   .
```

Викладене є лише принциповим вирішенням поставленого нами питання, бо у кожній конкретній задачі можуть виникнути свої нюанси. Найчастіше стикаємося з такою проблемою: сучасні комп'ютери є такими швидкими, що $T(10)$ може відрізнятися від, скажімо, $T(1000)$ лише у якийсь позиції після коми, а це є, як правило, "похибкою вимірів". Що ж робити? Пропонуємо зробити так, як поводиться експериментатор у фізиці: зробити багато вимірів та взяти загальний витрачений час або середній результат.

Та перш, аніж повторювати одне й те саме кілька разів, треба створити цей об'єкт заданої складності $Arg(n)$. З цією метою робимо програму на кшталт

Лістінг *Timing.m*

```
function Timing(N)
%Штучна програма для вимірів ефективності програми Prog
inc=10;
for n=10 : inc : N    % цикл задає різні n до граничного N
    Arg=Generator(n)      % Генерування об'єкту Arg(n)
    tic
    for i=1 : 1000   % багатократне повторення Prog(Arg)
        Prog(Arg(n));
    end
```

```
        T(n)=toc
    end
    plot(10:inc:N, T, 'g*-'),   hold on
    xlabel('Складність задачі n')
    ylabel('Витрачений час T, сек.')
```

Генерувати завдання певної складності *Generator*(*n*) можна різними шляхами, залежно від проблеми. Окремим та доволі складним є питання, як аналізувати отримані дані. Цьому присвячено розділ 9.7. А зараз розглянемо кілька прикладів.

9.2. Аналіз програми *MyMax*

Загальну програму п. 9.1 проілюструємо на прикладі функції `MyMax`, що була розроблена нами у 3.7.1. Вона у числовому векторі довжиною *n* знаходить найбільший елемент, [1], п.3.7.1. Ось конкретизована програма `Timing` з поясненнями:

Приклад 1.

```
function Timing(N,Color)
i0=1000; inc=1000;
Arg0=i0:inc:N;
i=0;     %Номер наступного циклу
for Arg=i0:inc:N
    i=i+1;
    %Генеруємо вектор з 2*Arg0 елементів:
    X=randi([-i0,i0], 1, Arg0(i));% Generator
    tic
    for k=1:1000 %Тисячократне повторення!
        MyMax(X); %спочатку нашу функцію
        %max(X);  %Потім - вбудовану
    end
    T(i)=toc;
end
plot(Arg0,T,Color), hold on
```

Таким чином, об'єкт заданої складності `n` генерували командою `randi([-i0,i0])` як вектор цілих випадкових чисел між `-i0` та `+i0` у кількості `Arg0(i)`, яка змінювалася

від `i0` до `N` з кроком `inc`. А можна створити такий вектор й іншим шляхом, наприклад: `X=[1:i0/2, i0/2:-1:0]`.

Вимірюємо продуктивність програми `MyMax`:

>> `Timing`(1000000,'-ro'), *hold on*

***Увага*:** для повільного комп'ютера слід зменшити аргумент або кількість повторень у циклі!

Тепер закоментуємо `MyMax` у рядочку 11, розкоментуємо внутрішню команду MATLAB `max` та виконуємо з іншим аргументом `Color`:

>> `Timing`(1000000,'-g+')
>>*xlabel*('Кількість елементів масиву'), *ylabel*('Час у сек. ')
>> *legend*('1-Наша програма', '2-Стандартна')

Маємо для порівняння зелену лінію, рис. 9.1. Висновки будуть такі:

1. На комп'ютері авторів програма працює дуже швидко – один мільйон елементів масиву упорядковує тисячу разів і потребує для цього лише 0,012 секунди! Дякуємо, інженери!

2. Та MATLAB-івська програма `max(X)` працює приблизно у 6 разів швидше. Не дивно: фірма MathWorks залучає найкращих математиків і програмістів світу!

3. Не менш важливо, що обидві лінії є прямими, тобто в обох випадках час, що програма потребує, росте пропорційно її складності n, $T = C \cdot n$, де C стала. На вашому комп'ютері конкретні числа на осі Оy будуть іншими, та лінійний характер залежності й відношення швидкості одної програми до іншої лишаються незмінними! Якщо для задачі складності n, програма потребуватиме час T, то для задачі подвійної складності $2n$ потребуватиме час $2T$, для задачі потрійної складності $3n$ потребуватиме час $3T$ і т.д.

9.3. Порівняння програм *MyCramer* і *MyGauss*

Ці програми ми розробили для розв'язання системи лінійних алгебраїчних рівнянь довільного порядку n; перша програма використовує відомий метод Крамера, пов'язаний з обчисленням визначників, друга – інший відомий метод,

метод Гауса, оснований на лінійних еквівалентних перетвореннях матриці (розділ 2.1 в [1]). Наводимо модифіковану для цього програму "*Timing.m*":

Лістінг *SLAE_Timing.m*

```
function SLAE_Timing(N,Color)
p=0;
for i=5:5:N
    p=p+1;
    M=rand(i,i);
    b=(1:i)';
    tic
    for k=1:10
        X=MyCramer(M, b);
        %X=MyGauss(M, b);
        %X=M/b;
    end
    T(p)=toc;
end
plot(5:5:N, T, Color), hold on
```

Тут генератором об'єкта складності *i* є команда створення матриці *i*i* з випадкових чисел *rand(i,i)*. А можна було б згенерувати таку матрицю й іншим шляхом, наприклад

```
i=5; Str1=1:i; A(1, :)=Str1;
for k=2 : i
    A(k-1,:)=Str1.^k;
end
```

Отримуємо матрицю *A* у першому рядку якої йдуть послідовні натуральні числа, у другому – їх квадрати, у третьому – куби і так далі. З вищої алгебри відомо, що визначник такої матриці завжди відмінний від нуля, і тому можна не турбуватися: ця СЛАР має завжди один розв'язок.

Результати вимірювань демонструє рис. 9.2. На ньому можна бачити таке. Метод Гауса 2 набагато швидше за метод Крамера 1. Не дивно, бо в останньому випадку маємо багато разів обчислювати громіздкі визначники. Та у порівнянні з ними багатократно виграє внутрішній метод MATLAB *X=M/b;* (аби "запустити" вимірювання для нього,

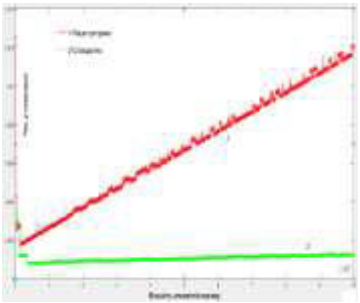

Рис. 9.1. Дослідження `MyMax` та порівняння із стандартною

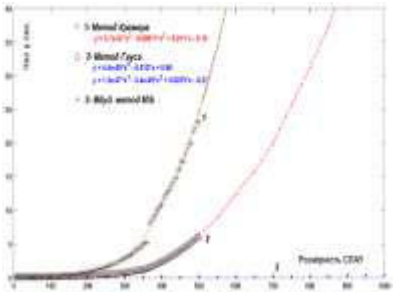

Рис. 9.2. Порівняння швидкості програм: 1 – *MyCramer*, 2 – *MyGauss* та 3 внутрішньої

розкоментуємо відповідний рядок у програмі `SLAE_Timing`). Останній є, скоріше за все, лінійно залежним від n, $T = C \cdot n$. На відміну від цього, дві попередні залежності походять на степеневі залежності виду $T = C \cdot n^2$ або $T = C \cdot n^3$ (див. далі). Останні можна записати у загальному вигляді

$$T \sim n^k, \tag{9.2}$$

де $k = 1$ (лінійна залежність), $k = 2$ (квадратична) або $k = 3$ (кубічна).

Зверніть увагу, що результуючі криві нібито складаються з двох. Отримати розрахунки до $n = 400$ або $n = 500$ (ті частини кривих, що позначені значками) потребує вже значного часу. Аби його скоротити, та просунутись далі, від СЛАР розміру n=500 до n=1000, доцільно змінити початок розрахунку та інкремент, а саме `i0`=50 та `inc`=50. Саме таким чином отримано штрихові криві, що продовжують попередні. Вони лише підтверджують виявлену закономірність.

Є ще одна особливість у кривих, що отримано: вони не дуже "гладкі". Деякі точки вириваються у верх, деякі випадають трохи вниз. Це тому, що у наших вимірах, так само як у фізиці або у техніці, виникають деякі ***похибки***. Якби ми не зробили б багатократне повторення у програмі

Timing – криві були б ще "гірші", менш регулярні. Багатократне повторення дозволяє осереднити результати однократних вимірювань. У розділах 9.6 і 9.7 обговоримо, як з ними поводитися.

Примітка. Та ще є така цікавинка: часто-густо студенти знаходять, що "ми неправі", що насправді метод Крамера швидше за Гауса! І це дійсно відбувається у діапазоні $0 < n < 500$ або $0 < n < 800$. Пояснення до такого ефекту див. далі.

9.4. Виміри ефективності інших програм

Аналогічним чином дослідимо інші програми, що розробили раніше.

9.4.1. Порівняння обчислення чисел Фібоначчі ітераційним алгоритмом "*MyFibbonachi*1.*m*" та рекурсивним алгоритмом "*MyFibbonachi*2.*m*". Модифікація програми "*Timing.m*", що тут потрібна, достатньо проста, тому наводимо "*TimerFibon.m*" без пояснень. Результати її роботи подано у вигляді графіків на рис. 9.3*а*.

Лістінг файлу *TimerFibon.m*

```
function [Tr,Ti,Arg]=TimerFibon(i0, inc, N, Color, Color2)
    %рекурсивний та ітеративний методів для чисел Фібоначчі:
    % i0 – початкова складність задачі, N – кінцева;
    % inc – крок її зміни.
    % Color, Color2 – колір та позначка для рекурсії та ітерації.
    % Приклад використання:
    %>>[Tr,Ti,Arg]=TimerFibon(1,1,10,'g^','b+')
    %% Копірайт К.Хаврай, ІАН-103а; 30.05.15
    Arg0=i0 : inc : N;
    i=0;
    for Arg=i0:inc:N;
        i=i+1;
        tic         %Метод рекурсії
           MyFibbonachi1(Arg);
```

```
        T(i)=toc; %готово для даного i
        M=5;   %параметр "осереднення"
        tic   %Метод Ітерації
        for j=1: M
            MyFibbonachi2(Arg);
        end
        R(i)=toc/M;  %готово для даного i
end
figure(1)
plot(Arg0,T,Color),hold on
plot(Arg0,R,Color2)
legend('Рекурсія', 'Ітерація')
title('Порівняння методів для чисел Фібоначчі')
```

Бачимо з рис. 9.3*а*, що час роботи за методом рекурсії зростає набагато швидше, аніж за методом ітерацій. Не дивно: на зберігання проміжних результатів, які тут потрібні витрачається доволі багато часу. Складається враження, що метод ітерацій зовсім не витрачає часу! Що це не так, свідчить рис. 9.3*б*, де початкова область розрахунків збільшена у тисячу разів ("*на три порядки*" кажуть науковці).

Та й такий результат для ітераційного методу не є достатньо переконливим! Продовжимо дослідження, та й запустимо програму *TimerFibon* лише для останнього,

>> [*Tr*,*Ti*,*Arg*]=*TimerFibon*(1, 1, 1000, '*go*', '*b**');

закоментувавши попередньо все, що відноситься до методу рекурсії. (Це необхідно, аби не чекати результатів до наступного сторіччя -:). Результат, який показано на рис. 9.3*в* підтверджує, що розрахунки не є миттєвими. Час, потрібний для метода ітерацій, дійсно на кілька порядків менший за метод рекурсії. Цей час зростає за лінійним законом. "Експериментальні" точки трохи "витанцьовують" відносно проведеної прямої, та все ж вона відображає генеральну закономірність. Там навіть її рівняння виписано. Як отримано пряму та її рівняння – див. нижче.

У наведеній вище програмі зовнішні аргументи *Tr*, *Ti*, *Arg* дозволяють нам не лише знов побудувати такі само графіки незалежно від цієї програми, а й робити над її

результатами будь-які перетворення. Візьмемо логарифми ***ln*** від векторів часу *Tr*, *Ti* та залишимо незмінним аргумент *Arg*,

