Università degli Studi di Roma
"LA SAPIENZA"

FACOLTÀ DI SCIENZE MATEMATICHE E FISICHE NATURALI

LAUREA TRIENNALE IN TECNOLOGIE INFORMATICHE

RELAZIONE DI TIROCINIO

"Un automa cellulare per la simulazione dei processi di diffusione"

RESPONSABILE INTERNO
PROF. Pietro Cenciarelli

STUDENTE
Danilo Abbasciano

anno accademico 2004/2005

Se esistono due soluzioni
per lo stesso problema
la migliore è quella più semplice.

IL RASOIO DI OCCAM

È difficile in poche righe ricordare tutte le persone che, a vario titolo, hanno contribuito in maniera più o meno diretta e più o meno consapevole a rendere "migliori" questi anni accademici. È stata anche la loro presenza e i loro consigli che mi hanno permesso di raggiungere questo traguardo importante.

Dal punto di vista del lavoro desidero ringraziare tutti coloro che con la loro pazienza e disponibilità mi hanno seguito durante il tirocinio. Un ringraziamento particolare va al Dr. Pietro Cenciarelli per il suo appoggio professionale e per avermi offerto l' opportunità di lavorare su un argomento decisamente stimolante.

Mi piacerebbe poi fare la lista di tutti gli amici "universitari" e non, ma evito per paura di dimenticarne qualcuno; comunque penso che chi mi conosce sa benissimo che non mi dimenticherò mai di loro.

Desidero inoltre ringraziare Claudia per essermi stata vicino quel tanto che basta per spronarmi a percorrere la mia strada aiutandomi a crescere con il suo immancabile buon senso, per il suo affetto, per aver diviso con me momenti di ansia e delusione, ma anche tante situazioni divertenti e gratificanti.

Infine grazie di cuore a mio fratello per i suoi consigli piuttosto cinici e pragmatici; ai miei genitori per avermi permesso, concretamente, di arrivare fino a quì, per la loro fiducia, il loro affetto e per il loro strano modo di sostenermi moralmente.

Indice

Capitolo 1

Introduzione

Lo spostamento di masse di fluidi è uno dei processi fisici più importanti, esso riguarda, infatti, il movimento di gas e di liquidi. Per entrambi, l' andameto è descritto da alcune equazioni, con il solo ma importante cambiameto dei valori dei parametri si riesce a descrivere il modo in cui le sostanze si muovono nel fluido. Il movimento di sostanze può avvenire in tre differeti modi: *diffusione*, *trascinamento* e *dispersione*. In questo lavoro verrà studiato il movimento diffusivo delle sostanze.

Lo sposatamento della materia dovuto al movimento randomico delle molecole è chiamato *processo di diffusione*. La diffusione tende a minimizzare la differenza di concentrazione di una sostenza tra due regioni; possiamo osservare un tipico esempio di diffusione facendo cadere della tintura in un recipiente con dell' acqua stagnante: dopo poco tempo la macchia si spande e l' intensità del suo colore decresce lentamente fino a quando tutto il recipiente assume una colorazione uniforme. Ciò che si osserva, dunque, è che la sostanza si muove da una regione ad alta concentrazione ad una a bassa concentrazione senza influire, se il fluido è immobile, sul baricentro della macchia.

Il fenomeno della diffusione è applicato in molti contesti differenti. Esso infatti si presenta in chimica nella composizione di soluti, nell' edilizia per studiare la corrosione del cemento, nell' elettronica per conoscere il movimento dei neuroni, nella medicina per studiare i fenomeni legati all' assorbimento di farmaci e, ovviamente, trova applicazione nell' ambito in cui effettivamente è stata formulata la legge della diffusione: la simulazione numerica di problemi elettrochimici.

L' equazione che descrive il processo di diffusione è stata formulata per la prima volta da *Fick* []. Tale equazione è posta nella forma differenziale, per questo, la sua applicazione a problemi pratici presenta alcune difficoltà. Dato che il calcolo delle soluzioni non è banale, il loro studio, per la maggior parte dei casi, è ristretto a zone molto piccole ed a situazioni semplici. In altre parole, la loro forma differenziale comporta grosse limitazioni.

La soluzione matematica della forma differenziale al problema della diffuzione, la cui applicazione non è idonea per i modelli sperimentali, acquista praticità grazie a metodi di *analisi numerica*. L' applicazione di questi metodi alla legge di Fick è stato il primo passo effettuato per sviluppare un programma di simulazione basato sul concetto di *automa cellulare* [1].

Un automa cellulare è un sistema dinamico in cui spazio, tempo e stati del sistema sono discretizzati. Gli automi cellulari fecero la loro prima comparsa nello studio di fenomeni biologici e meccanismi di funzionamento e autoriproduzione di esseri viventi. Ogni elemento dell'automa in una griglia spaziale regolare è detto cella e può assumere uno degli stati finiti. Gli stati delle celle variano secondo una regola locale, cioè lo stato di una cella ad un dato istante di tempo dipende dallo stato della cella stessa e dagli stati delle celle vicine all' istante precedente. Gli stati di tutte le celle sono aggiornati contemporaneamente in maniera sincrona. L' insieme degli stati delle celle compongono lo stato dell'automa.

Quindi lo stato globale dell'automa evolve in passi temporali discreti. Secondo questo modello un sistema viene rappresentato come composto da tante semplici parti ed ognuna di queste parti per evolvere ha una propria regola interna ed interagisce solo con le parti ad essa adiacenti.

Il sumulatore è basato sull' applicazione di due metodi matematici differenti: *esplicicto* ed *implicito*. Applicado il primo metodo il programma si comporta come un automa cellulare. Lavorando con il metodo implicito viene stravolto il concetto di automa cellulare, in quanto, lo stato di una cella non dipende univocamente dallo stato delle celle adiacenti all' istante precedente $(T - \delta T)$ ma anche dalle celle al tempo stesso (T).

Per far evolvere l' automa con il metodo implicito è necessario risolvere le equazioni di tutte le celle contemporaneamente. Per ogni passo temporale dovrà essere risolto un sistema lineare la cui soluzione corrisponde allo stato glogale dell' automa nell' istante successivo.

Capitolo 2

Aspetti matematici

Sommario

Viene di seguito riportato lo studio matematico utile per la realizzazione e la comprensione dell' automa cellulare. Verranno illustrate le leggi di Fick e tutti i passaggi che permettono la loro discretizzazione, utilizzando metodi di analisi numerica per trovare le soluzioni dell' equazione differenziale di Fick. Vedremo il metodo esplicito e quello implicito, quest' ultimo permette di trasformare l' equazione in un sistema lineare a più incognite che verrà risolto applicando il metodo di Gauss.

2.1 Le leggi di Fick

La seconda legge di Fick descrive il processo di diffusione di un fluido immerso in un ambiente. L' equazione della diffusione è

$$\frac{\partial C}{\partial t} = D\frac{\partial^2 C}{\partial x^2}. \tag{2.1}$$

Essa descrive la variazione della concentrazione C nel tempo. La variazione di concentrazione è funzione del tempo e dello spazio; D è il coefficiente di diffusione, dipende strettamente dall' ambiente in cui è immerso il fluido ed indica la rapidità di propagazione. La legge (2.1) riguarda lo spostamento nella sola direzione x, studieremo il caso di diffusione bidimensionale, l' equazione da utilizzare è perciò nella forma

$$\frac{\partial C}{\partial t} = D\left(\frac{\partial^2 C}{\partial x^2} + \frac{\partial^2 C}{\partial y^2}\right). \tag{2.2}$$

L' espressione (2.2) considera una regione piana rettangolare. In questo caso la funzione C dipenderà da tre variabili, x ed y per quanto riguarda lo spazio e t per il tempo.

2.2 Integrazione numerica

Per simulare la diffusione bisogna applicare all' equazione differenziale (2.2) un opportuno schema di integrazione numerica. In generale si hanno due tipologie di integrazione numerica:

- schema di integrazione esplicito;
- schema di integrazione implicito.

Per poter discretizzare la funzione di diffusione si divide sia l' area in esame, sia il tempo, in intervalli della stessa lunghezza. Siano δX e δY i passi di integrazione spaziale e δT il passo di integrazione temporale. Abbiamo costruito così una griglia che copre tutto lo spazio per ciascun passo temporale. Esprimiamo le coordinate in questo modo: $x = i\ \delta X$, $y = j\ \delta Y$ e $t = n\ \delta T$ con i, j, n numeri naturali. In quasto modo si può indicare il valore della concentrazione in ciascun punto della griglia come $C(i\delta X, j\delta Y, n\delta T)$ che per semplicità scriveremo $C_{i,j,n}$.

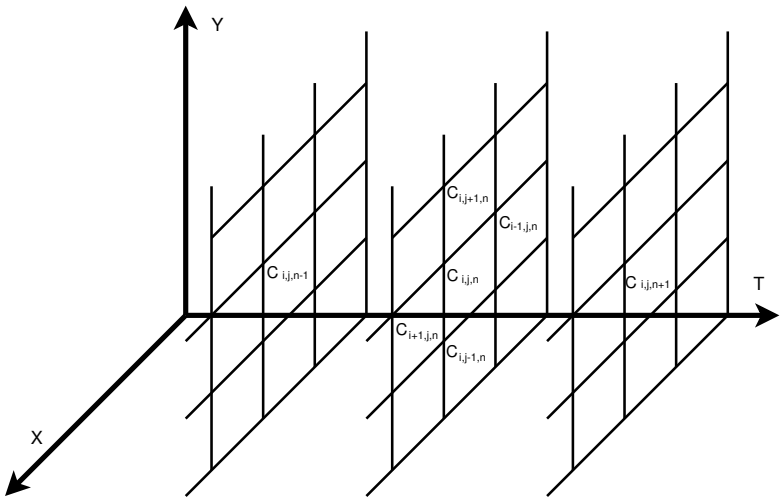


Figura 2.1: discretizzazione del dominio

2.2.1 Serie di Taylor

Sia $f(x)$ una funzione reale definita in un intervallo $(a, b) \in \mathbb{R}$ e sia $x_0 \in (a, b)$ si vuole costruire, se possibile, un polinomio $P_n(x)$ di grado $\leq n$ che abbia in x_0 il medesimo comportamento della $f(x)$, nel senso che, posto

$$P_n(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)^2 + \cdots + a_n(x - x_0)^n \qquad (2.3)$$

risulti $P_n^{(h)}(x_0) = f^{(h)}(x_0)$, con $h = 0, 1, ..., n$. Derivando il polinomio nel punto x_0 tutti gli addendi dopo il primo si annullano e risulta $h!\, a_h = f^{(h)}(x_0)$ per ogni $h \in \mathbb{N}$. Ne segue necessariamente che $a_h = f^{(h)}(x_0)/h!$, sostituendo la formula nella (2.3) si ottiene

$$f(x) = \sum_{h=0}^{n} \frac{f^{(h)}(x_0)}{h!}(x - x_0)^h + R_n(x). \tag{2.4}$$

La codizione necessaria e sufficiete affinché aumentando n la formula (2.4) approssimi meglio la funzione $f(x)$ è che $\lim_{n\to\infty} R_n(x) = 0$. Si ottiene così la cosiddetta *serie di Taylor* [3]

$$f(x) = \sum_{h=0}^{\infty} \frac{f^{(h)}(x_0)}{h!}(x - x_0)^h. \tag{2.5}$$

2.2.2 Metodo esplicito

Si considera la regione in esame coperta da una griglia tridimensionale, Figura 2.1, si può usare la serie di Taylor (2.5) nella direzione T ponendo X ed Y costanti:

$$C_{i,j,n+1} = C_{i,j,n} + \delta T \left(\frac{\partial C}{\partial T}\right)_{i,j,n} + \frac{1}{2}(\delta T)^2 \left(\frac{\partial^2 C}{\partial T^2}\right)_{i,j,n} + \cdots, \tag{2.6}$$

dalla (2.6) ne consegue che

$$\left(\frac{\partial C}{\partial T}\right)_{i,j,n} = \frac{C_{i,j,n+1} - C_{i,j,n}}{\delta T} + O(\delta T). \tag{2.7}$$

Il termine $O(\delta T)$ significa che i termini omessi sono dell' ordine di δT. Similmente si applica la serie di Taylor (2.5) nella direzione X considerando Y e T costanti

$$C_{i+1,j,n} = C_{i,j,n} + \delta X \left(\frac{\partial C}{\partial X}\right)_{i,j,n} + \frac{1}{2}(\delta X)^2 \left(\frac{\partial^2 C}{\partial X^2}\right)_{i,j,n} + \cdots; \tag{2.8}$$

$$C_{i-1,j,n} = C_{i,j,n} - \delta X \left(\frac{\partial C}{\partial X}\right)_{i,j,n} + \frac{1}{2}(\delta X)^2 \left(\frac{\partial^2 C}{\partial X^2}\right)_{i,j,n} - \cdots; \tag{2.9}$$

sommando la (2.8) e la 2.9 ricaviamo

$$\left(\frac{\partial^2 C}{\partial X^2}\right)_{i,j,n} = \frac{C_{i+1,j,n} - 2C_{i,j,n} + C_{i-1,j,n}}{(\delta X)^2} + O(\delta X)^2. \tag{2.10}$$

