СОФИЙСКИ УНИВЕРСИТЕТ СВ. КЛИМЕНТ ОХРИДСКИ
ФИЗИЧЕСКИ ФАКУЛТЕТ

Ръководство по изчислителна физика

гл. ас. д-р Стоян Писов

София, 17 януари 2018 г.

Съдържание

Увод

Настоящото ръководство е предназначено за студентите по физика във Физически факултет на Софийски университет „Св. Климент Охридски" от бакалавърските програми по Медицинска Физика (МФ) и Квантова и Космическа Теоретична Физика (ККТФ). Съдържанието на ръководството е взаимствано от книгите *Числени рецепти* [8], *Изчислителна физика* [5, 12], на него трябва да се гледа като съпътстващо учебно пособие към по-пълни добре развити учебници по числени методи и изчислителна физика.

Разгледаните примерни физични задачи предполагат студентите да усвоили серията курсовете по Математични Методи на Физиката (ММФ), Теоретична Механика (ТМ), Квантова Механика (КМ), курсът по Термодинамика и Статистическа Физика (ТСФ) е препоръчителен. В ръководството са използвани програмните езици *Fortran 90* и *C/C++*, а в някои случаи решенията на задачите са демонстрирани с езици за символни пресмятания: *Mathematica*, *Maple* и *Matlab*. Използваните два програмни езика не са избрано случайно. С годините *Fortran* се е наложил като език за програмиране на научната общност поради своят опростен синтаксис, лесния начин за представяне на алгебрични изрази, поддръжката на аритметика както с реални така и с комплексни числа. В практиката съществуват множество комерсиални компилатори на програмният език *Fortran* на софтуерни разработчици като Intel, The Portland Group, PathScale, IBM XL Fortran и др., а също и свободни такива като GFortran, G95 и Oracle Solaris Studio. Сериозното присъствие на лицензни компилатори подсказва съществуването на сериозен интерес към този програмен език. В допълнение трябва да се отбележи наличието на множество програмни библиотеки за математични пресмятания специализирани за задачи от Линейната алгебра и Фурие трансформациите които имат реализации (интерфейси) за програмният език *Fortran*.

Използванто на програмният език *C/C++* се налага от неговата изключителна популярност, която може би се дължи на факта че този програмен език се смята за основен или системен език за програмиране. Той често се разглежда и като основен език за програмиране в повечето университетски курсове по програмиране.

Глава 1

Линейни системи

В практиката много задачи от физиката се свеждат до решаването на системи линейни уравнения, например: Задачата за намиране собствените стойности и собствените вектори в квантовата механика, експлицитните методи за решение на частни диференциални уравнения, нелинейното напасване на функционални зависимости, намирането на електростатичният потенциал (уравнение на Поасон) и много други. Поради своето широко приложение ще бъде добре да разгледаме сравнително обширно методите за решаване на системи линейни уравнения. В първия раздел от настоящата глава се припомнят общите положения, дефиниция на системи линейни уравнения, нейното матрично представяне, като се прави кратък обзор на основните числени методи и широко използвани библиотеки, които ги реализират. Следващите раздели разглеждат последователно методите са решаване на системи линейни уравнения: Гаус-Жордан, LU-декомпозиция, разглеждат се специални техники за решаване на тридиагонални и разредени линейни системи. В последният раздел се разглежда примерната задача за намиране формата на не-разтеглива нишка с тегло m и дължина L, при която се сравнява аналитичното и числени решения.

1.1 Системи линейни уравнения

Нека разгледаме система от M на брой линейни уравнения за N неизвестни $\{x_j\}$ където $j = 1, ..., N$, която може да се запише в най-общия случай като:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + ... + a_{1N}x_N &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + ... + a_{2N}x_N &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + ... + a_{3N}x_N &= b_3 \\ \\ a_{M1}x_1 + a_{M2}x_2 + a_{M3}x_3 + ... + a_{MN}x_N &= b_M \end{aligned} \tag{1.1}$$

Ще предполагаме също така че познаваме множеството от коефициенти $\{a_{ij}\}$ за $i = 1, ..., M$ и $j = 1, ..., N$, както и коефициентите $\{b_i\}$ за $i = 1, ..., M$ от дясната страна

на уравнението (1.1). В случаите когато $M = N$ имаме същият брой уравнения M както неизвестни N и можем да очакваме да намерим единствено решение за x_j, $j = 1, ..., N$ при дадено множество $\{b_i\}$ (или обратната задача). Намирането на решение на системата 1.1 зависи обаче от множеството коефициенти $\{a_{ij}\}$ и $\{b_i\}$, т.е по-същество те определят решението на нашето уравнение 1.1. Затова нега обърнем внимание на свойствата на тези коефициенти $\{a_{ij}\}$, които могат да притежават.

Една система от линейни уравнения се нарича изродена когато можем да намерим линейна зависимост между елементите $\{a_{ij}\}$, които принадлежат на даден ред или колона. В случаите когато поне едно от M-те уравнения може да се представи като линейна комбинация на други уравнения от системата (1.1) ще казваме че една система линейни уравнения притежава *изграждане по редове*, когато **всички** уравнения съдържат дадени променливи x_j в една и съща линейна комбинация ще казваме че линейната система (1.1) е *изродена по колони*. Ако $M \equiv N$, т.е. разглеждаме случаят на квадратна матрица изграждането по редове е равносилно на изграждането по стълбове. Линейна система която притежава такова изграждане се нарича сингулярна.

В допълнение, при числено решаване на система линейни уравнения трябва да се вземат предвид още две особености:

1. Възможно е някои от редовете в системата (1.1) да бъдат "близко" до линейна зависимост в границите на грешката от закръгляване, в който случай числената процедура може да **не** намери решение въпреки че то съществува.

2. Натрупаната грешка от закръгляване по време на численото решение може да промени истинското решение. Например в случаите когато N е много голямо число, тогава числената процедура няма да върне грешка от алгоритмична гледна точка, но ще получим резултат, който ще се различава от истинският извън рамките на изчислителната грешка, както може да се провери заместваики полученото решение x_j $j = 1, ..., N$ отново в уравнение (1.1).

Съвременните изчислителни алгоритми се стремят да избегнат или намалят ефекта на изброените по-горе две особености. В практиката съществува грубо правило в случай на числено решение на система линейни уравнения, което може да намали ефекта от натрупването на грешки от закръгляване, според което за системи с размер $N < 50$ е допустимо използването на аритметика с единична точност. За системи линейни уравнения с по-голям размер обаче използването на аритметика с двойна точност е препоръчително, тогава ограничаващият фактор ще бъде времето необходимо за пресмятане решението на линейната система.

1.1.1 Матрично представяне

Уравнението (1.1) може да бъде записано в съкратена матрична форма:

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{1.2}$$

където с "$\cdot$" ще означаваме операция по матрично умножение а символите $\mathbf{x}$ и $\mathbf{b}$ се представят като вектор стълбове:

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,N} \\ a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,N} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{M,1} & a_{M,2} & a_{M,3} & \cdots & a_{M,N} \end{bmatrix}; \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix}; \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_M \end{bmatrix} \tag{1.3}$$

по дефиниция първият индекс на елементите a_{ij} сочи номера на реда i, докато вторият индекс номера на стълба j, на който принадлежи елемента. В компютърната памет обикновенно матриците се представят като двумерни масиви, като начинът на подредба на елементите обаче зависи от програмният език. Например във *Fortran* елементите на двумерните масиви са последователно подредени по колони:

$$a_{11}, a_{21}, a_{31}, ..., a_{M1}; a_{12}, a_{22}, a_{32}, ..., a_{M2}; ...; a_{1N}, a_{2N}, a_{3N}, ..., a_{MN}$$

докато в $C/C++$ или *Pascal* по редове:

$$a_{11}, a_{12}, a_{13}, ..., a_{1N}; a_{21}, a_{22}, a_{23}, ..., a_{2N}; ...; a_{M1}, a_{M2}, a_{M3}, ..., a_{MN}$$

Тази техническа особеност трябва да се взима предвид, в случаите когато реализираме операции, в които желаем да прочетем или променим съдържанието на матрични елементи, които се намират последователно или близо един до друг в матрицата $\mathbf{A}$, ако желаем да подобрим производителнстта на нашата програма. Например при умножение на две матрици или матрица с вектор. Нека разгледаме примерен случай на умножение на матрицата $\mathbf{A}$ с вектора $\mathbf{x}$ реализирана на двата програмни езика C и $Fortran$.

В случай на езика $C/C++$ матрицата $\mathbf{A}$ може да бъде представена в компютърната памет като двумерен масив $a[M][N]$ с индекси $i = 0, ..., M-1$ и $j = 0, ..., N-1$, векторът $\mathbf{x}$ като едномерен масив $x[N]$ с N на брой елемента и индекси $j = 0, ..., N-1$, резултатът от умножението ще бъде записан в едномерният масив $b[M]$: $i = 0, ..., M-1$ който ще представлява вектора $\mathbf{b}$ в уравнението 1.2.

```
1  ...
2  double a[M][N];
3  double b[M], x[N];
4  int i, j;
5  ...
6  for (i = 0; i < M; i++) {
7     b[i] = 0.0;
8     for (j = 0; j < N; j++) {
9        b[i] += a[i][j]*x[j]
10    }
11 }
12 ...
```

Обърнете внимание че при акумулирането на резултатът в масива $b[M]$ обхождането на елементите става във външният цикъл по редове докато във вътрешният цикъл става по колони. Това се прави с цел да се повиши скоростта на пресмятане на сумите като се предполага че поради подредбата на елементите на масива $a[M][N]$ последователно по редове в паметта достъпът до тях ще бъде по-бърз и затова трябва тяхното обхождане да се реализира във вътрешният цикъл. Тази техника е позната като подобряване на ефективността на кеша на процесора - *cache efficient algorithms*.

Същото произведение на матрицата $\mathbf{A}$ с вектора $\mathbf{x}$ реализирано на програмният език *Fortran* обаче ще изглежда принципно различно. Ако желаем да се възползваме от подредбата на елементите на масива $a(M, N)$ с индекси $i = 1, ..., M$ и $j = 1, ..., N$ по колони. Програмният ни код трябва да изглежда по следният начин:

```
1  ...
2  double precision :: a(M,N)
3  double precision :: b(M), x(N)
4  integer i, j
5  ...
6  b(:) = 0.d0
7  do j = 1, N
8    do i = 1, M
9    b(i) = b(i) + a(i,j) * x(j)
10   end do
11 end do
12 ...
```

Важното което трябва да се отбележи в горният примерен код на *Fortran* е че индексите на работните цикли са разменени в сравнени с тези на $C/C++$ с цел да се отчита близката подредба на елементите $a(M, N)$ по колони. Добрата новина в случаят на програмният език *Fortran* е че горният израз може да бъде елегантно заменен с вътрешната функция за матрично умножение `matmul`

```
1 ...
2 double precision :: a(M,N)
3 double precision :: b(M), x(N)
4 integer i, j
5 ...
6 b(:) = matmul(a, x)
7 ...
```

Вградената функция `matmul(mat_a, mat_b)` връща резултатът от произведението на двете матрици `mat_a` и `mat_b` които се предполага че са от един и същи тип `INTEGER`, `REAL`, `LOGICAL` или `COMPLEX`. Рангът на двете матрици може да бъде от първи или втори ред, с условието последната размерност на матрицата `mat_a` да съвпада с първата разменост на матрицата `mat_b`. Използването на вградени функции и процедури за математични операции като `matmul` е силно препоръчителна поради следните две причини:

1. Водят до съществено опростяване на изходният код.
2. Вградените процедури могат да бъдет специално оптимизирани от компилатора с цел бързодействие.

Можем да завършим с това че съществуват множество от библиотеки с процедури които извършват алгебрични операции като умножение на две матрици, транспониране на матрица, умножение на матрица с вектор и много други. Една широко установена библиотека е **B**asic **L**inear **A**lgebra **S**ubroutines (BLAS). Добра практика е използването на такива външни библиотечни процедури пред потребителски написаните, по-този начин се избягват евентуалните алгоритмични грешки, които могат са се допуснат, гарантира се по-бързо изпълнение на програмният код и опо-ефективно използване на системната памет.

1.1.2 Цели и задачи на изчислителната линейна алгебра

От практическа гледна точка могат да се обобщят четири задачи свързани със система от линейни уравнения (1.2):

1. Намиране решение на уравнение (1.2) за неизвестен вектор $\mathbf{x}$ при известни $\mathbf{A}$ и $\mathbf{b}$.

2. Намиране на повече от едно решения на уравнение (1.2) а именно множеството $\mathbf{A} \cdot \mathbf{x}^j = \mathbf{b}^j$ за $j = 1, 2, ...$; където всяко решение $\mathbf{x}^j$ отговаря на различно $\mathbf{b}^j$, докато матрицата $\mathbf{A}$ се запазва.

3. Пресмятането обратната матрица $\mathbf{A}^{-1}$ на $\mathbf{A}$ така че да изпълнява условието $\mathbf{A}^{-1} \cdot \mathbf{A} = \mathbf{1} = \mathbf{A} \cdot \mathbf{A}^{-1}$, тази задача е еквивалентна на задача 2 в случай че се разглежда множеството от вектори $\mathbf{b}^j$ за $j = 1, ..., N$ където $\mathbf{b}^j$ е единичен вектор с ненулев елемент на ред j, матрицата получена от вектор стълбовете $\mathbf{x}^j$ определя обратната матрица $\mathbf{A}^{-1}$.

4. Определяне детерминантата на матрицата $\mathbf{A}$.

В случаите когато $M < N$ или $M = N$ но системата е изродена имаме по-малък брой уравнения от неизвестните и по същество е възможно да не съществува решение на системата (1.1) или да има повече от едно решения $\mathbf{x}$ за дадено $\mathbf{b}$. Когато $M > N$ системата е преопределена, т.е. **не** може да се намери решение $\mathbf{x}$ в най-общия случай. Възможно е обаче да се търси така нареченото "най-близко" решение. В по-следващото изложение ще смятаме че разглежданите от нас матрици са несингулярни и $M = N$.

1.1.3 Стандартни библиотеки за решаване на линейни уравнения. Програми за символни пресмятания.

Съществуват развити и добре документирани програмни пакети и библиотеки за решаване системи от линейни уравнения. Те са разработени предимно с цел да се справят с големи системи ($N >> 1$), имат добро управление на паметта, демонстрират по-добра производителност от потребителски написаните алгоритми и не на последно място практически са изчистени от грешки. Освен това всеки един числен алгоритъм

има развити няколко версии, които разглеждат специални случаи като: симетрични матрици, матрици с три-диагонална структура, блочни матрици, положително определени матрици и други. В примерите по-долу са разгледани процедури от две широко разпространени библиотеки за линейна алгебра **L**inear**A**lgebra**PACK**age (LAPACK) и **G**nu **S**cientific **L**ibrary (GSL). Може да се каже че LAPACK е стандарт в числените методи. Написана на *Fortran* в началото на 90-те години на миналият век, тази софтуерна библиотека се е наложила като бърз, надежден и добре документиран софтуерен пакет за линейна алгебра. LAPACK се използва в примерните задачи към настоящата глава с цел студентите да добият практически умения в използването ѝ. GSL е научна библиотека с отворен код, която е в процес на сериозно развитие, поради което не може да се очаква че тя ще притежава стабилността и бързината на LAPACK, но включва в себе си числени методи отвъд линейната алгебра, лицензът ѝ позволява да бъде използвана и развивана свободно.

Струва си да се отбележи, че програмните пакети за символни пресмятания като *Mathematica*, *Matlab*, *Maple* и други притежават също вградени функции за решавани на системи линейни уравнения.

1. $Mathematica \rightarrow x = LinearSolve[A, b]$.
2. $Matlab \rightarrow x = linsolve(A, b)$.
3. $Maple \rightarrow x = LinearSolve(A, b)$.

Въпреки улесненият потребителски интерфейс, програмите за символни пресмятания нямат същата производителност каквито биха имали потребителски написаните или библиотечни функции. Приемливо е да се решават системи със сравнително малък размер ($N < 100$).

1.2 Метод на Гаус-Жордан

Методът на Гаус-Жордан позволява намирането едновременно на обратната матрица $\mathbf{A}^{-1}$ както и решенията $\mathbf{x}^j$ за един или повече вектора $\mathbf{b}^j$. Като недостатък на този алгоритъм може да се изтъкне че всички елементи от дясната страна $\mathbf{b}^j$ трябва да се съхраняват и обработват едновременно по време на пресмятането. Освен това в случаите, когато нямаме нужда от пресмятането на обратната матрица $\mathbf{A}^{-1}$ този метод е три пъти по-бавен в сравнение с останалите алгоритми. Предимствата на този подход са че е числено стабилен, т. е. винаги се достига до решение ако такова съществува, ясно определен е алгоритмично, което го прави удобен за кодиране. В практиката се предпочита метода на LU декомпозиция вместо елиминация с Гаус-Жордан, като по-бърз. Все пак методът на Гаус-Жордан може да се използва като резервен вариант в случаите, когато линейната система е малка или искаме да проверим например за грешки друг алгоритъм.

За да онагледим алгоритъма нека разгледаме следната система от уравнения, която за простота ще използваме матрици с размер 4×4 елемента:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{44} & a_{44} \end{bmatrix} \cdot \left[\begin{pmatrix} x_{11} \\ x_{21} \\ x_{31} \\ x_{41} \end{pmatrix} \sqcup \begin{pmatrix} x_{12} \\ x_{22} \\ x_{32} \\ x_{42} \end{pmatrix} \sqcup \begin{pmatrix} x_{13} \\ x_{23} \\ x_{33} \\ x_{43} \end{pmatrix} \sqcup \begin{pmatrix} y_{11} & y_{12} & y_{13} & y_{14} \\ y_{21} & y_{22} & y_{23} & y_{24} \\ y_{31} & y_{32} & y_{33} & y_{34} \\ y_{41} & y_{42} & y_{44} & y_{44} \end{pmatrix} \right] = \\ = \left[\begin{pmatrix} b_{11} \\ b_{21} \\ b_{31} \\ b_{41} \end{pmatrix} \sqcup \begin{pmatrix} b_{12} \\ b_{22} \\ b_{32} \\ b_{42} \end{pmatrix} \sqcup \begin{pmatrix} b_{13} \\ b_{23} \\ b_{33} \\ b_{43} \end{pmatrix} \sqcup \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \right] \tag{1.4}$$

където знака "$\cdot$" означава матрично умножение а знакът "$\sqcup$" добавяне на операнд към произведението, в съкратен матричен вид уравнението (1.4) може да се запише като:

$$[\mathbf{A}] \cdot [\mathbf{x}^1 \sqcup \mathbf{x}^2 \sqcup \mathbf{x}^3 \sqcup Y] = [\mathbf{b}^1 \sqcup \mathbf{b}^2 \sqcup \mathbf{b}^3 \sqcup \mathbf{1}] \tag{1.5}$$

където $\mathbf{A}$ и $\mathbf{Y}$ са квадратни матрици, горното уравнение (1.5) може да се разбие на четири отделни уравнения:

$$\begin{aligned} \mathbf{A} \cdot \mathbf{x}^1 = \mathbf{b}^1; \mathbf{A} \cdot \mathbf{x}^2 = \mathbf{b}^2; \\ \mathbf{A} \cdot \mathbf{x}^3 = \mathbf{b}^3; \mathbf{A} \cdot \mathbf{Y} = \mathbf{1}; \end{aligned} \tag{1.6}$$

От теорията на линейната алгебра са ни известни следните няколко свойства на системи от линейни уравнения:

1. Размяната на кои да е два реда на матрицата $\mathbf{A}$ и съответните редове на вектор стълба $\mathbf{b}$ и единичната матрица $\mathbf{1}$, не променят или разбъркват решението $\mathbf{x}$, т.е. линейната система се представя в друг ред.

2. Подобно, решението $\mathbf{x}$ няма да се промени или елементите му няма да се разместят ако кой да е ред от матрицата $\mathbf{A}$ се замени с линейна комбинация включваща самият ред и/или други редове от матрицата, доколко същата операция се извършва и върху векторите $\mathbf{b}$ и единичната матрица $\mathbf{1}$.

3. Размяната на две колони ще ни даде същото решение само ако разменим съответните редове на вектора $\mathbf{x}$.

1.2.1 Смяна на диагоналният елемент - пивотинг

В класическият си вариант метода на Гаус-Жордан използва само свойство 2 с цел матрицата $\mathbf{A}$ да се доведе до единичната матрица чрез последователно елиминиране на елементите извън диагонала, например: започвайки от първият ред разделяме всички елементи на стойността на диагоналният елемент a_{11} след което елиминираме

извъндиагоналните елементи принадлежащи на първата колона на матрицата като умножаваме първи ред със стойността на i-тия елемент от колоната a_{i1} и изваждаме първи ред от i-ти ред последователно за всички $i = 2, ..., N$ по този начин първата колона ще се приведе до форма на колона от единичната матрица. Същата процедура можем да повторим за следващите колони докато не приведем цялата матрица $\mathbf{A}$ до единичната матрица $\mathbf{1}$. След приключване на диагонализацията векторите $\mathbf{b}^j$ ще съдържат търсеното решение $\mathbf{x}^j$, докато единичната матрицата $\mathbf{1}$ ще съдържа обратната матрица на $\mathbf{A}^{-1}$. Въпреки че изглежда изключително проста процедурата се разглежда като числено нестабилна.
Проблем може да възникне в случаите когато диагоналния елемент е равен или близък (по отношение на машинният епсилон) до нулата, затова се налага смяна на диагоналния/централния елемент. **Гаус - Жордан без смяна на диагоналния елемент е числено нестабилна процедура.** Методът има своите модификации:

а) при размяна само на редове с промяна на водещия диагонален елемент метода се грижи за частично насочване/пивот

б) в случай че се разменят както редове така и колони - пълно насочване/пивот

И двата случая начина на подбор на редове и колони трябва да е такъв, че да не нарушава вече съществуващата диагонална структура на матрицата $\mathbf{A}$, за тази цел се избират:

- редове, които все още не са или в моментна се превеждат в диагонален вид
- колони от дясната част на матрицата $\mathbf{A}$, които ще бъдат или са в процес на нулиране.

Един практически съвет за избора на нов диагонален елемент е по големината на абсолютната му стойност.

В допълнението А.1.1 са разгледани два примерни кода написани на програмните езици *C/C++* и *Fortran*. Те реализират процедура *gaussj* която използва алгоритъм с пълен пивотинг. Процедурата приема като входни/изходни параметри матрицата a с размер $N \times N$ и множество от вектори $\mathbf{b}^j$ представени като матрица b с размер $N \times M$, където M е броят известни вектори от дясната страна на уравнението 1.5. Изходният код на двете процедури с известни промени е взаимстван от книгата *Числени рецепти* [8].

1.3 Метод на LU декомпозиция

Нека предположим, че матрицата $\mathbf{A}$ може да се представи като произведение на две матрици:

$$\mathbf{L} \cdot \mathbf{U} = \mathbf{A} \tag{1.7}$$

където $\mathbf{L}$ е ниско/долно триъгълна матрица (т.е притежава не нулеви елементи само под главния си диагонал влкючително) и $\mathbf{U}$ е горно диагонална матрица. В случая на матрици 4×4 израза 1.7 може да се представи в компонентен вид като:

$$\begin{bmatrix} \alpha_{11} & 0 & 0 & 0 \\ \alpha_{21} & \alpha_{22} & 0 & 0 \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & 0 \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & \alpha_{44} \end{bmatrix} \cdot \begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ 0 & \beta_{22} & \beta_{23} & \beta_{24} \\ 0 & 0 & \beta_{33} & \beta_{34} \\ 0 & 0 & 0 & \beta_{44} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \tag{1.8}$$

решението на линейното уравнение 1.2 в този случай може да се изрази като

$$\mathbf{A} \cdot \mathbf{x} = (\mathbf{L} \cdot \mathbf{U}) \cdot \mathbf{x} = \mathbf{L} \cdot (\mathbf{U} \cdot \mathbf{x}) \tag{1.9}$$

ако намерим междинното решение за вектора $\mathbf{y}$ така че:

$$\mathbf{L} \cdot \mathbf{y} = \mathbf{b} \tag{1.10}$$

след което ще можем да намерим търсеното от нас $\mathbf{x}$:

$$\mathbf{U} \cdot \mathbf{x} = \mathbf{y} \tag{1.11}$$

Предимството на този подход е че в случая когато $\mathbf{L}$ и $\mathbf{U}$ са три диагонални известни матрици. Решенията $\mathbf{x}$ и $\mathbf{y}$ могат да се намерят лесно чрез следните итеративни уравнения:

$$y_1 = \frac{b_1}{\alpha_{11}}; \quad y_i = \frac{1}{\alpha_{ii}} \left[b_i - \sum_{j=1}^{i-1} \alpha_{ij} y_j \right], \quad i = 2, 3, \ldots, N \tag{1.12}$$

по подобен начин за $\{x_i\}$

$$x_N = \frac{y_N}{\beta_{NN}}; \quad x_i = \frac{1}{\beta_{ii}} \left[y_i - \sum_{j=i+1}^{N} \beta_{ij} x_j \right], \quad i = N-1, N-1, \ldots, 1 \tag{1.13}$$

Итеративното решение има две основни предимства. Първо то много лесно може да се кодира и второ намалява изчислителната сложност на алгоритъма, т.е. общият брой математически операции необходим за определянето на решението $\{x_j\}$ е от порядъка на размера на матрицата $\sim \frac{1}{3}N^3$ за разлика от метода описан в предишната Глава 1.2, където сложността на метода е от порядък $\sim N^3$. Сега предстои предизвикателството да намерим самите матрици $\mathbf{L}$ и $\mathbf{U}$.

1.3.1 Как да намерим матриците L и U?

В практиката има множество методи. В конкретния пример ще разгледаме най-широко разпространения метод за плътни матрици.

1.3.1.1 Метод на Крут

Нека да разгледаме в детайли i, j - тия елемент на матрицата $\mathbf{A}$ в уравненията 1.7 и 1.8, a_{ij}:

$$\alpha_{i1}\beta_{1j} + \alpha_{i2}\beta_{2j} + \cdots + \alpha_{iN}\beta_{Nj} = a_{ij} \tag{1.14}$$

Броя на членовете в горната сума 1.14 зависи от индексите i и j, поради тридиагоналната структура на матриците $\mathbf{L}$ и $\mathbf{U}$ по същество могат да се разглеждат три случая:

$$\begin{aligned} i < j: &\ \alpha_{i1}\beta_{1j} + \cdots + \alpha_{ii}\beta_{ij} = a_{ij} \\ i = j: &\ \alpha_{i1}\beta_{1j} + \cdots + \alpha_{ii}\beta_{jj} = a_{ij} \\ i > j: &\ \alpha_{i1}\beta_{1j} + \cdots + \alpha_{ij}\beta_{jj} = a_{ij} \end{aligned} \tag{1.15}$$

доколко броят ненулеви елементи на една горно или долно диагонална квадратна матрица $N \times N$ е равен на $\dfrac{N(N+1)}{2}$, тогава общият брой неизвестни елементи $\{\alpha_{ij}\}$ и $\{\beta_{ij}\}$ е $N^2 + N$, общият брой уравнения обаче в 1.15 са точно N^2 затова свободно можем да изберем елементите $\alpha_{ii} \equiv 1$ за $i = 1, 2, \ldots N$.

Решението на системата уравнения 1.15 може да стане лесно, просто чрез пренареждане на елементите в дадена последователност. По същество алгоритъма се състои в следните стъпки:

I) Нека $\alpha_{ii} = 1$ за $i = 1, \ldots, N$

II) за всяко $j = 1, \ldots, N$ се повтарят следните две стъпки:

- Първо за $i = 1, \ldots, j$ използвайки системата 1.15 решаваме спрямо β_{ij}

$$\beta_{ij} = a_{ij} - \sum_{k=1}^{i-1} \alpha_{ik}\beta_{kj} \tag{1.16}$$

- Второ - за $i = j+1, j+2, \ldots, N$ отново използвайки 1.15 можем да намерим α_{ij}

$$\alpha_{ij} = \frac{1}{\beta_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} \alpha_{ik}\beta_{kj} \right) \tag{1.17}$$

Обърнете внимание че в горе описаният алгоритъм матричните елементи принадлежащи и на трите матрици $\mathbf{L}$, $\mathbf{U}$ и $\mathbf{A}$ се ползват последователно един след друг, т.е. α-те и β-те които се появяват в дясната част на уравненията 1.16 и 1.17 са вече известни. Още повече всеки елемент от a_{ij} се използва само веднъж при пресмятането на матричните коефициенти, следователно можем да използваме едно и също място

в което да съхраняваме коефициентите α, β и а по същество методът на Крут запазва коефициентите в една комбинирана матрица:

$$\begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ \alpha_{21} & \beta_{22} & \beta_{23} & \beta_{24} \\ \alpha_{31} & \alpha_{32} & \beta_{33} & \beta_{34} \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & \beta_{44} \end{bmatrix} \tag{1.18}$$

От гледна точка на числена стабилност методът отново използва пивотинг поради деленето на диагоналният елемент β в израза 1.17, в случаят е приложим само частичен пивотинг т.е. възможно е само размяна на редове матрицата $\mathbf{A}$ по същество ние разделяме (декомпозираме) не матрицата $\mathbf{A}$ а нейно пермутирано копие по редове, което както обсъдихме вече не променя решението на линейната система. Ако разгледаме уравненията 1.16 и 1.17 в случаите когато $i = j$ можем да установим че двата израза са индентични и факторът β_{jj} на който делим се пресмята от коефициенти α и a които вече са определени, т.е. техните индекси на реда са по-малки от i следователно пермутацията на кои да е два реда с индекси по-големи от i няма да повлияе на вече пресметнатите α и β.

1.3.2 LU декомпозиция. Числени реализации

1. Изходният код на C - версията на процедурата `ludcmp(a, n, indx, d)` взаимствана от книгата *Числени рецепти* може да бъде разгледан в допълнение А.1.2 процедурата приема като вход квадратната матрица $\mathbf{A}$ записана в масива `a` с размер $n \times n$ елемента. Като резултат се връща комбинираната матрица 1.18 записана в масива `a` в целочисленият масив `indx` с размер `n` елемента са записани пермутациите на редовете на матрицата `a` и накрая променливата `d` $= \pm 1$ е изходен и в нея се записва четността на пермутациите.

2. Подобно в приложение А.1.2 е показан изходният код на версията на *Fortran* на същата процедура `ludcmp(a, n, indx, d)`

3. Математическата библиотека LAPACK също има реализирана LU декомпозиция. Процедурата `DGETRF( M, N, A, LDA, IPIV, INFO )` факторизира матрицата $\mathbf{A}$, като $\mathbf{A} = \mathbf{P} \times \mathbf{L} \times \mathbf{U}$:

   `M` - (входен) целочислен параметър равен на броя на редовете на матрицата A. $M \geq 0$

   `N` - (входен) целочислен параметър равен на броя на колоните на матрицата A. $N \geq 0$

   `A` - (входно/изходен) масив с двойна точност с разменост `(LDA, N)`

   `LDA` - (входен) целочислен параметър, работна размерност матрицата A. $LDA \geq max(1, M)$

IPIV - (изходен) целочислен масив с размер min(M, N) в който се запазват пермутациите на матрицата A. Например реда в матрицата с индекс i е бил заменен с ред IPIV(i).

INFO - (изходен) целочислена променлива в която се записва код на приключването на процедурата

Горната процедура може да се използва в комбинация с друга процедура DGETRS за да се намери решения на една от следните задачи от линейната алгебра: $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ или $\mathbf{A}^T \cdot \mathbf{x} = \mathbf{b}$. В общият случай B може да бъде матрица от вектори с размерност (LDA, NRHS). Параметрите на процедурата DGETRS(TRANS, N , NRHS, A, LDA, IPIV, B, LDB, INFO) са както следва

TRANS (входен) - символна променлива, която указва формата на матрицата A:

'N' - нормална линейна система, $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$

'T' - транспонирана линейна система, $\mathbf{A}^T \cdot \mathbf{x} = \mathbf{b}$

'C' - комплексно линейна спрегната, $\overline{\mathbf{A}}^{\mathrm{T}} \cdot \mathbf{x} = \mathbf{b}$

N - (входен) целочислен параметър указващ размера на матрицата A

NRHS - (входен) целочислен параметър указващ броя на стълбовете на матрицата B

A - (входно/изходен) масив с двойна точност с разменост (LDA, N)

LDA - (входен) целочислен параметър, работна размерност матрицата A. $LDA \geq max(1, N)$

IPIV - (изходен) целочислен масив с размер min(M, N) в който се запазват пермутациите на матрицата A. Например реда в матрицата с индекс i е бил заменен с ред IPIV(i).

B - (входно/изходен) масив с двойна точност с разменост (LDB, NRHS)

LDB - (входен) целочислен параметър, работна размерност матрицата B. $LDA \geq max(1, N)$

INFO - (изходен) целочислена променлива в която се записва код на приключването на процедурата

Пример: Ще разгледаме примерен код на прогрманият език *Fortran 90*, който решава следната линейна система $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$:

$$\begin{bmatrix} 1 & -2 & 3 \\ 2 & 1 & 1 \\ -3 & 2 & -2 \end{bmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 7 \\ 4 \\ -10 \end{pmatrix}$$

с очаквано решение $\mathbf{x} = (2, -1, 1)$. Примерният код използва масиви с динамично заделяне на паметта като променливата n е параметър указващ размера на матрицата a

Програмен код 1.1: Примерна програма на *Fortran 90* която решава линейна система от трети ред

```
1  program lu_lapack
2  implicit none
3
4  integer :: i, n, info, nrhs = 1
5  double precision, dimension(:,:), allocatable :: a
6  double precision, dimension(:), allocatable :: b
7  integer, dimension(:), allocatable :: ipiv
8
9  n = 3
10
11 allocate(a(n, n))
12 allocate(b(n))
13 allocate(ipiv(n))
14
15 a = reshape((/ 1., 2., -3., -2., 1., 2., 3., 1., -2. /), shape(a))
16
17 b = reshape((/7., 4., -10./), shape(b))
18
19 call DGETRF( n, n, a, n, ipiv, info )
20
21 call DGETRS('N', n, nrhs, a, n, ipiv, b, n, info)
22
23 do i = 1, n
24   write(0,'(A2,I1,A4,F10.2)')'x(',i,')␣=␣',b(i)
25 end do
26
27 deallocate(a)
28 deallocate(b)
29 deallocate(ipiv)
30
31 end program lu_lapack
```

инструкции за компилирине и изпъление на програмата. Описаният по-горе примерен код използва външни процедури DGETRF и DGETRS от библиотеката LAPACK. За да може да се компилира и свърже изпълнимият файл с библиотеката е необходимо да се укаже допълнителне параметър -llapack в командата за компилиране :

```
1  gfortran lu_lapack.f90 -llapack -o lu_lapack.x
2  ./lu_lapack.x
```

Задача: Модифицирайте примерният код на linsolve.f90 така че да направите проверка на полученото решение, т.е. пресметнете израза $\mathbf{A} \cdot \mathbf{x}$ и проверете дали резултатът съвпада с $\mathbf{b} = (7, 5, -10)$. Можете да използвате вградената функция за матрично умножение във Фортран matmul.

Задача: Добавете функционалност в примерният код linsolve.f90 потребителят да въвежда като входен параметър размерът на матрицата n и последователно елеметите на матрицата $\mathbf{A}$ и вектора $\mathbf{b}$.

1.4 Тридиагонални линейни системи.

Един често срещан случай на линейни системи представляват тридиагоналните линейни системи. Те могат да се разглеждат като специален случай на разредени системи линейни уравнения за които повечето матрични еленти са равни на нула. Изключение правят само диагоналните елементи плюс и/или минус една допълнителна колона около главният диагонал. LU декомпозицията за тридиагонални системи и тяхното решение (намирането на вектора $\mathbf{b}$ отнема $\sim N$ брой операции. Поради своята опростена структура решението на подобна система може да бъде кодирано по много лесен и елегантен начин. Освен това с цел икономия на компютърни ресурси, тридиагоналните матрици се запазват в компютърната памет не като $N \times N$ брой елемента а се пазят само елементите от главният диагонал и неговите съседни елементи, т.е. $3 \times N$ брой елемента. Матричният запис на една тридиагонална система може да се запише като:

$$\begin{bmatrix} b_1 & c_1 & 0 & \cdots & 0 & 0 & 0 \\ a_2 & b_2 & c_2 & \cdots & 0 & 0 & 0 \\ 0 & a_3 & b_3 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & b_{N-2} & c_{N-2} & 0 \\ 0 & 0 & 0 & \cdots & a_{N-1} & b_{N-1} & c_{N-1} \\ 0 & 0 & 0 & \cdots & 0 & a_N & b_N \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{N-2} \\ x_{N-1} \\ x_N \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_{N-2} \\ d_{N-1} \\ d_N \end{bmatrix} \tag{1.19}$$

добре е да се отбележи че доколко коефициентите a_1 и c_N не са определени и следователно не се използват при намиране на решението $\mathbf{x}$ въпреки че участват във входният масив. Методът за решаване на тридиагоналната система уравнение включва две стъпки. Първоначално подменяме коефициентите c_i извън главният диагонал с помощта на следните итерации:

$$c'_i = \begin{cases} c_i/b_i & ; \quad i = 1 \\ c_i/\left(b_i - a_i c'_{i-1}\right) & ; \quad i = 2, 3, \cdots, n-1 \end{cases} \tag{1.20}$$

и елементите от главният диагонал d_i:

$$d'_i = \begin{cases} d_i/b_i & ; \quad i = 1 \\ \left(d_i - a_i d'_{i-1}\right) / \left(b_i - a_i c'_{i-1}\right) & ; \quad i = 2, 3, \cdots, n \end{cases} \tag{1.21}$$

Следващата стъпка включва намиране на решението x_i:

$$x_i = \begin{cases} d'_i & ; \quad i = n \\ d'_i - c'_i x_{i+1} & ; \quad i = n-1, n-2, \cdots, 1 \end{cases} \tag{1.22}$$

Поради простотата на алгоритъма и честата му употреба по-нататък в ръководството по-долу ще разгледаме две реализации на програмните езици *C* и *Fortran*. Библиотеката LAPACK има две процедури които решават тридиагонални системи

в случаите когато тридиагоналната матрица има най-общ вид може да се използва SUBROUTINE DGTSV(N, NRHS, DL, D, DU, B, LDB, INFO), където

- N - (входен) целочислен параметър равен на размера на квадратната матрица A. $N \geq 0$
- NRHS - (входен) целочислен параметър равен на броя на колоните на матрицата B. $NRHS = \max(1, N)$
- DL - (входно/изходен) едномерен масив от тип реални числа с двойна точност с размер $N - 1$ брой елемента. При началното извикване трябва да съдържа елементите на горният диагонал. След приключване изпълнението на процедурата съдържа елементите на горният диагонал на U матрицата.
- D - (входно/изходен) едномерен масив от тип реални числа с двойна точност с размер N брой елемента. При началното извикване трябва да съдържа елементите на главният диагонал. След приключване изпълнението на процедурата съдържа главните елементи на U матрицата.
- DU - (входно/изходен) едномерен масив от тип реални числа с двойна точност с размер $N - 1$ брой елемента. При началното извикване трябва да съдържа елементите на долният диагонал. След приключване изпълнението на процедурата съдържа елементите на горният диагонал на L матрицата.
- B - (входно/изходен) масив от реални числа с двойна точност с размерност (LDB , NRHS). При началното извикване съдържа матрицата от вектор стълбовете на дясната част на линейната система. След приключване на извикването ако INFO = 0 съдържа матрица от вектор стълбовете на решението x.
- LDB - (входен) целочислен параметър, работна размерност матрицата B. LDB >= max(1,N)
- INFO - (изходен) целочислена променлива в която се записва код на приключването на процедурата:

  N = 0: успешно приключване

  N < 0: ако N = -i тогава i-тия елемент има неправилна числена стойност (Inf, NaN)

  N > 0: тогава при факторизацията i-тия елемент U(i,i) е равен на нула

В случай на симетрични матрици можем да използваме специално оптимизираната процедура SUBROUTINE DPTSV(N, NRHS, D, E, B, LDB, INFO), където

- N - (входен) целочислен параметър равен на размера на квадратната матрица A, N >= 0

- NRHS - (входен) целочислен параметър равен на броя на колоните на матрицата B. NRHS = max(1, N)
- D - (входно/изходен) едномерен масив от тип реални числа с двойна точност с размер N брой елемента. При началното извикване трябва да съдържа елементите на главният диагонал. След приключване изпълнението на процедурата съдържа главните елементи на U матрицата.
- E - (входно/изходен) едномерен масив от тип реални числа с двойна точност с размер N - 1 брой елемента. При началното извикване трябва да съдържа съдесните елементи на главният диагонал.
- B - (входно/изходен) масив от реални числа с двойна точност с размерност (LDB , NRHS). При началното извикване съдържа матрицата от вектор стълбовете на дясната част на линейната система. След приключване на извикването ако INFO = 0 съдържа матрица от вектор стълбовете на решението x.
- LDB - (входен) целочислен параметър, работна размерност матрицата B. LDB >= max(1,N)
- INFO - (изходен) целочислена променлива в която се записва код на приключването на процедурата:

  N = 0: успешно приключване

  N < 0: ако N = -i тогава i-тия елемент има неправилна числена стойност (Inf, NaN)

  N > 0: тогава при факторизацията i-тия елемент U(i,i) е равен на нула

1.5 Метод на Якоби

Методът на Якоби може да се използва за решаване на системи линейни уравнения в случаите когато в матрицата **A** доминират диагоналните елементи. Алгоритъма е итеративен и включва приближеното решение от предишната итерация за подобряване новото решение. В основата на метода лежи идеята че матрицата **A** може да декомпозира като сума от строго диагонална матрица **D**, и остатък **R**:

$$\mathbf{A} = \mathbf{D} + \mathbf{R}$$

където диагоналната матрица се записва като:

$$\mathbf{D} = \begin{bmatrix} a_{11} & 0 & \dots & 0 \\ 0 & a_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & a_{NN} \end{bmatrix}$$

остатъчната матрица $\mathbf{R}$ се определя чрез:

$$\mathbf{R} = \begin{bmatrix} 0 & a_{12} & \dots & a_{1N} \\ a_{21} & 0 & \dots & a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \dots & 0 \end{bmatrix}$$

Формално решението на линейната система може да се запише като

$$\mathbf{A} \cdot \mathbf{x} = (\mathbf{D} + \mathbf{R}) \cdot \mathbf{x} = \mathbf{D} \cdot \mathbf{x} + \mathbf{R} \cdot \mathbf{x} = \mathbf{b} \tag{1.23}$$

$$\mathbf{x} = \mathbf{D}^{-1} \cdot (\mathbf{b} - \mathbf{R} \cdot \mathbf{x}) \tag{1.24}$$

В уравнение 1.24 можем да намерим решението на неизвестният вектор $\mathbf{x}$ като разликата между векторите $\mathbf{D}^{-1} \cdot \mathbf{b}$ и $\mathbf{D}^{-1} \cdot \mathbf{R} \cdot \mathbf{x}$. Изразът за $\mathbf{x}$ е неявен. Ако предположим обаче че модула на вектора$|\mathbf{D}^{-1} \cdot \mathbf{R} \cdot \mathbf{x}|$ е достатъчно малко число. Можем да разглеждаме вторият член в израза 1.24 като "поправка" която подобравя "текущото" решение на $\mathbf{x}$. В този случай решението на линейната система $\mathbf{x}$ се намира итеративно с помощта на следният израз:

$$\mathbf{x}^{k+1} = \mathbf{D}^{-1} \cdot \left(\mathbf{b} - \mathbf{R} \cdot \mathbf{x}^{k}\right). \tag{1.25}$$

където $\mathbf{x}^k$ е решение на уравнението 1.2 след k-тата итерация, горното уравнение 1.25 може да се запише в компонентен вид като:

$$x_i^{k+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^k \right)$$

за $i = 1, 2, ..., N$, пресмятането на x_i^{k+1} изисква познаването на всички компоненти на вектора $\mathbf{x}^k$ с изключение на i-тия елемент. За разлика от метода на Гаус-Жордан описан в раздел 1.2, сега не можем да презапишем x_i^k с x_i^{k+1} доколко този елемент ни е необходим и последващите пресмятания.

На схематично програмно ниво алгоритъма може да се обобщи в следните няколко стъпки:

Data: Избираме начално решение $\mathbf{x}^0 = \mathbf{b}$
Data: k = 0
repeat
  for *i = 1 ... N* **do**
    $\sigma = 0$ **for** *j = 1 ... N* **do**
      if $i \neq j$ **then**
        $\sigma = \sigma + a_{ij}x_j^k$
      end
    end
    $x_i^{k+1} = (b_i - \sigma)\,/a_{ii}$
  end
  k = k + 1
until $|x^{k+1} - x^k| < \epsilon$;

Алгоритъмът ще бъде сходящ когато спектралният радиус на итериращата матрица е по-малък от 1, т.е.

$$\rho\left(D^{-1} \cdot \mathbf{R}\right) = \max\left\{|\,\lambda_1|, |\lambda_2|, \ldots, |\lambda_N|\right\} < 1.$$

където $|\lambda_i|$ е i-тата собствена стойност на матрицата $D^{-1} \cdot \mathbf{R}$. Методът гарантирано ще бъде сходящ в случаите когато матрицата $\mathbf{A}$ е стриктно диагонално доминираща. Например за една стриктно диагонална матрица по редове, абсолютната стойност на i-тия диагоналният елемент трябва да бъде по-голям от сумата на абсолютните стойности на останалите елементи от даден ред:

$$|a_{ii}| \geq \sum_{j \neq i} |a_{ij}| \tag{1.26}$$

Понякога е възможно методът на Якоби да е сходящ дори и ако горното условие 1.26 не е в сила. Методът може да се използва изключително успешно при решаването на частни диференциални уравнения, по-долу в книгата ще бъде разгледната задачата за решаване на едномерното уравнението на Поасон използвайки именно този числен метод.

1.6 Примерна задача

Нека разгледаме задачата за решаването на три-диагонална симетрична линейна система от четвърти ред която в матричен вид се записва като:

$$A \tag{1.27}$$

Глава 2

Числено диференциране

В настоящата глава ще разгледаме техники за пресмятане производните на функция на една $f = f(x)$ или много променливи $f = f(\mathbf{x})$. Въпреки сравнително ограниченото директно използване на числено диференциране. Диференчните схеми разгледани в тази глава ще ни бъдат изключително полезни когато ще се запознаем с техниките за решаване на системи обикновенни диференциални уравнения, частни диферемциални уравнения както и при интерполацията и екстраполация на функции. Диференчните формули които ще получим по-долу използват разлагане в ред на Тейлър и могат да бъдат намерени в книгата на М. Абрамович и И. Щегун *Наръчник по математични функции* [1], Глава 25.1. В практиката съществуват алгоритми които позволяват намирането на произволна диференчна схема, например като тези описани в работата на Форнберг [4] които няма да бъдат разглеждани тук.

2.1 Диференциране на функция на една променлива

Като начало нека разгледаме функция на една променлива $f = f(x)$ като приемем че можем да пресметнем стойността и в решетка от точки $\{x_n\}$ където $n \in \mathbb{Z}$, точките са равно отдалечени една от друга на разстояние h което ще наричаме стъпка на дискретизация, където $h = x_{n+1} - x_n$ за произволно n. Можем да дефинираме множеството от функционални стойности $\{f_n\}$ върху решетката от точки $\{x_n\}$ на функцията $f = f(x)$

$$f_n = f\left(x_n\right),\ n \in \mathbb{Z} \tag{2.1}$$

нашата задача е да намерим израз с който да пресметнем първата производна на функцията f в произволна точка x_n. Нека за целта да разгледаме функцията $f = f(x)$ представена в ред на Тейлър в околността на свободно избрана точка x_n

$$f(x - x_n) = f(x_n) + \frac{(x - x_n)}{1!} f'(x_n) + \frac{(x - x_n)^2}{2!} f''(x_n) + \frac{(x - x_n)^3}{3!} f'''(x_n) + \dots \tag{2.2}$$

ако заместим променливата x в горният израз 2.2 за функцията $f(x)$ в точките $x_{n\pm1}$ и $x_{n\pm2}$, ще получим следната система от четири уравнения

$$f(x_{n\pm1}) = f(x_n) \pm \frac{h}{1!}f'(x_n) + \frac{h^2}{2!}f''(x_n) \pm \frac{h^3}{3!}f'''(x_n) + O(h^4) \quad (2.3)$$

$$f(x_{n\pm2}) = f(x_n) \pm \frac{2h}{1!}f'(x_n) + \frac{4h^2}{2!}f''(x_n) \pm \frac{8h^3}{3!}f'''(x_n) + O(h^4) \quad (2.4)$$

в случаите когато функцията f е достатъчно гладка и абсолютните стойности на нейните производни f', f'' и f''' в точките x_n са от един и същ порядък можем да приемем че грешката при пресмятането на функциите 2.3 и 2.4 е от ред $O(h^4)$. За да получим израза за първата производна на функцията f в точката x_n нека разгледаме разликата между $f(x_{n+1})$ и $f(x_{n-1})$, тогава

$$f'(x_n) = f'_n = \frac{f_{n+1} - f_{n-1}}{2h} - \frac{h^2}{6}f'''_n + O(h^4) \quad (2.5)$$

горното уравнение дефинира три-точкова диференчна схема за намиране първата производна на функцията f за произволна точка x_n с точност $O(h^2)$. Използвайки само едно от уравненията 2.3 можем да получим не-симетрична дву точкова диференчна формула от по-нисък ред за лявата и дясната първа производна на функцията f

$$f'_n = \frac{f_{n+1} - f_n}{h} + O(h) \quad (2.6)$$

$$f'_n = \frac{f_n - f_{n-1}}{h} + O(h) \quad (2.7)$$

Възможно е да се подобри точността на пресмятането на първата производна на функция ако се включат още "съседни" точки в околността на x_n. Например нека разгледаме 5-точкова диференчна схема с грешка от порядъка на $O(h^4)$

$$f'_n = \frac{1}{12h}\left[f_{n-2} - 8f_{n-1} + 8f_{n+1} - f_{n+2}\right] \quad (2.8)$$

Горната формула предполага че функцията $f = f(x)$ може да се апроксимира достатъчно добре с полином от четвърта степен в интервала $[x_{n-2}, x_{n+2}]$ и въпреки по-високата цена от гледна точка на брой пресмятания на функцията f повишава точността на пресмятането. Тук е мястото да отбележим че пресмятането на производната на функция е числено "нестабилна" процедура. В смисъл че с намаляване стъпката на дискретизация h грешката от пресмятането на производната на функцията намалява систематично и след определена стъпка започва да расте отново. Този числен ефект зависи основно от точността на представяне на числата с плаваща запетая която използваме. За да демонстрираме числената нестабилност нека разгледаме следният примерен код реализиран на програмният език *Fortran 90*, който използва две, три и пет точкови диференчни схеми и различна големина на стъпката на дискретизация h, като за пробна функция е избрана $\sin x$:

Програмен код 2.1: Примерна програма на *Fortran 90* която пресмята първата производна на функцията $f(x) = sin(x)$ в точката $x = 1.0$

```
1  program firstderiv
2  implicit none
3
4  real :: x, h, f0, fp1, fp2, fm1, fm2, fprime
5  real :: fprime_two_points, fprime_three_points, fprime_five_points
6  integer :: i
7
8  x = 1.e0
9  h = 0.1e0
10
11 do i = 1, 10
12   f0 = sin(x)
13   fp1 = sin(x+h)
14   fm1 = sin(x-h)
15   fp2 = sin(x+2*h)
16   fm2 = sin(x-2*h)
17   fprime = cos(x)
18
19   fprime_two_points = (fp1 - f0) / h
20   fprime_three_points = (fp1 - fm1) / (2 * h)
21   fprime_five_points = (fm2 - 8*fm1 + 8*fp1 - fp2) / (12 *h)
22
23   write(*,'(4F15.8)') h, abs(fprime_two_points-fprime), abs(
         fprime_three_points-fprime), abs(fprime_five_points-fprime)
24   h = h / 10
25 end do
26
27 end program firstderiv
```

Резултатите от пресмятането на първата производна на функцията $f(x) = \sin x$ за точката $x = 1.0$ могат да се обобщят в таблица 2.1

h	уравнение 2.6	уравнение 2.5	уравнение 2.8
0.10000000	0.04293787	0.00089991	0.00000167
0.01000000	0.00421214	0.00001001	0.00000107
0.00100000	0.00034374	0.00001389	0.00001389
0.00010000	0.00031191	0.00001389	0.00001389
0.00001000	0.00210005	0.00088018	0.00137687
0.00000100	0.00386041	0.00386041	0.00386041
0.00000010	0.65179074	0.35376751	0.40343809
0.00000001	0.54030228	0.54030228	0.54030228

Таблица 2.1: Грешка от пресмятането на първата производна за различни диференчни схеми и големина на стъпката на дискретизация h

вижда се че грешката започва да расте за стъпка на дискретизация в интервала от $h = 0.01$ до $h = 0.00001$ за аритметика с единична точност. Можем да отбележим също

така въпреки факта че грешката при пресмятане на производната да расте за по-големи стойност на h при диференчените схеми с повече точки абсолютната стойност на грешката се запазва в по-добри граници. Следователно в случаите когато можем да си позволим от изчислителна гледна точка е по-добре да използваме диференчни схеми с повече "възли" и относително по-голяма стъпка на дискретизация в сравнение с използването на схеми от по-нисък ред и малка стъпка на дискретизация h.

Упражнение:

Компилириайте и изпълнете програмният код 2.1 показан по-горе. Променете типа на представяне на числата с плаваща запетая от представане с единична точност `real` на представяне с двойна точност `real * 8` (или `double precision`). Сравнете получените резултати за грешката.

Подобно на уравнение 2.5 можем да намерим изрази за производни от по-висок ред. Например ако вземем сумата между f_{n+1} и f_{n-1} можем да "изключим" всички членове с нечетни степени на h и получим формула за втората производна f''

$$f''_n = \frac{f_{n+1} - 2f_n + f_{n-1}}{h^2} + O(h^2) \tag{2.9}$$

В практиката съществуват множество диференчни схеми които включват различен брой "съседни" точки които могат да се изпозлват съобразно нашите цели и изисквания. На таблицата по-долу са представени по-известните диферечни схеми които включват четири или пет съседни точки

	четири точкова схема	пет точкова схема
hf'_n	$\pm\frac{1}{6}\left(-2f_{n\mp1} - 3f_n + 6f_{n\mp1} - f_{n\pm2}\right)$	$\frac{1}{12}\left(f_{n-2} - 8f_{n-1} + 8f_{n+1} - f_{n+2}\right)$
$h^2f''_n$	$f_{n-1} - 2f_n + f_{n+1}$	$\frac{1}{12}\left(-f_{n-2} + 16f_{n-1} - 30f_n + 16f_{n+1} - f_{n+2}\right)$
$h^3f'''_n$	$\pm\left(-f_{n\mp1} + 3f_n - 3f_{n\pm1} + f_{n\pm2}\right)$	$\frac{1}{2}\left(-f_{n-2} + 2f_{n-1} - 2f_{n+1} + f_{n+2}\right)$
$h^4f^{(4)}_n$		$f_{n-2} - 4f_{n-1} + 6f_n - 4f_{n+1} + f_{n+2}$

2.2 Диференциране на функция на много променливи

Доколко диференчните схеми за функция $f = f(\mathbf{x})$ на многомерна променлива $\mathbf{x}$ може да бъде произволно сложна. Ние ще се спрем на случаите които представялват най-голям практически интерес за нас, а именно ще разгледаме функции на две и три промелнливи $f = f(x, y)$ и $f = f(x, y, z)$. Отново можем да използваме същият подход както в едномерният случай само че сега разглеждаме функциите дискретизирани върху дву- и три-мерни решетки. Нека последователно разгледаме и двете схеми

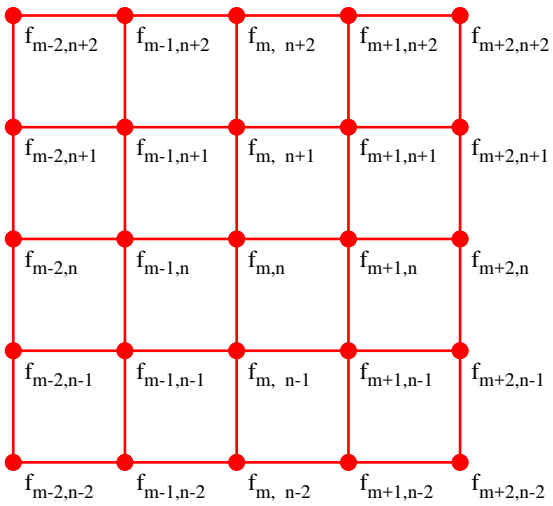
2.2.1 Двумерен случай

Нека разгледаме функцията $f = f(x, y)$ която се дискретизира върху двумерна решетка от точки които се описват чрез множеството дискретни стойности на независимите

променлива $\{x_m, y_n\}$ за $m, n \in \mathbb{Z}$. Да обърнем внимание че в общият случай стъпката на дискретизация е различна в двете направления, т.е. $h_x = x_{m+1} - x_m$ и $h_y = y_{n+1} - y_n$. Също така можем да дефинираме и множеството от функциални стойности f_{mn} върху двумерната дискретна решетка от стойности $\{x_m, y_n\}$.

$$f_{mn} = f(x_m, y_n) \text{ за всички } m, n \in \mathbb{Z} \tag{2.10}$$

за да онагледим идеята нека разгледаме функцията 2.10 в решетка от точки в интервала $[x_{m-2}, x_{m+2}] \times [y_{n-2}, y_{n+2}]$ което прави общо 25 точки на дискретизация които са илюстрирани на фигурата 2.1 по-долу:



Фигура 2.1: Дискретизационна схема на функция на две променливи $f = f(x, y)$

По-нататък нека разгледаме функцията $f = f(x, y)$ представена в ред на Тейлър за произволно избрана двойка x_m, y_n. С цел опростяване на израза ще се ограничим само до втора степен в разложението което ще ни бъде достатъчно за да илюстрираме начина за намиране на дискретни производни

$$\begin{aligned} f(x - x_m, y - y_n) =& f(x_m, y_n) + \frac{(x - x_m)}{1!}\partial_x f_{mn} + \frac{(y - y_n)}{1!}\partial_y f_{mn} + \\ &+ \frac{(x - x_m)^2}{2!}\partial_{xx} f_{mn} + \frac{(y - y_n)^2}{2!}\partial_{yy} f_{mn} + \\ &+ 2\frac{(x - x_m)(y - y_m)}{2!}\partial_{xy} f_{mn} + \dots \end{aligned} \tag{2.11}$$

По същество процедурата за намирането дискретни схеми за частните производни на функцията f които не са смесени е същата като раздела по-горе. Например ако искаме да получиме три-точкова диференча формула за първата частна производа спрямо променливата x нека извадим двата члена $f_{m+1,n} = f(x_m + h_x, y_n)$ и $f_{m-1,n} = f(x_m - h_x, y_n)$ един от друг замествайки във формула 2.11 $x = x_m + h$ и $x = x_m - h$ съответно, докато $y = y_n$

$$\partial_x f_{mn} = \frac{f_{m+1,n} - f_{m-1,n}}{2h_x} + O(h^2) \tag{2.12}$$

следвайки същата процедура можем да намерим и останалите диференчни схеми за първите и вторите частни производни

$$\partial_x f_{mn} = \frac{f_{m+1,n} - f_{m-1,n}}{2h_x} + O(h^2) \tag{2.13}$$

$$\partial_y f_{mn} = \frac{f_{m,n+1} - f_{m,n-1}}{2h_y} + O(h^2) \tag{2.14}$$

$$\partial_{xx} f_{mn} = \frac{f_{m+1,n} - 2f_{mn} + f_{m-1,n}}{h_x^2} + O(h^2) \tag{2.15}$$

$$\partial_{yy} f_{mn} = \frac{f_{m+1,n} - 2f_{mn} + f_{m,n-1}}{h_y^2} + O(h^2) \tag{2.16}$$

$$\partial_{xy} f_{mn} = \frac{f_{m+1,n+1} - f_{m-1,n+1} - f_{m+1,n-1} + f_{m-1,n-1}}{4h_x h_y} + O(h^2) \tag{2.17}$$

$$\partial_{xy} f_{mn} = -\frac{f_{m+1,n} + f_{m-1,n} + f_{m,n+1} + f_{m,n-1} - 2f_{mn} - f_{m+1,n+1} - f_{m-1,n-1}}{2h_x h_y} \tag{2.18}$$

2.2.2 Тримерен случай

В случай на функция на три променливи $f = (x, y, z)$ дискретизацията се осъщесвява върху тримерна решетка от стойности на независимите променливи $\{x_m, y_n, z_l\}$, където $m, n, l \in \mathbb{Z}$ и стъпки на дискретизация $h_x = x_{m+1} - x_m$, $h_y = y_{n+1} - y_n$ и $h_z = z_{l+1} - z_l$. Функцията представена върху решетката придобива три индекса

$$f_{mnl} = f(x_m, y_n, z_l) \tag{2.19}$$

отново диференчните схеми се получават чрез представяне на функцията в ред на Тейлор около произволна точка $\{x_m, y_n, z_l\}$ и нейните съседи в интервала $[x_{m-2}, x_{m+2}] \times [y_{n-2}, y_{n+2}] \times [z_{l-2}, z_{l+2}]$ в случая общо 125. За пълнота нека запишем някой от по-често използваните диференчни схеми

$$\partial_x f_{mnl} = \frac{f_{m+1,nl} - f_{m-1,nl}}{2h_x} + O(h^2) \quad (2.20)$$

$$\partial_y f_{mnl} = \frac{f_{m,n+1,l} - f_{m,n-1,l}}{2h_y} + O(h^2) \quad (2.21)$$

$$\partial_z f_{mnl} = \frac{f_{mn,l+1} - f_{mn,l-1}}{2h_z} + O(h^2) \quad (2.22)$$

$$\partial_{xx} f_{mnl} = \frac{f_{m+1,nl} - 2f_{mnl} + f_{m-1,nl}}{h_x^2} + O(h^2) \quad (2.23)$$

$$\partial_{yy} f_{mnl} = \frac{f_{m,n+1,l} - 2f_{mnl} + f_{m,n-1,l}}{h_y^2} + O(h^2) \quad (2.24)$$

$$\partial_{zz} f_{mnl} = \frac{f_{mn,l+1} - 2f_{mnl} + f_{mn,l-1}}{h_z^2} + O(h^2) \quad (2.25)$$

2.3 Задача за решаване едномерното уравнение на Поасон

Нека разгледаме едномерното уравнение на Поасон с хомогенни гранични условия на Дирихле в интервала $x \in [0, 1]$:

$$\begin{aligned} -\frac{d^2u(x)}{dx^2} &= f(x) \\ u(0) &= 0 \\ u(1) &= 0 \end{aligned}$$

Функцията $f(x)$ може да се дискретизира върху набор от точки $x_i = i/(N+1)$ за $i \in 0, 1, ..., N+1$, така че $u_i = u(x_i)$. Втората производна в горното уравнение ще представим с помощта на диференчни разлики, използвайки уравнение 2.9 като:

$$-\frac{d^2u(x_i)}{dx^2} = \frac{-u(x_{i+1}) + 2u(x_i) - u(x_{i-1})}{h^2}$$

Където $h = 1/(N+1)$ ще наричаме стъпка на дискретизация. Ако заместим приближеният израз на втора производна в едномерното уравнение и за улеснение използваме съкратеният запис $u_i = u(x_i)$ и $f_i = f(x_i)$, ще получим дискретна схема за решение на уравнението на Поасон:

$$\begin{aligned} -u_{i-1} + 2u_i - u_{i+1} &= h^2 f_i \text{ за } 1 \leq i \leq N \\ u_0 &= 0 \\ u_{N+1} &= 0 \end{aligned}$$

Горният запис представлява система от N-брой линейни уравнения за неизвестните u_i $(i \in 1, 2, 3, \ldots, N)$ и може да се представи в матрична форма $\mathbf{T}u = h^2 f$, където матрицата $\mathbf{T}$, вектор стълбовете u и f имат вида:

$$\mathbf{T} = \begin{bmatrix} 2 & -1 & 0 & 0 & & 0 \\ -1 & 2 & -1 & 0 & & 0 \\ 0 & -1 & 2 & -1 & & 0 \\ & & \ddots & \ddots & \ddots & \\ & & & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & -1 & 2 \end{bmatrix}; u = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{N-1} \\ u_N \end{bmatrix}; f = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_{N-1} \\ f_N \end{bmatrix}$$

Напишете код реализиран на някой от следните програмни езици *Fortran*, *C* или *C++* който решава едномерното уравнение на Поасон чрез намиране неизвестният вектор u в матричното уравнение $\mathbf{T}u = h^2 f$ използвайки итеративният метод на Якоби, разгледан в Глава 1.5 за функцията $f(x)$

$$f(x) = -x\,(x+3)\,e^x$$

Сравнете полученото числено решение за различен брой точки на дискретизация $N = 10, 100, 1000$ с аналитичното решение:

$$u(x) = x\,(x-1)\,e^x$$

Решение: Елементите на матрицата $\mathbf{T}$ могат да се опишат с помощта на символите на Кронекер δ_{ij}

$$\mathbf{T}_{ij} = -\delta_{i-1,j} + 2\delta_{ij} - \delta_{i+1,j} \tag{2.26}$$

Не е трудно да се провери че за произволен ред от матрицата е изпълнено условието за сходимост на метода на Якоби записано в уравнение 1.26. Съответно матрицата $\mathbf{T}$ може да се прадстави като сума $\mathbf{T} = \mathbf{D} + \mathbf{R}$ от диагоналнат аматрицат $D_{ij} = 2\delta_{ij}$ и остатъчната матрица $R_{ij} = -\delta_{i-1,j} - \delta_{i+1,j}$. Изразът за итеративното решение $\mathbf{u}^k$ на линейната система може да се получи комбинирайки израза 1.25 и явният вид на матрицата 2.26 представен с помощта на матриците $\mathbf{D}$ и $\mathbf{R}$

$$\begin{aligned} u_i^{k+1} &= D_{ij}^{-1}\left(h^2 f_j - R_{jm} u_m^k\right) \\ &= \frac{1}{2}\delta_{ij}\left(h^2 f_j + \left(\delta_{j-1,m} + \delta_{j+1,m}\right) u_m^k\right) \\ &= \frac{1}{2}\left(h^2 f_i + u_{i-1}^k + u_{i+1}^k\right) \end{aligned} \tag{2.27}$$

По-долу е показана заготовка на кода който реализира задачата. Броят на точките на дискретизация се запазва във входната Целочислена променлива `n`, която и определя размера на работните масиви `u(0 : n + 1)`, `unew(0 : n + 1)` и `f(0 : n + 1)`. Където масивът `u` пази текущото решение $\mathbf{u}^k$, масива `unew` пази "новото" решение $\mathbf{u}^{k+1}$, докато `f`

съдържа дискретните стойност на функцията $f_i = f(x_i)$ върху решетката от дискретни точки $x_i = ih + x_{min}$, където $i = 0, 1, \ldots, n, n+1$. Граничните точки на интервала на решение се пазят в променливите `xmin` и `xmax`. Стъпката на дискретизация `h` се определя с помощта на формулата $h = \dfrac{x_{max} - x_{min}}{n+1}$. Програмата търси решение само за вътрешните индекси на масивите `u`$(1:n)$ и `unew`$(1:n)$. Докато стойностите на масива на граничните точки `u`(0) и `u`$(n+1)$ се запазват равни на нула. Итерационният цикъл на Якоби се контролира от два параметъра: целочислената променлива `maxit` която указва максималният брой итерации за търсене на решението и реалната променлива `eps` която определя точността на решението като разликата `diff` между "новото" `unew` и текущото решение `u`.