```
>> figure(2), plot(Arg0(5:end), log(T(5:end)),Color), hold on
>> plot(Arg0(5:end), log(R(5:end)), Color2)
>> legend('Рекурсія', 'Ітерація')
```

(п'ять перших точок не враховано). Що ж бачимо на рис. 9.3*г*?

Логарифми дозволяють нам зіставляти не самі величини (бо вони незрівнянно відрізняються!), а ***порядки*** цих величин. Бачимо, що різниця часу, потрібного для двох методів для n=30 складає до 2-(-14)=16 порядків! Час, що його витрачає програма рекурсії, поки n змінюється від 0 до 30, складає 2-(-12)=12 порядків – так нестримно ця величина зростає!

9.4.2. Порівняння методів обчислення визначника методом еквівалентних перетворень "*MyDeterminant.m*", розділ 3.6, та методом рекурсії "*MyDet2.m*", розділ 7.4, принципово нового також нічого не містить. Для генерації квадратної матриці розмірності `n` використано команду `M=rand(n,n)`. Як і у попередньому прикладі, обидві програми виявляються принципово різними за своєю швидкодією, з якої причини їх порівнювати важко. Тому відразу використаємо логарифмічні координати. Замість величин `n` на горизонтальній осі і величин `T(n)` на вертикальній на рис. 9.4*а* використані величини `n` та `ln(T)` відповідно (напівлогарифмічні координати). Відзначимо, що логарифми витраченого часу `ln(T)` стають такими, що їх можна співставляти, та різниця між ними 15 – 20 порядків! Зверніть увагу, що час, витрачений алгоритмом рекурсії, стає лінійним у цих координатах, тобто

$$\ln T = an + b;$$

залежність для рекурсивного методу лишається кривою. (Конкретно, на нашому комп'ютері y=1.9*`n` – 16, тобто $a = 1.9$, b ролі не грає) На правому рис. 9.4б використані логарифмічні координати, *ln*(*n*) та *ln*(*T*) відповідно. Дивно,

що тут, навпаки, експериментальні точки методу ітерацій "лягають на пряму". Це означає, що час залежить від складності наступним чином,

$$\ln T = a \ln n + b, \qquad (9.3)$$

де $a = 6.3 \cdot 10^{-6}$, $b = 0.0014$. Що це означає – будемо аналізувати далі.

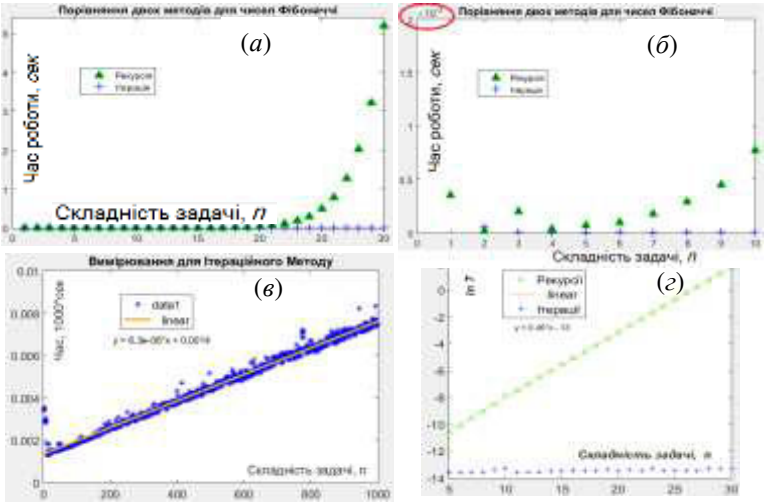

Рис. 9.3. Порівняння ефективності двох методів обчислення чисел Фібоначчі: (*а*) результат розрахунку "*TimerFibon.m*"; (*б*) збільшення у 1000 разів; (*в*) більш детальне дослідження методу ітерацій; (*г*) глибший аналіз методу рекурсій.

9.5. Аналітична оцінка ефективності програм

Ми робили експерименти з нашими програмами, подібно до експериментів у фізиці. А як із теорією? Не завжди це можливо, та є випадки, коли “навести теорію” не складно.

Розглянемо програму *SnowFlake* (див. розділ 7.2). Працюючи із нею, важко не помітити, як швидко зростає час побудови відповідного фракталу із збільшенням “замовленого” його рівня *n*. Розглянемо це аналітично.

Очевидно, що час переходу від фракталу одного рівня n до фракталу наступного рівня n+1 є пропорційним кількості його сторін L на даному рівні n. Під час такого переходу програма *SnowFlake* має замість кожної сторони AB створити 4 нові $AA1$, $A1D$, $DA2$ та $A2B$. Тож і маємо:

n=1, кількість сторін $L_1 = 3$;

n=2, кількість сторін $L_2 = L_1 \cdot 4 = 3 \cdot 4$;

n=3, кількість сторін $L_3 = L_2 \cdot 4 = 3 \cdot 4 \cdot 4 = 3 \cdot 4^2$;

n=4, кількість сторін $L_4 = L_3 \cdot 4 = 3 \cdot 4^2 \cdot 4 = 3 \cdot 4^3$;

Аналізуючи залежність, що виникає, знаходимо загальну формулу $L_n = L_{n-1} \cdot 4 = 3 \cdot 4^{n-1}$. Природно, що й час роботи програми має підкорятися формулі

$$T(n) = C \cdot 4^n, \qquad (9.4)$$

де значення коефіцієнту C залежить від властивостей конкретного комп'ютера.

Залежність (9.4) "набагато гірше", аніж степенева залежність (9.2). Справді, порівняємо:

t	t^2	t^3	t^4	2^t	***Час***
А	***Б***	***В***	***Г***	***Д***	***Е***
3	9	27	81	8	8 сек
8	64	512	4096	256	4,3 мин
14	196	2744	38416	1634	27 мин
15	225	3375	50625	32768	9,1 ч
16	256	4096	65536	65536	18,2 ч
17	289	4913	83521	131072	1,6 діб
18	324	5832	104978	262144	3 діб

Легко бачити, що $t < t^2 < t^3 < t^4$ (стовпчики *А*-:-*Г*, степеневі функції), ці функції зростають тим швидше, чим більше показник степеня. Тепер порівняємо ці стовпчики із стовпчиком *Д* (показникова функція). Бачимо, що функція $T=2^t$ (всі коефіцієнти прийнято C=1) спочатку веде себе "скромно", нижче за функцію t^k, та після певного значення її обганяє. Так, у порівнянні з t^4 показникова функція починає її

обганяти і зростати швидше лише при t=16. Ні, не просто "швидше" – нестримно зростати, набагато випереджаючи сусідні функції! Аби це було ще виразніше, додамо ще колонку E, де останні числа "перераховано" у час виконання завдання. І якщо перші задачі виконуються за секунди, то останні потребують кількох діб!

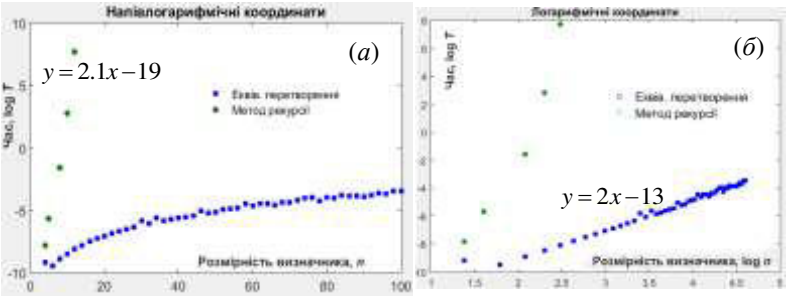

Рис. 9.4. Дослідження швидкодії програм обчислення визначників: (*а*) у напівлогарифмічних координатах метод еквівалентних перетворень дає лінійну залежність; (*б*) у логарифмічних координатах метод рекурсій дає лінійну залежність.

Ось чому програмісти вважають важливим досліджувати такі якості своїх програм та використовують спеціальні математичні позначення для характеристики їх повільності:

$$\text{ʊ}(n^1),\quad \text{ʊ}(n^2),\quad \text{ʊ}(n^3),\quad \ldots..,\quad \text{ʊ}(a^n). \tag{9.5}$$

Літера ʊ , грецьке *омікрон*, означає "*росте як*", "*пропорційно*" а саме: "*Час росте як перший степінь складності задачі*",

"*як другий степінь складності задачі*",

"*пропорційно третьому степеню складності*",

. . . . . .

"*Час зростає як* 2^n, як 3^n, тощо", саме значення a принципової ролі вже не грає.

Підсумовуючи щодо програми *SnowFlake* скажемо тепер: із складністю завдання час виконання зростає як $\text{ʊ}(4^n)$. Так прогнозує теорія. А що скаже експеримент з програмою?

9.5. Аналітична оцінка ефективності програм

Ми робили експерименти з програмами двох типів. Тепер експериментуємо з побудовою фракталів Коха *SnowFlake*, розділ 7.2. Попередньо закоментуємо "штучні затримки" командами *pause*. Програма для експериментальних вимірів `SnowFlakeTiming` виглядає наступним чином:

Лістінг програми `SnowFlakeTiming`

```
function [Ns,Time]=SnowFlakeTiming(N,Color)
%для вимірювань часу роботи програми SnowFlake;
%приклад, як запускати SnowFlakeTiming:
% >> [Ns,Time]= SnowFlakeTiming(7,'pg');
%Copyright Ye.Gayev,   May 2015
Ns=1:N;
for i=1:N
    tic
    SnowFlake(i)
    Time(i)=toc %вивід даних на кожному циклі
end
figure(1)
plot(Ns,Time,Color), hold on
title('Вихідні змінні для SnowFlakeTiming')
xlabel('Складність, n'); ylabel('Час T, сек')
figure(2),plot(Ns,log(Time),Color), hold on
title('Напівлогарифмічні змінні')
xlabel('Складність, log n');
ylabel('Час T, сек')
figure(3),plot(log(Ns),log(Time),Color), hold on
title('Логарифмічні змінні')
xlabel('Складність, log n');
ylabel('Час log T, сек')
```

Для кожного n програма запускає `SnowFlake` лише один раз. Особливістю процесу є те, що програма працює занадто довго вже при $N=7$ і ми можемо захотіти штучно перервати обчислення. Тому не ставимо крапку з комою після команди `Time(i)=toc` – тоді результати неповних обчислень можна прочитати у командному вікні. Крім того, на графічну побудову фракталів на кожному циклі потрібен суттєвий

додатковий час; можна виключити ці витрати, якщо закоментувати відповідні команди у `SnowFlake.m`.

Результати випробувань на комп'ютері авторів є у таблиці

Ns	1	2	3	4	5	6	7	8	9
Time1	0.0020	0.0023	0.0063	0.029	0.115	1.269	27.9	792	21362
Time2	0.0003	0.0004	0.0013	0.007	0.040	0.472	25.2	768	20117
Time3	0.0006	0.0009	0.0018	0.0055	0.0395	0.4816	25.4	775	
Time4	0.0002	0.0003	0.0008	0.0040	0.0369	0.4812	25.5	771	20256

та показані на рис. 9.5 (лише 4 виміри, бо кожний вимагає багато часу). Бачимо, як круто підіймається крива. Тут лише здається, що 6 перших точок лежать на осі *X*-ів… Та такої картинки ще недостатньо, аби зробити не лише якісні висновки, та й ще кількісні. Чи підкоряються здобуті "експериментальні виміри" закону (9.3)? Чи відповідає теорія практиці? Узнаємо в 9.8.

9.6. Рівняння замість таблиці

Виникла нова задача: щойно отримані дані (рис. 9.5) *мають* підкорятися закону (9.3). Та чи підкорюються? А якщо так, то чому дорівнює стала *C*? Пам'ятаємо, що остання характеризує не явище, як таке, а швидкість конкретного комп'ютера.

Подібна задача відома у науці під назвою "*знаходження емпіричного рівняння*" і вже обговорювалася в [1], розділ 2.3. Вона часто виникає в експериментальній фізиці та у техніці. Та виникає у двох постановках.

1. Теоретичний закон, якому *мають підкорятися* дані, що виміряні, невідомий. Тож потрібна *хоч якась математична формула*, що дає "близькі" результати.

Навіщо ж нам така формула? Відповідей кілька:

(*i*) Тому, що нею користуватися зручніше, аніж таблицею даних. Окрім того, її збереження потребує менше пам'яті, аніж збереження даних таблиці.

(*ii*) Тому, що, знаючи дані "у вузлах", легко знаходимо значення у проміжках між ними (задача *інтерполяції*).

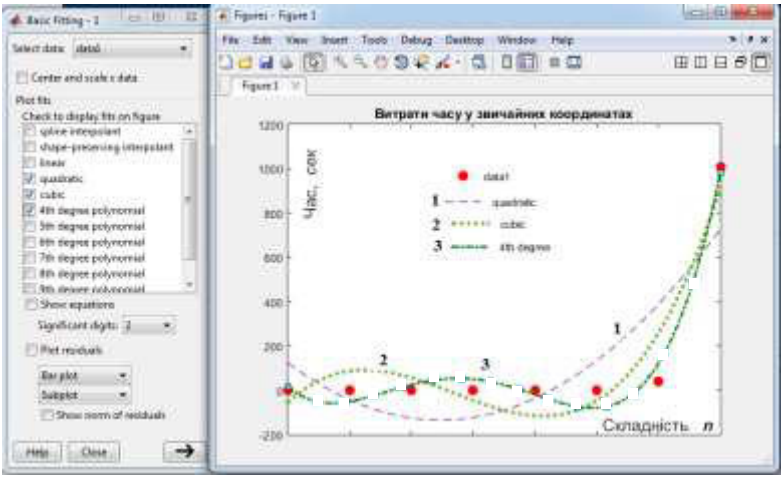

Рис. 9.5. Залежність часу роботи програми `SnowFlake` від складності завдання n (значки), разом з вікном ***Basic Fitting*** та отриманими кривими 1 – 3.

(*iii*) Хоча експериментальна таблиця обмежена найменшим та найбільшим значенням аргументу, формула дозволяє виходити за ці межі і обчислювати наперед, за найбільший аргумент (задача *прогнозу*); або, навпаки, для менших значень (*реставрація минулого*). Як же ж її, або їх, формули, знаходити? Обговоримо кілька випадків.

1.1. Нам відомі лише 2 (два) значення якоїсь функції y_1 та y_2 для, відповідно, двох значень аргументу x_1 і x_2. Яку формулу можна знайти у такому випадку? Лише лінійну, що, як відомо, має рівняння виду $y = ax + b$. Конкретні значення коефіцієнтів a та b треба знайти.

Оскільки її графік (пряма лінія) має проходити через зазначені точки, отримуємо два рівняння для двох невідомих a та b (увага: x_i та y_i – відомі координати точок):

$$\begin{cases} y_1 = ax_1 + b, \\ y_2 = ax_2 + b \end{cases}. \tag{9.6}$$

Більш детальний аналіз дає, що ця система завжди має розв'язок та лише один. Тож й лінійна функція, що ми шукаємо, буде лише одна.

(Звідси зрозуміло: якщо хтось уперто захоче шукати квадратичну або кубічну функцію – в нього будуть лише 2 рівняння (9.6) для більшої кількості коефіцієнтів. Однозначного розв'язку не буде! MATLAB повідомить: "*Polynomial is not unique: degree >= number of data points*").

1.2. Зустрічається й випадок трьох відомих значень y_1, y_2 та y_3 для, відповідно, трьох значень аргументу x_1, x_2 і x_3. Якщо шукаємо функцію у вигляді поліному, що ***точно проходить через дані точки***, то нею може бути крива другого порядку $y = ax^2 + bx + c$. Для трьох її коефіцієнтів a, b і c маємо аналогічно (9.6) три рівняння

$$\begin{cases} y_1 = ax_1^2 + bx_1 + c, \\ y_2 = ax_2^2 + bx_2 + c, . \\ y_3 = ax_3^2 + bx_3 + c. \end{cases}$$

Як і в попередньому випадку, такий розв'язок існує та, обов'язково, лише один.

Так само, якщо нам ***точно відомі*** чотири точки, через які має проходити крива, то її слід брати у вигляді многочлену не старше третього порядку, і так далі.

9.7. Інструмент *Fitting Toolbox*

Та виникають сумніви: навіщо вимагати **точного** проходження кривої через точки, адже у них є певна похибка? Такі сумніви виявляються революційними! Вони й призвели до винаходу методу найменших квадратів (МНК). Про останній ми розповіли у розділі 2.3 першої частини підручника, [1]. Цей метод в MATLAB реалізовано у вигляді інструменту *Fitting Toolbox*.

Повернемося до рис. 9.5, де великими кружечками показані "експериментальні" точки за останньою таблицею. Натиснемо кнопку *Tools* у верхній частині графічного вікна MATLAB. У новому вікні, що з'явиться, обираємо опцію

Basic Fitting (передостання унизу). З'являється вікно, що показано ліворуч на рис. 9.5. Треба пересвідчитися, що у рядку *Select data* обрано саме ті дані, що аналізуються. Тепер у вікні *Check to display fits on figure* відмічайте чек-бокси такі як *linear*, *quadratic*, *cubic*, *4th degree polynomial* тощо. На правому рисунку виникають різнокольорові криві, що *найкращим чином* наближають ваші точки. Треба мати на увазі: "*найкращим чином*" – лише з кривих замовленого типу, лінійних, квадратичних і т.д.

Що ж саме стоїть за цим означенням, "*найкращим чином*" – дивіться у розділі 2.3. Дивно ведуть себе ці криві по відношенню до вихідних точок: чим більший степінь многочлена обираємо, тим більш "старанно" прогинається крива до них! Та якщо відмічати чек-бокс *Show equation* маємо ще рівняння, знайдені *Fitting Toolbox*. На підставі нашого досвіду з многочленами Тейлора (розділ 1.7.7 та лабораторна робота №1 [1]) можна очікувати, що чим степінь більший – тим "краще". Та це зовсім не так, тут цей досвід не спрацьовує! Бо криві хвилясті, або такі, що йдуть до -∞, неможливі за змістом. Краще за все слід брати многочлени найнижчого степеню.

У разі ітеративного алгоритму для чисел Фібоначчі, рис. 9.3*в*, найкращим рівнянням виявилося лінійне; знайдено аналітичний вираз $T = 6.3 \cdot 10^{-6}\, n + 0.0014$. Іншими словами: витрати часу підкоряються закону $\Theta(n^1)$. Можна бути впевненим, що точки вище та нижче за пряму лінію містять певні похибки проти генеральної лінійної залежності.

А от для рекурсивного обчислення чисел Фібоначчі (рис. 9.3*г*) справа складніше: лінійному закону підкоряються не величини T та n, а $\ln T = 0.49\, n - 13$. Звідси маємо:

$$T = e^{0.49n-13} = e^{-13}\left(e^{0.49}\right)^n = 2.26 \cdot 10^{-6}\left(1.63\right)^n,$$

тобто витрати часу підкоряються показниковому закону $\Theta(a^n)$, де a=1.63.

Як же ж діяти, аби отримати правильний закон? Обирати слід з двох законів: степеневого (9.2), лінійного

зокрема; та показникового у формі (9.3). Пропонуємо наступний загальний алгоритм аналізу:

1. провести експериментальне випробування, "timing" алгоритму, що досліджується, накопичити якнаймога більшу таблицю даних n та відповідних T;
2. побудувати 3 рисунки з графіками $T = f(n)$, $T_1 = \ln T = f(n)$ та $T_2 = \ln T = f(n_1)$ де $n_1 = \ln n$;
3. обрати той з них, де наша крива стає найбільш близькою до прямої лінії. Тут інколи треба виключати кілька перших точок масиву, бо нас цікавить поведінка алгоритму за умови $n \to \infty$.

Продемонструємо такий на прикладі програм, що обчислюють визначник матриці, рис. 9.4. З рисунку 9.4*а*, де використані напівлогарифмічні координати, бачимо, що у рекурсивного методу. можна знайти рівняння

NB: Феномен Рунге https://ru.wikipedia.org/wiki/Феномен_Рунге https://en.wikipedia.org/wiki/Runge%27s_phenomenon

А от на рис. 9.4*б* ці точки знов утворюють якусь криву, а от точки, що відповідають методу еквівалентних перетворень розташувалися близько до прямої, її рівняння можимо отримати.

9.8. Експериментальне випробування *SnowFlake*

Повернемось до програми *SnowFlake*. Дані вимірів з таблиці в напівлогарифмічних та логарифмічних координатах представлено на рис. 9.6. Жодний з рисунків не дає прямої лінії.

9.8.1. Якщо формула $T=C*4^n$ є вірною, то лишається знайти значення коефіцієнта C:

```
>> C=T./4.^n
```

Вводимо дані з таблиці розділу 9.5:

```
>> T=[0.0002  0.0003..0.0008  0.0040  0.0369  0.4812  25.53  771.8] ;
>> n=1:8;
```

і отримуємо:

```
C =0.00005  0.000019  0.0000125  0.000015623  0.00003603515
```

0.000117480468750 0.00155823 0.01177673

Теоретичний закон відомий; треба знайти числові значення коефіцієнтів, що входять у запис його рівняння.

Перевірити практично та поглибити викладений матеріал вам допоможе лабораторна робота №13 наприкінці даного підручника.

* * *
*

На закінчення розділу хочеться розповісти, як самі студенти сприймали цей складний матеріал.

Даша М. пише у Висновках по даній лабораторній роботі: "*Я витратила масу часу у цій не дуже важливій роботі на те, аби з'ясувати, чому мої графіки відмінні від графіків інших студентів.*" Критичне відношення!

А ось Катя Х.: "*Я вважаю, що ця лабораторна робота була чудовим закінченням нашого курсу. Ми змогли визначити ефективність наших власних програм. Більш того, ми визначили, які з них найкращі*".

До таких висновків дівчата дійшли на підставі усних лекцій. Сподіваємось, що цей навчальний курс, тепер у друкованому вигляді, допоможе ним ще більше!

9.9. КОНТРОЛЬНІ ПИТАННЯ ДО РОЗДІЛУ 9

1. Як визначити поняття “складність програми”? Чи може бути інше визначення? Чи це важливо у комп’ютерній науці?
2. Якими командами вимірюють швидкодію програми?

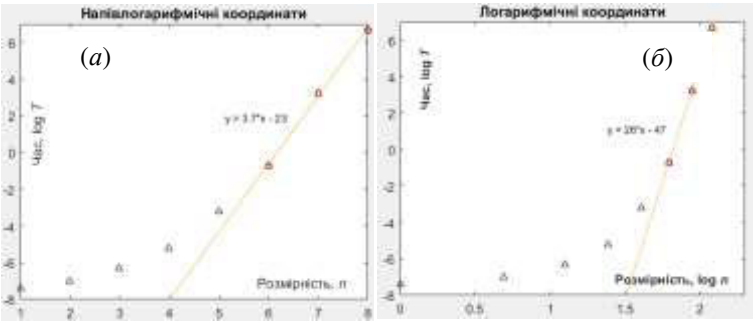

Рис. 9.6. "Експериментальні" виміри швидкодії програми *SnowFlake* у напівлогарифмічних (*а*) та логарифмічних координатах (*б*).

3. Чи може буде точним результат вимірювання часу, якщо він набагато менше одиниці, припустимо *Time*=5*10*e*-7? Чому? Що можна робити, аби покращити точність вимірів?
4. Як робити генерацію завдань тієї чи іншої складності *n*?
5. Ми встановили, припустимо, що час, що витрачається програмою на виконання завдань, залежить від складності *n* останніх наступним чином $T = an^2 + bn - c$, де a, b, c певні числа. Чому ми кажемо, що $T = an^2$, тобто не враховуємо b і c?
6. А якщо казати $T \sim n^2$, тобто "T *пропорційно* n^2", чи буде це менш інформативним висловом?
7. Якщо одна програма визначається залежністю $T \sim n^2$, а інша – $T \sim n$, то яка з них працює повільніше, яка швидше? Як це можна визначити за допомогою операції $\lim$?
8. Якщо одна програма визначається залежністю $T \sim n^k$, а інша – $T \sim a^n$. Яка з них працює повільніше, яка швидше? Як це можна визначити за допомогою операції $\lim$?
9. Нехай одна програма визначається залежністю $T \sim n$, а інша – $T \sim \ln n$. Яка з них працює повільніше, яка швидше?

10. Що можна сказати про програми ітераційні і рекурсивні з точки зору їх швидкодії?
11. Спробуйте встановити аналітично асимптотичну поведінку програм *SnowFlake* і *MyDaubechies* від складності завдань n.
12. Як можна визначити поведінку програм від складності завдань n, якщо аналітична оцінка занадто складна або взагалі неможлива?
13. Як працювати з MATLAB-інструментом *FittingToolbox*?
14. Які емпіричні залежності пропонує інструмент *Fitting Toolbox*?
15. Чи завжди від дає відповідь? Що означає відповідь на кшталт "*Polynomial is not unique . . .*"?
16. У чому сутність *метода найменших квадратів* (МНК), що покладено в основу *FittingToolbox*?
17. Як би ви розрізнили дві можливі задачі для експериментальної функції y_1, y_2, y_3, заданої для значень аргументу x_1, x_2, x_3: (*i*) "*Знайти квадратичну функцію* $y = ax^2 + bx + c$, *що* ***точно проходить*** *через задані точки*" та (*ii*) "*Знайти функцію* $y = ax^2 + bx + c$, *що* ***проходить якнайближче*** *до заданих точок*"?

Якщо не змогли відповісти на більшість питань – радимо проробити весь розділ з початку

Сподіваємось, Ви розв'язали більшість задач. Це означає, ви добре знаєте Комп'ютерну Науку та її практичну складову – програмування. Ви здатні тепер розробляти доволі складні програмні комплекси.

Модуль 5

10. Самостійне створення софта (курсові роботи)

Ви мріяли стати програмістом? Час прийшов! Цей модуль призначений для вашої ***самостійної роботи***. На кафедрі СУЛА Національного авіаційного університету у цьому модулі надається студентам приблизно місяць, аби (одночасно з рештою лабораторних робіт**-:)** вони зробили якийсь свій власний, за власним вибором програмний комплекс. Тих знань, що ви отримали, вже достатньо для цього! Це й буде Курсовою роботою.

Для початку ми запропонуємо вам щось, з чого ви й будете виходити.

10.1. Пропозиції для самостійних робіт

Зробити треба сучасну програму, тобто – з обов'язковим використанням GUI. А от що саме "загорнуто" у GUI – подумаємо разом. Головна мета – зробити чудову програму на ваш особистий смак, де використати якнайбільше вашої фантазії та знань з програмування. В Інтернеті ви бачили багато чудових програм. А самі щось таке зробити можете? Нехай не таке чудове (всеж-таки одинак, без колективу та без платні **-:**). Та самостійно! Ось які є початкові пропозиції.

10.1.1. Одна з найпростіших – програма *Watch*, годинник, рис. 10.1. З математики потрібні знання лише початку аналітичної геометрії: командою *T*=*clock* отримуємо інформацію про час у комп'ютері; четвертий, п'ятий та шостий елементи вектора *T* дають, відповідно, актуальний час, хвилину та секунду. Тепер слід знайти точку на колі, яка має відповідати положенню годинної стрілки, хвилинної та секундної, та провести кольорову лінію між нею та початком координат (0,0). Аналітичну геометрію не забули? Все це робити у циклі – маємо ілюзію руху стрілочок (анімацію).

Нібито просто, та попрацювати прийдеться. Застосування фантазії вітається, наприклад: додати меню вибору міст з іншої часової зони (Москва, Париж, Нью Йорк тощо), показувати час в обраному місті разом з проектуванням рисунку (типового фото того міста чи країни), виконаннням мелодії тієї країни (див. 5.4).

10.1.2. Графічна програма, де можемо обрати той чи інший фрактал (SnowFlake 7.2, фрактали Серпінського, рис. 7.15, або інші [19]) та будувати, як він виглядає на різних рівнях n, які в нього властивості для $n \to \infty$. Вигляд можливої програми показано на рис. 10.2.

10.1.3. Можливо зробити різні програми тестування знань студентів з математики, використовуючи незрівнянні можливості MATLAB у графіці та в аналітичному розв'язанні математичних задач. Одна така, присвячена темі "Аналітична геометрія площини у просторі", показана на рис. 10.3. Програма складається з кількох GUI, одна викликає іншу (ось чому у багатьох випадках ми говоримо про "програмний комплекс"). Програма спочатку пропонує обрати одну з кількох тем тестування. Далі викликається GUI, присвячене даній темі, і "задає питання" з неї. Припустимо, генерує (випадково) три трійки чисел $A_{i=}\{x_i,y_i,z_i\}$, i=1,2,3 і пропонує написати рівняння площини, що проходить через них. Оскільки MATLAB "вміє розв'язувати задачі", він і знаходить рівняння з його коефіцієнтами і подає його біля однієї з радіо-кнопок (праворуч унизу). Формуються також три помилкові відповіді (додаванням-відніманням від правильних коефіцієнтів), і подаються біля інших радіо-кнопок. Відповів студент правильно – отримує бал 1, помилково – бал 0. Бали сумуються з усіх задач, наприкінці тесту обчислюємо оцінку і повідомляємо студента.

На рис. 10.4 показано простішу програму (та те ж оціненну на найвищий бал). Праворуч подано 7 кнопок, що відповідають типовим поверхням другого порядку. Натискуємо на ту, що цікавить – отримуємо зображення поверхні. Та тут є секрет: не кожну поверхню легко побудувати у декартових координатах, відомих більшості;

Рис.10.1. Програма ***Watch***

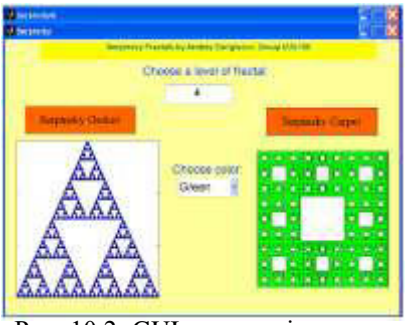

Рис. 10.2. GUI для дослідження фракталів, студент О.Шило, 2013 р.

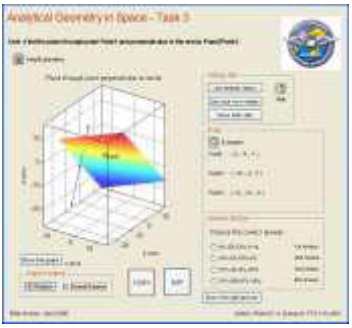

Рис. 10.3. Програма тестування знань з аналітичної геометрії, студент Р. Дамачі, 2006 р.

Рис. 10.4. Програма тестування знань з теми "Поверхні другого порядку", студентка В. Крицька, 2013 р.

авторці прийшлося познайомитися з параметричними представленнями цих поверхонь. Таким чином, студентська "програма тестування" випробовує не лише сторонню людину, а й розробника стимулює краще засвоїти матеріал підручника з математики, чи то оволодіти додатковим -☺.

Аналогічну програму тестування знань з теми "Многочлени" створив студент Б.Явдонич, 2014 р. Можете й ви обрати таку тему власної курсової!

10.1.4. Попередній підрозділ стосувався "точної" науки, де добре працює MATLAB. Аналогічні програми

тестування можна запропонувати й для “не точних” наук, таких як філософія, англійський тощо. Вони вже не можуть спиратися на обчислювальні можливості MATLAB; тут треба посилатися на якісь файли з питаннями та варіантами відповідей (правильних та помилкових). Зрозуміло, що слід використати можливості роботи з базами даних, розділи 6.4 – 6.6, лабораторна робота № 11.

Рис. 10.5 демонструє, як таку проблему вирішив студент О.Бойко, 2015. Перше GUI запрошує до тесту, знайомить з його правилами. Далі йдуть GUI з блоками питань (студент взяв з математики, як приклад) та з чотирма кнопками з відповідями; лише одна є вірною. Відповіді розміщено у випадковому порядку; використано команду ***randi***(4). Поки студент думає, грає музика. Вірно чи помилково обрав відповідь – з’являється та чи інша маска, чути той чи інших звук. Блок питань закінчено – з’являється нове GUI з новим блоком питань. Ця програма Олексія Бойка відтворює популярний телевізійний конкурс.

Програми, що маніпулюють з базами даних – також чудові об’єкти для розробки, хоча й доволі громіздкі. Це може бути програма *MyGroup*, що веде облік студентів вашої групи та оцінок з різних дисциплін, що вони отримують. Тут слід передбачити кнопки “*Отримати середній бал*” для кожної дисципліни, що реалізувати в MATLAB легко. Студенти Н.Овчарчін і Н.Пурдік додатково пропонували зробити перевірку права доступу, перш ніш відкрити файл з результатами – це вже доволі складно…

Студентка А.Скородєд чудово реалізовала створення бази даних та подальшу роботу з нею у програмі *MyLibrary*, рис. 10.6. Програма веде облік книг з персональної бібліотеки. У ній було вперше винайдені недокументовані можливості MATLAB у створенні GUI – спосіб проектування будь-якої картинки на GUI та на GUI-елементи (використано також О.Бойко, рис. 10.6 й іншими студентами-розробниками). Передбачена кнопка *Допомога* (ліворуч уверху, рис. 10.6); при натисканні з’являється вікно (показано ліворуч посередині) з пояснюючим текстом до програми; зрозуміло, вона реалізована засобами, обговореними в 8.3.2 –

Рис. 10.5. GUI з питаннями та маска, що з'являється у разі правильної відповіді (студент О.Бойко, 2015).

Рис. 10.6. Програма ведення бази даних *МояБібліотека*, студентка Н.Скородєд.

8.3.4. Перелік, що випадає праворуч, дає список наявних у базі книг. Кнопка правіше видає зображення титульної сторінки обраної книжки. Складним є програмування кнопок "*Додати книжку*" та "*Видалити книжку*", бо тут треба коректно працювати із складним типом даних *cell* або *struct*. Після здіснення таких операцій нову базу даних слід зберегти. Роботу програми може супроводжувати музика, див. нижче.

10.1.5. Хто з сучасних студентів не грав у комп'ютерні ігри? Багато хто хоче створювати власні ігри. Чому ні? Адже це може бути першим кроком у "дорослу гру" – створення, скажімо, якогось імітатора реальності, тренажера.

Рис. 10.7 і 10.8 представляють найпростіші такі ігри, "Хрестики-нуліки" та "Морський бій"; їх правила широко відомі. Зробити GUI з квадратними таблицями на них нескладно. Складніше створити "тип даних", що зберігає стан гри у пам'яті комп'ютера; він може базуватися на простій матриці.

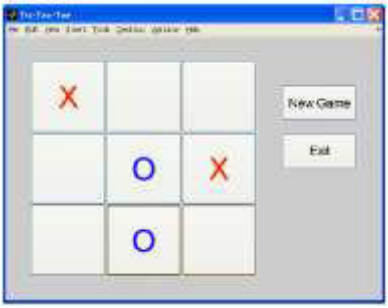

Рис. 10.7. Гра “*Хрестики-нолики*”, К.Авраменко, 2014 р.

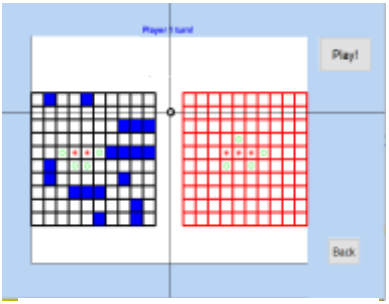

Рис. 10.8. “*Морський бій*”; І.Цуд, 2014

10.1.6. Теорія ймовірностей – класичний розділ математики, який часто вивчають окремо. Найпростіша її задача – кидати кубік з очками від одного до шести. Очевидно, що ймовірність випадіння, скажімо, одиниці є $\frac{1}{6}$. Однак зробіть фізичний експеримент з таким кубіком: кидайте E=10 разів та підрахуйте випадіння одиниці (результат називають “*частотою успіху*” f). Чи дорівнює $f=\frac{1}{6}$? Тепер кинемо E=100 разів. Досягли “співпадіння теорії і експерименту” $f=\frac{1}{6}$? А якщо кинути . . . E=10 000 разів? А якщо граней тіла не 6, а P?

Класичною є й така задача: P-тіло кидають N разів (і це є “один експеримент”). Питання: яка ймовірність того, що одиниця не випала жодного разу $p(0)$? Випала один раз $p(1)$? Два рази $p(2)$ і т.д. до $p(N)$? Складна комбінаторна задача! Зрозуміло лише, що $p(N+1)=0$. Один з можливих результатів розподілу ймовірностей показано штриховою лінією на рис. 10.9. Це теорія. А якщо реально кидати кубік, чи співпаде вона з експериментом? Якою має бути кількість експериментів E, аби досягти задовільного співпадіння? В теорії ймовірностей на це питання відповідає Теорема великих чисел. А в нас двоє студентів зробили MATLAB-програму, яка й теорію рахує й експеримент (за допомогою генератора випадкових чисел) проводить.

Зовнішній вигляд програми показано на рис. 10.9. Перше GUI пропонує спочатку обрати одну з трьох

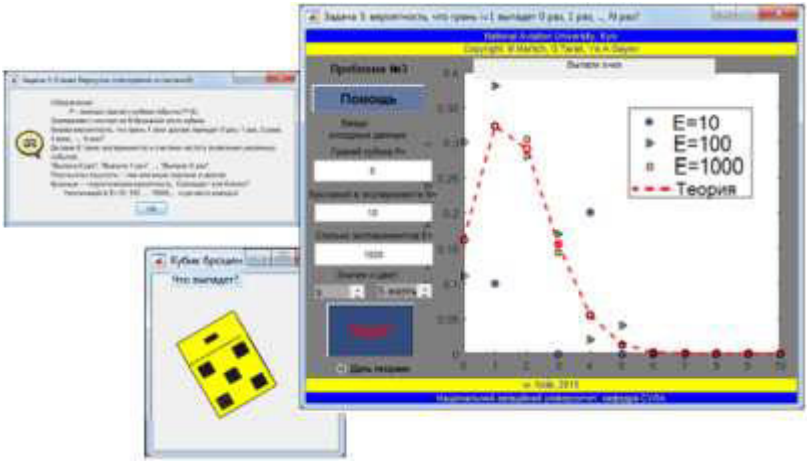

Рис. 10.9. Програма з терії ймовірностей М.Мартича і Г.Тарака [32].

ймовірнострих задач, пропонує *Help* для кожної (ліворуч уверху на рис. 10.9). Обрали задачу – виникає GUI для неї, де слід задати значення відповідних параметрів, позначку та колір результату та натиснути кнопку “Start!”. Та перш ніж видати результати (адже потрібні лише секунди для 100 000 “експериментів”-:) програма запускає програму з галузі механіки твердого тіла, анімує обертання кубіка, ліворуч унизу на рис. 10.9. Як само це зроблено – студенти опублікували у науковій статті [32].

10.1.7. У підрозділі 10.1.3 та рис. 10.4 згадували про параметричне представлення геометричних образів. Студентка першого курсу Д.Малініна дослідила параметричні рівняння так званої багатопелюсткової рози[31]

$$x=(r\sin(nt)+a)\cos(t+\varphi)\,,\quad y=(r\sin(nt)+a)\sin(t+\varphi)\,,$$