Analogamente, integrando lungo l' asse Y si ottiene

$$\left(\frac{\partial^2 C}{\partial Y^2}\right)_{i,j,n} = \frac{C_{i,j+1,n} - 2C_{i,j,n} + C_{i,j-1,n}}{(\delta Y)^2} + O(\delta Y)^2. \tag{2.11}$$

Sostituendo la (2.7), (2.10) e (2.11) nell' equazione della diffusione (2.2) si ottiene la legge che ci permette di calcolare la $C_{i,j,n+1}$, cioè la concentrazione al tempo $\delta T(n+1)$. Per fare ciò avremo necessariamente bisogno dei valori della concentrazione di partenza (tempo $T=0$) e dei valori al contorno. L' equazione ottenuta con lo schema di integrazione esplicito delle differenze finite è la seguente

$$C_{i,j,n+1} = C_{i,j,n} + \frac{\delta T}{(\delta X)^2} D(C_{i+1,j,n} - 2C_{i,j,n} + C_{i-1,j,n}) + \frac{\delta T}{(\delta Y)^2} D(C_{i,j+1,n} - 2C_{i,j,n} + C_{i,j-1,n}). \quad (2.12)$$

Questo metodo è computazionalmente molto performante di contro la condizione di stabilità è molto severa, impone l' uso di δT molto piccoli. La restrizione di stabilità si trova ponendo il coefficiente di $C_{i,j,n}$ maggiore o uguale a zero:

$$1 - 2D\left\{\frac{\delta T}{(\delta X)^2} + \frac{\delta T}{(\delta Y)^2}\right\} \geq 0;$$
$$D\left\{\frac{1}{(\delta X)^2} + \frac{1}{(\delta Y)^2}\right\} \delta T \leq \frac{1}{2}. \quad (2.13)$$

2.2.3 Metodo implicito di Crank-Nicolson

Un metodo alternativo, molto usato, fu proposto da Crank e Nicolson nel 1947. Questo metodo consiste nel sostituire $\partial^2 C/\partial X^2$ con la media delle differenze finite (2.10) al tempo n-esimo e (n+1)-esimo,

$$\left(\frac{\partial^2 C}{\partial X^2}\right)_{i,j,n} = \frac{1}{2}\left\{\frac{C_{i+1,j,n} - 2C_{i,j,n} + C_{i-1,j,n}}{(\delta X)^2} + \frac{C_{i+1,j,n+1} - 2C_{i,j,n+1} + C_{i-1,j,n+1}}{(\delta X)^2}\right\}. \quad (2.14)$$

Nello stesso modo si ricava la media per quanto riguarda la variabile Y

$$\left(\frac{\partial^2 C}{\partial Y^2}\right)_{i,j,n} = \frac{1}{2}\left\{\frac{C_{i+1,j,n} - 2C_{i,j,n} + C_{i-1,j,n}}{(\delta Y)^2} + \frac{C_{i+1,j,n+1} - 2C_{i,j,n+1} + C_{i-1,j,n+1}}{(\delta Y)^2}\right\}. \quad (2.15)$$

Combinando la legge di Fick (2.2) e la (2.7) inserendo la media delle differenze finite per le variabili X e Y:

$$\frac{C_{i,j,n+1} - C_{i,j,n}}{\delta T} = \frac{1}{2} D \left\{ \left(\frac{\partial^2 C}{\partial X^2} + \frac{\partial^2 C}{\partial Y^2} \right)_{i,j,n} + \left(\frac{\partial^2 C}{\partial X^2} + \frac{\partial^2 C}{\partial Y^2} \right)_{i,j,n+1} \right\}. \quad (2.16)$$

Sostituendo nella (2.16) la (2.14) e la (2.15) si ottiene

$$\begin{aligned} &2C_{i,j,n+1} - 2C_{i,j,n} = \\ &\frac{\delta T\, D}{(\delta X)^2}(C_{i+1,j,n} - 2C_{i,j,n} + C_{i-1,j,n} + C_{i+1,j,n+1} - 2C_{i,j,n+1} + C_{i-1,j,n+1}) + \\ &\frac{\delta T\, D}{(\delta Y)^2}(C_{i,j+1,n} - 2C_{i,j,n} + C_{i,j-1,n} + C_{i,j+1,n+1} - 2C_{i,j,n+1} + C_{i,j-1,n+1}); \end{aligned} \quad (2.17)$$

ponendo $\lambda_x = \delta T D/(\delta X)^2$ e $\lambda_y = \delta T D/(\delta Y)^2$ e separando la concentrazione dell' istante $n+1$ dall' istante n

$$\begin{aligned} -\lambda_x C_{i+1,j,n+1} + 2(1 + \lambda_x + \lambda_y) C_{i,j,n+1} - \lambda_x C_{i-1,j,n+1} - \\ \lambda_y C_{i,j+1,n+1} - \lambda_y C_{i,j-1,n+1} = \\ \lambda_x C_{i+1,j,n} + 2(1 - \lambda_x - \lambda_y) C_{i,j,n} + \lambda_x C_{i-1,j,n} + \\ \lambda_y C_{i,j+1,n} + \lambda_y C_{i,j-1,n}. \end{aligned} \quad (2.18)$$

Se N è il numero di tasselli spaziali lungo l' asse X ed M il numero di tasselli spaziali lungo l' asse Y allora $i \in [0, N-1]$ e $j \in [0, M-1]$. Il metodo di Crank–Nicolson richiede la soluzione simultanea di $(N-1)(M-1)$ equazioni algebriche per ogni passo temporale δT; questo perché cinque valori incogniti della concentrazione C compaiono in ciascuna equazione. Per giungere alla soluzione non si usano metodi di eliminazione diretta, ma il sistema viene risolto iterativamente. Questo metodo, in cui le soluzioni sono ottenute dalla risoluzione simultanea di equazioni è chiamato "metodo implicito". Si può procedere con il metodo di Crank–Nicolson usando sia intervalli temporali ampi che brevi.

2.2.4 Condizioni iniziali e al contorno

Per risolvere le equazioni è necessario definire le condizioni al contorno e le condizioni iniziali.

Le condizioni iniziali sono le concentrazioni al tempo zero, cioè $C(i, j, n = 0)$ $\forall i \in [0, N-1], j \in [0, M-1]$; le condizioni al contorno sono le concentrazioni al bordo dell' area in esame ($i = 0, i = N-1, j = 0, j = M-1$). Dato che comunque non è possibile conoscere cosa avviene al di fuori dell' area in

esame un calcolo della diffusione effettuato in una zona adiacente al margine risulterà comunque non preciso, possiamo limitarci a considerare le zone al contorno come se non fossero affatto influenzate dall' esterno. È importante prendere in considerazione aree molto più vaste della zona di studio per evitare che i risultati siano influenzati dalla vicinanza con il bordo.

2.3 Sistemi lieari

Di seguito vengono riportati i passaggi che permettono di arrivare ad un sistema lieare con n equazioni ed n incognite con $n = (N-1)(M-1)$ partendo dall' equazione di Crank–Nicolson.

Un sistema lineare può essere espresso come un' equazione sottoforma matricale. Si considera il seguente sistema lineare in n incognite $x_1, x_2, \cdots, x_n$

$$\begin{cases} a_{1,1}x_1 & + & a_{1,2}x_2 & +\cdots+ & a_{1,n}x_n & = & b_1 \\ a_{2,1}x_1 & + & a_{2,2}x_2 & +\cdots+ & a_{2,n}x_n & = & b_2 \\ \vdots & & \vdots & & \vdots & & \vdots \\ a_{n,1}x_1 & + & a_{n,2}x_2 & +\cdots+ & a_{n,n}x_n & = & b_n \end{cases}$$

La soluzione del sistema è un insieme di valori $x_1, x_2, \cdots, x_n$ che soddisfano simultaneamente tutte le equazioni. Si può scrivere il sistema con l' equazione

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

o più semplicemente, ponendo $A = (a_{i,j})$, $x = (x_j)$ e $b = (b_j)$, come

$$Ax = b. \tag{2.19}$$

Senza la perdita di informazioni possiamo trasformare la funzione a tre incognite C in una a due incognite. La trasformazione parte dal presupposto che si può utilizzare un solo indice k in luogo di i e j, per identificare lo spazio. k si ottiene in questo modo: $k = j + iN$. L' equazione della diffusione integrata (2.18) ora assume la forma

$$\begin{aligned} -\lambda_x C_{k+1,n+1} + 2(1+\lambda_x+\lambda_y)C_{k,n+1} - \lambda_x C_{k-1,n+1} - \\ \lambda_y C_{k+N,n+1} - \lambda_y C_{k-N,n+1} = \\ \lambda_x C_{k+1,n} + 2(1-\lambda_x-\lambda_y)C_{k,n} + +\lambda_x C_{k-1,n} + \\ \lambda_y C_{k+N,n} + \lambda_y C_{k-N,n} \end{aligned} \tag{2.20}$$

$\forall k \in [0, NM-1]$.

Per poter scrivere equazione (2.20) nella forma del sistema (2.19) introduciamo la matrice B.

La matrice B è una matrice sparsa. I suoi elementi sono per la maggior parte uguali a zero, ed ha al più cinque elementi diversi da zero per ogni riga. Per comprendere la sua struttura dividiamo B in sottomatici. Siano B_1 e B_2 due matrici di dimensioni $(N-1) \times (M-1)$

$$B_1 = \begin{bmatrix} -(\lambda_x + \lambda_y) & \lambda_x & 0 & \cdots & \\ \lambda_x & -(\lambda_x + \lambda_y) & \lambda_x & & \\ 0 & \lambda_x & -(\lambda_x + \lambda_y) & \ddots & \\ & & \ddots & \ddots & \lambda_x \\ & & 0 & \lambda_x & -(\lambda_x + \lambda_y) \end{bmatrix}$$

e B_2 ha la forma $B_2 = \lambda_y Id$. La matrice B è

$$B = \begin{bmatrix} B_1 & B_2 & \mathbf{0} & \cdots & & \\ B_2 & B_1 & B_2 & \mathbf{0} & \cdots & \\ \mathbf{0} & B_2 & B_1 & B_2 & \mathbf{0} & \\ & \mathbf{0} & B_2 & B_1 & \ddots & \\ & & & \ddots & \ddots & B_2 \\ & & & \mathbf{0} & B_2 & B_1 \end{bmatrix}$$

le sue dimensioni sono dell' ordine di $(NM \times MN)$. È stato utilizzato il simbolo $\mathbf{0}$ per indicare una matrice, di dimensioni opportune, con tutti gli elemeti uguali a zero.

Ponendo $A = 2Id + (-B)$, il vettore delle incognite x corrisponderà al vettore della concentrazione C_{n+1} e sostituendo a b la seguente espressione $b = (2Id + B)C_n$, il sistema $Ax = b$ diviene

$$(2Id + (-B))\, C_{n+1} = (2Id + B)\, C_n; \tag{2.21}$$

Id è chiamata matrice identità, i suoi elementi sulla diagonale principale sono uguali ad uno, gli altri sono zero.

Ogni equazione del sistema identifica una precisa zona dell' area in esame, per esempio la k-esima equazione si riferisce al punto di coordinate $(\lfloor k/N \rfloor, k \ mod \ N)$. Un interessante interpretazione della legge di Fick consiste nel considerare il coefficiente di diffusione D non come una costante ma come una funzione. In questo modo possiamo distinguere zone, della stessa area in esame, con un coefficiente di diffusione differente. Sia D una matrice diagonale delle stesse dimensioni di B, i suoi elementi nella diagonale esprimono i coefficienti di diffusione in ogni punto della regione. Si può osservare che i coefficienti di diffusione restano invariati nel tempo, quindi possono essere portati fuori dalla matrice $(2Id + (-B))$. Siano $\lambda'_x = \delta T/(\delta X)^2$ e $\lambda'_y = \delta T/(\delta Y)^2$ i coefficienti λ_x e λ_y senza il coefficiente di diffusione. Sostituendo nella matrice B_1 i coefficienti λ'_x e λ'_y al posto di λ_x e λ_y, la matrice

così ottenuta è la B'_1. Ponendo $B'_2 = \lambda'_y Id$. Sia B' la matrice ottenuta da B sostituendo la matrice B_1 con B'_1 e B_2 con B'_2, ora l'equazione (2.21) diviene

$$D\left\{2\,Id + (-B')\right\} C_{n+1} = D\left(2\,Id + B'\right) C_n. \tag{2.22}$$

2.4 Metodo di eliminazione di Gauss

Si riporta di seguito la risoluzione di un sistema di equazioni lineari usando il metodo di fattorizzazione LU, chiamato anche *eliminazione di Gauss* [5]. Si applicherà questo metodo per la soluzione dell' equazione (2.22).