Програмен код 2.2: Примерен код на задачата за решение на едномерното уравнение на Поасон. Метод на Якоби

```
1  program poisson
2  implicit none
3
4  integer :: i, k, n, maxit
5  double precision, allocatable, dimension(:) :: u, unew, f
6  double precision :: x, h, xmin, xmax, eps, diff
7
8  write(0,*)"n␣=␣"
9  read(*,*)n
10
11 allocate(u(0:n+1))
12 allocate(unew(0:n+1))
13 allocate(f(0:n+1))
14
15 xmin = 0.d0
16 xmax = 1.d0
17 h = (xmax - xmin) / (n+1)
18
19 do i = 0, n+1
20   x = i * h + xmin
21   f(i) = x * ( x + 3) * exp(x)
22 end do
23
24 maxit = 1000000
25 eps = 1.d-5
26
27 u(0:n+1) = 0.d0
28 k = 0
29 do
30  ...
31  Реализиране на уравение 2.27
32  ...
33    diff = sqrt(dot_product(unew(1:n)-u(1:n),unew(1:n)-u(1:n)))
34   u(1:n) = unew(1:n)
35
36   if ( diff .le. eps .or. k .eq. maxit ) exit
37   k = k + 1
38 end do
```

```
39
40  do i = 0, n+1
41    x = i * h + xmin
42    write(*,*)x,u(i),x*(1-x)*exp(x)
43  end do
44
45  deallocate(u)
46  deallocate(unew)
47  deallocate(f)
48
49  end program poisson
```

Нека разширим нашата задача като се опитаме да намерим решението с по-висока точност използвайки дискретна пет точкова схема от по-висок ред за израза на втората производна u''

$$u_i'' = \frac{-u_{i-2} + 16u_{i-1} - 30u_i + 16u_{i+1} - u_{i+2}}{12h^2} \tag{2.28}$$

трябва да се отбележи че с помощта на горното уравнение 2.28 можем да изразим втората производна само за точките $\{x_i, u_i\}$ с индекси $i \in 2, 3, \ldots, n-1$. За точките с индекси $i = 1$ и $i = n$ трябва да се изпозлва три точковата дискретна схема. Тогава линейната система приема вида

$$\begin{aligned} -u_{i-1} + 2u_i - u_{i+1} &= h^2 f_i \text{ за } i = 1 \text{ или } i = N \\ -u_{i-2} + 16u_{i-1} - 30u_i + 16u_{i+1} - u_{i+2} &= 12h^2 f_i \text{ за } 2 \leq i \leq N-1 \\ u_0 &= 0 \\ u_{N+1} &= 0 \end{aligned}$$

В матричен вид горната линейна система може да се обобщи като

$$\mathbf{T} = \begin{bmatrix} 24 & -12 & 0 & 0 & 0 & & 0 \\ -16 & 30 & -16 & 1 & 0 & & 0 \\ 1 & -16 & 30 & -16 & 1 & & 0 \\ & \ddots & \ddots & \ddots & \ddots & \ddots & \\ 0 & & 1 & -16 & 30 & -16 & 1 \\ 0 & & 0 & 1 & -16 & 30 & -16 \\ 0 & & 0 & 0 & 0 & -12 & 24 \end{bmatrix}; u = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{N-2} \\ u_{N-1} \\ u_N \end{bmatrix}; f = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_{N-2} \\ f_{N-1} \\ f_N \end{bmatrix}$$

За съжаление методът на Якоби за намиране решението на линейната система е неприложим защото съгласно уравнение 1.26 сумата на абсолютните стойности на извън диагоналните елементи надвишава абсолютната стойност на диагоналният елемент. Например за 3-ти ред можем да запишем

$$|30| < |1| + |-16| + |-16| + |1| = 34$$

За да решим системата уравнения обаче можем да изпозлваме библиотеката *LAPACK* и нейната възможност да решава линейни системи с общо диагонални матрици. В случая матрицата **T** е пет диагонална с по два допълнителни диагонала от ляво и дясно спрямо главният диагонал. Решаването на линейната включва последователното извикване на процедурите `DGBTRF` и `DGBTRS`

- Процедурата `DGBTRF(M, N, KL, KU, AB, LDAB, IPIV, INFO)` извършва LU декомпозиция на общо диагонална матрица `A` за линейната система $\mathtt{A} \cdot \mathtt{x} = \mathtt{b}$ Където аргументите са
  - `M` е целочислен (`integer`) параметър който пази броя на редовете в матрицата `A`, като `M >= 0`.
  - `N` е целочислен (`integer`) параметър който пази броя на колоните в матрицата `A`, като `N >= 0`.
  - `KL` е целочислен (`integer`) параметър който пази броя на диагоналите под главният диагонал в матрицата `A`, като `KL >= 0`.
  - `KU` е целочислен (`integer`) параметър който пази броя на диагоналите над главният диагонал в матрицата `A`, като `KU >= 0`.
  - `AB` е двумерна матрица с двойна точност (`double precision`) с размер (`LDAB`, `N`). На входа матрицата `AB` съдържа елементите на общо диагоналната матрица `A` в компактен запис, където редовете между $\mathtt{KL}+1$ и $2*\mathtt{KL}+\mathtt{KU}+1$ трябва да бъдат инициализирани с елементите на матрицата `A`, така че `j`-тата колона на матрицата `A` се записва в `AB` в следният ред $\mathtt{AB}(\mathtt{kl}+\mathtt{ku}+1+\mathtt{i}-\mathtt{j},\mathtt{j}) = \mathtt{A}(\mathtt{i},\mathtt{j})$ за $\mathtt{max}(1,\mathtt{j}-\mathtt{ku}) <= \mathtt{i} <= \mathtt{min}(\mathtt{m},\mathtt{j}+\mathtt{kl})$. При приключване на извикването матрицата `AB` съдържа факторизираната матрица `A` в компактен запис.
  - `LDAB` е целочислен (`integer`) параметър който пази водещият размер на матрицата матрицата `A`, като $\mathtt{LDAB} >= 2*\mathtt{KL}+\mathtt{KU}+1$.
  - `IPIV` е целочислен (`integer`) едномерен масив с размер $\mathtt{dimension}(\mathtt{min}(\mathtt{M},\mathtt{N}))$ който пази индексите на пивотиращите редове.
  - `INFO` е целочислен (`integer`) параметър който пази кода на операцията.
    - `INFO` $= 0$: декомпозицията е приключула с успех
    - `INFO` $= -\mathtt{i}$: `i`-тия аргумент на процедурата има грешна стойност.
    - `INFO` $= +\mathtt{i}$: $\mathtt{U}(\mathtt{i},\mathtt{i}) = 0$ факторизацията не може да бъде завършена
- Процедурата `DGBTRS(TRANS, N, KL, KU, NRHS, AB, LDAB, IPIV, B, LDB, INFO)` решава линейната система $\mathtt{A} \cdot \mathtt{x} = \mathtt{b}$ или $\mathtt{A}^{\mathtt{T}} \cdot \mathtt{x} = \mathtt{b}$. Където аргументите на процедурата са
  - *TRANS* е символ (`character * 1`) който задава формата на линейната система

  - $*$ 'N': $\texttt{A} \cdot \texttt{x} = \texttt{b}$ нормална задача
  - $*$ 'T': $\texttt{A}^{\texttt{T}} \cdot \texttt{x} = \texttt{b}$ транспонирана линейна система
  - $*$ 'C': $\texttt{A}^{\dagger} \cdot \texttt{x} = \texttt{b}$ Комплексно спрегната линейна система
- N е целочислен (integer) параметър който пази броя на колоните в матрицата A, като N >= 0.
- KL е целочислен (integer) параметър който пази броя на диагоналите под главният диагонал в матрицата A, като KL >= 0.
- KU е целочислен (integer) параметър който пази броя на диагоналите над главният диагонал в матрицата A, като KU >= 0.

- AB е двумерна матрица с двойна точност (double precision) с размер (LDAB, N). Съдържа факторизираната матрица A чрез процедурата DGBTRF.
- LDAB е целочислен (integer) параметър който пази водещият размер на матрицата матрицата A, като LDAB >= 2 * KL + KU + 1.
- IPIV е целочислен (integer) едномерен масив с размер dimension(min(M, N)) който пази индексите на пивотиращите редове.
- B е масив с двойна точност (double precision) с размер (LDB, NRHS). На входа съдържа дясната част на линейната система. При успешно приключване на извикването на процедурата съдържа решението x.
- LDB е целочислен (integer) параметър който пази водещият размер на масива B, като LDAB >= min(1, N).
- INFO е целочислен (integer) параметър който пази кода на операцията.
  - INFO = 0: намирането на решението е успешно
  - INFO = −i: i-тия аргумент на процедурата има грешна стойност.

Програмен код 2.3: Примерен код на задачата за решение на едномерното уравнение на Поасон, чрез пет точкова дискретизационна схема за втората производна. Метод на LU декомпозиция

```
1  program poisson
2  implicit none
3
4  interface
5    function delta(i,j)
6      implicit none
7      integer, intent(in) :: i, j
8      integer :: delta
9    end function delta
10 end interface
11
```

```
12 integer :: i, j, n
13 double precision, allocatable, dimension(:) :: u, f
14 double precision, allocatable, dimension(:,:) :: t
15 double precision :: x, h, xmin, xmax, eps, diff
16 integer, allocatable, dimension(:) :: ipiv
17 integer :: k, kl, ku, ldt, info
18 write(0,*)'n = '
19 read(*,*)n
20
21 kl = 2
22 ku = 2
23 ldt = 2*kl+ku+1
24 k = kl + ku + 1
25
26 allocate(u(0:n+1))
27 allocate(f(0:n+1))
28 allocate(t(ldt,n))
29 allocate(ipiv(n))
30
31 xmin = 0.d0
32 xmax = 1.d0
33 h = (xmax - xmin) / (n+1)
34
35 do i = 1, n
36   x = i * h + xmin
37   f(i) = - x * ( x + 3) * exp(x) * 12.d0 * h ** 2
38 end do
39
40 t(:,:) = 0.d0
41
42 do i = 1, n
43   do j = max(i-kl,1),min(i+ku,n)
44     if ( i.eq.1.or.i.eq.n) then
45       t(k+i-j,j) = -12.d0*(delta(i-1,j)+delta(i+1,j)) + 24.d0*delta(i,j)
46     else
47       t(k+i-j,j) = 1.d0*(delta(i-2,j)+delta(i+2,j))-16.d0*(delta(i-1,j)+delta(i
           +1,j)) + 30.d0*delta(i,j)
48     end if
49   end do
50 end do
51
52
53 call DGBTRF(n,n,kl,ku, t, ldt,ipiv,info)
54 if (info.eq.0) then
55   call DGBTRS('N',n,kl,ku,1,t,ldt,ipiv,f(1:n),n,info)
56   u(1:n) = f(1:n)
57
58   do i = 0, n+1
59     x = i * h + xmin
60     write(*,*)x,u(i),x*(x-1)*exp(x)
61   end do
62 else
63   write(0,*)"Error occured. The matrix can't be factorized."
```

```
64  end if
65
66  deallocate(u)
67  deallocate(f)
68  deallocate(t)
69  deallocate(ipiv)
70
71  end program poisson
72
73  function delta(i,j)
74  implicit none
75  integer, intent(in) :: i, j
76  integer :: delta
77
78  if ( i.eq.j) then
79    delta = 1
80  else
81    delta = 0
82  end if
83
84  end function delta
```

Глава 3

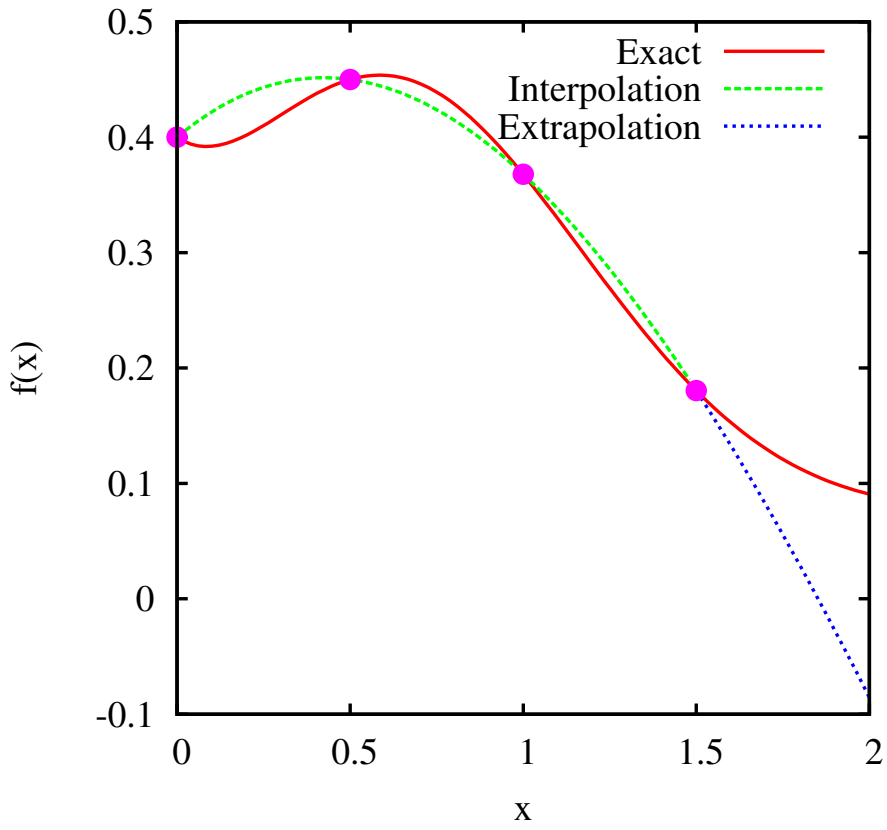
Интерполация и екстраполация на данни

Понякога без да познаваме явният вид на функцията $f = f(x)$ на една променлива можем да познаваме нейните стойности в ограничен брой точки от реалната ос x_0, x_1, ..., x_n, предполагайки че точките са подредени така че $x_0 < x_1 < \ldots < x_n$. Например в случаите когато измерваме някаква физична величина или стойнотите $f(x_i)$ са резултат от някаква сложна изчислителна процедура която не може да бъде представена в проста функционална форма. Нека отбележим също така че не е задължително множеството от точки $\{x_i\}$ да бъде еквидистантно.

Можем да си поставим за цел да пресметнем стойността на функцията за произволна стойност на независимият аргумент x, т. е. да се опитаме да построим гладка крива която свързва стойностите на функцията известни в точките $\{x_i\}$. В случаите когато x попада в интервала между най-малката и най-голяма стойности или $x \in [x_0, x_n]$ тази задача се нарича *интерполация*, ако x не принадлежи на този интервал процедурата се нарича *екстраполация*. В сравнение с *интерполацията*, *екстраполацията* на функцията f се разглежда като по-неточна задача, фигура 3.1.

Добре е процедурата по интерполация и екстраполация да описва неизвестната функцинална зависимостf вътре и извън интервала чрез познати нам аналитични функции. Подобна функционална зависимост трябва да бъде възможно най-обща така че да сме в състояние да описваме широк набор от функции които могат да възникнат в практиката. По-долу в тази глава ще разгледам най-широко разпространените функционални зависимости - интерполация с помощта на полиноми, раздел 3.1. Рационални функции в секция 3.2 могат да бъдат удобни в случаите на рязко-изменящи се зависимости или пик-образни форми. Интерполацията с помощта на тригометрични функции ще бъдат разгледани в отделна Глава 10, където се разглеждат Фурие методи. И най-накрая ще разгледаме интерполационна процедура с помощта на кубичен сплайн. Разбира се тези интерполационни процедури не изчерпват всички съществуващи в практиката интерполационни методи. Читателят може да разшири познанията си като разгледа в детайли глава 3 на книгата *Числени рецепти* [8].

Най-накрая нека отбележим също така че интерполацията е свързана с апроксима-



Фигура 3.1: Пример за интерполация на функцията $f(x)$ в интервала $[0, 1.5]$ и екстраполация извън него $\{1.5, \infty\}$. Интерполацията на функцията винаги съвпада със самата функция $f(x)$ в дефиниционните точки $\{0, 0.5, 1, 1.5\}$.

цията на функция но се различава от нея. При апроксимацията целта е да се намери по-проста функция която се манипулира и пресмята по-лесно за да опишем (приближено) по-сложна функционална зависимост. Освен това в случай на интерполация множеството точки в които познаваме функцията $f_i = f(x_i)$ е известно предварително, докато апроксимационните схеми предполагат че можем да пресметнем стойността на функцията $f(x)$ за произволна точка.

3.1 Полиномиална интерполация

Класическата интерполационна схема с помощта на полиноми разчита на геометричното правило според което през всеки $n+1$-брой точки в евклидовата равнина може да бъде построен еднозначно точно един полином от n-та степен. В практиката интерполиращата функция се нарича полином на Лагранж. За момент нека предположим че познаваме функцията в $n+1$ брой точки, т.е. имаме $n+1$ двойки $\{x_0, f_0\}$, $\{x_1, f_1\}$,

$\ldots$, $\{x_n, f_n\}$. Освен това точките $\{x_i\}$ не съвпадат или $x_i \neq x_j$ за произволни двойки $i, j \in 0, \ldots, n$. Тогава можем да дефинираме полином на Лагранж от степен n, $L_n(x)$ като

$$L_n(x) = \sum_{i=0}^{n} f_i l_i(x) \tag{3.1}$$

където степенните функции $l_i(x)$ се дефинират като

$$l_i(x) = \prod_{\substack{0 \leq m \leq n \\ m \neq i}} \frac{x - x_m}{x_i - x_m} = \frac{x - x_0}{x_i - x_0} \ldots \frac{x - x_{i-1}}{x_i - x_{i-1}} \frac{x - x_{i+1}}{x_i - x_{i+1}} \ldots \frac{x - x_n}{x_i - x_n} \tag{3.2}$$

Полиномите $l_j(x)$ имат точно n-брой нули в интервала $[x_0, x_n]$ а именно

$$l_i(x_j) = \delta_{ij} \tag{3.3}$$

В практиката интерполационната формула 3.1 се използва рядко заради относително по-трудна реализация на програмно ниво, както и невъзможността да се добавят нови точки $\{x_{n+1}, y_{n+1}\}$ към известното вече множество без да е необходимо да се пресмятат отново всички коефициенти. Нека разгледаме един алтернативен рекурсивен алгоритъм на Невил, който позволява произволно да се добавят нови точки в интерполационната схема. За целта ще дефинираме фамилия от полиномиални функции $P_i(x)$, $P_{ij}(x)$, $\ldots$, $P_{012\ldots n}(x)$. Смисълът на всяка една от тези серии функции е следният: Функциите с един индекс $P_i(x)$ описват полином от нулева степен за точка $\{x_i, f_i\}$, например $P_1(x) = f_1$. Полиномите с два индекса $P_{ij}(x)$ описват линейна функция която минава през точките $\{x_i, f_i\}$ и $\{x_j, f_j\}$. Съответно функцията $P_{012\ldots n}(x)$ описва полинома от степен n който всъщност търсим. Добрата новина е че съществува рекурентна зависимост между полиномите от по-висок ред и полиномоите от по-нисък порядък. За целта ще разгледаме полинома $P_{i(i+1)\ldots(i+m)}(x)$ от степен m, който минава през множеството точки $\{(x_i, f_i), \ldots, (x_{i+m}, f_{i+m})\}$. За него може да се дефинира следната рекуретна зависимост:

$$P_{i(i+1)\ldots(i+m)}(x) = \frac{(x - x_{i+m}) P_{i(i+1)\ldots(i+m-1)}(x) + (x_i - x) P_{(i+1)(i+2)\ldots(i+m)}(x)}{x_i - x_{i+m}} \tag{3.4}$$

Уравнение 3.4 комбинирано с условието $P_i(x) = f_i$ ни позволява да пресметнем полиномиалната интерполация за произволна стойност на аргумента x в интервала $[x_0, x_n]$. В случай на $n = 3$ можем да онагледим числената процедура, като построим итерационната "пирамида" на зависимостите между полиномите $P_{i(i+1)\ldots(i+m)}$ от различен ред.

Възможно е да се дефинира и съкратен запис на полиномите в уравнение 3.4 а именно $P_{i(i+1)\ldots j}(x) = P_{i,j}(x)$, където $j = i + m$. В който случай горното уравнение се записва като

$$
\begin{array}{lllllll}
x_0: & f_1 & = & P_0 & & & \\
 & & & & P_{01} & & \\
x_1: & f_2 & = & P_1 & & P_{012} & \\
 & & & & P_{12} & & P_{0123} \\
x_2: & f_3 & = & P_2 & & P_{123} & \\
 & & & & P_{23} & & \\
x_3: & f_4 & = & P_3 & & &
\end{array}
$$

$$P_{i,j}(x) = \frac{(x_j - x)P_{i,j-1}(x) + (x - x_i)P_{i+1,j}(x)}{x_j - x_i} \tag{3.5}$$

По-нататък нека разгледаме примерен код написан на програмният език *Fortran* който реализира алгоритъма на Невил

```
1  subroutine neville(xx, ff, x, f)
2  implicit none
3  !Input/Output variables
4  double precision, dimension(:), intent(in) :: xx, ff
5  double precision, intent(in) :: x
6  double precision, intent(out) :: f
7  !Local variables
8  double precision, dimension(:), allocatable :: d
9  double precision, dimension(:,:), allocatable :: Pij
10
11 integer :: i, j, m, n
12
13 n = size(xx) - 1
14
15 allocate(d(0:n))
16 allocate(Pij(0:n, 0:n))
17
18 do i=0,n
19    Pij(i,i)= ff(i+1)
20 enddo
21
22 d(:) = x - xx(1:n+1)
23
24 do m=0,n-1
25    do j=m,n
26       if (m.ne.j) then
27         i = j - m - 1
28         Pij(i,j) = ((-d(j))*Pij(i,j-1)+d(i)*Pij(i+1,j)) / (d(i)-d(j))
29       end if
30    end do
31 end do
32
33 f = Pij(0,n)
34
35 deallocate(d)
36 deallocate(Pij)
```

```
37
38  end subroutine neville
```

Процедурата `neville(xx, ff, x, f)` приема като входни двата масивите `xx(0 : n)` и `ff(0 : n)` с размер $(n + 1)$-брой елемента които съответно съхраняват координатите x_i и стойността на функцията f_i в тях. Предполага се също така че елементите в масива `xx` са подредени във възходящ ред. Процедурата `neville` пресмята матрицата от елементи P_{ij} (`Pij(i, j)`) за стойността на аргумента `x` и връща като резултат стойността на интерполираната фунцкия в променливата `f`. Доколко полиномиалната интерполацията на функции е широко разпространена изчислителна процедура, е добра практика да се използват вече съществуващи и проверени библиотеки и програми. Като не претендираме за пълна изчерпателност на същесвуващи програмни реализациии нека разгедаме някои от тях

1. В книгата *Числени рецепти*, глава 3.2, [8] е разгледана процедурата `polint` (`xa, ya, x, y, dy`), която реализира същият алгоритъм. Интерфейса към процедурата е подобен на примерният по-горе. В случая входният масив `xa` съдържа координатите на точките x_i, съответно входният масив `ya` съдържа масива от стойностите на функцията f върху точките x_i. Входната променливата `x` съдържа стойността на независимата променлива x, докато в изходната променлива `y` се записва резултатът от полиномиалната интерполация. Изходната променлива `dy` съдържа "оценка" за грешката от интерполацията.

2. Научната библиотека с отворен код GSL също има реализиран интерфейс към интерполационни процедури които включват и полиномиална интерполация. Процедурата се осъществява с помощта на три помощни функции

   - `gsl_interp * gsl_interp_alloc (const gsl_interp_type * T, size_t size)`

   Функцията връща указател към новосъздаден обект от тип `T` с размер (брой) `size` на интерполационните точки.

   - `int gsl_interp_init (gsl_interp * interp, const double xa[], const double ya[], size_t size)`

   Функцията инициализира обекта `interp` с данните (`xa`,`ya`) където `xa` и `ya` са масиви с размер `size`. Обектът (`gsl_interp`) не запазва самите данни (`xa`,`ya`) а съхранява само състоянието (коефициентите на интерполация) на вида интерполация. Отново както и преди масивът `xa` се предполага че е подреден във възходящ ред.

   - `void gsl_interp_free (gsl_interp * interp)`

   Тази функция освобождава обекта `interp`.

3. Полиномиалната интерполация е реализирана и в програмните продукти за символни пресмятания. Например нека разгледаме две сходни функции съществуващи в пакетите *Mathematica* и *Maple*, които връщат като израз полином от n-та степен

   - *Mathematica* - `InterpolatingPolynomial`$[\{\{\mathtt{x_0}, \mathtt{f_0}\}, \{\mathtt{x_1}, \mathtt{f_1}\}, \ldots, \{\mathtt{x_n}, \mathtt{f_n}\}\}, \mathtt{x}]$
   - *Maple* - `interp`$([\mathtt{x_0}, \mathtt{x_1}, \ldots, \mathtt{x_n}], [\mathtt{f_0}, \mathtt{f_1}, \ldots, \mathtt{f_n}], \mathtt{x})$

3.1.1 Примерна задача

За да демонстрираме процедурата по интерполация нека разгледаме функцията $f(x) = e^{-(x-3/4)^2}$ в интервала $x \in [0.25, 3]$. Интересуваме се от 5 и 10 точкови интерполационни схеми, като за целта нека разгледаме функцията $f(x)$ табулирана в интервала на интерес за случайно подбрани точки

x_i	0.250	0.500	0.670	0.700	0.740	1.000	1.500	2.300	2.880	3.000
f_i	0.779	0.939	0.994	0.998	1.000	0.939	0.570	0.090	0.011	0.006

Таблица 3.1: Табулирани стойности на пробната функция $f(x) = e^{(x-2/3)^2}$ за 10-точкова интерполационна схема.

Изходният код на примерната програма може да бъде разгледан в допълнение А.2. За да демонстрираме интерполационната процедура ще копираме проекта за интерполация и екстраполация от хранилището GitHub:

1. Направете копие на примерната задача от git хранилището с помощта на командата

   ```
   1 git clone https://github.com/pisov/int.ext.git
   ```

2. Сменете текущата работна директория с папката ./int.ext/polynomial/fortran/ и и следвайте указанията във файла readme.txt, като първо компилирате примерната задача с командата:

   ```
   1 cd int.ext/polynomial/fortran/
   2 make
   ```

3. Стартирайте създаденият изпълним файл poly.x като пренасочите изхода към файла polinom.dat:

   ```
   1 ./poly.x > polinom.dat
   ```

4. Gnuplot скрипта plot.gnu ще нарисува графика на истинската функция (непрекъсната линия) и полиномиалната зависимост (точки), като изпълните командата:

```
1 gnuplot plot.gnu
```

Разгледайте графиката и определете доколко има припокриване между двете функционални зависимости.

Задача: Модифицирайте изходният код на прогрмата `poly.f90` така че да реализира 5 точкова интерполационна схема за точките описани в таблица 3.2.

x_i	0.250	0.670	1.000	1.500	3.000
f_i	0.779	0.994	0.939	0.570	0.006

Таблица 3.2: Табулирани стойности на пробната функция $f(x) = e^{-(x-3/4)^2}$ за 5 точкова интерполационна схема.

Постройте отново графика на интеропилирана функция и определет отново доколко има припокриване между двете функционални зависимости.

3.2 Интерполация с рационални функции

Някои функционални зависимости не се описват добре с помощта на полиноми но могат да се опишат добре с помощта на рационални функции. Например в случаите когато функциите притежават "особени" точки (разходимости) или пик-образни области. Нека разгледаме рационалната функция $R_{i(i+1)\ldots(i+m)}(x)$ която преминава през множество от m-брой точки $\{(x_i, f_i), \ldots, (x_{i+m}, f_{i+m})\}$. Ще предположим следният явен вид за функцията $R_{i(i+1)\ldots(i+m)}(x)$:

$$R_{i(i+1)\ldots(i+m)}(x) = \frac{P_\mu(x)}{Q_\nu(x)} = \frac{p_0 + p_1 x + \ldots + p_\mu x^\mu}{q_0 + q_1 x + \ldots + q_\nu x^\nu} \tag{3.6}$$

доколкото в горното уравнение имаме $(\mu + \nu + 1)$-брой неизвестни p-та и q-та (в най-общият случай q_0 може да бъде произволно число различно от нула), трябва да приемем че $\mu + \nu + 1 = m + 1$. За разлика от полиномиалната интерполация, поради свободата в избора на μ и ν, така че $\mu + \nu = m$, съществува цяла фамилия от възможни рационални функции. Ние ще демонстрираме рекурсивен алгоритъм предложен от Булирш и Стоер подобен на метода на Невил от предходният раздел който позволява да се конструира така наречената "диагонална" рационална функция, а именно степенните на полиномите $P_\mu(x)$ и $Q_\nu(x)$ са равни в случай на четно m или степента на $P_\mu(x)$ е с единица по-голяма, когато m е нечетно число:

$$R_{i(i+1)\ldots(i+m)} = R_{(i+1)\ldots(i+m)} + \frac{R_{(i+1)\ldots(i+m)} - R_{i(i+1)\ldots(i+m-1)}}{\left(\dfrac{x - x_i}{x - x_{i+m}}\right)\left(1 - \dfrac{R_{(i+1)\ldots(i+m)} - R_{i(i+1)\ldots(i+m-1)}}{R_{(i+1)\ldots(i+m)} - R_{(i+1)\ldots(i+m-1)}}\right) - 1} \tag{3.7}$$

горната рекурентна зависимост генерира рационална функция която преминава през $(m+1)$ точки, като използва вече пресметнатите рационални функции които преминават през m и $m-1$ точки съответно. Отново както в случая на полиномиална интерполация началните стойности (m=0) на рационалната апроксимация са самите функционални стойности

$$R_i = f_i \tag{3.8}$$

както и случаите когато на $R_{i(i+1)\ldots(i+m)}$ с $m = -1$

$$R \equiv 0 \tag{3.9}$$

Без да навлизаме в допълнителни детайли относно програмната реализация на този числен алгоритъм, който може да бъде разгледан в подробности в раздел 3.4 на книгата *Числени рецепти* [8]. Ще спомена няколко числени процедури които реализират интерполация с помощта на рационални функции

1. В книгата *Числени рецепти*, [8], раздел 3.4 се описва процедурата `ratint` (`xa`, `ya`, `x`, `y`, `dy`), където масивите `xa` и `ya` съдържат входните данните за точките на дискретизация и стойстта на функцията в тях съответно. Процедурата ... връща пресметната стойност на функцията в изходната променлива `y` за дадено `x`. Изходната променлива `dy` съдържа оценка за „грешката" от интерполацията. Примерният код на процедурата за програмният език *Fortran* може да бъде намерен в допълнение А.3.

2. *Mathematica* - Функцията `RationalInterpolation`[`expr`, {`x`, `m`, `n`}, {$\mathtt{x}_0, \mathtt{x}_1, \ldots, \mathtt{x}_{n+m+1}$}] връща рационална функция на независимата променлива `x` която има полином от степен m в числителя и полином степен n в знаменателя. Функцията преминава през точките {$\mathtt{x}_0, \mathtt{x}_1, \ldots, \mathtt{x}_{n+m+1}$} и стойността на функцията се пресмята с помощта на израза `expr`.

3.3 Кубичен сплайн

Кубичният сплайн е широко разпространен интерполационен алгоритъм който често се използва в инженерните дисциплини поради факта че кривината на сплайн функцията наподобява естесвените форми на усукване и еластична деформация наблюдавани в природата. Например в случаите когато разгледаме изкривяването на греда под действието на външна сила. Равновесната ѝ форма може да се опише добре с помощта на кубичен сплайн. Кубичният сплайн е удобен и с това че при него не се наблюдава числената „нестабилност" на полинимиалната интерполация в случаите на бързо изменящи се функционални стиойности или малък брой интерполационни точки. Също така сплайн процедурата генерира достатъчно гладка функция с непрекъсната първа производна, което е прави удобна и в описанието на различни физични процеси. Кубичният сплайн интерполира всеки интервал между две съседни точки $[x_j, x_{j+1}]$ с

помощта на кубична функция, като се налага условието за гладка първа производна и непрекъсната втора производна в граничните точки на всеки интервал. За да получим интерполационната формула, нека първо разгледаме линейна интерполация на функцията:

$$y(x) = A(x)y_j + B(x)y_{j+1} \tag{3.10}$$

където полиномите от първа степен $A(x)$ и $B(x)$ имат следният функционален вид

$$A(x) \equiv \frac{x_{j+1} - x}{x_{j+1} - x_j} \tag{3.11}$$

$$B(x) \equiv 1 - A(x) = \frac{x - x_j}{x_{j+1} - x_j} \tag{3.12}$$

$$\tag{3.13}$$

така конструираната функция $y(x)$ се изменя линейно между стойностите y_j и y_{j+1} в границите на интервала $[x_j, x_{j+1}]$. За да получим обаче нашата кубична функция, нека за момент предположим че познаваме също така и стойността на втората производна на функцията в точките x_j и x_{j+1}. Тогава можем да добавим кубичен член към функцията $y(x)$, така че

$$y(x) = A(x)y_j + B(x)y_{j+1} + C(x)y''_j + D(x)y''_{j+1} \tag{3.14}$$

явният вид на функциите $C(x)$ и $D(x)$ се дефинира с условието $C(x_j) = C(x_{j+1}) = 0$, $D(x_j) = D(x_{j+1}) = 0$, $C(x)'' = A(x)$ и $D(x)'' = B(x)$ с цел да се запазят граничните условия от уравнение 3.10

$$C(x) \equiv \frac{1}{6}\left(A^3(x) - A(x)\right)(x_{j+1} - x_j)^2 \tag{3.15}$$

$$D(x) \equiv \frac{1}{6}\left(B^3(x) - B(x)\right)(x_{j+1} - x_j)^2 \tag{3.16}$$

използвайки горната система уравнение, можем да намерим аналитичните изрази за първата $y'(x)$ и втората $y''(x)$ производни на функцията $y(x)$ в интервала (x_j, x_{j+1})

$$\frac{d}{dx}y(x) = \frac{y_{j+1} - y_j}{x_{j+1} - x_j} - \frac{3A^2 - 1}{6}(x_{j+1} - x_j)y''_j + \frac{3B^2 - 1}{6}(x_{j+1} - x_j)y''_{j+1} \tag{3.17}$$

също така, не е трудно да се покаже че така конструирана функция има непрекъсната втора производна, т.е.

$$\frac{d^2}{dx^2}y(x) = A(x)y''_j + B(x)y''_{j+1} \tag{3.18}$$

Засега вторите прозводни на функцията $y(x)$ не са ни известни. Но можем да наметим $n-2$ от тях ако наложим условието за непрекъснатост на първата производна във всички "вътрешни" точки в интервала $\{x_1, x_n\}$. За целта нека приравним изразите за първата производна на функцията за два съседни интервала $[x_{j-1}, x_j]$ и $[x_j, x_{j+1}]$, или $y'|_{x \to x_j} = y'|_{x_j \leftarrow x}$. Замествайки стойността на първата производна дефинирана в уравнение 3.17 в точката x_j веднъж разгледана като дясна граница на интервала $[x_{j-1}, x_j]$ и втори път като лява граница на интервала $[x_j, x_{j+1}]$, ще получим следното уравнение

$$\frac{x_j - x_{j-1}}{6} y''_{j-1} + \frac{x_{j+1} - x_{j-1}}{3} y''_j + \frac{x_{j+1} - x_j}{6} y''_{j+1} = \frac{y_{j+1} - y_j}{x_{j+1} - x_j} - \frac{y_j - y_{j-1}}{x_j - x_{j-1}} \tag{3.19}$$

горната тридиагонална линейна система е в сила за всички $j = 2, \ldots, n-1$ и може да бъде решена спрямо вторите производни y''_i ако определим граничните стойности y''_1 и y''_n. В практиката са разпространени два начина за справяне с този проблем

- да се положи y''_1 и/или y''_n равни на нула, с което се задава „естествен" (натурален) кубичен сплайн, който има нулеви втори производни върху едната или двете граници на интервала
- или, вторите производни да се пресметнат с помощта на уравнение 3.17, което изисква да се определят стойности за първата производна на едната или двете граници на интервала. Например може да се получи "оценка" за първата производна на границата на интервала на чрез стойността на функцията y на границата и нейната съседна точка вътре в интервала.

Изчислителната процедура реализираща кубичен сплайн се изпълнява на две стъпки. Първоначално е необходимо да решим линейната тридиагонална система уравнения 3.19 спрямо вторите производни y''. Следващата стъпка е да конструираме функция която пресмята стойността на $y(x)$ за произволен интервал $[x_j, x_{j+1}]$ използвайки уравнение 3.14. Нека разгледаме няколко практически реализации на кубичен сплайн интерполация

1. Функцията `spline` и процедурата `splint`, описани в книгата *Числени рецепти*, раздел 3.3, [8]
2. В раздела 3.1 описващ полиномиална интерполация беше описан вече интерфейса към научната библиотека GSL с отворен код която позволява реализирането на няколко типа (`gsl_interp_type`) интерполационни процедури. В случаят типът на обекта на интерполация трябва да бъде избран като `gsl_interp_cspline`. За допълнителна информация можете да разгледате онлайн документацията на библиотеката GSL [2].

3. Програмният пакет *Mathematica* предлага функцията `Interpolation`$[\{\{x_1, y_1\}, \ldots, \{x_n, y_n\}\}$, `Method` $\rightarrow$ "`Spline`"$]$, където първият аргумент на функцията задава масива с данни в които познаваме аргумента x_i и стойността на самата функция y_i. Вторият аргумент указва интерполационният метод. Възможните опции са "`Spline`" или "`Hermite`".

4. Програмният пакет *Maple* също има реализирана кубична интерполация чрез функцията `CubicSpline`$(xy, opts)$, където `xy` е кратък запис за входният масив данни $[x_1, f_1], \ldots, [x_n, f_n]$. Втоприят аргумент указва възможните допълнителни настройки (опции) на процедурата по интерполация. Например `boundaryconditions` = `natural`, `clamped`(f_1'', f_n'')

3.3.1 Примерна задача:

Реализирайте код на програмният език *Fortran* или *C/C++* който извършва сплайн интерполация на едномерна функция $f(x)$ при зададен краен n брой точки на дискретизация $\{x_i, f_i\}$ за $i \in 1, \ldots, n$. Нека намирането коефициентите на **натурална** сплайн интерполация според уравнение 3.19 се реализира в отделна процедура `splint(x, y, ysec)` в която входните масивите `xx` и `y` съдържат дискретните стойности на независимата променлива x_i и y_i съответно. Масивът lstinlineysec съдържа вътрешните стойности на вторите производни. Създайте и функция `spline(x, y, ysec, xx)` която пресмята интерполираната стойност на функцията за стойност на независимият аргумент `xx`. Сравнете резултатът от полиномиалната интерполация на функцията $y(x) = e^{-(x-3/4)^2}$ в интервала $[0.25, 3]$ за случая на табулирани 5 точки 3.2.

Решение:

Програмен код 3.1: Примерен код реализиращ натурален кубичен сплайн

```
1  program cubicspline
2  implicit none
3
4  interface
5    subroutine splint(x, y, ysec)
6      double precision, dimension(:), intent(in) :: x, y
7      double precision, dimension(:), intent(out) :: ysec
8    end subroutine splint
9    function spline(x, y, ysec, xx)
10   double precision, dimension(:), intent(in) :: x, y, ysec
11   double precision :: xx, spline
12   end function spline
13 end interface
14
15
16 integer :: i, j, n, np
17 double precision, allocatable, dimension(:) :: x, y, ysec
18 double precision :: xx, yy, h, xmin, xmax
```

```
19
20  write(0,*)'np␣=␣'
21  read(*,*)np
22
23  n = 5
24
25  allocate(x(n))
26  allocate(y(n))
27  allocate(ysec(n))
28
29  x = (/0.25d0, 0.67d0, 1.d0, 1.5d0, 3.d0/)
30  y = (/0.779d0, 0.994d0, 0.939d0, 0.57d0, 0.006d0/)
31
32  call splint(x, y, ysec)
33
34  xmin = minval(x)
35  xmax = maxval(x)
36  h = (xmax - xmin) / (np - 1)
37
38  do i = 0, np - 1
39    xx = i * h + xmin
40    write(*,'(2F10.5)')xx,spline(x,y,ysec,xx)
41  end do
42
43  deallocate(x)
44  deallocate(y)
45  deallocate(ysec)
46
47  end program cubicspline
48
49  subroutine splint(x, y, ysec)
50  implicit none
51  double precision, dimension(:), intent(in) :: x, y
52  double precision, dimension(:), intent(out) :: ysec
53
54  double precision, dimension(size(x)) :: d, dl, du
55  integer, dimension(size(x)) :: ipiv
56  integer :: i, n, info
57
58  n = size(x)
59
60  d(1:n-2) = (x(3:n) - x(1:n-2))/3.d0
61  dl(1:n-2) = (x(2:n-1) - x(1:n-2))/6.d0
62  du(1:n-2) = (x(3:n) - x(2:n-1))/6.d0
63
64  ysec(2:n-1) = (y(3:n) - y(2:n-1))/(x(3:n)-x(2:n-1)) - (y(2:n-1)-y(1:n-2))/(x(2:n
        -1)-x(1:n-2))
65
66
67  call dgtsv (n-2, 1, dl, d, du, ysec(2:n-1), n-2, info)
68
69  ysec(1) = 0.d0
70  ysec(n) = 0.d0
71
```

```
72  end subroutine splint
73
74  function spline(x, y, ysec, xx)
75  implicit none
76  double precision, dimension(:), intent(in) :: x, y, ysec
77  double precision :: xx, spline
78
79  integer :: i, idx, n
80  double precision :: a, b, c, d, dx, dxmin
81
82  n = size(x)
83
84  idx = 1
85  dxmin = xx - x(1)
86  do i = 2, n - 1
87    dx = xx - x(i)
88    if (dx.gt.0.d0 .and. dx .lt.dxmin) then
89      idx = i
90      dxmin = dx
91    end if
92  end do
93
94
95  a = (x(idx+1) - xx)/(x(idx+1)-x(idx))
96  b = 1.d0 - a
97  c = (a**3-a)*(x(idx+1)-x(idx))**2
98  d = (b**3-b)*(x(idx+1)-x(idx))**2
99  spline = a*y(idx)+b*y(idx+1)+(c*ysec(idx)+d*ysec(idx+1))/6.d0
100 end function spline
```

Глава 4

Числено интегриране

Нека започнем настоящата глава като си припомним няколко дефиниции от математическитя анализ. Под определен интеграл I на реална функция $f(x)$ в интервала $[a, b]$, ще разбираме ориентираната площ на фигурата, заградена между вертикалните линии през точките a и b, абцисната ос и графиката на $f(x)$, като площта под абсцисата изваждаме. Под неопределен интеграл ще разбираме примитивната функция $F(x)$ на $f(x)$, т.е. ако е дадена функция $f(x)$, примитивна ще наричаме функцията $F(x)$, такава че производната $F'(x) = f(x)$ за всяко x в дефиниционния интервал. Определеният интеграл I може да се представи с помощта на примитивната функция $F(x)$ като:

$$I = F(b) - F(a) = \int_a^b f(x)dx \tag{4.1}$$

Също така интегралът I може да се дефинира и като $I = y(b)$, където $y(x)$ е решение на диференциалното уравнение $y'(x) = f(x)$ с гранично условие $y(a) = 0$.

Важно за нашите разглеждания е да отбележим следните две свойства на определните интеграли.

- Адитивност - Ако точка c принадлежи на интервала $[a, b]$, тогава $\int_a^c f(x)dx + \int_c^b f(x)dx = \int_a^b f(x)dx$
- Линейност - Ако α и β са скаларни величини а $f(x)$ и $g(x)$ функции дефинирани в интервала $[a, b]$, тогава $\int_a^b (\alpha f(x) + \beta g(x))\, dx = \alpha \int_a^b f(x)dx + \beta \int_a^b g(x)dx$

По-долу ще разгледаме числени методи за намирането на определеният интеграл I на произволна функция $f(x)$ дефинирана в интервала $[a, b]$. Необходимо е да споменем че процедурите за числено интегриране са възникнали още в зората на развитието на математичният анализ тъй-като за разлика от диференцирането, където за всяка аналитична функция може да се намери нейната производна, намирането на аналитично решение за неопределеният интеграл $F(x)$ не е гарантирано в общият случай. Главната ни цел ще бъде да разгледаме основните числени алгоритми за намиране на

едномерни интеграли основани на квадратурни формули, като в края на главата ще обърнем внимание и на многомерните интеграли.

4.1 Квадратурни методи

Квадратурните методи се основават на идеята че интегралът I може да се представи като сума от площта на малки елементи които си интерполират чрез полиномиални функции. Като начало ще разгледаме класическите формули за еквидистантна решетка от точки. Нека предположим че познаваме функцията $f(x)$ върху решетка от N-брой точки x_i в интервала $[a, b]$ равно-отдалечени една от друга със стъпка h. Стъпката на дискретизация h ще дефинираме като $h = (b-a)/(N-1)$, $x_i = (i-1)h+a$, за $i = 1, \ldots, N$. Съответно стойностите на функцията $f(x)$ ще записваме като $f_i \equiv f(x_i)$. Интеграционите схеми които използват точките в краищата на дефиниционният интервал a и b ще наричаме уравнения от *затворен* вид. В случаите когато е трудно или невъзможно да се пресметне функцията в краищата на интервала, обаче ще разгледаме интеграционни формули от *отворен* вид в който за пресмятането на интеграла са необходими само точки от вътрешността на интервала.

В основата на квадратурните методи стоят формулите за интегриране върху краен (елементарен) брой интервали. В този интервал могат да се използват различни интерполационни схеми и да получим съответно интеграционни схеми от различен ред. Например, трапицовидната интеграционна схема се основава на линейна интерполация. Докато интеграционните схеми на Симпсън използват полиномиална интерполация от по-висок ред.

4.1.1 Интеграционни формули на Нютон-Кот

Нека разгледаме интервал от две съседни точки x_1 и x_2. Ще намерим интеграционна формула от затворен вид в която можем да интерполираме функцията с полином от първа степен, доколкото познаваме функцията $f(x)$ в двете точки $f_1 = f(x_1)$ и $f_2 = f(x_2)$, можем да запишем

$$f(x) \approx \frac{x_2 - x}{x_2 - x_1} f_1 + \frac{x - x_1}{x_2 - x_1} f_2 + O(h^2) \tag{4.2}$$

Използвайки горният израз 4.2 като приближение за функцията $f(x)$ в интервала $[x_1, x_2]$, можем да намерим интеграла $\int_{x_1}^{x_2} f(x)dx$:

$$\int_{x_1}^{x_2} f(x)dx = h\left[\frac{1}{2}f_1 + \frac{1}{2}f_2\right] + O(h^3) \tag{4.3}$$

Уравнение 4.3 се нарича *правило на Трапеците*, защото ни дава оценка за интеграла на функцията $f(x)$ като площ на трапец със страни f_1 и f_2 и височина h. Формулата е абсолютно точна за полиноми от първа степен, докато грешката от интегриране на един елементарен интервал е пропорционална на третата степен на интеграционната

стъпка h. Сега можем да конструираме три точкова интеграционна формула която ще бъде точна за полиноми до втора степен. Нека разгледаме интервала $[x_1, x_3]$, който включва три точки в себе си x_1, x_2 и x_3. В него интервал можем да интерполираме функцията с полином от втора степен използвайки уравнение 3.1 за $n = 3$:

$$f(x) \approx \frac{(x_2 - x)(x_3 - x)}{(x_2 - x_1)(x_3 - x_1)} f_1 + \frac{(x - x_1)(x_3 - x)}{(x_2 - x_1)(x_3 - x_2)} f_2 + \frac{(x - x_2)(x - x_1)}{(x_3 - x_2)(x_3 - x_1)} f_3 + O(h^4) \quad (4.4)$$

Отново интегрирайки уравнение 4.4 в интервала $[x_1, x_3]$ ще получим *правилото на Симпсън* за интегриране

$$\int_{x_1}^{x_3} f(x)dx = h \left[\frac{1}{3} f_1 + \frac{4}{3} f_2 + \frac{1}{3} f_3 \right] + O(h^5) \quad (4.5)$$

Без да навлизаме в детайли ще запишем още две интеграционни схеми от по-висок ред от фамилията ураванения на Нютон-Кот. *Правило на Симпсън 3/8-ми* с полиноми от 3-та степен.

$$\int_{x_1}^{x_4} f(x) = h \left[\frac{3}{8} f_1 + \frac{9}{8} f_2 + \frac{9}{8} f_3 + \frac{3}{8} f_4 \right] + O(h^5) \quad (4.6)$$

И накрая пет точковата *формула на Боде*

$$\int_{x_1}^{x_5} f(x) = h \left[\frac{14}{45} f_1 + \frac{64}{45} f_2 + \frac{24}{45} f_3 + \frac{64}{45} f_4 + \frac{14}{45} f_5 \right] + O(h^7) \quad (4.7)$$

Трябва да отдебележим че интеграционните формули които включват в елеметарният интервал нечетен брой точки имат един порядък по-висока точност от очакванато поради особеност на интерполационнта формула, защото първият остатъчен член се съкращава.

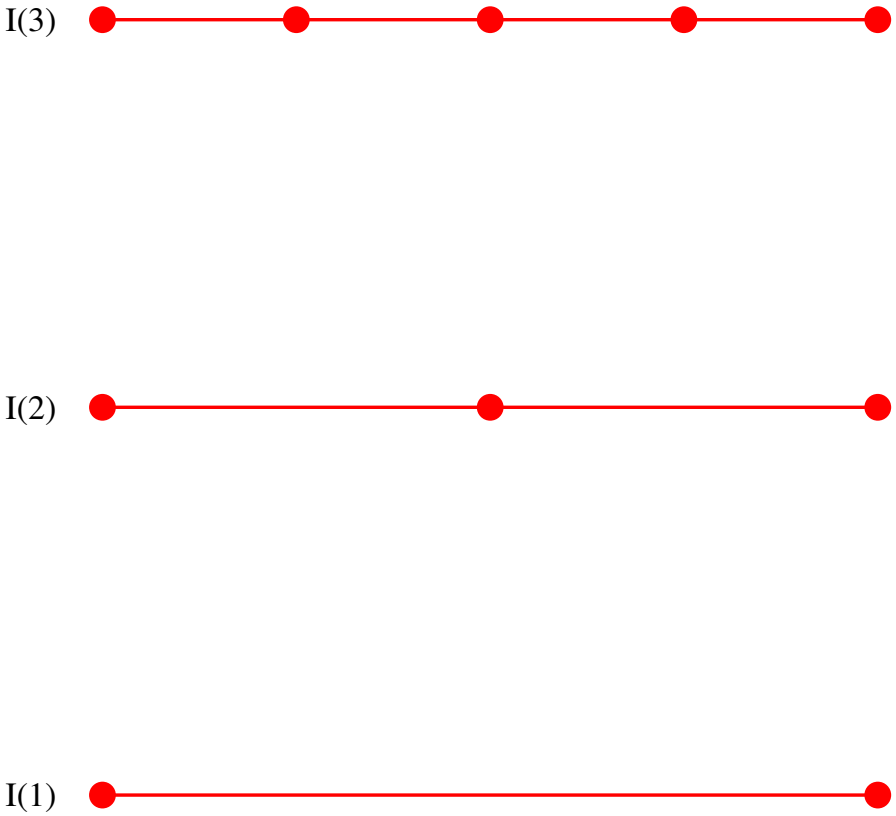
Досега разглеждахме интеграционни формули за "елементарен" интервал който включва само точките от интерполационната схема. Ако искаме обаче да "разширим" нашата интеграционна формула можем да приложим някоя от по-горните интеграционни формули и за останалите точки от интеграционният интервал.

4.1.2 Разширени интеграционни формули от затворен вид

Нека намерим интеграла на функцията $f(x)$ за целият интервал $[x_1, x_N]$. За целта можем да се възползваме от уравнение 4.3 за всичките N брой интервала (x_1, x_2), (x_2, x_3), ..., (x_{N-1}, x_N) и получим *разширеното правило на трапеците*

$$\begin{aligned} \int_{x_1}^{x_N} f(x)dx &= \int_{x_1}^{x_2} f(x)dx + \int_{x_2}^{x_3} f(x)dx + \ldots + \int_{x_{N-1}}^{x_N} f(x)dx \\ &= h \left[\frac{1}{2} f_1 + f_2 + f_3 + \ldots + f_{N-1} + \frac{1}{2} f_N \right] + O\left(\frac{(b-a)^3}{N^2} \right) \end{aligned} \quad (4.8)$$

оценката за грешката в горното уравнение се дава в термините на броя интервали на дискретизация N. Така лесно можем да получим оценка затова как ще се подобри точността на пресмятането на интеграла ако увеличим два пъти броят на интервалите, а именно грешката ще намалее четири пъти.



Фигура 4.1: Подобряването на формулата за интеграла в уравнение 4.8 може да се извърши като се удвои броя на под-интервалите в интеграционната област. На фигурата са схематично са показани решетки на дискретизация за различен четен брой вътрешни интервали, които могат да се използват итеративно за подобряване на решението.

Оказва се че горната формула може да се използва рекурсивно за подобряване точността на пресметнатият интеграл I. Нека разгледаме серия от интеграли $I(J)$, където $J = 1, 2, 3, \ldots$, като освен това нека броят на подинтервалите $N - 1$ в дефиниционната област $[a, b]$ бъде четен, например $N - 1 = 2^{J-1} = 2M$. Тогава стъпката на дискретизация h може да се запише като $h = (b - a)/(2M)$. Не е трудно да се покаже че съществува следната рекурентна зависисмост:

$$I(J) = \frac{I(J-1)}{2} + h\sum_{k=1}^{M} f_{2k} \tag{4.9}$$

За целта първо нека дефинираме $I(1) = (h/2)(f(a) + f(b))$, където $h = (b-a)$. Тогава за всяко $J \geq 2$ дефинираме $I(J) = I(f,h)$, където $I(f,h)$ е интеграла на функцията f пресметна с правилото на трапеците съгласно уравнение 4.3 със стъпка $h = (b-a)/2^{J-1}$. След което за произволно J нека разгледаме интеграла $I(J) = I(f,h)$:

$$I(J) = \frac{h}{2}\left(f_1 + 2f_2 + 2f_3 + 2f_4 \ldots + 2f_{2M-1} + 2f_{2M} + f_{2M+1}\right) \tag{4.10}$$

където $f_1 = f_a$ и $f_{2M+1} = f_b$. Сега можем да разгледаме интеграла $I(J-1) = I(f,2h)$, който може да се запише чрез под-множество от същите функционални стойности които притежават нечетен индекс:

$$I(J-1) = I(f,2h) = \frac{2h}{2}\left(f_1 + 2f_3 + 2f_5 + \ldots + 2f_{2M-1} + f_{2M+1}\right) \tag{4.11}$$

ако прегрупираме нечетните членове в уравнение 4.10 подобно на уравнение 4.11:

$$\begin{aligned} I(J) = I(f,h) &= \frac{1}{2}\left[\frac{2h}{2}\left(f_1 + 2f_3 + 2f_5 + \ldots + 2f_{2M-1} + f_{2M+1}\right)\right] \\ &\quad + \frac{h}{2}\left(2f_2 + 2f_4 + \ldots + 2f_{2M}\right) = \frac{1}{2}(f,2h) + h\sum_{k=1}^{M} f_{2k} \\ &= \frac{I(J-1)}{2} + h\sum_{k=1}^{M} f_{2k} \end{aligned} \tag{4.12}$$

По-долу в секция 4.1.5 ще разгледаме програмна реализация използваща *разширеното правило на Трапеците*, която (както ще се убедим по-късно) може да се приложи и при числената реализация и на следващото квадратурно правило.