[31] http://mathworld.wolfram.com/Rose.html

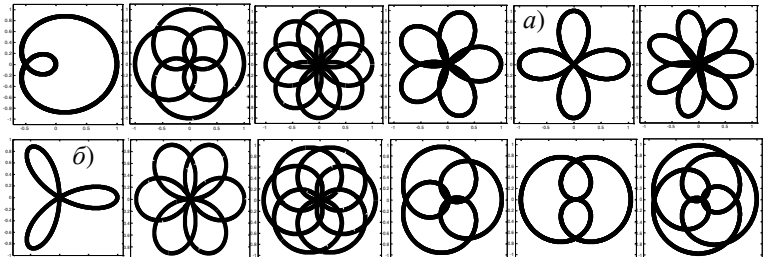

Рис. 10.10. Послідовна трансформація фігур при анімації багатопелюсткової рози: (*а*) p =*var*, $q=3$; (*б*) q =*var*, $p=3$. Офсет $a=0$.

де параметр $t\in[0,k\pi]$, а сталі коефіцієнти n, a, k и φ можуть приймати різні значення (і, як правило, $r=1$). У MATLAB легко пересвідчитись, що такі рівняння дають доволі різноманітні криві (рис. 5.1 та 10.10). Виникла ідея змінювати у часі один з двох параметрів, p або q, і отримати таким чином певний "рух" кривих на екрані, анімацію. "Одягнута у GUI", така програма показана на рис. 10.11. Вона подана для науково-педагогічної публікації [33].

Ми навели багато вдало розроблених студентами тем курсових робіт. Чекаємо на вашу фантазію! Та практика доводить, що створити вдалу розробку ще половина справи.

10.2. Як писати і захищати курсову роботу

В вас є всі необхідні знання та навички для створення власної програми, не гіршу за інших. Та ми, як педагоги, часто помічали, що людина може щось зробити, а от роз'яснити іншим – зась. Фахівець, не здатний пояснити, довести, захистити свою думку і розробку не може вважатися кваліфікованим. Цей останній модуль вчить вас, і вимагає від вас, окрім комп'ютерної програми

(*і*) зробити Звіт про розробку, власне "Курсову роботу", аналог майбутньої дипломної роботи, науково-технічної публікації, аналог супроводної документації на розробку;

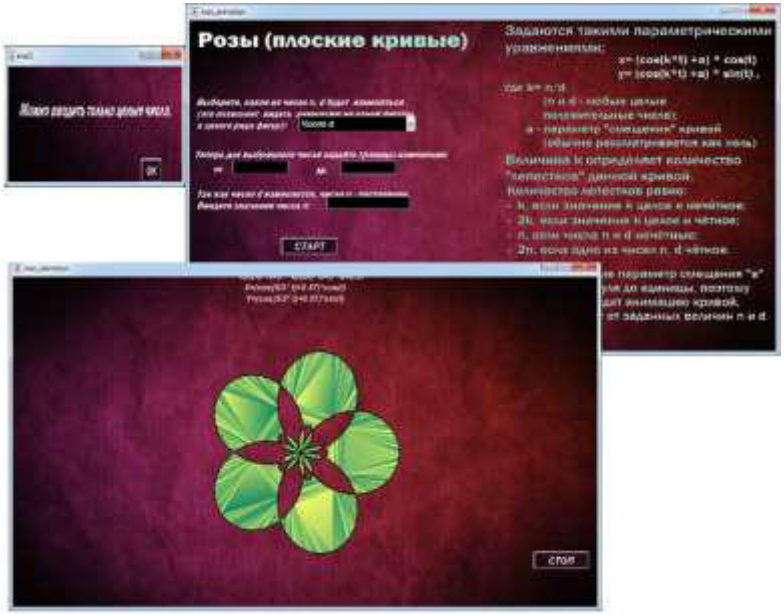

Рис. 10.11. Програма анімації “Світ параметричних кривих” Д.Малініної, 2015 [33].

(*ii*) захистити розробку і Звіт прилюдно, витримати критику таких само фахівців (студентів -:). Для цього обов’язково зробити усну доповідь; доцільно використати PowerPoint-презентацію. Тут потрібно вміти тримати себе перед аудіторією, володіти нею не підглядаючи у текст доповіді, відповідати на критичні запитання аудиторії. Для публічного захисту має діяти жартівливий студентський лозунг “*Здамо наші погані знання на Добре та Відмінно*”!

(*iii*) треба витримати критику (іноді – гостру) фахіфця, який спеціально ознайомився з вашою роботою, ***опонент*** називається. До речі, а самі бути *опонентом* вмієте? Терпляче розібратися у чужій роботі, доброзичливо вказати на її помилки та недоліки, порадити інший підхід чи метод? На захисті вам прийдеться не лише доповідати свою розробку, а й ще бути опонентом чужої. Робота опонентом також оцінюється викладачем!

10.2.1. Сучасні вимоги до курсової роботи. Робота, де роз'яснюється ваша розробка, доводиться, що вона "найкраща у світі" та заслуговує високої оцінки, має бути не емоційною, а максимально стриманою і об'єктивною. Науково-технічна творча спільнота виробила певні стандарти викладення такого роду матеріалу. Ось вони:

(i) Робота повинна мати якнайкоротшу, проте – повну, назву, що відображає її сутність.

(ii) Робота повинна починатися з її Анотації – коротким викладенням її змісту. Анотація створюється "ітераційним шляхом", поступовими наближеннями. На початку роботи пишете, як ви собі уявляєте майбутню курсову роботу; коли ж роботу закінчено повністю, Анотація редагується остаточно. Доцільно зробити й сторінку "Зміст роботи" з її головними розділами.

(*iii*) У короткому розділі "Вступ" обгрунтовують актуальність теми роботи (розробки); тут можна викласти мотивацію, чому ви обрали саме таку тему (Найпростіший приклад: "Ще у дитинстві я побачив аналогічну програму і вирішив її відтворити"-:).

(*iv*) Дуже важливим є розділ[32] "1. Стан питання". Тут ви маєте показати, чи є у світі (у Світі Інтернету зокрема) щось аналогічне до вашої роботи, до вашої розробки.

Буває зрідка і так, що ваша розробка аналогів не має, унікальна. Так було, колись, у молодого Томаса Едісона, в Ніколи Тесли, в Сергія Корольова, багатьох інших, збагативших людство своїми винаходами. Але така вдача буває не часто. Більшість ідей людство вже винайшло, і ваша ідея, скоріше за все, буде мати аналоги. У будь-якому випадку треба бути чесним. Якщо ідею роботи, або метод її реалізації в когось позичили – все ж намагайтеся зробити її самостійно. Лише після цього погляньте у "відповідь", чужу роботу, і чесно дайте на неї посилання.

Посилання на аналоги, що знайшли в Інтернеті, на книжки або статті в журналах треба робити приблизно такими

[32] Не даремно в американській літературі його красиво іменують ***State-of-the-Art***!

словами: “*аналогічну програму описано в* [№1, №2]”; В квадратних дужечках вказуються номери з переліку “Література”, див. далі пункт (*x*).

(*v*) Починаємо викладення роботи по суті (розділ, скажімо, 2). Назвіть його “2. Постановка задачі” або “2. Формулювання проблеми”. Пригадайте, як ви уявляли собі свою програму на початку розробки, що планували – це допоможе зрозуміти того, хто читає вашу роботу вперше. Зрозуміло, що коли мета досягнута, то пройдений шлях насправді зовсім не такий, як це уявлялося спочатку. Тому, остаточному опису результату слід, як правило, присвятити окремий розділ.

(*vi*) Розділ “3. Розв’язання проблеми” (номер 3 також є умовним). Після короткої загальної характеристики вашого розв’язку (програми) за пунктами 3.1, 3.2 і т.д. викладіть ті частини проблеми, етапи, що прийшлося пройти за час розробки, задля досягнення поставленої цілі.

Пригадайте: якщо була така мить, коли здавалося “*не знаю як діяти*”, то це чудова нагода відокремити належний сюди матеріал в окремий такий *підрозділ* 3.x! Для кожного з таких пунктів треба зформулювати, у чому полягає складність цієї частини програми, надати посилання на роботу або публікацію, що наштовхнула вас на потрібну ідею. Наприклад: “Робимо як у публікаціях [3 - 7]”.

(*vii*) Це ви розклали всю вашу роботу по частинах. Тепер слід після її *аналіз*у зробити *сінтез*[33], тобто Розділ “4. Опис остаточної програми” або “4. Інструкція користувачеві”. Наведіть приклади використання вашої програми, призначення всіх клавіш GUI. Зверніть увагу на особливі випадки, які можуть виникнути у користуванні.

До речі, тестування програмного забезпечення за останні десять років перетворилося у надзвичайно важливу і окрему галузь комп’ютерної науки [30,31]!

Якщо не весь матеріал вклався у названі пункти 1 – 3, можна зробити ще розділи 4 і 5; дайте їм назви.

---

[33] Смисл цих грецьких слів *аналіз* і *синтез* радимо подивіться у Вікіпедії https://uk.wikipedia.org/wiki/

(*viii*) Завершувати всяку науково-технічну роботу слід розділом “Висновки” (номер за традицією не ставлять). Тут коротко підсумувати, що саме зроблено вище, чому ви навчилися, де можна застосувати вашу програму та результати, які перспективи розвитку, поліпшення тощо.

(*ix*) Не обов’язково, але можливо, зробити короткий розділ “Подяки”. Наші студенти вміщують тут слова подяки колегам-студентам, на чиї роботи або поради вони спиралися. Можна згадати тут і інших людей, що сприяли вашій роботі.

(*x*) Нарешті, в останньому розділі “Література” або “Посилання” надають ті печатні або Інтернет-джерела, на які ви спиралися у виконанні вашої роботи, які згадували у розділах (*iii*) – (*vii*). Стандарти, за якими оформлювати посилання – шукайте в Інтернеті або в [34]. Обов’язково вказати авторів твору (одного-двох, якщо їх кілька, та додати “*та ін.*”), назву книжки чи статті, журнал, місто видання, видавництво, рік видання, кількість сторінок. Трохи складніше посилатися на Інтернет-ресурси: тут треба не лише надати посилання, а й поінформувати, про що там йдеться, та указати дату вашого останнього відвідування того сайту. Приблизно так у списку лытератури:

7. Про пакет MATLAB https://ru.wikipedia.org/wiki/MATLAB (дата звернення:17.12.2016).

Тут ми хочемо знов нагадати, що обов’язкове використання посилань – важлива вимога наукової етики, якій ще недостатньо приділяється уваги у нашій країні. Вона є складовою поняття ***Copyright***. Ми попереджаємо наших студентів, що списування, подання робіт “скачаних” з Інтернету суворо карається.

“Використання” та “подання чужого як власного” – дві великі різниці! І це стає очевидним саме за цим останнім розділом! “Використання” передбачає, що ви коректно послалися на чужі результати. Аби не сталося “*випадково забув послатися*”, радимо вам з самого початку роботи зробити файл ***Література.doc***, куди занотовувати ті

джерела, з якими працювали, частину з яких слід використати в остаточному Звіті (курсовій роботі).

10.2.2. Приклад роботи[34]: “Досліджуємо та створюємо звук та музику”.

Анотація: *Пропонуються програми аналізу звуку у MATLAB та створення звуку та навіть музики у власному комп’ютері. Хоча аналогічні програми, навіть набагато кращі, вже існують, та ми викладаємо, як це зробити самостійно в обмежений час. Вважаємо доцільним включати такі “математичні аспекти звуку” та його програмування до університетського курсу комп’ютерних наук та програмування.*

Ключові слова: програмування, MATLAB, цифровий звук, цмфрова музика, викладання та вивчення програмування

Вступ. Комп’ютерні науки стали ледь не найголовнішими дисциплінами сьогоднішньої системи вищої освіти. Сьогодні вони включають у себе не тільки власно програмування і різні мови програмування, а також багато самостійних, окремих дисциплін – алгоритми, структури даних, операційні системи та комп’ютерні мережі, комп’ютерна графіка, штучний інтелект та інші [1-5]. Інколи ці науки викладають доволі нудно. Та ми знаємо, що це не так!

Загальна теза про “всемогутність” комп’ютерної науки має бути підтверджена власним досвідом студентів. Проблема “цифровий звук” здивує студента комп’ютерними можливостями та стимулюватиме його до дослідницької діяльності. Використання звуку в кожній з відомих нині мов програмування – Pascal, C, C++, Java і тд. – завдання не з простих і потребує знання мови в значному обсязі. Проте, на початку викладання програмування все більшу популярність здобуває “матрична лабораторія” MATLAB, яка є чудовим

[34] Скорочений виклад спільної із студентами першого курсу О.Рожком та Н.Овчарчиним наукової статті [25]. Зміни зроблено для того, аби трохи “повернутися” до стилю курсових робіт, з яких все починалося. Нумерація рисунків має місце у межах даної “курсової роботи”. Посилання надані наприкінці розділу.

інструментом не лише оволодіння математикою [6], а й програмуванням в цілому [7-12]. У даній роботі викладено звукові можливості MATLAB як в плані викладання, так і в плані вивчення програмування в процесі розробки акустичних і музичних MATLAB-програм.

1. Постановка і сучасний стан питання. Запис, зберігання і відтворення звуку в комп'ютерних науках стало можливим лише після створення спеціальних пристроїв для комп'ютера (звукової карти, динаміка, т.п.), кодування сигналів в спеціальних форматах та збереження у файлах, розробки програмних засобів і протоколів їх взаємодії зі звуковими файлами. Відповідні проблеми, що почали розроблятися з 1970-х гг., багато в чому вже вирішені [13–19], хоча поліпшення і подальша розробка тривають. Темою даної статті є робота з готовими пристроями та інструментами, використання звуків в курсі програмування та алгоритмізації, їх відтворення і вивчення, а також програмування звуків і музики. Принципові питання даної проблематики обговорюються в роботах [20-22] та ін. Даний неповний список джерел та пошук в Інтернеті показують, що певна література з цього питання доступна, але вона дуже обмежена. Крім того, зазначені джерела присвячені складним та серйозним аспектам обробки звуку та орієнтовані, як правило, на професіоналів. З них одна лише книга [14], також розрахована на акустиків-професіоналів, має справу з середовищем MATLAB. Визначення ж "мінімального набору" відомостей для демонстрації цифрового звуку, так само як і його використання у викладанні основ програмування, ставиться тут вперше.

Вершиною використання звуку у практиці людства є музика. Та за останні роки у цій галузі культури виникла нова сфера, так звана комп’ютерна музика, музикально-комп’ютерні технології [15]. Багато студентів, що вивчають комп’ютерні науки та програмування, цікавляться нею, володіють тією чи іншою з існуючих складних програм для створення цифрової музики [16]. Та проблема полягає не у тому, аби створити конкурентно-спроможну аналогічну

програму [16], а у тому, щоб виділити основні і прості елементи з цієї галузі комп'ютерних наук та надати студентам можливість їх вивчати шляхом створення доволі простих власних програм, експериментувати з такими можливостями.

2. Найпростіший звук в MATLAB.

2.1 Випробування роботи із звуком доцільно розпочати виконанням наступної MATLAB-команди:

>> load handel, sound(*y*)

(позначка *>>* тут і надалі означає виконання с командного рядочка MATLAB; для використання у програмах, *m*-файлах, її треба опустити). Студенти чують фрагмент ораторії Генделя. Замість двійкового файлу *handel*, що зберігає певний звук, можна використати файли "*train*", "*laughter*", деякі інші. Передбачається, що з численних додаткових MATLAB-інструментів повинен бути встановлений AudioVideo Toolbox із стандартним розміщенням в ProgramFiles\MATLAB\...\ AudioVideo\). Учні охоче збільшують привабливість всіх своїх програм вставкою в кінець коду або його етапів тих чи інших звуків. В якості наступного кроку, можна потішитися "стисненням" і "розтягуванням" цих або записаних власноруч (див. далі) звуків,

>> load handel, sound(*y*)
>> FS = 4000; sound (y, FS)
>> FS=12000; sound (y, FS)

(нормальне звучання при частоті відтворення *FS*=8192 *Гц*), або їх "складанням"

*>> Y*1 = [*y*; *y*; *y*]; *sound*(*y*1) (1)

(спираємося на свободу матричних операцій MATLAB; в даному випадку – подовження стовпчику даних). Одночасно можна продемонструвати "портрет" того чи іншого звуку

>> plot(*y*)

(подібний до зображеного на рис. 1, ліворуч та рис. 2), обговорити фізичне походження і математичний зміст "сигналу *y*".

2.2. Виникає питання про запис звуку. Опишемо команди, які реалізують це в MATLAB у парадигмі об'єктно-орієнтованого програмування.

Запис через мікрофон протягом часу *T*, *сек*, може бути здійснена наступним фрагментом коду:

```
>>T=5;
>>% підготовка аудіо-об'єкту r:
>> r = audiorecorder;
>>% включаємо мікрофон:
>> disp(['Кажіть щось ', num2str(T), ' секунд: ']);
>> recordblocking(r, T ); (2)
>> disp('Закінчення запису.');
>> stop(r);
```

В цьому фрагменті коду відбувається наступне. Створюється "аудіо-об'єкт" ***r*** з частотою дискретизації, за умовчанням, *Fs*=40000 *Гц* (варіанти для більш економного запису 8000, 11025, 22050 *Гц*), 1 канал з "глибиною запам'ятовування" 8 байт. Аудіо-об'єкт ***r*** спочатку "порожній", з деякими полями, про які можна дізнатися командою *get*(***r***). Запит допомоги

```
>> help audiorecorder
```

дозволяє дізнатися і про методи об'єкта ***r***. До них крім вже використаних методів *recordblocking* і *stop* входять і методи *record*, *play*, *getaudiodata* та інші. Ось їх використання у можливому фрагменті коду:

```
>>% Відтворення запису r:
>> P = play(r);
>>% Перетворення запису r
>> % у числовий вектор:
>> mySpeech=getaudiodata(r);
>> %Побудова його графіку:
>> Plot(mySpeech)
```

2.3. Хоча це відноситься не до програмування, а до математики, але тут якраз вдалий привід надати студентам

фундаментально-важливе поняття про дискретне перетворення Фур'є як спосіб показати "частотний портрет" записаного звуку без складних математичних подробиць. Для цього до запису *mySpeech* слід застосувати MATLAB-команду *fft* (Fast Fourier Transform).

Та спочатку продемонструємо в інтерпретації [10], як саме MATLAB визначає „спектральний склад" сигналів. Створимо тестовий сигнал у часі t, $y = a_1 \sin\omega_1 t + a_2 \sin\omega_2 t$ (беремо задля простоти лише дві частоти ω_1 і ω_2, що його складають); „виміряємо" його у дискретні проміжки часу $t = 0 : dt : N \cdot dt$, де N – кількість вимірів. Наприклад:

```
>>%Спектр з частот 20 і 40 Гц з амплітудами a1, a2 кожна:
>>Omeg1=20; a1=2; Omeg2=40; a2=-3;                    (3)
>>%Обравши крок, маємо N моментів вимірів:
>> dt=0.0001; N=20000;  t=0:dt:N*dt;
>>%Вектор „вимірів":
>>y=a1*sin(Omeg1*t)+a2*sin(Omeg2*t);
>>P=2*pi*max([1/Omeg1,1/Omeg2]); %період коливання
```

Маємо „портрет сигналу":

```
>>plot(t,y)
>> xlabel('Час, \it{сек}')
>> ylabel('Звуковий сигнал')
>>title(['Сигнал; Період=',num2str(P)])
```

Його періодом є $T = \max(\frac{2\pi}{\omega_1}, \frac{2\pi}{\omega_2})$.

Задача полягає у тому, аби знайти складові частоти тестового сигналу та порівняти з тими, що його утворили. Тепер обернена задача: маємо якийсь сигнал y та рівномірний масив значень часу t, коли його отримано. Тепер покажемо, як за цією вхідною інформацією отримати спектральний склад сигналу:

```
>>% інкремент часу,кількість вимірів:
>> DT=t(2)-t(1); n=length(t);
>>% отримуємо дійсну частину FFT:
>> Y=fft(y); ReY=real(Y);
>> % отримуємо простір частот:
>> Omega=2*pi*(1:n)/(n*DT);
```

```
>> % спектральний „портрет”:
>>figure, plot(Omega(1:100), ReY(1:100))
>>  grid on
>> xlabel('Частота, \it{Гц}')
>> ylabel('Дійсна частина FFT')
>>title('Частотний портрет сигналу')
```

По „плесках” можна зробити наближений висновок про частоти, з яких сигнал утворено. У даному числовому випадку знайдемо $\omega \approx 20$ і $\omega \approx 40$ *Гц*; див. рис. 1, праворуч – тобто ті самі, що закладені у створений сигнал (3). На причинах можливих похибок не зупиняємось.

2.4. Практика показує, що положення 2.1 – 2.3 студенти освоюють легко і швидко. Але даний матеріал вже дає можливості для власних творчих фантазій студентів в програмуванні. Як приклад – рис. 2, графічний інтерфейс програми студентки 1-го курсу НАУ А.Грігорук (створення інтерфейсу командою *guide* див. нижче).

Натискання кнопки "*Record*" дозволяє записати через мікрофон будь-яку фразу в заданий у програмі час, фрагменти коду (2). Кнопка "*Listen*" дає можливість прослухати цей запис через вбудовані динаміки комп'ютера (фрагмент коду (1)). Кнопка "*Plot initial Soundwave*" дозволяє побачити "портрет" записаної фрази у вигляді графіка: по горизонтальній осі час в секундах, по вертикалі величина звукового сигналу.

Оскільки записаний звук – числовий вектор (умовно названий ***r***), то його елементи нескладно переписати в зворотному порядку, від останнього до першого. В MATLAB це виконується за допомогою оператора

```
>> r1 = r(end : -1: 1);
```

де ключове слово *end* закріплено за довжиною масиву і обчислюється автоматично. Тепер кнопка "*Reverse*" дозволяє прослухати запис "з кінця наперед", а кнопка "*Plot Soundwave backwards*" – побачити "портрет" перевернутого запису.

Аналогічно можна досліджувати "огрубіння" записаного звуку командою

```
>> r2 = r(1: 2: end);
```

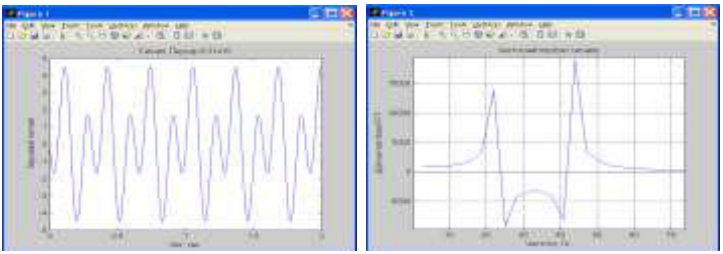

Рис. 1. „Звичайний” портрет сигналу (ліворуч) та його частотний склад за *fft* (праворуч)

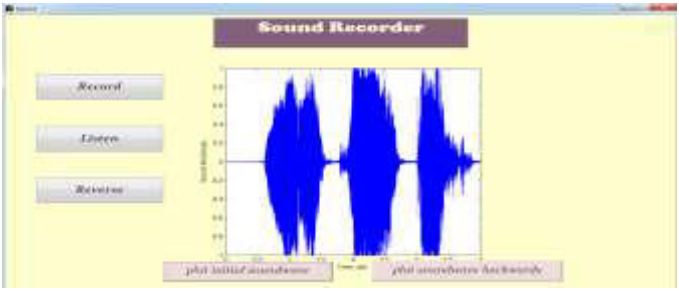

Рис. 2. Графічний інтерфейс для програми запису, відтворення та “перевертання” звуку (А.Григорук)

(елементи запису через один пропускаються).

Таким чином, наведені прості команди MATLAB дають учням можливість самостійно використовувати звук у власних програмах, поміркувати про фізичну і математичну природу звуку та проекспериментувати з цим. Складніші програми обговорюються далі.

3. Програмування звуку в MATLAB

Легко створити гармонійний звук (числовий вектор) *y* за допомогою функції *sin*():