Un approccio per risolvere il sistema (2.19) di n equazioni in n incognite consiste di calcolare A^{-1} e moltiplicare l' equazione per A^{-1} ottenendo $A^{-1}Ax = A^{-1}b$, in questo modo $x = A^{-1}b$. Questo metodo ha problemi di instabilità numerica, gli errori di arrotondamento tendono ad accumularsi esageratamente quando viene usata la rappresentazione in virgola mobile dei numeri al posto dei numeri reali ideali. Fortunatamente la fattorizzazione LU non presenta questi problemi e risulta molto vantaggiosa in termini computazionali.

L' idea che sta dietro la fattorizzazione LU consiste nel trovare due matrici $n \times n$ chiamate L ed U tali che $LU = A$, dove L è una matrice triangolare inferiore unitaria (che ha tutti 1 sulla diagonale principale) e U è una matrice triangolare superiore. Una volta trovata la fattorizzazione LU per la matrice A si può risolvere il sistema (2.19) $Ax = b$ risolvendo solo sistemi lineari triangolari. Sia

$$Ux = y, \tag{2.23}$$

l' eq (2.19) sapendo che $LU = A$ e poichè vale la (2.23) diviene

$$Ly = b. \tag{2.24}$$

Questo è un sistema triangolare inferiore che può essere risolto utilizzando il metodo di "sostituzione in avanti" che ha complessità $\Theta(n^2)$. Una volta risolto questo sistema si ottiene il vettore y, sostituendo il risultato ottenuto nell' equazione (2.23) e risolvendo un sistema triangolare superiore si ottiene x. Il vettore x sarà anche soluzione del sistema (2.19). Il sistema (2.23) viene risolto con il metodo di "sostituzione a ritroso" che ha compessità $\Theta(n^2)$.

2.4.1 Fattorizzazione LU

Il processo di fattorizzazione consiste nel trovare una scomposizione della matrice dei coefficienti A in due matrici L ed U in modo che sia $LU = A$. Per fare ciò si sottraggono i multipli della prima equazione del sistema dalle altre equazioni così che la prima variabile sia rimossa da quelle equazioni. Si sottraggono i multipli della seconda equazione dalla terza e dalle successive

così che la prima e la seconda variabile siano rimosse. Si itera il procedimento fino a quando la matrice non assume la forma triangolare superiore, si arriva così alla matrice U. La matrice L è composta da moltiplicatori che causano l' eliminazione delle variabili. L' algoritmo può essere facilmente descritto in modo induttivo su n. Se $n = 1$ si sceglie $L = 1$ ed $U = a_{1,1}$. Per $n > 1$ si taglia A in quattro parti

$$A = \left(\begin{array}{c|ccc} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ \hline a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{array}\right) = \begin{pmatrix} a_{1,1} & w^T \\ v & A' \end{pmatrix}$$

dove v è un vettore colonna e w^T è un vettore riga, entrambi di dimensione $(n-1)$. A' è una matrice $(n-1) \times (n-1)$. È facile verificare che A ammette la seguente scomposizione

$$A = \begin{pmatrix} a_{1,1} & w^T \\ v & A' \end{pmatrix} = \begin{pmatrix} 1 & \mathbf{0} \\ v/a_{1,1} & Id \end{pmatrix} \begin{pmatrix} a_{1,1} & w^T \\ \mathbf{0} & A' - vw^T/a_{1,1} \end{pmatrix}.$$

Si intende con $\mathbf{0}$ nella prima e nella seconda matrice, rispettivamente, vettori riga e colonna di dimensione $(n-1)$, che hanno tutti gli elementi uguali a zero.

Supponendo vero questo metodo per la matrice $A' - vw^T/a_{1,1}$ di dimensioni $(n-1) \times (n-1)$, si dimostra per A di dimensioni $n \times n$. Siano L' e U' le matrici ottenute dalla scomposizione della sottomatrice $A' - vw^T/a_{1,1}$, si ha

$$\begin{aligned} A &= \begin{pmatrix} 1 & \mathbf{0} \\ v/a_{1,1} & Id \end{pmatrix} \begin{pmatrix} a_{1,1} & w^T \\ \mathbf{0} & A' - vw^T/a_{1,1} \end{pmatrix} \\ &= \begin{pmatrix} 1 & \mathbf{0} \\ v/a_{1,1} & Id \end{pmatrix} \begin{pmatrix} a_{1,1} & w^T \\ \mathbf{0} & L'U' \end{pmatrix} \\ &= \begin{pmatrix} 1 & \mathbf{0} \\ v/a_{1,1} & L' \end{pmatrix} \begin{pmatrix} a_{1,1} & w^T \\ \mathbf{0} & U' \end{pmatrix} \\ &= LU. \end{aligned} \tag{2.25}$$

Poiché L' è tringolare inferiore unitaria, anche L lo è. Poiché U' è tringolare superiore, anche U lo è. Si ottiene così ottenuto una fattorizzazione LU per la matrice A.

Osserviamo che se l' elemento $a_{1,1}$ fosse uguale a zero questo metodo non funzionerebbe, non si può applicare, inoltre, se l'elemento $a'_{1,1}$ della matrice A' fosse zero. In questi casi è possibile applicare il procedimento dopo aver opportunamente permutato le righe di A. Fortunatamente si userà la fattorizzazione LU per una classe di matrici in cui non si verifica mai questo inconveniente.

Il codice della fattorizzazione LU è riportato nel cap 3.3.

2.4.2 Sostituzione in avanti e a ritroso

La *sostituzione in avanti* risolve il sistema (2.24) in tempo $\Theta(n^2)$. Si può scrivere l' eq. (2.24) come

$$\begin{array}{rcrcrcrcl} y_1 & & & & & & & = & b_1 \\ l_{2,1}y_1 & + & y_2 & & & & & = & b_2 \\ l_{3,1}y_1 & + & l_{3,2}y_2 & + & y_3 & & & = & b_3 \\ & & & & & & & \vdots & \\ l_{n,1}y_1 & + & l_{n,2}y_2 & + & l_{n,3}y_3 & + \cdots + & y_n & = & b_n \end{array} \tag{2.26}$$

y_1 si trova direttamente dalla prima equazione del sistema, una volta trovato y_1 si sostituisce nella seconda equazione ottenendo y_2. In generale, sostituendo $y_1, \ldots, y_{i-1}$ in avanti nella i-esima equazione si trova y_i.

$$y_i = b_i - \sum_{j=1}^{i-1} l_{i,j} y_j \tag{2.27}$$

La *sostituzione a ritroso* è analoga alla sostituzione in avanti, con lunica differenza che viene risolta prima l' n-esima equazione, poi si opera a ritroso fino alla prima equazione. Questo metodo serve per risolvere le equazioni di tipo (2.23). Riscrivendo il sistema come

$$\begin{aligned} u_{1,1}x_1 + u_{1,2}x_1 + \cdots + u_{1,n-1}x_n + u_{1,n}x_n &= y_1 \\ u_{2,2}x_1 + \cdots + u_{2,n-1}x_n + u_{2,n}x_n &= y_2 \\ &\vdots \\ u_{n,n}x_n &= y_n \end{aligned} \tag{2.28}$$

si possono calcolare $x_n, x_{n-1}, \cdots, x_1$ una dopo l' altra come segue:

$$\begin{aligned} x_n &= y_n/u_{n,n} \\ x_{n-1} &= (y_{n-1} - u_{n-1,n}x_n)/u_{n-1,n-1} \\ x_{n-2} &= (y_{n-2} - (u_{n-2,n-1}x_{n-1} + u_{n-2,n}x_n))/u_{n-2,n-2} \\ &\vdots \end{aligned} \tag{2.29}$$

la formula generale è la seguente:

$$x_i = \left(y_i - \sum_{j=i+1}^{n} u_{i,j} x_j \right) / u_{i,i}. \tag{2.30}$$

Entrambi i metodi vengono implementati nella funzione `LUsolve()` (vedere capitolo 20).

Capitolo 3

Il simulatore

Sommario

Nella prima parte del capitolo sono descritte le funzionalità implemetate dal simulatore. Poi nel manuale ne viene descritto l' uso, l' interfaccia con l' utente e tutte le opzioni ed i parametri richiesti dal programma. Infine verranno illustrate le tecniche impiegate per l' implementazione (algoritmi e strutture dati) di alcune parti del simulatore in riferimento allo studio matematico presentato.

3.1 Le fuzionalità del sistema

Fick è un tool per simulare la diffusione di fluidi, può operare sia su piccole che su vaste aree. Fick è in grado di elaborare una quatità molto grande di dati (nell' ordine di 10^7) mantenendo un buon grado di approssimazione (garantito dall' uso della fattorizzazione LU) e tempi di esecuzione ridotti. Uno dei suoi punti di forza è la facilità di utilizzo grazie alla sua semplice interfaccia a linea di comando.

Il simulatore è in grado di accettare le opzioni secondo lo stile dei programmi GNU–Linux. Lo standard de facto prevede la specifica delle opzioni attraverso delle lettere precedute dal simbolo '-'. Opzioni multiple possono essere specificate anche dopo un solo '-', ad esempio "-abc" è equivalente ad "-a -b -c". Se un opzione richiede un argomento, il valore dell' attributo viene posto subito dopo il carattere dell' opzione o può comparire come prossimo elemento di argv. Per esempio, "-d foo" e "-dfoo" rappresentano entrambi l' opzione 'd' con "foo" come argomento assumendo che 'd' richiede un argomento. Se l' opzione non richiede di specificare un argomento allora ciò che segue l' opzione verrà considerato come prossimo elemento di argv. Per esempio, se 'c' non richiede nessun argomento, "-cfoo" rappresenta l' opzione 'c' ed un argomento "foo". Se l' argomento da specificare ha come primo

carattere un '-' allora useremo la seguente sintassi "- - -d", in questo caso "-d" non verrà considerata un opzione ma un parametro. Le opzioni ed i parametri possono essere specificati in qualunque ordine (non viene rispetta quindi la tradizionale maniera di Unix,) la stringa "-a foo -b" considera "-a" e "-b" opzioni e "foo" come parametro.

In aggiunta alle tradizionali opzioni con un singolo carattere, Fick supporta le opzioni lunghe. Queste sono precedute da "- -". Per esempio per invcare la guida oltre a "-h", il programma accetta anche "- - help".

Come le opzioni brevi, anche le opzioni lunghe possono richiedere un parametro. Tale parametro può essere specificato ponendo il segno di uguale dopo il nome dell' opzione, in alternativa può essere specificato nell' argomento successivo. Per esempio: "- - output=foo" e "- - output foo" indicano entrambi l' opzione "output" con il parametro "foo".

Per poter eseguire il simulatore è necessario specificare, oltre al file immagine, tre parametri "- - dx", "- - dy" e "- - dt" che servono per settare rispettivamente δX, δY e δT. Se non vengono passati come opzioni il prgramma chiederà comunque all' utente di iserirli. I primi due servono per discretizzare lo spazio, indicano la distanza in numero di pixel tra un campionamento ed il succesivo. Il parametro δT influisce sulla rapidità della simulazione, più precisamente indica i passi temporali che intercorrono tra uno step ed il successivo della simulazione. A differenza dei primi due δT può assumere valori reali.

La memoria utilizzata dal simulatore è direttamente proporzionale al quadrato del numero dei campioni della concentrazione considerati. Se il simulatore non ha abbastanza memoria disponibile è necessario diminuire le dimensioni dell' immagine oppure aumentare i fattori δX e δY. L' aumento di quest' ultimi richiede l' utilizzo da parte del programma di un passo di integrazione più ampio comportando una diminuzione della precisione.

Se l' immagine della concentrazione e quella dei coefficieti di diffusione differiscono nelle dimensioni, la zona presa in considerazione sarà quella che avrà sia come altezza, sia come larghezza la più piccola (partendo dall' angolo in alto a sinistra) tra le due (Figura 3.1).