Ако използваме три точковата формула на Симпсън дефинирана в уравнение 4.5 и проинтегрираме интервалите $(x_1, x3)$, (x_3, x_5), ..., (x_{N-2}, x_N). Получаваме разширеното правило на Симпсън от затворен вид

$$\begin{aligned} \int_{x_1}^{x_N} f(x)dx &= \int_{x_1}^{x_3} f(x)dx + \int_{x_3}^{x_5} f(x)dx + \ldots + \int_{x_{N-2}}^{x_N} f(x)dx \\ &= h\left[\frac{1}{3}f_1 + \frac{4}{3}f_2 + \frac{1}{3}f_3\right] + h\left[\frac{1}{3}f_3 + \frac{4}{3}f_4 + \frac{1}{3}f_5\right] + \\ &\ldots + h\left[\frac{1}{3}f_{N-2} + \frac{4}{3}f_{N-1} + \frac{1}{3}f_N\right] + NO(h^5) \\ &= h\left[\frac{1}{3}f_1 + \frac{4}{3}f_2 + \frac{2}{3}f_3 + \frac{4}{3}f_4 + \ldots \frac{2}{3}f_{N-2} + \frac{4}{3}f_{N-1} + \frac{1}{3}f_N\right] + O\left(\frac{1}{N^4}\right) \end{aligned} \tag{4.13}$$

Разширената формула на Симпсън изисква четен брой интервали $N-1$, като точността се подобрява шестнадесет пъти ако увеличим броят на интервалите $N-1$ два пъти. Разбира се можем да конструираме други разширени формули като използваме интеграционни формули на Нютон-Кот от по-висок ред или комбинираме няколко интеграционни формули заедно. Например ако комбинираме разширеното интеграционно правило на Симпсън (уравнение 4.13) с правилото на трапеците (уравнение 4.3) по краищата на интеграционния интервал ще получим *разширена интеграционна формула от ред* $1/N^3$

$$\int_{x_1}^{x_N} f(x) = h\left[\frac{5}{12}f_1 + \frac{13}{12}f_2 + f_3 + f_4 + \right. \\ \left. \ldots + f_{N-2} + \frac{13}{12}f_{N-1} + \frac{5}{12}f_N\right] + O(\frac{1}{N^3}) \tag{4.14}$$

Без да навлизаме в детайли, ще споменем че разширената формула на Симпсън може да се запише чрез формулата 4.10 за две последователни стойности на J:

$$S(J) = \frac{4I(J) - I(J-1)}{3} \tag{4.15}$$

уравнение 4.15 е реализирано чрез процедурата `qsimp` в по-следващата секция 4.1.5.

4.1.3 Екстраполационни формули за единичен интервал

Преди да разгледаме разширени интеграционни формули за отворени и полу-отворени интервали, ще бъде бъде удобно ако получим екстраполационни формули за интеграла I с елементарен интервал от две точки, например $[x_1, x_2]$. Идеята е да намерим интеграционна формула за интервала, която обаче не включва една от външните точки, нека това да бъде x_1. Но пък може да включва функционални стойностии на вътрешни за разширеният интервал $[x_1, x_N]$ точки, например f_2, f_3, f_4 и т.н. Подобни формули можем да получим като екстраполираме под-интегралната функция са полиноми от различна степен. Например нека екстраполираме под-интегралната функция $f(x)$ в интервала $[x_1, x_2]$ с постоянна (константна) стойност f_2, или $f(x) = f_2 + O(h)$ за $x \in [x_1, x_2]$. Тогава екстраполационната формула може да бъде получена лесно, интегрирайки константа f_2 в интервала $[x_1, x_2]$:

$$\int_{x_1}^{x_2} f(x)dx = f_2 \int_{x_1}^{x_2} dx + O(h) \int_{x_1}^{x_2} dx = h\,[f_2] + O(h^2) \tag{4.16}$$

По подобен начин ако използваме екстраполационна формула от по-висок ред и приближим функцията $f(x)$ с линейна зависимост използвайки точките x_2 и x_3. Можем да използваме функционалната зависисмост

$$f(x) = \frac{x - x_2}{x_3 - x_2}f_3 + \frac{x_3 - x}{x_3 - x_2}f_2 + O(h^2)$$

не е трудно да се покаже че, екстраполационната схема включваща две точки от вътрешният иннтервал е

$$\int_{x_1}^{x_2} f(x) = h\left[\frac{3}{2}f_2 - \frac{1}{2}f_3\right] + O(h^3) \tag{4.17}$$

Използвайки екстраполационни формули от по-висок ред можем да получим изразите за три и четири точкови интеграционни формули за единичният интервал $[x_1, x_2]$

$$\int_{x_1}^{x_2} f(x) = h\left[\frac{23}{12}f_2 - \frac{16}{12}f_3 + \frac{5}{12}f_4\right] + O(h^4) \tag{4.18}$$

$$\int_{x_1}^{x_2} f(x) = h\left[\frac{55}{24}f_2 - \frac{59}{24}f_3 + \frac{37}{24}f_4 - \frac{9}{24}f_5\right] + O(h^5) \tag{4.19}$$

Уравненията 4.16, 4.17, 4.18 и 4.19 ще ни бъдат особено полезни при намирането на разширени интеграционни формули за оторени и полуотворени интервали. Затова без да губим набраната инерция нека преминем към следващата секция.

4.1.4 Разширени интеграционни формули за отворени и полуотворени интервали

Както споменахме вече по-горе при общите разглеждания, съществуват случаи когато пресмятането на интеграционната функция $f(x)$ не в възможно в единият или двата краища на интеграционният интервал. Например в тази точка е възможно функцията да бъде разходяща. Затова в практиката се налага разглеждането на интеграционни схеми които не включват (отворен интервал) или включват само една (полу отворен интервал) граничните точки a и b. Ако комбинираме уравнение 4.8 за всички вътрешни интервали и уравнение 4.16 за двата външни интервала (x_1, x_2) и (x_{N-1}, x_N), можем да конструираме разширена формула на *правилото на Трапеците* за отворен интервал

$$\begin{aligned}\int_{x_1}^{x_N} f(x)dx &= \int_{x_1}^{x_2} f(x)dx + \int_{x_2}^{x_{N-1}} f(x)dx + \int_{x_{N-1}}^{x_N} f(x)dx \\ &= h\left[f_2\right] + h\left[\frac{1}{2}f_2 + f_3 + \ldots + f_{N-2} + \frac{1}{2}f_{N-1}\right] + h\left[f_{N-1}\right] + O\left(\frac{1}{N^2}\right) \\ &= \left[\frac{3}{2}f_2 + f_3 + f_4 + \ldots + f_{N-2} + f_{N-1} + \frac{3}{2}f_{N-1}\right] + O\left(\frac{1}{N^2}\right)\end{aligned} \tag{4.20}$$

Можем да подобрим точността на интеграционната схема ако комбинираме уравнения 4.13 и 4.18, за да получим *разширеното правило на Симпсън за отворен интервал*

$$\begin{aligned}\int_{x_1}^{x_N} f(x)dx &= \int_{x_1}^{x_2} f(x)dx + \int_{x_2}^{x_{N-1}} f(x)dx + \int_{x_{N-1}}^{x_N} f(x)dx \\ &= h\left[\frac{23}{12}f_2 - \frac{16}{12}f_3 + \frac{5}{12}f_4\right] + h\left[\frac{1}{3}f_2 + \frac{4}{3}f_3 + \frac{2}{3}f_4 + \right. \\ &\left. \ldots + \frac{2}{3}f_{N-2} + \frac{4}{3}f_{N-1} + \frac{1}{3}f_{N-1}\right] + h\left[\frac{23}{12}f_2 - \frac{16}{12}f_3 + \frac{5}{12}f_4\right] + O\left(\frac{1}{N^4}\right) \\ &= h\left[\frac{27}{12}f_2 + 0 + \frac{13}{12}f_4 + \frac{4}{3}f_5 + \right. \\ &\left. \ldots + \frac{4}{3}f_{N-4} + \frac{13}{12}f_{N-3} + 0 + \frac{27}{12}f_{N-1}\right] + O\left(\frac{1}{N^4}\right)\end{aligned} \tag{4.21}$$

В практиката обаче се е наложила интеграционната формула на Ойлер-Маклорен чиято грешка е от порядък $O(1/N^2)$, поради своята симетрия. В нея се използват "междинните" точки на интервалите $[x_i, x_{i+1}]$, а именно $x_{i+1/2} = x_i + h/2$. Удобството на тази формула е че тя отново може да се използва рекурсивно за пресмятане интеграла I с предварително изискана от нас точност, тъй като ако увеличим 3-пъти броя на "междинните" точки можем да използваме итеративно стойността на интеграла от предишното пресмятане:

$$\int_{x_1}^{x_N} f(x)dx = h\left[f_{1/2} + f_{3/2} + \ldots + f_{N-3/2} + f_{N-1/2}\right] + O\left(\frac{1}{N^2}\right) \tag{4.22}$$

4.1.5 Основни алгоритми. Програмна реализация

Нека разгледаме примерен код на процедурата `trapezd`, взаимстван от книгата *Числени рецепти*, който пресмята "подобреният" интеграл `s` между интервала от точки (`a`, `b`) от ред `n`. Броят под-интервали в областта между точките `a` и `b` се избира да бъде четно число 2^{n-1}, като интеграционната формула на трапеците е от затворен вид. Използвайки рекурсивната зависимост 4.9. Процедурата `trapezd`(`func`, `a`, `b`, `s`, `n`), приема като входни аргументи името на реалната функция `func`(`x`) чийто интеграл пресмятаме, границите на интервала `a` и `b`. Променливата s е входно/изходна процедурата `trapezd` предполага че на входа s пази стойността на интеграла 4.10, или $s = I(n-1)$ и връща $s = I(n)$

```
1  subroutine trapzd(func,a,b,s,n)
2    integer :: n
3    double precision :: a,b,s,func
4    external func
5    integer :: it,j
6    double precision :: del,sum,tnm,x
7    if (n.eq.1) then
8      s=0.5*(b-a)*(func(a)+func(b))
9    else
```

```
10      it=2**(n-2)
11      tnm=it
12      del=(b-a)/tnm
13      x=a+0.5*del
14      sum=0.
15      do j=1,it
16        sum=sum+func(x)
17        x=x+del
18      enddo
19      s=0.5*(s+(b-a)*sum/tnm)
20    end if
21    return
22  end subroutine trapzd
```

Горната процедура може да се разглежда като съществената част (building block) в интеграционните процедури които ще разгледаме по-долу. Най-примитивният начин по който можем да пресметнем интеграла I в интервала $[a, b]$ е като извършим `m`-брой стъпки на "подобряване" на решението `s`. Тогава можем просто да направим `m` последователни извиквания към процедурата `trapzd` като на всяка стъпка подобряваме оценката на интеграла `s`:

```
1  do 11 j=1,m
2    call trapzd(func,a,b,s,j)
3  enddo 11
```

Едно по-елегантно решение разбира се би било ако дефинираме предварително исканата от нас точност `EPS` и извикваме `trapzd` последователно докато задоволим зададената от нас точност. Нека разгледаме процедурата `qtrap`(`func`, `a`, `b`, `s`), която пресмята интеграла на външната функция `func`(`x`) в затворения интервал между точките `a` и `b`, като връща резултатът в променливата `s`:

```
1  subroutine qtrap(func,a,b,s)
2    double precision :: a,b,func,s
3    EXTERNAL func
4    double precision, parameter :: EPS=1.d-6
5    integer, parameter :: JMAX=20
6
7    integer :: j
8    double precision :: olds
9    olds=-1.d30
10   do j=1,JMAX
11     call trapzd(func,a,b,s,j)
12     if (j.gt.5) then
13       if (abs(s-olds).lt.EPS*abs(olds).or.(s.eq.0..and.olds.eq.0.)) return
14     endif
15     olds=s
16   enddo
17   write(0,*) 'too many steps in qtrap'
18   stop
19 end subroutine qtrap
```

Процедурата `qtrap` използва няколко предефенирани константи `EPS` и `JMAX`. Първата променлива `EPS` $=$ `1.d` $-$ `6` задава относителната точност при пресмятането на

интеграла `s`. Променливата `JMAX` задава максимално възможният брой итерации. В допълнение можем да представим програмен код който реализира метода на Симпсън използвайки итеративната формула 4.15. Процедурата `qsimp` има същият интерфейс както и `qtrap` и отново използва `trapzd` за междинните пресмятания.

```
subroutine qsimp(func,a,b,s)
implicit none

double precision :: a,b,func,s
double precision, PARAMETER :: EPS=1.d-6
integer, parameter :: JMAX = 20
external func
integer :: j
double precision :: os,ost,st

ost=-1.d30
os= -1.d30
do j=1, JMAX
  call trapzd(func,a,b,st,j)
  s=(4.*st-ost)/3.
  if (j.gt.5) then
    if (abs(s-os).lt.EPS*abs(os).or.(s.eq.0..and.os.eq.0.)) return
  endif
  os=s
  ost=st
end do
write(0,*) 'too many steps in qsimp'
stop
end subroutine qsimp
```

В случай на интеграл върху отворен интеграл можем да използваме процедурата `midpnt` разгледна в книгата *Числени рецепти*, която подобно на процедурата `trapezd` реализира итеративно пресмятане на интеграла I, използвайки уравнение 4.22.

```
subroutine midpnt(func,a,b,s,n)
  integer n
  double precision a,b,s,func
  external func
  integer it,j
  double precision ddel,del,sum,tnm,x

  if (n.eq.1) then
    s=(b-a)*func(0.5*(a+b))
  else
    it=3**(n-2)
    tnm=it
    del=(b-a)/(3.*tnm)
    ddel=del + del
    x=a+0.5*del
    sum=0.
    do j=1,it
      sum = sum + func(x)
      x = x+ddel
      sum = sum + func(x)
```

```
21          x = x + del
22        enddo
23       s=(s+(b-a)*sum/tnm)/3.
24      end if
25    return
26    end subroutine midpnt
```

В случая процедурата `qtrap` отново може да се използва като интерфейс към `midpnt`, заменяйки `trapzd`.

4.2 Примерни задача

1. Използвайки процедурите `trapzd` и `qtrap` напишете програмен код който пресмята следният интеграл:

$$\int_0^2 x^4 \log_e \left(x + \sqrt[2]{x^2+1}\right) dx = \frac{8 - 40\sqrt{5} + 480 \arcsin(2)}{75} = 8.15336 \tag{4.23}$$

2. Задачата за решаването на едномерното уравнение на Поасон описана в Глава 2.3 притежава и аналитично решение с помощта на функции на Грийн. Без да влизаме в подробност в доказателството на решението можем да разгледаме следният аналитичен израз който дава решение на задачата описана в Глава 2.3:

$$u(x) = (1-x) \int_0^x y f(y) dy + x \int_x^1 (1-y) f(y) dy \tag{4.24}$$

Където явният вид на функцията $f(y)$ се записва като:

$$f(y) = -y(y+3)e^y \tag{4.25}$$

Отново използвайки процедурите `trapzd` и `qtrap` напишете програмен код който пресмята интеграл описан в уравнение 4.24 за произволно x. Нарисувайте графика на функцията $u(x)$ и я сравнете с точното решение:

$$u(x) = x\,(x-1)\,e^x \tag{4.26}$$

3. Използвайки подходяща смяна на променливите и промените извикването към процедурата `trapzd` с това към `midpnt` в процедурата `qtrap`. Пресметнете следният интеграл по метода на трапеците

$$\int_0^\infty x e^{-x} dx = 1 \tag{4.27}$$

Глава 5

Търсене корени на нелинейни уравнения

В настоящата глава ще разгледаме една от най-често срещаните процедури а именно числено решаване на уравнения спрямо независимата променлива x, в най-общият случай уравненията имат членове и от двете страни, които обаче могат да бъдат преместени само от лявата страна:

$$f(x) = 0 \tag{5.1}$$

често тази задача се нарича и търсене на корен или корени на уравнението 5.1. В случай че имаме само една независима променлива x уравнението е едномерно. Когато независимите променливи са повече от една се означават като вектор от независими променливи $\mathbf{x} = \{x_1, x_2, \ldots, x_N\}$, като можем да се очакваме че съществуват повече от едно уравнения които трябва да бъдат удовлетворени. Според имплицитната функционална теорема решение на системата уравнения е възможно когато броят им M е равен на броят неизвестните $\{x_i\}$, т.е. $M \equiv N$, системата се записва в съкратена форма:

$$\mathbf{f}(\mathbf{x}) = \mathbf{0} \tag{5.2}$$

където $\mathbf{f} = \{f_1, f_2, \ldots, f_N\}$ е множество от N брой независими функции на променливите $\mathbf{x}$. Когато зависимостта от $\mathbf{x}$ е линейна по същество задачата за намирането на решението на системата се покрива от Глава 1. В най-общият случай на многомерна нелинейна система обаче съществуването на решение дори не е гарантирано. Трябва да спомена също че намирането на решение в многомерната задача е принципно различно от едномерният случай. Причината е в това че в едномерният случай решението или "корените" могат да бъдат ограничени и затворени върху едномерната ос, докато това не е вярно за многомерната задача. С изключение не линейното уравнение търсенето на корен е итеративна задача, т.е. стартираме от някаква начална точка $\mathbf{x}^0$ или "решение" след което последователно подобряваме оценката за корените на уравнението. Изборът на добра начална точка е много важен за успеха на търсенето.

Затова стойностите й трябва да бъдат внимателно подбрани анализирайки системата уравнения 5.2. Например в едномерният вариант на задачата винаги е добра идея да нарисувате предварително функционалната зависимост. Това ще помогне да определите визуално областите от стойности които са близо до вашето решение или решения. Важно е да помним че търсенето на корени на уравнение е числено нестабилна задача, възможно е поради лош избор на начална точка итеративната процедура да бъде разходяща, т.е да се отдалечаваме от истинското решение. Възможно е също така процедурата да не е сходяща просто защото не съществува решение. Представените по-долу алгоритми могат да се разглеждат като напътствия или най-добри практики за определен вид уравнения които можете да срещнете в работата си:

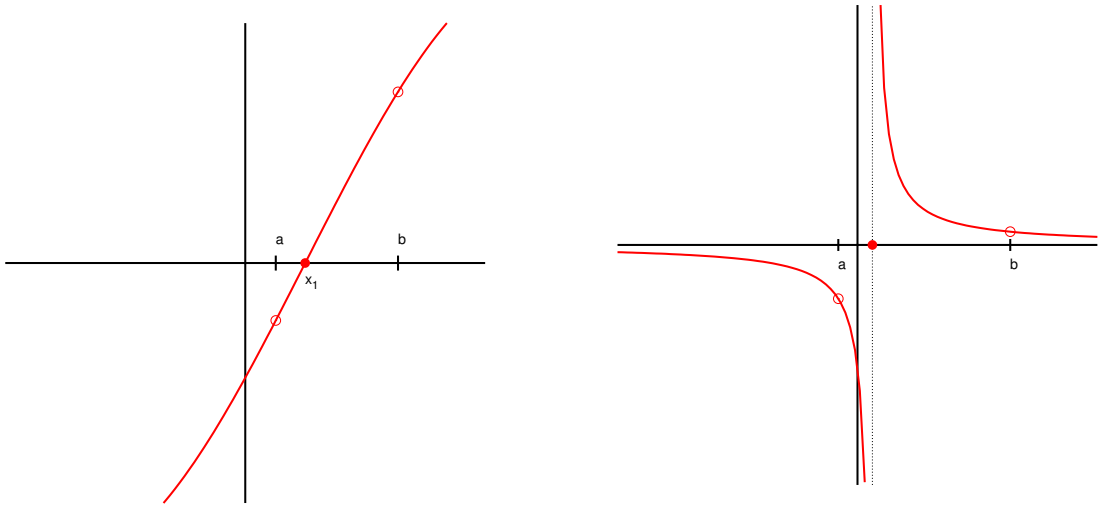
- Алгоритъм на Брент (Brent) е удачен в търсенето на корени на едномерно уравнение в случаите когато производната му не може (или е трудно) да се пресметне.
- Когато може да се пресметне първата производна на едномерна система се препоръчва метода на Нютон - Ралфсон (Newton - Ralphson)
- Метода на Халес е удобен в случаите когато могат да се пресметнат и втората производна на уравнението. (Halles)
- В случай, че търсим корени на полиноми се препоръчва методът на Лагер (Laguerre)
- И най-накрая в многомерният случай - най-лесният метод е този на Нютон - Ралфсон (Newton - Ralphson).

5.1 Търсене корени на едномерно уравнение

Нека първо разгледаме задачата за намирането решение на едномерно уравнение 5.1. Първата стъпка от търсенето корен на нашето уравнение, както споменахме по-горе е да нарисуваме $f = f(x)$ като функция на x, графиката на функцията ни дава важна информация за началният интервал на търсене.

5.1.1 Разделяне на корените на едномерно уравнение и Бисекция

В случай на едномерно уравнение е възможно търсеното от нас решение да бъде "ограничено" в интервал между две точки a и b от реалната ос. Ще казваме, че един корен е разделен ако за интервала $[a, b]$ стойностите на функцията f в граничните точки $f(a)$ и $f(b)$ имат противоположни знаци. Ако функцията $f(x)$ е непрекъсната тогава съществува поне един корен фигура 5.1а. В случай, че функцията е прекъсната но ограничена, тогава може да съществува особена точка, в която функцията търпи "скок", който пресича абцисната ос. Тези точка от числена гледна точка може да интерпретира също като корен, фигура 5.1б. Нека разгледаме един прост алгоритъм взаимстван



(а) Съществува единствено решение в интервала ограничено в интервала $[a, b]$.

(б) Функцията $f(x)$ няма корени в интервала на делене въпреки че са изпълнени граничните условия.

Фигура 5.1: Графика на зависимостти $y = f(x)$ и интервала на делене $[a, b]$

от процедурата `zbrac` описан в книгата "*Числени рецепти*" [8], който търси разделен интервал $[a, b]$ за функцията $f(x)$ започвайки от точките x_1 и x_2, примерен псевдо код.

Изходен код реализиращ алгоритъма на делене написан на програмният език Fortran 90 е показан в допълнение А.4. Съществува и друг подход за търсене на интервали на делене на функцията $f(x)$ при които търсенето се извършва вътре в в областта $[a, b]$ като ширината на интервала не се променя. Самият интервал се разделя на множество m-брой под-интервали с еднаква дължина като се избират само тези които делят функцията $f(x)$, подобен алгоритъм е описан в процедурата `zbrak` в допълнение А.5. Веднъж открили интервал на делене, можем да приложим алгоритъм, който търси корени в него интервал подобно на двоично търсене, метода последователно разделя интервала на две части и проверява в кой от двата подинтервала функцията си сменя знака и т.н. след n на брой итерации коренът ще се намира в интервал с дължина ϵ_n, $\epsilon_n = \epsilon_0/2^n$ и $\epsilon_{n+1} = \epsilon_n/2$. Имайки тази рекурентна зависимост можем да направим оценка за броя на итерации, необходими да се достигне интервал с размер на машинният ϵ_m а именно именно:

$$n = \log_2 \frac{\epsilon_0}{\epsilon_m} \tag{5.3}$$

Този подход е сигурен и винаги ще намира решение в случай че интервала $[x_1, x_2]$ е разделен. В случай че интервала съдържа повече от един корен, алгоритъма гарантира че поне един от тях ще бъде намерен, а когато в интервала има сингулярно поведение на функцията $f = f(x)$ точката на неопределеност ще бъде намерена. Важ-

```
/* Процедура zbrac псевдо код                                          */
/* променливата FACTOR контролира мащаба на преоземряване на
   интервала на делене [x1, x2]                                        */
FACTOR = 1.6
/* променливата MAXIT определя максималният брой повторения преди
   приключване процедурата по търсене на интервала на делене          */
MAXIT = 50
f1 = f(x1)
f2 = f(x2)
for i = 1 ... MAXIT do
    /* на всяка итерация проверяваме дали знаците на функцията в точките
       x1 и x2 се различават                                           */
    if f1 * f2 < 0 then
        return true
    end
    /* В случай че интервала не е разделен координатите x1 или x2 се
       преместват в посока намаляване абсолютната стойност на функцията
       f(x)                                                            */
    if abs(f1) < abs(f2) then
        x1 = x1 + FACTOR * (x1 - x2)
        f1 = f (x1)
    else
        x2 = x2 + FACTOR * (x2 - x1)
        f2 = f (x2)
    end
    return false
end
```

но е да обсъдим и интервала или критерия за сходимост на алгоритъма на двоично делене или бисекция доколко практически е невъзможно да се открие точката x за която $f(x) = 0$ с този метод. Интервал от порядъка на $\approx 10^{-10}$ може да бъде напълно приемлив в случайите когато корените се намират около 1.0, но ще бъде невъзможно да се достигне в случаите когато корените се намира в диапазона на 10^{26} поради факта че гъстотата на двоичното представяне на числата върху реалната ос не е равномерно. Често в практиката един добър критерий за определяне минималната големина на интервала на делене е $\epsilon_m(x_1 + x_2)/2$, където ϵ_m е машинната нула а точките x_1 и x_2 представят границите на първоначалният интервал на търсене. По-долу можем да разгледаме един примерен алгоритъм взаимстван от функцията `zrtbis` описана в книгата "*Числени рецепти*", тя приема като входни параметри функцията $func$, границите на началният интервал x_1 и x_2 определни с някой от алгортмите за разделяне на интервала на търсене описани по-горе и точността или критерия за сходимост `xacc`, функцията връща стойността на x в средата на интервала $x_{1/2}$:

```
/* Псевдо код реализираш двоично търсене в предварително разделен
   интервал [x_1, x_2]. Резълтатът се връща в променливата rtg ако не се
   достигне максималният брой итерации JMAX.                          */
f_{1/2} = f(x_2);
f = f(x_1);
if f ⋆ f_{1/2} ≥ 0 then
 | return ГРЕШКА: интервалът не е разделен
end
if f < 0 then
 | rtbis = x_1;
 | dx = x_2 − x_1;
else
 | rtbis = x_2;
 | dx = x_1 − x_2;
end
for j = 1, ..., MAXIT do
 | dx = dx/2;
 | x_{1/2} = rtbis + dx;
 | f_{1/2} = f(x_{1/2});
 | if f_{1/2} ≤ 0 then
 |  | rtbis = x_{1/2};
 | end
 | if abs(dx) < xacc OR f_{1/2} = 0 then
 |  | return x_{1/2}
 | end
end
return ГРЕШКА: Максималният брой итерации достигнат
```

5.2 Метод на секущата и “regula falsi” (фалшива позиция)

В случаите когато функцията $f(x)$ е достатъчно гладка е възможно да се направи друга оценка за границата на делене на интервала, в който се намира “нула” като се предположи, че функцията може да се разглежда като линейна в него. Ще разгледаме два алгоритъма които търсят решението итеративно като корена се апроксимира с точката в която линейното приближение на функцията $f(x)$ пресича абцисната ос, като след всяка стъпка една от граничните точки се подменя с новата точка. Единствената разлика между избора на следващата точка която затваря интервала на търсене е че при метода на секущата следващата точка не е задължително да дели интервала докато при “regula falsi” метода точките се избират по начин че да делят интервала на търсене.

Степента на сходимост в случай на достъчно гладка функция за метода на секу-

щата може да се представи като рекуретна зависимост на интервала на делене която има степенна зависимост, позната още като *златното правило*:

$$\lim_{k \leftarrow \infty} |\epsilon_{k+1}| \approx const \times |\epsilon_k|^{1.618...} \tag{5.4}$$

недостък на метода на секущата е че доколко интервала на търсене не дели задължително някой от корените на функцията $f(x)$ е възможно алгоритъма на търсене да бъде разходящ ако функцията не е достатъчно гладка например. По-долу ще представим алгоритмочно двата метода на търсене. За начало ще разледаме функцията `rtsec` която търси корена на функцията `func`, за даден начален интервал $[x_1, x_2]$ и избрана точност `xacc`:

`/* Метод на секущата */`
$f_L = f(x_1)$;
$f = f(x_2)$;
if $abs(f_L) < abs(f)$ **then**
  $rtsec = x_1$;
  $x_L = x_2$;
  разменяме стойностите на x_L и $rtsec$ така че да изберем текущо решение $rtsec$ с по-ниска функционална стойност;
else
  $x_L = x_1$;
  $rtsec = x_2$;
end
for *j = 1, ..., MAXIT* **do**
  $dx = (x_L - rtsec) \star f / (f - f_L)$;
  $x_L = rtsec$;
  $f_L = f$;
  $rtsec = rtsec + dx$;
  $f = f(rtsec)$;
  if $abs(dx) < xacc$ *OR* $f == 0$ **then**
    return $rtsec$
  end
end
return *ГРЕШКА: Максималният брой итерации достигнат*

алгоритъмът по-горе предполага следната линейна зависимост за функцията $f(x)$:

$$f(x) = f + \frac{f_L - f}{x_L - rtsec}(x - rtsec)$$

което уравнение се решава за $f(x') = 0$, следователно:

$$x' = rtsec + f\frac{x_L - rtsec}{f - f_L} = x_L + dx$$

Сега остава да разгледаме псевдо кода на алгоритъма който реализира метода на фалшива позиция:

```
/* "regula falsi" метод                                                    */
f_L = f(x_1);
f_H = f(x_2);
if f_L ⋆ f_H > 0 then
  | return ГРЕШКА: Интервалът трябва да бъде ограничен
end
if f_L < 0 then
  | x_L = x_1;
  | x_H = x_2;
else
  | x_L = x_2;
  | x_H = x_1;
  | Размени стойностите f_L и f_H
end
dx = x_H − x_L;
for j = 1, ..., MAXIT do
  | rtf = x_L + dx ⋆ f_L/ (f_L − f_H);
  | f = f(rtf);
  | if f < 0 then
  |   | del = x_L − rtf;
  |   | x_L = rtf;
  |   | f_L = f;
  | else
  |   | del = x_H − rtf;
  |   | x_H = rtf;
  |   | f_H = f;
  | end
  | dx = x_H − x_L;
  | if abs(del) < xacc OR f == 0 then
  |   | return rtf
  | end
end
return ГРЕШКА: Максималният брой итерации достигнат
```

5.3 Алгоритъм на Брент

Алгоритъма на Брент комбинира разделянето на интервали, бисекция и обратна квадратична интерполация за да открие корените, доколко първите два метода бяха разгледани вече нека обърнем внимание на търсенето на корени с помощта на обратна кубична интерполция, в този метод подобно на метода на секущата се използва приближение на функцията $f(x)$ с полином, в случая от втора степен. Обратната квадратична интерполация предполага познаването стойността на функцията $f(x)$ в три

точки в интервала $[a, b]$, нека приемем че това са двере крайните точки a, b и една точка c, вътре в него $\{a, f(a)\}$, $\{b, f(b)\}$ и $\{c, f(c)\}$. Чрез тези три двойки числа можем да интерполираме функцията $f(x)$ с помощта на полином от втора степен:

$$\begin{aligned}\overline{f}\,(x) = &\frac{[y-f(a)][y-f(b)]c}{[f(c)-f(a)][f(c)-f(b)]} + \frac{[y-f(b)][y-f(c)]a}{[f(a)-f(b)][f(a)-f(c)]} + \\ &+ \frac{[y-f(c)][y-f(a)]b}{[f(b)-f(c)][f(b)-f(a)]},\end{aligned} \tag{5.5}$$

ако положим $\overline{f}(x) = 0$ и решим уравнението 5.5 спрямо независимата променлива x ще получим оценка за корена на уравнението: $x = b + P/Q$, $R = f(b)/f(c)$, $S = f(b)/f(a)$, $T = f(a)/f(c)$, $P = S[T(R-T)(c-b) - (1-R)(b-a)]$, $Q = (T-1)(R-1)(S-1)$. Процедурата може да се повторя итеративно докато не се намери точната стойност на корена на уравнението или не се доближим до него с предварително желана от нас точност.

5.4 Метод на Нютон - Ралфсон. Едномерен случай.

Широко разпространен метод, който работи добре и за многомерният случай. Изисква пресмятането на $f'(x)$. По същество метода стартира в точка, от която се екстраполира права линия, която пресича абсцисата и използва нея като следващо приближение. Методът може да се получи от Тейлърово развитие в ред на функцията около точката x в околност Δx.

$$f(x + \Delta x) \simeq f(x) + f'(x)\Delta x + \frac{f''(x)\Delta x^2}{2} + \ldots \tag{5.6}$$

Ако се ограничи само да първите два члена в развитието на ред на Тейлър и предположим, че $f(x + \Delta x) = 0$ можем да решим уравнението спрямо "поправката" Δx

$$\Delta x = -\frac{f(x)}{f'(x)} \tag{5.7}$$

методът е итеративен, т. е. за да намерим решението на уравнение 5.2 ние стартираме от някоя начална точка x_0 и последователно "подобряваме" решението с поправка Δx, така че $x_{i+1} = x_i + \Delta x$, докато не достигнем критерия ни за сходимост $|f(x_{i+1}) - f(x_i)| \leq \epsilon$. За бъде сходящ алгоритъма има нужда да стартира от начална точка x_0 която се намира сравнително близо до тъсенето от нас решение. Затова е подходящо метода да се комбинира с някоя друга схема при която предварително е ограничен интервалът на търсене.

5.5 Метод на Нютон - Ралфсон. Многомерен случай.

Можем да развием числениият метод на Нютон - Ралфсон за многомерният случай като разгледме за простота двумерният случай на система от две уравнения на две променливи:

$$f_1(x_1, x_2) = 0 \tag{5.8}$$
$$f_2(x_1, x_2) = 0 \tag{5.9}$$

нека представим горните две функции в ред на Тейлър около точката $\mathbf{x} = (x_1, x_2)$ за малки отклонения $\mathbf{dx} = (dx_1, dx_2)$ на функциите $\mathbf{f} = (f_1(x), f_2(x))$ интересувайки се само от линейната част е представането на реда:

$$f_1(x_1 + dx_1, x_2 + dx_2) = f_1(x_1, x_2) + \partial_{x_1} f_1 dx_1 + \partial_{x_2} f_1 dx_2 + \dots \tag{5.10}$$
$$f_2(x_1 + dx_1, x_2 + dx_2) = f_2(x_1, x_2) + \partial_{x_1} f_2 dx_1 + \partial_{x_2} f_2 dx_2 + \dots \tag{5.11}$$

горното уравнение може да бъде систематизирано в матрична форма

$$\mathbf{f}\left(\mathbf{x} + \mathbf{dx}\right) = \mathbf{f}\left(\mathbf{x}\right) + \mathbf{J} \cdot \mathbf{dx} + \dots \tag{5.12}$$

матрицата $\mathbf{J}$ от първите производни на функциите $\mathbf{f}$ се нарича Якобиан и тя може да се запише като

$$\mathbf{J} = \begin{bmatrix} \dfrac{\partial f_1}{\partial x_1} & \dfrac{\partial f_1}{\partial x_2} \\ \dfrac{\partial f_2}{\partial x_1} & \dfrac{\partial f_2}{\partial x_2} \end{bmatrix} \tag{5.13}$$

подобно на едномерният случай можем да преположим че търсим това $\mathbf{dx}$ за което $\mathbf{f}\left(\mathbf{x} + \mathbf{dx}\right) = \mathbf{0}$, в който се случва следното матрично уравнение

$$\mathbf{f}\left(\mathbf{x} + \mathbf{dx}\right) = \mathbf{0} = \mathbf{f}\left(\mathbf{x}\right) + \mathbf{J} \cdot \mathbf{dx} \tag{5.14}$$

Подобно на уравнение 5.7 можем да запишем израза за "корекцията" $\mathbf{dx}$. В случая е необходимо да решим горното линейно уравнение 5.14:

$$\mathbf{dx} = -\mathbf{J}^{-1} \cdot \mathbf{f}\left(\mathbf{x}\right) \tag{5.15}$$

5.6 Интерфейс към програми за символни пресмятания

В случаите когато търсим еднократно корени на дадена функция $f(x)$ за предпочитане е да се използва интерфейс към програми за символни пресмятания като

Mathematica, Maple, Matlab и *Octave*. Използването на външни библиотечни функции за търсене на корени е оправдано в случаите когато търсим многократно корени на функция на една или много променливи или търсенето е част от друг числен алгоритъм. Сега нека представим извикванията към програмите за символни пресмятания които реализират търсене на корени:

1. *Mathematica* - Функцията `FindRoot`$[\mathtt{f}, \{\mathtt{x}, \mathtt{x_0}\}]$, търси корените на функцията `f` за независимата променлива `x` стартирайки от начална точка $\mathtt{x_0}$
2. *Maple* - `Roots`$(\mathtt{f(x)}, \mathtt{x = a..b}, \mathtt{opts})$, търси корените на функцията `f`$(\mathtt{x})$ в интервала $[\mathtt{a}, \mathtt{b}]$, като `opts` са допълнителните параметри на търсене които не са задължителни
3. *Matlab* - Изразът $\mathtt{x} = \mathtt{fzero}(\mathtt{fun}, \mathtt{x_0})$, връща корените `x` на функцията `fun` използвайки функцията `fzero`, като търсенето стартира от началната точка $\mathtt{x_0}$.
4. *Octave* - Използва същият интерфейс към функцията `fzero`, както е случая на *Matlab*.

5.7 Примерни задачи.

1. Разгледайте програмата `rootfind.f90` която реализира търсенето на корените на квадратното уравнение:

$$x^2 + 200x - 0.000015 = 0 \tag{5.16}$$

   като използва предварително разделяне на интервала $[\mathtt{x1}, \mathtt{x2}]$ въведен от потребителя с помощта на процедурата `zbrac`, след което използва процедурата `zbrent` за откриe корена с абсолютна точност `tol`. Корените на уравнение 5.16 са съответно $\mathtt{7.5e-8}$ и $-\mathtt{200.000000075}$. Опитайте се да модифицирата шаблонният файл `rootfind_newrft.f90` така че да реализирате търсене с помощта на метода на Нютон-Ралфсон, като изпозлвате процедурата `rtnewt` .
2. Използвайки като заготовка програмният код от предходната задача намерете корените на следната функция:

$$f(x) = \frac{(2x-3)(x+3)}{x(x-2)}$$

3. Използвайки метода на Нютон-Ралфсон за многомерният случай описа в уравнение 5.15. Намерете корените на следната система уравнения:

$$f_1(x_1, x_2) = x_1 + x_2 - x_1 x_2 + 2 \tag{5.17}$$
$$f_2(x_1, x_2) = x_1 * exp(-x_2) - 1 \tag{5.18}$$

Глава 6

Статистическа обработка на информация

Глава 7

Минимизиране на функция

Нека разгледаме функция f на една или много независими променливи $\mathbf{x}$, нашата цел е да намерим тези стойности на променливата $\mathbf{x}$, за които функцията f притежава максимална или минимална стойност. Доколко задачата за максимизиране на f може да се сведе до минимизиране на $-f$. За нас ще бъде достатъчно да разгледаме числени методи за търсене минимум на функцията f. Екстремумът на една функция може да бъде или глобален (наистина най-голямата или най-малката стойност на функцията) или локален (най-голямата или най-малката стойност на функцията в ограничена област от дефиниционното пространство). Намирането на глобален минимум в общият случай е трудна задача и няма еднозначна процедура. В общият случай намирането на глобален минимум се свежда до последователно намиране на "локални" минимуми:

1. Търсене на локални минимуми от различни начални стойности, след което се избират "най-добрите" минимуми и максимуми, т.е. тези с по-ниски стойности на минимизираната функция f.

2. При вече намерен локален минимум състоянието се пертурбира леко, след което се търси ново състояние

Подобно на търсенето на корен на система уравнения тук отново методите се разделят принципно в случаите на едномерна (функция на една променлива) и многомерна задача (функция на много променливи). В случаите на минимизиране на функция на една променлива ще разгледаме: *Метод на златното сечение* и *Метод на Брент*. Докато за многомерна минимизация: *Метод на спускането*, *Метод на Пауъл* и *Метод на спрегнатите градиенти*

7.1 Минимизиране на функция на една променлива. Едномерна минимизация

7.1.1 Разделяне на интервала

В глава 5 раздел 5.1.1 вече разгледахме техниките за разделяне на дефиниционният интервал в случай на търсене на корен на функция f. Тогава беше достатъчно да намерим интервал $[a, b]$ в който функцията си сменя знака, т.е. $f(a)f(b) \leq 0$. В случай че търсим минимум на функция ние ще търсим три точки $a < b < c$ за които е изпълнено условието $f(b)$ да бъде по-малка от $f(a)$ и $f(c)$. В случаите когато функцията е непрекъсната това условие гарантира също ствуването на поне един локален минимум в интервала $[a, c]$. Процедурата за разделяне на интервала е съществена за алгоритъма на Брент. По-долу сме представили примерен код който разделя интервала на функцията `f(x)`

7.1.2 Метод на златното сечение

Минимумът на една функция в едномерното пространство може да се раздели между три точки $a < b < c$, в който случай функцията f притежава минимум в интервала $[a, c]$ с изключение на случаите когато f е сингулярна в интервала. Оценка за точността на решението не е от порядъка на ϵ. Нека разгледаме минимума в точката b, в пред на Тейлър:

$$f(x) \simeq f(b) + \frac{1}{1!}f'(b)(x-b) + \frac{1}{2!}f''(b)(x-b)^2 \tag{7.1}$$

вторият член (първата производна) в горното уравнение може да се игнорира в случая. Тогава членът пропорционален на втората производна ...

$$\begin{aligned} &\frac{1}{2}f''(b)(x-b)^2 \\ &\qquad |x-b| \;<\; \sqrt{\epsilon}b\sqrt{\frac{2|f(b)|}{b^2 f''}} \end{aligned}$$

7.1.3 Метод на Брент

7.2 Минимизиране на функция на много променлива. Многомерна минимизация

7.2.1 Метод на спускането

7.2.2 Метод на Пауъл

7.2.3 Метод на спрегнатите градиенти

Глава 8

Напасване на функции

8.1 Общи положения

8.2 Случай на линейно напсване. Линейна регресия

8.3 Нелинейно напсване

8.4 Примерни задачи

Глава 9

Обикновенни диференциални уравнения

9.1 Общи положения

Обикновенните диференциални уравнения са съществена част от математичният апарат на физиката. В настоящата глава ще разгледаме числените методи за решаване на системи обикновенни диференциални уравнения (ОДУ) в случаите на задача на Коши и граничната задача на фон Нойман. В общият случай една система обикновенни диференциални уравнения може да се дефинира явно:

$$y^{(n)}(x) = F\left(x, y(x), y'(x), y''(x), \ldots, y^{(n-1)}(x)\right) \tag{9.1}$$

или по не-явен начин:

$$F\left(x, y(x), y'(x), y''(x), \ldots, y^{(n-1)}(x), y^{(n)}(x)\right) = 0 \tag{9.2}$$

В разглежданията по-долу ще смятаме че системата ОДУ е представена в явен вид. Известно е също така от общата теория на ОДУ че всяка система ОДУ от ред n може да се представи като разширена система от n-брой ОДУ-та от 1-ви ред с подходяща функционална замяна. Например нека разгледма явно определена система ОДУ от n-ти ред 9.1. Ще изберем следната функционална замяна

$$y'(x) = y_1(x) \tag{9.3а}$$
$$y''(x) = y_2(x) \tag{9.3б}$$
$$y'''(x) = y_3(x) \tag{9.3в}$$
$$\vdots \quad \vdots \tag{9.3г}$$
$$y^{(n-1)}(x) = y_{n-1}(x) \tag{9.3д}$$
$$\tag{9.3е}$$

тогава можем да запишем уравнението 9.1 може да се запише като система от n на брой обикновенни диференциални уравнения от първи ред

$$y'(x) = y_1(x) \tag{9.4а}$$
$$y_1'(x) = y_2(x) \tag{9.4б}$$
$$y_2'(x) = y_3(x) \tag{9.4в}$$
$$\vdots \quad \vdots \tag{9.4г}$$
$$y_{(n-1)}'(x) = F\left(x, y(x), y_1(x), y_2(x), \ldots, y_{n-1}(x)\right) \tag{9.4д}$$

За простота на представянето можем да запишем лявата и дясната част на системата уравнения 9.4 с помщта на два обобщени вектор функции $\mathbf{y} = \{y(x), y_1(x), \ldots, y_{n-1}(x)\}$ и $\mathbf{f} = \{y_1(x), y_2(x), \ldots, F(x, y(x), y_1(x), \ldots, y_{n-1}(x))\}$ с което можем да улесним записа на горното уравнение

$$\mathbf{y}'(x) = \mathbf{f}\left(x, \mathbf{y}(x)\right) \tag{9.5}$$

До колко произволна система ОДУ може да бъде записана във вида на уравнение 9.5 числените методи разгледани в тази глава ще предполагат системи ОДУ от първи ред. Всяка система ОДУ дефинирна фамилия от решения които зависят от граничните условия. Според тази класификация могат да се дефинират два типа задачи

- Задача на Коши - когато познаваме стойностите $\mathbf{y}_i(x_s)$ за всички $i = 1, \ldots, n$ за дадена стойност на независимата променлива x_s и търсим "еволюцията" на функционалният вектор $\mathbf{y}$ докато достигнем някаква крайна точка x_f на независимата променлива x.
- Гранична задача - в случаите когато стойностите на функцията $\mathbf{y}$ са зададени за повече от една точки (обикновенно това са граничните такива x_s и x_f).

Числените методи които ще разгледаме в тази глава ще организираме според типа на граничната задача

9.2 Задача на Коши

9.2.1 Метод на Ойлер

Методът на Ойлер се разглежда като най-простият интеграционен алгоритъм за решаване на обикновенни диференциални уравнения. Неговата "локална" грешка за една интеграционна стъпка е от порядъка на $O(h^2)$, което го прави неприложим за големи системи ОДУ или случаите в които желаем висока точност на численото решение или се търси решение за голям интервал на независимата променлива x. За да получим формулата на Ойлер нека разгледаме експлицитна система ОДУ от първи ред

$$\mathbf{y}' = \mathbf{f}(x, \mathbf{y}) \tag{9.6}$$

Решението $\mathbf{y} = \mathbf{y}(x)$ което търсим ще бъде дискретизирано върху решетка от стойности x_n на независимата променлива x, където $n \in \mathbb{Z}$ и всеки x_n са равно-отдалечени със стъпка h, където $h = x_{n+1} - x_n$ за произволно n. Стъпката h се нарича стъпка на интегриране и тя играе важна роля за точността на числено интегриране на уравнението 9.6. За всяка дискретна стойност на независимата променлива x_n ние можем да дефинираме решение $\mathbf{y}_n = \mathbf{y}(x_n)$. Важно е да отбележим че този подход ни позволява да намерим **приближено** решение на нашата система ОДУ което може да се обобщи в множеството от стойности $\{\mathbf{y}_n\}$ за всички x_n. Стойностите на $\mathbf{y}_n$ между две последователни интегрирания не са известни но могат да бъдат интерполирани с някой от методите разглеждани в Глава 3. Съответно функцията на първите производни $\mathbf{y}'(x)$ също може да се дискретизира върху решетката от стойности x_n, $\mathbf{y}'_n = \mathbf{y}'(x_n)$. Така уравнение 9.6 приема дискретна форма

$$\mathbf{y}'_n = \mathbf{f}(x_n, \mathbf{y}_n) \tag{9.7}$$

Разглеждайки уравнение 9.7 можем да кажем че се намираме на една стъпка от формулата на Ойлер. Това което ни остава е да запишем първата производна на функцията $\mathbf{y}(x)$ в крайни разлики за произволна точка x_n съгласно уравнение 2.6 които разгледхме в Глава 2.

$$\frac{d}{dx}\mathbf{y}(x)\bigg|_{x=x_n} = \mathbf{y}'_n = \frac{\mathbf{y}_{n+1} - \mathbf{y}_n}{h} + O(h) \tag{9.8}$$

изразявайки $\mathbf{y}_{n+1}$ чрез $\mathbf{y}_n$ и $\mathbf{y}'_n$ от горното уравнение достигаме до уравнението на Ойлер

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\mathbf{y}'_n + O(h^2) = \mathbf{y}_n + h\mathbf{f}(x_n, \mathbf{y}_n) + O(h^2) \tag{9.9}$$

доколко остатъчният член $O(h^2)$ в уравнение 9.9 е от порядъка на квадрата на стъпката на интегриране h^2 можем да заключим че "локалната" грешка от интегрирането ще бъде в същите граници. В случай че познаваме частните производни на функцията $\mathbf{f}(x, \mathbf{y})$, $\partial\mathbf{f}/\partial x$ и $\partial\mathbf{f}/\partial\mathbf{y}$ можем да повишим точността на метода на Ойлер до ред $O(h^3)$. За целта нека представим решението $\mathbf{y}(x)$ за произволно избрана от нас стойност на независимата променлива x_n в ред на Тейлър предполагайки че стъпката на дискретизация h е достатъчно малка

$$\mathbf{y}_{n+1} = \mathbf{y}(x_n + h) = \mathbf{y}_n + h\mathbf{y}'_n + \frac{1}{2}h^2\mathbf{y}''_n + O(h^3) \tag{9.10}$$

взимайки предвид уравнение 9.6 можем да изразим втората производна на функцията $\mathbf{y}(x)$ като

$$\frac{d}{dx}\mathbf{y}'(x) = \frac{d}{dx}\mathbf{f}(x, \mathbf{y}) = \frac{\partial}{\partial x}\mathbf{f}(x, \mathbf{y}) + \mathbf{f}(x, \mathbf{y})\frac{\partial}{\partial y}\mathbf{f}(x, \mathbf{y}) \tag{9.11}$$

комбинирайки уравнения 9.10 и 9.11 стигаме до методът на Ойлер от по-висок ред

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\mathbf{f}(x_n, \mathbf{y}_n) + \frac{1}{2}h^2 \left[\frac{\partial}{\partial x}\mathbf{f}_n + \mathbf{f}_n \frac{\partial}{\partial y}\mathbf{f}_n \right] + O(h^3) \tag{9.12}$$

като единствен недостатък на това уравнение може да се изтъкне необходимостта да се познават и съответно да се пресмятят производните $^{\partial \mathbf{f}}/_{\partial x}$ и $^{\partial \mathbf{f}}/_{\partial \mathbf{y}}$ което може да усложни програмната част при реализирането на метода. Повечето числени алгоритми изискват познаването само на функцията $\mathbf{f} = \mathbf{f}(x, \mathbf{y})$.

9.2.2 Многостъпков метод

Друг подход за намаляване грешката от интегрирането е използването на „информация" за решението от предходни стъпки на интегриране което се реализира в многостъпковите методи. Нека интегрираме уравнение 9.6 в интервала на независимата променлива $[x_n, x_{n+1}]$ тогава можем да запишем

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \int_{x_n}^{x_{n+1}} \mathbf{f}(x, \mathbf{y}(x))dx \tag{9.13}$$

вторият член от дясната страна на уравнение 9.13 в общият случай не е известен доколко ние не познаваме явният вид на функцията $\mathbf{f} = \mathbf{f}(x)$. Възможно е обаче да използваме приближение на $\mathbf{f}$ в интервала $[x_n, x_{n+1}]$. Един начин да получим такова приближение е чрез екстраполиране на функцията f от предходен интервал на интегриране. Например нека разгледаме интерала $[x_{n-1}, x_n]$ включващ само две предходни точки на интегриране x_{n-1} и x_n. В този предходен интервал функцията $\mathbf{f} = \mathbf{f}(x)$ може да се интерполира чрез полином на Лагранж от първи ред с помощта на следният израз

$$\mathbf{f}(x) = \frac{x - x_{n-1}}{h}\mathbf{f}_n - \frac{x - x_n}{h}\mathbf{f}_{n-1} + O(h^2) \tag{9.14}$$

ако заместим израза 9.14 в уравнението 9.6 можем да получим интеграционната формула от първи ред за многостъпковата схема. Но преди това нека се улесним като пресметнем двата подинтегрални члена имайки предвид $x_m - x_n = (m - n)h$

$$\int_{x_n}^{x_{n+1}} \frac{x - x_{n-1}}{h}\mathbf{f}_n dx = \frac{\mathbf{f}_n}{h}\frac{(x - x_{n-1})^2}{2}\bigg|_{x_n}^{x_{n+1}} = \frac{\mathbf{f}_n}{2h}\left[4h^2 - h^2\right] = \frac{3}{2}h\mathbf{f}_n \tag{9.15}$$

подобно

$$\int_{x_n}^{x_{n+1}} \frac{x - x_n}{h}\mathbf{f}_{n-1} dx = \frac{\mathbf{f}_{n-1}}{h}\frac{(x - x_n)^2}{2}\bigg|_{x_n}^{x_{n+1}} = \frac{\mathbf{f}_{n-1}}{2h}\left[h^2 - 0\right] = \frac{1}{2}h\mathbf{f}_{n-1} \tag{9.16}$$

обединявайки урванения 9.15 и 9.16 с 9.13 получаваме окончателният израз за многостъпков метод от първи ред

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\left[\frac{3}{2}\mathbf{f}_n - \frac{1}{2}\mathbf{f}_{n-1}\right]] + O(h^3) \tag{9.17}$$

Използвайки интерполационни схеми от по-висок ред можем да подобрим точността на инеграционната схема като включим други вече пресметнати стойности на функцията $\mathbf{f}$. Например без да навлизаме в технически детайли можем да запишем интеграционната формула на алгоритъма на Адамс-Бишфорт която се възползва от полиномиална интерполация от трети ред (кубичен полином)

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{h}{24}\left[55\mathbf{f}_n - 59\mathbf{f}_{n-1} + 37\mathbf{f}_{n-2} - 9\mathbf{f}_{n-3}\right] \tag{9.18}$$

Многостъпковите методи позволяват да се повиши точността на численото интегриране на сравнително малка цена, като единствен недостатък на този подход може да се изтъкне проблемът със стартирането на интеграционната процедура което очевидно изисква познаването на функцията y в няколко точки предварително. Този проблем може да бъде избегнат ако тези начални решения бъдат намерени с помощта на друг интеграционен алгоритъм от по-нисък ред.

9.2.3 Не-явни методи

Не-явните интеграционни схеми експлоатират случаите в които ние можем да „познаваме" някакво приближено решение $\mathbf{y}_{n+1}$ което приближение да използваме за подобряване точността на нашето пресмятане. Например нека разгледаме междинната точка $x_{n+1/2} = x_n + h/2$ която се намира в средата на произволен интеграционен интервал $[x_n, x_{n+1}]$ за тази междинна точка можем да запишем следните две уравнения

$$\left.\frac{d}{dx}\mathbf{y}\right|_{x=x_{n+1/2}} = \mathbf{f}(x_{n+1/2}, \mathbf{y}_{n+1/2}) \approx \frac{1}{2}\left[\mathbf{f}(x_n, \mathbf{y}_n) + \mathbf{f}(x_{n+1}, \mathbf{y}_{n+1})\right] \tag{9.19}$$

$$\left.\frac{d}{dx}\mathbf{y}\right|_{x=x_{n+1/2}} = \mathbf{y}'_{n+1/2} = \frac{\mathbf{y}_{n+1} - \mathbf{y}_n}{h} + O(h^3) \tag{9.20}$$

приравнявайки двата израза 9.19 и 9.20 стигаме до не явният израз за решението $\mathbf{y}_{n+1}$ в точката x_{n+1}

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{h}{2} + \left[\mathbf{f}(x_n, \mathbf{y}_n) + \mathbf{f}(x_{n+1}, \mathbf{y}_{n+1})\right] + O(h^3) \tag{9.21}$$

Тъй-като решението $\mathbf{y}_{n+1}$ присъства и в дясната част на равенството 9.21 то не може бъде пресметнато директно. Без да навлизаме в детайли ще споменем че са възможни два подхода за намирането му

- Рекурсивно - Стартирайки от някакво приближено решение $\mathbf{y}^0_{n+1}$ и замействайки го в дясната част на уравнение 9.21 получаваме следващото приближение $\mathbf{y}^1_{n+1}$ и т.н., докато разликата между решенията $\mathbf{y}^i_{n+1}$ и $\mathbf{y}^{i+1}_{n+1}$ за две последователни

итерации не е стане по-малко от предварително избрано малко число ϵ, $|\mathbf{y}_{n+1}^{i+1} - \mathbf{y}_{n+1}^{i}| < \epsilon$

- В комбинация с друг интеграционен метод - Може да се използва решение $\mathbf{y}_{n+1}$ намерено чрез друг явен метод което да се замести в уравнение 9.21

9.2.4 Комбиниран много стъпков неявен моетод. Схема на Адамс - Мултън

Идеята зад тази интеграционна схема се състои в това под-интегралната функция в израза 9.13 да се интерполира (за разлика от секция 9.2.2 където използвахме екстраполация на функцията $\mathbf{f}$) с помощтта на полином на Лагранж от втора степен в интервала $[x_{n-1}, x_{n+1}]$ което превръща метода в многостъпков и имплицитен (неявен). Например

$$\mathbf{f}(x) = \frac{(x - x_{n-1})(x - x_n)}{2h^2}\mathbf{f}_{n+1} - \frac{(x - x_{n-1})(x - x_{n+1})}{h^2}\mathbf{f}_n + \frac{(x - x_{n+1})(x - x_n)}{2h^2}\mathbf{f}_{n-1} + O(h^3) \tag{9.22}$$

спестявайки техническата част при интегрирането на функцията 9.22 заместена в уравнение 9.13 ще запишем окончателният израз

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{h}{12}\left[5\mathbf{f}_{n+1} + 8\mathbf{f}_n - \mathbf{f}_{n-1}\right] + O(h^4) \tag{9.23}$$

В заключение можем да споменем че при числено интегриране с помощта на горната схема са в сила същите аргументи както в по-горната секция 9.2.3 при определяне на $\mathbf{y}_{n+1}$

9.2.5 Метод на Рунге-Кута

Методът на Рунге-Кута може да се класифицара като много-стъпков неявен метод. Алгоритъмът е придобил особено широка популярост при числено интегриране на обикновенни диференциални уравнения от най-общ вид поради своята висока точност и числена стабилност. Това е може би методът който трябва да изберете в случаите когато ще ви бъде необходимо да намерите решение на произволна система ОДУ. Нека резгледаме схемата по която можем да получим уравненията описващи интеграционната схема на Рунге-Кута от втори ред. Уравненията за най-широко разпространената в практиката интеграционна схема от пети ред ще вземем на готово с цел да не усложняваме представянето тук.

Нека отново се върнем на уравнение 9.13 и използваме за приближение на функцията $\mathbf{f} = \mathbf{f}(x)$ в интервала $[x_n, x_{n+1}]$ ред на Тейлор около междинната точка $x_{n+1/2}$

$$\mathbf{f}(x) = \mathbf{f}_{n+1/2} + \mathbf{f}'_{n+1/2}(x - x_{n+1/2}) + O(h^2) \tag{9.24}$$

ако забравим за момент че не познаваме функцията $\mathbf{f}_{n+1/2}$ и нейната производната $\mathbf{f}'_{n+1/2}$ в средната точка $x_{n+1/2}$ и заместим уравнение 9.24 в 9.13 ще получим

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \mathbf{f}_{n+1/2}\int_{x_n}^{x_{n+1}} dx + \mathbf{f}'_{n+1/2}\int_{x_n}^{x_{n+1}} (x - x_{n+1/2})dx + O(h^3) = \mathbf{y}_n + h\mathbf{f}_{n+1/2} + O(h^3) \tag{9.25}$$

неизвестната стойност на $\mathbf{f}_{n+1/2} = \mathbf{f}(x_{n+1/2}, \mathbf{y}_{n+1/2})$ може да намерим с помощтта на уравнението на Ойлер 9.9

$$\mathbf{y}_{n+1/2} = \mathbf{y}_n + \frac{h}{2}\mathbf{f}_n + O(h^2) \tag{9.26}$$

Комбинирайки двете уравнения 9.25 и 9.26 можем да обобщим интеграционната схема на Рунге-Кута от втори ред

$$\begin{aligned} \kappa &= h\mathbf{f}(x_n, \mathbf{y}_n) && (9.27)\\ \mathbf{y}_{n+1} &= \mathbf{y}_n + h\mathbf{f}\left(x_n + h/2, \mathbf{y}_n + \kappa/2\right) && (9.28) \end{aligned}$$

Накрая нека представим системата уравнения реализираща интегриране на ОДУ по метода ма Рунге-Кута от четвърти ред, която изисква пресмятане на функцията $\mathbf{f}$ четири пъти на всяка интеграционна стъпка като локалната грешка е от порядък $O\left(h^5\right)$. Интеграционната схема е често предпочитана поради добрият баланс между точността на пресмятане и изчислителни ресурси

$$\begin{aligned} \kappa_1 &= h\mathbf{f}(x_n, \mathbf{y}_n) && (9.29а)\\ \kappa_2 &= h\mathbf{f}(x_n + h/2, \mathbf{y}_n + \kappa_1/2) && (9.29б)\\ \kappa_3 &= h\mathbf{f}(x_n + h/2, \mathbf{y}_n + \kappa_2/2) && (9.29в)\\ \kappa_4 &= h\mathbf{f}(x_n, \mathbf{y}_n + \kappa_3) && (9.29г)\\ \mathbf{y}_{n+1} &= \mathbf{y}_n + \frac{1}{6}\left[\kappa_1 + 2\kappa_2 + 2\kappa_3 + \kappa_4\right] + O(h^5) && (9.29д) \end{aligned}$$

9.2.6 Интегриране на Хамилтонови системи. Симплектични методи.