```
>> F=2093; T=3;
>>yC=sin(0:T*550*pi);                    (4)
>> sound(yC, F)
```

При даних параметрах – частоті відтворення *F*=2093 *Гц*, отримаємо звук "До" першої октави ("C" в німецькій нотації) [23]; його тривалість дорівнює близько *T*=3 секунди. Звук

"Ре" ("D" в німецькій нотації) отримаємо, якщо, відповідно до теорії музики [23], помножимо частоту відтворення на $2^{\frac{1}{6}}$; інтервал дискретизації тут дорівнює 1. В командному вікні це виглядає так:

```
>>F=F*2^(1/6),yD=sin(0:300*pi*F/F0);
>>sound(yD, F)                                    (5)
```

(частота *F*=2349 *Гц*). Повторивши останню команду, аналогічним чином відтворимо звук "Мі" ("Е"), частота *F*=2637 *Гц*. Продовжуючи множити частоту відтворення на $2^{\frac{1}{6}}$, можна "виконати" найпростішу гаму. Тривалість кожної ноти буде близько 1*c*, що забезпечується подовженням числового вектора щоразу на *F/F0*.

3.2. Графічний інтерфейс для виконання гами. Викликаємо у командному вікні середовище створення графічного інтерфейсу

```
>> guide,
```

у вікні GUI (Graphical User Interface) розташовуємо сім елементів PushButton (за кількістю нот від До3 до До4), тобто створюємо дизайн програми у вигляді клавіш рояля, рис. 3, і в відповідному *m*-файлі за кожної "клавішею" прописуємо коди за зразком (4), (5). Послідовне натискання клавіш зліва направо і потім назад "виконує" гаму До-мінор. Звичайно, можна виконати і інший простий музичний фрагмент в межах однієї октави при однаковій тривалості звучання нот.

3.3. Тембр звуку. Вище, у розділі 3.1, ми утворювали „чистий", гармонійний звук однієї частоти, що називається фундаментальною для даної ноти. Та музикальні твори цінують за „багатство" звуку, що виражається перш за все у понятті „тембр" звуку (timbre). Під цим розуміють додавання до фундаментальної частоти звуків кратних частот із зменшеною амплітудою, [24,25]. Тобто, замість синусоїди з фундаментальною частотою ω_0 виконують звук

$$y = a_0 \sin \omega_0 + a_2 \sin 2\omega_0 + \ldots + a_k \sin k\omega_0,$$

де $a_k < a_{k-1} < \ldots < a_2 << a_0$. Кількість частот k та амплітуд $a_2,\ldots,a_k$ – справа суб'єктивного смаку. Нами підібрано тембр, що визначається доданком

$$dy = \frac{a_0}{2}\sin 3\omega_0 + \frac{a_0}{2^2}\sin 5\omega_0 + \\ + \frac{a_0}{2^3}\sin 7\omega_0 + \frac{a_0}{2^4}\sin 9\omega_0,$$

див. далі.

3.4. Простий синтезатор. Програма синтезу числових векторів *t*,

t=*mySynth*(*freq*, *duration*, *timbred*), (6)

що відповідають тій чи іншій ноті згідно до п. 3.1, зберігається у файлі *mySynth.m* і складається з кількох підпрограм. До підпрограми *freqParse*(*note*), що визначає частоту *freq*, відповідну тій чи іншій ноті *note*, звертатимемося з текстовим аргументом *note*, який буде складатися з двох або трьох символів, що позначають ноту, її модифікацію (дієз або бемоль) і номер октави. наприклад:

'C # 4' - До дієз четвертої октави,
'E1' - Мі першої октави,
'Ab3' - Ля бемоль малої октави.

Названа підпрограма в основних рисах виглядає таким чином:

```
function freq = freqParse(note)
% Обраховує значення частоти, Гц.
  if (ischar(note) && length(note) > 1)
     octave = note (end);
     note_ = note(1:end-1);
% Частота еталонної Ля 1-ї октави:
     base = 440;
        switch(note_)
          case 'Cb' % нота До
             freq = base/(2^(10/12));
          case 'C'
```

```
            freq = base/(2^(9/12));
        case 'C#' % До дієз
            freq = base/(2^(8/12));
        case 'Db' % нота Ре бемоль
            freq = base/(2^(8/12));
        case 'D' % нота Ре
            freq =base/(2^ (7/12));
. . . . . .
        case 'B' % Сі бемоль
            freq =base*2^ (1/12);
        case 'H' % нота Ре
            freq = base*2^(2/12);
        case 'H #'
            freq =base*2;
        otherwise
            error ('Немає такої ноти!')
    end
% зміна частоти враховуючи октаву:
    switch(octave)
        case '0'
            freq = freq/16;
        case '1'
            freq = freq/8;
        . . . . . .
        case '7'
            freq = freq*8;
    end
end
```

Пояснимо її. 1). Перший оператор функції *freqParse* – умовний ***if…end***; він аналізує текстовий аргумент *note* і відокремлює від нього текстову складову аргументу *note_* (один або 2 перші символи) і цифрову складову *octave* (останній символ).

2) Далі перший блок ***switch ... end*** по базовій частоті *base* = 440 *Гц* обчислює частоту звучання тієї чи іншої ноти *freq*. Наприклад, нота 'До дієз' ('C#' в міжнародній нотації) звучить з частотою *freq=base*/(2^(8/12))=77.18 *Гц*.

Якщо текст, поданий на вхід програми *mySynth* не відповідає ні одній ноті, отримаємо діагностику "*Немає такої ноти*!", оператор *error* у разі *otherwise*.

3) Другий блок ***switch...end*** призначений для уточнення значення цієї частоти відповідно до *octave*, тобто до цифри в кінці текстового аргументу *note*. Як бачимо, для цього, знайдена в першому блоці величина частоти *freq* множиться або ділиться на деяку ступінь двійки.

Іншими аргументами програми *mySynth.m* є *duratio*n і *timbred*, що визначають, відповідно, тривалість синтезованих сигналів у секундах та їх тембр, тобто доданок до основної синусоїди. Останній аргумент приймає значення 0 (гармонічний сигнал не змінюється) або 1 (сигнал модифікується). Він використовується у підпрограмі *oscillator.m*. Ця підпрограма утворює числовий вектор, що відповідає „замовленій" ноті, її тривалості та тембру.

В результаті програма (6) видає числовий вектор *t*, що відповідає частоті *freq* замовленої ноти *note*, її тривалості *duration* та присутності або відсутності модифікації її тембру *timbred*. Далі цей вектор може бути „виконаний" командою (5).

3.4. Виконання музичного твору. На основі описаного синтезатора стандартних музичних нот можна програмувати цифрову музику за її нотним записом. Наводимо скрипт *havaNagila.m*, що програмує в MATLAB фрагмент популярної мелодії, ноти якого показані на рис. 4.

```
timbre = 1; T=44100;
mus = [ ]; %пустий спочатку вектор
%поступове накопичення вектору mus:
mus = [mus  mySynth('C4', 0.5, timbre)];
mus=[mus  mySynth('C4', 0.75, timbre)];
mus=[mus  mySynth('E4', 0.25, timbre)];
mus=[mus  mySynth('C#4',0.25, timbre)];
mus=[mus  mySynth('C4', 0.25, timbre)];
mus =[mus  mySynth('E4', 0.5, timbre)];
mus =[mus  mySynth('E4', 0.75, timbre)];
mus =[mus mySynth('G4', 0.25, timbre)];
mus =[mus mySynth('F4', 0.25, timbre)];
```