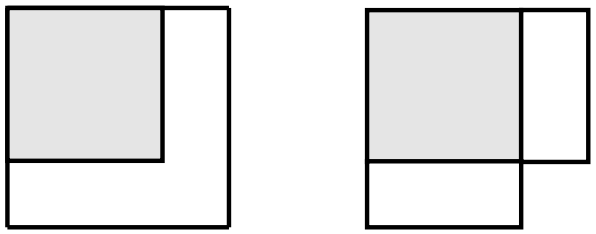


Figura 3.1: Immagini in input di dimensioni diverse

3.2 Manuale

NOME

fick − simulatore di diffusione

VERSIONE

0.2

SINTASSI

fick [*OPTIONS*] *ImageDistribition* [*ImageDiffusion*]

DESCRIZIONE

fick un tool per simulare l'espansione di un fluido in un ambiente. Il programma puo' prende in esame due file immagini, il primo si riferisce alla concentrazione di sostanza in ogni punto dell'area in esame, il secondo al coefficiente di diffusione. Questi valori variano da punto a punto nella zona in esame e corrispondono al colore blu per la concentrazione e al colore rosso per i coefficienti di diffusione. Se non viene specificato il file *ImageDiffusion* si assume che il coefficiente di diffusione e' costante in tutta l'area.

Questo tool sfrutta le equazioni di Fick discretizzate secondo metodi di analisi numerica.

I tipi di immagine supportati sono:

TGA: TrueVision Targa (.tga)

BMP: Windows Bitmap(.bmp)

PNM: Portable Anymap (.pnm)
.pbm = Portable BitMap (mono)
.pgm = Portable GreyMap (256 scala di grigi)
.ppm = Portable PixMap (colori)

XPM: X11 Pixmap (.xpm)

XCF: formato nativo di GIMP (.xcf) (XCF = eXperimental Computing Facility?) Questo formato tuttora in evoluzione, il caricamento di immagini di questo tipo potrebbe fallire. Sarebbe meglio se si carica il file con GIMP e si converte in un formato meglio supportato.

PCX: ZSoft IBM PC Paintbrush (.pcx)

GIF: CompuServe Graphics Interchange Format (.gif)

JPG: Joint Photographic Experts Group JFIF format (.jpg or .jpeg)

TIF: Formato Tagged Image File (.tif or .tiff)

LBM: Interleaved Bitmap (.lbm or .iff) DA : ILBM o PBM(packed bitmap) HAM6, HAM8 e 24bit che non sono supportati.

PNG: Portable Network Graphics (.png)

COMANDI

ESC q
Esce dal programma.

f Abilita e disabilita la modalit a tutto schermo.

\+ Incrementa Dt di uno.

- Se Dt maggiore di uno allora lo decrementa.

OPZIONI

Fick accetta solo opzioni a linea di comando che hanno questa sintassi:

-f --fullscreen
Abilita la vista a tutto schermo. E' possibile farlo anche mentre il programma sta girando, basta premere il taso "f" per abilitarla. Per togliere la modalit a tutto schermo premere nuovamente "f".

-e --explicit
Usa le leggi di Fick discretizzate utilizzando il metodo esplicito. Molto piu' performate del metodo implicito ma implica una restrizione di stabilita' piu' severa.

-x *arg* **--dx** *arg*
Setta il valore Dx necessario per la dicretizzazione dell'immagine. Corrisponde al numero di pixel.

-y *arg* **--dy** *arg*
Setta il valore Dy necessario per la dicretizzazione dell'immagine. Corrisponde al numero di pixel.

-t *arg* **--dt** *arg*
Setta il valore Dt. Corrisponde all'intervallo di tempo che intercorre tra uno stadio ed un altro della simulazione.

-s *n* **--steps** *n*
Setta il numero di passi che deve effettuare il programma dopodiche termina. Il programma pu comunque essere terminato chiudendo la finestra o premendo il taso escape.

-d *s* **--delay** *s*
millisecondi che intercorrono tra la visualizzazione di due immagini. Se per esempio vogliamo visualizzare le immagini con una frequenza di 20 immagini al secondo setteremo *s* a 50. Se non si specifica questa opzione il ritardo sar 250.

-W *n* **--Width** *n*
Specifica la larghezza della granularit dell'immagine nella finestra di output. Se viene usata insieme all'opzione **−o** allora anche l'immagiene salvata avr la granularit specificata. Se non si specifica questa opzione la larghezza sar uguale a Dy.

-H *n* **--Height** *n*
Specifica l'altezza della granularit dell'immagine nella finestra di output. Se viene usata insieme all'opzione **−o** allora anche l'immagiene salvata avr la granularit specificata. Se non si specifica questa opzione la larghezza sar uguale a Dx.

-o *file* **--output** *file*
Salva i risultati della simulazione in un file BMP di nome *file*. La concentrazione salvata si riferisce a quella ottenuta nell'ultima computazione del programma.

-v --version
Stampa la versione del programma.

-h --help
Stampa l'help del programma.

VALORE DI RITORNO

Il programma ritorna 0 se tutto andato a buon fine, 1 se si verificato un errore.

AUTORE

Fick e' stato scritto da Danilo Abbasciano.

REPORTING BUGS

Report bugs to <danilo.abbasciano@gmail.com>

3.3 L' architettura

Il simulatore, implementato completamente in *ANSI C* [6], è composto da quattro file: header.h, fick.c, function.c e 3d.c.

Il file header.h contiene tutte le inclusioni delle librerie, le dichiarazioni di costanti, le macro, le variabili globali e le dichiarazioni delle strutture dati. Le librerie usatilizzate sono: stdio.h, stdlib.h, string.h, SDL.h, SDL_image.h e getopt.h. SDL.h e SDL_image.h sono necessarie per la manipolazione delle immagini e la gestione degli eventi mentre getopt.h per la gestione dei parametri passati al programma.

In tutti i file .c del progamma viene effettuata l' inclusione di header.h, la direttiva al preprocessore #ifndef HEADER posta all' inizio del file header.h ci garantisce che il link viene effettuato una sola volta.

Nel file fick.c c'è il main del programma, il file 3d.c comprende tutte le funzioni che riguardano la vista in tre dimensioni della simulazione e nel file function.c è presente l' implementazione di tutte le altre funzioni.

Il main, a seconda del metodo di simulazione che si vuole adottare, implicito o esplicito, è in grado di variare i passi da eseguire, come si può vedere nelle figure 3.2 e 3.3.

Nella sezione 2.3 si è trattato di come sia possibile cambiare una funzione a tre incognite in una equivalente a due incognite senza la perdita di alcuna informazione. Questo cambiamento è stato possibile grazie al modo di riferirsi ad uno spazio bidimensionale utilizzando un solo indice. Questa trasformazione in termini di programmazione si traduce memorizzando una matrice bidimensionale in vettore (matrice unidimensionale). Di segiuto si ha la defiizione della struttura `Matrice`

```
typedef struct{
  int color; /* R, G or B */
  int r,c;   /* matrix dimenction */
  Uint8 *m;
}Matrice;
```

I valori della matrice vengono memorizzati nel vettore `m`, gli interi `r` e `c` mantengono le dimensioni della matrice, rispettivamente righe e colonne.

È riportata dettagliatamente l' implementazione di alcune fuzioni.

coefficient_matrix()

Per poter applicare il *metodo di Gauss* è necessario scrivere il sistema (2.22) nella forma $Ax = b$ dove x è il vettore di incognite che corrisponde alla concetrazione al tempo $n+1$ (C_{n+1}), b è il vettore dei coefficienti noti ed A è la matrice dei coefficieti. La funzione `coefficient_matrix()` riempie la matrice A, ponendola uguale a $\{2\,Id + (-B')\}D$. La (2.22) ora assumerà la forma $AC_{n+1} = \{2\,Id + B'\}D\,C_n$.

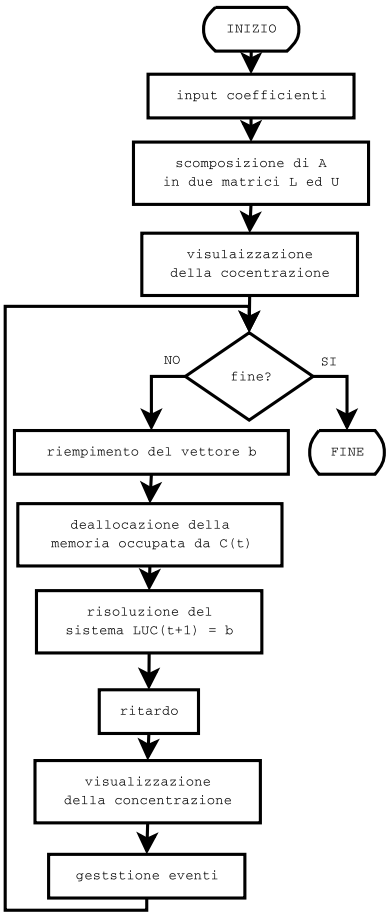


Figura 3.2: Metodo implicito

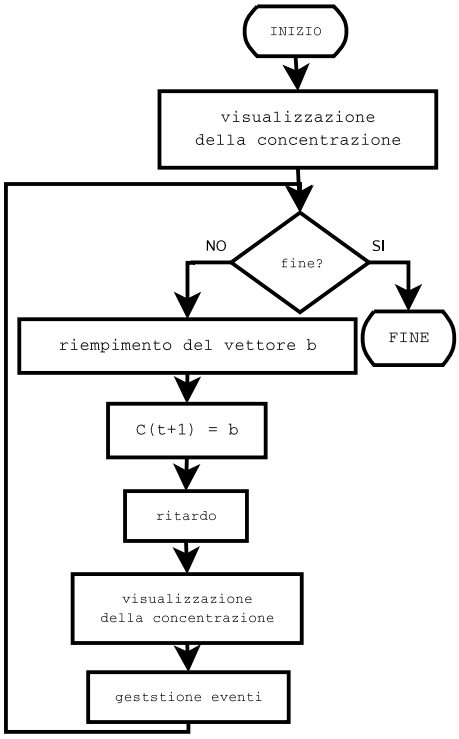


Figura 3.3: Metodo esplicito

Per risolvere $A = \{2\,Id + (-B')\}D$ il simulatore non esegue operazioni tra matrici (addizione, prodotto tra matrice e vettore, prodotto di una matrice per uno scalare e prodotto tra matrici), se avessi usato questo metodo, la sua complessità sarebbe $\Theta(n^3)$. Ho sfruttato il fatto che gli elementi della matrice A sono per la maggior parte nulli.

La funzione è la seguente:

```
float **coefficient_matrix(int Dx, int Dy, double Dt,
                           Matrice *D) {
  float **A;
  int k;
  int dim = D -> r * D -> c;

1   if ((A = (float **)malloc(dim * sizeof(float *))) == NULL)
2     exit(1);
```

```
3   for (k = 0; k < dim; k++)
4     if ((A[k] = (float *)calloc(dim, sizeof(float))) == NULL) {
5       printf(``the memory is below to run fick.\n'');
6       exit(1);
7     }

8   for (k = 0; k < dim; k++) {
9     A[k][k] = 2*(1 + Lx(k) + Ly(k));   /* 2(1 + Lx + Ly) */

10    if (k - 1 >= 0 && (k-1)%D->c != D->c - 1)
11      A[k][k-1] = -1 * Lx(k);            /* -Lx */

12    if (k + 1 < dim && (k+1)%D->c != 0)
13      A[k][k+1] = -1 * Lx(k);            /* -Lx */

14    if (k - D->c >= 0)
15      A[k][k - D->c] = -1 * Ly(k);       /* -Ly */

16    if (k + D->c < dim)
17      A[k][k+D->c] = -1 * Ly(k);         /* -Ly */
18  }
19  return A;
20}
```

Dopo la dichiarazione di alcune variabili locali, nelle linee 1–7 si riserva la memoria per la matrice dinamica A. Usando la fuzione `calloc()` la memoria viene automaticamete inizializzata a zero. Dato che la matrice ha al più cinque elementi diversi da zero per ogni riga, il ciclo (linee 8–18) ha il compito di scrivere tali valori. Le istruzioni `if` all' interno del ciclo garantiscono che l' area in esame non venga considerata cilindrica, cioè eliminano la possibilità che una zona situata al margine sia influezata dalla concentrazione della zona del margine opposto.