Хамилтновите системи са специален клас динамични системи които произлизат от формализма на Хамилтън и могат да се представят чрез система обикновенни диференциални уравнения (ОДУ) от първи ред. Хамилтоновите системи са важна част от класическата механика и намират приложение в небесната механика, молекулната динамика и др. Хамилтоновите системи се описват с помощта на функцията на Хамилтон $H = H(\mathbf{q}, \mathbf{p}, t)$ където величината $\mathbf{q}$ има смисъл на обобщен вектор на координатите, докато $\mathbf{p}$ се нарича обобщен вектор на импулса на системата. Можем да

добавим че, доколко двата вектора $\mathbf{q}$ и $\mathbf{p}$ са спрегнати един друг те имат един и същ размер. В случаите когато функцията на Хамилтон (или за по-кратко можем да е наричаме Хамилтониан) няма явна зависимост от времето $H = H(\mathbf{q}, \mathbf{p})$ системата се нарича стационарна и Хамилтонианът е интеграл на движението $H = E$, където E ще наричаме пълна енергия на системата. За простота в разглежданията по-долу ще използваме стационарна хамилтонова система. Както вече споменахме уравненията на Хамилтон дефинират система ОДУ чието решаване ни дава времевата еволюция на динамичните променливи $\mathbf{q} = \mathbf{q}(t)$ и $\mathbf{p} = \mathbf{p}(t)$

$$\frac{\partial}{\partial t}\mathbf{q} = \frac{\partial}{\partial \mathbf{p}} H(\mathbf{q}, \mathbf{p}) \tag{9.30}$$

$$\frac{\partial}{\partial t}\mathbf{p} = -\frac{\partial}{\partial \mathbf{q}} H(\mathbf{q}, \mathbf{p}) \tag{9.31}$$

Важно свойство на Хамилтоновите системи е че те са симплектични структури и времевата еволюция на системата може да бъде записана в термините на вектора на състоянието $\mathbf{r} = (\mathbf{q}, \mathbf{p})$

$$\frac{d}{dt}\mathbf{r} = S_N \cdot \nabla_{\mathbf{r}} H(\mathbf{r}) \tag{9.32}$$

матрицата S_N може да се дефинира в следната блочна форма

$$S_N = \begin{bmatrix} 0 & \mathbf{1}_N \\ -\mathbf{1}_N & 0 \end{bmatrix} \tag{9.33}$$

където $\mathbf{1}_N$ е идентичната матрица с размер $N \times N$, докато израза $\nabla_{\mathbf{r}} H(\mathbf{r})$ представлява обобщен запис на

$$\nabla_{\mathbf{r}} H(\mathbf{r}) = \begin{bmatrix} \frac{\partial}{\partial \mathbf{q}} H(\mathbf{q}, \mathbf{p}) \\ \frac{\partial}{\partial \mathbf{p}} H(\mathbf{q}, \mathbf{p}) \end{bmatrix} \tag{9.34}$$

едно следствие от това свойство е че ако разгледаме инфитизимален обем от фазовото пространство той би трябвало да се запази при интегрирането (еволюцията във времето) на Хамилтоновата система.

Друго важна характеристика Хамилтоновите системи са нейните първи интеграли. Например в случай на време независим Хамилтониан $H = H(\mathbf{q}, \mathbf{p})$ това са

- Пълната енергия на системата $E = H(\mathbf{q}, \mathbf{p})$
- Линейният момент на импулса $\mathbf{P} = \sum_{i=1}^{N} p_i$

В зависимст от симетрията и свойсвата на Хамилтониана на системата е възможно да съществуват и други интеграли на движението, например в случай че разглеждаме движението на n-брой обекта в тримерното пространство и дефинираме обобщените

координати q_i и импулси p_i за всеки обект $i = 1, \ldots, n$, тогава момента на импулса L също се запазва

$$L = \sum_i^n q_i \times p_i \tag{9.35}$$

В допълнение време-независимите Хамилтонови системи притежават още едно интересно свойство те са обратими във времето. Поради своите своите свойства интегрирането на Хамилтонови системи изисква специален клас интегратори които запазват с висока точност горе-изброените свойства. Този набор интеграционни схеми се нарича симплектични тъй като запазва симплектичната форма на Хамилтоновите уравнения при интегрирането. Досега ние разгледахме един такъв метод а именно интеграционната схема на Ойлер който се разглежда като симплектичен интегратор от първи ред. В практиката обаче е придобил широка популярност друга серия от интеграционни алгоритми наречен методи на Верле.

Нека предположим че Хамилтонианът на система може да се раздели на две части по отношение независимите променливи $\mathbf{q}$ и $\mathbf{p}$ или $H(\mathbf{q}, \mathbf{p}) = T(\mathbf{p}) + V(\mathbf{q})$ в този случай можем да разгледаме едно формално решение на Хамилтоновата система 9.32 което може да се запише с помощта на оператор на еволюцията

$$\mathbf{r}(\tau) = e^{\tau S_N \cdot \nabla_{\mathbf{r}} H} \mathbf{r}(0) = e^{\tau S_N \cdot (\nabla_{\mathbf{r}} T + \nabla_{\mathbf{r}} V)} \mathbf{r}(0) \tag{9.36}$$

изразът $e^{\tau S_N \cdot \nabla_{\mathbf{r}} H}$ има смисъл на интегрален оператор който евоюлира състоянието на системата за интервал от време τ. За съжаление в общият случай за прозволен потенциал на взаимодействие $V(\mathbf{q})$ явният вид на оператора на еволюцията не е известен. Затова симплектичните интеграционни алгоритми апроксимират оператора на еволюцията като произведение от κ-брой интегрални оператори от вида $e^{\tau S_N \cdot \nabla_{\mathbf{r}} T}$ и $e^{\tau S_N \cdot \nabla_{\mathbf{r}} V}$

$$e^{\tau S_N \cdot \nabla_{\mathbf{r}} (T+V)} = \prod_{i=1}^{\kappa} e^{c_i \tau D_T} e^{d_i \tau D_V} + O(\tau^{k+1}) = \ldots e^{c_1 \tau D_T} e^{d_1 \tau D_V} \ldots e^{c_\kappa \tau D_T} e^{d_\kappa \tau D_V} + O(\tau^{\kappa+1}) \tag{9.37}$$

където $D_T = S_N \cdot \nabla_{\mathbf{r}} T$ и $D_V = S_N \cdot \nabla_{\mathbf{r}} V$ са съкратен запис на скобките на Поасон за функциите на кинетичната T и потенциална V енергии. Например

$$D_T \mathbf{r} = S_N \cdot \nabla_{\mathbf{r}} T \mathbf{r} = \{\mathbf{r}, T\} = (\dot{\mathbf{q}}, 0) \tag{9.38}$$

$$D_V \mathbf{r} = S_N \cdot \nabla_{\mathbf{r}} V \mathbf{r} = \{\mathbf{r}, V\} = (0, -\dot{\mathbf{p}}) \tag{9.39}$$

$$\tag{9.40}$$

Числата c_i и d_i са реални и изпълняват условието $\sum_i c_i = \sum_i d_i = 1$. Съществено свойство за разглежданото представяне е че интегралните оператори $e^{c_i \tau D_T}$ и $e^{d_i \tau D_V}$ също реализират симплектична трансфорция и следователно произведението

на операторите също конструира симплектична карта. Доколко $D_T^2\mathbf{r} = \{\{\mathbf{r}, T\}, T\} = \{(\dot{\mathbf{q}}, 0), T\} = (0, 0)$ за произволно $\mathbf{r}$ можем да приемем че $D_T^2\mathbf{r} = \mathbf{0}$. Не е трудно да се покаже по подобен начин че $D_V^2\mathbf{r} = \mathbf{0}$ е също в сила за произволно $\mathbf{r}$. Следователно експоненциалната форма на оператора на еволюцията може да се опрости използвайки горните две свойства и развитието му в ред на Тейлор

$$\exp \tau D_T = \sum_{n=0}^{\infty} \frac{(\tau D_T)^n}{n!} = 1 + \tau D_T \tag{9.41}$$

$$\exp \tau D_V = \sum_{n=0}^{\infty} \frac{(\tau D_V)^n}{n!} = 1 + \tau D_V \tag{9.42}$$

сега можем да запишем в явен вид действието на интегралните оператори $\exp \tau D_T$ и $\exp \tau D_V$ за достатъчно кратък интервал време τ върху произволен вектор на състоянието $\mathbf{r} = (\mathbf{q}, \mathbf{p})$

$$e^{c_i \tau D_T} (\mathbf{q}, \mathbf{p}) = \left(\mathbf{q} + c_i \tau \frac{\partial T}{\partial \mathbf{p}}, \mathbf{p} \right) \tag{9.43}$$

$$e^{d_i \tau D_V} (\mathbf{q}, \mathbf{p}) = \left(\mathbf{q}, \mathbf{p} - d_i \tau \frac{\partial V}{\partial \mathbf{q}} \right) \tag{9.44}$$

Нека разгледаме няколко примера за симплектични интеграционни алгоритми от различен ред

- **Първи ред** Методът на Ойлер дискутирен вече в началото на тази глава се класифицира като симплектичен интеграционен алгоритъм от първи ред $\kappa = 1$ и коефициенти $c_1 = d_1 = 1$

- **Втори порядък** Методът на Верле може да се дефинира за $\kappa = 2$ и $c_1 = c_2 = \frac{1}{2}$, както и $d_1 = 1$ а $d_2 = 0$. Струва си да запишем интеграционната схема в детайли доколко този алгоритъм се е наложил в областта на числената физика поради бързината и симплектичната му форма

$$\begin{aligned} e^{\frac{1}{2}\delta t D_T} e^{\delta t D_V} e^{\frac{1}{2}\delta t D_T} (\mathbf{q}(t), \mathbf{p}(t)) = \\ = e^{\frac{1}{2}\delta t D_T} e^{\delta t D_V} \left(\mathbf{q}(t) + \frac{\delta t}{2m} \mathbf{p}(t), \mathbf{p}(t) \right) = \\ = e^{\frac{1}{2}\tau D_T} \left(\mathbf{q}(t + \frac{\delta t}{2}), \mathbf{p}(t) - \delta t \frac{\partial V}{\partial \mathbf{q}}(t + \frac{\delta t}{2}) \right) = \\ = \left(\mathbf{q}(t + \frac{\delta t}{2}) + \frac{\delta t}{2m} \mathbf{p}(t + \frac{\delta t}{2}), \mathbf{p}(t + \delta t) \right) \end{aligned} \tag{9.45}$$

следователно получаваме тристъпкова интеграционна схема

$$\mathbf{q}(t+\frac{\delta t}{2}) = \mathbf{q}(t) + \frac{\delta t}{2m}\mathbf{p}(t) \quad (9.46)$$

$$\mathbf{p}(t+\delta t) = \mathbf{p}(t) - \delta t\frac{\partial V}{\partial \mathbf{q}}(t+\frac{\delta t}{2}) \quad (9.47)$$

$$\mathbf{q}(t+\delta t) = \mathbf{q}(t+\frac{\delta t}{2}) + \frac{\delta t}{2m}\mathbf{p}(t+\frac{\delta t}{2}) \quad (9.48)$$

Тази интеграционна схема в практиката се нарича *Leapfrog* и реализира интегриране с отместени координати или *position Verlet*. Използвайки друго множество коефициенти $c_1 = 0$, $d_1 = \frac{1}{2}$, $c_2 = 1$ и $d_2 = \frac{1}{2}$ можем да получим друга *Leapfrog* интеграционна схема с отместени скорости или *velocity Verlet*

$$\mathbf{p}(t+\frac{\delta t}{2}) = \mathbf{p}(t) - \frac{\delta t}{2}\frac{\partial V}{\partial \mathbf{q}}(t) \quad (9.49)$$

$$\mathbf{q}(t+\delta t) = \mathbf{q}(t) + \frac{\delta t}{m}\mathbf{p}(t+\frac{\delta t}{2}) \quad (9.50)$$

$$\mathbf{p}(t+\delta t) = \mathbf{p}(t+\frac{\delta t}{2}) - \frac{\delta t}{2}\frac{\partial V}{\partial \mathbf{q}}(t+\delta t) \quad (9.51)$$

- **Трети ред** Методът на Роналд Рут [10] описва симплектична интеграционна схема от трети ред със следните коефициенти

$$\begin{array}{lll} c_1 = 1, & c_2 = -\frac{2}{3}, & c_3 = \frac{2}{3}, \\ d_1 = -\frac{1}{24}, & d_2 = \frac{3}{4}, & d_3 = \frac{7}{24}. \end{array}$$

- **Четвърти ред** Съществува интеграционна схема от по-висок ред открита от Роналд Рут [3] от четвърти ред $\kappa = 4$ с коефициенти

$$c_1 = c_4 = \frac{1}{2(2-2^{1/3})}, \quad c_2 = c_3 = \frac{1-2^{1/3}}{2(2-2^{1/3})},$$
$$d_1 = d_3 = \frac{1}{2-2^{1/3}}, \qquad d_2 = -\frac{2^{1/3}}{2-2^{1/3}},\ d_4 = 0.$$

9.2.6.1 Методи на Верле

Стандартната интеграционна схема на Верле може да бъде получена директно от уравненията на движение 9.30, 9.31 използвайки представяне в ред на Тейлър за дискретни интервали време t_{n-1}, t_n и t_{n+1}. За простота нека разгледаме едночастична система с маса m която за даден момент от време t се описва напълно с помощта на радиус вектора на координатата $\mathbf{r}(t) = \mathbf{q}(t)$ и скоростите $\mathbf{v}(t) = \mathbf{p}(t)/m$. Сега нека запишем израза за втората производна на радиус вектора $\mathbf{r}(t)$ около точката t_n чрез дискретната схема описана в уравнение 2.9 на Глава 2

$$\mathbf{a}(t_n) = \frac{\mathbf{r}(t_n + \delta t) - 2\mathbf{r}(t_n) + \mathbf{r}(t_n - \delta t)}{\delta t^2} + O\left(\delta t^2\right) \tag{9.52}$$

в записът който ползваме по-горе сме избрали стъпка на дискретизация по времето δt вместо h с цел по-интуитивен запис. Решавайки уравнение 9.52 спрямо $\mathbf{r}(t_n + \delta t)$ получаваме следната интеграционна схема

$$\mathbf{r}(t_n + \delta t) = 2\mathbf{r}(t_n) - \mathbf{r}(t_n - \delta t) + \delta t^2 \mathbf{a}(t_n) + O\left(\delta t^4\right) \tag{9.53}$$

$$\mathbf{v}(t_n) = \frac{\mathbf{r}(t_n + \delta t) - \mathbf{r}(t_n - \delta t)}{2\delta t} + O\left(\delta t^2\right) \tag{9.54}$$

второто уравнение в по-горната интеграционна схема представялва симетрична диференциална схема за намиране първата производна на функция разгледана в уравнение 2.5. Недостатък на този числен алгоритъм е че не познаваме радиус вектора на координатата $\mathbf{r}$ и скоростите $\mathbf{v}$ в един и същи момент време, което прави невъзможно пресмятането на функцията на пълната енергия на системата за новият момент време колко тя е сума на кинетичната и потенциалната енергия на системата.

В практиката по-широка популярност е придобила друга интеграционна схема наречена - *Алгоритъм на Верле със скорости*. Тази интеграционна схема може да бъде получена разлагайки в ред на Тейлор радиус вектора на позициите на системата $\mathbf{r}$ за момента време $t_n + \delta t$

$$\mathbf{r}(t_n + \delta t) = \mathbf{r}(t_n) + \delta t \mathbf{v}(t_n) + \frac{1}{2}\delta t^2 \mathbf{a}(t_n) + O\left(\delta t^3\right) \tag{9.55}$$

$$\mathbf{v}(t_n + \delta t) = \mathbf{v}(t_n) + \frac{\delta t}{2}\left[\mathbf{a}(t_n) + \mathbf{a}(t_n + \delta t)\right] \tag{9.56}$$

дясната част на уравнение 9.56 се получава след като изразим първата производна на скоростта $\mathbf{v}$ в междинната точка $t_n + \delta t/2$ за интервала $[t_n, t_n + \delta t]$ и приравним резултатът на усреднената стойност на ускорението $\mathbf{a}$ за двата края на интервала

$$\mathbf{a}(t_n + \delta t/2) = \frac{\mathbf{v}(t_n + \delta t) - \mathbf{v}(t_n)}{2(\delta t/2)} = \frac{1}{2}\left[\mathbf{a}(t_n) + \mathbf{a}(t_n + \delta t)\right] \tag{9.57}$$

9.2.7 Балистична задача

Нека разгледаме движението на обект с маса m и начална скорост $\vec{v}_0$ в гравитационното поле на Земята. В случаите когато началната скорост на обекта е достатъчно малка можем да игнорираме кривината на земната повърхност и тя да се разглежда плоска и успоредна на една от осите X на координатната ни система. За по-голяма простота можем да разглеждаме траекторията на обекта върху двумерна равнина XY образувана от вектора на началната скорост $\vec{v}$ и вектора на гравитационното ускорение $\vec{g}$.

Уравнението на движение на обекта можем да получим от вторият закон на Нютон

$$\vec{F} = m\vec{a} = m\frac{d^2}{dt^2}\vec{r} \tag{9.58}$$

Силата действаща на обекта включва два компомената

$$\vec{F} = m\vec{g} - \kappa\vec{v} \tag{9.59}$$

първият член в горният израз описва гравитационната сила действа на обект с маса m в гравитационното поле на Земата с гравитационно ускорение g в приближението кога височината на издигане на обекта е пренебрежимо малка. Вторият член в израза 9.59 представя силата на въздушно съпротивление която е право пропорционална на скоростта на обекта v с коефициент на въздушно съпротивление κ и действа винаги противоположно на посоката на разпространение на тялото. Дефинирайки скоростта на обекта като $\vec{v} = \frac{d}{dt}\vec{r}$ може да запишем обикновенното диференциоанлно уравнение от втори ред (9.58) като система от две обикновеени диференциални уравнения от първи ред

$$\frac{d}{dt}\vec{r} = \vec{v} \tag{9.60}$$
$$\frac{d}{dt}\vec{v} = m\vec{g} - \kappa\vec{v} \tag{9.61}$$

в допълнение можем да запишем системата ОДУ от първи ред по-горе по компоненти на векторите $\vec{r} = \{x, y\}$, $\vec{v} = \{v_x, v_y\}$ и $\vec{g} = \{0, -g\}$

$$\frac{d}{dt}x = v_x \tag{9.62}$$
$$\frac{d}{dt}y = v_y \tag{9.63}$$
$$\frac{d}{dt}v_x = -\kappa v_x/m \tag{9.64}$$
$$\frac{d}{dt}v_y = -g - \kappa v_y/m \tag{9.65}$$

При зададени начални условия като начална скорост $v_0 = 44.7\ m/s$, ъгъл на излитане $\theta = 45^{\circ}$, гравитационно ускорение $g = 9.81\ m/s^2$, маса на обекта $m = 0.145\ kg$ и коефициент на въздушно съпротивление $\kappa = 0.0431\ kg/s$. Времето за полет на обекта е $T = 5.17\ s$, дължината на полет е $L = 83.5\ m$ и максималната височина на издигане е $H = 52.0\ m$.

Инструкции за работа: Примерният код и работните скриптове се намират приложени в архива *ballistic.zip*, следвайте следните стъпки за да компилирате и изпълните примерният код:

1. Разархивирайте примерният код в избрана от вас работна папка с помощта на командата:
```
unzip -x ballistic.zip
```

2. Разгледайте програмният код *ballistic.f90* който реализира интегрирането на уравненията на движение (9.62-9.65) по метода на Ойлер от първи ред (9.9). Компилирайте изходния код с помощта на следната команда изпълена от терминалния прозорец:
```
gfortran ballistic.f90 -o ballistic.x
```

3. Изпълнете примерният код *ballistic.x* като пренасочите стандартният изход на програмата към файла *plot.dat*. Програмата ще изиска като вход да въведете интеграционната стъпка dt в $[s]$ например $dt = 0.1\ s$:
```
./ballistic.x > plot.dat
```

4. Нарисувайте траекторията на обекта като изпълнете командата (за да излезете от графичният режим на програмата *gnuplot* натиснете клавиша *Enter*):
```
gnuplot plot.gnu
```

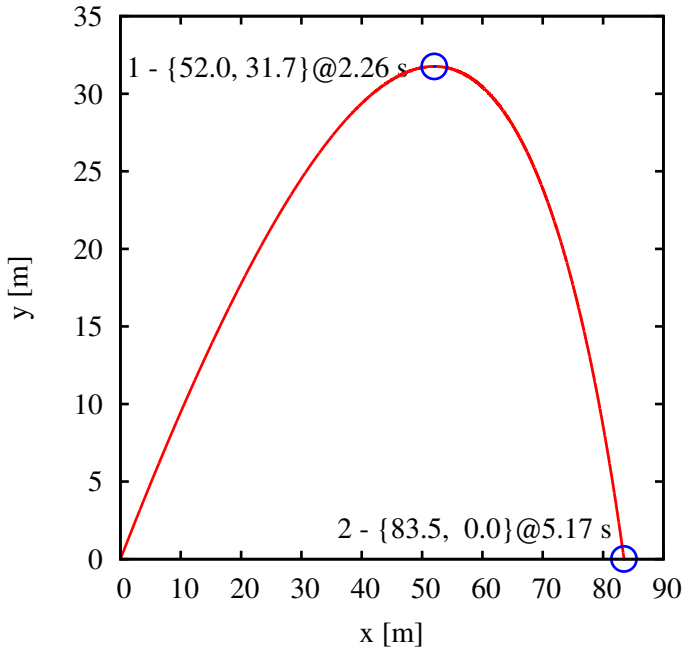
Файлът *plot.dat* съдържа координатите $\{x, y\}$ и скоростите $\{v_x, v_y\}$ на обекта за всяка интеграционна стъпка t_n. От него можете да определите продължителността на полета T, дължината на полета L както и максималната височина H.

Задача: Модифицирайте изходният код на програмата *ballistic.f90* така че да намирате момента време t_H и максималната височина на полета на обекта H.

Задача: Модифицирайте изходният код на програмата *ballistic.f90* и по-специално процедурата *integ(t, dt, y, ynew)*

```
subroutine integ(dt, t, y, ynew)
implicit none

real, intent(in) :: dt, t
real, dimension(:), intent(in) :: y
real, dimension(:), intent(out) :: ynew

real, dimension(size(y)) :: dydt

call deriv(t,y,dydt)
ynew(:) = y(:) + dt*dydt(:)

end subroutine integ
```

така че да реализирате схема на интегриране по метода на Рунге-Кута от пети ред (9.29) сравнете точността на пресмятането за една и съща интеграционна стъпка $dt = 0.01\ s$.



Фигура 9.1: Балистична траектория на обекта за интеграционна стъпка $dt = 0.001\ s$. С окръжностти са маркирани критичните точки от траекторията, позиция 1 отбелязва точката от траекторията с най-голяма височина $\{52.0, 31.7\}$ за момента време $t = 2.26\ s$. Краят на траекторията $\{52.0, 31.7\}$ за $t = 5.17\ s$ се маркира от точка 2.

9.2.8 Задача за n-тела

Във физиката задачата за n-тела разглежда движението на група небесни обекти, които взаимодействат помежду си посредтвом гравитационно поле. Първоначално този проблем е възникнал с интереса към движението на планетите около Слънцето. Днес тази задача е свързана и с определянето траекторията на астероиди и комети в Слънчевата система, еволюцията на звездни клъстери и други задачи от астрофизиката.
Нека разгледаме класическата задача за определяне траекторията на няколко небесни тела в Слънчевата система. Зе целта ще разгледаме система от n-брой небесни обекта всеки, от които се описва с радиус вектора на позицията $\mathbf{q}_i$ и импулса $\mathbf{p}_i = m_i\mathbf{v}_i$. Индекса i пробягва стойностите от $i \in (1 \ldots n)$. Хамилтонианът на системата може да се запише като:

$$H = T - U = \sum_{i=1}^{n} \frac{\mathbf{p}_i^2}{2m_i} - \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} G\frac{m_i m_j}{|\mathbf{q}_j - \mathbf{q}_i|} \tag{9.66}$$

Уравненията на Хамилтън дефинират система от $6n$-брой обикновени диференциални уравнения от първи ред, които могат да се запишат като:

$$\frac{d\mathbf{q}_i}{dt} = \frac{\partial H}{\partial \mathbf{p}_i} = \mathbf{p}_i/m_i = \mathbf{v}_i \tag{9.67}$$

$$\frac{d\mathbf{p}_i}{dt} = -\frac{\partial H}{\partial \mathbf{q}_i} = \sum_{\substack{j=1 \\ j\neq i}}^{n} G\frac{m_i m_j}{|\mathbf{q}_j - \mathbf{q}_i|^3}\left(\mathbf{q}_j - \mathbf{q}_i\right) = \mathbf{F}_i \tag{9.68}$$

Според теорията на небесната механика е добре известно че всяка система от n-тела притежава 10 интеграла на движение. Ще дефинираме три от тях, които ще бъдат интересни за проверка на стабилността и тозност на числено решение, което търсим:

1. Пълната енергия на системата: $E = T - U$, където кинетичната енергия T се записва като

   $$T = \sum_{i=1}^{n} \frac{\mathbf{p}_i^2}{2m_i} = \sum_{i=1}^{n} \frac{\mathbf{v}_i^2 m_i}{2}$$

   и потенциалната енергия U се задава с израза

   $$U = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} G\frac{m_i m_j}{|\mathbf{q}_j - \mathbf{q}_i|}$$

   .

2. Ъгловият момент на системата: $\mathbf{L} = \sum_{i=1}^{n} \mathbf{q}_i \times \mathbf{p}_i$

3. Центърът на масите на системата: $C = \frac{\sum_{i=1}^{n} m_i \mathbf{q}_i}{\sum_{i=1}^{n} m_i}$

За интегрирането на системата уравнения 9.67 и 9.68 ще използваме схема на Верле със скорости, взимайки предвид връзката между импулса $\mathbf{p}_i$ и скоростта $\mathbf{v}_i = \mathbf{p}_i/m_i$ на небесното тяло. Интеграционната схема на Верле е интересна с това че е симплектичен алгоритъм от втора степен:

$$\mathbf{q}_i\left(t + \delta t\right) = \mathbf{q}_i\left(t\right) + \mathbf{v}_i\left(t\right)\delta t + \frac{1}{2}\mathbf{a}_i \delta t^2 \tag{9.69}$$

$$\mathbf{v}_i\left(t + \delta t\right) = \mathbf{v}_i\left(t\right) + \frac{1}{2}\left[\mathbf{a}_i\left(t\right) + \mathbf{a}_i\left(t + \delta t\right)\right]\delta t \tag{9.70}$$

Ускорението $\mathbf{a}_i\left(t\right)$ се определя благодарение втория закон на Нютон:

$$\mathbf{a}_i\left(t\right) = \frac{\mathbf{F}_i\left(t\right)}{m_i} = \sum_{\substack{j=1 \\ j\neq i}}^{n} G m_j \frac{\mathbf{q}_j\left(t\right) - \mathbf{q}_i\left(t\right)}{|\mathbf{q}_j\left(t\right) - \mathbf{q}_i\left(t\right)|^3} \tag{9.71}$$

Примерен код: Разгледайте приложената програма, която реализира задачата за n тела в случай на гравитационно поле, конкретно ще разглеждаме движението на 8 небесни тела: Слънцето, Меркурий, Венера, Земята, Луната, Марс, Юпитер и Сатурн. В задачачата е удобно да се използват астрономически единици, т.е. масата се записва в единици Слънчеви маси $[M_\odot]$, позициите в единици разстояние между Земята и Слънцето $[A]$, а скоростта в единици разстояние между Земята и Слънцето за един астрономически ден $[A/D]$. В използваните единици гравитационната константа G може да се представи чрез Гаусовата гравитационна константа $k = 0.0172020985\ A^{3/2}D^{-1}M_\odot^{-1/2}$, като $G = k^2$. За център на отправната система е избрано Слънцето. Изчислителният код `runge_kuta` е написан на програмния език *Fortran 90* и е реализиран с помощта на процедурите `derivs` и `rk4`. Интегрирането на уравненията на движение се извършва с помощта на алгоритъм на Рунге-Кута от четвърти ред (уравнения 9.29), който е реализиран чрез процедурата `rk4` взаимствана от книгата *Числени рецепти* [8]. Позициите $\mathbf{q}_i$ и скоростите $\mathbf{v}_i$ на небесните тела се пазят в един обобщен масив `y` с размер $6 \times n$ елемента с индекси $\texttt{y(1)}, \texttt{y(2)}, \texttt{y(3)}, \ldots, \texttt{y(6} * \texttt{n)}$, в който последователно се записват координатите и скоростите на първият, вторият и т.н. обект. Времето t което е и независима променслива в случая се записва с `x`. В новият запис координатите на i-тото небесно тяло съотвестват на следните елементи от масива `y`

$$\begin{aligned} q_x^i &= y((i-1)6+1) \\ q_y^i &= y((i-1)6+2) \\ q_z^i &= y((i-1)6+3) \\ v_x^i &= y((i-1)6+4) \\ v_y^i &= y((i-1)6+5) \\ v_z^i &= y((i-1)6+6) \end{aligned}$$

С помощта на представянето по-горе уравненията на движение 9.67 и 9.68 могат да бъдат записани в термините на масивите `y` и `dydx`. Където масивът `dydx` пази първите производни на елементите на масива `y`:

$$\frac{q_x^i}{dt} = v_x^i \quad \rightarrow \quad dydx((i-1)6+1) = y((i-1)6+4)$$
$$\frac{q_y^i}{dt} = v_y^i \quad \rightarrow \quad dydx((i-1)6+2) = y((i-1)6+5)$$
$$\frac{q_z^i}{dt} = v_z^i \quad \rightarrow \quad dydx((i-1)6+3) = y((i-1)6+6)$$
$$\frac{p_x^i}{dt} = f_x^i \quad \rightarrow \quad dydx((i-1)6+4) = force((i-1)*3+1)/m(i)$$
$$\frac{p_y^i}{dt} = f_y^i \quad \rightarrow \quad dydx((i-1)6+5) = force((i-1)*3+2)/m(i)$$
$$\frac{p_z^i}{dt} = f_z^i \quad \rightarrow \quad dydx((i-1)6+6) = force((i-1)*3+3)/m(i)$$

Процедурата `derivs` пресмята дясната част на горната система:

```
! Subroutine calculating the derivatives in dydx using the input array y
    and
! independent variable x
! Array m keep the masses for all objects in Sun mass units
! dydx is a output with same size as y

subroutine derivs(x,y,dydx,m)

  implicit none
  DOUBLE PRECISION, intent(in) :: x
  DOUBLE PRECISION, dimension(:), intent(in) :: y,m
  DOUBLE PRECISION, dimension(:), intent(out) :: dydx
...
```

Процедурата *rk4* интегрира уравненията на движение със стъпка h като приема входният масив `y` и връща резултатът в масива `yout`:

```
SUBROUTINE rk4(y,dydx,x,h,yout,derivs, m)
        IMPLICIT NONE
        DOUBLE PRECISION, DIMENSION(:), INTENT(IN) :: y,dydx, m
        DOUBLE PRECISION, INTENT(IN) :: x,h
        DOUBLE PRECISION, DIMENSION(:), INTENT(OUT) :: yout
...
```

Инструкции за работа: Примерният код и работните скриптове се намират приложени в архива `nbody.zip`. За да компилирате и изпълните примерният код:

1. Разархивирайте примерният код в избрана от вас работна папка с помощта на командата:
   ```
   unzip -x nbody.zip
   ```

Таблица 9.1: Начални позиции и маси на небесните обекти

Обект	$q_x[A]$	$q_y[A]$	$q_z[A]$	$m[M\odot]$
Слънце	0.0	0.0	0.0	1.0
Меркурий	$2.95429702454060 \times 10^{-1}$	$-2.88017335923406 \times 10^{-1}$	$-5.06384726552837 \times 10^{-2}$	$1.66051140935277 \times 10^{-7}$
Венера	$-2.31948050444103 \times 10^{-1}$	$-6.8767314386583 \times 10^{-1}$	$3.96117537575149 \times 10^{-3}$	$2.44827371182131 \times 10^{-6}$
Земя	$-9.50208321594961 \times 10^{-1}$	$-3.15717462298336 \times 10^{-1}$	$1.57198229708770 \times 10^{-5}$	$3.00329789031573 \times 10^{-6}$
Луна	$-9.51589385022176 \times 10^{-1}$	$-3.13406237417784 \times 10^{-1}$	$-2.32705675327481 \times 10^{-4}$	$3.69566879083390 \times 10^{-8}$
Марс	-1.53752079761479	$-5.16672364557355 \times 10^{-1}$	$2.69136558810522 \times 10^{-2}$	$3.22773848604808 \times 10^{-7}$
Юпитер	-2.03584173478037	4.81530286628982	$2.55567832389706 \times 10^{-2}$	$9.5453256251810 \times 10^{-4}$
Сатурн	-6.515625090299700	-7.44592445978397	$3.88855499747995 \times 10^{-1}$	$2.85796542595990 \times 10^{-4}$

Таблица 9.2: Начални скорости на небесните обекти

Обект	$v_x[A/D]$	$v_y[A/D]$	$v_z[A/D]$	$m[M_\odot]$
Слънце	0.0	0.0	0.0	1.0
Меркурий	$1.40647151351859 \times 10^{-2}$	$2.14767415722188 \times 10^{-2}$	$4.64388279401861 \times 10^{-4}$	$1.66051140935277 \times 10^{-7}$
Венера	$1.90279747068330 \times 10^{-2}$	$-6.54953831064287 \times 10^{-3}$	$-1.18786394395136 \times 10^{-3}$	$2.44827371182131 \times 10^{-6}$
Земя	$5.15047488204331 \times 10^{-3}$	$-1.63880236594644 \times 10^{-2}$	$4.88814780957372\times^{-7}$	$3.00329789031573 \times 10^{-6}$
Луна	$4.67082413907776 \times 10^{-3}$	$-1.66731922899673 \times 10^{-2}$	$2.72731515511742 \times 10^{-6}$	$3.69566879083390 \times 10^{-8}$
Марс	$4.98105530733175 \times 10^{-3}$	$-1.20686343200058 \times 10^{-2}$	$-3.75139805044549 \times 10^{-4}$	$3.22773848604808 \times 10^{-7}$
Юпитер	$-7.04790430971189 \times 10^{-3}$	$-2.58403463856278 \times 10^{-3}$	$1.68428277376958 \times 10^{-4}$	$9.5453256251810 \times 10^{-4}$
Сатурн	$3.89034603207636 \times 10^{-3}$	$-3.69103383177723 \times 10^{-3}$	$-9.06858466369869 \times 10^{-5}$	$2.85796542595990 \times 10^{-4}$

2. Компилирайте изходния код с помощта на следната команда изпълена от терминалния прозорец:

```
1 gfortran solar.f90 -o solar.x
```

3. Изпълнете примерният код като пренасочите съдържанието на файла `init_cond.dat`, който пази параметрите на симулацията както и началните позиции, скорости, маси на обектите, към стандартния вход на програмата `solar.x`:

```
1 ./solar.x < init_cond.dat > evolution.trj
```

На първия ред от входния файл се записва продължителността на симулацията в астрономически дни D. Вторият ред задава интеграционната стъпка `h` (δt) в същите единици. **Останалите редове не бива да бъдат променяни**.

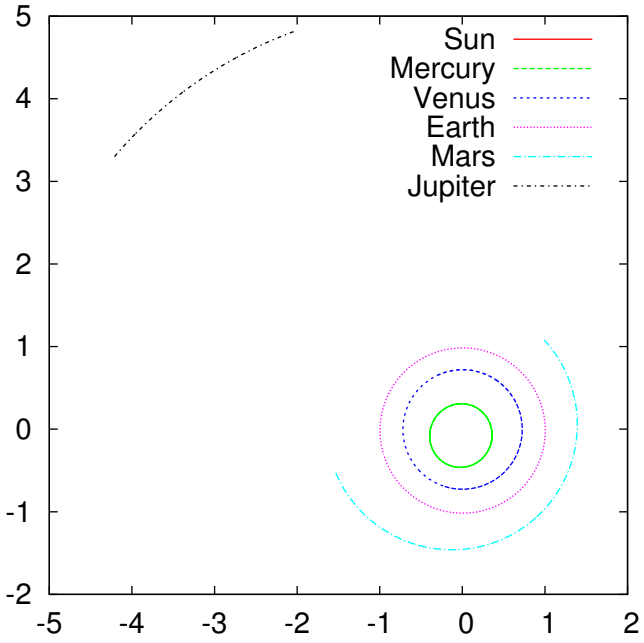
4. Визуализирайте траекториите с помощта на `gnuplot` скрипта `plot.gnu` прикрепен в архива:

```
1 gnuplot plot.gnu
```

Задача: Модифицирайте примерният код като добавите следните възможности:

1. Добавете функциите `epot(y,m)` и `ekin(y,m)` така че да пресмятат потенциалната U и кинетична енергия T съответно. Добавете отпечатване на стойностите на U, T и тяхната разлика $T - U$ и демонстрирайте, че величана $T - U$ може да се разглежда като интеграл на движението

2. Заменете интеграционната схема реализирайки интегриране по метода на Верле със скорости съгласно уравнения 9.69 и 9.70.



Фигура 9.2: Траектория на небесните тела за период от 365 дни и интеграционна стъпка $\delta t = 0.05D$

9.3 Случай на гранична задача

Когато върху нашето ОДУ са наложени гранични условия за повече от една стойност на независимата променлива задачата се нарича *гранична*. Обичайно граничните условия се налагат в две точки които са началната и крайната точка от интегрирането. В общият случай обаче тези точки могат да бъдат "граничните" точки върху които има наложени условия могат да бъдат вътрешни за интеграционният интервал, като например особени точки. Съществената разлика между задачата на Коши и граничната задача е че в горният случай ние можем да стартираме решението от дадена начална точка и намерим неговата "еволюция" последователно до краят на интеграционният интервал. В случаят на гранична задача налагането само на начално условие не е достатъчно за намирането на единствено решение. Граничната задача за решаване на система ОДУ може да бъде обобщена като: Намирането на решение за система от N обикновнни диференциални уравнения от първи ред, които задоволяват n_1 брой начални условия в началната точка x_1 и съответно изпълняват $n_2 = N - n_1$ допълнителни условия в крайната точка x_M. Подобна система ОДУ може да се запише като:

$$\frac{dy_i(x)}{dx} = f_i(x, y_1, y_2, ..., y_N) \qquad i = 1, 2, ...N \tag{9.72}$$

С наложени следните гранични условия за началната точка x_1:

$$B_j^1(x_1, y_1, y_2, ..., y_N) = 0 \qquad j = 1, ..., n_1 \tag{9.73}$$

И съответно в крайната точка останалите n_2 гранични условия:

$$B_k^2(x_1, y_1, y_2, ..., y_N) = 0 \qquad k = 1, ..., n_2 \tag{9.74}$$

9.3.1 Метод на стрелбата

9.3.1.1 Примерна задача едномерно стационарно уравнение на Шрьодингер

9.3.2 Релаксационен метод

Методът на релаксацията предполага че ние сме дискретизирали нашето ОДУ върху решетка от точки в интеграционният интервал $[x_1, x_\mathrm{M}]$, апроксимирайки диференциалното уравнение с Уравнение в Крайни Разлики (УКР). Нека разгледаме като пример една система от N-брой ОДУ от първи ред:

$$\frac{d\mathbf{y}(x)}{dx} = f(x, \mathbf{y}) \tag{9.75}$$

Горното уравнение може да бъде представено с помощта на крайни разлики като:

$$\mathbf{y}_k - \mathbf{y}_{k-1} - (x_k - x_{k-1})\, f\left[\frac{1}{2}(x_k + x_{k-1}), \frac{1}{2}(\mathbf{y}_k + \mathbf{y}_{k-1})\right] = 0 \tag{9.76}$$

Уравнението с крайни разлики (9.76) може да бъде изразено по много начини до колкото съществуват различни методи за дискретизация на ОДУ. В общия случай уравненията се представят като УКР върху решетка от M точки. Решението, което търсим се изразява в N зависими една от друга функции представени във всички M-брой точки или $N \times M$ брой променливи като цяло. Съществуват няколко релаксационни схеми, най-разпространената от които се базира на многомерен вариант на метода на *Newton* - за търсене решение на алгебрично уравнение, което се свежда до матрично уравнение. Нека разгледаме система алгебрични уравнения:

$$\begin{aligned} 0 = \mathbf{E}_k \equiv \mathbf{y}_k - \mathbf{y}_{k-1} - (x_k - x_{k-1})\, f_k(x_k, x_{k-1}, \mathbf{y}_k, \mathbf{y}_{k-1}) \\ k = 2, \ldots, M \end{aligned} \tag{9.77}$$

с наложени n_1 гранични условия на лявата граница ($x = x_1$) и $n_2 = N - n_1$ гранични условия на дясната граница ($x = x_M$) с $\mathbf{y}_k$ ще записваме вектора на решението. Системата уравнения $\mathbf{E}_k$ представлява N на брой уравнения свързващи $2N$ променливи в точките k и $k-1$ съществуват $M-1$ точки, за които уравнение (9.76) е в сила, с което (9.76) определя $(M-1)N$-брой уравнения за $M \times N$ неизвестни. Останалите N - брой уравнения се допълват от наложените гранични условия, първото:

$$0 = \mathbf{E}_0 \equiv \mathbf{B}(x_1, y_1) \tag{9.78}$$

и второто:

$$0 = \mathbf{E}_M \equiv \mathbf{C}\,(x_M, y_M) \tag{9.79}$$

където вектора $\mathbf{B}$ има само n_1 нули, като свободно могат избрани да бъдат като първите n_1, т.е. $E_j, 0 \neq 0$ за $j = n_1, n_1 + 1, \ldots, N - 1$. На другата граница, първите n_2 компоненти на $\mathbf{E}_{M-1} \neq 0$ за $j = 0, 1, \ldots, n_2 - 1$

Решението на системата уравнения (9.77), (9.78) и (9.79) съдържат $y_{j,k}$ стойности на N променливи $\mathbf{y}_j$ дискретизирани върху множество от $M\{x_k\}$ точки. За целта имаме нужда от начално предположение $y_{j,k}$, след което ние намиране/търсим промяната $\Delta y_{j,k}$, така че $y_{j,k} + \Delta y_{j,k}$ е подобреното решение на нашата сиситема (9.77). Търсената промяна може да бъде намерена с помощта на представянето в ред на Тейлор на подобреното решение $\mathbf{E}_k$

$$\mathbf{E}_k(\mathbf{y}_k + \Delta\mathbf{y}_k, \mathbf{y}_{k-1} + \Delta\mathbf{y}_{k-1}) \simeq \mathbf{E}_k(\mathbf{y}_k, \mathbf{y}_{k-1}) + \sum_{n=0}^{N-1} \frac{\partial \mathbf{E}_k}{\partial y_{n,k-1}} \Delta y_{n,k-1} + \sum_{n=0}^{N-1} \frac{\partial \mathbf{E}_k}{\partial y_{n,k}} \Delta y_{n,k} \tag{9.80}$$

Решението, което търсим трябва да води до $\mathbf{E}(\mathbf{y} + \Delta\mathbf{y}) \to 0$ Следователно уравнение (9.80) за точките от вътрешността приема вида:

$$\sum_{n=0}^{N-1} s_{j,k} \Delta y_{n,k-1} + \sum_{n=N}^{2N-1} s_{j,k} \Delta y_{n-N,k} = -E_{j,k} \tag{9.81}$$

за $j = 1, \ldots, N$, където:

$$s_{j,k} = \frac{\partial E_{j,k}}{\partial y_{n,k-1}}; \qquad s_{j,n+N} = \frac{\partial E_{j,k}}{\partial y_{n,k}} \tag{9.82}$$

за $n = 1, 2, \ldots, N$
Матриците $s_{j,k}$ представляват $N \times 2N$ матрици за всяко k. Граничните условия (9.78) и (9.79) могат да се запишат като:

$$\sum_{n=0}^{N-1} s_{j,n} y_{n,0} = -E_{j,0}; \qquad s_{j,n} = \frac{\partial E_{j,0}}{\partial y_{n,0}} \tag{9.83}$$

където $j = n_2, n_2 + 1, \ldots, N$ и $n = 1, 2, \ldots, N$. За втората граница:

$$\sum_{n=0}^{N-1} s_{j,n} \Delta y_{n,M-1} = -E_{j,M}; \qquad s_{j,n} = \frac{\partial E_{j,M}}{\partial y_{n,M-1}} \tag{9.84}$$

където $j = 1, 2, \ldots, n_2 - 1$ и $n = 1, 2, \ldots, N$.

Решението се състои в итеративно решение на системата (9.81) докато се получи сходимост.

9.3.2.1 Примерна задача едномерно уравнение на Поасон

Нека разгледаме едномерният вариант на уравнението на Поасон:

$$\nabla^2 y(x) = \frac{\partial^2 y(x)}{\partial x^2} = f(x) \tag{9.85}$$

в интервала $x \in [0, 1]$ с гранични условия $y(0) = 0$ и $y(1) = 0$. Горното уравнение може да бъде дискретизирано върху решетка решетка от n-брой точки равно отдалечени на разстояние h:

$$\frac{1}{h^2}\left(y_{i+1} - 2y_i + y_{i-1}\right) = f_i \tag{9.86}$$

Нека за момент разгледаме приблизителното решение $\tilde{y}_i$:

$$\frac{1}{h^2}\left(\tilde{y}_{i+1} - 2\tilde{y}_i + \tilde{y}_{i-1}\right) \approx f_i \tag{9.87}$$

Подобно уравнение 9.80 можем да дефинираме резидиума (остатъка) E_i като:

$$E_i = \frac{1}{h^2}\left(\tilde{y}_{i+1} - 2\tilde{y}_i + \tilde{y}_{i-1}\right) - f_i \tag{9.88}$$

Търсим поправка $\delta\tilde{y}_i$ към нашето текущо решение $\tilde{y}_i$ с цел да нулираме израза за резидиума 9.88. За тази цел подобно на 9.83 можем да развием в ред на Тейлор:

$$E_i(\tilde{y}_i + \delta\tilde{y}_i) \approx E_i(\tilde{y}_i) + \frac{\partial E_i}{\partial \tilde{y}_i}\delta\tilde{y}_i = \mathrm{E}_i - \frac{2}{h^2}\delta\tilde{y}_i = 0 \tag{9.89}$$

следователно можем да получим израз за корекцията $\delta\tilde{y}_i$:

$$\delta\tilde{y}_i = \frac{h^2}{2}E_i \tag{9.90}$$

$$\tilde{y}'_i = \tilde{y}_i + \frac{h^2}{2}E_i \tag{9.91}$$

В практиката уравнение 9.91 се модифицира като в него се добавя допълнителен параметър α с помощта на който може да се контролира сходимостта на релаксационният алгоритъм:

$$\tilde{y}'_i = \tilde{y}_i + \alpha\frac{h^2}{2}E_i \tag{9.92}$$

Обновяването на новите стойности $\tilde{y}'_i$ може да бъде изпълнено по два начина:

1. Гаус-Зибел релаксация при която резидиума E_i и новата стойност $\tilde{y}'_i$ се обновяват едновременно с оптимална стойност α около 1

```
...
do i = 1, n
  residue = ...
  ynew(i) = ...
end do
...
```

2. Релаксация на Якоби при която първо се пресмята множеството от всички резидиуми E_i след което се обновяват и $\tilde{y}'_i$ с препоръчителна стойност за $\alpha = 0.7$

```
...
do i = 1, n
  residue(i) = ...
end do
do i = 1, n
  ynew(i) = ...
end do

...
```

Задача: Напишете програма която решава едномерното уравнение уравнението на Поасон в случай $f(x) = -x\,(x+3)\,e^x$ с гранични условив $x \in [0,1]$. Сравнете резултатът с точното решение $y(x) = x\,(x-1)\,e^x$. Анализирайте сходимостта на двата релаксационни алгоритъма на Гаус-Зибел и Якоби.

9.4 Алгоритъм на Нумеров

В случаите на ОДУ от втори ред в който първата производна не присъства явно може да бъде приложен методът на Нумеров, който се класифицира като релаксационен алгоритъм. Подобни ОДУ имат приложение за някои физични задачи. Например описание процесите на едномерна дифузия, намиране стационарно решение на Уравнението на Шрьодингер и др. Нека разгледаме следното едномерно ОДУ от първа степен:

$$\left(\frac{d^2}{dx^2} + f(x)\right) y(x) = 0 \tag{9.93}$$

От разложението в ред на Тейлор около произволна точка x_0 имаме:

$$y(x) = y(x_0) + (x - x_0)y'(x_0) + \frac{(x-x_0)^2}{2}y''(x_0) + \ldots \tag{9.94}$$

Нека $x - x_0 = h$ и $x = x_0 + h$, тогава:

$$y(x_0 + h) = y(x_0) + hy'(x_0) + \frac{h^2}{2!}y''(x_0) + \frac{h^3}{3!}y'''(x_0) + \ldots$$

и

$$y(x_0 - h) = y(x_0) - hy'(x_0) + \frac{h^2}{2!}y''(x_0) - \frac{h^3}{3!}y'''(x_0) + \dots$$

може да се обобщи за $x_n = x_0 + nh$ и $y_n = y(x_n)$. Нека съберем $y_{n-1} + y_{n+1}$, тогава:

$$y_{n-1} + y_{n+1} = 2y_n + h^2 y_n'' + \frac{h^4}{12}y_n^{iv} + o(h^6)$$

освен това уравнение (9.93) ни дава $y_n'' = -f_n y_n$, ако го заместим в горното уравнение получаваме:

$$y_{n-1} + y_{n+1} = 2y_n - h^2 f_n y_n + \frac{h^4}{12}y_n^{iv} + o(h^6)$$

Използвайки ураванение (9.93) отново и израза за втората производна на функция в термините на крайни разлики можем да представим четвъртата производна на функцията y като:

$$y_n^{iv} = \frac{d^2}{dx^2}y'' = \frac{d^2}{dx^2}\left(f(x)y(x)\right) \Rightarrow$$

$$y_n^{iv} = -\frac{f_{n-1}y_{n-1} - 2y_n f_n + f_{n+1}y_{n+1}}{h^2}$$

$$0 = y_{n+1}\left(1 + \frac{h^2}{12}f_{n+1}\right) + y_{n-1}\left(1 + \frac{h^2}{12}f_{n-1}\right) - 2y_n\left(1 + \frac{5h^2}{12}f_n\right)$$

$$y_{n+1} = \frac{\left(2 - \frac{5h^2}{6}f_n\right)y_n - \left(1 + \frac{h^2}{12}f_{n-1}\right)y_{n-1}}{1 + \frac{h^2}{12}f_{n+1}}$$

Глава 10

Фурие трансформации

10.1 Общи положения

Нека разгледаме естествен процес който може да се опише с помощта на физична величина h която е функция на времето или $h = h(t)$. Например това може да бъде сигналът (напрежението, токът) измерен от някаква физична апаратура. Освен във времевият домейн функцията $h(t)$ може да бъде разглеждана и във честотният домейн $H(f)$ с помощта на подходящата функционална трансформация. Тази функционална трансформация се нарича *Фурие трансформация* и е взаимно обратима:

$$H(f) = \int_{-\infty}^{\infty} h(t) \exp^{2i\pi ft}\, dt \tag{10.1}$$

$$h(t) = \int_{-\infty}^{\infty} H(f) \exp^{-2i\pi ft}\, df \tag{10.2}$$

където независимата променлива f се разглежда като спрегната променлива на времето t и има смисъл на честота. По-често в практиката на вместо честотата f се използва кръговата честота $w = 2\pi f$, тогава горните трансформации придобиват вида:

$$H(w) = \int_{-\infty}^{\infty} h(t) \exp^{iwt}\, dt \tag{10.3}$$

$$h(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} H(w) \exp^{-iwt}\, dw \tag{10.4}$$

функциите $H(w)$ и $h(t)$ се наричат съответно "прав" и "обратен" фурие образ. Възможно е понякога независимата променлива на функцията h да има смисъл на разстояние x, или да разглеждаме функцията $h = h(x)$. Тогава "спрегната" променлива във фурие пространството k има смисъл на момент на импулса. За простота на представянето в уравненията по-долу ще предполагаме че разглеждаме функцията h във времевият

домейн. Но всички получени резултати могат тривиално да се презапишат и в термините на пространствените координати x. Нека разгледаме две сигнални функции $h(t)$ и $g(t)$ дефинирани във времевият домейн. Използвайки дефинициите за Фурие трансформация 10.3 и 10.4 лесно могат да докажат следните важни свойства които ние ще вземем наготово:

- Линейна трансформация. Фурие образ на сума на две функции:

$$h(t) + g(t) \leftrightarrow H(w) + G(w)$$

- Адитивност. Умножение на функцията h с константа λ:

$$\lambda h(t) \leftrightarrow \lambda H(w)$$

- Времево скалиране. Умножение с константа a на независимата променлива t:

$$h\left(at\right) \leftrightarrow \frac{1}{|a|} H\left(\frac{f}{a}\right)$$

- Честотно скалиране. Умножение с константа b на независимата променлива f:

$$\frac{1}{|b|}\left(\frac{t}{b}\right) \leftrightarrow bH\left(b\right)$$

- Времево отместване на сигнала с постоянна t_0:

$$h\left(t - t_0\right) \leftrightarrow H\left(f\right) \exp^{2i\pi f t_0}$$

- Честотно отместване на сигнала с постоянна f_0:

$$h\left(t\right) \exp^{-2i\pi f_0 t} \leftrightarrow H\left(f - f_0\right)$$

Също така са в сила следните симетрични връзки между фурие образите на функциите във времевият и честотни домейни

Оказва се че намирането на конволюцията на две фунцкции $g(t)$ и $h(t)$ в честотният домейн се свежда до просто умножение на техните фурие образи, или:

$$Conv(g,h)(t) = \left(g \star h\right)(t) = \int_{-\infty}^{\infty} g(\tau)h(t-\tau)d\tau \leftrightarrow G(f)H(f) \tag{10.5}$$

по-подобен начин може да се изрази и корелацията между две функции $g(t)$ и $h(t)$:

$$Corr(g,h)(t) = \left(g \star h\right)(t) = \int_{-\infty}^{\infty} g(\tau+t)h(t)d\tau \leftrightarrow G(f)H^*(f) \tag{10.6}$$

Времеви домейн	Честотен домейн
$h(t) \in \mathbb{R}$	$H(-f) = \lvert H(f)\rvert^*$
$h(t) \in \mathbb{I}$	$H(-f) = \lvert H(f)\rvert^*$
$h(t) = h(-t)$	$H(f) = H(-f)$
$h(-t) = -h(t)$	$H(-f) = -H(f)$
$h(t) = h(-t) \in \mathbb{R}$	$H(f) = H(-f) \in \mathbb{R}$
$h(-t) = -h(t) \in \mathbb{R}$	$H(-f) = -H(f) \in \mathbb{I}$
$h(t) = h(-t) \in \mathbb{I}$	$H(f) = H(-f) \in \mathbb{I}$
$h(-t) = -h(t) \in \mathbb{I}$	$H(-f) = -H(f) \in \mathbb{R}$

съответно в случайте котато се интересуваме от автокорелационната функция $Corr(g, g)$ можем да изпозлваме следният израз:

$$Corr(g, g)(t) = (g \star h)(t) = \int_{-\infty}^{\infty} g(\tau + t)g(t)d\tau \leftrightarrow |G(f)|^2 \tag{10.7}$$

използвайки горното равенство можем да запишем и равенстото на Парсевал което ни дава връзка между интеграла на квадрата на модула на функциите $h(t)$ и $H(f)$:

$$\int_{-\infty}^{\infty} |h(t)|^2 dt = \int_{-\infty}^{\infty} |h(f)|^2 df \tag{10.8}$$

10.2 Дискретни Фурие трансформации (ДФТ/DFT)

Често в практиката ние познаваме функциите само в дискретен набор от точки, което прави неприложими аналитичните изрази за "права" и "обратна" Фурие трансформации. Нека разгледаме отново функкцията $h(t)$ дискретизирана за екви-дистантни интервали от време Δ, така че

$$h_n = h(n\Delta) \;\; n = \ldots, -3, -2, -1, 0, 1, 2, 3, \ldots \tag{10.9}$$

реципрочната стойност на Δ ще наричаме *честота на семплиране*, ако Δ се измерва във единици за време тогава честотата на семплиране ще се измерва като брой стойности за единица време. За всяка стъпка на дискретизация се дефинира и критичната *честота на извадка (семплиране) на Никвист*, f_c

$$f_c \equiv \frac{1}{2\Delta} \tag{10.10}$$

Критичната честота на Никвист е важна с това че ако за функция $h(t)$ която е дискретизираната със стъпка Δ се окаже че нейният Фурие образ е честотно ограничен, т.е. $H(f) = 0$ за всички $|f| \geq f_c$, тогава функцията $h(t)$ може да се изрази напълно с помощта на дискретния набор от функционални стойности h_n

$$h(t) = \Delta \sum_{n=-\infty}^{+\infty} h_n \frac{sin\left[2\pi f_c(t - n\Delta)\right]}{\pi(t - n\Delta)} \tag{10.11}$$

един интересен извод от тази теорема е че "информационната стойност" на една честотно ограничена функция е безкрайно по-малък от този на една непрекъсната функция в общия случай, доколко за нас е необходимо да познаваме функцията само в дискретен набор от точки. Също така в случаите когато работим със сигнали получени от устройства които са честотно ограничени (например аудио усилвател) следва че цялата информация за сигнала може да бъде възтановена ако информацията е записана с честота Δ^{-1} равна на два пъти максималната честота на работа на устройството. В случаите когато честотният спектър $H(f)$ е неограничен обаче, ние винаги ще "губим" част от информацията на сигналната функция $h(t)$ при семплирането.

Дискретните Фурие трансформации могат да се дефинират за стойностите $h_k = h(t_k)$ на функцията $h(t)$ разгледната в краен брой точки N в интервала време $t \in [0, T]$, така че $\Delta = T/\left(N - 1\right)$, $t_k \equiv k\Delta$ и $k = 0, 1, 2, \ldots, N-1$. За простота нека предположим че броя на точките на дискретизация N е четен. В случите когато $h(t)$ е различна от 0 за краен интервал тогава без ограничение можем да изберем тези N точки да попадат изцяло в този интервал. За моменти време $t > T$ функцията $h(t)$ се разглежда периодична или $h(t+T) = h(t)$. Имайки N брой точки на дискретизация на функцията в реалното пространство можем да очакваме че ще намерим същият брой точки и при Фурие трансформацията. Затова нека пресметнем само стойностите на $H(f)$ само за дискретен набор от честоти в интервала $[-f_c, f_c]$

$$f_n \equiv \frac{n}{N\Delta} \;\; n = -\frac{N}{2}, \ldots, \frac{N}{2} \tag{10.12}$$

като началните и крайни стойности на f_n отговарят точно на критичната честота на Никвист. За да намерим израза за $H(f_n)$ ще използваме формулата 10.1, така че

$$H(f_n) = \int_{-\infty}^{\infty} h(t) \exp^{2i\pi f_n t}\, dt \approx \sum_{k=0}^{N-1} h_k \exp^{2i\pi f_n t_k}\, \Delta = \Delta \sum_{k=0}^{N-1} h_k \exp^{2i\pi kn/N} \tag{10.13}$$

В общия случай дискретните Фурие трансформации преобразуват N комплексни числа h_k в други N комплексни числа H_n. Технически ние преубразуваме един комплексен вектор $\mathbf{h}$ с N елемента в друг $\mathbf{H}$ и обратно. В практиката често с H_n се означва израза

$$H_n = \sum_{k=0}^{N-1} h_k \exp^{2i\pi kn/N} \tag{10.14}$$

тогава $H(f_n) \approx \Delta H_n$, използвайки горният израз 10.14 не е трудно да се покаже че дискретната функция H_n е периодична с период N. Следователно за отрицателните стойности на индексите n можем да запишем че $H_{-n} = H_{N-n}$ за $n = 1, 2, \ldots, \frac{N}{2} - 1$

с помощта на този запис можем в общият случай да разглеждаме $n = 0, \ldots, N-1$. Според този начин записване на индексите следва че нулевата честота отговаря на индекс $n = 0$, положителните честоти $0 < f < f_c$ отговрят на стойности на n за които $1 \leq n \leq N-1$, докато отрицателните честоти $-f_c < f < 0$ съотвестват на стойности $\frac{N}{2} + 1 \leq n \leq N-1$

Аналогично обратната Фурие трансформация се дава с израза

$$h_k = \frac{1}{N} \sum_{n=0}^{N-1} H_n \exp^{-2i\pi kn/N} \tag{10.15}$$

изразите за "права" 10.14 и "обратна" 10.15 Фурие трансформация се различават само по знака пред степентта на експонентата и фактора $1/N$ пред сумата в израза 10.15. В случаите на дискретна Фурие трансформация фомулата на Пърсивал се обобщава до следното равенство

$$\sum_{k=0}^{N-1} |h_k|^2 = \frac{1}{N} \sum_{n=0}^{N-1} |H_n|^2 \tag{10.16}$$

Като частен случай на дискретните Фурие трансформации можем да разгледаме и *синусовите* и *косинусовите* трансформации. Особеното при тях е че тези трансформации се извършват върху реални симетрични или асиметрични функции.