```
mus =[mus mySynth('E4', 0.25, timbre)];
mus =[mus mySynth('F4', 0.5, timbre)];
mus =[mus mySynth('F4', 0.75, timbre)];
mus =[mus mySynth('Ab4', 0.25, timbre)];
mus =[mus mySynth('G4', 0.25, timbre)];
mus =[mus mySynth('F4', 0.25, timbre)];
mus =[mus mySynth('G4', 0.5, timbre)];
mus =[mus mySynth('E4', 0.25, timbre)];
mus =[mus mySynth('C#4', 0.25, timbre)];
mus =[mus mySynth('C4', 0.75, timbre)];
% „Виконання” вектора mus
sound(mus, T);
```

Як пояснено у коментарях, програма поступово утворює вектор *mus*, що є спочатку пустим, та поступово накопичує числові елементі. Наприкінці, „виконання” музичного вектора *mus* здійснюється командою *sound* з частотою відтворення *T* (такою ж вона має бути у програмі синтезу *mySynth.m*). Кількість елементів в *mus* є 155 000 при *T*=44100, і 38 750 при *T*=5000 *Гц*.

Висновки

Таким чином, середовище рішень математичних задач і програмування MATLAB дозволяє просто, без відволікання на спеціальні питання, використовувати звук і музику як для їх самостійного відтворення та вивчення їх фізико-математичних аспектів, так і як доповнення до задач програмування. Саме останньому дана робота призначена. Використання звуку та музики може бути хорошим, захоплюючим для студентів розвитком стандартного курсу програмування.

У даній роботі ми надали як відомі MATLAB-команди, що використовують звук і можуть покращити якість і привабливість вже існуючого в лектора матеріалу з програмування, так і нові прості розробки, що можуть суттєво розширити арсенал лекцій з даної дисципліни.

Подальшим розвитком теми можуть бути й інші проблеми, робота над якими поведе учнів до нового кола актуальних практично значущих завдань. У їх числі можна назвати, наприклад, розпізнавання звуку, набір тексту за

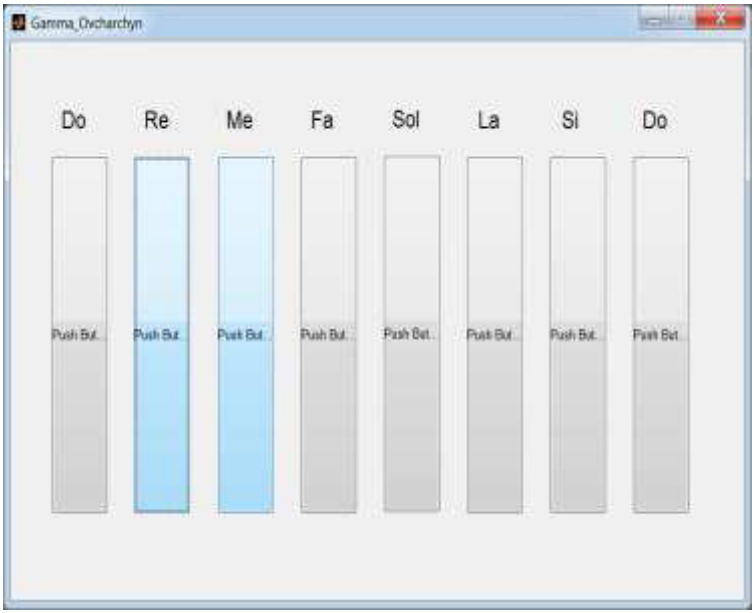

Рис. 3. Графічна програма, що відтворює третю октаву; кожна клавіша відповідає певній стандартні ноті (надписана). Може виконувати гамми та кілька більш складні музичні фрагменти

Рис. 4. Фрагмент музичного твору, що відтворено у програмі *havaNagila.m*.

диктуванням, створення голосового пароля для комп'ютера тощо. Цьому можуть бути присвячені наступні курсові роботи студентів.

Література

1. Computer Science Curricula 2013. http://www.acm.org/education/CS2013-final-report.pdf
2. Рекомендации по преподаванию информатики в университетах Computing Curricula 2001: Computer Science. – С.-Петербург, 2002 (http://window.edu.ru/window/library? p_rid=23885).

3. Стаття англ. Вікіпедії "Computer science", http://en.wikipedia.org/ wiki/ Computer_science
4. Статті російської Вікіпедії “Информатика”, “Компьютерные науки”, https://ru.wikipedia.org/wiki.
5. Чен К., Джиблин П., Ирвинг А. MatLab в математическом исследовании. – М.: Мир, 2001. – 346 с.
6. Austin M., Chancogne D. Introduction to Engineering Programming: in C, MATLAB, and Java, 1999.
7. Лазарєв Ю.Ф. Початки програмування в середовищі MatLAB. Навч. посібник. – К.: “Політехніка”, 2000. – 396 с
8. Кондратов В.Е., Королев С.Б. MATLAB как система программирования научно-технических расчетов. – М.: Мир, 2002. – 350 с.
9. Гаєв Є.О., Нестеренко Б.М. Універсальний математичний пакет MatLab і типові задачі обчислювальної математики. Навчальний посібник.– К.: НАУ, 2004. – 176 с.
10. Gayev Ye.A., Nesterenko B.N. MATLAB for Math and Programming: Textbook. – Zaporozhye: Polygraph, 2006 – 102 p.
11. Гаєв Є.О., Нестеренко Б.Н. Опыт и предложения по использованию MATLAB в курсах математики и информатики\\ Тези XI Міжнародної наукової конференції ім. акад.. М.Кравчука. К.: КПІ, 2006.
12. Гордеев О. Программирование звука в Windows. 1999. – ?? с.
13. Ананьев А.Б., Ананьева Е.А., Путилова А.Ю. MATLAB для акустиков, а также всех, кто собирается создавать и обрабатывать различного рода сигналы: учебное пособие. - К.: [ПП ВФ], 2007. - 192 с.
14. Статті російської Вікіпедії “Компьютерная музыка”, “Музыкально-компьютерные технологии”, “Звуковой и музыкальный компьютинг”, https://ru.wikipedia.org/wiki.
15. Sound and Music Computing, List of Software Tools, http://smcnetwork.org/ view/software.
16. Чесебиев И.А. Компьютерное распознавание и порождение речи. М.: Спорт и культура-2008. 128 с.
17. Digital audio http://en.wikipedia.org /wiki/Digital_sound.
18. Цифровий звук. Стаття в українській Вікіпедії http://uk.wikipedia.org/wiki
19. Музыченко Е. Низкоуровневое программирование звука в Windows Компьютер Пресс №6, 2000. http://www.rsdn.ru/article/multimedia/winsnd.xml

20. Программирование звука в Windows (C++) http://soundcoding.ru/
21. Секунов Н. Обработка звука на PC в подлиннике.
22. https://ru.wikipedia.org/wiki/ Статьи “ДО (нота)”, “РЕ (нота)”, “Октавная система”.
23. https://ru.wikipedia.org/wiki/, Стаття "Тембр".
24. https://en.wikipedia.org/wiki/Timbre, стаття "Timbre".
25. Гаєв Є.О., Рожок А., Овчарчин Н. Звук та музика в курсі програмування. Інженерія програмного забезпечення. 2014. №19. С. 41-47.

10.3. Висновки за розділом 10

У цьому, останньому, розділу навчального посібника ми виклали наші пропозиції до ваших самостійних курсових робіт. Віднесіться до них творчо! Це значить, що ми вітаємо всі ваші зустрічні пропозиції. Наше коригування ваших ідей буде полягати у тому, аби надати вашій фантазії рис реальності, відокремити складову, яку дійсно реально зробити в обмежений час (від двох тижнів до місяця), з тими засобами (знаннями), що ми маємо. Врахуйте і те, що і викладач у певних галузях має обмежений досвід, або не має його взагалі. Тобто певний проміжок часу піде на “перемовини” з викладачем.

А далі – за роботу. Самостійну! Зверніть увагу, що ваша програма і вся проведена робота має бути “укладена” у певний формат Звіта. Зразок, наведений в 10.2.2 містить, як ми попереджали у 10.2.1, стандартні розділи *Вступ*, *Постановка питання*, *Сучасний стан питання*, поділ на підзавдання у головних розділах 2 і 3, *Висновки* та, *Література*.

Післямова

Наприкінці ***хотілося б*** сказати "*Ми з вами вивчили весь MATLAB*". Шкода, та це було б неправдою, так само, як стверджувати "*Ми з вами вивчили всі науки*". Бо MATLAB – це і є **майже всі науки**! І дійсно, у цьому пакеті багато чого ще лишилося, наприклад: Aerospace Blockset, Aerospace Toolbox, Bioinformatics Toolbox, Financial Instruments Toolbox, SimBiology, Simulink, 3D Animation, Audio System Toolbox, Computer Vision System Toolbox, Image Processing Toolbox, Simscape Fluids – і таке інше, загалом 66 продуктів (*Toolbox*-ів).

Насправді, те що вивчили, чим оволоділи – називається **Програмування**, **Computer Science**. Ми оволоділи алгоритмізацією лише деяких проблем, далеко не простих, та навчилися "викладати" ці алгоритми комп'ютеру у середовищі MATLAB. Це середовище виявилося дуже зручним та ефективним – не даремно MATLAB став стандартом у більш як 5000 (п'яти тисячах!) університетах світу.

Наша порада вам на майбутнє – використовуйте MATLAB у вашому подальшому навчанні. Так, у наступних дисциплінах Національного авіаційного університету, що ви будете вивчати, систематично використовуються Control System Toolbox (інструментарій систем управління), Data Acquisition Toolbox (інструментарій аналізу даних), Signal Processing Toolbox (для обробки та аналізу будь-яких сигналів), Optimization Toolbox (для знаходження "найкращих", "найвигідніших" з можливих рішень проблеми), Simulink. З нашого підручника ви взяли все потрібне для оволодіння також і названим матеріалом.

Особливі перспективи має Parallel Computing Toolbox. Це – нова, сучасна можливість набагато прискорити всі обчислення. Ви знаєте, що сьогодні не лише комп'ютери, а й навіть мобільни телефони (вони є також, власне, комп'ютери!) мають кілька процесорів. Тобто, математична проблема розділяється на кілька допоміжних задач, кожна з них розв'язується на окремому процесорі, а потім розв'язки

“зустрічаються” на одному з них, що керує, і на їх підставі виробляється остаточний розв’язок. Новітня технологія розпаралелювання алгоритмів не лише прискорює розв’язки, а й “перевертає” науку алгоритмізації з ніг на голову (а, може, з голови на ноги**-:)**. Наприклад, розв’язок СЛАР за допомогою метода Крамера, який, здавалося, залишився лише для навчальних цілей, знов набуває нового значення!

Звертаємо також увагу на такий особливо важливий інструмент MATLAB, як Simulink. Він дозволяє робити майже візуальне дослідження складних систем, як технічних, так і економічних, соціальних. Вивченню такого інструменту присвячені десятки книжок. Радимо вам присвятити йому певний час і увагу!

Проте, MATLAB – лише одна з десяти найбільш популярних у світи алгоритмічних мов програмування високого рівня. І хоча науковий прогрес рухається у напрямку створення єдиної мови спілкування з комп’ютером, доцільно ознайомитися прийнамні ще з однією з них. Програмуванню із мовою Java присвячуємо наступну частину нашого підручника. Ви узнаєте багато нового. Та одночасно упевнитись, як багато з комп’ютерної науки вам вже знайомо! План цієї третьої частини підручника надано у Змісті.

(:-

ЛАБОРАТОРНІ РОБОТИ ДО МОДУЛІВ 3 і 4

Мета цього розділу полягає у тому, аби студенти на практиці засвоїли і закріпили все, що вивчають у теоретичній частині даного посібника. Окрім конкретних завдань, що тренують ваші навички програмування, деякі роботи містять й приблизний план – що саме зробити, на що звернути увагу. Студентам пропонується створити програми, випробувати ті чи інші підходи, пояснити зроблене у Звіті, у хелповій частині програм (аби вони нагадували чудові "рідні" програми MATLAB) та у коментарях, а також провести певне дослідження та експерименти з об'єктом, що запрограмовано.

Викладач очікує від вас Звіт за кожною з лабораторних робіт у довільному форматі: що саме ви зробили, що цікавого знайшли та – чому ні? – чи є якісь відкриття ☺? Бажано не лише виконувати та описувати для викладача, та ще робити висновки: заради чого це? Що саме ви узнали? Та обов'язково Висновок наприкінці: чи досягнута мета роботи?

Лабораторна робота № 7

Все ще перший рівень майстерності програміста

Мета: *пригадати засоби програмування, що вивчалися у першій частині посібника* [1], *застосувати їх* у *програмах, доволі ще простих.*

1. Пояснити використання операторів логіки ***if***, ***else***, ***elseif***. Використайте їх у програмуванні кусково-періодичних функції (з періодом $T=2\pi$), визначених на всій числовій осі $-\infty < x < \infty$. Номер варіанта обирайте у відповідності до вашого номера у списку студентів групи[35]:

   1.1. $f(x) = x^2, \quad x \in (-\pi, \pi)$;

[35] Тобто номер варіанта *VarN*=*mod*(*N_list*,3)+1, якщо ваш номер у списку є *N_list*. (Подивіться, >> *help mod*).

1.2. $f(x)=\begin{cases} -x-\dfrac{\pi}{2}, & x\in(-\pi,0) \\ +x-\dfrac{\pi}{2}, & x\in(0,\pi) \end{cases}$;

1.3. $f(x)=\begin{cases} -1, & -\pi<x<0; \\ \dfrac{x}{2}, & 0<x<\pi. \end{cases}$

(Нагадаємо, що діяти слід крок за кроком: (*i*) зробити програму для поодинокого аргументу не звертаючи увагу на період *T*; (*ii*) модифікацію програми для векторного аргументу; та нарешті (*iii*) врахувати періодичність).

2. Поясніть використання операторів ***for … end***, ***while … end***. Застосуйте їх до однієї з задач анімації відповідно до номеру у списку групи:

   2.1. Обертати на екрані PC трикутник за годинниковою стрілкою або проти неї.

   2.2. Обертати на екрані PC квадрат за годинниковою стрілкою або проти неї.

   2.3. Зробіть на екрані PC круг, що пульсує (маленький спочатку; росте-росте; зменшується, знов росте – і так кілька разів).

3. Зробіть програму, що обертає на екрані правильний *N*-кутник того чи іншого кольору.

4. Програми 2 і 3 є в вас скриптом чи *m*-функцією? Перетворіть їх в діалогову програму (Обов'язково зберігши під новим ім'ям, як модифікацію попередньої програми!).

Цю роботу та всі подальші лабораторні роботи з програмування треба здавати викладачеві не лише із Звітом, та ще й з CD-R або флешкою, де записані (перезаписані) ваші *m*-файли.

Лабораторна робота № 8
Більше цікавих програм!

Ви вже оволоділи *більшістю засобів процедурного програмування, і здатні тепер робити розумні програми, що реалізують доволі складні алгоритми. Отже, тренуйтеся. Навчитися можна лише на практичних задачах!*

1. Розробіть програму ***MyMaxArray*** (***MyMinArray***), що знаходить найбільший (найменший) елемент числового масиву у кожному рядочку (колонці) аналогічно до ***MyMinArray*** у посібнику. Поясніть алгоритм у словах та за допомогою блок-схеми [13,14]. Першу роблять студенти з парними номерами у списку групи, другу – з непарними.

2. Розробіть програму ***MySortArray***, що пересортировує числа з заданого 2*d*-масиву відповідно до ключового слова ***Increase*** (із збільшенням) або ***Decrease*** (із зменшенням), як це було з програмою ***MyOrder***, та використаний алгоритм поясніть за допомогою блок-схем (студенти з парними номерами упорядковують за стовпчиками матриці, з непарними – за рядочками).

   Порада: креслення блок-схем легко робиться програмою ***Diagram Designer*** [12] або інструментами ***Word***. В останньому випадку треба використати **Автофигуры** за шляхом **Сервис>Настройка>Рисование**. Також користуйтеся меню **Группировка**, **Порядок**(на передний план, на задний) та **Формат объекта**…, що викликаються правою клавішою миші. Треба стандарти для блок-схем [13,14]. Намагайтеся також використовувать ваші власні програми, розроблені раніше.

3. Пригадайте лабораторну роботу № 1, завдання 15. Там ви обчислювали “руками”, а тут – розробіть програми обчислення *MySin* або *MyCos* залежно від номера у списку групи, парний чи непарний. Програма має використовувати задану кількість доданків ряду Тейлора,

3.1. $$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots + (-1)^n \frac{x^{2n+1}}{(2n+1)!} = \sum_{k=0}^{n} (-1)^k \frac{x^{2k+1}}{(2k+1)!},$$

3.2. $\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots + (-1)^n \frac{x^{2n}}{(2n)!} = \sum_{k=0}^{n} (-1)^k \frac{x^{2k}}{(2k)!}$.

4. Розробіть іншу версію попередньої програми, що гарантує задану точність обчислення *MySin*Eps, або *MyCos*Eps, обираючи кількість доданків, *n*, залежно від точності ε, скажімо $\varepsilon = 0{,}1$ або $\varepsilon = 0{,}001$. Не забудьте про блок-схему програми!

Лабораторна робота № 9

Робимо більш інтелектуальні програми

Мета: *поглибити досвід роботи з алгоритмами сортування, оволодіти складними типами даних, такими як* ***cell*** *та* ***struct***, *використати їх у доволі складних алгоритмах та програмах.*

1. Зробити програму, що працює з листом із слів *Names* та пересортовує їх за алфавітом чи у зворотньому порядку. Приклад: якщо *Names*={‘Sasha’,’Наташа’, ‘Masha’, ’Аня’, ‘Alex’}, програма повертає *Y*={‘Alex’, ‘Masha’, ‘Sasha’, ’Аня’ , ‘Наташа’} (за алфавітом).
2. Модифікуйте вашу програму ***MyOrder.m*** (повертає чисельний вектор або матрицю у якості вхідного аргументу, але у порядку зростання або падіння, див. 5.3.2) таким чином, аби ‘розуміти’ кілька вхідних та/або вихідних аргументів.
   Вказівка: використати ключові слова *var**arg**in*, *var**arg**out* та *n**arg**in*, *n**arg**out*.
3. Зробіть невеличку, просту колекцію даних “*MyGroup*”, “*MyLibrary*” або інше за вашим вибором, разом з програмою для її використання. Студенти з парними номерами у списку групи використовують тип даних ***cell**s*, з непарними номерами – ***struct***.

4. Зробіть програму обчислення визначника (детермінанта) квадратних матриць ***n*n***; алгоритм обчислення базувати на відомій теоремі для диагональних матриць

$$\begin{vmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ 0 & a_{22} & \dots & a_{2n} \\ \dots & \dots & & \\ 0 & \dots & 0 & a_{nn} \end{vmatrix} = a_{11}a_{22}..a_{nn}$$

(Якщо квадратная матриця є диагональною, тобто всі її елементи нижче головної діагоналі дорівнюють нулю, то її визначник дорівює добутку діагональних елементів).