La funzione ha complessità $\Theta(n)$ dove n è il numero di righe della matrice.

calcolate_b()

La funzione `calcolate_b()` restituisce un vettore; viene chiamata sia se il simulatore lavora in modalità esplicita, sia se lavora in modalità implicita. In modalità esplicita la funzione trova direttamente C_{n+1}. Per capire il suo funzionamento si parte dall' equazione (2.12); sommando i termini comuni ed effettuando le sostitzioni $\lambda_x = (\delta T\ D)/(\delta X)^2$ e $\lambda_y = (\delta T\ D)/(\delta Y)^2$ si

ottiene

$$C_{i,j,n+1} = \lambda_x C_{i+1,j,n} + (1 - 2\lambda_x - 2\lambda_y) C_{i,j,n} + \lambda_x C_{i-1,j,n} + \lambda_y C_{i,j+1,n} + \lambda_y C_{i,j-1,n}. \quad (3.1)$$

Confrontando il lato destro della (3.1) e il lato destro dell' equazione (2.18) del metodo implicito si ha che sono uguali a meno di $C_{i,j,n}$. Detto questo risulta semplice unire in una sola funzione il calcolo del lato destro delle due modalità operative. La funzione, lavorando in modalità implicita, restituisce un vettore b dato da $\{2\,Id + B'\}D\,C_n$. Questo sarà il vettore di termini noti del sistema $A\,C_{n+1} = b$.

```
float *calcolate_b(int Dx, int Dy, double Dt,
                   Matrice *D, Matrice *C) {
  int k;
  float *b;

1   if ((b = (float *)malloc(D->r*D->c * sizeof(float))) == NULL)
2     exit(1);

3   for (k = 0; k < (D -> r * D -> c); k++) {

4     b[k] = ((implicit+1) - 2*Lx(k) - 2*Ly(k))* C->m[k];
                              /* (2-2Lx-2Ly)C(k) implicit */
                              /* (1-2Lx-2Ly)C(k) explicit */
5     if ((k-1)%D->c != D->c - 1)
6       b[k] += Lx(k) * C->m[k - 1];          /* LxC(k-1) */

7     if ((k+1)%D->c != 0)
8       b[k] += Lx(k) * C->m[k + 1];          /* LxC(k+1) */

9     if (k - D->c >= 0)
10      b[k] += Ly(k) * C->m[k - D->c];       /* LyC(k-N) */

11    if (k + D->c < D -> r * D -> c)
12      b[k] += Ly(k) * C->m[k + D->c];       /* LyC(k+N) */

13  }
14  return b;
15}
```

Nelle linee 1–2 si riserva la memoria per il vettore b, nel `for` (linee 3–13) si scrivono gli elemeti del vettore. Nella riga 4 è presente la variabile globale `implicit` che assume il valore 1 se si sta utilizzando la modalità implicita,

0 altrimenti. Le istruzioni `if` hanno la stesso scopo di quelle della funzione `coefficient_matrix()`. La sua complessità è $\Theta(n)$.

LUdecomposition

Il codice per la *fattorizzazione* LU della matrice A segue la strategia ricorsiva descritta nel paragrafo 2.4.1, comunque, per ottimizzare la funzione, è stata sostituita la ricorsione con un ciclo iterativo. Poiché la matrice di output L ha degli zero sopra la diagonale e ha dei valori uno nella diagonale, la funzione non si occuperà di riempire quelle posizioni. Analogamete, poiché la matrice U ha tutti gli elementi sotto la diagonale uguali a zero neanche queste posizioni saranno riempite. Il codice calcola, pertanto, soltanto gli elementi significativi delle matrici L ed U.

```
void LUdecomposition(float **A, int dim) {
  int k, i, j;
1   for (k = 0; k < dim; k++) {
2     for (i = k + 1; i < dim; i++) {
3       A[i][k] = A[i][k] / A[k][k];
4     }
5     for (i = k + 1; i < dim; i++)
6       for (j = k + 1; j < dim; j++)
7         A[i][j] = A[i][j] - A[i][k] * A[k][j];
8   }
9 }
```

La funzione, per ottimizzare la memoria utilizzata, memorizza le matrici L ed U nella matrice A stessa, terminata la funzione l' elemento $a_{i,j}$ apparterrà ad L se $i > j$, oppure ad U se $i \leq j$.

Il ciclo `for` esterno che comincia dalla riga 1, si ripete per ogni passo ricorsivo. Dentro il ciclo `for` alle linee 2–4, che non vengono eseguite quando $k = dim$, si pone in $a_{i,k}$ il valore v_i (linea 3). L' elemeto w_i^T sarà invece il valore in $a_{k,i}$. Infine la sottomatrice A' verrà memorizzata nella matrice A stessa nelle linee 5–8. `LUdecomposition()` impiega tempo $\Theta(n^3)$ poiché l' istruzione della linea 7 è interna a tre cicli.

LUsolve

La funzione `LUsolve` viene invocata solo quado si utilizza il metodo implicito, essa trova il vettore C_{n+1} memorizzandolo in `v`, combinando le sostituzioni in avanti e a ritroso (par. 2.4.2). Si ricava y usando la sostituzione in avanti nelle linee 5–10, e poi trova v usando la sostituzione all' indietro nelle linee 16–22. Nella sostituzione all' idietro le linee 16–19 si occupano di porre il valore della concentrazione a 255 in caso quasto lo superi, altrimenti nelle

linee 19–20 si approssima il valore ottenuto dalla sostituzione a ritroso e si memorizza in v.

```
void LUsolve(float *LU, float *b, int dim, Uint8 *v) {
  int i, j;
  float *y, *vett, sum;

1   if (((y = (float *)malloc(dim*sizeof(float))) == NULL) ||
2       ((vett = (float *)malloc(dim*sizeof(float))) == NULL) ||
3       ((v = (Uint8 *)malloc(dim*sizeof(Uint8))) == NULL))
4     exit(1);

5   for (i = 0; i < dim; i++) {
6     sum = 0;
7     for(j = 0; j < i; j++)
8       sum += LU[indd(i,j)] * y[j];
9     y[i] = b[i] - sum;
10  }

11  for (i = dim - 1; i >= 0; i--) {
12    sum = 0;
13    for(j = i + 1; j < dim; j++)
14      sum += LU[indd(i,j)] * vett[j];
15    vett[i] = ((y[i] - sum) / LU[indd(i,i)]);

16    if (((y[i] - sum) / LU[indd(i,i)]) > 255) {
17      printf("%f\n", (y[i] - sum) / LU[indd(i,i)] );
18      vett[i] = 255;
19    }
20    else vett[i] = ((y[i] - sum) / LU[indd(i,i)]) + 0.5;
21    v[i] = (Uint8)vett[i];
22  }
23  free(y);
24  free(vett);
25}
```

Il tempo di eseuzione è $\Theta(n^2 + n^2) = \Theta(n^2)$.

Capitolo 4

Esempi

Sommario

In questo capitolo verranno presentati vari esempi. Per ognuno di essi si mostrano prima le due immagini in input al simulatore, rispettivamente la concentrazione e la diffusione, poi viene mostrato il risultato dell' elaborazione con le relative opzioni passate al simulatore per ottenere tali risultati.

4.1 Coefficienti costanti

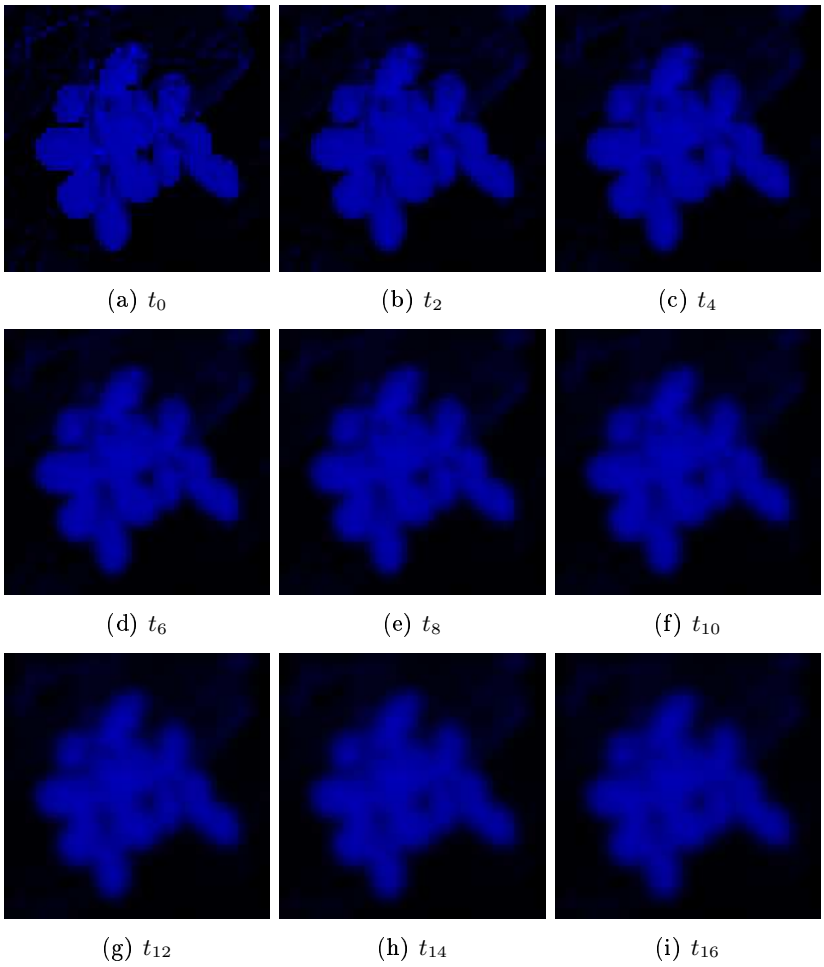
Un caso studio molto semplice del processo di diffusione consiste nel mantenere costanti i coefficienti di diffusione.

Figura 4.1: fiore

Utilizzeremo, per questo esempio, la gradazione del colore blu della figura 4.1 come concentrazione della materia. Lanciando il simulatore in questo modo:

$ fick −−dx 4 −−dy 4 −−dt 2 test/fiore.jpg

otterremo il seguete risultato:



(a) t_0 (b) t_2 (c) t_4

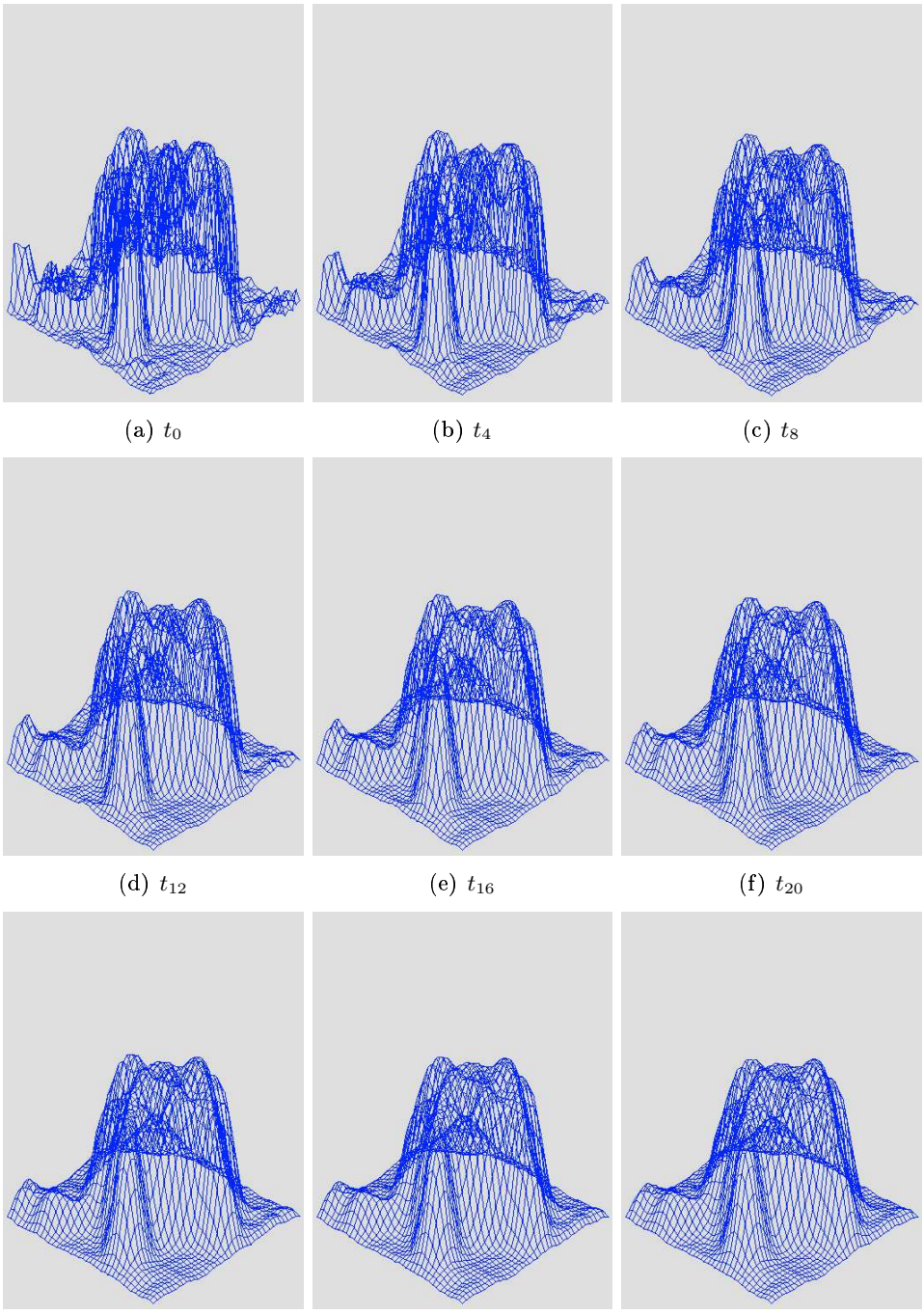
(d) t_6 (e) t_8 (f) t_{10}

(g) t_{12} (h) t_{14} (i) t_{16}

Figura 4.2: Variazione della concentrazione con coefficienti costanti

Possiamo visualizzare la variazione della cocentrazione in modalità 3D semplicemnete aggiungendo l' opzione "−−3d".