10.2.1 Дискретни синусови трансформации ДСТ/DST

Формално дисктретната синусова трансоформация се дефинира като линейна, обратима функция $F : \mathbb{R}^N \to \mathbb{R}^N$ (където $\mathbb{R}^N$ е N мерно реално пространство). Функцията F може и да се представи като $N \times N$ мерна ортогонална матрица. В практиката съществуват четири типа Фурие преобразования

DST-I

$$H_n = 2 \sum_{k=0}^{N-1} h_k \sin \left[\frac{\pi}{N+1} (n+1)(k+1) \right] \qquad n = 0, \ldots, N-1 \tag{10.17}$$

трябва да се отбележи матрицата $F^{nk} = \sin \left[\frac{\pi}{N+1} (n+1)(k+1) \right]$ е ортогонална с точност до множител, като DST-I трансформацията е еквивалентна на дискретна Фурие трансформация DFT в случаите когато множеството точки h_k е нечетна функция. Без да навлизаме в допълнителни детайли ще запишем останалите варианти на дискретни синусови трансформации за пълнота

DST-II

$$H_n = 2 \sum_{k=0}^{N-1} h_k \sin \left[\frac{\pi}{N} (n+1)(k+\frac{1}{2}) \right] \qquad n = 0, \ldots, N-1 \tag{10.18}$$

DST-III

$$H_n = (-1)^n h_{N-1} + 2\sum_{k=0}^{N-2} h_k \sin\left[\frac{\pi}{N}(n+1)(k+\frac{1}{2})\right] \qquad n = 0, \ldots, N-1 \tag{10.19}$$

DST-IV

$$H_n = 2\sum_{k=0}^{N-1} h_k \sin\left[\frac{\pi}{N+1}(n+1)(k+1)\right] \qquad n = 0, \ldots, N-1 \tag{10.20}$$

Обратната трансформация на DST-I е отново DST-I умножена с фактор $1/(2(N+1))$. Обратната трандформация на DST-IV е DST-IV умножена с $1/(2N)$. Обратната трансформация на DST-II е DST-III умножена с фактор $1/(2N)$, докато обратана трандофрмация на DST-III е DST-II умножена отново със същият фактор $1/(2N)$.

По аналогичен начин могат да се дефинират и дискретните косинусови трансформации ДКТ/DCT. Отново те се разглеждат като линейна трансформация $F : \mathbb{R}^N \to \mathbb{R}^N$

DCT-I

$$H_n = (h_0 + (-1)^n h_{N-1}) + 2\sum_{k=1}^{N-2} h_k \cos\left[\frac{\pi}{N-1}nk\right] \qquad n = 0, \ldots, N-1 \tag{10.21}$$

DCT-II

$$H_n = 2\sum_{k=0}^{N-1} h_k \cos\left[\frac{\pi}{N}n(k+\frac{1}{2})\right] \qquad n = 0, \ldots, N-1 \tag{10.22}$$

DCT-II е може би най-често използваната дискретна синусова трансформация и често в литературата се реферира просто като DCT. С точност до множител тя е еквивалентна на DFT трансформация за реална дискретна четна функция разглеждана в $4N$ брой точки.

DCT-III

$$H_n = h_0 + 2\sum_{k=1}^{N-1} h_k \cos\left[\frac{\pi}{N}(n+\frac{1}{2})k\right] \qquad n = 0, \ldots, N-1 \tag{10.23}$$

DCT-IV

$$H_n = 2\sum_{k=0}^{N-1} h_k \cos\left[\frac{\pi}{N}(n+\frac{1}{2})(k+\frac{1}{2})\right] \qquad n = 0, \ldots, N-1 \tag{10.24}$$

Обратната трансформация на DCT-I е DCT-I умножена с фактор $/2(N-1)$. Обратната трансформация на DCT-IV е DCT-IV умножена с фактор $1/2N$ докато обратната трансформация на DCT-II е DCT-III умножена отново с фактор $1/N$.

10.3 Бързи Фурие трансформации (БФТ/FFT)

В общият случай броят математиматически операции необходими за извършването на "права" или "обратна" дискретна Фурие трансформация е пропорзионален на N^2. Например преобрабразованието 10.14 може да се представи като матрична операция се изпозлва следният запис $W \equiv e^{2i\pi/N}$ тогава 10.14 приема вида

$$H_n = \sum_{k=0}^{N-1} W^{nk} h_k \tag{10.25}$$

с други думи компонентите на вектора h_k се умножават с $N \times N$ матрицата $\mathbf{W}$ за да получим коефициентите H_n това е преобразоевание което включва точно N^2 математатични операции. Добрата новина е че има развити числени алгоритми които могат да пресметнат елементите H_n с по-малък брой операции $N \log(N)$

10.3.1 Библиотека FFTW

FFTW е библиотека с отворен код реализирана на програмният език *C* която имплементира FFT алгоритъма и има интерфейс към програмният език Fortran също. Нейните функции са добре документирани в примерните задачи по-долу са изпозлвани именно извиквания към fftw. Нека разгледаме няколко процедури от тази библиотека които реализират DFT, DST и DCT трансофрмации върху едномерни масиви числа с размер `n`

10.3.2 Процедура `fftw_plan_dft_1d`. Едномерна комплексна DFT трансформация

Нека разгледаме два едномерени комплексни вектора: входен `in(n)`, който има смисъла на h_k в горните записи и изходен `out(n)`, който има смисъла на H_n

```
1 double complex, dimension(n) :: in, out
2 integer*8 plan
```

променливата `plan` е помощна и тя се изпозлва за дефинирането на операцията по DFT което се прави с извикването `fftw_plan_dft_1d`, след което се извършва самата операция по трансформация с помощта на извикването `fftw_execute`. Веднъж дефинирали операцията в променливата `plan` чрез `fftw_plan_dft_1d`, в общия случай ние можем да извършим многократно DFT трансформация като променяме съдържанието на входният вектор `in(n)`

```
1 call dfftw_plan_dft_1d(plan,n,in,out,FFTW_FORWARD,
      FFTW_ESTIMATE)
2 call dfftw_execute_dft(plan, in, out)
3 call dfftw_destroy_plan(plan)
```

процедурата `fftw_plan_dft_1d(plan, n, in, out, sign, flags)` приема шест аргумента

- plan - целочислена изходна променлива в която реферира към дефинираната от нас DFT операция
- n - цяло число указващо размера на входно изходните масиви
- in - входен комплексен едномерен масив с размер поне n елемента има смисъл на вектора от фукционални стойности h_k
- out - изходен комплексен едномерен масив с размер поне n елемента има смисъл на вектора от фукционални стойности H_k
- sign - цяло число указващо кода на операцията може да приема една от двете предефенирани константи FFTW_FORWARD = -1 или FFTW_BACKWARD = 1
- flags - цяло положително число което може да заема една от двете предефенирани константи FFTW_MEASURE или FFTW_ESTIMATE. Константата FFTW_MEASURE инструктира FFTW библиотеката да стартира и измери няколко от възможните FFT алгоритъма за трансформация за да определи най-оптималният от тях. Извикването обикновенно отнема няколко секунди и презаписва векторите in/out. Докато FFTW_ESTIMATE не изпълнява никакви тестове а прави само оценка на времето за изпълнение и масивите in/out остават незасегнати

10.3.3 Процедурата fftw_plan_r2r_1d. Едномерна реална DFT трансформация

В случаите когато искаме да реализираме дискретна синусова или косинусова трансформации DST/DFT можем да изпозлваме процедурата fftw_plan_r2r_1d(plan, n, in, out, kind, flags))

- plan - целочислена изходна променлива в която реферира към дефинираната от нас DFT операция
- n - цяло число указващо размера на входно изходните масиви
- in - входен реален едномерен масив с размер поне n елемента има смисъл на вектора от фукционални стойности h_k
- out - изходен реален едномерен масив с размер поне n елемента има смисъл на вектора от фукционални стойности H_k
- sign - цяло число указващо кода на операцията може да приема предефенирани константи
  - FFTW_REDFT00 (DCT-I): четна около j=0 и j=n-1.

  - FFTW_REDFT10 (DCT-II, класическа DCT): четна около j=-1/2 и j=n -1/2.
  - FFTW_REDFT01 (DCT-III, класическа обратна IDCT): четна около j=0 и нечетна около j=n.
  - FFTW_REDFT11 (DCT-IV): четна около j=-1/2 и нечетна около j=n-1/2.
  - FFTW_RODFT00 (DST-I): нечетна около j=-1 и j=n.
  - FFTW_RODFT10 (DST-II): нечетна около j=-1/2 и j=n-1/2.
  - FFTW_RODFT01 (DST-III): нечетна около j=-1 и четна около j=n-1.
  - FFTW_RODFT11 (DST-IV): нечетна около j=-1/2 и четна около j=n-1/2.

- flags - цяло положително число което може да заема една от двете предефенирани константи FFTW_MEASURE или FFTW_ESTIMATE. FFTW_MEASURE инструктира FFTW библиотеката да стартира и измери няколко от възможните FFT алгоритъма за трансформация за да определи най-оптималният от тях. Извикването обикновенно отнема няколко секунди и презаписва векторите in/out. Докато FFTW_ESTIMATE не изпълнява никакви тестове а прави само оценка на времето за изпълнение и масивите in/out остават незасегнати

Примерен код който реализира DST-I преубразование на програмният език Фортран

```
...
include "fftw3.f"
...
double precision, dimension(n) :: in, out
integer*8 plan
...
call fftw_plan_r2r_1d(plan,n,in,out,FFTW_RODFT00,FFTW_ESTIMATE)
call dfftw_execute_dft(plan, in, out)
call dfftw_destroy_plan(plan)
```

Забележка: Използването на fftw като външна библиотека изисква да се включи хедър файла fftw3.f с помощта на извикването include "fftw3.f" в изходният код, както и трябва да се добави референция към самата fftw библиотека по време на компилиране "-lfftw3" и да с укаже папката в който хедър файла (обикновенно това е директорията /usr/include) . Например

```
gfortran code.f90 -I/usr/inlcude -lfftw3 -o code.x
```

10.4 Задача: Времево отместване на сигнална функция. Фурие трансформация на функцията на Хевисайд

Напишете програма която извършва времево отместване на сигнална функция $h(t) \rightarrow h(t-t_0)$ използвайки подходяща Фурие образа $H(f)$ на функцията h(t) и свойството

$h(t-t_0) \leftrightarrow H(f)\exp^{2i\pi f t_0}$. Конкретно ще разгледаме функцията на Хевисайд $h(t)$ дефинирана в интервала $t \in [0,1]$

$$h(t) = \begin{cases} 1 & 0 \leq t < 1/2 \\ 0 & 1/2 \leq t \leq 1 \end{cases}$$

Можем да дискетизираме функцията $h(t)$ върху `n = 128` точки $t_i =$ `delta * i`, където `i = 0, ..., n - 1` и стъпката на дискретизация `delta` се пресмята като $\Delta = 1/(n-1)$ в масива $h_i = h(t_i) =$ `hTime(i+1)`. Фурие образа на функцията $H(f)$ съответно се дискретизира в масива `hFreq`, където $H_i = H(f_i)/\Delta =$ `hFreq(i+1)`. Важно е да отбележим как се пресмятат честотите f_i

$$f_i = \begin{cases} i & 0 \leq i \leq n/2 \\ i-n & n/2 < i \leq n-1 \end{cases}$$

по-долу е предстваен примерен код (заготовка) който дискретизира функцията $h(t_i) \rightarrow h_i$ и извършва дискретна Фурие трансформация за да получи коефициентите H_i. Допълнете кода така че да реализирате времевото преместване на сигналната функция $h(t)$.

Заготовка:

```
1  program heaviside
2  implicit none
3        include "fftw3.f"
4
5        integer n,nf
6        parameter (n=128)
7
8        integer*8 plan, iplan
9
10       double complex in, out
11       dimension hTime(n), hFreq(n)
12       double precision :: hi, pi, x
13       double precision :: timeDomainSum, freqDomainSum
14       double precision :: deltai, fi
15
16       integer :: i
17
18       pi = 3.14159265358979 3d0
19       delta = 1.d0 / (n-1)
20
21       do i=0, n-1
22          x = i * delta
23          if (x < 0.5d0) then
24            hTime(i) = cmplx(1.d0, 0.d0)
25          else
26            hTime(i) = cmplx(0.d0, 0.d0)
27          end if
28       enddo
29
```

```
30        call dfftw_plan_dft_1d(plan, n, hTime, hFreq, FFTW_FORWARD,
              FFTW_ESTIMATE)
31
32        call dfftw_execute(plan)
33
34        do i=1,n
35           if (i < n/2) then
36              fi = i - 1
37           else
38              fi = i - n - 1
39           end if
40           write(*,'(6F15.5)')(i-1.d0) / (n - 1), fi, real(in(i)), real(out(i)), aimag(
                 out(i)), abs(out(i))
41        enddo
42
43        write(0,*) 'Parsewal sum time domain: ',sum(abs( in)**2)
44        write(0,*) 'Parsewal sum freq domain: ',sum(abs(out)**2 / n)
45
46        call dfftw_destroy_plan(plan)
47
48        end
```

можете да клонирате изходният код на примерните задачи от хранилището в *Github* с помощта на командата:

```
1  git clone http://github.com/pisov/dft.git
2  cd dft/heaviside/fortran/
```

10.5 Примерна задача: Уравнение на Поасон

Нека разгледаме едномерното уравнение на Поасон с хомогенни гранични условия на Дирихле в интервала $x \in (0, 1)$:

$$\begin{aligned} -\frac{d^2u(x)}{dx} &= f(x) \\ u(0) &= 0 \\ u(1) &= 0 \end{aligned}$$

Като функцията $f(x)$ се дискретизира върху вътрешни за интервала $(0, 1)$ набор от точки x_i където $i = 0, 1, \ldots, N-1$, така че $u_i = u(x_i)$. Втората производна в горното уравнение може да се апроксимира с помощта на диференчни разлики като:

$$-\frac{d^2u(x_i)}{dx^2} = \frac{-u(x_{i+1}) + 2u(x_i) - u(x_{i-1})}{h^2} = f(x_i) \tag{10.26}$$

Където $h = 1/(N+1)$ ще наричаме стъпка на дискретизация, $x_i = (i+1)h$. Удобно е да разгледаме горното уравнение записано чрез Фурие образите $U(k)$ и $F(k)$ на функциите $u(x)$ и $f(x)$. В случай на краева задача с гранични условия на Дирихле

е подходящо да се ползва дискретна синусова трансформация която описва нечетна функция (DST-I) а именно:

$$U_k = 2\sum_{j=0}^{N-1} u_j \sin\left(\pi\frac{(k+1)(j+1)}{N+1}\right) = 2\sum_{j=0}^{N-1} S^{kj}u_j \tag{10.27}$$

$$\begin{aligned} u_j &= \frac{1}{2(N+1)}\sum_{k=0}^{N-1} U_k \sin\left(\pi\frac{(k+1)(j+1)}{N+1}\right) \\ &= \frac{1}{2(N+1)}\sum_{j=0}^{N-1} S^{jk}U_k = \frac{1}{2(N+1)}S^{jk}U_k \end{aligned} \tag{10.28}$$

където $S^{kj} = \sin\left(\pi(k+1)(j+1)/(N+1)\right)$ за $k = 0, 1, \ldots, N-1$. Използвайки тригонометричното равенство $\sin(\alpha \pm \beta) = \sin(\alpha)\cos(\beta) \pm \cos(\alpha)\sin(\beta)$ нека разгледаме уравнението 10.26 във Фурие пространството:

$$-\left\{S^{(j+1)k} - 2S^{jk} + S^{(j-1)k}\right\}U_k = h^2 S^{jk} F_k \tag{10.29}$$

$$2\left\{1 - \cos\left(\pi\frac{k+1}{N+1}\right)\right\}U_k = h^2 F_k \tag{10.30}$$

Правата и обратна трансформации 10.27, 10.28 се реализират с помощта на процедурата `DFFTW_PLAN_R2R_1D` от тип (kind) `FFTW_RODFT00` на библиотеката `fftw`. Напишете код реализиран на някой от следните програмни езици *Fortran 77*, *Fortran 90/95* или *C/C++* който решава едномерното уравнение на Поасон чрез намиране неизвестният вектор u_j използвайки уравнение

$$U_k = \frac{F_k * h^2}{2\left(1 - \cos\left(\pi\frac{k+1}{N+1}\right)\right)} \tag{10.31}$$

за $k = 0, \ldots, N-1$. Процедурата по намиране решението u_j изисква първо получаването на фурие образа F_k на функцията f_j, след което се приложи уравнение 10.31 и накрая се извърши обратна трансформация на Фурие за да се получат коефициентите u_j. Алгоритъма е схематично представен на фигурата по-долу:

$$\begin{array}{ccc} f_i & \xrightarrow{\texttt{FFTW\_RODFT00}} & F_k \\ & & \Big\downarrow \dfrac{h^2}{2\left(1 - \cos\left(\pi\frac{k+1}{N+1}\right)\right)} \\ u_j & \xleftarrow{\texttt{FFTW\_RODFT00}} & U_k \end{array}$$

В случай на дискретна синусова трансформация от вид DST-I типът на правата и обратна Фурие трансформации е един и същ `FFTW_RODFT00`.

Накрая нека дефинираме и функцията $f(x)$ като:

$$f(x) = -x\,(x+3)\,e^x$$

Сравнете полученото числено решение за различен брой точки точки на дискретизация N+1 = 8, 64 и 128 с аналитичното решение:

$$u(x) = x\,(1-x)\,e^x$$

Заготовка на решението:

```
module functions
implicit none
contains
    function f(x)
        double precision, intent(in) :: x
        double precision :: f
            f = -x * (x + 3) * exp(x)
    end function f
end module functions

program poisson
use functions
implicit none
      include "fftw3.f"

      integer :: n
      parameter (n=128)

      integer*8 plan, iplan

      double precision, dimension(N) :: uReal, uFourier
      double precision, dimension(N) :: fReal, fFourier
      double precision, dimension(N) :: sReal, sFourier
      double precision :: pi, x, freq, delta, L, xmin, xmax
      double precision :: timeDomainSum, freqDomainSum

      integer :: i

      pi = 3.141592653589793d0
      xmin = 0.d0
      xmax = 1.d0
      L = xmax - xmin
      delta = L / (n+1)

      do i=1,n
         x = xmin + i * delta
         fReal(i) = f(x)
         sReal(i) = x * (x-1) * exp(x)
      enddo

      call DFFTW_PLAN_R2R_1D(plan, n, fReal, fFourier,
          FFTW_RODFT00, FFTW_ESTIMATE)
      call dfftw_execute(plan)

```

```
44          do i = 1, n
45              x = xmin + i * delta
46              write(*,'(9F15.5)')x,uReal(i),sReal(i)
47          end do
48
49          write(0,*) 'Parsewal sum time domain: ',sum(abs(uReal)**2)
50          write(0,*) 'Parsewal sum freq domain: ',sum(abs(uFourier)**2)/(2*(n
                +1))
51          call dfftw_destroy_plan(plan)
52
53          end
```

примерният изходен код се намира в папката dft/poisson/fortran/ на проекта dft който може да бъде свален от github хранилището с помощта на командата:

```
1 git clone http://github.com/pisov/dft.git
```

Пълният код с решението на задачата се намира в допълнение A.7

Глава 11

Частни диференциални уравнения

11.1 Общи положения. Класификация

Частните диференциални уравнения имат много важна роля във физиката. Повечето природни закони могат да се представят с помощта на едно или система от частни диференциални уравнения. В най-обобщения си вариант, частното диференциално уравнение (ЧДУ) представлява диференциално уравнение в което участва неизвестна функция на много променливи и нейни частни производни. В известен смисъл обикновените диференциални уравнение (ОДУ) могат да се разглеждат като специален случай на частни диференциални уравнения, но методите за решаването на двата типа уравнения се различават съществено. В глава 9 вече разгледахме числените методи за решаване на системи ОДУ. В тази глава ще се съсредочим върху задачата за численото решаване на линейни частни диференциални уравнения от втори ред. Линейни диференциални уравнения са такива, за които линейна комбинация от решенията на уравнението отново е решение (принцип за суперпозицията). Можем да запишем едно линейно диференциално уравнение в общ вид по следния начин: $\mathbf{L}[u(x_1, x_2, \ldots, x_n)] = f(x_1, x_2, \ldots, x_n)$, където диференциалният оператор $\mathbf{L}$ е линеен. С други думи, линейни диференциални уравнения са такива, за които търсената функция участва в уравнението с коефициенти пред нея, които могат да са постоянни или да зависят от някоя от независимите променливи, по които се диференцира, но не и от самата функция. Едно линейно частно диференциално уравнение от втори ред за неизветната функция $u = u(x_1, x_2, \ldots, x_n)$ може да бъде записано като:

$$a_{ij}(x_1, x_2, \ldots, x_n)\frac{\partial^2 u}{\partial x_i \partial x_j} + b_j(x_1, x_2, \ldots, x_n)\frac{\partial u}{\partial x_j} + c(x_1, x_2, \ldots, x_n)u = -d(x_1, x_2, \ldots, x_n).$$

Тук a_{ij}, b_j, c_j и d_{ij} са известни функции на независимата променлива x. Примери за частни диференциални уравнения от втори ред са уравнението на Поасон, уравнението на Лаплас, уравнението за топлопроводност, вълновото уравнение и други.

Методите за решение на едно частно диференциално уравнение зависят от неговата "симетрия". Необходимостта от класификацията на ЧДУ се корени във физичните

процеси, които се асоциират с всеки един вид уравнение. По аналогия с коничните сечения (елипса, парабола и хипербола), частните диференциални уравнения могат да се класифицират като елиптични, параболични и хиперболични. Също както елипсата, която е гладка, затворена крива, решенията на елиптичите уравнения в общия случай също са "гладки". Например елиптичните частни диференциални уравнения описват процеси на дифузия които са достигнали равновесие като температурно поле. Друг пример за елиптични ЧДУ е функцията на електростатичния потенциал, който се получава в уравнението на Поасон. Хиперболата е "отворена" конична крива. Аналогично решенията на хиперболичните ЧДУ могат да описват времеви процеси, в които функцията е прекъсната или търпи резки промени, например разпространение на шокова вълна в среда. Хиперболичните ЧДУ могат да възникнат при разглеждането на процеси на конвекция или трептене на струна или мембрана. От математическа гледна точка параболичните ЧДУ представляват междинна връзка между елиптичните и хиперболичните ЧДУ. От физична гледна точка, в повечето случаи параболичните ЧДУ описват времезависими процеси, като например процеси на дифузия, топлинен трансфер или времезависимото уравнение на Шрьодингер.

Нека разгледаме едно линейно частно диференциално уравнение от втори ред на функцията $u = u(x, y)$ в най-общ вид:

$$au_{xx} + 2bu_{xy} + cu_{yy} + du_x + eu_y + ju = g \tag{11.1}$$

където долните индекси $_x$ и $_y$ представляват кратък запис на частните производни по съответната променлива, например $u_x = \partial u/\partial x$. Ще предполагаме, че познаваме функцията $g = g(x, y)$, докато a, b, c, d, e и j са числови константи. Ако заменим u_x с α и u_y с β, горното уравнение може да се запише като полином от втора степен спрямо променливите α и β:

$$P(\alpha, \beta) = a\alpha^2 + 2b\alpha\beta + c\beta^2 + d\alpha + e\beta + j \tag{11.2}$$

Математичното решение на уравнение 11.1 се определя до голяма степен от алгебричните свойства на полинома $P(\alpha, \beta)$. От друга страна, числениият подход силно зависи от типа математично решение и следователно от алгебричните свойства на полинома P. Частните диференциални уравнение се класифицират като хиперболични, параболични или елиптични по знака на дискриминаната на полинома P, $D = b^2 - ac$, съответно ако е положителна, нула или отрицателна. Ще демонстрираме класификацията на линейните ЧДУ с няколко примера.

Уравнение на Поасон

$$u_{xx} + u_{yy} + f(x, y) = 0 \tag{11.3}$$

Това уравнение се класифицира като елиптично уравнение, защото дискриминанатата $b^2 - ac = 0^2 - 1 \cdot 1 = -1 < 0$ е отрицателна.
Уравнението на Поасон и неговият частен случай при $f = 0$ (Уравнение на Лаплас):

това уравнение описва стационарни състояния на физични процеси като дифузия или електростатичен потенциал с плътност на заряда $\sim -f(x, y)$.

В практиката съществуват няколко различни подхода за решението на подобен тип ЧДУ и в настоящата книга в раздел 10.5 е разгледан метод, който използва Фурие трансформации. Друг възможен подход представя горното уравнение 11.3 като система линейни уравнения - този подход ще разгледаме по-долу в секция 11.2.

Дифузионно уравнение

$$u_t - u_{xx} = 0 \tag{11.4}$$

В случая дискриминантата $b^2 - ac = 0^2 - (-1) \cdot 0$ е равна на 0, което класифицира уравнението като параболично. Както вече споменахме, параболичните ЧДУ описват физични процеси като дифузия или топлопренос. Решенията $u(x, t)$ в повечето случаи са гладки функции, подобно на решенията на елиптичните ЧДУ. Съществуват и решения, които еволюират в области с голям градиент на функцията u. Числените методи за решаване на този клас ЧДУ апроксимират решението в следващи една друга времеви стъпки $u(x, t_{n-1})$, $u(x, t_n)$ и $u(x, t_{n+1})$ и т.н., като решението за даден момент време t_{n+1} включва решаване на линейна система, която може включва едно или повече решения от предходни моменти време t_{n-1}, t_{n-2} и т.н.

Едномерно вълново уравнение

$$u_{xx} - u_{tt} = 0 \tag{11.5}$$

Едномерното вълново уравнение притежава положителна дискриминанта $D = 0^2 - (1)(1) = 1$ и се класифицира като хиперболично частно диференциално уравнение. Този вид ЧДУ могат да възникнат от много области на физиката: акустиката, еластичността, хидродинамиката електродинамиката и метеорологията. Важно е да се знае, че числените решения на хиперболичните ЧДУ са толкова гладки, колкото граничните и начални условия. В допълнение може да се каже, че наличието на прекъснати или пик-образни фронтове на функцията u не затихват и се отразяват от границите на областта, в която се разглежда решението. Нелинейните ЧДУ могат да генерират разходими или скок-образни решения, дори когато граничните и начални условия са гладки. Следователно, от изчислителна гледна точка, този тип ЧДУ се решават най-трудно.

11.2 Елиптични диференциални уравнения. Уравнение на Поасон

Преди да демонстрираме численото решение на елиптично диференциално уравнение, ще разгледаме няколко примера от физиката, като уравнението на Поасон:

$$\nabla^2 u(\mathbf{r} = f(\mathbf{r}) \tag{11.6}$$

където $u(\mathbf{r})$ е неизвестната функция, която търсим, а $f(\mathbf{r})$ се нарича източник. Уравнението 11.6 се появява в различни задачи от физиката. Например в електростатиката функцията на потенциалната енергия $V(\mathbf{r})$ за известна плътност на разпределение на заряда $\rho(\mathbf{r})$ може да са намери, ако се реши уравнението:

$$\nabla^2 V(\mathbf{r} = -4\pi\epsilon_0\rho(\mathbf{r}) \tag{11.7}$$

където ϵ_0 е диелектичната проницаемост във вакуум. Подобно уравнение възниква и в случаите, когато се интересуваме от стационарното разпределение на плътността на температурното поле $T(\mathbf{r})$, създадено от топлинен източник $h(\mathbf{r})$:

$$\nabla^2 T(\mathbf{r}) = h(\mathbf{r}) \tag{11.8}$$

в случаите, когато топллиният източник h или заряда в обема ρ отсъстват, уравненията 11.8 и 11.7 се свеждат до уравнение на Лаплас. Решението се определя единствено от наложените гранични условия.

Един последен пример е уравнението за налягането $p(\mathbf{r})$ на флуид в системата уравнения на Навие-Стокс:

$$\nabla^2 p(\mathbf{r}) = f(\mathbf{r}), \tag{11.9}$$

където функцията $f(\mathbf{r})$ описва конвекцията на флуида.

11.2.1 Дискретизация и решение чрез на метода на Якоби

Нека продължим разглеждането в случая на функция $u = u(x, y)$ на две независими променливи, като имаме предвид, че метода, описан по-долу, може лесно да се обобщи в случаите на функция u на повече променливи. Ще разглеждаме функцията u в правоъгълна област с дължини L_x и L_y. Нека изберем стъпки на дискретизация h_x и h_y съответно за двете независими променливи x и y. Също така броят на точките на дискретизация N и M може да бъде различен. Тогава можем да разглеждаме функцията $u = u(x, y)$ за дискретен набор от $(N + 2) \times (M + 2)$ точки, $x_i = ih_x$ за $i \in 0, \ldots N + 1$ и $y_j = jh_y$ за $j \in 0, \ldots M + 1$. Точките с координати x_0, x_N, y_0 и y_M принадлежат на границите на областта, в която търсим функцията u, и ще предполагаме, че са ни известни, т.е. познаваме функциите:

$$\begin{array}{rcl} e(y) & = & u(x_0, y) \\ g(y) & = & u(x_{N+1}, y) \\ p(x) & = & u(x, y_0) \\ q(x) & = & u(x, y_{M+1}) \end{array} \tag{11.10}$$

Това се нарича задача на Дирихле.

Можем да дефинираме $u_{i,j} = u(x_i, y_j)$ и по подобен начин $f_{i,j} = f(x_i, y_j)$ за всички вътрешни точки на интервала, чиито индекси са в границите $i \in 1, \ldots, N$ и $j \in 1, \ldots, M$. Използвайки приближен израз (уравнение 2.9) за втората производна, уравнение 11.3 може да се запише в следната форма:

$$-\left[\frac{u_{i+1,j} + u_{i-1,j} - 2u_{i,j}}{h_x^2} + \frac{u_{i,j+1} + u_{i,j-1} - 2u_{i,j}}{h_y^2}\right] = f_{ij}. \tag{11.11}$$

Горното уравнение е в сила за всички вътрешни точки и може да се запише като линейна система уравнения за дискретните стойности на неизвестната функция $u_{i,j}$ заедно с уравнения 11.10 за граничните условия. Зе целта нека "подредим" стойностите на $u_{i,j}$ последователно с помощта на нов индекс $k = j + (i-1)M$. Тази нова подредба е взаимнообратима доколко ако за всяка стойност на k можем да възстановим i и j с помощта на съотвествието

$$\begin{aligned} i &= k \div M + 1 \\ j &= k \mod M \end{aligned} \tag{11.12}$$

където символът $\div$ означава целочислено делене. Можем да разгледаме дискретизираната функция $u_{i,j}$ като вектор φ_k:

$$\varphi = \begin{pmatrix} \varphi_1 = u_{1,1} \\ \varphi_2 = u_{1,2} \\ \ldots \\ \varphi_{M-1} = u_{1,M-1} \\ \varphi_M = u_{1,M} \\ \varphi_{M+1} = u_{2,1} \\ \varphi_{M+2} = u_{2,2} \\ \ldots \\ \varphi_{NM} = u_{N,M} \end{pmatrix} \tag{11.13}$$

Функцията $f_{i,j}$ може да се запише чрез вектора s_k, който вклюва и граничните условия 11.10

$$\mathbf{s} = \begin{pmatrix} s_1 = f_{1,1} - u_{1,0}/h_y^2 - u_{0,1}/h_x^2 \\ s_2 = f_{1,2} \\ \ldots \\ s_{M-1} = f_{1,M-1} \\ s_M = f_{1,M} - u_{0,1}/h_x^2 - u_{1,M}/h_y^2 \\ s_{M+1} = f_{2,1} - u_{1,0}/h_y^2 \\ s_{M+2} = f_{2,2} \\ \ldots \\ s_{NM} = f_{N,M} - u_{N,0}/h_y^2 - u_{0,M}/h_x^2 \end{pmatrix} \tag{11.14}$$

като индексите на векторите φ_k и s_k се изменят в границите $k \in 1, \ldots, NM$. С помощта на новия запис, уравнение 11.11 се представя като система линейни уравнения

$$\mathbf{A} \cdot \varphi = \mathbf{s} \tag{11.15}$$

Матрицата $\mathbf{A}$ е силно разредена матрица с преобладаващо ненулеви стойност по главният диагонал

$$A_{k,l} = \left(-2\delta_{k,l} + \delta_{k-M,l} + \delta_{k+M,l}\right)/h_x^2 + \left(-2\delta_{k,l} + \delta_{k,l-1} + \delta_{k,l+1}\right)/h_y^2 \tag{11.16}$$

където $k, l \in 1, \ldots, NM$. Матрицата $A_{k,l}$ може да се запише като сума на две матрици $\mathbf{A} = \mathbf{D} + \mathbf{R}$, където

$$\begin{aligned} D_{k,l} &= -2\delta_{k,l}\left(1/h_x^2 + 1/h_y^2\right) && (11.17)\\ R_{k,l} &= \left(\delta_{k-M,l} + \delta_{k+M,l}\right)/h_x^2 + \left(\delta_{k,l-1} + \delta_{k,l+1}\right)/h_y^2 && (11.18) \end{aligned}$$

Формално решението φ се свежда до

$$\varphi = \mathbf{A}^{-1}\mathbf{s} \tag{11.19}$$

Както споменахме в Глава 1, в случаите, когато матрицата $\mathbf{A}$ на линейната система е силно разредена, един удачен подход за нейното решаване, е итеративният метод на Якоби, при който решението се търси чрез матриците $\mathbf{D}$ и $\mathbf{R}$

$$\begin{aligned} (\mathbf{D} + \mathbf{R}) \cdot \varphi &= \mathbf{s} && (11.20)\\ (\mathbf{D} \cdot \varphi + \mathbf{R} \cdot \varphi) &= \mathbf{s} && (11.21)\\ \varphi &= \mathbf{D}^{-1}\left(\mathbf{s} - \mathbf{R} \cdot \varphi\right) && (11.22) \end{aligned}$$

Итеративно решение може да се запише с помощта на следното просто уравнение

$$\varphi_{new} = \mathbf{D}^{-1}\left(\mathbf{s} - \mathbf{R} \cdot \varphi_{old}\right) \tag{11.23}$$

Добрата новина е, че в горното уравнение 11.23, познавайки явният вид на матриците $\mathbf{D}$ и $\mathbf{R}$, можем да го пренапишем в термините на функциите $u_{i,j}$ и $f_{i,j}$

$$\mathbf{u}_{new} = \mathbf{D}^{-1}\left(\mathbf{f} - \mathbf{R} \cdot \mathbf{u}_{old}\right) \tag{11.24}$$

в матричен запис

$$u_{i,j}^{new} = \frac{h_x^2 h_y^2}{2(h_x^2 + h_y^2)}\left[f_{i,j} - \left(u_{i-1,j}^{old} + u_{i+1,j}^{old}\right)/h_x^2 + \left(u_{i,j+1}^{old} + u_{i,j-1}^{old}\right)/h_y^2\right] \tag{11.25}$$

Решението е итеративно. Можем да използваме следният критерий за сходимост на итеративният цикъл: $||u_{new} - u_{old}|| \leq \varepsilon$, т.е. приемаме търсенето решение, когато разликата между двете решения u_{new} и u_{old} за дадената итерация е по-малко от предварителтно зададено малко число $\epsilon < 1$.

11.2.2 Задача - Решение на двумерното уравнение на Поасон

Нека разгледаме задачачата за намиране на неизвестаната функция $u = u(x, y)$ в интервала $x \in [0, 1] \cap y \in [0, 1]$ за която се задава чрез уравнението на Поасон

$$u_{xx}(x, y) + u_{yy}(x, y) = -f(x, y)$$

където $f(x, y) = 2x^3 - 6xy(1-y)$. Сравнете числеото решение с аналитичното $u(x, y) = y(1 - y)x^3$ в случай на следните гранични условия: $u(x, 0) = 0$, $u(x, 1) = 0$, $u(0, y) = 0$ и $u(1, y) = y(1 - y)$

Заготовка на решението:

```
program poisson_2d
implicit none

integer :: i, j, n, m
double precision :: hx, hy, x, y, eps, xmin, xmax, ymin, ymax, diff

double precision, allocatable, dimension(:, :) :: uold, unew, source, uexact


write(0,*)'Please␣enter␣the␣number␣discretiozation␣points␣in␣x-direction␣n␣=␣'
read(*,*)n

write(0,*)'Please␣enter␣the␣number␣discretiozation␣points␣in␣y-direction␣m␣=␣'
read(*,*)m

!Allocate reserved memory
allocate(uold(0:n+1, 0:m+1))
allocate(unew(0:n+1, 0:m+1))
allocate(source(0:n+1, 0:m+1))
allocate(uexact(0:n+1, 0:m+1))

!Set boundaries in x-direction [0, 1]
xmin = 0.d0
xmax = 1.d0
!Set boundaries in y-direction [0, 1]
ymin = 0.d0
ymax = 1.d0

!Calculate the discretization step for x
hx = (xmax - xmin) / (n+1)
!Calculate the discretization step for y
hy = (ymax - ymin) / (m+1)

do j = 1, m
  y = j * hy + ymin
  do i = 1, n
    x = i * hx + xmin
    !Init source function f(x,y) = 2x^3-6xy(1-y)
    source(i, j) = 2 * x **3 - 6 * x * y * (1.d0 - y)
```

```
40      !Init exact solution Uexact(x,y) = y(1-y)x^3
41      uexact(i, j) = y * (1.d0 - y) * x ** 3
42    end do
43    !Applying non-zero boundary condition u(1,y) = y(1-y)
44    uold(n+1, j) = y * (1.d0 - y)
45  end do
46
47  !Initialization of Uold for Jakobi solver method u0(x, y) = f(x, y)
48  uold(1:n, 1:m) = source(1:n, 1:m)
49  !Applying zero boundary conditions
50  uold(0, :) = 0.d0
51  uold(:, 0) = 0.d0
52  uold(:, n+1) = 0.d0
53  !Unew initialization
54  unew(:,:) = 0.d0
55
56  !Set the complete criteria
57  eps = 1.d-5
58  !Calculate difference between "old" and "new" solution
59  diff = sum(abs(unew(1:n, 1:m) - uold(1:n, 1:m)))
60
61  do while (diff .gt. eps)
62    !Apply here the Poisson equation
63    diff = sum(abs(unew(1:n, 1:m) - uold(1:n, 1:m)))
64    uold(1:n, 1:m) = unew(1:n, 1:m)
65  end do
66
67  !Write out the calculated and exact solution
68  do j = 1, m
69    y = j * hy + ymin
70    do i = 1, n
71      x = i * hx + xmin
72      !write out x, y, u(x, y), uexact(x, y)
73      write(*,'(4E20.10)')x,y,uold(i,j),uexact(i,j)
74    end do
75  end do
76
77  !Deallocate reserved memory
78  deallocate(uold)
79  deallocate(unew)
80  deallocate(source)
81  deallocate(uexact)
82
83  end program
```

Примерният код може да бъде клониран, компилиран и изпълнен с помощта на следните команди в терминалния прозорец:

```
1  git clone http://github.com/pisov/pde.git
2  cd pde/elliptic/poisson/2d/
3  gfortran -O3 poisson_2d.f90 -o poisson_2d.x
4  ./poisson_2d.x > plot.dat
```

Файлът с данни plot.dat съдържа четири колони с данни $x_i, y_j, u_{solution}(x_i, y_j), u_{exact}(x_i, y_j)$,

където x_i, y_j са координатите на точките на дискретизация, докато $u_{solution}(x_i, y_j)$ е намереното числено решение, а $u_{exact}(x_i, y_j)$ е точното решение. Данните могат да бъдат визуализирани с помощта на програмният пакет `gnuplot`.

```
gnuplot plot.gnu
```

Решение:

```
    unew(1:n, 1:m) = hx **2 * hy ** 2 * (source(1:n, 1:m) &
    & + (uold(2:n+1, 1:m) + uold(0:n-1, 1:m)) / hx ** 2&
    & + (uold(1:n, 2:m+1) + uold(1:n, 0:m-1)) / hy ** 2)&
    & / (2 * (hx ** 2 + hy ** 2))
```

11.3 Параболично уравнение. Метод на Кранк - Николсон

Нека разгледаме отново параболично ЧДУ в най-общ вид за функция на две независими променливи $u = u(x, t)$. В случая променливата x има смисъл на координата в едно направление, а променливата t има смисъл на време.

$$u_t(x, t) - u_{xx}(x, t) = 0 \tag{11.26}$$

Търсеното решение представлява еволюцията на функцията $u = u(x, t)$ във времето спрямо първоначален момент време $t_0 = 0$, в който предполагаме, че познаваме $u_0(x) = u(x, 0)$. Също така ще приемем, че търсим решение, ограничено в пространството за $x \in [x_{min}, x_{max}]$. Отново можем да дискретизираме горното уравнение 11.26 като използваме стъпка на дискретизация Δt за момента време t и h_x за независимата пространствена координата x. Тогава ще разглеждаме функцията $u(x, t)$ в дискретен набор от точки $t_n = n\Delta t$, за $n = 0, 1, \ldots,$ и $x_i = ih_x + x_{min}$, за $i = 0, \ldots, N + 1$ или $u_i^n = u(x_i, t_n)$. Тогава уравнение 11.26 може да се дискретизира чрез следните две схеми:

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{h_x^2} \qquad \text{експлицитна (явна) форма} \tag{11.27}$$

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \frac{u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1}}{h_x^2} \qquad \text{имплицитна (неявна) форма} \tag{11.28}$$

решението на експлицитното уравнение 11.27 може да се запише много лесно, просто като се прехвърлят от дясната страна на уравнението всички членове, пропорционални на $\sim u^n$, тогава

$$u_i^{n+1} = \alpha' \left(u_{i+1}^n + u_{i-1}^n\right) + (1 - 2\alpha')u_i^n$$

където $\alpha' = \Delta t / h_x^2$. В случаите, когато $\alpha' > 1$, численото решение на явното уравнение 11.27 е нестабилно и трябва да се избягва. В практиката съществува алгоритъм, развит

от Кранк и Николсон който комбинира двете уравнения 11.27 и 11.28. Нека разгледаме сумата на двете урвнения 11.27 и 11.28

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \frac{1}{2}\left[\frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{h_x^2} + \frac{u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1}}{h_x^2}\right] \tag{11.29}$$

Ако в горното уравнение 11.29 прегрупираме всички членове $\sim u^{n+1}$ от лявата страна, а тези $\sim u^n$ - от дясната, можем да презапишем уравнението като:

$$-\alpha u_{i+1}^{n+1} + (1+2\alpha)u_i^{n+1} - \alpha u_{i-1}^{n+1} = \alpha u_{i+1}^n + (1-2\alpha)u_i^n + \alpha u_{i-1}^n, \tag{11.30}$$

където $\alpha = \Delta t/(2h_x^2)$. Ако разгледаме горното уравнение 11.30 за индексите $i = 1, \ldots, N$ то представлява три-диагонална линейна система за неизвестният вектор u_i^{n+1}. Уравнението може да се запише в матричен вид, ако изпозлваме символичния запис за дискретизирания оператор $\mathbf{D}$, пропорционален на оператора на втората производна по независимата променлива x с коефициент $1/h_x^2$.

$$D_{i,j}u_j^n = u_{i+1}^n - 2u_i^n + u_{i-1}^n \tag{11.31}$$

Тогава уравнение 11.30 примема матрична форма

$$(\mathbf{1} + \alpha\mathbf{D}) \cdot \mathbf{u}^{n+1} = (\mathbf{1} - \alpha\mathbf{D}) \cdot \mathbf{u}^n \tag{11.32}$$

$$\mathbf{u}^{n+1} = (\mathbf{1} + \alpha\mathbf{D})^{-1} \cdot (\mathbf{1} - \alpha\mathbf{D}) \cdot \mathbf{u}^n \tag{11.33}$$

доколко и двете матрици $(\mathbf{1} + \alpha\mathbf{D})$ и $(\mathbf{1} - \alpha\mathbf{D})$ са тридиагонални.

Решението на уравнение 11.33 може да се извърши с методите, описани в раздел 1.4 от ръководството. За примерните задачи, които ще разгледаме по-долу, ще използваме процедурите `SUBROUTINE ZGTSV( N, NRHS, DL, D, DU, B, LDB, INFO )` или `SUBROUTINE DGTSV( N, NRHS, DL, D, DU, B, LDB, INFO )` от библиотеката *LAPACK*, която решава линейната система $\mathbf{A} \cdot \mathbf{x} = \mathbf{B}$

- `N` - (входен) целочислен параметър, равен на размера на квадратната матрица $\mathbf{A}$, `N >= 0`
- `NRHS` - (входен) целочислен параметър, равен на броя на колоните на матрицата B. `NRHS = \max(1, N)`
- `DL` - (входно/изходен) едномерен масив от тип реални числа с двойна точност с размер `N - 1` брой елемента. При началното извикване трябва да съдържа елементите на горния диагонал на матрицата $\mathbf{A}$. След приключване изпълнението на процедурата съдържа елементите на горният диагонал на `U` матрицата.
- `D` - (входно/изходен) едномерен масив от тип реални числа с двойна точност с размер `N` брой елемента. При началното извикване трябва да съдържа елементите на главният диагонал на матрицата $\mathbf{A}$. След приключване изпълнението на процедурата съдържа главните елементи на `U` матрицата.

- DU - (входно/изходен) едномерен масив от тип реални числа с двойна точност с размер N - 1 брой елемента. При началното извикване трябва да съдържа елементите на долния диагонал. След приключване изпълнението на процедурата съдържа елементите на горния диагонал на L матрицата.

- B - (входно/изходен) масив от реални числа с двойна точност с размерност (LDB , NRHS). При началното извикване съдържа матрицата от вектор-стълбовете на дясната част на линейната система. След приключване на извикването, ако INFO = 0, съдържа матрица от вектор стълбовете на решението x.

- LDB - (входен) целочислен параметър, работна размерност матрицата B. LDB >= max(1,N)

- INFO - (изходен) целочислена променлива, в която се записва код на приключването на процедурата:

  N = 0: успешно приключване

  N < 0: ако N = -i тогава i-тия елемент има неправилна числена стойност (Inf, NaN)

  N > 0: тогава при факторизацията i-тия елемент U(i,i) е равен на нула.

Процедурата ZGTSV има същия интерфейс със заменени комплексни променливи с двойна точност COMPLEX16

11.3.1 Дифузионно уравнение

$$\frac{\partial}{\partial t}u(x,y,t) = \nabla \cdot (D(x,y)\nabla u(x,y,t)) \tag{11.34}$$

където $D(x,y)$ е дифузионна функция, която в частен случай може да бъде и константа D. Дифузионното уравнение може да се класифицира като параболично ЧДУ.

11.3.2 Уравнение на Шрьодингер

Времезависимото уравнение на Шрьодингер е едно от фундаменталните уравнения в класическата нерелативистична квантова механика. То описва времевата еволюция на вълновата функция $\Psi(\mathbf{r},t)$ на квантово механична система описвана с помощта на оператора на Хамилтон $\hat{\mathbf{H}}$:

$$i\hbar\frac{\partial}{\partial t}\Psi(\mathbf{r},t) = \hat{\mathbf{H}}\Psi(\mathbf{r},t) = -\frac{\hbar^2}{2m}\nabla^2\Psi(\mathbf{r},t) + V(\mathbf{r})\Psi(\mathbf{r},t) \tag{11.35}$$

където $\hat{\mathbf{T}} = -\frac{\hbar^2}{2m}\nabla^2$ е оператора на кинетичната енергия, а $\hat{\mathbf{V}} = V(\mathbf{r})$ е операторът на потенциалната енергия. Числено решение чрез метода на Кранк-Николсон включва дискретизирането на вълновата функция Ψ във времевия и пространствения домейни

$\Psi^n_{ijk} = \Psi(x_i, y_j, z_k, t_n)$, където $x_i = i * \delta_x$, $y_j = j * \delta y$, $z_k = k * \delta z$ и $t_n = n * \delta t$ индексите $i, j, k \in \mathbf{Z}$ принедлежат на множеството на целите числа. Докато индекса $n \in \mathbf{N}$ обикновено принадлижи на множеството на естествените числа. Използвайки уравнението на Кранк-Николсон от предходния раздел 11.33 дискретизираното уравнение 11.35 приема формата:

$$\left(1 + i\frac{1}{2}\hat{\mathbf{H}}\delta t\right)\Psi^{n+1} = \left(1 - i\frac{1}{2}\hat{\mathbf{H}}\delta t\right)\Psi^n \tag{11.36}$$

$$\Psi^{n+1} = \left(1 + i\frac{1}{2}\hat{\mathbf{H}}\delta t\right)^{-1}\left(1 - i\frac{1}{2}\hat{\mathbf{H}}\delta t\right)\Psi^n \tag{11.37}$$

11.3.3 Задача - Разпространение на вълнови пакет. Времево зависимо уравнение на Шрьодингер

За да демонстрираме числеността уравнение на Шрьодингер по метода на Кранк-Николсон описано в уравнение 11.37, ще разгледаме задачата за разпространение на вълнови пакет с камбановидна форма в едно измерение през потенциална бариера с височина v_0 в нормировка $\hbar \equiv m \equiv 1$. Нека $\Psi(x,t)$ е комплекса функция на координатата $x \in [x_{min}, x_{max}]$ и времето $t \in [0, \infty)$. Ще предположим, че познаваме началното разпределение на вълновата функция $\Psi_0(x) = \Psi(x, 0)$, което има камбановидна форма с център точката x_0

$$\Psi_0(x)' = \frac{1}{\sqrt[4]{2\pi\sigma^2}} \exp^{(x-x_0)^2/(4\sigma^2)}$$

началната функция $\Psi_0(x)$ се умножава със собствена функция на свободна частица с импулс k или $\exp^{ikx}$, тогава

$$\Psi_0(x) = \frac{1}{\sqrt[4]{2\pi\sigma^2}} \exp^{(x-x_0)^2/(4\sigma^2)+ikx}$$

като кинетичната енергия E_{kin} за горната вълнова функция е свързъна с импулса k чрез равенството $k = \sqrt{2E_{kin} - 1}$. За простота ще предположим, че потенциалната бариера има проста правоъгълна форма с височина V_0 и ширина L със среда разположена в точка x_L или

$$V(x) = \begin{cases} V_0, x \in [x_L - L/2, x_L + L/2] \\ 0, x \notin [x_L - L/2, x_L + L/2] \end{cases} \tag{11.38}$$

Нека дискретизираме функцията $\Psi(x,t)$ във времевият домейн със стъпка Δt, така че $t_n = n\delta t$ и $n = 0, \ldots,$. Докато за пространствената координата използваме стъпка h_x, така че $x_i = ih_x + x_{min}$ за $i = 0, 1, 2, \ldots, N, N+1$. Ще предполагаме, че в граничните точки стойността на вълновата функция $\Psi(x,t)$ е постоянна и равна на нула,

т.е. $\Psi(x_0, t_n) = \Psi(x_{N+1}, t_n) = 0$. Тогава можем да запишем дискретната форма на фукцията Ψ

$$\Psi_i^n = \Psi(x_i, t_n)$$

Операторът на Хамилтон за системата може тогава да се запише във вида

$$\hat{H} = -\frac{1}{2}\frac{d^2}{dx^2} + \hat{V}(x) = -\frac{1}{2}\frac{d^2}{dx^2} + V_0\mathcal{H}(x - x_L + L/2)(1 - \mathcal{H}(x - x_L + L/2)) \tag{11.39}$$

$\mathcal{H}(x)$ е функцията на Хевисайд. Сега можем да запишем уравнение 11.37 в компонентен вид спрямо дискретните стойности на функцията Ψ_i^n и Ψ_i^{n+1}:

$$\begin{aligned} \Psi_i^{n+1} - \imath\frac{\alpha}{2}\left(\Psi_{i+1}^{n+1} - 2\Psi_i^{n+1} + \Psi_{i-1}^{n+1}\right) + \imath\frac{\delta t}{2}\Psi_i^{n+1}V_i = \\ \Psi_i^n + \imath\frac{\alpha}{2}\left(\Psi_{i+1}^n - 2\Psi_i^n + \Psi_{i-1}^n\right) - \imath\frac{\delta t}{2}\Psi_i^n V_i \end{aligned} \tag{11.40}$$

където $\alpha = \delta t/2h_x^2$. Горното уравнение 11.40, всъщност както споменахме и преди, представлява една линейна тридиагонална система спрямо неизвестните стойности на вълновата функция Ψ_i^{n+1} в момента време t_{n+1} за всички "вътрешни" индекси $i = 1, \ldots, N$. Ако положим:

$$b_i = \Psi_i^n + \imath\frac{\alpha}{2}\left(\Psi_{i+1}^n - 2\Psi_i^n + \Psi_{i-1}^n\right) - \imath\frac{\delta t}{2}\Psi_i^n V_i \tag{11.41}$$

$$y_i = \Psi_i^{n+1} \tag{11.42}$$

$$D_i = 1 + \imath\left(\alpha + \frac{\delta t}{2}V_i\right) \tag{11.43}$$

$$L_i = -\imath\frac{\alpha}{2} \tag{11.44}$$

$$U_i = -\imath\frac{\alpha}{2} \tag{11.45}$$

тогава уравнението на Кранк-Николсон 11.40 може да се запише елегантно в матричен вид като:

$$\mathbf{A} \cdot \mathbf{y} = \mathbf{b} \tag{11.46}$$

където $\mathbf{A}$ е тридиагонална симетрична матрица с вектор на главният диагонал $\mathbf{D}$ и съответно $\mathbf{L}$ долен и $\mathbf{U}$ горни диагонали:

$$\mathbf{A} = \begin{bmatrix} D_1 & U_1 & 0 & \dots & 0 & 0 \\ L_1 & D_2 & U_1 & \dots & 0 & 0 \\ 0 & L_1 & D_3 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & D_{N-1} & U_{N-1} \\ 0 & 0 & 0 & \dots & L_{N-1} & D_N \end{bmatrix}$$

Примерен програмен код реализиращ решение на време зависимото уравнение на Шрьодингер по метода на Кранк-Николсон:

```
1  program tdse
2  implicit none
3
4  double precision, parameter :: PI = 3.14159265359d0
5  double complex, allocatable, dimension(:) :: Psi, PsiPrime
6
7  ! Hamiltonian matix definition since is hermitian we
8  ! define the matrix with main diagonal - d(n+1)
9  ! and first neighbour diagonal - e(n)
10 ! the right side of the equation (1 + i dt H / 2) * PsiNew = (1 - i dt
       H / 2) Psi
11 ! is vector b(n+1) = (1 - i dt H / 2) Psi
12
13 double complex, allocatable, dimension(:) :: D, DL, DU
14 double complex, allocatable, dimension(:) :: Pot
15 double complex :: Im, One, Zero
16 integer :: i, k, n, steps, cnt, info
17 double precision xmin, xmax, l0, T0, x, dx, t, dt, tout, sig, norm, xpmin,
       xpmax, Factor
18 double precision :: Ekin, Epot, V0, k0
19 double precision, allocatable, dimension(:) :: TrapCoef
20 character (len=32) :: filename
21
22 !Set domain size
23 xmin = -25.d0
24 xmax = 25.d0
25
26 !Set the potential energy barrier position left xpmin point
27 !and xpmax right point
28 xpmin = 4.d0
29 xpmax = 6.d0
30
31 !Wave packet initial position offset
32 l0 = -5.d0
33 !Wave packet spread
34 sig = 0.5d0
35 !Define constants
36 Im = cmplx(0.d0, 1.d0)
37 One = cmplx(1.d0, 0.d0)
38 Zero = cmplx(0.d0, 0.d0)
39
40 write(0,*)'Please␣eneter␣grid␣resolution␣dx␣=␣'
```

```
41  read(*,*)dx
42
43  norm = dx **2 / ( 1 + v0 * 2 * dx **2)
44
45  write(0,*)'Please␣enter␣time␣integration␣step␣(dt␣<␣', norm ,')␣dt␣=␣'
46  read(*,*)dt
47
48  write(0,*)'Please␣enter␣kinetic␣energy␣1/2␣>␣Ekin␣␣=␣'
49  read(*,*)Ekin
50
51  if ( Ekin.lt.0.5d0) then
52    write(0,*)'Kinetic␣energy␣should␣be␣>␣0.5'
53    stop
54  end if
55
56  !Calculate the impulse k0
57  k0 = dsqrt(2*Ekin-1.d0)
58
59  write(0,*)'Enter␣potential␣well␣deep␣V0␣=␣'
60  read(*,*)V0
61
62  write(0,*)'Enter␣integration␣period␣T␣=␣'
63  read(*,*)T0
64
65  write(0,*)'Enter␣write␣out␣period␣(Tout␣<␣',T0,')␣Tout␣=␣'
66  read(*,*)tout
67
68  n = nint((xmax - xmin) / dx) - 1
69
70  steps = tout / dt
71
72  !Calculate Factor multiplier
73  Factor = 0.5d0 * dt / dx ** 2
74
75  allocate(Psi(0:n+1))
76  allocate(PsiPrime(1:n))
77  allocate(Pot(0:n+1))
78  allocate(D(1:n))
79  allocate(DL(1:n-1))
80  allocate(DU(1:n-1))
81  allocate(TrapCoef(1:n))
82
83  !Initialise coefficients for Trapezoidal rule
84  TrapCoef(2:n-1) = 2.d0
85  TrapCoef(1) = 1.d0
86  TrapCoef(n) = 1.d0
87
88  do i = 0, n + 1
89    x = xmin + i * dx
90  !Initialise wave function
91    Psi(i) = (exp( -(x-l0) ** 2 / (4 * sig **2) ) / ( dsqrt(dsqrt (2 * PI) * sig )))
          * cmplx(cos(k0*(x-l0)), sin(k0*(x-l0)))
92  !Initialise the potential energy
93    if (x.ge.xpmin.and.x.le.xpmax) then
```

```
94        Pot(i) = cmplx(v0, 0.d0)
95      else
96        Pot(i) = cmplx(0.d0, 0.d0)
97      end if
98    end do
99
100   ! Apply boundary conditions
101   Psi(0) = Zero
102   Psi(n+1) = Zero
103
104   norm = dsqrt(dx*real(sum(Psi(:) * conjg(Psi(:)))))
105   Psi(:) = Psi(:) / norm
106
107   t = 0.d0
108   cnt = 0
109
110   do while ( t < T0)
111     !Solve the right side of Crank Nicolson equation
112     ! PsiRight(t) = (1 - 0.5 * H * dt) * Psi(t)
113     Psi(1:n) = (One - Im * Factor - Im * 0.5d0 * dt * Pot(1:n)) * Psi(1:n) + 0.5
          d0 * Im * Factor * (Psi(0:n-1) + Psi(2:n+1))
114
115     D(1:n) = One + Im * Factor + Im * 0.5d0 * dt * Pot(1:n)
116     DL(1:n-1) = - 0.5d0 * Im * Factor
117     DU(1:n-1) = - 0.5d0 * Im * Factor
118     !Solve linear system (1 + 0.5 * H * dt) * Psi(t+dt) = PsiRight(t)
119     !Crank Nicolson equation left side
120     call zgtsv(n, 1, DL, D, DU, Psi(1:n), n, info)
121
122     !Calculate first derivative of Psi
123     PsiPrime(1:n) = 0.5d0*(Psi(2:n+1) - Psi(0:n-1))/dx
124
125     !Calculate the kinetic energy
126     Ekin = 0.25d0*dx*(sum(TrapCoef(1:n)*PsiPrime(1:n)*conjg(PsiPrime(1:n))
          ))
127
128     !Calculate the potential energy
129     Epot = 0.5d0*dx*sum(TrapCoef(1:n)*conjg(Psi(1:n))*Pot(1:n)*Psi(1:n))
130
131     !Calculate the wave function norm
132     norm = 0.5d0*dx*sum(TrapCoef(1:n)*conjg(Psi(1:n))*Psi(1:n))
133
134     !Write data file every 'steps' count
135     if (mod(cnt,steps).eq.0) then
136       write(*,'(5F15.7)')t,Ekin,Epot,Ekin+Epot,norm
137       write(filename,'(A3I0.6A4)')'wf-',cnt,'.dat'
138       open(unit=10, file=filename)
139       write(10,'(A1F15.7)')'#',t
140       do i = 0, n + 1
141         x = xmin + i * dx
142         write(10,'(7F15.7)')x,real(Psi(i)), aimag(Psi(i)), cdabs(Psi(i)),real(Pot(i)
              ),Ekin,Epot
143       end do
```

```
144       close(10)
145     end if
146
147     t = t + dt
148     cnt = cnt + 1
149   end do
150
151   deallocate(Psi)
152   deallocate(PsiPrime)
153   deallocate(Pot)
154   deallocate(D)
155   deallocate(DL)
156   deallocate(DU)
157   deallocate(TrapCoef)
158
159   end program tdse
```

Може би е важно да отбележим ключовата част от кода, където се реализират уравнения от 11.41 до 11.45. Дясната страна на линейната система 11.46 или уравнение 11.41 се реализира като се презапише работният масив Psi:

```
1  Psi(1:n) = (One - Im * Factor - Im * 0.5d0 * dt * Pot(1:n)) * Psi(1:n) + 0.5d0
       * Im * Factor * (Psi(0:n-1) + Psi(2:n+1))
```

Инициализирането на диагоналните вектори D, DL и DU според уравнения 11.43, 11.44 и 11.45 става последователно с присвояването:

```
1  D(1:n) = One + Im * Factor + Im * 0.5d0 * dt * Pot(1:n)
2  DL(1:n-1) = - 0.5d0 * Im * Factor
3  DU(1:n-1) = - 0.5d0 * Im * Factor
```

Решаването на самата линейна система 11.46 се извършва като се прави извикване към процедурата zgtsv на библиотеката *LAPACK*.