Лабораторна робота № 10

Алгоритми ітерації

Ітерації, ітерування *– один з важливіших алгоритмів комп'ютерних наук. Пропонуємо дві задачі, змістовні і красиві за їх математичним змістом, та складні з точки зору програмування.* ***Мета*** *– оволодіти цим важливим алгоритмом!*

1. Починаємо вивчати і досліджувати новий та важливий алгоритм ***Ітерації***. Дослідіть збіжність ітерацій, якщо цим методом розв'язувати такі рівняння:

   а. Трансцендентне (не існує аналітичного розв'язку)

   $$2^x + 5x - 3 = 0\,.$$

   Як тут *гарантувати задану точність* розв'язку, скажімо $\varepsilon = 0{,}05$? Як гарантувати $\varepsilon = 0{,}001$? Це означає таке: скільки треба ітерацій, перш ніж зупинити ітераційний процес?

b. Алгебраїчне рівняння $x^2-5x+6=0$. Обирайте початкове значення з диапазону $|x|\le 3$ та дослідіть, чи збігаються ітерації до коренів цього рівняння?

Порада: графічне представлення результату ітерування може бути різним; спробуйте кілька – яке краще? Якщо коренів декілька, чи однаково поводять себе ітерації по відношенню до них?

2. Самостійно, не підглядаючи у наш підручник, розробіть діалогову програму, що запитує в користувача ввести якусь функцію $f(x)$ та початкове наближення x_0, після чого утворює "павутину" ('SpiderWeb'), як то запропоновано в підручнику [15]:

a. Обчислюємо $x_1=f(x_0)$ та проводимо вертикаль від $(x_0,0)$ до $(x_0 , x_1=f(x_0))$;

b. Останню з'єднуємо горизонтальною лінією з точкою (x_1,x_1), що лежить на діагоналі $y=x$;

c. Робимо ще ітерацію $x_2=f(x_1)$ та проводимо знов вертикальний сегмент від (x_1 , x_1) до (x_1 , x_2) на кривій $y=f(x)$, як в п. *a*;

d. Знов проводимо горизонталь до (x_2 , x_2), як в *b*;

e. Робимо таке много разів – отримуємо красиву "павутину" та, наприкінці, висновок щодо збіжності ітерацій $x_{i+1}=f(x_i)$.

Приклади для дослідження: (1) $f(x)=\cos(x)+1$, $x_0=-pi/4$,

(2) $f(x)=\cos(x)+\sin(2*x)$, $x_0=$ pi/4 or pi.

3. Розробіть програму, що крок за кроком будує відомий фрактал Коха "Сніжинка". Пропонуємо не підглядати у підручник, а самостійно реалізувати такий алгоритм:

3.0. Будуємо правильний трикутник (Seeding Triangle);

3.1. Кожну сторону ділимо на 3 сегменти однакової довжини;

3.2. За двома точками, що отримано в 3.1, знаходимо третю, що утворює правильний трикутник з ними;

3.3. За усіма щойно отриманим точками проводимо полігон;

3.4. Повторюємо 3.1 – 3.3 багато разів, кожного разу отримуючи все красивішу "Сніжинку", рис. 7.7-7.10.

Лабораторна робота № 11

Алгоритми рекурсії

Мета: *Оволодіти рекурсивними алгоритмами*, *порівняти їх із "звичайними"*, *ітеративними програмими.*

1. Самостійно зробіть програми для обчислення
   a. factorial $n! = 1 \cdot 2 \cdot \ldots \cdot n$;
   b. Чисел Fibonacci 0, 1, 2, 3, 5, 8, 13, 21, …;

У двох модифікаціях – з рекурсією *MyFactorialR* та без неї, *MyFactorialI*, *MyFibonacciR* та *MyFibonacciI*.

2. Визначника довільного порядку *n* як з рекурсією, так і без неї; ваша думка про ці методи, який краще?

Алгоритми завдань № 1 та № 2 поясніть Блок-схемами, [14,15].

3. Розробіть програми обчислення *Daubechies* (Добєши) *Scaling Function* $\varphi(r)$, що визначається рекурсивно:

$$\varphi(r) = h_0 \cdot \varphi(2r) + h_1 \cdot \varphi(2r-1) + h_2 \cdot \varphi(2r-2) + h_3 \cdot \varphi(2r-3),$$

при цьому

$\varphi(r) = 0,\ if\ \ r \le 0,\ \varphi(1) = 2h_0,\ \ \varphi(2) = 2h_3$ та $\varphi(r) = 0,\ if\ \ r \ge 3,$

з коефіцієнтами $h_0 = \frac{1+\sqrt{3}}{4}$, $h_1 = \frac{3+\sqrt{3}}{4}$, $h_2 = \frac{3-\sqrt{3}}{4}$, $h_3 = \frac{1-\sqrt{3}}{4}$. Побудуйте її графік. Яка ваша думка, це функція *дискретна* чи *неперервна*?

4. Обчисліть функцію *Daubechies Wavelet Function* $\psi(r)$, що визначається через попередню (непряма рекурсія)

$$\psi(r) = -h_0 \cdot \varphi(2r-1) + h_1 \cdot \varphi(2r) - h_2 \cdot \varphi(2r+1) + h_3 \cdot \varphi(2r+2).$$

Лабораторна робота № 12

Створення першої програми з графічним інтерфейсом, GUI.

Мета: *навчитися "обертати" у GUI ваші програми, поглибити ваші можливості у створенні не лише розумних, та й ще красивих та сучасних програм.*

1. Зробіть GUI-програму, де можна ввести дані про правильний *N*-кутник: скільки вершин *N* хочемо мати, якого кольору, у якому напрямку та з якою швидкістю обертати.

   Радимо робити такими кроками:

   1.1. виконати команду ***guide***, оберіть заготовку GUI;

   1.2. вирішіть, ввід яких параметрів має відбутися, якими GUI-компонентами, що запропоновані *guide*;

   1.3. задайте властивості обраних GUI-елементів через *Property Inspector*; особлива увага на ***Tag***

   1.4. ієрархію GUI-елементів подивіться через *Object Browser*;

   1.5. збережіть ваше GUI;

   1.6. працюйте з *m*-файлом, асоційованим із створеним GUI, через *Callback*s організуйте зв'язок між підпрограмами.

   1.7. збережіть остаточну програму, тестуйте її відлагоджуйте.

2 Як приклад, радимо відтворити програму ***Helicopter***, де передбачити потрібні GUI-елементи для обертання за годинниковою стрлкою та протинеї, управління кольором, шириною "гвинта" тощо.

3. Дослідіть й напів-готові GUI-засоби:

   3.1. команду ***menu***: як отримати інформацію щодо її елементів, як можна впливати на властивості графчного меню, яке вона створює? Поясніть використання команд ***gcf***, ***gco***, ***get*** та ***set***, що тут потрібні.

   3.2. дослідіть команду ***uicontrol***: які "стилі" GUI-елементів вона створює? Як управляти на їх властивості?

4. Подібним чином дослідіть та “пограйтеся” з командами ***uisetcolor, uiputfile, fwrite, save, saveas, errordlg, warndlg, helpdlg, msgbox, questdlg, inputdlg, uigetfile, uiputfile, waitbar.*** Наведіть приклади, як їх можна використати у програмах!

5. ***Для просунутих студентів***: Зробіть програму подібну до № 2, що обертає *N*-пелюсткову розу (наприклад, для *N*=3 *ezplot*('sin(3**t*)*cos(*t*) '**,** 'sin(3**t*)*sin(*t*)',[0, *pi*])).

> **Увага:** Маючи GUI-програму, викладачеві треба тепер здавати на CD або флешці не лише *m*-файли, та й ще *fig*-файли!

Лабораторна робота № 13
Оцінка швидкодії ваших програм

Мета: *поглибити знання про фундаментальні проблеми комп’ютерної науки*, *оцінити ефективність раніше створених MATLAB-програм аналітично* (*якщо можливо*) *та “експериментально”*, *дати не лише якісний аналіз* (*краще-гірше, швидше-повільніше*), *а й кількісний за допомогою інструменту* ***Tools>Basic Fitting*** [1].

1. Як можна характеризувати *ефективність* програм? Проілюструйте ваше розуміння порівнянням пари нижченаведених програм.
2. Підрахуйте, скільки часу роботи вашого комп’ютера потрибують програми розв’язку СЛАЕ в залежності від її вимірності *n*, порядок (складність) СЛАЕ, а саме програми “*MyKramer.m*”, “*MyGauss.m*”, “рідні” програми *inv* та \-метод. Побудуйте графіки, що ілюструють залежність “ефективності” від складності *n*, що необмежено зростає, виявити порядок зростання $O(n)$. Використайте ***Tools>Basic Fitting***.

Ваш висновок?

3. Порівняйте складність (*ефективність*) рекурсивних та ітеративних програм роботи № 12. Власний варіант оберіть за формулою *VarN*=*mod*(*N_list*,3)+1):
   3.1. ваших MATLAB-програм, що обчислюють факторіал;
   3.2. ваших MATLAB-програм, що обчислюють числа Фібоначчі;
   3.3. ваших MATLAB-програм для обчислення визначників.

Ваш висновок?

4. Спробуйте оцінити ефективність наступних ваших програм аналітично:
   4.1. ***SnowFlake***;
   4.2. ***Daubechies Function*.**

Також, аналогічно до № 3, виміріть час “експериментально”; порівняйте з теорією.

(Для вибору варіанту знов використайте алгоритм *VarN*=*mod*(*N_list*,2)+1).

Висновок?

С П И С О К Л І Т Е Р А Т У Р И

1. **Азарсков В.М., Гаєв Є.О.** Сучасне програмування. Модулі 1,2 “Програмування та математика із другом MATLABом”. К.: НАУ, 2014. – 256 с.
2. **Gayev Ye.A., Nesterenko B.N.** MATLAB for Math and Programming: Textbook. – Zaporozhye: Polygraph, 2006 – 102 p. (також http://www.exponenta.ru/educat/systemat/gayev/index.asp , http://www.twirpx.com/file/1707755/)
3. **Гаєв Є.О., Нестеренко Б.М.** Універсальний математичний пакет MATLAB і типові задачі обчислювальної математики: Навчальний посібник. – К.: НАУ, 2004. – 176 с.
4. Інтернет-музей Інституту кібернетики НАН України, http://www.icfcst.kiev.ua/MUSEUM/museum-map_r.html, http://ukrainiancomputing.org/museum-map_r.html, https://ru.wikipedia.org/wiki/МИР-2
5. Wikipedia, Graphical User Interface. http://en.wikipedia.org/wiki/Graphical_user_interface, https://uk.wikipedia.org/wiki/Графічний_інтерфейс_користувача
6. **Tuck M.** The Real History of the GUI. http://www.sitepoint.com/article/real-history-gui.
7. **Патий Е.** 19 ступеней вверх, или История графических пользовательских интерфейсов. – "IT News", #18/2005 (http://smoking-room.ru/data/pnp/gui_history/)
8. https://uk.wikipedia.org/wiki/Методологія_програмування
9. https://uk.wikipedia.org/wiki/Структурне_програмування
10. https://uk.wikipedia.org/wiki/ /Процедурне_програмування
11. https://uk.wikipedia.org/wiki/Об'єктно-орієнтоване_програмування
12. Програма *Diagram Designer* http://www.cyberforum.ru/algorithms/thread101900.html
13. Блок-схема. http://ru.wikipedia.org/wiki/Блок-схема
14. Блок-схема алгоритма. http://shkolo.ru/blok-shema-algoritma/
15. **Чен К., Джиблин П., Ирвинг А.** MATLAB в математическом исследовании. – М.: Мир, 2001. – 346 с.

16. **Вирт Н.** Алгоритмы и структуры данных. СпБ..: Изд-во "Невский диалект", 2001. – 352 с.
17. **Ахо А.В., Хопкрофт Д.Э., Ульман Д.Д.** Структуры данных и алгоритмы. – Москва-Киев: Изд. дом. "Вильямс", 2003. – 384 с.
18. Стаття “Крива Коха”, https://uk.wikipedia.org/wiki/Крива_Коха
19. **Гринченко В.Т., Маципура В.Т., Снарский А.А.** Фракталы: от удивления к рабочему инструменту. – К.: Наук.думка, 2013. – 270 с.
20. **Каліон В.І., Черняк О.І., Хартонов О.М.** Основи інформатики. Структурне програмування на ПАСКАЛІ. Практикум. К.: Центр учбової літератури, 2007. – 248 с.
21. **Пойа Д.** Как решать задачу. М.: Учпедгиз, 1959, 207 с.
22. https://ru.wikipedia.org/wiki/Порочный_круг
23. **Nievergelt Y.** Wavelets made easy. Birkhöuser Boston, 1999. – 297 p.
24. **Сэломон Д.** Сжатие данных, изображений и звука – М.: Техносфера, 2004. - 368с.
25. Цікаві приклади рекурсії. https://lurkmore.co/Рекурсия https://lurkmore.co/Рекурсия
26. **Лазарєв Ю.Ф.** Початки програмування в середовищі MatLAB. Навч. посібник. – К.: “Політехніка”, 2000. – 396 с.
27. **Кривилев А.** Основы компьютерной математики с использованием системы MATLAB. М.: Лекс-Книга, 2005
28. **Austin M., Chancogne D.** Introduction to Engineering Programming: in C, MATLAB, and Java. John Wiley & Sons, Inc., 1999.
29. **Гаєв Є.О., Рожок А., Овчарчин Н.** Звук та музика в курсі програмування. Інженерія програмного забезпечення. 2014. №19. С. 41-47.
30. Тестування програмного забезпечення, https://uk.wikipedia.org/wiki/Тестування_програмного_забезпечення
31. **Савин Р.** Тестирование Дот Ком, или Пособие по жестокому обращению с багами в интернет-стартапах. — М.: Дело, 2007. — 312 с.

32. **Гаев Е.А., Мартич М., Тарак Г.** Программы моделирования случайных явлений для изучения программирования и математики. Информационные технологии в образовании, 2015, № 23, с. 30-42. (http://ite.kspu.edu/webfm_send/829)
33. **Гаев Е.А., Малинина Д.** Параметрическая роза – предмет математики, программирования, эстетики. Информационные технологии в образовании. (у друку).
34. Національний стандарт України ДСТУ 8302:2015 "Бібліографічне посилання". К: ДП «УкрНДНЦ», 2016. -- 20 с.

Покажчик команд і термінів

GUI, графічний інтерфейс
guide – команда виклику середовища розробки GUI
input( )

int – команда інтегрування функції (заданої аналітично або чисельно)

inv()

ischar( ) та isletter( )
iscell

isinteger( )
isnumeric( )
inputdlg – діалогова команда
legend( )
load(

lower( )

questdlg – діалогова команда з позначкою «?»
menu – діалогова команда
msgbox – діалогова команда

MyClererOrder – авторська програма з довільною кількість аргументів, що упорядкує масив чисел, поданих на вхід

MyCramer – авторська програма розв'язку СЛАР методом Крамера

*MyDet*1 – авторська програма обчислення визначника методом Гаусса

*MyDet*2 – авторська програма обчислення визначника методом рекурсії

MyGauss – програма розв'язку довільної СЛАР методом Гауса

MyGroup – авторська MATLAB-програма, що оперує з базою даних про студентську групу

MyFactorial – програма обчислення факторіалу n! методами циклу та рекурсією

MyFibonacci – програма обчислення чисел Фібоначчі методами циклу та рекурсією

MyLibrary – авторська MATLAB-програма, що оперує з базою даних про книжки персональної бібліотеки

MyMax, *MyMin* – програми знаходження найбільшого й найменшого значень векторного аргументу

MySinEps – програма обчислення функції sin(*x*) з заданою точністю *Eps*

MyOrder – авторська програма упорядкування вектора чисел, поданого на вхід

MyOrderArray – авторська програма упорядкування масива чисел, поданого на вхід

MySinN – програма обчислення функції sin(*x*) з заданою кількістю *N* членів ряду Тейлора

MyWordOrder – авторська програма упорядкування вектора слів, поданого на вхід

NormalPoint – підпрограма для фракталу *SnowFlake*

*num*2*str*() – команда, що повертає число-аргумент як текстову величину

pause( ) – команда, що зупиняє програму до торкання будь-якої клавіши, або на вказану в аргументі кількість секунд

pretty( ) –

prod( )
plot( ) –

rand() – команда генерації випадкових чисел від 0 до 1

randi() – команда генерації цілих випадкових чисел

save – команда для збереження файлу (даних) аргументу

set( ) – команда для встановлення властивостей об'єкту

SeedingTriangle – підпрограма для фракталу *SnowFlake*

size() –

sound() –

simplify( ) – аргумент типу *syms* повертає у „найпростішому", як вона вважає, вигляді

simple( ) – аргумент об'єкт типу syms; повертає кілька таких же, аби користувач обрав „найкращий".
SnowFlake, фрактал Коха – авторська програма
SpiderWeb авторська програма
strcat
str2num( )
struct

switch-case-end – одна з структур програмування

syms( )

tic, *toc* – пара команд, що дозволяю вимірювати час виконання програмного блоку, розташованого між ними

title
Trisection – підпрограма для SnowFlake

try-catch-end – програмна структура, яка дозволяє розділяти потік виконання у залежності від аргументів

uicontrol – команда програмування елементів GUI

uigetfile – діалогова команда пошуку файла

uiputfile – діалогова команда запису файла у пам'ять комп'ютера

uisetcolor – діалогова команда вибору кольору

upper() – команда повертає текстовий вектор з лише великими літерами

VarArgIn – ключеве слово для обробки довільної кількості вхідних аргументів програми (всі літери писати у нижньому регистрі!)

VarArgOut – ключеве слово для обробки довільної кількості вихідних аргументів програми (всі літери писати у нижньому регистрі!)

vertcat –

Fitting Toolbox – інструмент для пошуку кривої і її рівняння, "найближчої" до заданих точок
бази даних – с.

GUI (графічний інтерфейс)

Java – посилання на відповідну мову програмування

Багатопелюсткова роза (роза)

Блок-схеми

Вейвлет-функція Добєши

визначник

Навчальне видання

ГАЄВ Евген Олександрович
АЗАРСКОВ Валерій Миколайович

Сучасне програмування
Частина 2: "Складні типи даних та алгоритми, інтелектуальні програми"

Навчальний посібник

В авторській редакції
Коректор Єрмолаєва Л.І.

Студенти англомовного проекту про цей курс:

Назарій Овчарчин, 2013/2014:

Now, I understand that I really like that. ...
When program is ready, you can see that it works.
You feel yourself in paradise!

Пурдик Наталія, 2013/2014:

Laboratory Works were useful for me in MATLAB class because we have taught the topics from them.