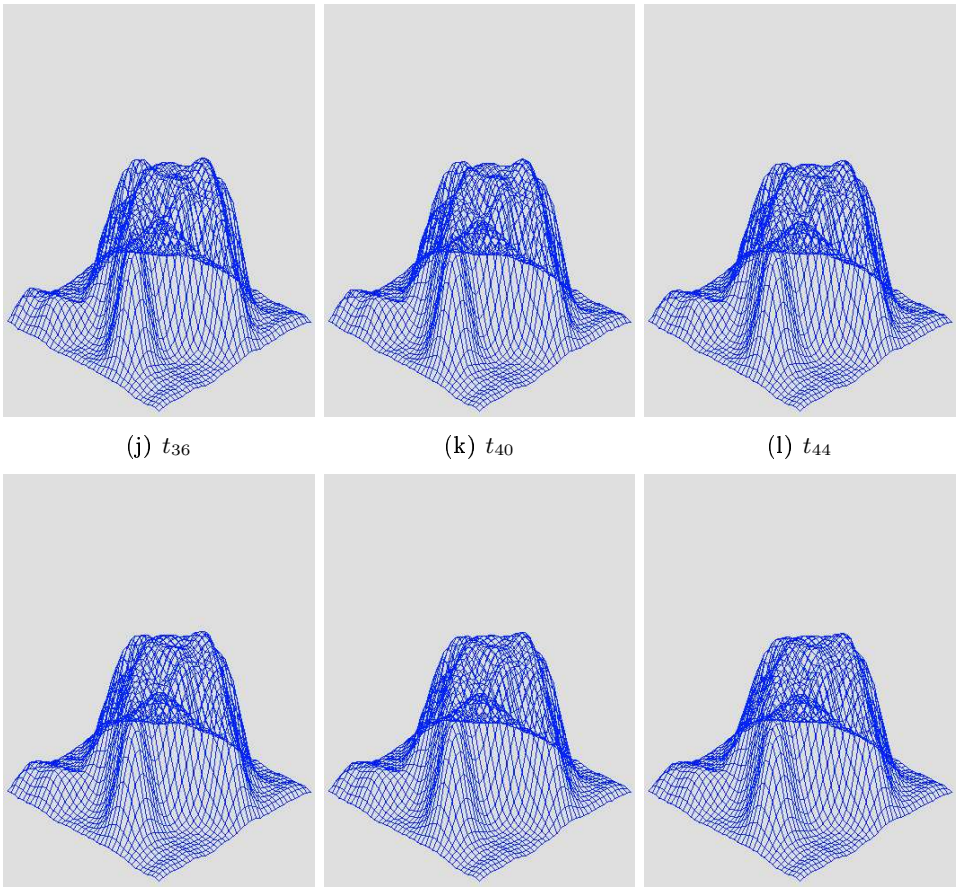
$ fick −−dx 5 −−dy 5 −−dt 4 test/fiore.jpg −−3d



(a) t_0 (b) t_4 (c) t_8

(d) t_{12} (e) t_{16} (f) t_{20}

(g) t_{24} (h) t_{28} (i) t_{32}



(j) t_{36} (k) t_{40} (l) t_{44}

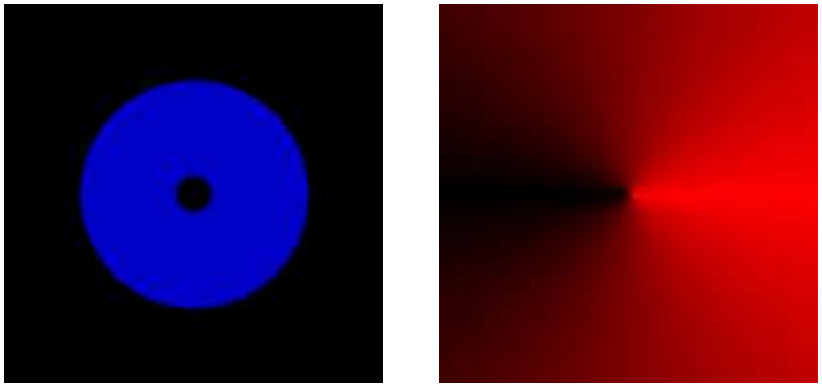
(m) t_{48} (n) t_{52} (o) t_{56}

Figura 4.3: coefficienti costanti in 3D

4.2 Coefficienti variabili

Abbiamo considerato un caso in cui i coefficienti di diffusione restavano uguali ad uno in tutta l' area di studio. In generale il risultato di una simulazione risulta essere un buon modello se si avvicina il più possibile alla realtà. Nella maggior parte delle situazioni in cui si può osservare un fenomeno diffusivo la zona in osservazione difficilmente risulta uniforme. Esistono molti fattori che influiscono sul moto delle molecole come ad esempio la temperatura, la pressione, l' umidità, la densità, i campi magnetici, la luminosità...

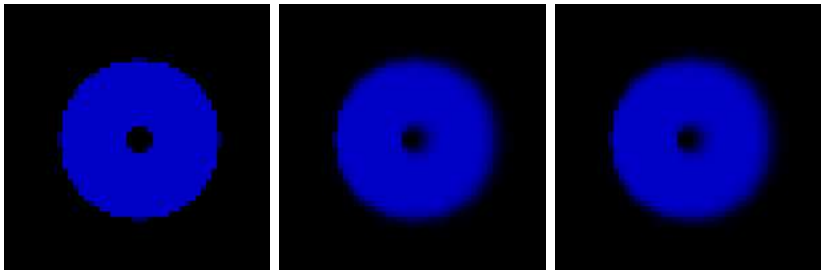
Tutti questi fattori, che possono variare da un punto all' altro della zona sotto esame, vegono considerati nel coefficiente di diffusione. Nell' esempio seguente il fluido sarà tutto concentrato verso il centro, quello che varierà saranno i coefficienti di diffusione che andranno a formare un gradiente simmetrico (Figura 4.4).



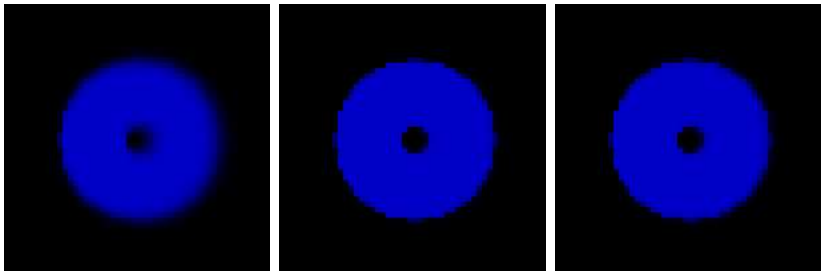
(a) concentrazione (b) coefficienti di diffusione

Figura 4.4: Coefficienti variabili

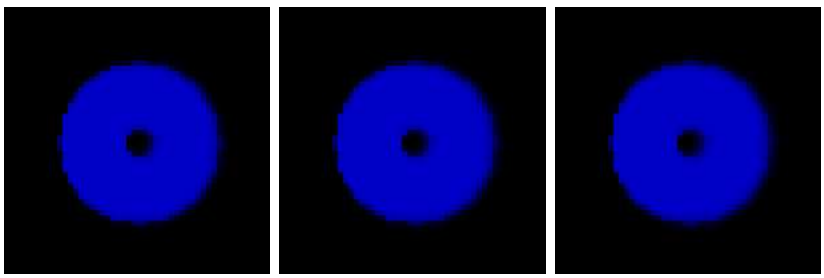
$ fick -x 4 -y 4 --dt 2 test/ciambella.bmp test/grad_simm.bmp



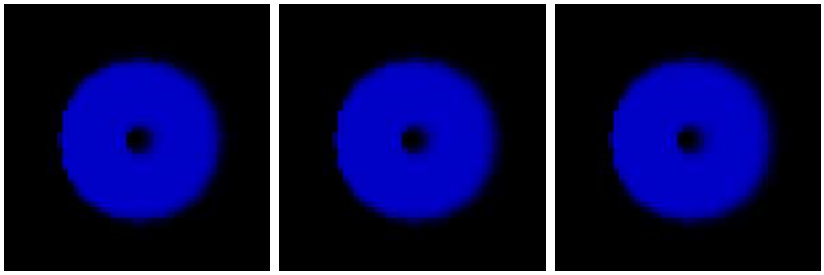
(a) t_0 (b) t_2 (c) t_4



(d) t_6 (e) t_8 (f) t_{10}



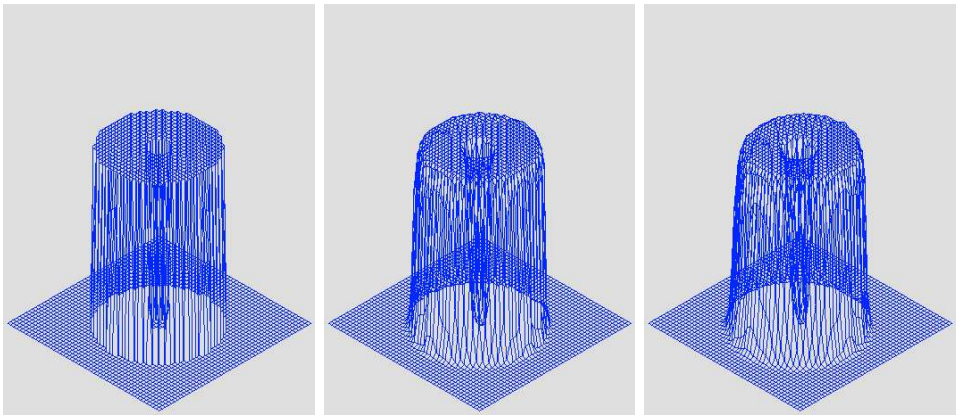
(g) t_{12} (h) t_{14} (i) t_{16}



(j) t_{18} (k) t_{20} (l) t_{22}

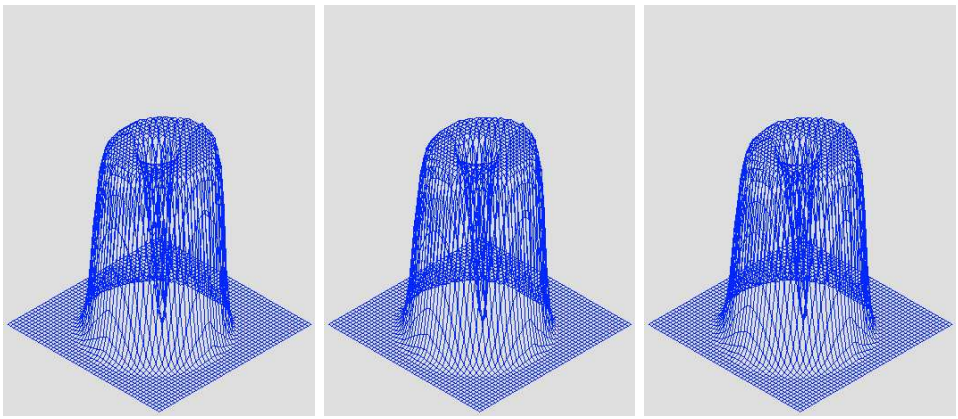
Figura 4.5: Variazione della concentrazione con coefficienti variabili