За да компилирате и изпълните примерният изходен код, можете първо да клонирате хранилището *pde* от *Github*:

```
1  git clone http://github.com/pisov/pde.git
```

след което да компилирате и изпълните самият изходен код, като следвате инструкциите описани във файла parabolic/time.depend.schrodinger/1d/fortran/readme.txt

```
1  cd parabolic/time.depend.schrodinger/1d/fortran/
2  gfortran -O3 tdse.f90 -o tdse.x -llapack
3  rm -f wf*
4  ./tdse.x < input > energy.dat
5  ./makeanim.sh
```

Задача:

- Променете примерният изходен код tdse.f90 така, че формата на потенциалната бариера, която се описва с помощта на функцията $V(x)$ да бъде триъгълна с височина V_0 и център отново точката $x = 0$

- Променете граничните условия на задачата. В примерния код има наложени гранични условия на Дирихле с условието $\Psi(x_{min}) = \Psi(x_{min}) \equiv 0$. Нека новите гранични условия описват периодична функция, т.е. $\Psi(x) = \Psi(x + \Delta)$, където Δ е ширината на интервала $\Delta = x_{max} - x_{min}$.

11.4 Хиперболично частно диференциално уравнение

11.4.1 Едномерно вълново уравнение

Ще демонстрираме няколко числени алгоритъма за решаване на линейно хиперболично ЧДУ като разгледаме частния случай на едномерното вълново уравнение [9]:

$$\frac{\partial^2 u(x,t)}{\partial t^2} = v^2 \frac{\partial^2 u(x,t)}{\partial x^2} \tag{11.47}$$

където ще предолагаме, че v е скаларана константа. Горното уравнение може да се запише като система от две линейни ЧДУ от първи ред, използвайки следния запис

$$r(x,t) = v\frac{\partial u(x,t)}{\partial x} \tag{11.48}$$

$$s(x,t) = \frac{\partial u(x,t)}{\partial t} \tag{11.49}$$

Тогава уравнение 11.47 може да се запише като следната система от три ЧДУ от първи ред по времето:

$$\frac{\partial r}{\partial t} = v\frac{\partial s}{\partial x} \tag{11.50}$$

$$\frac{\partial s}{\partial t} = v\frac{\partial r}{\partial x} \tag{11.51}$$

$$\frac{\partial u}{\partial t} = s \tag{11.52}$$

В практиката съществуват няколко експлицитни схеми за интегриране на ЧДУ от първи ред по времето като: дискретизация срещу посоката на разпространение (Upwind), експлицитна (явна) форма - дискретизация на Ойлер напред във времето (forward Euler, FTCS), дискретизация с междинна стойност (Leapfrog), метод на Лакс-Фридих и Лакс-Вендроф. В настоящата секция ще разгледаме методите на Лакс-Фридих и Лакс-Вендроф, като последният се основава на дискретната схема на Лакс-Фридих. За да демонстрираме алгоритъма, нека първо разгледаме хиперболично ЧДУ от първи ред, което описва процес на адвекция със скорост v:

$$\frac{\partial u(x,t)}{\partial t} - v\frac{\partial u(x,t)}{\partial x} = 0 \tag{11.53}$$

В линейният случай ($v = constant$). Нека дискретизираме горното уравнение чрез дискретизационна схема на Ойлер напред във времето

$$\begin{aligned} \frac{u_i^{n+1} - u_i^n}{\Delta t} &= v\frac{\partial u^n}{\partial x}_i & (11.54)\\ u_i^{n+1} &= u_i^n + v\Delta t\frac{\partial u_i^n}{\partial x} & (11.55)\\ u_j^{n+1} &= u_j^n + \frac{v\Delta t}{2h}\left[u_{i+1}^n - u_{i-1}^n\right] & (11.56) \end{aligned}$$

Лакс и Фридих предлагат, за да се подобри числената стабилност на алгоритъма в уравнение 11.56, израза за стойността на решението u_i^n да се замени с неговата средна стойност между двете съседни функционални стойности в дискретизационната решетка (u_{i+1}^n, u_{i-1}^n), така че $u_j^n = (u_{i+1}^n + u_{i-1}^n)/2$. Тогава изразът 11.56 приема формата

$$u_i^{n+1} = \frac{1}{2}\left[u_{i+1}^n + u_{i-1}^n\right] + \frac{\alpha}{2}\left[u_{i+1}^n - u_{i-1}^n\right] \tag{11.57}$$

където с α ще записваме $\alpha = v\Delta t/h$. Отново за да бъде стабилен численият алгоритъм константата α трябва да изпълнява условието $\alpha < 1$. За да достигнем до схемата на дискретизация на Лакс-Вендроф трябва въведем допълнителна "средна" точка в интеграционната схема за моменти време $t_{n\pm1/2} = t_n \pm \Delta t/2$ и пространствени координати $x_{i\pm1/2} = x_i \pm h/2$. Тогава можем да включим в дискретизационната схема на Лакс-Фридих от уравнение 11.57 и междинните точки, така че да пресметнем стойността на функцията u за момент време $t_{n+1/2}$.

$$\begin{aligned} u_{i+1/2}^{n+1/2} &= \frac{1}{2}\left[u_{i+1}^n + u_i^n\right] + \frac{\alpha}{2}\left[u_{i+1}^n - u_i^n\right] & (11.58)\\ u_{i-1/2}^{n+1/2} &= \frac{1}{2}\left[u_i^n + u_{i-1}^n\right] + \frac{\alpha}{2}\left[u_i^n - u_{i-1}^n\right] & (11.59) \end{aligned}$$

Сега можем да комбинираме уравнения 11.58 и 11.59, за да намерим търсеното от нас решение за момент време t_{n+1}.

$$\begin{aligned} u_i^{n+1} = u_i^n + \alpha\left(u_{i+1/2}^{n+1/2} - u_{i-1/2}^{n+1/2}\right) = \\ = u_i^n + \frac{\alpha}{2}\left(u_{i+1}^n - u_{i-1}^n\right) + \frac{\alpha^2}{2}\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right) \end{aligned} \tag{11.60}$$

Горното уравнение описва дискретизационната схема на Лакс-Вендроф за ЧДУ от първи ред 11.53, описващо процес на адвекция. Сега ако се върнем на системата уравнения 11.50, 11.51 и 11.52. Можем да обърнем внимание че първите две уравнения 11.50 и 11.51 по същество са от същия тип уравнение, така че можем да презапишем дискретизационната схема на Лакс-Фридрих за тях:

$$r_i^{n+1} = \frac{1}{2}\left[r_{i+1}^n + r_{i-1}^n\right] + \frac{\alpha}{2}\left[s_{i+1}^n - s_{i-1}^n\right] \quad (11.61)$$

$$s_i^{n+1} = \frac{1}{2}\left[s_{i+1}^n + s_{i-1}^n\right] + \frac{\alpha}{2}\left[r_{i+1}^n - r_{i-1}^n\right] \quad (11.62)$$

Тогава решението на 11.50 и 11.50 по метода на Лакс-Вендроф 11.60 или

$$r_i^{n+1} = r_i^n + \frac{\alpha}{2}\left(s_{i+1}^n - s_{i-1}^n\right) + \frac{\alpha^2}{2}\left(r_{i+1}^n - 2r_i^n + r_{i-1}^n\right) \quad (11.63)$$

$$s_i^{n+1} = s_i^n + \frac{\alpha}{2}\left(r_{i+1}^n - r_{i-1}^n\right) + \frac{\alpha^2}{2}\left(s_{i+1}^n - 2s_i^n + s_{i-1}^n\right) \quad (11.64)$$

Последното уравнение 11.52 може да се реши тривиално с помощта на следната дискретизационна схема:

$$u_i^{n+1} = u_i^n + \frac{\Delta t}{2}\left(s_i^{n+1} + s_i^n\right) \quad (11.65)$$

11.4.1.1 Налагане на гранични условия

Числено решение изисква дискретизация на функциите r, s и u върху решетка от точки в координатното пространство с две допълнителни точки така, че ако $x_i = i * h + x_{min}$, тогава $x_0 = x_{min}$ и $x_{N+1} = x_{max}$, т.е. точките с индекси $i = 1, \ldots, N$ са вътрешни за дефиниционния интервал, а точките x_0 и x_{N+1} се явяват гранични. Стойността на функциите r, s и u в граничните точки се определя със специални правила, които се наричат гранични условия. Ще опишем два типа гранични условия: Дирихле (Установена постоянна стойност на функцията u на границата) и Зомерфелд. Граничните условия се налагат **само** за стойностите на функцията u, докато стойностите на r и s на границата се изчисляват.

11.4.1.2 Гранични условия на Дирихле

Налагането на граничните условия става на две стъпки. Първо е необходимо да се установят стойностите на функцията u на границите на интервала, т.е. $u(x_{min})$ и $u(x_{max})$. След което, използвайки уравненията 11.48 и 11.49, да се определят граничните стойности

$$s(x_{min}, t) = \frac{\partial u(x,t)}{\partial t}|_{x=x_{min}} = \frac{\partial u(x_{min})}{\partial t} \equiv 0 \quad (11.66)$$

$$s(x_{max}, t) = \frac{\partial u(x,t)}{\partial t}|_{x=x_{max}} = \frac{\partial u(x_{max})}{\partial t} \equiv 0 \quad (11.67)$$

$$r(x_{min}, t) = v\frac{\partial u(x,t)}{\partial x}|_{x=x_{min}} \quad (11.68)$$

$$r(x_{max}, t) = v\frac{\partial u(x,t)}{\partial x}|_{x=x_{max}} \quad (11.69)$$

горните уравнения могат да се запишат в дискретна форма така, че

$$\begin{aligned} s_0^{n+1} &= 0 \\ s_{N+1}^{n+1} &= 0 \\ r_0^{n+1} &= v\frac{u_0^{n+1} - u_1^{n+1}}{-h} \\ r_{N+1}^{n+1} &= v\frac{u_{N+1}^{n+1} - u_N^{n+1}}{h} \end{aligned} \tag{11.70}$$

11.4.1.3 Периодични гранични условия

Периодичните гранични условия са интересни в случаите, когато разглеждаме системи с "безкраен" размер, но притежават периодична структура, например кристали. Математическото условие, на което трябва да се подчнява решението $u(x,t)$, което търсим, е:

$$u(x+L,t) = u(x,t) \tag{11.71}$$

Тук L е размерът на областта на интерес. Ние приехме схема на дискретизация върху $N+2$ брой точки върху пространствената координата $x_i = ih$, така че $x_0 = x_{min}$ и $x_{N+1} = x_{max}$.

11.4.1.4 Зомерфелд

Примерен код: Числено решение на едномерното вълново ЧДУ $\partial^2 u/\partial t^2 = v^2 \partial^2 u/\partial x^2$ в областта $x \in [0,1]$ за даден интервал време $t \in [0, T_{end}]$. Формата на функцията в началният момент време е избрана с нормално подобно разпределение $u(x,0) = \exp^(-(x-1/2)^2/(2\sigma^2))$, центрирано около средната точка на интервала $[0,1]$. Входни параметри на програмата са

- `v` - скоростта на разпространение на едномерната вълна, (`0.1`)
- `np` - броят точки на дискретизация във вътрешния интервал. Общият брой точки на дискретизация е `np+2`, (`1023`)
- `dt` - стъпка на дискретизация по времето, (`0.001`)
- `Tend` - Периодът време на интегриране, (`5`)
- `Tout` - Периодът време на запис на функцията u , (`0.05`)

Инструкции за компилиране и изпълнение на примерния код:

1. Клонирайте хранилището *pde*, съдържащо примерния код и помощните скриптове с помощта на командите:

```
git clone http://github.com/pisov/pde.git
cd pde/hyperbolic/wave.eq/1d/fortran
```

2. Стартирайте скрипта `run.sh` с командата: `./run.sh`. В текущата папка ще се създадат серия от файлове с име `wave-NNNNNN.dat`, които съдържат функцията u за интеграционна стъпка NNNNNN.

3. Стартирайте скрипта `makeanim.sh` с командата `./makeanim.sh`, който създава анимация на еволюцията на функцията u във времето като анимиран GIF файл: `wave.gif`

```
program wave
implicit none

double precision, allocatable, dimension(:) :: u, unew, r, rnew, s, snew
double precision :: x, y, h, dt, t, tend, tout, v, Q
double precision :: xmin, xmax, alpha, sigma
integer :: i, j, k, nsteps, np, step, stepout
character (len=32) :: filename

xmin = 0.d0
xmax = 1.d0
sigma = 0.05d0

write(0,*)'Please␣enter␣velocity␣v␣=␣'
read(*,*)v

write(0,*)'Please␣enter␣number␣of␣points␣of␣discretization␣N␣=␣'
read(*,*)np

h = (xmax - xmin) / (np + 1)

write(0,'(A32F15.7A4)')'Please␣enter␣time␣step␣dt␣<␣(',h/v,')␣=␣'
read(*,*)dt

alpha = dt*v/h
Q = (1.d0 - alpha) / (1.d0 + alpha)

if (alpha.ge.1.d0) then
  write(0,*)'Wrong␣discretization␣parameters␣dt*v/h␣>␣1␣:',alpha
end if

write(0,*)'Please␣enter␣the␣total␣time␣period␣of␣integration␣Tend␣=␣'
read(*,*)Tend

write(0,*)'Please␣enter␣the␣writeout␣period␣Tout␣=␣'
read(*,*)Tout

stepout = nint(Tout / dt)

!Allocate memory arrays
allocate( u(0:np+1))
```

```
42  allocate(unew(0:np+1))
43  allocate( r(0:np+1))
44  allocate(rnew(0:np+1))
45  allocate( s(0:np+1))
46  allocate(snew(0:np+1))
47
48  !Set the initial state of r, s and u
49
50  do i = 0, np+1
51    x = i * h + xmin
52    u(i) = exp(-(x-0.5d0)**2/(2*sigma**2))
53  end do
54
55  s(:)=0.d0
56  r(1:np) = v * (u(2:np+1) - u(0:np-1)) / (2 * h)
57
58  !Main iteration loop
59  t = 0.d0
60  step = 0
61  do while ( t.le.tend )
62    !Evolve solution with one time step
63    ! evolve r
64    rnew(1:np) = r(1:np) + alpha*0.5d0*(s(2:np+1)-s(0:np-1)+alpha*(r(2:np
          +1)-2*r(1:np)+r(0:np-1)))
65    ! evolve s
66    snew(1:np) = s(1:np) + alpha*0.5d0*(r(2:np+1)-r(0:np-1)+alpha*(s(2:np
          +1)-2*s(1:np)+s(0:np-1)))
67    ! evolve u (leapfrog)
68    unew(0:np+1) = u(0:np+1) + 0.5d0*dt*(snew(0:np+1)+s(0:np+1))
69
70    rnew(np+1) = r(np) - rnew(np) * Q + r(np+1) * Q
71    rnew(0) = r(1) - rnew(1) * Q + r(0) * Q
72    snew(np+1) = s(np) - snew(np) * Q + s(np+1) * Q
73    snew(0) = s(1) - snew(1) * Q + s(0) * Q
74
75    if (mod(step,stepout).eq.0) then
76      write(filename,'(A5I0.6A4)')'wave-',step,'.dat'
77      open(unit=10, file=filename)
78      write(10,'(A1F15.7)')'#',t
79      do i = 0, np+1
80        x = i * h + xmin
81        write(10,'(7F15.7)')x,u(i)
82      end do
83      close(10)
84    end if
85
86    r(:) = rnew(:)
87    s(:) = snew(:)
88    u(:) = unew(:)
89
90    t = t + dt
91    step = step + 1
92  end do
```

```
93
94  !Deallocate memory arrays
95  deallocate(u)
96  deallocate(unew)
97  deallocate(r)
98  deallocate(rnew)
99  deallocate(s)
100 deallocate(snew)
101
102 end program wave
```

Задача 1: Модифицирайте примерния код на програмата `wave` така, че формата на началана функцията u да бъде триъгълник:

$$u(x,0) = \begin{cases} 0; 0 \leq x < 1/4 \\ 4(x - \frac{1}{4}); 1/4 \leq x < 1/2 \\ 4(\frac{1}{2} - x) + 1; 1/2 \leq x < 3/4 \\ 0; 3/4 \leq x < 1 \end{cases}$$

Задача 2: Модифицирайте вашия изходен код така, че да описва времевата еволюция на трептеното на струна с фиксирани краища на интервала $x \in [0, 1]$. Честотата на трептене f и "скоростта" на стоящата вълна v са свързани с равенството $f = \frac{v}{2L}$, където L е дължината на струната и в нашия случай е равна на 1. Началното условие на функцията нека бъде $u(x, 0) = \sin(\pi m x)$, $m = 1, 2, 3, \ldots$. Променете кода така, че да реализира гранични условия на Дирихле, използвайки уравнения 11.70, където $u(0) = u(1) = 0$. Нека параметъра $v = 1/2$ и $T_{end} = 4$. Подберете подходящи стойности за стъпката на дискретизация по времето `dt`.

11.4.2 Двумерно вълново уравнение

Нека разгледаме двумерната форма на линейно вълново ЧДУ:

$$\frac{\partial^2 u}{\partial t^2} = v^2 \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) \tag{11.72}$$

Отново както в едномерния случай, горното уравнение може да се презапише като система от едномерни ЧДУ от първи ред

$$\frac{\partial r}{\partial t} = v \frac{\partial s}{\partial x} \tag{11.73}$$

$$\frac{\partial l}{\partial t} = v \frac{\partial s}{\partial y} \tag{11.74}$$

$$\frac{\partial s}{\partial t} = v \left(\frac{\partial r}{\partial x} + \frac{\partial l}{\partial y} \right) \tag{11.75}$$

където функциите r, l и s се дефинират като

$$r = v\frac{\partial u}{\partial x} \tag{11.76}$$

$$l = v\frac{\partial u}{\partial y} \tag{11.77}$$

$$s = \frac{\partial u}{\partial t} \tag{11.78}$$

Ще разгледаме функциите r, l, s и u върху решетка от точки (x_i, y_j), където $x_i = ih_x + x_{min}$ и $y_j = jh_y + y_{min}$, за отделни моменти време $t^n = n\Delta t$. По-нататък ще използваме съкратеният запис $u^n_{i,j} = u(x_i, y_j, t_n)$

11.4.2.1 Схема на Лакс-Вендорф

Доколко извода на уравненията е усложнен, ще разгледаме дискретната схема на алгоритъма в краен вид:

$$r^{n+1}_{i,j} = r^n_{i,j} + \frac{\alpha_x}{2}\left(s^n_{i+1,j} + s^n_{i-1,j}\right) + \frac{\alpha_x^2}{2}\left(r^n_{i+1,j} - 2r^n_{i,j} + r^n_{i-1,j}\right) \tag{11.79}$$

$$l^{n+1}_{i,j} = l^n_{i,j} + \frac{\alpha_y}{2}\left(s^n_{i,j+1} + s^n_{i,j-1}\right) + \frac{\alpha_y^2}{2}\left(l^n_{i,j+1} - 2l^n_{i,j} + l^n_{i,j-1}\right) \tag{11.80}$$

$$\begin{aligned} s^{n+1}_{i,j} = {} & s^n_{i,j} + \frac{\alpha_x}{2}\left(r^n_{i+1,j} + r^n_{i-1,j}\right) + \frac{\alpha_y}{2}\left(l^n_{i,j+1} + l^n_{i,j-1}\right) \\ & + \frac{\alpha_x^2}{2}\left(s^n_{i+1,j} - 2s^n_{i,j} + s^n_{i-1,j}\right) + \frac{\alpha_y^2}{2}\left(s^n_{i,j+1} - 2s^n_{i,j} + s^n_{i,j-1}\right) \end{aligned} \tag{11.81}$$

където коефициентите α_x и α_y се дефинират като $\alpha_x = v\Delta t/h_x$ и $\alpha_y = v\Delta t/h_y$ съответно. Горната система уравнения 11.79 може да се запише в компактна форма:

$$r^{n+1}_{i,j} = r^n_{i,j} + \frac{\alpha_x}{2}D_x s^n + \frac{\alpha_x^2}{2}D_{xx} r^n \tag{11.82}$$

$$l^{n+1}_{i,j} = l^n_{i,j} + \frac{\alpha_y}{2}D_y s^n + \frac{\alpha_y^2}{2}D_{yy} l^n \tag{11.83}$$

$$s^{n+1}_{i,j} = s^n_{i,j} + \frac{\alpha_x}{2}D_x r^n + \frac{\alpha_y}{2}D_x l^n + \frac{\alpha_x^2}{2}D_{xx} s^n + \frac{\alpha_y^2}{2}D_{yy} s^n \tag{11.84}$$

където дискретните оператори D_x, D_y, D_{xx} и D_{yy} се дефинират чрез:

$$\begin{aligned} D_x\phi_{i,j} &= \phi_{i+1,j} - \phi_{i-1,j} \\ D_y\phi_{i,j} &= \phi_{i,j+1} - \phi_{i,j-1} \\ D_{xx}\phi_{i,j} &= \phi_{i+1,j} - 2\phi_{i,j} + \phi_{i-1,j} \\ D_{yy}\phi_{i,j} &= \phi_{i,j+} - 2\phi_{i,j} + \phi_{i,j-1} \end{aligned}$$

Глава 12

Търсене собствени вектори и собствени стойности

Задачата за търсене на собствени вектори и собствени стойности може да се запише в матрична форма с помощта на следното уравнение

$$\mathbf{A} \cdot \mathbf{x} = \lambda \mathbf{x} \tag{12.1}$$

където вектора $\mathbf{x}$ се нарича собствен вектор на матрицата $\mathbf{A}$, докато λ собствена стойност на вектора $\mathbf{x}$. Горното уравнение 12.1 може да бъде пренаписано като

$$(\mathbf{A} - \lambda \mathbf{1}) \cdot \mathbf{x} = \mathbf{0} \tag{12.2}$$

уравнението трябва да бъде в сила за всички двойки собствени стойности и вектори $(\lambda, \mathbf{x})$. Освен тривиалното решение $\mathbf{x} = 0$ съществуването на решение на уравнение 12.2 предполага че можем да намерим корените λ_i на следният полином

$$\det(\mathbf{A} - \lambda \mathbf{1}) = 0$$

В практиката съществуват множество алгоритми които реализират търсене на собствени вектори и собствени стойности. Поради сложността на численото реализиране на тези алгоритми ние ще разгледаме само метода на Якоби за намиране собствени вектори и собствени стойности. В примерните задачи по-долу ще изпползваме процедурите от библиотеката *LAPACK* за търсене на собствени вектори и собствени стойности на симетрични реални `dsyev` и комплексни ермитови матрици `zheev`. Интерфейс към процедурата `dsyev(JOBZ, UPLO, N, A, LDA, W, WORK, LWORK, INFO)`

- `JOBZ` - символ (`character*1`), входен параметър, възможните му стойности са `'N'`, тогава процедурата пресмята само собствените стойности или `'V'`, тогава процедурата освен собствените стойнсти пресмята и собствените вектори
- `UPLO` - символ (`character*1`), входен параметър, възможните му стойности са `'U'`, тогава "горните" елементи на матрицата `A` са записани или `'L'`, тогава "долните" елементи на матрицата са представени

- N - цяло число (integer), входен параметър, размерът на матрицата A, трябва N > 0
- A - входно/изходен двумерен масив от числа с двойна точност (double precision) с размерност (LDA, N). Съдържа матричните елементи в триъгълна форма в зависимост от стойността на параметъра UPLO = 'U' значими си горните елементи заедно с главният диагонал, UPLO = 'L' значими са долните елементи заедно с главният диагонал. При успешно приключване INFO = 0 в случаите когато JOBZ = 'V' мастрицата A съдържа собствените вектори
- LDA - цяло число (integer), входен параметър, водещата размерност на матрицата A, където LDA >= max(1,N)
- W - изходен едномерен масив от числа с двойна точност (double precision) с размер N. Ако INFO = 0 съдържа собствените стойности на матрицата A в нарастваща последователност
- WORK - едномерен работен масив от числа с двойна точност (double precision) с размер max(1, LWORK). След успешно приключване на операцията INFO = 0, WORK(1) съдържа оптималният размер на матрицата
- LWORK - цяло число (integer), размерът на работният масив WORK
- INFO - цяло число (integer), изходна променлива която съдържа код за приключване на операцията
  - INFO = 0 - успешна операция
  - INFO > 0 - INFO = i, алгоритъма не е сходящ, i- брой елемента извън главният диагонал не сходящи до 0
  - INFO < 0 - INFO = -i, i-тия аргумент на процедурата dsyev има грешна стойност

интерфейс към процедурата zheev(JOBZ, UPLO, N, A, LDA, W, WORK, LWORK, RWORK, INFO)

- JOBZ - символ (character*1), входен параметър, възможните му стойности са 'N', тогава процедурата пресмята само собствените стойности или 'V', тогава процедурата освен собствените стойнсти пресмята и собствените вектори
- UPLO - символ (character*1), входен параметър, възможните му стойности са 'U', тогава "горните" елементи на матрицата A са записани или 'L', тогава "долните" елементи на матрицата са представени
- N - цяло число (integer), входен параметър, размерът на матрицата A, трябва N > 0

- A - входно/изходен двумерен масив от комплексни числа с двойна точност (complex *16) с размерност (LDA, N). Съдържа матричните елементи в триъгълна форма в зависимост от стойността на параметъра UPLO = 'U' значими си горните елементи заедно с главният диагонал, UPLO = 'L' значими са долните елементи заедно с главният диагонал. При успешно приключване INFO = 0 в случаите когато JOBZ = 'V' мастрицата A съдържа собствените вектори
- LDA - цяло число (integer), входен параметър, водещата размерност на матрицата A, където LDA >= max(1,N)
- W - изходен едномерен масив от числа с двойна точност (double precision) с размер N. Ако INFO = 0 съдържа собствените стойности на матрицата A в нарастваща последователност
- WORK - едномерен работен масив от числа с двойна точност (double precision) с размер max(1, LWORK). След успешно приключване на операцията INFO = 0, WORK(1) съдържа оптималният размер на матрицата
- LWORK - цяло число (integer), размерът на работният масив WORK
- RWORK - едномерен работен масив от числа с двойна точност (double precision) с размер max(1, 3*N-2).
- INFO - цяло число (integer), изходна променлива която съдържа код за приключване на операцията
  - INFO = 0 - успешна операция
  - INFO > 0 - INFO = i, алгоритъма не е сходящ, i- брой елемента извън главният диагонал не сходящи до 0
  - INFO < 0 - INFO = -i, i-тия аргумент на процедурата dsyev има грешна стойност

12.1 Метод на Якоби

Този алгоритъм може да се използва в случаите когато търсим собствените вектори и собтсвените стойности на реални и симетрични матрици. Метода е числено стабилен лесен за реализация и ефективен за сравнително малки матрици [13, 11]. Основната идея на алгоритъма се състои в това че върху матрицата $\mathbf{A}$ се извършват серия от ортогонални трансформация с помощта на матрица на въртене $\mathbf{P} = \mathbf{P}(p, q, \Theta)$ които я привеждат в диагонален вид

$$\mathbf{P} = \begin{bmatrix} 1 & & & & & & & & \\ & \ddots & & & & & & & \\ & & 1 & & & & & & \\ & & & c & & s & & & \\ & & & & 1 & & & & \\ & & & -s & & c & & & \\ & & & & & & 1 & & \\ & & & & & & & \ddots & \\ & & & & & & & & 1 \end{bmatrix}$$

където $c = \cos(\Theta)$ и $s = \sin(\Theta)$, така че $P_{pp} = c$, $P_{qq} = c$, $P_{pq} = s$ и $P_{qp} = -s$

$$\tilde{\mathbf{A}} = \mathbf{P}^T(\Theta) \cdot \mathbf{A} \cdot \mathbf{P}(\Theta)$$

посоката на въртене се избира така че извъндиагоналните елементи на новата матрица $\tilde{\mathbf{A}}$ да бъдат по-малки или равни на нула

$$\tilde{A}_{pp}^2 + \tilde{A}_{qq}^2 \geq A_{pp}^2 + A_{qq}^2$$

т.е. диагоналните елементи увеличават стойността си за сметка на извъндиагоналните на всяка една итерация, като за дадена двойка (p, q) ъгълът Θ се определя на базата на следното правило

$$\begin{aligned} \tan(2\Theta) &= \frac{2A_{pq}}{A_{qq} - A_{pp}}, \ (A_{qq} \neq A_{pp}) \\ \Theta &= \frac{\pi}{4}, \ (A_{qq} = A_{pp}) \end{aligned}$$

ако предположим че сме извършили серия от ортогонални трансформации $\mathbf{P}_i$ така че сме довели матрицата $\mathbf{A}$ до диагонална форма $\mathbf{\Lambda}$

$$\mathbf{\Lambda} = \mathbf{P}^T \cdot \mathbf{A} \cdot \mathbf{P}$$

където общата ортогонална матрица $\mathbf{P}$ може да се дефинира като серия от ортогонални трансформации

$$\mathbf{P} = \mathbf{P}_1 \cdot \mathbf{P}_2 \cdot \mathbf{P}_3 \cdot \ldots$$

тогава диагоналните елементи λ_i на матрицата $\mathbf{\Lambda}$ отговарят на търсените собствени стойности на матрицата $\mathbf{A}$. Собствените вектори $\mathbf{x}_i$ на матрицата $\mathbf{A}$ могат да бъдат намерени с помощта на обратната трансформация

$$\begin{aligned} \mathbf{\Lambda} \cdot \mathbf{e}_i = \mathbf{P}^T \cdot \mathbf{A} \cdot \mathbf{P} \cdot \mathbf{e}_i &= \lambda_i \mathbf{e}_i \\ \mathbf{A} \cdot \mathbf{P} \cdot \mathbf{e}_i \cdot \mathbf{P}^T = \lambda_1 \mathbf{P} \cdot \mathbf{e}_1 \cdot \mathbf{P}^T &= \lambda_i \mathbf{x}_i \\ \mathbf{P} \cdot \mathbf{e}_i \cdot \mathbf{P}^T &= \mathbf{x}_i \end{aligned}$$

12.2 Задача - Квантови състояние на частица в едномерна потенциална яма с крайна дълбочина

Нека разгледаме задачата за намиране възможните квантово механични състояния хипотетична частица намираща се в едномерното пространство с координата x под действието на потенциал $V(x)$. Състоянието ѝ ще се описва с помощта на вълновата функция $\psi(x)$ за която е изпълнено нормировъчното условие:

$$\int_{-\infty}^{\infty} \psi^*(x)\psi(x)dx = 1$$

Потенциалът $V(x)$ на взаимодействие на частицата с форма на "яма" с крайна дълбочина V_0 може да се запише като:

$$V(x) = \begin{cases} 0 & |x| \leq L/2 \\ V_0 & |x| > L/2 \end{cases} \tag{12.3}$$

ще се интересуваме само от "ограничени" стационарни решения за които енергията на частицата E е по-малка от дълбочината на ямата V_0 и е изпълнено граничното условие $\psi(x)_{x\to\pm\infty} \to 0$. За целта е необходимо да намерим решение на едномерното стационарно уравнение на Шрьодингер:

$$\hat{H}\psi(x) = E\psi(x) \tag{12.4}$$

по същество горното уравнение поставя задачата за намирането на собствените функции $\psi(x)$ на оператора на Хамилтониана $\hat{H}$ със собствени стойности E. В този смисъл възможните стойности на енергията на частицата намираща се в потенциалната яма не са произволни и трябва да приемат дискретен набор от стойности. Хамилтонианът на системата $\hat{H}$ включва в себе си операторите на кинетичната $\hat{T}$ и потенциалната $\hat{V}$ енергии:

$$\hat{H} = \hat{T} + \hat{V}(x) = -\frac{\hbar^2}{2m}\frac{d^2}{dx^2} + V(x)$$

Разглежданата задача е интересна с това че може да бъде решена числено чрез няколко различни подхода. Като начало обаче можем да разгледаме аналитичното решение. След което ще обърнем и внимание на възможните числени методи и ще сравним резултатите от двата подхода.

12.2.1 Аналитично решение.

За да намерим решение на нашата задача за частица намираща се в потенциална яма с крайна дълбочина ще разглеждаме състоянието на вълновата функция $\psi(x)$ в три области на едномерното пространство къде потенциалът $V(x)$ има различни стойности:

$$\psi(x) = \begin{cases} \psi_1(x) & x < -L/2 \\ \psi_2(x) & |x| \leq L/2 \\ \psi_3(x) & x > L/2 \end{cases}$$

вълновите функции ψ_1 и ψ_3 са ненулеви защото дълбочината на потенциалната яма V_0 е крайна. В случай $V_0 \to \infty$ съответно $\psi_1 \equiv \psi_3 \to 0$. В който случай решението е тривиално $\psi_2^n(x) = \sin(\pi n x/L)$, където $n \in \mathbb{N}$.

Първо нека разгледаме случая когато частицата се намира "вътре" в потенциалната яма, т.е. $|x| \leq L/2$. Тогава използвайки дефиницията на оператора на потенциалната енергия $\hat{V}(x)$ стационарното уравнение на Шрьодингер 12.4 се записва като уравнение на свободна частица:

$$-\frac{\hbar^2}{2m}\frac{d^2}{dx^2}\psi_2(x) = E\psi_2(x)$$

ако положим израза $a = \sqrt{2mE}/\hbar > 0$ тогава горното уравнение може да се презапише като:

$$\frac{d^2}{dx^2}\psi_2(x) = -a^2\psi_2(x) \tag{12.5}$$

обикновенното диференциално уравнение 12.5 има аналитично решение което в най-общ вид може да се запише като:

$$\psi_2(x) = A\sin(ax) + B\sin(ax)$$

доколко в тази област потенциалната енергия е равна на нула. Енергията на частицата E е

$$E = \frac{a^2\hbar^2}{2m}$$

Сега може да разгледаме областите на движение "извън" потенциалната яма. Ще отбележим че от гледна точка на класическата физика тези области са "недостъпни" в случаите когато енергията на частицата е по-малка от дълбочината на потенциалната яма V_0 или $E < V_0$. От гледна точка на квантовата механика обаче съществува ненулева вероятно частицата да "проникне" и в тези области на конфигурационното пространство. Нека разгледаме случая когато $x < -L/2$ тогава уравнението на Шрьодингер 12.4 приема следната форма:

$$-\frac{\hbar^2}{2m}\frac{d^2}{dx^2}\psi_1(x) + v_0\psi_1(x) = E\psi_1(x)$$

прегруприрайки членовете които умножават $\psi_1(x)$ с E и V_0 можем да запишем горното уравнение във вида:

$$-\frac{\hbar^2}{2m}\frac{d^2}{dx^2}\psi_1(x) = (E - V_0)\,\psi_1(x) \tag{12.6}$$

уравнение 12.6 принципно има две различни решения в зависимост от знака на разликата $E - V_0$. В случай $E - V_0 > 0$ решението ще описва състояние на свободна чатица. Докато ако $E - V_0 < 0$ ще описва свързано състояние. Доколко както споменахме в началото на този раздел ние се интересуваме само от решения описващи "свързани" състояния ще разглеждаме случая $E - V_0 < 0$, тогава можем да декларираме величината α:

$$\alpha = \frac{\sqrt{2m\left(V_0 - E\right)}}{\hbar} \tag{12.7}$$

уравнението на Шрьодингер 12.7 може да се презапише като:

$$\frac{d^2}{dx^2}\psi_1(x) = \alpha^2\psi_1(x) \tag{12.8}$$

в който случай аналитичното решение в най-общ вид се записва като:

$$\psi_1(x) = Fe^{-\alpha x} + Ge^{\alpha x}$$

без да изпадаме в допълнителни детайли можем да запишем и решението за областта $x > L/2$ където режима на движение е същият:

$$\psi_3(x) = He^{-\alpha x} + Ie^{\alpha x}$$

Нека обобщим получихме аналитичнитя вид на функциите ψ_1, ψ_2 и ψ_3 които описват състоянието на частицата в областта вътре и извън потенциалната яма. Вълновите функциии обаче включват в себе си общо 6 неизвестни константи (A, B, C, D, E, F, G и I) както и двата параметъра a и α които са свързани с енергията на частицата E:

$$\begin{aligned}
\psi_1(x) &= Fe^{-\alpha x} + Ge^{\alpha x} \\
\psi_2(x) &= A\sin(ax) + B\sin(ax) \\
\psi_3(x) &= He^{-\alpha x} + Ie^{\alpha x}
\end{aligned}$$

Част от константите могат да се изключат ако наложим условието за непрекъснатост на вълновата функция $\psi(x)$ и нейната първа производна $\psi'(x)$ в двете гранични точки $-L/2$ и $L/2$:

$$\begin{array}{ll}
\psi_1(-L/2) = \psi_2(-L/2) & \psi_2(L/2) = \psi_3(L/2) \\
\frac{d}{dx}\psi_1(-L/2) = \frac{d}{dx}\psi_2(-L/2) & \frac{d}{dx}\psi_2(L/2) = \frac{d}{dx}\psi_3(L/2)
\end{array} \tag{12.9}$$

заедно с граничните условия $\psi(x) \to 0$ при $x \to \pm\infty$ води до две възможни фамилия решения на нашата задача:

- симетрична вълнова функция при $A = 0$ и $G = H$
- анти-симетрична вълнова функция за $B = 0$ и $G = -H$

Нека първо разгледаме случая на симетрично решение тогава системата 12.9 ще приеме формата:

$$\begin{array}{rcl} He^{-\alpha L/2} & = & B\cos(aL/2) \\ -\dfrac{\alpha}{2}He^{-\alpha L/2} & = & -\dfrac{a}{2}B\sin(aL/2) \end{array} \tag{12.10}$$

разделяйки горните две уравнения едно на друго можем да получим по-компактното алгебрично условие $\alpha = a\tan(aL/2)$. Следвайки същите стъпки ще да достигнем до аналогичен израз в случая на анти-симетрична вълнова функция, тогава $\alpha = -a\cot(aL/2)$. Да обобщим следни две връзки между коефициентите α и a в случаите на симетрично и анти-симетрични решения на вълновата функция $\psi(x)$:

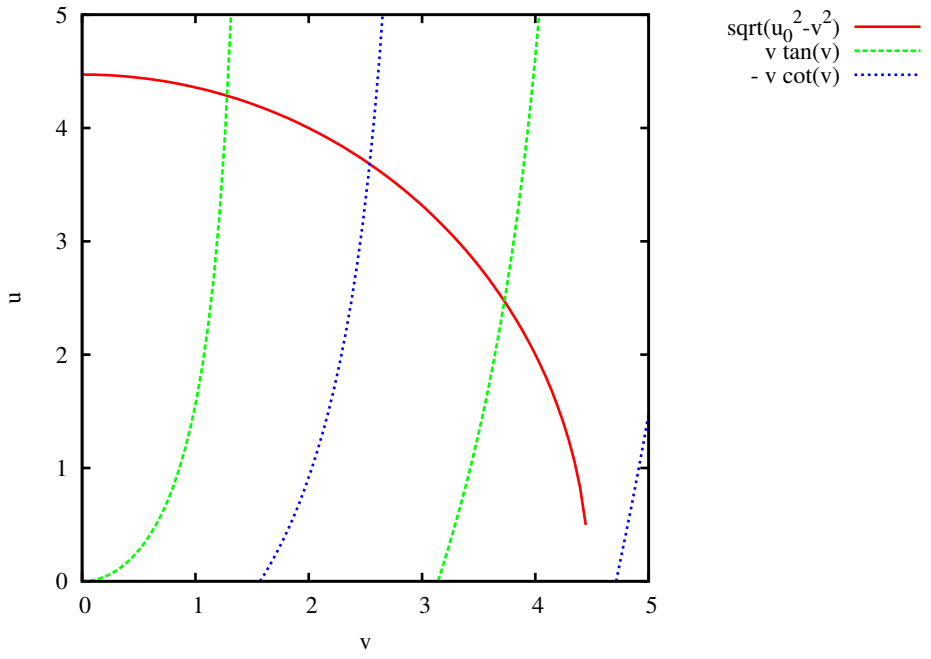
$$\begin{array}{rcl} \alpha & = & a\tan(aL/2) \\ \alpha & = & -a\cot(aL/2) \end{array} \tag{12.11}$$

за да намерим решение за коефициентите α и a обаче ни е необходима още една връзка, която можем да получим като си припомним дефиницията $\alpha = \dfrac{\sqrt{2m(V_0 - E)}}{\hbar}$ можем да запишем че $\alpha^2 = \left(\dfrac{2mV_0}{\hbar}\right)^2 - a^2$. След като направим смяна на променливите $u = \alpha L/2$ и $v = aL/2$, можем да запишем $u^2 = u_0^2 - v^2$, където $u_0^2 = \dfrac{mV_0L^2}{2\hbar^2}$ тогава системата 12.11 се пренаписва като:

$$\sqrt{u_0^2 - v^2} = \begin{cases} v\tan(v) \\ -v\cot(v) \end{cases} \tag{12.12}$$

Системата уравнения 12.12 е нелинейна и няма аналитично решение в общият случай. Затова можем да разгледаме "геометрично" решение което да ни даде израза за възможните стойности на енергията на частицата E. Разглеждаме координатната система (u, v) в която трябва да нарисуваме веднъж графиката на функцията $u(v) = \sqrt{v_0^2 - v^2}$ и след това двете фамилии графики $u(v) = vtan(v)$ в случаите на симетрични решения и $u(v) = -v\cot(v)$ в случаите на анти-симетрично решение. Пресечните точки между първата крива и двете фамилии криви ще ни даде възможните стационарни решения $\{u_i, v_i\}$ за разглежданата система 12.12. Например можем да разгледаме възможните решения за $V_0 = 10$ и $L = 2$ при нормировка $\hbar = m = 1$, $v_0^2 = 20$ и от графика 12.1 е видно че съществуват три възмони стационарни решения $v_1 = 1.28042$, $v_2 = 2.53809$ и $v_3 = 3.72713$, които съответно отговарят на енергии $E_1 = 0.819738$, $E_2 = 3.22095$ и $E_3 = 6.94575$.

Задача: Напишете числен код който използва някой от методите описан в Глава 5 за намиране възможните стойности на енергията на частицата намираща се в потенциална яма като V_0 и L се разглеждат като входни параметри. Сравнете резултатът от програмата ви за случаят на $V_0 = 10$ и $L = 2$.



Фигура 12.1: Графично решение за пресечните точки на системата уравнения 12.12 за случаят на $v_0^2 = 20$.

12.2.2 Числено решение - матричен подход.

Стационарното уравнение на Шрьодингер 12.4 може да се запише в матрична форма ако се възползваме от представянето на решението $\psi(x)$ в произволен пълен базис от ортонормирани вълнови функции $\psi_n(x)$. Условието за ортонормираност на функциите $\phi_n(x)$ води до:

$$\int_{-\infty}^{\infty} \phi_m^*(x)\phi_n(x)dx = \delta_{mn} = \langle\phi_m|\phi_n\rangle \tag{12.13}$$

в случаят с $\langle\psi|\phi\rangle$ ще записваме скаларното произведение между две произволни вълнови функции $\psi(x)$ и $\phi(x)$. Тогава можем да представим търсеното от нас решение $\psi(x)$ като следната безкрайна сума:

$$\psi(x) = \sum_{n=0}^{\infty} \mathbb{C}_n\phi_n(x) \tag{12.14}$$

комбинирайки уравнения 12.13 и 12.14 не е трудно да се покаже че:

$$\mathbb{C}_n = \langle\phi_n|\psi\rangle \tag{12.15}$$

Сега е момента да отбележим че познаването на пълният набор от коефициенти на разложение $\mathbb{C}_n$ в базиса на вълновите функции $\phi_n(x)$ дефинира еднозначно и взаимно обратимо функцията $\psi(x)$. Това свойство е съществено при численото решение на

задачата защото ние можем да представим решението на стационарното уравнение на Шрьодингер 12.4, функцията $\psi(x)$ само с нейните коефициенти $\mathbb{C}_n$ и да боравим само с тях. В практиката обаче винаги съществува физическо ограничение върху максималният брой коефициенти $\mathbb{C}_n$ които могат да се съхраняват в компютърната памет, което налага ограничаването или "орязването" на броя на базисните вълнови функции $\phi_n(x)$. По-нататък ще предполагаме че максималният брой вълнови функции $\phi_n(x)$ е ограничен до стойността N. Изборът на оптимална стойност на N се свързва със "сходимостта" на числено решение. Например ако няма съществено изменение в стойностите на енергията на частицата E при увеличаване на N. Нека сега разгледаме уравнение 12.4 в което запишем $\psi(x)$ по начина представен в уравнение 12.14:

$$\hat{H}\psi(x) = \sum_{n=0}^{N} \mathbb{C}_n \hat{H}\phi_n(x) = E \sum_{m=0}^{N} \mathbb{C}_m \phi_m(x)$$

за да елиминираме вълновите функции $\phi_n(x)$ от горното уравнение можем да изберем произволна вълнова функция $\phi_j(x)$ и вземем скаларното произведение $\langle \phi_j | \, \hat{H} \, \psi \rangle$ тогава:

$$\left\langle \phi_j | \hat{H}\psi \right\rangle = \sum_{n=0}^{N} \langle \phi_j | \, \hat{H} \, \phi_n \rangle \, \mathbb{C}_n = E\mathbb{C}_j$$

доколко горното уравнение е вярно за произволна функция $\phi_j(x)$ можем да запишем следната система от N-брой уравнения:

$$\sum_{n=0}^{N} H_{jn}\mathbb{C}_n = E\mathbb{C}_j$$

горното уравнение е компонентен запис на матричното уравнение за намирането на собствени стойности E и собствени вектори $\mathbf{c}$, $\mathbf{H} \cdot \mathbf{c} = E\mathbf{c}$. В случая $\mathbf{H}$ е ермитова матрица с размер $N \times N$ чийто компоненти се дефинират чрез:

$$H_{mn} = \langle \phi_m | \, \hat{H} \, \phi_n \rangle$$

докато компонентите $\mathbb{C}_n$ на вектора $\mathbf{c}$ се разглеждат като неизвестни. Решението на уравнение 12.2.2 може да се извърши с помощта някой от разгледаните по-горе в настоящата глава числени методи или с външна библиотека като *LAPACK* например. Но преди това пред нас стои избора на базис от вълнови функции $\phi_n(x)$ с помощта на които можем да конструираме матрицата на Хамилтониана $\mathbf{H}$.

12.2.2.1 Фурие базис. Плоски вълни, реализация с библиотеката за бързи Фурие трансформации FFT

За улеснение на решението ще разглеждаме решението в област от абцисната ос x ограничена в интервала $x \in [0, \Delta]$, където периодът Δ е достатъчно голям така че да включва областта с потенциалната "яма" в себе си, т.е. $L \ll \Delta$. Тогава функцията

на потенциалната енергия $V(x)$ дефинирана в уравнение 12.16 е необходимо да се транслира по в положителна посока на разстояние $\Delta/2$ или:

$$V(x) = \begin{cases} 0 & |x - \Delta/2| \leq L/2 \\ V_0 & |x - \Delta/2| > L/2 \end{cases} \tag{12.16}$$

тогата базисните функции $\phi_j(x)$ се дефинират като:

$$\phi_j(x) = \frac{1}{\sqrt{\Delta}} e^{2\imath\pi jx/\Delta}$$

където $j \in \mathbb{Z}$. Базисните функции $\phi_j(x)$ са периодични с период равен на Δ, така че $\phi(x) = \phi(x \pm \Delta)$. Периодът Δ очевидно има значение за формата на вълновите функции и трябва да се подбира по такъв начин че очакваното решение трябва да се "побира" в интервала Δ, в случая очакваме решението $\psi(x)$ на стационарното уравнение 12.4 да клони към 0 в краищата интервала или $\psi(0) = \psi(\Delta) \equiv 0$. Освен това базисните функции изпълняват и условието за ортогоналност:

$$\int_0^{\Delta} \phi_i^*(x)\phi_j(x)dx = \delta_{ij}$$

което е вярно за произволна двойка цели числа (i, j). Понякога този вид разложение по базови функции се нарича разложение в базис на плоски вълни. Тогава факторът $k = 2j\pi/\Delta$ придобива смисъл на момент на импулса на плоска вълна с вълнова функция $\phi_k(x) = e^{\imath kx}/\sqrt{\Delta}$. Добрата новина е че в този базис матрицата на оператора на кинетичната енергия $\hat{T}$ е диагонална. Това може да се покаже много лесно като за произволно m разгледаме скаларното произведение $\langle\phi_i|\,\hat{T}\,\phi_j\rangle$:

$$\langle\phi_i|\,\hat{T}\,\phi_j\rangle = -\frac{\hbar^2}{2m}\int_{-\Delta/2}^{\Delta/2} \phi_i^*(x)\frac{d^2}{dx^2}\phi_j(x)dx = \frac{\hbar^2}{2m}\left(\frac{2\pi j}{\Delta}\right)^2 \delta_{ij}$$

Пресмятането на матричните елементи на оператора на потенциалната енергия $\hat{V}(x)$ обаче изисква намирането на интеграла:

$$\langle\phi_i|\,\hat{V}\,\phi_j\rangle = \int_{-\Delta/2}^{\Delta/2} \phi_i^*(x)V(x)\phi_j(x)dx \tag{12.17}$$

При числено решение на задачата ще дискретизираме вълновата функция върху краен набор от точки както и ще ограничим на броят на базисните функции до N на брой. Тогава индексите на базисните функции ϕ_j ще приемат стойности: $j = 0, \ldots, N-1$. Също така можем да дискретизираме точките в координатното пространство, така че $x_m = i\Delta/N$, като индексът m приема същите стойности $j = 0, \ldots, N-1$. Представянето на произволна функция $\psi(x)$ в този базис от N-брой функции ϕ_j в дискретен набор от точки x_m може да се запише с помощта на следното равенство:

$$\psi(x_m) = \sum_{j=0}^{N-1} \mathbb{C}_j\phi_j(x_m) = \frac{1}{\sqrt{\Delta}}\sum_{j=0}^{N-1} \mathbb{C}_j e^{2i\pi jx_m/\Delta} = \frac{1}{\sqrt{\Delta}}\sum_{j=0}^{N-1} \mathbb{C}_j e^{2i\pi jm/N}$$

т.е. ако разглеждаме произволна функция $\psi(x)$ в така подбраният набор от дискретни точки x_m, тогава представянето на функцията е еквивалентно на дискретна Фурие трансформация **DFT** според уравнение 10.14. Тогава коефициентите $\mathbb{C}_j$ имат двоен смисъл веднъж могат да се интерпретират като дискретният Фурие образ на функцията $\psi(x)$ и втори път имат смисъл на коефициенти на разложение върху базисните функции (уравнение 12.15). Нека намерим как се представят матричните елементи на операторите на кинетичната T_{mn} и потенциална енергии V_{nm}. За целта нека запишем уравнение 12.17 като представим в явен вид базисните функции $\phi_j(x)$:

$$\begin{aligned} V_{mn} = \langle \phi_m | \hat{V} \phi_n \rangle &= \int_0^{\Delta} \frac{1}{\sqrt{\Delta}} e^{2\imath\pi - mx/\Delta} V(x) \frac{1}{\sqrt{\Delta}} e^{2\imath\pi nx/\Delta} dx \\ &= \frac{1}{\Delta} \int_0^{\Delta} V(x) e^{2\imath\pi(n-m)x/\Delta} dx \\ &\approx \frac{1}{\Delta} \sum_{k=0}^{N-1} V(x_k) e^{2\imath\pi(n-m)x_k/\Delta} \delta x \\ &= \frac{1}{N} \sum_{k=0}^{N-1} V(x_k) e^{2\imath\pi(n-m)k/N} \end{aligned} \tag{12.18}$$

по-горе в приближеният израз за интеграла 12.18 предполагаме $dx \approx \delta x = \Delta/N$, доколко $m, n = 0, \ldots, N-1$, тогава можем да дефинираме целочисленият индекс j така че $n - m = j = -(N-1), \ldots, N-1$ ако за момент разгледаме само горно диагоналните матрични елементи на ермитовата матрица V_{mn} включително главният диагонал, т.е. тези елементи за които $n \geq m$. Тогава индекса j ще приема стойнсоти $j = 0, \ldots, N-1$ и горното уравнение 12.18 може да се интепретира като Фурие образа на функцията $V(x)$ или:

$$V_{mn} = \frac{1}{N} \mathbb{V}_j = \frac{1}{N} \sum_{k=0}^{N-1} V(x_k) e^{2\imath\pi jk/N} = \frac{1}{N} \sum_{k=0}^{N-1} V_k e^{2\imath\pi jk/N}$$

където съответстието между индекстите (m, n) на матрицата и j-тия елемент на фурие образа $\mathbb{V}$ може да се запише с помощта на равенството $j = n - m$, така достигаме до окончателнитя израз за матричните елементи на оператора на потенциалната енергия $\hat{V}$:

$$V_{mn} = \begin{cases} \frac{1}{N} \mathbb{V}_{n-m}, n \geq m \\ \frac{1}{N} \mathbb{V}^*_{n-m}, n < m \end{cases}$$

Матричните елементи на оператора на кинетичната енергия T_{mn} могат да бъдат пресметнати също с помощта на дискретни фурие трансформации. Като начало нека намерим как действа втората производна върху експоненциалният фактор $W_{j,k} = e^{2\imath\pi j x_k/\Delta} = e^{2\imath\pi jk/N}$

$$\begin{aligned}\frac{d^2}{dx^2}W_{jk} = D_{xx}W_{jk} & \\ &= \frac{1}{\delta x^2}\left(W_{j,k-1} - 2W_{j,k} + W_{j,k+1}\right) \\ &= \frac{N^2}{\Delta^2}\left(e^{2\imath\pi jk/N} + e^{-2\imath\pi jk/N} - 2\right)W_{j,k}\end{aligned}$$

тогава матричните елементи $T_{m,n}$ могат да бъдат намерени като

$$\begin{aligned}T_{mn} = \langle\phi_m|\,\hat{T}\,\phi_n\rangle &= -\frac{1}{2}\int_0^{\Delta}\frac{1}{\sqrt{\Delta}}e^{-2\imath\pi mx/\Delta}\frac{d^2}{dx^2}\frac{1}{\sqrt{\Delta}}e^{2\imath\pi nx/\Delta}dx \\ &\approx -\frac{1}{2}\frac{1}{\Delta}\sum_{k=0}^{N-1}W_{-m,k}D_{xx}W_{n,k}\delta x \\ &= -\frac{1}{2}\frac{N}{\Delta^2}\sum_{k=0}^{N-1}\left(W_{-m,k}e^{2\imath\pi n/N}W_{n,k} + W_{-m,k}e^{-2\imath\pi n/N}W_{n,k} - 2W_{-m,k}W_{n,k}\right) \\ &= \frac{1}{2}\frac{N^2}{\Delta^2}\left(2 - e^{2\imath\pi n/N} - e^{2\imath\pi - n/N}\right)\delta_{m,n}\end{aligned} \tag{12.19}$$

с условието за ортогоналност $\delta_{m,n} = \frac{1}{N}\sum_{k=0}^{N-1}W_{-m,k}W_{n,k}$.

Задача: Реализирайте числен код написан на програмният език *Fortran 90* който намира решение на матричното уравнение 12.2.2 в случай на потенциална яма с възможност потребителят да въвежда като входни параметри дълбочината на ямата V_0, нейната широчина L (`2*length`), както и периодът Δ (`Delta`) на функциите на Фурие. Задачата се разглежда нормировка $\hbar = m = 1$. Потребителят също така трябва да може да въвежда като входни параметри броя на базисните функции N (`n = 2048`) или максималната кинетична енергия на базисните вълнови функции *cutoff* $E_{cutoff} = 2\pi^2N^2/\Delta^2$, тогава $N = \sqrt{\frac{E_{cutoff}}{2}}\frac{\Delta}{\pi}$.

Решение:

```
!
! Finite Well calculation calculation using matrix approach
! with harmonic oscillator basis set
! developed by Dr. Stoyan Pisov 2012
!
program finitewell
implicit none
include "fftw3.f"

double precision, parameter :: pi = 3.14159265358979323846264338327d0
! N - Size of basis set, should be odd number
```

```
12  integer, parameter :: n = 2048
13  ! lwork - work variable required by ZHEEV subroutine
14  integer, parameter :: lwork = 3*n - 1
15  ! Length - half of the distance of finite well
16  double precision, parameter :: length = 1.d0
17  ! Define imaginary number
18  complex*16, parameter :: img = ( 0.d0, 1.d0)
19  ! Define complex number one
20  complex*16, parameter :: one = ( 1.d0, 1.d0)
21
22
23  ! X - Vector with discrete points
24  double precision, dimension(n) :: x
25
26  ! HF - array containing first 0 - N harmonic oscillator basis
27  ! function discretized over M points
28  ! BWF - calculated basis function of finite well
29  complex*16, dimension(n,n) :: bwf, pbwf, sbwf
30  double precision, dimension(n,n) :: prbf
31
32  !Work array for calculating the potential energy matrix with FFT
33  complex*16, dimension(n) :: invec, outvec
34
35  ! work variable for creating FFT plan
36  ! plan - forward plan
37  ! iplan - backward plan
38  integer*8 :: plan, iplan
39
40  ! Epot - N x N potential energy matrix calculated in oscillator basis
41  ! Ekin - N x N kinetic energy matrix calculated in oscillator basis
42  ! Etot - N x N total energy matrix calculated in oscillator basis
43  complex*16, dimension(1:n,1:n) :: epot,ekin,etot
44  ! Energies - Eigenvalue vector of Etot
45  double precision, dimension(1:n) :: energies, epotf
46  ! work - array required by ZHEEV
47  double precision :: work(lwork), rwork(lwork)
48
49  !Potential energy level V0
50  double precision, parameter :: V0 = 10.d0
51
52  !work variables
53  double precision :: xmin, xmax, Delta, step
54  integer :: i, j, info, np, nbo
55  complex*16 :: integ
56
57  character*1 :: jobz, uplo
58
59  ! We request eigenvector as well
60  jobz = 'V'
61  uplo = 'U'
62
63  ! Define function range
64
65  Delta = 20.d0
66  xmin = 0.d0
```

```
67  xmax = Delta
68
69  ! Calculate discretization step
70  step = Delta /n
71
72  ! Initialize coordinate vector X
73  do i = 1, n
74    x(i) = (i-1) * step + xmin
75  end do
76
77  ! Calculate potential energy profile finite well with limit equal V0
78  invec(:) = cmplx(V0, 0.d0)
79  do i = 1, n
80    if(abs(x(i)-0.5*Delta).le.(length)) then
81      invec(i) = (0.d0, 0.d0)
82    end if
83  end do
84
85  epotf(:) = real(invec(:))
86
87  ! Create FFT forward plan
88  call dfftw_plan_dft_1d( plan, n,invec, outvec,fftw_forward,
        FFTW_ESTIMATE)
89
90  ! Create FFT backward plan
91  call dfftw_plan_dft_1d(iplan, n,outvec, invec,fftw_backward,
        FFTW_ESTIMATE)
92
93  ! Execute the plan itself
94  call dfftw_execute(plan)
95
96  ! Norm the array
97  outvec(:) = outvec(:) / n
98
99  do i = 1, n
100   do j = i, n
101     epot(i,j) = outvec(j - i + 1)
102     if (i.ne.j) then
103       epot(j, i) = dconjg(outvec(j - i + 1))
104     end if
105   end do
106 end do
107
108
109 ekin(:, :) = cmplx(0.d0, 0.d0)
110 do i = 1, n
111   ekin(i, i) = (one - 0.5d0*exp(2*img*pi*i/n) - 0.5d0*exp(-2*img*pi*i/n))*n
        **2/Delta**2
112 end do
113
114 ! Calculate Epot, Ekin, Etot matrices
115 etot(:, :) = ekin(:, :) + epot(:, :)
116
117 ! External LAPACK subroutine call return eigenvalues and
```

```
      eigenvectors of Etot
118 call zheev (jobz, uplo, n, etot, n, energies, work, lwork, rwork, info)
119
120 ! Calculate finite well eigenfunctions
121 do i = 1, n
122   outvec(:) = etot(:, i)
123   call dfftw_execute(iplan)
124   sbwf(:, i) = invec(:) / sqrt(Delta)
125   prbf(:, i) = cdabs(sbwf(:, i))**2 + energies(i)
126 end do
127
128 ! Print out the data
129 ! Print first five eigenvalues to standard output
130 do i = 1, 5
131   write(0,'(A1,I1,A3,F15.7)')'E',i,' = ',energies(i)
132 end do
133
134 ! Print first four eigenvectors and potential energy profile
135 ! to standard error output
136 do i = 1, n
137   write(6,'(6F15.7)')x(i),prbf(i,1),prbf(i,2),prbf(i,3),prbf(i,4),epotf(i)
138 end do
139
140 end program finitewell
```

примерната програма може да се компилира с помощта командата

```
1 git clone http://github.com/pisov/eigenvalues.and.eigenvectors.git
2 cd eigenvalues.and.eigenvectors/finite.well/fft/fortran/
3 gfortran -O3 fw.f90 -o fw.x -I/usr/inlcude -llapack -lfftw3
```

и да се изпълни с

```
1 ./fw.x > plot.dat
2 gnuplot fwplot.gnu
```

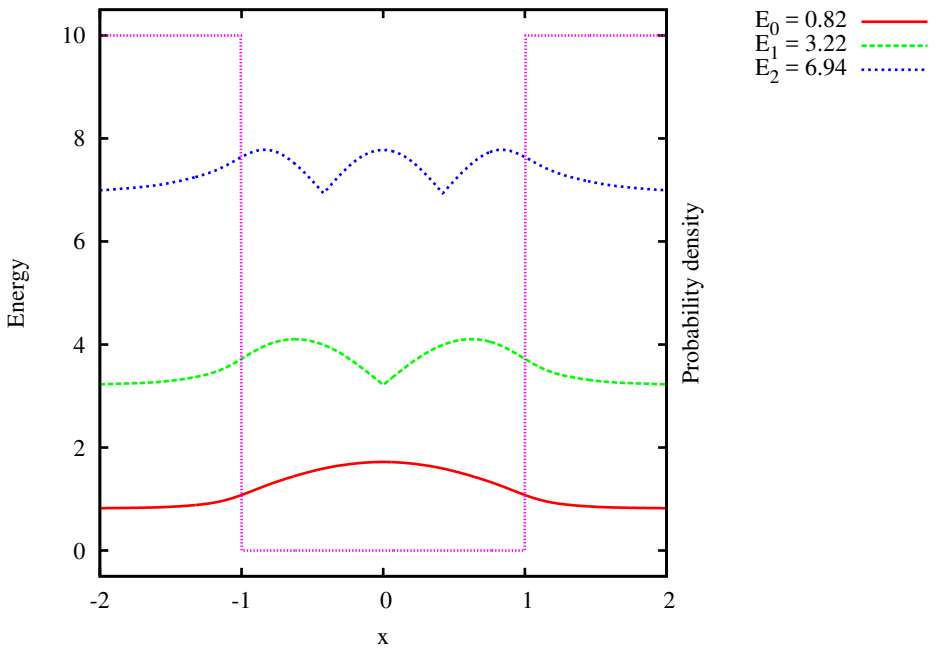
Задача: Модифицирайте програмният код за решаване задачата за едномерна частица намираща се в потенциална яма с крайна дълбочина V_0 така че да пресметне собствените състояния в случаят на хармоничен потенциал $V(x) = \frac{x^2}{2}$. Аналитичното решение за енергетичните състояние се дава с израза $E_n = n + \frac{1}{2}$, $n = 0, 1, 2, \ldots$. Сравнете резултатите от численото решение с аналитичният израз.