$ fick -x 4 -y 4 -t 4 --3d test/ciambella.bmp test/grad_simm.bmp

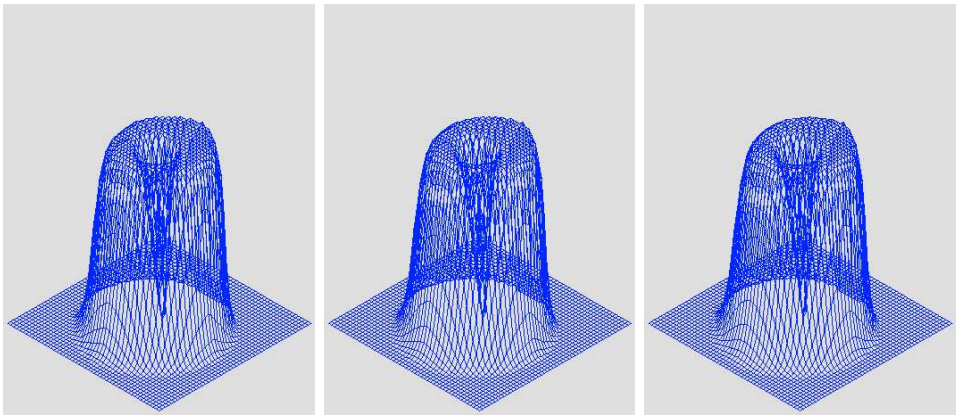


(a) t_0 (b) t_4 (c) t_8

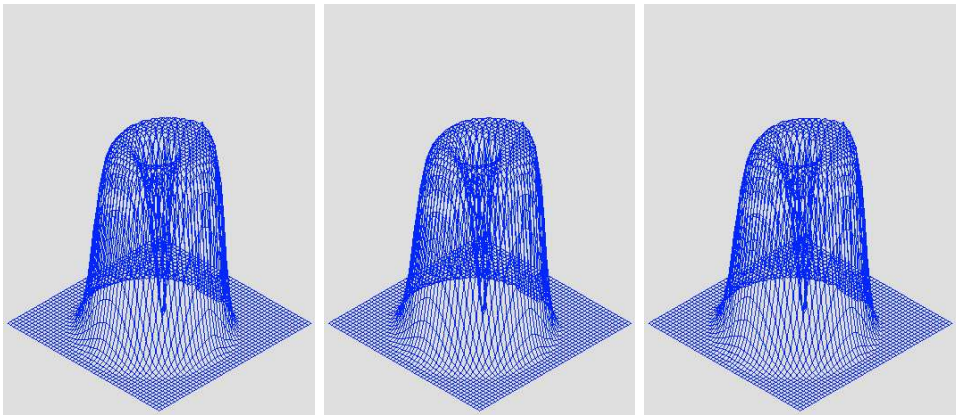
(d) t_{12} (e) t_{16} (f) t_{20}



(g) t_{24} (h) t_{28} (i) t_{32}



(j) t_{36} (k) t_{40} (l) t_{44}



(m) t_{48} (n) t_{52} (o) t_{56}

Figura 4.6: coefficienti costanti in 3D

4.3 Esperimento

bla bla...

Appendice A

Codice sorgente

A.1 header.h

```
/* include */

#ifndef HEADER

#include "config.h"

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <SDL/SDL.h>
#include <SDL/SDL_image.h>
#include <getopt.h>

/* const */

enum {R, G, B}; /* Red, Green, Blue */

#define HEADER
#define BPP           32  /* max number of bits per pixel */
#define TIME_INTERVAL 250 /* ms */
#define FILE_OUT      "out.bmp"
#define SIN30         0.5             /* sin(30) */
#define COS30         0.866025403784438646763723170754 /* cos(30) */
#define BORDER        5 /* pixel */

#if SDL_BYTEORDER == SDL_BIG_ENDIAN
  #define    rmask  0xff000000
  #define    gmask  0x00ff0000
```

```
  #define    bmask  0x0000ff00
  #define    amask  0x000000ff
#else
  #define    rmask  0x000000ff
  #define    gmask  0x0000ff00
  #define    bmask  0x00ff0000
  #define    amask  0xff000000
#endif

/* macro */

#define Min(a,b)  ((a) < (b))? (a) : (b)
#define Ind(x,y)  ((x)*(M->c) + (y))
#define Lx(index) (Dt * D->m[index] / (float)(Dx*Dx*255))
#define Ly(index) (Dt * D->m[index] / (float)(Dy*Dy*255))

/* global variables */

SDL_Rect background;

static Uint32 full_screen = 0;
static int mode3d = 0, implicit = 1;

/* structure definition */

typedef struct {
  int color; /* R G or B */
  int r, c;  /* matrix dimenction */
  Uint8 *m;
}Matrice;

typedef struct {
  int r, c;
}Limit;

static struct option long_options[] = {
  {"3d",          no_argument,       &mode3d, 1},
  {"expicit",     no_argument,       &implicit, 0},
  {"fullscreen",  no_argument,       0, 'f'},
  {"help",        no_argument,       0, 'h'},
  {"version",     no_argument,       0, 'v'},
  {"dx",          required_argument, 0, 'x'},
  {"dy",          required_argument, 0, 'y'},
  {"dt",          required_argument, 0, 't'},
```

```
  {"steps",       required_argument, 0, 's'},
  {"output",      required_argument, 0, 'o'},
  {"delay",       required_argument, 0, 'd'},
  {"Width",       required_argument, 0, 'W'},
  {"Height",      required_argument, 0, 'H'},
  {0, 0, 0, 0}
};


#endif
```

A.2 fick.c

```
/*
 * Fick - Version 0.2
 *
 *  Copyright (C) 2005  Danilo Abbasciano
 *
 *  This program is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation; either version 2 of the License, or
 *  (at your option) any later version.
 *
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with this program; if not, write to the Free Software
 *  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 *
 *  Bug Report: <danilo.abbasciano@gmail.com>
 *
 */

#include "header.h"

int main(int argc, char **argv) {
  SDL_Event event;
  SDL_Surface *screen;
  char *file_out = NULL;
  int Dx = 0, Dy = 0, w = 0, h = 0;
  float Dt = 0;
  float **A, *b;
  int option_index, c, steps = -1, delay = TIME_INTERVAL;
  Matrice D, C;
  char t[20];

  while (1) {
    option_index = 0;
    if ((c = getopt_long(argc, argv, "hvqex:y:t:s:o:d:W:H:f", long_options,
 &option_index)) == -1)
      break;
```

```
switch (c) {
case 0:
  break;
case 'h':
  printf(PACKAGE" version "PACKAGE_VERSION"\n"
 "Usage: %s [OPTIONS] ImageDistribition [ImageDiffusion]\n\n"
 "This program supports images from the following formats:\n"
 " TGA, BMP, PNM (include: pnm, pbm, pgm, ppm), XPM, XCF, PCX,\n"
 " GIF, JPG, JPEG, TIF, TIFF, LBM, IFF, PNG\n\n"
 "OPTIONS:\n"
 "  -f --fullscreen\t\t: full screen mode\n"
 "  -e --explicit\t\t\t: use explicit method\n"
 "  --3d\t\t\t\t: 3d mode\n"
 "  -x <arg> --dx <arg>\t\t: set Dx\n"
 "  -y <arg> --dy <arg>\t\t: set Dy\n"
 "  -t <arg> --dt <arg>\t\t: set Dt\n"
 "  -s <n> --steps <n>\t\t: the program stops after <n> steps\n"
 "  -d <s> --delay <s>\t\t: ms between two image (default 250)\n"
 "  -W <n> --Width <n>\t\t: width of the surface\n"
 "  -H <n> --Height <n>\t\t: height of the surface\n"
 "  -o <file> --output <file>\t: save results as a BMP <file>\n"
 "  -v --version\t\t\t: print version info and exit\n"
 "  -h --help\t\t\t: display this help and exit\n\n"
 "Report bugs to <"PACKAGE_BUGREPORT">\n", argv[0]);
  return 1;
case 'v':
  printf(PACKAGE_STRING"\n\n"
 "Written by Danilo Abbasciano.\n\n"

 "This is free software; see the source for copying conditions."
 "There is NO\n"
 "warranty; not even for MERCHANTABILITY or FITNESS FOR A"
 "PARTICULAR PURPOSE.\n");
  return 1;
case 'e': implicit = 0;
  break;
case 'x': sscanf(optarg, "%d", &Dx);
  break;
case 'y': sscanf(optarg, "%d", &Dy);
  break;
case 't': sscanf(optarg, "%f", &Dt);
  break;
case 's': sscanf(optarg, "%d", &steps);
  break;
```

```
  case 'o': file_out = optarg;
    break;
  case 'd': sscanf(optarg, "%d", &delay);
    break;
  case 'W': sscanf(optarg, "%d", &w);
    break;
  case 'H': sscanf(optarg, "%d", &h);
    break;
  case 'f': full_screen = SDL_FULLSCREEN;
    break;
  case '?': default:
    return 1;
  }
}

if (argc - optind != 1 && argc - optind != 2) {
  fprintf(stdout, "error: Try `%s --help' for more information.\n", argv[0]);
  return 1;
}

/* Verify the stability restriction */
if (!implicit && Dt > Dx*Dx*Dy*Dy / (2.0 * (Dx*Dx + Dy*Dy))) {
  Dt = Dx*Dx*Dy*Dy / (2.0 * (Dx*Dx + Dy*Dy));
  printf("For rispect the stability restriction must be used Dt = %f\n", Dt);
}

/* inizializing SDL */
if (SDL_Init (SDL_INIT_VIDEO | SDL_INIT_TIMER) < 0) {
  fprintf(stderr, "Impossibile inizializzare SDL: %s\n", SDL_GetError() );
  exit(1);
}
atexit(SDL_Quit);

while (Dx <= 0) {
  printf("Dx > ");
  scanf("%d", &Dx);
}
while (Dy <= 0) {
  printf("Dy > ");
  scanf("%d", &Dy);
}
while (Dt <= 0) {
  printf("Dt > ");
  scanf("%f", &Dt);
```

```
}

if (w == 0) w = Dx;
if (h == 0) h = Dy;

/* Load the D and C matrix with the color R(Red) of diffuzion image file
   and color B(Blue) of distribution image file. One element of matrix are
   the mean of a partition of image.*/

C.color = B;
image2matrix(loadImage(argv[optind]), Dx, Dy, &C);

D.color = R;
image2matrix(loadImage(argv[optind + 1]), Dx, Dy, &D);

C.r = D.r = Min(C.r, D.r);
C.c = D.c = Min(C.c, D.c);

SDL_WM_SetCaption(PACKAGE, NULL);

printf("Wait...\n");

if (implicit) {
  /* making a values on A matrix */
  A = (float **)coefficient_matrix(Dx, Dy, Dt, &D);

  /* decompose the matrix A in two matrix L and U. L and U are stored in A */
  LUdecomposition(A, D.r*D.c);
}

if (mode3d) {

  screen = SDL_SetVideoMode(2*BORDER + COS30*(w * (C.r - 1) + h*(C.c - 1)),
    2*BORDER + SIN30*(w * (C.r - 1) + h*(C.c - 1)) + 255,
    BPP,
    SDL_HWSURFACE | SDL_ANYFORMAT | full_screen );

  background.w = screen -> w;
  background.h = screen -> h;

  matrix2image3d(screen, w, h, &C);
} else {
  screen = SDL_SetVideoMode(C.c * w, C.r * h, BPP,
    SDL_HWSURFACE | SDL_ANYFORMAT | full_screen );
```

```
    matrix2image(screen, w, h, &C);
  }

  SDL_UpdateRect(screen, 0, 0, screen -> w, screen -> h);

   sprintf(t, "ex2_3d-00.bmp");
   SDL_SaveBMP(screen, t);

  printf("Start simulation\n");

  while (steps != 0) {
    /* calcolate_b resolv: b = (2Id + A)Ct end return the b vector. */
    b = (float *)calcolate_b(Dx, Dy, Dt, &D, &C);

    if (implicit) {
      /* delete the concentration at time t */
      free(C.m);

      /*  LUsolve resolv the system L*U*C(t+1) = b,
  return the results in a vector.  */
      LUsolve(A, b, D.r*D.c, C.m);
    } else { /* explicit mode */
      for (c = 0; c < D.r*D.c; c++) {
if (b[c] + 0.5 > 255) C.m[c] = 255;
else C.m[c] = (Uint8)(b[c] + 0.5);
      }
      free(b);
    }

    if (mode3d) matrix2image3d(screen, w, h, &C);
    else matrix2image(screen, w, h, &C);

    SDL_Delay(delay);
    /* showImage  */
    SDL_UpdateRect(screen, 0, 0, screen -> w, screen -> h);

    if (20 - steps < 10)
      sprintf(t, "ex2_3d-0%d.bmp", 20 - steps);
    else sprintf(t, "ex2_3d-%d.bmp", 20 - steps);
    SDL_SaveBMP(screen, t);


    while (SDL_PollEvent(&event) == 1) {
```

```
      switch (event.type) {
      case SDL_KEYDOWN:
if(event.key.keysym.sym == SDLK_f)
  SDL_WM_ToggleFullScreen(screen);

if (event.key.keysym.sym == SDLK_ESCAPE ||
    event.key.keysym.sym == SDLK_q)
  stop(screen, file_out, C.m, D.m, A, D.r*D.c);

if (event.key.keysym.sym == SDLK_PLUS)
  Dt++;

if (event.key.keysym.sym == SDLK_MINUS)
  if (Dt > 0) Dt--;

if (event.key.keysym.sym == SDLK_o) {
   if (SDL_SaveBMP(screen, FILE_OUT) == -1)
     fprintf(stderr, "error\n");
}

break;
      case SDL_QUIT:
stop(screen, file_out, C.m, D.m, A, D.r*D.c);
      }
    }
    steps = steps - 1;
  }
  stop(screen, file_out, C.m, D.m, A, D.r*D.c);
  return 0;
}
```