12.2.2.2 Базис на хармонични вълнови функци.

Друг удобен базис който може да се използва за решение на задачата за хармоничните функции решение които се явяват решение на задачата за частица в хармоничен потенциал:

$$\psi_n(x) = \frac{1}{\sqrt{2^n n!}} \pi^{-1/4} e^{-x^2/2} H_n(x) \tag{12.20}$$

където $H_n(x)$ представлява полином на Ермит от ред n:



Фигура 12.2: Графика на плътността на вероятност за първите три стационарни състояния на задачата за частица намираща се в потенциална яма с крайна дълбочина $V_0 = 10$, $L = 2$, $\Delta = 40$ и нормировка $\hbar = m = 1$. Решението е получен в базис на плоски вълни.

$$
\begin{aligned}
H_0(x) &= 1 \\
H_1(x) &= 2x \\
H_2(x) &= 8x^3 - 12x \\
H_3(x) &= 16x^4 - 48x^2 + 12 \\
&\dots
\end{aligned}
$$

Полиномите на Ермит могат да се дефинират чрез следната рекуретна зависимост:

$$H_{n+1}(x) = 2xH_n(x) - H'_n(x) \tag{12.21}$$
$$H'_n(x) = 2nH_{n-1}(x) \tag{12.22}$$

В най общият случай за произвлен полином на Ермит можем да запишем че:

$$H_n(x) = \sum_{i=0}^{n} A_i^n x^i \tag{12.23}$$
$$H'_n(x) = \sum_{i=0}^{n} -1(i+1)A_{i+1}^n x^i \tag{12.24}$$

За пресмятането на коефициентите A^n може да се използва примерната подпрограма hermite_coeff написана на програмният език *Fortran 90*:

```
!************************************************
!* Hermite polynomial coefficients evaluation by *
!* means of recursion relation. The order of the *
!* polynomial is n. The coefficients are returned *
!* in A(i). *
!************************************************
subroutine hermite_coeff(n,A,B)
  integer, intent(in) :: n
  double precision, dimension(:), intent(inout) :: A
  double precision, dimension(:,:), intent(inout) :: B

  integer :: i, j

  !Establish l0 and l1 coefficients
  B(0,0)=1.d0 ; B(1,0)=0.d0 ; B(1,1)=2.d0
  !Return if order is less than two
  if (n>1) then
    do i = 2, n
      B(i,0)=-2.d0*(i-1)*B(i-2,0)
      do j = 1, i
        !Basic recursion relation
        B(i,j)=2.d0*B(i-1,j-1)-2.d0*(i-1)*B(i-2,j)
      end do
    end do
    do i = 0, n
      A(i)=B(n,i)
    end do
  end if
  return
end
```

Задача: Реализирайте числен код написан на програмният език *Fortran 90* който намира решение на матричното уравнение 12.2.2 в случай на потенциална яма с възможност потребителят да въвежда като входни параметри дълбочината на ямата V_0, нейната широчина L, използвайки базис от хармонични функции 12.20. Задачата се разглежда нормировка $\hbar = m = 1$. Потребителят също така трябва да може да въвежда като входни параметри броя на базисните функции N, както и броят на точките M или стъпката dx на дискретизация.

Глава 13

Монте Карло методи

Монте Карло е понятие в числените методи което се свързва с изпозлването на поредица от случайни числа. Исторически този метод е развит през 40-те години на миналият век от Станислав Улам, Джон фон Нойман, Енрико Ферми и Никола Метрополис. Работейки върху проектът по създаването на първата атомна бомба, наречен "Манхатън". Физиците се сблъскали със задачата по определянето на пробега неутронното "лъчение" през различни среди. Въпреки че среднят пробег между два последователни сблъсъка на неутрон с атом от средата, както и количеството енергия които той губи са били известни. Тази задача била нерешима с помощта на конвенциналните детерминистични математични модели. Тогава възниква идеята да се "проиграе" многократно преминаването на неутрон през средата моделирайки последователни сблъсъци с атомите в нея, като резултат се определяло усредненото разстояние необходимо на неутроните да се "термализират". Този модел бил програмиран от Джон фон Нойман върху компютъра *ENIAC*. Разработеният метод е трябвало да има засекретено име и затова фон Нойман избрал името на известното казино *Монте Карло* в Монако. В днешни дни Монте Карло методите имат широк кръг от приложение например при численото интегриране, генерирането на вероятностни разпределения, квантовата механика, статистическата физика и моделиране процеса на преминаване на йонизиращо лъчение през среда (радиационен транспорт). В няколко глави по-долу ще разгледаме основните техники които реферират към Монте Карло методите. Като ще започнем нашето представяне с генераторите на случайни числа. Получаването на множество случайно разпределни числа не е тривиална задача от изчислителна гледна точка и трябва да се разглежда като съществена част от всеки един Монте Карло метод.

13.1 Генератори на случайни числа

Случайните числа се определят с това че те не могат да бъдат предсказани. Например, ако генерираме последователност от случайни числа. Вероятностното разпределение на всяко ново число в поредицата е напълно независимо от всички предходни случай-

ни числа. Тази концепция може да се онагледи с последователно хвърляне на зарчета. Вероятността за събитието да се "изтегли" числото 3 например е напълно независима от всички числа които са се получили дотогава. Подобни случайни процеси могат да се наблюдават и в природата. Например всяко радиоактивно ядро има определена вероятност да се разпадне. Но никой не може да каже кога точно това ще се случи. От друга страна случайните числа генерирани от компютрите не са "истински" случайни в този смисъл на думата. Повечето алгоритми използват вече генерираните случайни числа за да получат новите с помощта на подходяща математична формула. Това означава че всяко случайно число може да се определи знаейки предходните стойности. Въпреки това генерираните числа с помощта на тези алгоритми често имат задоволителни свойства и могат да бъдат използвани като "случайни" числа. В този смисъл на думата генерираните случайни числа с помощта на компютрите се наричат *псевдо-случайни* числа. Един добър генератор на случайни числа се очаква че трябва да генерира последователност която трудно се отличава от истинска извадка случайни числа. За удобство по-нататък ще пропускаме представката "псевдо" пред случайните числа. Повечето стандартни генератори на случайни числа са равновероятно резпределени в някакъв интервал, например интервала $[0, 1]$. Това означава че всяко число в интервала между 0 и 1 има една и съща вероятност да се "случи". На практика обаче само едно подмножество от числата в интервала $[0, 1]$ могат да бъдат реализирани поради крайното представяне на числата с плаваща запетая. Все пак можем да приемем че плътността на това под-множество числа е достатъчно за повечето задачи. Можем да дефинираме функцията на плътността $P(x)$ така че произведението $P(x)dx$ да ни дава вероятността да намерим случайно число в интервала $[x, x + dx]$. В случай на равномерно разпределение $P(x)$ се дефинира като:

$$P(x) = \begin{cases} 1 & x \in [0, 1] \\ 0 & x \notin [0, 1] \end{cases} \tag{13.1}$$

Както споменахме по-горе важен критерий за качеството на един генератор на случайни числа е отсъствието на "корелация" (или връзка) между последователните стойности които се генерират. Това условие може да се изрази с по-сложна функция на разпределение $P(x_i, x_{i+1})$, която дава вероятността за две последователни числа x_i и x_{i+1} да се "случат". Отсъствието на корелация налага условието:

$$P(x_i, x_{i+1}) = P(x_i)P(x_{i+1}) = 1 \tag{13.2}$$

подобно условие се налага и върху всеки две произволно отдалечени в редицата случайни числа x_i и x_j, така че $P(x_i, x_j) = 1$ за $j > i$.

В компютърната памет псевдо-случайните числа се преставят чрез краен брой битове. Това означва че в най-общият случай тези числа могат да се интерпретират като цели. Ако тези случайни числа са разпределени равномерно, ние можем да получим разпределение 13.1 просто като преместим най-малкото цяло число в редицата в 0 и делим на най-голямото цялото число. Може би най-широко разпространеният алгоритъм за генерериране на такава редица от псевдо-случайни е линейно-конгруетният

метод. За дадени цели числа a, c и m, можем да конструираме рекурентна зависимост за генериране на цяло случайно число x_i използвайки предишното такова:

$$x_i = (ax_{i-1} + c) \mod m \tag{13.3}$$

Началното число x_0 от което стартира цялата поредица числа x_0 се нарича засявка (*seed*). Можем да получим реално числа чрез така получените цели стойности x_i, просто като ги разделим на m. Понякога c може да бъде избрано да бъде равно на 0. Началната стойност на x_0 трябва да бъде по-голяма от 0 за да изключим израждане на поредицата генерирани случайни числа. Важна особеност на този метод е "цикличността", т.е. след определен брой генерирани цели числа x_i ние получаваме число което вече е "изтеглено", което ще повтори отново поредицата. В общият случай рекурентната формула 13.3 може да генерира общо $m-1$ на брой различни числа, но това изисква специален подбор на коефициентите a и m. Например нека разгледаме случая $a = 12$ и $m = 143$. В такъв случай:

$$x_{i+1} = 12x_i \mod 143 \tag{13.4}$$
$$x_{i+2} = 12x_{i+1} \mod 143 \tag{13.5}$$
$$= 144x_i \mod 143 = x_i \tag{13.6}$$

в който случай цикличността на получената поредица ще бъде само 2 за произволна начална стойност x_0. Съществуват следните критерии при подбора на коефициентите a, m, c и засявката x_0, които гарантират максимална дължина (период) на генерираната редица случайни числа:

- x_0 трябва да бъде просто число по отношение на m, т.е. x_0 и m не трябва да имат общи делители
- a е необходимо да бъде примитивен елемент на модула m, което значи че, числото a трябва да притежава най-големият показател по модул m. В такъв случай числата a и m са взаимно прости и показателят на m, който можем да означим с λ, изпълнява условието $a^\lambda \mod m = 1$.

Максималната дължина на редицата случайни числа в този случай ще бъде λ на примитивният елемент a. Ако m е просто число тогава максималната дължина е $m-1$. В случай че изберем обаче $m = 2^r$, максималната дължина на редицата числа ще бъде 2^{r-1}

13.2 Вградени генератори на случайни числа

Всеки един програмен език от високо ниво има своя реализация на генератор на случай числа. Свойствата на тези генератори на случайни числа като: цикличност,

гъстота на числата и корелация, могат да се различават от версията на компилатора на програмният език, т.е. няма гаранция че една и съща библиотечна функция на даден генератор ще има едни и същи свойства. Затова използването на вградени генератори трябва да се прави с необходимото внимание и освен в случите на тестови задачи или с цел обучение трябва по възможност да се избягва. Нека все пак да разгледаме вградение генератори на случайни числа в програмните езици *Fortran 90* и *C*.

13.2.1 *Fortran 90*

Процедурата `random_number(r)` връща едно или масив от случайни числа с равномерно разпределение в интервала $0 \leq \texttt{r} < 1$. Можем да демонстрираме работата с този генератор с една примерна програма която инициализира двумерен масив `r(5,5)` със случайни числа

```
1  program random
2    implicit none
3    real :: r(5,5)
4
5    call random_number(r)
6  end program random
```

инициализирането на засявката на генератора на случайни числа `random_number(r)` се извършва с помощта на процедурата `random_seed([SIZE, PUT, GET])`. Процедурата може да се използва и за да се получи размера и стойността на текущата засявка или да се зададе нейната стойност. Примерният код по-долу инцилизира генератора на случайни числа със стойност от операционната система

```
1  program random
2    implicit none
3    real :: r(5,5)
4    integer, allocatable, dimension(:) :: seed
5    integer :: i, seed_size, counter
6
7    call random_seed(size=seed_size)
8    allocate(seed(seed_size))
9    call system_clock(counter)
10   seed = counter + 37*[(i, i=0, seed_size - 1)]
11   call random_seed(put=seed)
12
13   call random_number(r)
14   deallocate(seed)
15 end program random
```

13.2.2 *C*

Програмният език *C* също разбира се има своя вградена функция за генериране на случайни числа която е част от стандартната библиотека. Функцията `int rand(void)`

връща цяло число в диапазона $[0, \texttt{RAND\_MAX}]$. Тогава за да генерираме реално число `r` в интервала $[0, 1]$ можем да нормираме случайните числа на `RAND_MAX`

```
#include <stdlib.h>
int main() {
  float r;

  r = (float)rand()/RAND_MAX;

  return 0;
}
```

засявката на генератора на случайни числа се извършва с помощта на процедурата `void srand (unsigned int seed)`, която приеме като аргумент цяло число без знак като засявка на поредицата. Един начин за "произволно" инициализиране на генератора на случайни числа е да се използва текущия момент време в подходящ формат. Например може да се използва системната функция `time_t time(time_t *seconds)`, която връща текущият момент време спрямо датата 1 Януари 1970 г. 00:00:00 UTC в секунди

```
#include <stdlib.h>
int main() {
  float r;

  srand(time(NULL));
  r = (float)rand()/RAND_MAX;

  return 0;
}
```

13.3 Генерериране на случайни разпределения с произволна плътност

Имайки на разположение "добър" генератор на случайни числа в интервала $[0, 1]$ с равномерно разпределение. Можем да разгледаме няколко техники за генериране на случайно разпределение с произволна плътност. За целта нека предположим че познаваме функцията за дадено вероятностно разпределение $w = w(x)$ дефинирано в произволен интервал точки $[a, b]$. Очакваме също то да бъде нормирано така че:

$$\int_a^b w(x)dx = 1 \tag{13.7}$$

В такъв случай можем да конструираме кумулутивната вероятността функцията $p(x)$:

$$p(x) = \int_a^x w(z)dz \tag{13.8}$$

Доколко функцията $w(x)$ е изцяло положителна в дефиниционният интервал, следва че функцията $p(x)$ ще бъде монотонно разстяща от $p(a) = 0$ до $p(b) = 1$. С което е

изпълнено необходимото условие за съществуването на обратна функция на $p(x)$. В случите когато можем да намерим такава:

$$x = x(p) \tag{13.9}$$

Твърдим че избирайки случайни числа p с равномерна вероятност в интервала $p \in [0, 1]$, числата $x = x(p)$ ще бъдат разпределени с плътност на вероятност $w(x)$ в интервала $[a, b]$. Това може да се покаже лесно като вземем първата производна на $p(x)$, тогава:

$$\frac{dp}{dx} = w(x) \tag{13.10}$$

т.е. $1dp = w(x)dx$. Изразът от дясната страна $w(x)dx$ ни дава вероятността да "изтеглим" случайно число в интервала $[x, x + dx]$ с плътност на вероятност $w(x)$. Докато изразът от лявата страна на уравнението ни дава вероятността да изтеглим случайно число в интервала $[p, p + dp]$ с плътност на веоятност $w(p) \equiv 1$. Следователно ако съществува решение на израза 13.9 за функцията $p(x)$, можем да генерираме редица от равномерно разпределни случайни числа $p_1, p_2, p_3, \ldots, p_i, \ldots$ в интервала $[0, 1]$. Тогава точките $x(p_1), x(p_2), x(p_3), \ldots, x(p_i), \ldots$ ще бъдат разпределени с плътност на вероятност $w(x)$ в интервала $[a, b]$.

Например, нека разгледаме случая на равномерно разпредление $w(x)$ за произволен интервал $[a, b]$ различен от първоначалният $[0, 1]$. Тогава можем да използваме горната процедура за да получим аналитичен израз за генериране извадка от равномерно разпределени случайни числа в интервала $[a, b]$. Нека разгледаме функцията $w(x)$:

$$w(x) = \frac{1}{b - a}$$

така че

$$\int_a^b w(x)dx = 1$$

Кумулутивната вероятностна функция $p(x)$ можем да намерим използвайки уравение 13.8:

$$p(x) = \int_a^x \frac{1}{b - a} dz = \frac{x - a}{b - a} \tag{13.11}$$

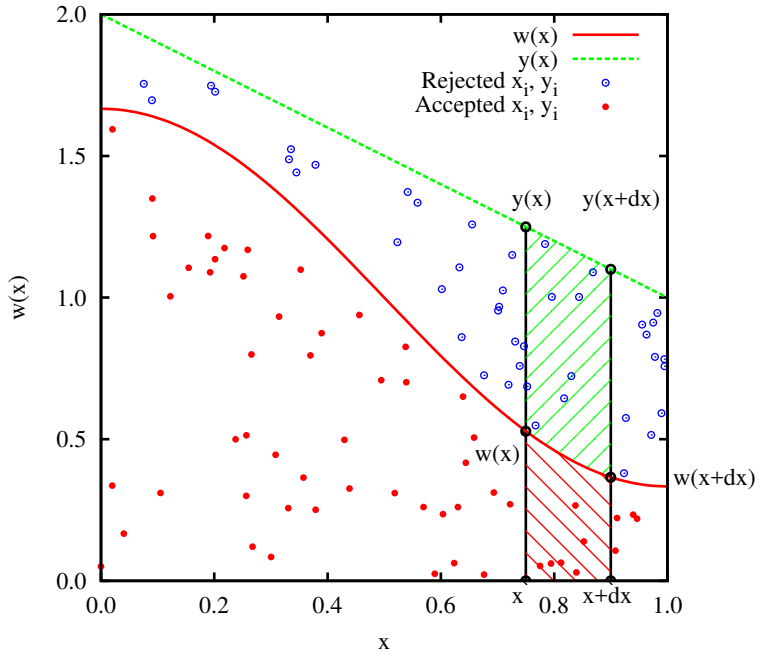
В случаят обратната функция може да се намери лесно като:

$$x = (b - a)p - a \tag{13.12}$$

Когато намирането на обратната функция $x = x(p)$ не е възможно. Може да се приложи методът на фон Нойман - *Извадка с отхвърляне.*

13.3.1 Извадка с отхвърляне. Метод на фон Нойман.

Нека разгледаме отново функцията $w(x)$ дефинирана в интервала $[a, b]$, така че е изпълнено условието за нормировка 13.7. Доколкото $w(x)$ е положително дефинирана, графиката на функцията се намира изцяло над абцисната ос. Освен това нека разгледаме една помощна функция $y(x)$ отново дефинирана в интервала $[a, b]$ с условието $w(x) < Sy(x)$ за всички $x \in [a, b]$. Където е число $S > 1$. Можем да генерираме множество точки $\{x_i, y_i\}$ които са разпределени с плътност на вероятност $y(x)$ по оста x в интервала $[a, b]$ и равновероятно разпеделени в интервала $[0, Sy(x_i)]$ за всяко x_i. По-нататък за да получим желаната от нас извадка е необходимо да изберем само тези точки $\{x_i, y_i\}$ за които е вярно условието $y_i \leq w(x_i)$, т.е. приемаме само тези точки които попадат в геометричната област ограничена от абцисната ос и графиката функцията $w(x)$.



Фигура 13.1: Примерна извадка получена с помощта на метода на фон Нойман. От генерираното множество равномерно разпределени точки $\{x_i, y_i\}$ в областта ограничена между абцисната ос и функцията $y(x)$ се подбират само тези точки за които $y_i < w(x_i)$.

За да покажем че така генерираното множество от приети стойности x_i има плътност на разпределение върху абцисната ос пропорционална на $w(x)$, нека разгледаме един инфитизимален интервал $[x, x+dx] \subset [a, b]$. Плътността на вероятност на всички точки "избрани" и отхвърлени е равна на $y(x)dx$. От друга страна вероятността да "приемем" точка от същият интервал е пропорционална на отношението $p = S_w/S_y$ на площта на трапеците S_w ограничен между точките $(x, 0)$, $(x+dx, 0)$, $(x+dx, w(x+dx))$

и $(x, w(x))$ и S_y ограничен между точките $(x, 0)$, $(x+dx, 0)$, $(x+dx, y(x+dx))$ и $(x, y(x))$. Можем да приемем че $S_w \approx (w(x) + w(x + dx))\, dx/2$ и $S_y \approx S\,(y(x) + y(x + dx))\, dx/2$. Окончателната плътност на вероятност ще бъде пропорционална на произведението на двете вероятности:

$$w \sim y(x)dx \times p = y(x)\frac{w(x) + w(x + dx)}{S\,(y(x) + y(x + dx))}dx \approx \frac{w(x)}{S}dx \tag{13.13}$$

Когато оставим $dx \to 0$ и $N \to \infty$ както и нормираме нашата извадка само за "приетите" точки. Тази вероятност клони точно към произведението $w \to w(x)dx$ когато $dx \to 0$. По-долу в секция 13.4.2 се разглежда примерната задача за пресмятането площта на кръг с помощта на Монте-Карло интегриране където се генерира плътност на вероятност с помощта на метода на фон Нойман.

13.3.2 Генериране на случайна извадка с нормално разпределение

Поради широко разпространеното използване на нормално разпределение в природо научните дисциплини си заслужава да обърнем внимание и на няколко метода за генериране на Гаусово разпределение. Разбира се представените по-долу алгоритми не изчерпват всички съществуващи такива. Стандартното разпределение се дефинира с помощта на следната функция на разпределение на плътността:

$$w(\sigma, \mu; x) = \frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{13.14}$$

където ширината σ на разпредлението и неговата средна стойност μ се разглеждат като параметри. Разпределението е нормирано така че:

$$\int_{-\infty}^{\infty} w(\sigma, \mu; x)dx = 1$$

Повечето числени алгоритми генерират нормално разпределение със средна стойност $\mu = 0$ и дисперсия $\sigma^2 = 1$. В случай че има нужда от разпределение с други стойности на μ и σ се извършва така наречението "рескалиране" което е директно следствие от теорията на вероятностите. Например ако p е случайна величина генерирана с плътност на вероятност $w(1, 0; x)$, тогава величината p':

$$p' = p\sigma + \mu \tag{13.15}$$

ще има плътност на вероятност $w(\sigma, \mu; x)$.

13.3.2.1 Нормално разпределение като сума на равновероятни разпределения.

Можем да се възпозлваме от централната гранична теорема в теорията на вероятностите според която ако разгледаме множество от N-брой независими случайни величи $X = \{X_1, X_2, \ldots, X_N\}$ с едно и също разпределение със средна стойност μ и дисперсия $0 < \sigma^2 < \infty$, тогава за случайната величина:

$$Y_N = \frac{X_1 + X_2 + \ldots + X_N}{N} \tag{13.16}$$

В случаите когато $N \to \infty$ е вярно че разликата между Y_N и средната стойност μ умножена с фактора $\sqrt{N}$ клони към нормално разпределение с $w(\sigma, \mu; x)$, т.е. величината:

$$p_N = \sqrt{N}\,(Y_N - \mu) \tag{13.17}$$

ще има плътност на вероятност $w(\sigma, \mu; x)$ при $N \to \infty$. В частен случай ако използваме съществуващ вече генератор на случайни числа с равномерно разпределение който има средна стойност $\mu = 1/2$ и дисперсия $\sigma^2 = 1$ можем да реализираме генератор на случайни числа с Гаусово разпределение. Примерният код по-долу демонстрира функция `normdist` която използва този подход:

```
1  function normdist(n)
2  double precision :: normdist
3  integer, intent(in) :: n
4
5  integer :: i
6  double precision, allocatable, dimension(:) :: x
7
8  allocate(x(n))
9
10 normdist = 0.d0
11
12 call random_number(x)
13
14 normdist = sum(x) / n - 0.5d0
15 normdist = sqrt(dble(n)) * normdist
16
17 deallocate(x)
18
19 end function NormDist
```

13.3.2.2 Алгоритъм на Оде и Евънс [6]

За дадена стойност на p в интервала $0 < p < 1$, променливата x ще притежава "нормална" (Гаусово) плътност на разпределение ако задоволява условието

$$p = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} \exp\left(-t^2/2\right) dt \tag{13.18}$$

като стойността на x може да се апроксимира като сума на два члена

$$\begin{aligned} x &= y + S(y)/T(y) && (13.19)\\ y &= \sqrt{\log\left(1/p^2\right)} && (13.20)\\ S(y) &= s_0 + s_1 y + s_2 y^2 + s_3 y^3 + s_4 y^4 && (13.21)\\ T(y) &= t_0 + t_1 y + t_2 y^2 + t_3 y^3 + t_4 y^4 && (13.22) \end{aligned}$$

където $S(y)$ и $T(y)$ са полиноми от четвърта степен. Алгоритъма гарантира грешка $x - y \leq 10^{-7}$. Коефициентите $\{s_i\}$ и $\{t_i\}$ могат да бъдат намерени в таблица 13.1

Таблица 13.1: Коефициенти на параметризация на полиномите в алгоритъма на Оде и Евънс

	0	1	2	3	4
s	0.322232431088	1.0	0.342242088547	$0.204231210245 \times 10^{-1}$	$0.453642210148 \times 10^{-4}$
t	0.099348462606	0.588581570495	0.531103462366	0.103537752850	$0.385607006340 \times 10^{-2}$

И накрая нека разгледаме примерен код на функцията `random(id, a, b)` която връща случайно число в интервала $[a, b]$ разпределено равновероятно в случай на `id = 0` или "нормално" ако `id = 1`

```
1  real(8) function random (id,a,b)
2
3  ! constants
4    real(8) :: p0,p1,p2,p3,p4
5    real(8) :: q0,q1,q2,q3,q4
6
7    parameter (p0 = 0.322232431088)
8    parameter (p1 = 1.0)
9    parameter (p2 = 0.342242088547)
10   parameter (p3 = 0.204231210245e-1)
11   parameter (p4 = 0.453642210148e-4)
12   parameter (q0 = 0.099348462606)
13   parameter (q1 = 0.588581570495)
14   parameter (q2 = 0.531103462366)
15   parameter (q3 = 0.103537752850)
16   parameter (q4 = 0.385607006340e-2)
17
18
19 ! input variables
20   integer, intent(in) :: id
21   real(8), intent(in) :: a, b
22 ! local variables
23   real(8) :: x,p,q,t
24
25 ! select random distribution by id:
26 ! 1 - normal distribution
27 ! a -> mean value
28 ! b -> sigma ( b > 0 )
```

```
29  ! p(x) = exp(-(x-a)^2/(2*b^2))
30  ! Uses a very accurate approximation of the normal idf due to Odeh &
        Evans,
31  ! J. Applied Statistics, 1974, vol 23, pp 96-97.
32  !
33  ! 0 - uniform deistribution (default)
34  ! a -> xmin
35  ! b -> xmax
36
37    selectcase(id)
38      case(0)
39        call random_number(x)
40        random = (xmax - xmin)*x + xmin
41      case(1)
42        call random_number(x)
43        if (x < 0.5) then
44          t = sqrt(-2.0 * log(x));
45        else
46          t = sqrt(-2.0 * log(1.0 - x));
47        end if
48
49        p = p0 + t * (p1 + t * (p2 + t * (p3 + t * p4)));
50        q = q0 + t * (q1 + t * (q2 + t * (q3 + t * q4)));
51
52        if (x < 0.5) then
53          z = (p / q) - t;
54        else
55          z = t - (p / q);
56        end if
57
58        random = (a + b * z);
59
60    end select
61
62  end
```

13.4 Монте Карло интегриране

13.4.1 Основни положения

Едно от основните преимущестества ма Монте Карло методите е възможността за пресмятането на интеграли и по-специално многомерните интеграли. За простота на записа обаче ще демонстрираме основните принципи на Монте Карло интегрирането за едномерна функция $f(x)$. Нека разгледаме определеният интеграл

$$I = \int_a^b f(x)dx = (b-a)\bar{f} \tag{13.23}$$

на функцията $f(x)$ дефинирана в интервала $x \in [a, b]$. Интеграла I може да се пресметне като средната стойност на функцията $f(x)$, т.е. може да се представи като сума

от N на брой функционални стойности $f(x_i)$, където точките x_i са подбрани равновероятно в интервала $[a, b]$

$$I \approx \frac{b-a}{N} \sum_{i=1}^{N} f(x_i) \tag{13.24}$$

доколко интеграла I в случая се представя като средно аритметична сума на функционалните стойности на $f(x)$ можем да запишем и оценка за горната граница на грешката от пресмятането на интерграла I която е пропорционална на дисперсията на

$$\sigma_I^2 \approx \frac{1}{N}\sigma_f^2 = \frac{1}{N}\left[\frac{1}{N}\sum_{i=1}^{N} f_i^2 - \left[\frac{1}{N}\sum_{i=1}^{N} f_i\right]^2\right] \tag{13.25}$$

където σ_f^2 е дисперсията на функцията $f(x)$. Горната формула е много важна защото тя дава оценка за грешката σ_I от Монте карло интегиране която е обратно пропорционална на $\sqrt{N}$ независимо от размеността на интеграла. Другият важен извод който можем да направим е че доколко грешката σ_I е пропорционална на σ_f за да е намалим трябва да намерим начин да изчислим средната стойност $\bar{f}$ с по-ниска стойност на дисперсията σ_f. За целта може да се изпозлва един модифициран метод за интегриране предложен от фон Нойман. Нека разгледаме една помощна функция $w(x)$ която има смисъл на вероятностна функция

$$\int_a^b w(x)dx = 1 \tag{13.26}$$

тогава интеграла I може да се запише като

$$I = \int_a^b w(x)\frac{f(x)}{w(x)}dx \tag{13.27}$$

ако изберем подходящта смяна на променливите

$$y(x) = \int_a^z w(z)dz \tag{13.28}$$

така че $\dfrac{dy}{dx} = w(x)$ и $y(x = a) = 0$ и $y(x = b) = 1$, тогава можем да запишем интеграла I като

$$I = \int_0^1 \frac{f(x(y))}{w(x(y))}dy \tag{13.29}$$

сега интеграла I може да се представи като следната сума

$$I \approx \frac{1}{N}\sum_{i=1}^{N} \frac{f(x(y_i))}{w(x(y_i))} \tag{13.30}$$

където точките y_i са избрани равновероятно в интервала $[0,1]$. Ако подберем функцията $w(x)$ в по такъв начин че отношението $f(x)/w(x)$ се запазва приблизително постоянно в целия дефиниционен интервал $[a,b]$, можем да очакваме че $\sigma_{f/w} < \sigma_f$ което води до намаляване на общата грешка σ_I.

Задача: Напишете програма която пресмята стойността на интеграла

$$\int_0^1 \frac{dx}{1+x^2}dx = \frac{\pi}{4} \tag{13.31}$$

използвайки модифицираната версия на алгоритъма на Монте Карло с теглова функция

$$w(x) = \frac{4-2x}{3} \tag{13.32}$$

13.4.2 Примерна задача намиране площта на кръг. Метод на фон Нойман.

Метода за генериране на случайно разпределение $w(x)$ чрез извадка с отхвърляне описан в глава 13.3 може да бъде демонстрирана чрез задачата за намиране площта на фигура описана от окръжност. Нека разгледаме окръжност с радиус $R = 1$ с център в координатната система. За удобство ще намерим площта на фигурата която се намира само в I-ви квадрант описвана с монотонно намаляващата функция $w(x) = \sqrt[2]{R^2 - x^2}$. Площта на кръга с радиус R е равен на:

$$S = 4I = 4\int_0^R w(x)dx = 4\int_0^R \sqrt[2]{R^2 - x^2}dx \equiv \pi R^2 \tag{13.33}$$

ако можем да генерираме множество от точки с плътност пропорционална на $w(x)$ тогава интеграла I в уравнение 13.33 може да се пресметне с помощта на израза:

$$I \approx \frac{1}{N_{tot}} \sum_i^{N_{acc}} 1 = \frac{N_{acc}}{N_{tot}} \tag{13.34}$$

където N_{tot} е пълният брой генерирани точки в областта. Докато N_{acc} е броят точки които са "приети". За ограничаваща функция можем да изберем постоянна зависимост $y(x) \equiv 1$ и коефициент $S = 1$. Тогава можем да генерираме множество от N_{total}-брой точки $\{x_i, y_i\}$ разпределени с равномерна вероятност в интервала $x_i \in [0,1]$ и $y_i \in [0,1]$. За всяка една точка (x_i, y_i) от множестовото $\{x_i, y_i\}$ се прави проверка дали $y_i \leq w(x)$ и в случай че това е вярно точката се "приема" и акумулира в N_{acc}. По-долу е представен код на програмният език *Fortran 90* който реализира алогиритъма за $R = 1$ в който случай резултатът от пресмятането на интеграла $S = \pi$:

```
1 program mc
2 implicit none
3
```

```
4  integer(kind = 8) :: Np ! Number of generated points
5  integer(kind = 8) :: Nacc! Number of accepted points
6  integer(kind = 8) :: i ! Work variable for do loops
7  double precision :: pi ! Accumnulated result
8  double precision :: x ! x - coordinate [0, 1]
9  double precision :: y ! y - coordinate [0, 1]
10 double precision :: r2 ! Scuare distance of (x, y) point
11
12 write(0,'(A)', advance='NO')'Please␣enter␣the␣number␣of␣MC␣steps␣Np␣=␣'
13 read(*,*)Np
14
15 Nacc = 0
16 if (Np > 1) then
17   call random_seed()
18   do i = 1, Np
19     call random_number(x)
20     call random_number(y)
21     r2 = x**2 + y**2
22     if ( r2 .le. 1.d0) then
23       Nacc = Nacc + 1
24     end if
25   end do
26
27   pi = 4.d0 * dble(Nacc) / Np
28
29   write(0,'(A20,F15.7)') 'Estimate␣value␣Pi␣=␣',pi
30 else
31   write(0,*)'Bad␣Np␣value␣!!!'
32   stop
33 end if
34
35 end program mc
```

13.5 Метрополис Монте Карло

Метрополис Монте Карло алгоритъма е числен метод за генериране вероятностно разпределение $w(\mathbf{X})$ в много-мерно конфигурационно пространство $\mathbf{X} = \{X_1, X_2, X_4, \ldots\}$. Стартирайки от произволна начална точка $\mathbf{X}^0$ методът итеративно генерира серия от точки в конфигурационното пространство $\mathbf{X}^0, \mathbf{X}^1, \mathbf{X}^2, \ldots, \mathbf{X}^M, \ldots$ чиято плътност на разпределение клони към $w(\mathbf{X})$ при $M \to \infty$, процеса може да се разглежда като случайно блуждаене в конфигурационното пространство. В този смисъл серията от точки в конфигурационното пространство се нарича "пешеходец". Генерираната серия от точки се разглежда като Марковски процес, т.е. вероятността за избиране на ново състояние $\mathbf{X}^{M+1}$ зависи изцяло само от текущото състояние $\mathbf{X}^M$. Алгоритъма на Метрополис е изключително удобен при моделирането на статистически ансамбли както ще демострираме по долу в примерната задача в секция 13.7 (Модел на Изинг в двумерието).

Нека разгледаме един от възможните алгоритми които реализират Метрополис

Монте Карло. Ще предположим че искаме да гериране вероятностно разпределение $w(\mathbf{X})$ дефинирано в многомерно конфигурационно пространство $\mathbf{X}$. Стартирайки от произволна началана точка $\mathbf{X}^0$ ние генерираме серия от точки $\mathbf{X}^M$ чиято плътност на разпределение $N_M(\mathbf{X})$ трябва да клони към $w(\mathbf{X})$ при $M \to \infty$. Нека за произволно M разгледаме точката $\mathbf{X}_M$ ще дефинираме следното правило за намиране на следващата точка $\mathbf{X}^{M+1}$ от разпределението: Избираме по произволен начин "пробна" точка $\mathbf{X}_r$ от конфигурационното пространство, например $\mathbf{X}^t = \mathbf{X}^M + \mathbf{R}$, където $\mathbf{R}$ може да бъде случен радиус вектор. Тогава точката $\mathbf{X}^t$ се "приема" или "отхвърля" с според стойността на отношението

$$r = \frac{w(\mathbf{X}^t)}{w(\mathbf{X}^M)} \tag{13.35}$$

Ако r в по-голямо от единица тогава новата стъпка се примена и ние полагаме $\mathbf{X}^{M+1} = \mathbf{X}^t$, в случаите когато $r < 1$ пробната конфигурация $\mathbf{X}^t$ се приема с вероятност r. Това лесно може да се направи ако изберем равно вероятно случайно число $\nu \in [0, 1]$ и приемем $\mathbf{X}^t$ в случаите когато $\nu < r$. В случаите когато пробната стъпка $\mathbf{X}^t$ не се приема, т.е. се отхвърля за следваща точка $\mathbf{X}^{M+1}$ се избира $\mathbf{X}^{M+1} = \mathbf{X}^M$ отново.

Доказателството че описаният алгоритъм наистина конструира множество от точки в конфигурационното пространство $\mathbf{X}$ с плътност на разпредление $w(\mathbf{X})$, нека предположим че имаме голям брой "блуждаещи" случайно пешеходци които стартират от различни точки в конфигурационното пространство $\mathbf{X}$. Тогава можем да разгледаме плътността на разпредление на пешеходците $N_M(\mathbf{X})$ за дадена стъпка M от алгоритъма и точка $\mathbf{X}$ от конфигурационното пространство. Ако изберем две произволни точки $\mathbf{X}$ и $\mathbf{Y}$ от конфигурационното пространство, тогава броят на пешеходците които ще се преместят от $\mathbf{X}$ към $\mathbf{Y}$ на следващата стъпка може да се запише като

$$\begin{aligned} \Delta N(\mathbf{X}) &= N_M(\mathbf{X})P(\mathbf{X} \to \mathbf{Y}) - N_M(\mathbf{Y})P(\mathbf{Y} \to \mathbf{X}) \\ &= N_M(\mathbf{Y})P(\mathbf{X} \to \mathbf{Y}) \left[\frac{N_M(\mathbf{X})}{N_M(\mathbf{Y})} - \frac{P(\mathbf{Y} \to \mathbf{X})}{P(\mathbf{X} \to \mathbf{Y})} \right] \end{aligned}$$

където $P(\mathbf{X} \to \mathbf{Y})$ е вероятността за преход от точка $\mathbf{X}$ към точка $\mathbf{Y}$ в конфигурационното пространство. Според горното уравнение състояние на равновесие, т.е. $\Delta N(\mathbf{X}) = 0$ ще бъде достигнато като за произволни две точки $\mathbf{X}$ и $\mathbf{Y}$ се изпълнява условието

$$\frac{N_M(\mathbf{X})}{N_M(\mathbf{Y})} = \frac{N_e(\mathbf{X})}{N_e(\mathbf{Y})} = \frac{P(\mathbf{Y} \to \mathbf{X})}{P(\mathbf{X} \to \mathbf{Y})} \tag{13.36}$$

където с N_e ще означаваме равновесното разпределение.

Остава да покажем че равновестната плътност N_e която се генерира чрез Метрополис алгоритъма оттговаря на търсената от нас плътност на разпределение $w(X)$, т.е. $N_e(X) \sim w(X)$. Вероятността за преход между $\mathbf{X}$ и $\mathbf{Y}$ според метода на Метрополис

може да се запише като

$$P(\mathbf{X} \to \mathbf{Y}) = T(\mathbf{X} \to \mathbf{Y})A(\mathbf{X} \to \mathbf{Y}) \tag{13.37}$$

където $T(\mathbf{X} \to \mathbf{Y})$ е верочтността за "пробна" стъпка между $\mathbf{X}$ и $\mathbf{Y}$, докато $A(\mathbf{X} \to \mathbf{Y})$ е вероятността за приемането и. Ако точката $\mathbf{Y}$ може да бъде достигната от точка $\mathbf{X}$ за една стъпка, т.е. разстоянието между тях е по-малко от $\mathbf{r}$ тогава

$$T(\mathbf{X} \to \mathbf{Y}) = T(\mathbf{Y} \to \mathbf{X})$$

следователното условеието за равновесие на алгоритъма на случайно блуждаене на Метрополис може да се запише като

$$\frac{N_e(\mathbf{X})}{N_e(\mathbf{Y})} = \frac{A(\mathbf{Y} \to \mathbf{X})}{A(\mathbf{X} \to \mathbf{Y})} \tag{13.38}$$

ако $w(\mathbf{X}) > w(\mathbf{Y})$ тогава $A(\mathbf{Y} \to \mathbf{X}) = 1$ и

$$A(\mathbf{X} \to \mathbf{Y}) = \frac{w(\mathbf{Y})}{w(\mathbf{X})}$$

докато ако $w(\mathbf{Y}) > w(\mathbf{X})$ тогава

$$A(\mathbf{Y} \to \mathbf{X}) = \frac{w(\mathbf{X})}{w(\mathbf{Y})}$$

Следователно и в двата случая уравнението за равновесие на разпределението изпълнява едно и също условие

$$\frac{N_e(\mathbf{X})}{N_e(\mathbf{Y})} = \frac{w(\mathbf{X})}{w(\mathbf{Y})} \tag{13.39}$$

13.6 Примерна задача: Радиационен транспорт - движение на електрони в газова среда

Радиационният транспорт е процес на преминаване на йонизиращо лъчение през среда. Практическият интерес към този процес например възниква при изучаването на ефекта от радиацията върху живите организми в медицината, детекторите на йонизиращо лъчение, ядрената енергетика и др. Моделирането на самият процес на радиационен транспорт включва множество физични процеси на субатомно ниво като еластично и не еластично разсейване, вътрешен фотоефект, ефект на Комптън, спирачно лъчение и т.н. Поради сложноста и различните времеви мащаби на процесите аналитичните решения на задачата за радиационен транспорт са със сравнително ограничено приложение и няма да бъдат разглеждани в този раздел. За нас представлява интерес численото моделиране на процеса на радиациоенен транспорт, който

ще онагледим с една примерна задача, която може да послужи като основа за понататъшното задълбочено разглеждане на теорията. В общият случай процесът на радиационен транспорт е случаен или стохастичен. Моделът създава серия от събития (физически процеси) които се реализират с определена вероятност. В нашата примерна задача ще разглеждаме движението на електрон под действието на външно електрично поле в газова среда, който изпитва серии от случайни еластични сблъсъци (или разсейвания) с молекулите на идеалният газ. Моделът позволява да се оцени средното време на пробег T_D необходимо за преминаване на електрона през газовата среда с дебелина $D = 0.2\,m$ в посока z на координатната система. Самите електрони се движат под действието електостатична сила в следствие на приложено външно електрично $\mathbf{E}$ поле с интензитет $E = 500\,V/m$, което е ориентирано също в посока z на координатната система. Моделът използва един важен параметър наречен честота на сблъсъците $C_f = 1\,GHz$, който се използва в стохастичната динамика при която движението се разглежда като праволинейно с ускорение $\mathbf{a} = e\mathbf{E}/m_e$ между два последователни еластични сблъсъка. След всеки еластичен удар посоката и големината на скоростта се променят. Ако с $\mathbf{v}_i$ обозначим скоростта на движение на електрона преди еластичният сблъсък а с $\mathbf{v}_f$ скоростта му след сблъскъка. Използвайки закона за запазване на импулса и енергия в случай на еластичен удар

$$m_e\mathbf{v}_i \;\; = \;\; m_e\mathbf{v}_f \tag{13.40}$$

може да се покаже че съществува следната зависимост между абсолютната големина на векторите $\mathbf{v}_i$ и $\mathbf{v}_f$.

$$|\mathbf{v}_f| = |\mathbf{v}_i|\sqrt{1 - \frac{2m_e}{m_n}(1 - \cos\theta)} \tag{13.41}$$

където m_e и m_n са съответно масите на електрона и атомната маса на химичният елемент с който електрона изпитва еластичен удар, θ е ъгъл на отклонение между скоростите $\mathbf{v}_i$ и $\mathbf{v}_f$. Новата посока на разпространение $\mathbf{v}_f$ може да се укаже относително спрямо старата посока $\mathbf{v}_i$ с помощта на Ойлеровите ъгли θ и ϕ фиг. **??**. Промяната на координатите на скоростта може да представи а помощтта на матрицата на трансформация $\mathbf{A}$ която се определя чрез Ойлеровите ъгли (ϕ, θ)

$$\begin{aligned}\mathbf{A} = \mathbf{A}_x \cdot \mathbf{A}_y = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix} \cdot \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} = \\ \begin{bmatrix} \cos\theta & \sin\phi\sin\theta & \cos\phi\sin\theta \\ 0 & \cos\phi & -\sin\phi \\ -\sin\theta & \sin\phi\cos\theta & \cos\phi\cos\theta \end{bmatrix}\end{aligned} \tag{13.42}$$

ако посоката на скоростта преди сблъсъка $\mathbf{v}_i$ съвпада с оста z тогава трансформацията на вектора на скоростта $\mathbf{v}_f$ може да се запише чрез следният израз

$$\mathbf{v}_f = \mathbf{A} \cdot \mathbf{v}_i \sqrt{1 - \frac{2m_e}{m_n}(1 - \cos\theta)} \tag{13.43}$$

за да приложим горната трансформация обаче е необходимо да извършим допълнителна трансформация с ъгли на Ойлер (β, α), така че да ориентираме оста z на координатната система в посока на вектора $\mathbf{v}_i$. Самите ъгли β и α могат да се определят чрез

$$\beta = \arctan\left(v_i^y / v_i^x\right) \tag{13.44}$$
$$\alpha = \arccos\left(v_i^z / |\mathbf{v}_i|\right) \tag{13.45}$$

матрицата на въртене ще се запише тогава като

$$\mathbf{B} = \begin{bmatrix} \cos\alpha & \sin\beta\sin\alpha & \cos\beta\sin\alpha \\ 0 & \cos\beta & -\sin\beta \\ -\sin\alpha & \sin\beta\cos\alpha & \cos\beta\cos\alpha \end{bmatrix}$$

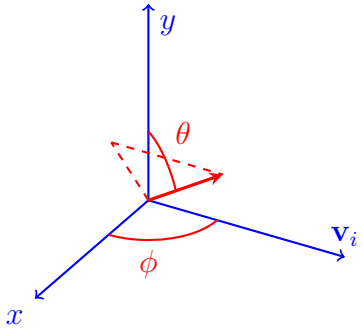
окончателно трансоформацията на векторите на скоростта преди сблъсъка $\mathbf{v}_i$ и след него $_f$ се записват със следното уравнение

$$\mathbf{v}_f = \mathbf{B}^{-1} \cdot \mathbf{A} \cdot \mathbf{B} \cdot \mathbf{v}_i \sqrt{1 - \frac{2m_e}{m_n}(1 - \cos\theta)} \tag{13.46}$$

горното уравнение 13.46 описва следните 5 стъпки

- Определят се Ойлеровите ъгли на завъртане (β, α) чрез вектора $\mathbf{v}_i$
- Извършва се трансформация на координатната система така че оста z да сочи в посока на вектора $\mathbf{v}_i$
- Извършва се допълнително завъртане на координатната система на случайно избрани Ойлерови ъгли (θ, ϕ) така че остта z да сочи в посока на скоростта $\mathbf{v}_f$
- Променя се големината на вектора на скоростта така че да се отчита промяна на енергията след еластичият удар: $\mathbf{v}_f^2 = \mathbf{v}_i^2 - 2\Delta\epsilon/m_e$
- Извършва се обратна трансформация на координатната система с ъгли на Ойлер $(-\alpha, -\beta)$ към първоначалната ориентация на координатната система

Ойлеровите ъгли на завъртане (θ, ϕ) можем да определим от условието че избора на новата посока на разпространение трябва да бъде равновероятен, т.е. сечението за разсейване $\sigma_0(\theta, \alpha)$ е постоянна величина и не зависи от ъглите (θ, ϕ). Следователно



Фигура 13.2: Схема на скоростите на разпространение преди еластичен удар $\mathbf{v}_i$ и след него $\mathbf{v}_f$.

$$\sigma(\theta,\phi) = \sigma_0 \tag{13.47}$$
$$dp(\theta,\phi) = \sigma(\theta,\phi)d\Omega = \sigma_0 \sin\theta d\theta d\phi \tag{13.48}$$
$$\int_{4\pi} dp \equiv 1 \tag{13.49}$$

където $\sigma(\theta,\phi)$ е сечението за разсейване на елекрона в посока определена с пространствени ъгли (θ,ϕ), докато $dp(\theta,\phi)$ е вероятността за разсейване на електрона в пространствен ъгъл $d\Omega$, последното условие $\int dp \equiv 1$ е условието за нормировка. Взимайки предвид условието за нормировка в уравнение 13.49 можем да запишем че:

$$1 = 2\pi\sigma_0 \int_0^\pi \sin\theta d\theta = 2\pi\sigma_0$$

или $\sigma_0 = 1/2\pi$. Моделът приема като входни параметри: дължината на слоят вещество - D, интензитета на електричното поле - $\mathbf{E} = (0,0,E_z)$ и честотата на сблъсъци C_f, като пресмята усредненото време за пробег T_D за N брой електрона. Уравненията на движение на които се почдинява електрона могат да се запишат във векторна форма, приемайки $\mathbf{r} = (x,y,z)$ за радиус вектор с координатите на електрона, тогава:

$$\frac{d\mathbf{r}}{dt} = \mathbf{v}$$
$$\frac{d\mathbf{v}}{dt} = \mathbf{a} = e\mathbf{E}/m_e$$

електроните започват движението си нулева начална скорост $\mathbf{v} = (0,0,0)$ и се ускоряват непрекъснато в посока на приложеното външно електрично поле $\mathbf{E}$. В дискретна форма горната система обикновенни диференциални уравнения приема следният вид:

$$\mathbf{r}(t+dt) = \mathbf{r}(t) + \mathbf{v}dt + \frac{1}{2}\mathbf{a}dt^2 \tag{13.50}$$

$$\mathbf{v}(t+dt) = \mathbf{v}(t) + \mathbf{a}dt \tag{13.51}$$

за разлика то класическите интеграционни схеми тук интервалът време dt между две последователни стъпки не е еквидостантен, а се пресмята по случен начин чрез честотата на сблъсък C_f, или:

$$p(dt) = 1 - \exp^{-C_f dt} \tag{13.52}$$

$$dt(p) = -\frac{\log(1-p)}{C_f} \tag{13.53}$$

така, интервалът време dt е функция на случайното число p равновероятно разпределено в интервала $[0, 1)$. Следващата задача е да се изберат ъгли на разсейване θ и ϕ на пространственият ъгъл $\Omega = \sin\theta d\theta d\phi$ с равна вероятност. Доколко ϕ е пространствен ъгъл в интервала $[0, 2\pi]$ можем да запишен че:

$$\phi = 2\pi q \tag{13.54}$$

където q случайно число с равновероятна извадка е интервала $q \in [0, 1)$. Другият множител $\sin\theta$ в пространственият ъгъл обаче трябва да бъде избран в интервала $\theta \in [0, \pi]$ с плътност на вероятност $dp(\theta) = \sin\theta d\theta$, използвай условието $\int_0^p i dp(\theta) = 1$, получаваме че $p(\theta) = (1 + \cos\theta)/2$, еледователно:

$$\theta = \arccos(2p - 1) \tag{13.55}$$

където p е случайно разпределено число с равнавероятност в интервала $p \in [0, 1)$. Веднъж избрали ъглите на еластично разсейване θ и ϕ, можем да получим компонентите на скоростта след еластичният удар $\mathbf{v}_f = (v_f^x, v_f^y, v_f^z)$

$$v_f^x = \tag{13.56}$$

$$v_f^y = \tag{13.57}$$

$$v_f^z = \tag{13.58}$$

интегрирането на уравненията на движение се извършва докато координатата z не достигне границите на областта D. Доколко процесът на преминава не стохастичен акумулираното време за преминаване на областта T_D ще бъде случайно число с определена извадка. Затова е необходимо да се извършат достатъчно голям брой "проигравяния" на процеса и получените времена на преминаване T_D да се усреднят. По-долу е представен изходният код на алгоритъма който симулира процеса на радиационен транспорт написан на програмният език *Fortran*:

```
1  program drift
2    implicit none
3  ! Electron mass [Kg]
4    double precision, parameter :: xmasse = 9.11d-31
5  ! Nuclon mass [Kg]
6    double precision, parameter :: xmassp = 1.67d-27
7  ! Electric field strength [V/m]
8    double precision, parameter :: efield = 500.
9  ! Collision frequensy [Hz]
10   double precision, parameter :: colf = 1.d9
11 ! Target width [m]
12   double precision, parameter :: xend = 0.2
13 ! Number of nuclons in the nucleus [Neon n = 20]
14   integer, parameter :: n = 20
15 ! Electric field acceleration [Kg m / s^2]
16   double precision, parameter :: acx = efield * 1.6d-19 / xmasse
17 ! Define Pi constant
18   double precision :: pi = 3.14159265d0
19
20   integer :: i, ielectrons
21   double precision :: vdrift = 0.
22   double precision :: xnorm = 0.
23   double precision :: x, v, vx, vy, vz, dt, ttotal
24   double precision :: alpha, beta, phi, theta
25   double precision :: xmassn
26   double precision :: p, q
27
28 ! Caclulate the nucleus mass
29   xmassn = n * xmassp
30
31   write(0,'(A)')'Number␣of␣electrons␣to␣iterate␣=␣'
32   read(*,*)ielectrons
33
34   do i = 1, ielectrons
35     x = 0.
36     ttotal = 0.
37     vx = 0.
38     vy = 0.
39     vz = 0.
40
41 ! Caclulate the time period to the next collision
42 50 call random_number(p)
43     dt = - (1. / colf) * log(1. - p)
44
45 ! Make one integration step further
46     x = x + vx * dt + 0.5 * acx * dt * dt
47     vx = vx + acx * dt
48 ! Accumulate total drift time
49     ttotal = ttotal + dt
50
51 ! Check wheter end of target has been reached
52 ! if not -> collision
53     if (x.lt.xend) then
54       v = sqrt(vx*vx+vy*vy+vz*vz)
```

```
55
56          beta = atan2(vy,vx)
57          alpha = acos(vz/v)
58
59    ! Generate collision angles (theta, phi)
60          call random_number(p)
61          theta = acos(2. * p - 1.)
62          call random_number(q)
63          phi = 2 * pi * q
64
65    ! Calculate new velocity after elastic collision
66          v = v * sqrt(1. - (2.*xmasse/xmassn)*(1.-cos(theta)))
67
68    ! Calcluate new direction, i.e. rotate with (theta, phi)
69          vx = v*(cos(beta)*cos(alpha)*sin(theta)*cos(phi) +&
70         & cos(beta)*sin(alpha)*cos(theta)-&
71         & sin(beta)*sin(theta)*sin(phi))
72          vy = v*(sin(beta)*cos(alpha)*sin(theta)*cos(phi)+&
73         & sin(beta)*sin(alpha)*cos(theta)+&
74         & cos(beta)*sin(theta)*sin(phi))
75                vz = v*(-sin(alpha)*sin(theta)*cos(phi)+&
76         & cos(alpha)*cos(theta))
77          goto 50
78        end if
79    ! Accumulate drift velocity for all electrons
80        vdrift = vdrift + x/ttotal
81        xnorm = xnorm + 1.
82      end do
83    ! Average drift velocity for all electrons
84      vdrift = vdrift / xnorm
85      write(0,'(A,F20.5,A)')'vdrift␣=␣',vdrift, '␣[m/s]'
86      write(0,'(A,F20.5,A)')'Ekin␣␣␣=␣',0.5*xmasse*vdrift**2*6.2415097d+18, '␣[
            eV]'
87    end
```

примерният код може да "свален", компилиран и изпълнен с помощта на следните няколко команди в терминален прозорец:

```
1  git clone https://github.com/pisov/mc.git
2  cd mc/drift/fortran
3  make
4  ./drift.x
```

Задача: Модифицирайте изходният код на примерната програма drift.f90 така че да пресмята средно "разширение" на разсейването в равнината перпендикулярна на посоката на разпространение z, т.е. пресметнете средната стойност на разтоянието $d_{xy} = \sqrt{x^2 + y^2}$ и отпечатайте резултатът заедно с v_{drift}.

13.7 Примерна задача: Модел на Изинг в двумерието

Моделът на Изинг се използва за описанието на магнитните свойства на системи изградени от еднотипни обекти (частици) притежаващи собствен магнитен момент μ и спин σ. В частен случай за двумерният вариант съществува точно решение на Онсагер [7] което описва фазов преход между феро $\leftrightarrow$ пара магнитно състояние. Моделът представлява двумерна решетка от "възли", като на позицията на всеки възел се намира частица със спин σ_i. Възможните стойностии на спина са само две, а именно $\sigma_i \in \{+1, -1\}$. Магнитният момент за всеки възел μ_i се дефинира като произведение на спина на частицата σ_i с големината на магнитният и момент μ, или $\mu_i = \mu\sigma_i$. Хамилтонианът на система се записва:

$$H(\sigma) = -\sum_{\langle i,j \rangle} J_{ij}\sigma_i\sigma_j - h\mu \sum_{\langle i \rangle} \sigma_i \tag{13.59}$$

където сумата в първият член на горният израз описва локалното взаимодействие между магнитните моменти и се извършва върху всички двойки $\langle i, j \rangle$ съседни "възли" без повторение. Вторият член на Хамилтониана описва взаимодействието на системата с външно магнитно поле h, като сумата се извършва върху всички индекси на възлите от решетката $\langle i \rangle$. Символът $\boldsymbol{\sigma}$ схематично представя едно конкретно "състояние" на всички спинове на σ_i от решетката. Съответно статистическата сума на системата можем да представим като:

$$Z(\beta) = \sum_{\boldsymbol{\sigma}} \exp^{-\beta H(\boldsymbol{\sigma})} \tag{13.60}$$

Свободната енергия на системата $F(\beta)$ се дефинира като натурален логаритъм от статистическата сума умножен с обратната температура β:

$$F(\beta) = -\beta \ln\left(Z(\beta)\right)$$

вероятността за състояние σ при зададена обратна температура $\beta = (k_B T)^{-1}$ се определя от отношението:

$$p(\boldsymbol{\sigma}, \beta) = \frac{\exp^{-\beta H(\boldsymbol{\sigma})}}{Z(\beta)} \tag{13.61}$$

Познавайки функцията $p(\boldsymbol{\sigma}, \beta)$ можем да намерим очакваната стойност за физична величина $f(\boldsymbol{\sigma})$ свързана със системата:

$$f(\beta) = \langle f \rangle_\beta = \sum_{\boldsymbol{\sigma}} f(\boldsymbol{\sigma}) p(\boldsymbol{\sigma}, \beta) \tag{13.62}$$

записа $\sum_{\boldsymbol{\sigma}}$ схематично описва сума по всички възможни дискретни изброимо много състояния $\boldsymbol{\sigma}$ на решетката. Тогава можем да запишем изразите за магнитният момент M и енергията E на система:

$$E = -\frac{\partial F(\beta)}{\partial T} = \sum_{\boldsymbol{\sigma}} H(\boldsymbol{\sigma}) p(\boldsymbol{\sigma}, \beta) \tag{13.63}$$

$$M = -\frac{\partial F}{\partial h} = \mu \sum_{\boldsymbol{\sigma}} \boldsymbol{\sigma} p(\boldsymbol{\sigma}, \beta) \tag{13.64}$$

както и свързаните с техни производни величини като магнитната възприемчивост χ и топлинният капацитет C:

$$C = \frac{\partial^2 F(\beta)}{\partial \beta^2} = \frac{\beta}{T} \sum_{\boldsymbol{\sigma}} \left(H(\boldsymbol{\sigma}) - E\right)^2 p(\boldsymbol{\sigma}, \beta) \tag{13.65}$$

$$\chi = -\frac{\partial^2 F}{\partial h^2} = \sum_{\boldsymbol{\sigma}} \left(\boldsymbol{\sigma} - M\right)^2 p(\boldsymbol{\sigma}, \beta) \tag{13.66}$$

от уравненията по-горе е видно че двете величини се определят като средноквадратичните отклонения на енергията E и магнитният момент M с точност до множител.