A.3 functio.c

```
#include "header.h"

Uint32 getpixel(SDL_Surface *surface, int x, int y) {
    int bpp = surface->format->BytesPerPixel;
    /* Here p is the address to the pixel we want to retrieve */
    Uint8 *p = (Uint8 *)surface->pixels + y * surface->pitch + x * bpp;

    switch(bpp) {
    case 1:
        return *p;

    case 2:
        return *(Uint16 *)p;

    case 3:
        if(SDL_BYTEORDER == SDL_BIG_ENDIAN)
            return p[0] << 16 | p[1] << 8 | p[2];
        else
            return p[0] | p[1] << 8 | p[2] << 16;

    case 4:
        return *(Uint32 *)p;

    default:
        return 0;
    }
}

void putpixel(SDL_Surface *s, int x, int y, Uint32 pixel) {
  int bpp = s -> format -> BytesPerPixel;
  Uint8 *p = (Uint8 *)s -> pixels + y * s -> pitch + x * bpp;

  switch (bpp) {
  case 1: *p = pixel;
    break;
  case 2: *(Uint16 *)p = pixel;
    break;
  case 3:
    if (SDL_BYTEORDER == SDL_BIG_ENDIAN) {
      p[0] = (pixel >> 16) & 0xff;
      p[2] = pixel & 0xff;
    }else {
```

```
      p[0] = pixel & 0xff;
      p[2] = (pixel >> 16) & 0xff;
    }
    p[1] = (pixel >> 8) & 0xff;
    break;
  case 4: *(Uint32 *)p = pixel;
  }
}

Uint8 getColor(SDL_Surface *s, int x, int y, int color) {
  Uint8 v[3];
  SDL_GetRGB(getpixel(s, x, y), s -> format, &v[R], &v[G], &v[B]);
  return v[color];
}

/* Lock the screen for direct access to the pixels */
void lockScreen(SDL_Surface *s) {
  if (SDL_MUSTLOCK(s))
    if ( SDL_LockSurface(s) < 0 ) {
      fprintf(stderr, "Can't lock screen: %s\n", SDL_GetError());
      exit (1);
    }
}

void unlockScreen(SDL_Surface *s) {
  if (SDL_MUSTLOCK(s))
    SDL_UnlockSurface(s);
}

void image2matrix(SDL_Surface *image,
       int Dx, int Dy, Matrice *M) {
  int i, j;

  M -> r = (image -> h / Dy);
  M -> c = (image -> w / Dx);

  if ((M -> m = (Uint8 *)calloc(M -> r * M -> c, sizeof(Uint8))) == NULL) {
    printf("Program exit\n");
    exit(1);
  }

  lockScreen(image);

  for (i = 0; i < M -> r; i++)
```

```
    for (j = 0; j < M -> c; j++)
      M -> m[Ind(i, j)] = getColor(image, j*Dx, i*Dy, M -> color);

  unlockScreen(image);
}

void matrix2image(SDL_Surface *s, int scalax, int scalay, Matrice *M) {
  int i, j;
  Uint8 v[3] = {0, 0, 0};

  lockScreen(s);

  for (i = 0; i < s -> h; i++)
    for (j = 0; j < s -> w; j++) {
      /*
      SDL_GetRGB(getpixel(s, i, j), s -> format, &v[R], &v[G], &v[B]); */

      v[M -> color] = M -> m[Ind((i/scalay), (j/scalax))];
      putpixel(s, j, i,
       SDL_MapRGB(s -> format, v[R], v[G], v[B]));
    }
  unlockScreen(s);
}

SDL_Surface *loadImage(char *file) {
  SDL_Surface *image;

  if (file != NULL) {
    image = IMG_Load_RW(SDL_RWFromFile(file, "rb"), 1);

    background.x = background.y = 0;
    background.w = image -> w;
    background.h = image -> h;

  } else {
    image = SDL_CreateRGBSurface(SDL_HWSURFACE, background.w, background.h,
 BPP, rmask, gmask, bmask, amask);

    SDL_FillRect(image, &background,
 SDL_MapRGB(image -> format, 255, 0, 0));

  }
  if (image == NULL) {
    fprintf(stderr, "Error: %s\n", IMG_GetError());
```

```
    exit(1);
  }
  return image;
}

/* D e' la matrice dei coefficienti di diffusione */
float **coefficient_matrix(int Dx, int Dy, double Dt, Matrice *D) {
  float **A;
  int k;
  int dim = D -> r * D -> c;

  if ((A = (float **)malloc(dim * sizeof(float *))) == NULL) exit(1);

  for (k = 0; k < dim; k++)
    if ((A[k] = (float *)calloc(dim, sizeof(float))) == NULL) {
      printf("the memory is below to run fick.\n");
      exit(1);
    }

  /* scrive i 5 elementi diversi da 0 per ogni riga di A */
  for (k = 0; k < dim; k++) {
    A[k][k] = 2*(1 + Lx(k) + Ly(k)); /* 2(1 + Lx + Ly) */

    if (k - 1 >= 0 && (k-1)%D->c != D->c - 1)
      A[k][k-1] = -1 * Lx(k);  /* -Lx */

    if (k + 1 < dim && (k+1)%D->c != 0)
      A[k][k+1] = -1 * Lx(k);  /* -Lx */

    if (k - D->c >= 0)
      A[k][k - D->c] = -1 * Ly(k);  /* -Ly */

    if (k + D->c < dim)
      A[k][k+D->c] = -1 * Ly(k);  /* -Ly */
  }
  return A;
}

/*  b[k] = 2(1-Lx-Ly)C(k) + LxC(k-1) + LxC(k+1) + LyC(k-N) + LyC(k+N)
 */
float *calcolate_b(int Dx, int Dy, double Dt, Matrice *D, Matrice *C) {
  int k;
  float *b;
```

```
  if ((b = (float *)malloc(D -> r * D -> c * sizeof(float))) == NULL) exit(1);

  for (k = 0; k < (D -> r * D -> c); k++) {

    b[k] = ((implicit+1) - 2*Lx(k) - 2*Ly(k))* C->m[k];
                                          /* (2-2Lx-2Ly)C(k) implicit */
                                          /* (1-2Lx-2Ly)C(k) explicit */
    if ((k-1)%D->c != D->c - 1)
      b[k] += Lx(k)* C->m[k - 1];             /* LxC(k-1) */

    if ((k+1)%D->c != 0)
      b[k] += Lx(k)* C->m[k + 1];             /* LxC(k+1) */

    if (k - D->c >= 0)
      b[k] += Ly(k)* C->m[k - D->c];        /* LyC(k-N) */

    if (k + D->c < D -> r * D -> c)
      b[k] += Ly(k)* C->m[k + D->c];      /* LyC(k+N) */

  }
  return b;
}

void LUdecomposition(float **A, int dim) {
  int k, i, j;

  for (k = 0; k < dim; k++) {
    for (i = k + 1; i < dim; i++) {
      A[i][k] = A[i][k] / A[k][k];
    }
    for (i = k + 1; i < dim; i++)
      for (j = k + 1; j < dim; j++)
A[i][j] = A[i][j] - A[i][k] * A[k][j];
  }
}

void LUsolve(float **LU, float *b, int dim, Uint8 *v) {
  int i, j;
  float *y, *vett, sum;

  if (((y = (float *)malloc(dim*sizeof(float))) == NULL) ||
      ((vett = (float *)malloc(dim*sizeof(float))) == NULL) ||
      ((v = (Uint8 *)malloc(dim*sizeof(Uint8))) == NULL))
    exit(1);
```

```
  for (i = 0; i < dim; i++) {
    sum = 0;
    for(j = 0; j < i; j++)
      sum += LU[i][j] * y[j];
    y[i] = b[i] - sum;
  }

  for (i = dim - 1; i >= 0; i--) {
    sum = 0;
    for(j = i + 1; j < dim; j++)
      sum += LU[i][j] * vett[j];
    vett[i] = ((y[i] - sum) / LU[i][i]);

    if (((y[i] - sum) / LU[i][i]) > 255) {
      printf("%f\n", (y[i] - sum) / LU[i][i] );
      vett[i] = 255;
    }
    else vett[i] = ((y[i] - sum) / LU[i][i]) + 0.5;
    v[i] = (Uint8)vett[i];
  }
  free(y);
  free(vett);
}

void stop(SDL_Surface *screen, char *file, Uint8 *a, Uint8 *b, float *f[], int dim) {
  int i;
  if (file != NULL) {
    if (SDL_SaveBMP(screen, file) == -1)
      printf("error\n");
      }
  if (a != NULL) free(a);
  if (b != NULL) free(b);
  if (f != NULL) {
    for (i = 0; i < dim; i++)
      free(f[i]);
    free(f);
  }
  SDL_FreeSurface(screen);
  exit(0);
}
```

A.4 3d.c

```
#include "header.h"


void line(SDL_Surface *s, int x0, int y0, int x1, int y1, Uint32 color) {
  int cont;

  float m = (y1 - y0) / (float)(x1 - x0);

  if (x0 == x1)  /* vertical line */
    for( cont = y0; cont != y1;  cont = (y0<y1)? cont+1 : cont-1 )
      putpixel(s, x0, cont, color);
  else {
    if (abs(y1 - y0) < abs(x1 - x0)) /* -1 < m < 1 */

      for (cont = x0; cont != x1; cont = (x0<x1)? cont+1 : cont-1 )
putpixel(s,
 cont,
 (int)(m*(cont - x0) + y0),
 color);

    else /* m < -1 or m > 1*/

      for (cont = y0; cont != y1; cont = (y0<y1)? cont+1 : cont-1 )
putpixel(s,

 (int)((m * x0 + cont - y0)/m),
 cont,
 color);
  }
}

#define COLOR_LINE SDL_MapRGB(s -> format, 0, 40, 255)
#define COLOR_POINT SDL_MapRGB(s -> format, 0, 0, 255)
#define COLOR_BACKGROUND SDL_MapRGB(s -> format, 222, 222, 222)

#define UNIT_Y(i, j) (SIN30 * (scalai * (i) + scalaj * (j)))
#define UNIT_X(i, j) (COS30 * (scalai * (i) - scalaj * (j)))

#define ORIGINE_X (COS30 * scalaj * (M -> c - 1) + BORDER)
#define ORIGINE_Y 255 + BORDER

void matrix2image3d(SDL_Surface *s, int scalai, int scalaj, Matrice *M) {
```

```
  int i,j;

  /* clear Surface */
  SDL_FillRect(s, &background, COLOR_BACKGROUND);

  lockScreen(s);

  for (i = 0; i < M -> r; i++)
    for (j = 0; j < M -> c; j++) {
      if (i > 0)
line(s,(int)(ORIGINE_X + UNIT_X(i,j)),
     (int)(ORIGINE_Y + UNIT_Y(i,j) - M->m[Ind(i,j)]),
     (int)(ORIGINE_X + UNIT_X(i-1, j)),
     (int)(ORIGINE_Y + UNIT_Y(i-1,j) - M->m[Ind(i-1,j)]),
     COLOR_LINE);

      if (j > 0)
line(s,(int)(ORIGINE_X + UNIT_X(i,j)),
     (int)(ORIGINE_Y + UNIT_Y(i,j) - M->m[Ind(i,j)]),
     (int)(ORIGINE_X + UNIT_X(i, j-1)),
     (int)(ORIGINE_Y + UNIT_Y(i,j-1) - M->m[Ind(i,j-1)]),
     COLOR_LINE);
      putpixel(s,
       (int)(ORIGINE_X + UNIT_X(i,j)),
       (int)(ORIGINE_Y + UNIT_Y(i,j) - M->m[Ind(i,j)]),
       COLOR_POINT);

    }
  /*
  putpixel(s,ORIGINE_X, ORIGINE_Y,
   SDL_MapRGB(s -> format, 255, 0, 0));
  */
  unlockScreen(s);
}
```

Bibliografia

[1] Stephen Wolfram. *Cellular automata and complexity.* Addison–Wesley, 1994

[2] J.Crank. *The mathematics of diffusion.* second ed. Oxford University Press, 1975

[3] A.Ghizzetti, F. Rosati. *Analisi matematica.* Masson, 1996

[4] G.D.Smith. *Numerical solution of partial differential equations.* third ed. Oxford University Press, 1985

[5] T.H. Cormen, C.E. Leiserson, L. Rivest. *Introduzione agli algoritmi.* Jackson Libri

[6] Brian W. Kernigham, Dennis M. Ritchie. *The C programming language.* second edition. Pretince-Hall