Численото решение на модела на Изинг за двумерна решетка чрез Метрополис Монте-Карло метод предполага че разглеждаме правоъгълна решетка от "възли" с размер $N \times N$ с наложени периодични гранични условия и в двете направления. Всеки един елемент от решетката тогава може да се идентифицира с двойка цели числа (i, j), където $i, j \in 1, 2, \ldots, N$. Ще разглеждаме случаят на отсътсващо външно магнитно поле $h = 0$. Матрицата на взаимодействие $\mathbb{J}$ включа само най-близките четири съседа, с постоянна константа на взаимодействие J:

$$J_{\langle ij,mn\rangle} = \frac{J}{2}\left(\delta_{i+1,m}\delta_{j,n} + \delta_{i-1,m}\delta_{j,n} + \delta_{i,m}\delta_{j+1,n} + \delta_{i,m}\delta_{j-1,n}\right) \tag{13.67}$$

така Хамилтонианът H и магнитният момент M на система се опростява до следните изрази:

$$H = -J \sum_{i=1}^{N} \sum_{j=1}^{N} \sigma_{ij} \left(\sigma_{i+1,j} + \sigma_{i-1,j} + \sigma_{i,j+1} + \sigma_{i,j-1}\right) \tag{13.68}$$

$$M = \mu \sum_{i=1}^{N} \sum_{j=1}^{N} \sigma_{ij} \tag{13.69}$$

За числените пресмятания ще бъде удобно ако температурата T на системата се представя в единици $[\beta J]$. Докато пълният магнитен момент в единици на елементарният магнитен момент μ. Сега можем да запишем отново вероятността за дадено състояние σ на двумерната магнитна решетка

$$p(\sigma, \mathrm{T}) = \frac{1}{Z(\beta)} \exp\left(\frac{1}{T}\sum_{i=1}^{N}\sum_{j=1}^{N}\sigma_{ij}\left(\sigma_{i+1,j} + \sigma_{i-1,j} + \sigma_{i,j+1} + \sigma_{i,j-1}\right)\right) \tag{13.70}$$

Директното премятане на сумата $\sum_{\boldsymbol{\sigma}}$ е практически невъзможно за $N > 4$ доколко пълният брой възможни състояния на решетката е $2^{N\times N}$. Затова е удачно да се приложи друг числен подход за генериране на извадка от конфигурации $\boldsymbol{\sigma}$ които имат плътност на разпределение 13.70. На помощ идва описаният в глава 13.5 алгоритъм на Метрополис. Този метод обаче изисква да дефинираме подходяща трансформация която да описва преминаване от едно състояние на решетката $\boldsymbol{\sigma}$ към следващото $\boldsymbol{\sigma}'$. Изборът на подходящата трансформация е важен за ефективността на Монте-Карло алгоритъма. Ако енергията на избраната нова конфгурация се различава твърде много от предишната вероятността тя да бъде отхвърлена е по-голяма. Затова например е удачно да се избира преход при който само един произволно избран възел с индекси (i, j) си сменя знака от σ_{ij} към $-\sigma_{ij}$. Тогава промяната в конфигурационната енергия на системата ΔE е сравнително малка и отношението на двете вероятности $p(\boldsymbol{\sigma})$ и $p(\boldsymbol{\sigma}')$ ще бъде числото близо до 1. Което улеснява по-бързото обхождане на конфигурационното пространство. Всъщност за промяната на енергията ΔE можем да получим много прост и удобен израз, доколко се променя само знакът на магнитният момент на σ_{ij}, тогава:

$$\Delta E = H(\sigma') - H(\sigma) = 2J\sigma_{ij}\left(\sigma_{i+1,j} + \sigma_{i-1,j} + \sigma_{i,j+1} + \sigma_{i,j-1}\right) \tag{13.71}$$

отношението на двете вероятности $p(\boldsymbol{\sigma}')/p(\boldsymbol{\sigma})$ може да се опрости съществено изпозлвайки горният израз в уравнение 13.71:

$$\frac{p(\boldsymbol{\sigma}')}{p(\boldsymbol{\sigma})} = \exp\left(\frac{1}{T}\left(H(\boldsymbol{\sigma}') - H(\boldsymbol{\sigma})\right)\right) = \exp\left(\Delta E/T\right) \tag{13.72}$$

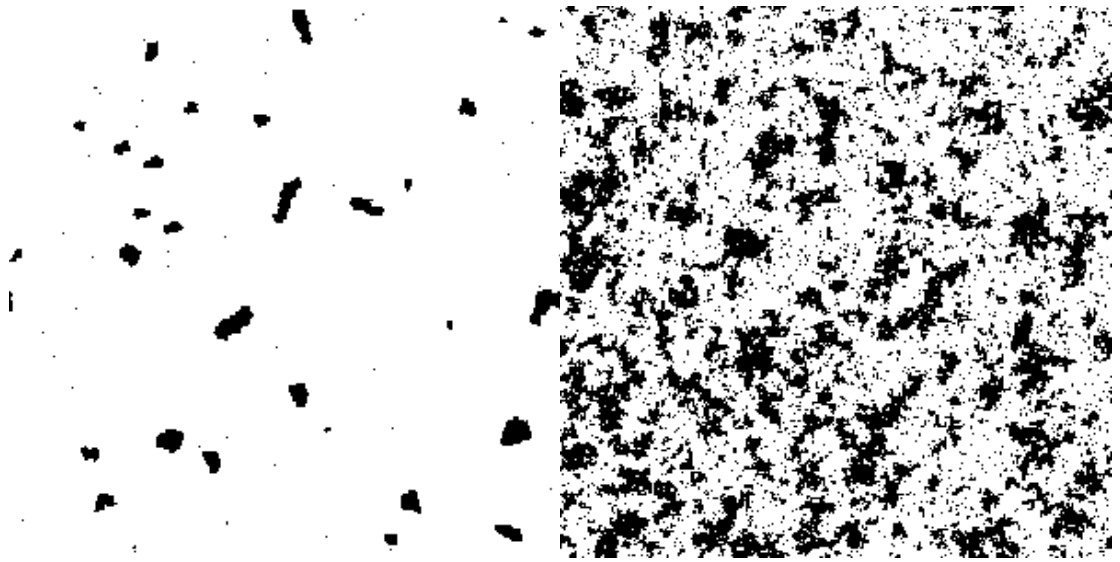
Нека обобщим алгоритъма на Метрополис за разглеждана от нас двумерна магнитна решетка $\texttt{sigma}(\texttt{0} : \texttt{n} + \texttt{1}, \texttt{0} : \texttt{n} + \texttt{1})$. Масива `sigma` включва по два допълнителни реда и колони с цел да се съхраняват и граничните условия в двете направления. Например в реда с индекс `0` се копира съдържанието на ред с индекс n. Така всички елементи от ред с индекс `1` ще имат за "горни" съседи елементите от ред с индекс `n`. Съответно елементите от ред с индекс `1` ще се копират в ред $\texttt{n} + \texttt{1}$. елементите от колона с индекс `n` ще се копират в колона с индекс `0` и накрая елементите от колона с индекс `1` ще се копират в колона $\texttt{n} + \texttt{1}$. По-долу е представен примерен псевдо код който описва алгоритъма:

Data: Избор на температура `T` и размер на решетката `n`
Data: Избор на брой стъпки за "термализиране" на системата `NiterTerm`
Data: Избор на брой продуктивни Монте-Карло стъпки `Niter`
Data: Инициализиране на двумерната решетка `sigma(i, j)`
Data: Инициализиране на генератора на случайни числа
`E = 0.0 devE = 0.0 M = 0.0 devM = 0.0` **for** `k = 1 ... NiterTerm + Niter` **do**
  Копиране на граничните условия в масива `sigma` `sigma(0, :) = sigma(n, :)`
  `sigma(n + 1, :) = sigma(1, :)`
  `sigma(:, 0) = sigma(:, n)`
  `sigma(:, n + 1) = sigma(:, 1)`
  Data: Избор на произволен възел от решетката с индекси `(i, j)`
  #Пресмятане на промяната в енергията на системата `dE`
  `dE = 2 * sigma(i, j) * (sigma(i + 1, j) + sigma(i − 1, j)`
  `+sigma(i, j + 1) + sigma(i, j − 1))`
  #Пресмятане на вероятността за преход `p`
  Data: `p = exp(dE/T)`
  if `p > 1` **then**
    $sigma(i, j) = -sigma(i, j)$
  end
  else
    Data: Избери случайно число `w` $\in [0, 1]$
    #Избор на новата конфигурация с вероятност `p`
    if `w < p` **then**
      $sigma(i, j) = -sigma(i, j)$
    end
  end
  if `k > NiterTerm` **then**
    #Акумулиране на термодинамичните величини
    `E = E + Energy(sigma)`
    `devE = devE + Energy(sigma) * *2`
    `M = M + Magnet(sigma)`
    `devM = devM + Magnet(sigma) * *2`
  end
end
#Нормализиране на термодинамичните величини
`E = E/Niter`
`M = M/Niter`
`devE = devE/Niter − E * *2`
`devM = devM/Niter − M * *2`

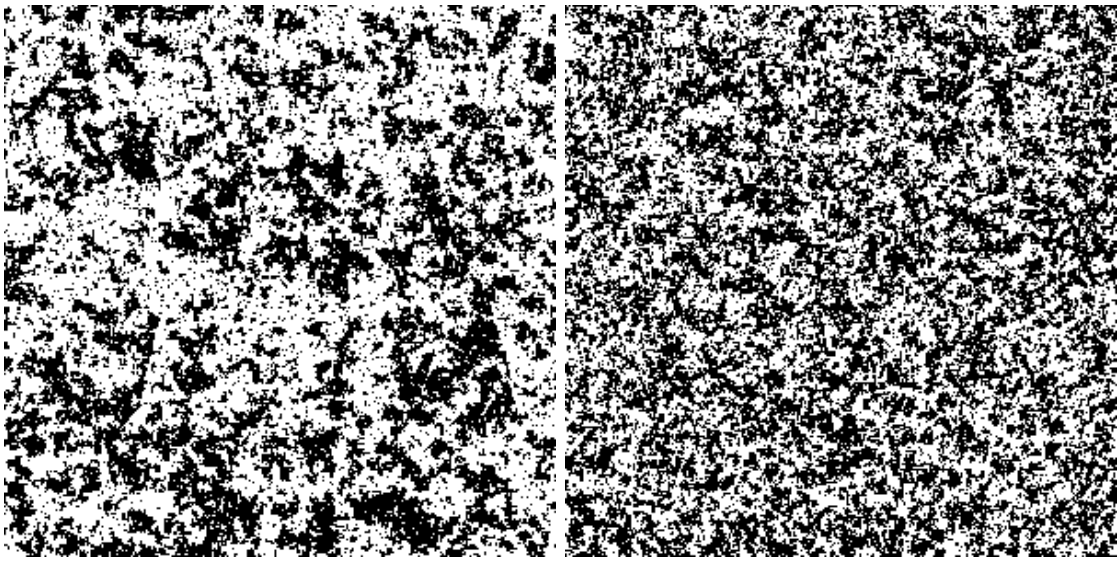
Двумерният модел на Изинг търпи фазов преход от феромагнитно към парамагнитно състояние при стойност на критичната температура $T_c = 2.2692\,[\beta J]$. Това поведение може да се демонстрира на фигура 13.4а където магнетизацията на двумерната решетка се визуализира чрез черни и бели квадрати. Квадратите с черен цвят отго-

ворят на магнетизация в посока "долу" докато белите квадрати отговарят на възел с магнетизация в посока "горе". За температури T под критичната T_c системата достига равновесно състояние с приоритетна магнетизация само в една от двете посоки. В разглежданият пример $M = 0.90\,\mu$. Когато температурата достигне критичната стойност T_c в системата се появяват домейни които се приоритетно ориентирани в едната и другата посоки фигура 13.3б, тогава общата магнетизация намалява $M = 0.38\,\mu$. Двата вида доймени (или области) съществуват съвместно подобно на състоянието пари-вода при фазовият преход свързан с кипенето на водата. За температури над критичната наблюдаваната обща магнетизация на системата е около 0μ, фигури 13.3в и 13.3г.

Показателни за описаният фазов преход между двата вида магнитно състояние на системата са и кривата на пълна магнетизация като функция на температурата $M = M(T)$, фигура 13.4а и топлинният капацитет $C_V = C_V(T)$, фигура 13.4б.

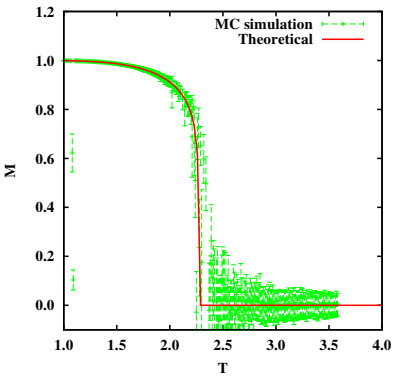


(а) $T = 1.0\,\beta J$, $\mathrm{M} = 0.90\,\mu$ (б) $T = T_c$, $\mathrm{M} = 0.38\,\mu$

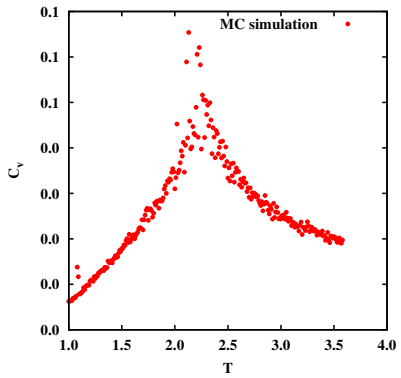


(в) $T = 2.5\,\beta J$, $\mathrm{M} = 0.19\,\mu$ (г) $T = 3.0\,\beta J$, $\mathrm{M} = 0.01\,\mu$

Фигура 13.3: Еволюция на магнетизация на двумерна решетка получена чрез модела на Изинг за различни стойности на температурата T. Белите и черни квадрати отговрят на магнетизация в посока "горе" и "долу" съответно. Демонстрирани са магнетизации за няколко възможни температури T около критичната температура $T_c = 2.2692\,[\beta J]$



(а) $T = 2.0\,\beta J$, M $= 0.91\,\mu$



(б) $T = T_c$, M $= 0.52\,\mu$

Фигура 13.4: Графики на температурната зависимост на махнитният момент $M = M(T)$ и топлинният капацитет $C_V = C_V(T)$ получени чрез Монте Карло симулацията с 5000000 стъпки и решетка с размер 100×100 възела.

Глава 14

Изчислителна физика за напреднали

14.1 Нелинейни частни диференциални уравнения

В тази част на ръководството ще опишем един метод за получаване на точни решения на нелинейни частни диференциални уравнения. Целта е да демонстрираме въпросния метод и да използваме случая, за да покажем как се пресмятат символно нелинейни алгебрични системи с помощта *Maple*. Едновременно е полезно човек да се запознае с част от апарата на нелинейните диференциални уравнения, както и да пресменте нещо с помощта на *Maple*, което притежава физичен смисъл.

Линейните диференциални уравнения описват в добро приближение част от физичните процеси. Голяма част от процесите във физиката обаче са нелинейни. В нелинейните диференциални уравнения може да има произведения от различни стенени на търсената функция и някои от нейните производни, както и членове, съдържащи разнообразни произведения, включващи функция от търсената функция. Добре известно е, че за нелинейните частни диференциални уравнения е трудно да се получи общо решение при зададени начални и гранични условия.

Нелинейната динамика е изключително интересна и всеобхватна област. Например математическата биология използва апарата на динамичните системи. Тя намира своите приложения в медицината, биотехнологиите, биологията и екосистемите. Социалните науки също са свързани с математическата биология и успешно може да се направи аналогия между социалните явления и явленията в екосистемите и биологията.

При получаването на решения на нелинейните частни диференциални уравнения няма универсални подходи, които да дават еднозначно решение на произволна задача. Нямаме единственост на решенията в глобален смисъл при фиксирани гранични или начални условия. Към различни класове уравнения се подхожда по различен начин. Затова тук е разгледан подробно един широко приложим метод за получаване на точни решения на нелинейни частни диференциални уравнения, наречен *модифициран метод на най-простото уравнение*, като за т. нар. *най-просто* уравнение се използва уравнението за функцията $\frac{1}{\cosh^2(\alpha x+\beta t)}$. Може да се използва и друг вид уравнение за

най-просто.

Този метод е разширен успешно до приложимост към уравнения с поведение на решението $\frac{1}{\cosh^n(\alpha x+\beta t)}$, където n е произволно реално положително число. Модифицираният метод на най-простото уравнение намира своите приложения в разнообразни области, включително при повърхнинните вълни в механиката на флуиди. Функцията $\frac{1}{\cosh^n(\alpha x+\beta t)}$, представлява в същността си солитон.

14.1.1 Описание на солитона

Първото описание на солитон дава Джон Скот Ръсел през 1834г. Той наблюдава дълга водна вълна, която не изменя вида си и се придвижва по Единбургския канал. Приети са три основни принципа за солитона: да има непроменлива форма; вълната да е локализирана и да клони към константа на безкрайност; отделни солитони да могат да взаимодействат помежду си, като не променят формата си.

Солитонът е вид уединена вълна. Уединената вълна запазва размера и формата си с течение на времето, докато солитона запазва тези свои характеристики дори след взаимодействие с друг солитон.

14.2 Основни уравнения за водни вълни

Нека имаме безвискозен флуид с постоянна плътност ρ, без повърхнинни напрежения, в басейн с фиксирана дълбочина h.

Пренебрегваме повърхнинните сили, защото в разглежданите от нас системи в случая, те са много по-малки в сравнение с другите действащи сили на флуида. Случаи, в които не можем да направим такова приближение, са например системи от водни капки.

Приемаме, че $x-y$ равнината е свободната повърхност на флуида, а положителната посока на z е насочена нагоре от тази повърхност.

Свободната повърхност на височина h се описва от $z = \zeta(x, y, t)$. Уравнението $z = 0$ задава хоризонталното твърдо дъно. Свободната повърхност се задава от уравнението $z = h + \zeta$.

За да опишем нелинейните водни вълни в плитка вода е необходимо да опростим системата на общата нелинейна задача за вълни на свободната повърхност на флуид.

Параметърът l е характерна дължина, λ е дължината на вълната, a е амплитудата, $c = \sqrt{gh}$ е характерната скорост за водните вълни.

Въвеждаме следните два параметъра ϵ и δ, които характеризират нелинейните плитки вълни. Тези параметри са важни, защото чрез тях определяме кои членове са от значение в уравненията и кои можем да пренебрегнем в различните случаи.

Линеаризираната теория за повърхнинните вълни се получава, когато параметрите $\epsilon = a/h$ и $\kappa = ak$ са малки, където a е амплитудата на повърхнинната вълна, k е вълновото число, h е дълбочината на басейна.

$$\epsilon = \frac{a}{h}, \quad \delta = \frac{h^2}{l^2}.$$

Основните уравнения за водни вълни могат да бъдат записани в безразмерна форма

$$\delta\left(\frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2}\right) + \frac{\partial^2\phi}{\partial z^2} = 0, \tag{14.1}$$

$$\frac{\partial\phi}{\partial t} + \frac{\epsilon}{2}\left[\left(\frac{\partial\phi}{\partial x}\right)^2 + \left(\frac{\partial\phi}{\partial y}\right)^2\right] + \frac{\epsilon}{2\delta}\left(\frac{\partial\phi}{\partial z}\right)^2 + \zeta = 0 \text{ при } z = 1 + \epsilon\zeta, \tag{14.2}$$

$$\delta\left[\frac{\partial\zeta}{\partial t} + \epsilon\left(\frac{\partial\phi}{\partial x}\frac{\partial\zeta}{\partial x} + \frac{\partial\phi}{\partial y}\frac{\partial\zeta}{\partial y}\right)\right] - \frac{\partial\phi}{\partial z} = 0 \text{ при } z = 1 + \epsilon\zeta, \tag{14.3}$$

$$\frac{\partial\phi}{\partial z} = 0 \text{ при } z = 0. \tag{14.4}$$

Ако $\epsilon \ll 1$, членовете, съдържащи ϵ, могат да бъдат пренебрегнати в системата (14.1 - 14.4).

Характерно свойство на теорията на плитките вълни е, че $\delta \ll 1$. С известни пресмятания и преобразувания, може да се достигне до трите *безразмерни уравнения за плитка вода*:

$$\frac{\partial u}{\partial t} - \frac{\delta}{2}\left(\frac{\partial^3 u}{\partial t\partial x^2} + \frac{\partial^3 v}{\partial t\partial y\partial x}\right) + \epsilon\left(u\frac{\partial u}{\partial x} + v\frac{\partial v}{\partial x}\right) + \frac{\partial\zeta}{\partial x} = 0, \tag{14.5}$$

$$\frac{\partial v}{\partial t} - \frac{\delta}{2}\left(\frac{\partial^3 u}{\partial t\partial x\partial y} + \frac{\partial^3 v}{\partial t\partial y^2}\right) + \epsilon\left(u\frac{\partial u}{\partial y} + v\frac{\partial v}{\partial y}\right) + \frac{\partial\zeta}{\partial y} = 0. \tag{14.6}$$

$$\frac{\partial\zeta}{\partial t} + \frac{\partial[u(1+\epsilon\zeta)]}{\partial x} + \frac{\partial[v(1+\epsilon\zeta)]}{\partial y} = \frac{\delta}{6}\left(\frac{\partial(\Delta u)}{\partial x} + \frac{\partial(\Delta v)}{\partial y}\right). \tag{14.7}$$

Използваме факта, че ϕ_0 е безвихрово течение, т.е. $\frac{\partial u}{\partial y} = \frac{\partial v}{\partial x}$ и пренебрегваме $\delta-$членове в уравнения (14.5 – 14.7), с което получаваме фундаменталните уравнения за плитка вода:

$$\frac{\partial u}{\partial t} + \epsilon\left(u\frac{\partial u}{\partial x} + v\frac{\partial v}{\partial x}\right) + \frac{\partial\zeta}{\partial x} = 0 \tag{14.8}$$

$$\frac{\partial v}{\partial t} + \epsilon\left(u\frac{\partial u}{\partial y} + v\frac{\partial v}{\partial y}\right) + \frac{\partial\zeta}{\partial y} = 0 \tag{14.9}$$

$$\frac{\partial\zeta}{\partial t} + \frac{\partial[u(1+\epsilon\zeta)]}{\partial x} + \frac{\partial[v(1+\epsilon\zeta)]}{\partial y} = 0. \tag{14.10}$$

Тази система от три свързани нелинейни уравнения е затворена и позволява някои интересни решения за u, v, ζ. Линеаризира се при $\epsilon \ll 1$.

14.2.1 Система на Бусинеск

Забележка: Ще означаваме производни по променливи с долни индекси след функциите.

Нека разгледаме едномерния случай на системата уравнения (14.5 – 14.7) и запазим членовете от първи порядък по ϵ и δ. Тогава получаваме уравненията на Бусинеск:

$$u_t + \epsilon u u_x + \zeta_x - \frac{\delta}{2} u_{txx} = 0$$

$$\zeta_t + (u(1 + \epsilon\zeta))_x - \frac{\delta}{6} u_{xxx} = 0.$$

Еквивалентна на тази система (с размерни величини и $\delta < 1$), е:

$$u_t + u u_x + g\zeta_x = \frac{1}{3} h^2 u_{txx}$$

$$\zeta_t + ((h + \zeta)u)_x = 0.$$

Тя описва еволюцията на дълги водни вълни, разпространяващи се в положителната и отрицателната посока на x. Елиминирайки ζ и пренебрегвайки членове, сравними с $o(\epsilon, \delta)$, стигаме до уравнение на Бусинеск:

$$u_{tt} - c^2 u_{xx} + \frac{1}{2}(u^2)_{xt} = \frac{h^2}{3} u_{xxtt}. \tag{14.11}$$

Уравнението на Бусинеск е моделно уравнение за плитки водни вълни, приложимо за слабо нелинейни дълги водни вълни.

Нормирана форма на уравнението на Бусинеск:

$$u_{tt} - u_{xx} - \frac{3}{2}(u^2)_{xx} - u_{xxxx} = 0 \tag{14.12}$$

Общи вид на уравнението на Бусинеск:

$$u_{tt} - u_{xx} - u_{xxxx} - K(u^2)_{xx} = 0$$

Запазващи се величини

Бусинеск е установил три инвариантни физични величини:

- обем на солитона

$$Q = \int_{-\infty}^{+\infty} \zeta dx, \tag{14.13}$$

- енергия на солитона

$$E = \int_{-\infty}^{+\infty} \zeta^2 dx, \tag{14.14}$$

- "момент на нестабилност"

$$M = \int_{-\infty}^{+\infty} \left(\frac{\partial \zeta}{\partial x}\right)^2 - 3\left(\frac{\zeta}{h}\right)^3 dx. \tag{14.15}$$

Вариационната задача $\delta M = 0$ при фиксирана енергия E води до решение тип солитон.

Бусинеск е получил и точни изрази за амплитудата и обема на солитона за фиксирана енергия:

$$a = \frac{3E^{3/2}}{4h}, \quad Q = 3hE^{1/3}$$

14.2.2 Уравнение на Кортевег-де Фриз

Приемаме, че ϵ и δ са сравними по големина - запазваме членовете, които ги съдържат в системата (14.5 – 14.7). Уравненията придобиват следната форма:

$$u_t - \frac{\delta}{2} u_{txx} + \epsilon u u_x + \zeta_x = 0, \tag{14.16}$$

$$\zeta_t + (u(1+\epsilon\zeta))_x - \frac{\delta}{6} u_{xxx} = 0. \tag{14.17}$$

Търсим устойчиви решения, които се разпространяват по положителната посока на x: $u = u(x-Ut), \zeta = \zeta(x-Ut), c = 1$. Предполагаме решение от вида: $u = \zeta + \epsilon P + \delta Q$.

P, Q са неизвестни функции, които ще определим.

Заместваме предположеното решение в уравненията (14.16) и (14.17). Те са съвместими, което означава че приравняваме изразите пред съответните степени. Пренебрегвайки членове от втори порядък и знаейки, че ϵ/δ е от първи порядък, стигаме до едно уравнение за профила на вълната ζ:

$$\zeta_t + (1 + \frac{3}{2}\epsilon\zeta)\zeta_x + \frac{\delta}{6}\zeta_{xxx} = 0 \tag{14.18}$$

Това е уравнението на Кортевег-де Фриз [**?**], което е изведено през 1895г. То описва дълги повърхнинни вълни в течност. Можем да получим точното му решение като приложим модифицирания метод на най-простото уравнение за решаване на нелинейни частни диференциални уравнения, който ще разгледаме подробно в следващата глава.

Запазващи се величини

Величина, за която може да се запише уравнение за непрекъснатост (**??**), се нарича запазваща се величина за конкретната задача. Запазващите величини могат да улеснят решаването на една задача до известна степен.

Нека запишем уравнението за непрекъснатост по следния начин:

$$\frac{\partial E}{\partial r} + \nabla.(\vec{J}(E)) = 0, \tag{14.19}$$

където E е величината, а $\vec{J}(E)$ е нейният поток.

Разглеждаме уравнението на Кортевег-де Фриз в каноничен вид

$$u_t - 6uu_x + u_{xxx} = 0.$$

Доказано е, че уравнението на Кортевег-де Фриз има безкраен брой запазващи се величини. [**?**] Такива уравнения с безкраен брой запазващи се величини имат солитонни решения, което подсказва за връзка между двете свойства. Примери за запазващи се величини в уравнението на Кортевег-де Фриз са следните:

$$E_1 = u \quad \text{и} \quad J_1 = u_{xx} - 3u^2,$$

чиито физически смисъл е запазваща се маса,

$$E_2 = u^3 + \frac{1}{2}(u_x)^2 \quad \text{и} \quad J_2 = -\frac{9}{2}u^4 + 3u^2u_{xx} - 6u(u_x)^2 + u_xu_{xxx} - \frac{1}{2}(u_{xx})^2,$$

чиито физически смисъл е запазваща се плътност на енергията и

$$E_3 = \frac{1}{2}u^2 \quad \text{и} \quad J_3 = uu_x - 2u^3 - \frac{1}{2}(u_x)^2,$$

което се получава, когато умножим израза с u и го запишем във вид на уравнение за непрекъснатост. Физическия смисъл на последния закон за запазване е запазващ се хоризонтален импулс на вълните.

Можем да съобразяваме с какво да умножаваме уравнението, за да се получи нов закон за запазване, но при високи порядъци това става много сложна задача.

14.2.3 Модифициран метод на най-простото уравнение за решаване на нелинейни частни диференциални уравнения

Методът се състои в следното: Чрез подходящо предположение за решение свеждаме нелинейното ЧДУ до нелинейно ОДУ

$$P(u, u_\xi, u_{\xi\xi}, ...) = 0 \tag{14.20}$$

След това редът $u(\xi) = \sum_{\mu=-\nu}^{\nu_1} p_\mu [f(\xi)]^\mu$ е заместен в горното уравнение. p_μ са коефициенти и $f(\xi)$ е решение на по-просто уравнение, наречено най-просто.

Резултатът от заместването е полином на $f(\xi)$. Функцията $u(\xi)$ е решение на $P = 0$, ако всички коефициенти на получения полином на $f(\xi)$ са равни на 0. Това условие води до система нелинейни алгебрични уравнения. Всяко нетривиално решение на тази система съответства на решение на изследваното нелинейно ЧДУ.

Нека имаме решения тип бягаща вълна $u(x,t) = u(\xi) = u(\alpha x + \beta t)$, построени въз основата на най-простото уравнение

$$f_\xi^2 = n^2 f^2 - n^2 f^{\frac{2n+2}{n}} \tag{14.21}$$

чието решение е

$$f(\xi) = \frac{1}{\cosh^n(\xi)},$$

където n е произволно реално положително число.

Търсим решения във вид на уединени вълни за класовете нелинейни ЧДУ, съдържащи членове с производни, чиито ред по отношение на участващите производни е четен, и членове с производни, чиито ред по отношение на участващите производни е нечетен.

Теорема

Нека P е полином на функцията $u(x,t)$ и нейните производни. Функцията $u(x,t)$ принадлежи на класа на диференцируемост C^k, където k е най-високият ред производна, участваща в P. Полиномът P може да съдържа някои или всички от следните части:

- полином на u;
- членове, съдържащи производни на u по x и/или произведения от такива производни. Всеки такъв член може да бъде умножен по полином на u;
- членове, съдържащи производни на u по t и/или произведения от такива производни. Всеки такъв член може да бъде умножен по полином на u;
- членове, съдържащи смесени производни на u по x и t и/или произведения от такива производни. Всеки такъв член може да бъде умножен по полином на u;
- членове, съдържащи произведения от производни на u по x и производни на u по t. Всеки такъв член може да бъде умножен по полином на u; (F) членове, съдържащи произведения от производни на u по x и смесени производни на u по x и t.
- членове, съдържащи произведения от производни на u по x и смесени производни на u по x и t. Всеки такъв член може да бъде умножен по полином на u;
- членове, съдържащи произведения от производни на u по t и смесени производни на u по x и t. Всеки такъв член може да бъде умножен по полином на u;
- членове, съдържащи произведения от производни на u по x, производни на u по t и смесени производни на u по x и t. Всеки такъв член може да бъде умножен по полином на u.

Нека е дадено следното нелинейно ЧДУ:

$$P = 0 \tag{14.22}$$

Търсим решение от вида

$$u(\xi) = \gamma f(\xi), \xi = \alpha x + \beta t.$$

Тук γ е параметър и $f(\xi)$ е решение на най-простото уравнение

$$f_\xi^2 = n^2 f^2 - n^2 f^{\frac{2n+2}{n}}.$$

Заместването на това решение в (14.22) води до връзка R от вида

$$R = \sum_{i=0}^{N} C_i f(\xi)^i + f_\xi \sum_{j=0}^{M} D_j f(\xi)^j, \tag{14.23}$$

където M и N са естествени числа, зависещи от вида на полинома P.

Коефициентите C_i и D_j зависят от параметрите на уравнение (14.22) и от α, β, γ. Тогава всяко нетривиално решение на нелинейната алгебрична система

$$C_i = 0, i = 1, 2, ..., N; D_j = 0, j = 1, 2, ...M \tag{14.24}$$

води до солитонно решение на нелинейното частно диференциално уравнение (14.22).

Доказателство

Ще докажем следното:

Нека $f(\xi)$ е решение на нелинейното ОДУ

$$f_\xi^2 = n^2 f^2 - n^2 f^{\frac{2n+2}{n}}.$$

Тогава четните производни на $f(\xi)$ съдържат само полиноми на $f(\xi)$. Нечетните производни на $f(\xi)$ съдържат полиноми на $f(\xi)$, умножени по $\frac{df(\xi)}{d\xi}$. Ще използваме метода на индукцията. Горното твърдение е вярно за втора, трета и четвърта производна на $f(\xi)$. Нека $2n$-тата производна на f бъде полином на f:

$$F(f) = \frac{d^{2n} f}{d\xi^{2n}}$$

Тогава

$$\frac{d^{2n+1} f}{d(\xi)^{2n+1}} = \frac{dF(f)}{df}\frac{df}{d\xi},$$

$$\frac{d^{2n+2} f}{d(\xi)^{2n+2}} = \frac{dF^2(f)}{df^2}\left(\frac{df}{d\xi}\right)^2 + \frac{dF}{d\xi}\frac{d^2 f}{d\xi^2} = F^*(f).$$

Следователно, всяка четна производна на f е полином на f и всяка нечетна производна на f е полином на f, умножен по $\frac{df}{d\xi}$.

Нека нелинейното ЧДУ $P = 0$ бъде редуцирано чрез предположено решение тип бягаща вълна до нелинейно ОДУ $Q = 0$, където всеки член има нечетни или четни степени по отношение на участващите производни. За членове с нечени степени, избраният вид на решението редуцира допълнително съответстващата част от Q до връзка от вида $\sum_{i=0}^{N} C_i f(\xi)^i$. За членовете с четни степени, избраният вид на решението редуцира допълнително съответстващата част от Q до връзка от вида $f_\xi \sum_{j=0}^{M} D_j f(\xi)^j$.

По този начин Q е редуцирано до връзка от вида R,

$$R = \sum_{i=0}^{N} C_i f(\xi)^i + f_\xi \sum_{j=0}^{M} D_j f(\xi)^j.$$

Тогава всяко нетривиално решение на нелинейната алгебрична система (ако съществува такова решение)

$$C_i = 0, i = 1, 2, ..., N; D_j = 0, j = 1, 2, ...M,$$

води до решение във вид на уединена вълна на нелинейното частно диференциално уравнение $P = 0$. □

14.2.3.1 Случай $n = 2$

Уравнения, които могат да бъдат решени, използвайки модифицирания метод с предположение за решението $1/\cosh^2(\xi)$, описват повърхнинни вълни в механиката на флуиди. Примери за такива уравнения са известните уравнения на Кортевег-де Фриз, Дегасперис-Процеси, Бусинеск и други.

В случая $n = 2$ най-простото уравнение има вида

$$f_\xi^2 = 4f^3 - 4f^2. \tag{14.25}$$

Тук функцията $f(\xi)$ е решение на най-простото уравнение.

Пример за такова уравнение:

$$\nu u_{ttt} + \pi u u_{xx} + \sigma u_t^2 + (\delta + \mu u) u_x + \omega u^3 = 0, \tag{14.26}$$

където $u(\xi) = \gamma f(\xi)$ е търсената функция и $\xi = \alpha x + \beta t$.

Заместването на конкретния вид решение $u(x,t) = u(\xi) = \gamma f(\xi)$ в (14.26) води до обикновено диференциално уравнение, което съдържа членове от четна и нечетна степен по отношение на производните на $u(\xi)$. То изглежда по следния начин:

$$\nu \frac{d^3u}{d\xi^3}\frac{d^3\xi}{dt^3} + \pi u \frac{d^2u}{d\xi^2}\frac{d^2\xi}{dx^2} + \sigma\left(\frac{du}{d\xi}\frac{d\xi}{dt}\right)^2 + (\delta + \mu u)\frac{du}{d\xi}\frac{d\xi}{dx} + \omega u^3 = 0 \tag{14.27}$$

Вземаме предвид, че $u(\xi) = \gamma f(\xi)$. От най-простото уравнение (14.25) можем да добием вида на втората, третата, четвъртата производна и така нататък, изразени

именно чрез f. Заместваме ги в (14.27) и от коефициентите пред различните степени на f всъщност съставяме системата от нелинейни алгебрични уравнения. Прилагането на Теоремата от по-горе води до следната система нелинейни алгебрични уравнения:

$$-12\beta^3\nu + \alpha\gamma\mu = 0$$

$$4\beta^3\nu + \alpha\delta = 0$$

$$-6\pi\alpha^2 - 4\beta^2\sigma + \gamma\omega = 0$$

$$\pi\alpha^2 + \beta^2\sigma = 0$$

Целта е да изразим параметрите α, β, γ чрез уравненията от системата. Това правим с помощта на *Maple*. Едно възможно решение на тази система е

$$\alpha = -\frac{1}{2}\frac{(-\sigma\delta^2\nu^2)^{3/4}}{\delta\nu^2\pi^{3/4}}; \beta = \frac{1}{2}\frac{(-\sigma\delta^2\nu^2)^{1/4}}{\nu\pi^{1/4}};$$

$$\gamma = -\frac{3\delta}{\mu}; \omega = \frac{1}{6}\frac{\sigma\mu(-\sigma\delta^2\nu^2)^{1/2}}{\delta\nu^2\pi^{1/2}}$$

Съответстващото солитонно решение на уравнение (14.26) е

$$u(x,t) = -\frac{3\delta}{\mu\cosh^2(-\frac{1}{2}\frac{(-\sigma\delta^2\nu^2)^{3/4}}{\delta\nu^2\pi^{3/4}}x + \frac{1}{2}\frac{(-\sigma\delta^2\nu^2)^{1/4}}{\nu\pi^{1/4}}t)}$$

За повърхнинните вълни изведохме уравнението на Бусинеск за профила на повърхността:

$$\zeta_{tt} - c^2\zeta_{xx} = \frac{h^2}{3}\zeta_{xxxx} + \frac{3}{2}(\frac{\zeta^2}{h})_{xx}$$

Можем да приложим модифицирания метод на най-простото уравнение и да получим точно решение на уравнението на Бусинеск във вид на солитон. Пресмятаме нелинейната алгебрична система с помощта на компютърната програма *Maple* и получаваме:

$$\zeta(x,t) = \frac{10\alpha^4}{\cosh^2(\alpha x + \sqrt{30\alpha^4 + 4\alpha^2 + 1}\alpha t)}$$

Също така можем да получим и решение на уравнението на Кортевег-де Фриз (14.18), описващо дългите повърхнинни вълни в течност, с помощта на този метод. Уравнението на Кортевег-де Фриз за профила на повърхността на вълната е:

$$\zeta_t + (1 + \frac{3}{2}\epsilon\zeta)\zeta_x + \frac{\delta}{6}\zeta_{xxx} = 0$$

Решението за профила на повърхността изглежда по следния начин:

$$\zeta(x,t) = \frac{8\alpha^3\frac{\delta}{\epsilon}}{\cosh^2(\alpha x - \alpha(4\alpha^2\delta + 1)t)}$$

14.3 Молекулна динамика

14.4 Квантово-механични пресмятания

Приложение A

Изходен код

A.1 Линейни системи

A.1.1 Метод на Гаус-Жордан

Решаване на линейни системи чрез метода на Гаус-Жордан изходен код. Версия на програмният език *Fortran*

```
1   SUBROUTINE gaussj(a,b)
2   USE nrtype; USE nrutil, ONLY : assert_eq,nrerror,outerand,outerprod,swap
3   IMPLICIT NONE
4   REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a,b
5   INTEGER(I4B), DIMENSION(size(a,1)) :: ipiv,indxr,indxc
6   LOGICAL(LGT), DIMENSION(size(a,1)) :: lpiv
7   REAL(SP) :: pivinv
8   REAL(SP), DIMENSION(size(a,1)) :: dumc
9   INTEGER(I4B), TARGET :: irc(2)
10  INTEGER(I4B) :: i,l,n
11  INTEGER(I4B), POINTER :: irow,icol
12  n=assert_eq(size(a,1),size(a,2),size(b,1),'gaussj')
13  irow => irc(1)
14  icol => irc(2)
15  ipiv=0
16  do i=1,n
17    lpiv = (ipiv == 0)
18    irc=maxloc(abs(a),outerand(lpiv,lpiv))
19    ipiv(icol)=ipiv(icol)+1
20    !if (ipiv(icol) > 1) call nrerror('gaussj: singular matrix (1)')
21    if (ipiv(icol) > 1) exit
22    if (irow /= icol) then
23      call swap(a(irow,:),a(icol,:))
24      call swap(b(irow,:),b(icol,:))
25    end if
26    indxr(i)=irow
27    indxc(i)=icol
28    if (a(icol,icol) == 0.0) &
```

```
29          exit
30                              !call nrerror('gaussj: singular matrix (2)')
31
32                              pivinv=1.0_sp/a(icol,icol)
33        a(icol,icol)=1.0
34        a(icol,:)=a(icol,:)*pivinv
35        b(icol,:)=b(icol,:)*pivinv
36        dumc=a(:,icol)
37        a(:,icol)=0.0
38        a(icol,icol)=pivinv
39        a(1:icol-1,:)=a(1:icol-1,:)-outerprod(dumc(1:icol-1),a(icol,:))
40        b(1:icol-1,:)=b(1:icol-1,:)-outerprod(dumc(1:icol-1),b(icol,:))
41        a(icol+1:,:)=a(icol+1:,:)-outerprod(dumc(icol+1:),a(icol,:))
42        b(icol+1:,:)=b(icol+1:,:)-outerprod(dumc(icol+1:),b(icol,:))
43      end do
44      do l=n,1,-1
45        call swap(a(:,indxr(l)),a(:,indxc(l)))
46      end do
47      END SUBROUTINE gaussj
```

Версия на програмният език *C++*

```
1   #include <math.h>
2   #define NRANSI
3   #include "nrutil.h"
4   #define SWAP(a,b) {temp=(a);(a)=(b);(b)=temp;}
5
6   void gaussj(float **a, int n, float **b, int m)
7   {
8     int *indxc,*indxr,*ipiv;
9     int i,icol,irow,j,k,l,ll;
10    float big,dum,pivinv,temp;
11
12    indxc=ivector(1,n);
13    indxr=ivector(1,n);
14    ipiv=ivector(1,n);
15    for (j=1;j<=n;j++) ipiv[j]=0;
16    for (i=1;i<=n;i++) {
17      big=0.0;
18      for (j=1;j<=n;j++)
19        if (ipiv[j] != 1)
20          for (k=1;k<=n;k++) {
21            if (ipiv[k] == 0) {
22              if (fabs(a[j][k]) >= big) {
23                big=(float)fabs(a[j][k]);
24                irow=j;
25                icol=k;
26              }
27            } else if (ipiv[k] > 1) nrerror("gaussj:␣Singular␣Matrix-1");
28          }
29      ++(ipiv[icol]);
30      if (irow != icol) {
31        for (l=1;l<=n;l++) SWAP(a[irow][l],a[icol][l])
```

```
32          for (l=1;l<=m;l++) SWAP(b[irow][l],b[icol][l])
33        }
34        indxr[i]=irow;
35        indxc[i]=icol;
36        if (a[icol][icol] == 0.0) nrerror("gaussj:␣Singular␣Matrix-2");
37        pivinv=1.0f/a[icol][icol];
38        a[icol][icol]=1.0;
39        for (l=1;l<=n;l++) a[icol][l] *= pivinv;
40        for (l=1;l<=m;l++) b[icol][l] *= pivinv;
41        for (ll=1;ll<=n;ll++)
42          if (ll != icol) {
43            dum=a[ll][icol];
44            a[ll][icol]=0.0;
45            for (l=1;l<=n;l++) a[ll][l] -= a[icol][l]*dum;
46            for (l=1;l<=m;l++) b[ll][l] -= b[icol][l]*dum;
47          }
48      }
49      for (l=n;l>=1;l--) {
50        if (indxr[l] != indxc[l])
51          for (k=1;k<=n;k++)
52            SWAP(a[k][indxr[l]],a[k][indxc[l]]);
53      }
54      free_ivector(ipiv,1,n);
55      free_ivector(indxr,1,n);
56      free_ivector(indxc,1,n);
57    }
58    #undef SWAP
59    #undef NRANSI
```

А.1.2 LU - декомпозиция

Изходен код на процедурата`ludcmp` реализиран на програмният език C.

```
1   #include <math.h>
2
3   #define TINY 1.0e-20;
4
5   void ludcmp(a,n,indx,d)
6   int n,*indx;
7   float **a,*d;
8   {
9     int i,imax,j,k;
10    float big,dum,sum,temp;
11    float *vv,*vector();
12    void nrerror(),free_vector();
13
14    vv=vector(1,n);
15    *d=1.0;
16    for (i=1;i<=n;i++) {
17      big=0.0;
18      for (j=1;j<=n;j++)
```

```
19          if ((temp=fabs(a[i][j])) > big) big=temp;
20        if (big == 0.0) nrerror("Singular matrix in routine LUDCMP");
21        vv[i]=1.0/big;
22      }
23      for (j=1;j<=n;j++) {
24        for (i=1;i<j;i++) {
25          sum=a[i][j];
26          for (k=1;k<i;k++) sum -= a[i][k]*a[k][j];
27          a[i][j]=sum;
28        }
29        big=0.0;
30        for (i=j;i<=n;i++) {
31          sum=a[i][j];
32          for (k=1;k<j;k++)
33            sum -= a[i][k]*a[k][j];
34          a[i][j]=sum;
35          if ( (dum=vv[i]*fabs(sum)) >= big) {
36            big=dum;
37            imax=i;
38          }
39        }
40        if (j != imax) {
41          for (k=1;k<=n;k++) {
42            dum=a[imax][k];
43            a[imax][k]=a[j][k];
44            a[j][k]=dum;
45          }
46          *d = -(*d);
47          vv[imax]=vv[j];
48        }
49        indx[j]=imax;
50        if (a[j][j] == 0.0) a[j][j]=TINY;
51        if (j != n) {
52          dum=1.0/(a[j][j]);
53          for (i=j+1;i<=n;i++) a[i][j] *= dum;
54        }
55      }
56      free_vector(vv,1,n);
57    }
58
59    #undef TINY
```

Изходен код на процедуратаludcmp реализиран на програмният език *Fortran*.

```
1   SUBROUTINE ludcmp(a,n,np,indx,d)
2   PARAMETER (nmax=100,tiny=1.0e-20)
3   DIMENSION a(np,np),indx(n),vv(nmax)
4   d=1.
5   do i=1,n
6       aamax=0.
7       do j=1,n
8           if (abs(a(i,j))>aamax) aamax=abs(a(i,j))
```

```
9       end do
10      if (aamax==0.) pause 'singular matrix.'
11      vv(i)=1./aamax
12  end do
13  do j=1,n
14      if (j>1) then
15          do i=1,j-1
16              sum=a(i,j)
17              if (i>1) then
18                  do k=1,i-1
19                      sum=sum-a(i,k)*a(k,j)
20                  end do
21                  a(i,j)=sum
22              endif
23          end do
24      endif
25      aamax=0.
26      do i=j,n
27          sum=a(i,j)
28          if (j>1) then
29              do k=1,j-1
30                  sum=sum-a(i,k)*a(k,j)
31              end do
32              a(i,j)=sum
33          endif
34          dum=vv(i)*abs(sum)
35          if (dum>=aamax) then
36              imax=i
37              aamax=dum
38          endif
39      end do
40      if (j/=imax) then
41          do k=1,n
42              dum=a(imax,k)
43              a(imax,k)=a(j,k)
44              a(j,k)=dum
45          end do
46          d=-d
47          vv(imax)=vv(j)
48      endif
49      indx(j)=imax
50      if (j/=n) then
51          if (a(j,j)==0.) a(j,j)=tiny
52          dum=1./a(j,j)
53          do i=j+1,n
54              a(i,j)=a(i,j)*dum
55          end do
56      endif
57  end do
58  if(a(n,n)==0.) a(n,n)=tiny
59  END SUBROUTINE ludcmp
```

А.2 Изходен код към примерните програми за полиномиална интерполация

Програмата `example_polint.f90` реализира полиномиална интерполация върху 10 случайно подбрани точки в интервала $x \in [0.25, 3]$ за функцията $f(x) = e^{(x-2/3)^2}$

```
PROGRAM neville
USE utils
IMPLICIT NONE

REAL, DIMENSION(10) :: x, y
REAL :: h,xmin, xmax, xx, yy, erry, step
INTEGER i, n, alloc_stat

xmin = 5.e-1
xmax = 1.e0

x = (/0.25, 0.5 , 0.67, 0.7 , 0.74 , 1.0, 1.5, 2.3, 2.88, 3.0/)
y(:) = exp(-(x(:)- 0.75e0)**2)

xmin = minval(x(:))
xmax = maxval(x(:))

n = 20
step = (xmax - xmin) / n


DO i = 0, n
  xx = xmin + step * i
  call polint(x,y,xx,yy,erry)
  print *,xx,yy,erry,exp(-(xx-0.75e0)**2)
END DO


END PROGRAM neville
```

изходен код на помощната процедура `polint`

```
MODULE utils
CONTAINS
SUBROUTINE polint(xa,ya,x,y,dy)
IMPLICIT NONE
REAL, DIMENSION(:), INTENT(IN) :: xa,ya
REAL, INTENT(IN) :: x
REAL, INTENT(OUT) :: y,dy
INTEGER :: m,n,ns
INTEGER, DIMENSION(1) :: indx
REAL, DIMENSION(size(xa)) :: c,d,den,ho
n=size(xa)
c=ya
d=ya
ho=xa-x
indx=minloc(abs(x-xa))
ns=indx(1)
```

```
y=ya(ns)
ns=ns-1
do m=1,n-1
  den(1:n-m)=ho(1:n-m)-ho(1+m:n)
  if (any(den(1:n-m) == 0.0)) then
    write(*,*)'polint: calculation failure'
                  stop
       end if
  den(1:n-m)=(c(2:n-m+1)-d(1:n-m))/den(1:n-m)
  d(1:n-m)=ho(1+m:n)*den(1:n-m)
  c(1:n-m)=ho(1:n-m)*den(1:n-m)
  if (2*ns < n-m) then
    dy=c(ns+1)
  else
    dy=d(ns)
    ns=ns-1
  end if
  y=y+dy
end do
END SUBROUTINE polint
END MODULE utils
```

А.3 Изходен код на процедурата ratint

```
SUBROUTINE ratint(xa,ya,n,x,y,dy)
INTEGER n,NMAX
REAL dy,x,y,xa(n),ya(n),TINY
PARAMETER (NMAX=10,TINY=1.e-25)
INTEGER i,m,ns
REAL dd,h,hh,t,w,c(NMAX),d(NMAX)
ns=1
hh=abs(x-xa(1))
do i=1,n
  h=abs(x-xa(i))
  if (h==0.)then
    y=ya(i)
    dy=0.0
    return
  else if (h<hh) then
    ns=i
    hh=h
  endif
  c(i)=ya(i)
  d(i)=ya(i)+TINY
end do
y=ya(ns)
ns=ns-1
do m=1,n-1
  do i=1,n-m
    w=c(i+1)-d(i)
```

```
      h=xa(i+m)-x
      t=(xa(i)-x)*d(i)/h
      dd=t-c(i+1)
      if(dd==0.) pause 'failure in ratint'
      dd=w/dd
      d(i)=c(i+1)*dd
      c(i)=t*dd
    end do
    if (2*ns<n-m)then
      dy=c(ns+1)
    else
      dy=d(ns)
      ns=ns-1
    endif
    y=y+dy
  end do
  END SUBROUTINE ratint
```

А.4 Изходен код на процедурата zbrac

```
                SUBROUTINE zbrac(func,x1,x2,succes)

                IMPLICIT NONE
                REAL, INTENT(INOUT) :: x1,x2
                LOGICAL, INTENT(OUT) :: succes
                INTERFACE
                        FUNCTION func(x)
                        IMPLICIT NONE
                        REAL, INTENT(IN) :: x
                        REAL :: func
                        END FUNCTION func
                END INTERFACE
                INTEGER, PARAMETER :: NTRY=50
                REAL, PARAMETER :: FACTOR=1.6e0
                INTEGER :: j
                REAL :: f1,f2
                if (x1 == x2) then
                        write(0,*)'zbrac: you have to guess an initial range'
                        stop
                end if
      f1=func(x1)
                f2=func(x2)
                succes=.true.
                do j=1,NTRY
                        if ((f1 > 0.0 .and. f2 < 0.0) .or. &
                                (f1 < 0.0 .and. f2 > 0.0)) RETURN
                        if (abs(f1) < abs(f2)) then
                                x1=x1+FACTOR*(x1-x2)
                                f1=func(x1)
                        else
                                x2=x2+FACTOR*(x2-x1)
                                f2=func(x2)
```

```
33                      end if
34                  end do
35                  succes=.false.
36                  END SUBROUTINE zbrac
```

А.5 Изходен код на процедурата zbrak

```
1                   SUBROUTINE zbrak(func,x1,x2,n,xb1,xb2,nb)
2                   IMPLICIT NONE
3                   INTEGER, INTENT(IN) :: n
4                   INTEGER, INTENT(OUT) :: nb
5                   REAL, INTENT(IN) :: x1,x2
6                   REAL, DIMENSION(:), POINTER :: xb1,xb2
7                   INTERFACE
8                       FUNCTION func(x)
9                       IMPLICIT NONE
10                      REAL, INTENT(IN) :: x
11                      REAL :: func
12                      END FUNCTION func
13                  END INTERFACE
14                  INTEGER :: i
15                  REAL :: dx
16                  REAL, DIMENSION(0:n) :: f,x
17                  LOGICAL, DIMENSION(1:n) :: mask
18                  LOGICAL, SAVE :: init=.true.
19                  if (init) then
20                      init=.false.
21                      nullify(xb1,xb2)
22                  end if
23                  if (associated(xb1)) deallocate(xb1)
24                  if (associated(xb2)) deallocate(xb2)
25                  dx=(x2-x1)/n
26  ! x=x1+dx*arth(0,1,n+1)
27                  do i = 0, n
28                    x(i) = x1 + dx*i
29                  end do
30                  do i=0,n
31                      f(i)=func(x(i))
32                  end do
33                  mask=f(1:n)*f(0:n-1) <= 0.0
34                  nb=count(mask)
35                  allocate(xb1(nb),xb2(nb))
36                  xb1(1:nb)=pack(x(0:n-1),mask)
37                  xb2(1:nb)=pack(x(1:n),mask)
38                  END SUBROUTINE zbrak
```

А.6 Изходен код на процедурата zbrent

```
1           FUNCTION zbrent(func,x1,x2,tol)
2                   IMPLICIT NONE
3                   REAL, INTENT(IN) :: x1,x2,tol
```

```
4               REAL :: zbrent
5               INTERFACE
6                       FUNCTION func(x)
7                       IMPLICIT NONE
8                       REAL, INTENT(IN) :: x
9                       REAL :: func
10                      END FUNCTION func
11              END INTERFACE
12              INTEGER, PARAMETER :: ITMAX=100
13              REAL, PARAMETER :: EPS=epsilon(x1)
14              INTEGER :: iter
15              REAL :: a,b,c,d,e,fa,fb,fc,p,q,r,s,tol1,xm
16              a=x1
17              b=x2
18              fa=func(a)
19              fb=func(b)
20              if ((fa > 0.0 .and. fb > 0.0) .or. (fa < 0.0 .and. fb < 0.0))
                    then
21                      write(0,*)'root must be bracketed for zbrent'
22                      STOP
23              end if
24              c=b
25              fc=fb
26              do iter=1,ITMAX
27                      if ((fb > 0.0 .and. fc > 0.0) .or. (fb < 0.0 .and. fc <
                            0.0)) then
28                              c=a
29                              fc=fa
30                              d=b-a
31                              e=d
32                      end if
33                      if (abs(fc) < abs(fb)) then
34                              a=b
35                              b=c
36                              c=a
37                              fa=fb
38                              fb=fc
39                              fc=fa
40                      end if
41                      tol1=2.0*EPS*abs(b)+0.5*tol
42                      xm=0.5*(c-b)
43                      if (abs(xm) <= tol1 .or. fb == 0.0) then
44                              zbrent=b
45                              RETURN
46                      end if
47                      if (abs(e) >= tol1 .and. abs(fa) > abs(fb)) then
48                              s=fb/fa
49                              if (a == c) then
50                                      p=2.0*xm*s
51                                      q=1.0-s
52                              else
53                                      q=fa/fc
54                                      r=fb/fc
```

```
55                                      p=s*(2.0*xm*q*(q-r)-(b-a)*(r-1.0))
56                                      q=(q-1.0)*(r-1.0)*(s-1.0)
57                                  end if
58                                  if (p > 0.0) q=-q
59                                  p=abs(p)
60                                  if (2.0*p < min(3.0*xm*q-abs(tol1*q),abs(e*q
                                        ))) then
61                                      e=d
62                                      d=p/q
63                                  else
64                                      d=xm
65                                      e=d
66                                  end if
67                              else
68                                  d=xm
69                                  e=d
70                              end if
71                              a=b
72                              fa=fb
73                              b=b+merge(d,sign(tol1,xm), abs(d) > tol1 )
74                              fb=func(b)
75                      end do
76                      write(0,*)'zbrent:␣exceeded␣maximum␣iterations'
77                      zbrent=b
78                      END FUNCTION zbrent
```

A.7 Решение на задачата решаване на уравнението на Поасон чрез бързи Фурие трансформации FFT секция 10.5

```
1  module functions
2  implicit none
3  contains
4      function f(x)
5          double precision, intent(in) :: x
6          double precision :: f
7
8          if (x .gt. 0.d0) then
9              f = -x * (x + 3) * exp(x)
10         else
11             f = abs(x) * (abs(x) + 3) * exp(abs(x))
12         end if
13     end function f
14 end module functions
15
16 program poisson
17 use functions
18 implicit none
19       include "fftw3.f"
20
21       integer :: n
```

```
22        parameter (n=128)
23
24        integer*8 plan, iplan
25
26        double precision, dimension(N) :: uReal, uFurier
27        double precision, dimension(N) :: fReal, fFurier
28        double precision, dimension(N) :: sReal, sFurier
29        double precision :: pi, x, freq, delta, L, xmin, xmax
30        double precision :: timeDomainSum, freqDomainSum
31
32        integer :: i
33
34        pi = 3.141592653589793d0
35        xmin = 0.d0
36        xmax = 1.d0
37        L = xmax - xmin
38        delta = L / (n+1)
39
40        call DFFTW_PLAN_R2R_1D(plan, n, fReal, fFurier,
              FFTW_RODFT00, FFTW_ESTIMATE)
41        call DFFTW_PLAN_R2R_1D(iplan, n, uFurier, uReal,
              FFTW_RODFT00, FFTW_ESTIMATE)
42
43        do i=1,n
44           x = xmin + i * delta
45           fReal(i) = f(x)
46           sReal(i) = x * (x-1) * exp(x)
47        enddo
48
49      call dfftw_execute(plan)
50        call DFFTW_PLAN_R2R_1D(plan, n, sReal, sFurier,
              FFTW_RODFT00, FFTW_ESTIMATE)
51        call dfftw_execute(plan)
52
53      do i = 1, n
54            freq = real(i,8)
55            uFurier(i) = - 0.5d0 * fFurier(i) * delta ** 2/ (1.d0 - cos(pi*freq/(n
                  +1)))
56        end do
57
58        call dfftw_execute(iplan)
59        uReal(:) = uReal(:) / (2*(n+1))
60
61        do i = 1, n
62            x = xmin + i * delta
63            freq = i
64                    write(*,'(9F15.5)')x, freq, uReal(i), uFurier(i),&
65     & fReal(i), fFurier(i), sReal(i), sFurier(i),&
66     & sFurier(i)/fFurier(i)
67        end do
68
69        write(0,*) 'Parsewal sum time domain: ',sum(abs( uReal)**2)
70        write(0,*) 'Parsewal sum freq domain: ',sum(abs(uFurier)**2)/(2*(n
              +1))
```

```
71
72          call dfftw_destroy_plan(plan)
73          call dfftw_destroy_plan(iplan)
74
75          end
```

Библиография

[1] M. Abramowitz. *Handbook of mathematical functions : with formulas, graphs, and mathematical tables*. Dover Publications, New York, 1970.

[2] G. Fondation. Gsl online documentation, 2015.

[3] E. Forest and R. D. Ruth. Fourth-order symplectic integration. *Physica D Nonlinear Phenomena*, 43:105–117, May 1990.

[4] B. Fornberg. Generation of Finite Difference Formulas on Arbitrarily Spaced Grids. *Mathematics of Computation*, 51(184):699–706, 1988.

[5] S. Koonin and D. Meredith. *Computational Physics: Fortran Version*. Addison-Wesley, 1998.

[6] R. E. Odeh and J. O. Evans. Algorithm as 70: The percentage points of the normal distribution. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 23(1):96–97, 1974.

[7] L. Onsager. Crystal statistics. i. a two-dimensional model with an order-disorder transition. *Phys. Rev.*, 65:117–149, Feb 1944.

[8] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 3 edition, 2007.

[9] L. Rezzolla. Numerical methods for the solution of partial differential equations, 2011.

[10] R. D. Ruth. A canonical integration technique. *IEEE Transactions on Nuclear Science*, 30(4):2669–2671, Aug. 1983.

[11] S. Sinayoko. Eigenvalue problems i: Introduction and jacobi method, 2017.

[12] J. Thijssen. *Computational Physics*. Cambridge University Press, 1999.

[13] Wikipedia. Jacobi eigenvalue algorithm, 2017.