Università degli studi di Roma Tre
Facoltà di Scienze Matematiche Fisiche e Naturali
Corso di Laurea in Matematica

Tesi di Laurea in Matematica di

Cristiana Vigliaroli

Tecniche per il calcolo scientifico distribuito in Java

Relatore
Prof. Marco Liverani

Il Candidato Il Relatore

Anno Accademico 2000/2001
Novembre 2001

Classificazione AMS: 68C05, 68C25, 68E10
Parole chiave: Java, programmazione distribuita, calcolo scientifico, EDO

Indice

Introduzione

Negli ultimi anni abbiamo assistito a delle grandi innovazioni per ciò che riguarda l'approccio automatico, con l'ausilio di un calcolatore, alla soluzione di problemi di carattere gestionale o di rilevanza computazionale e numerica. L'Informatica e l'industria che segue e sollecita le innovazioni di carattere teorico e le intuizioni innovative, ultimamente ci hanno abituato non soltanto ad assistere ad una rapidissima evoluzione delle capacità di calcolo delle macchine, ma anche alla nascita di nuovi modelli, di nuove architetture e di nuovi approcci alla risoluzione di problemi mediante un computer.

Accanto alla produzione di calcolatori sempre più veloci (la potenza di un normale Personal Computer oggi è forse superiore a quella di un *mainframe* degli anni settanta), sono state sviluppate nuove "tecnologie abilitanti" e nuove modalità operative. Per tecnologie abilitanti intendiamo tutti quegli strumenti tecnici che consentono di accedere a nuove possibilità nell'uso di una macchina e nella progettazione di nuovi servizi. Solo pochi anni fa era impensabile l'attuale diffusione di telefoni cellulari, mentre oggi troviamo abbastanza normale poter utilizzare la rete Internet da ogni parte del mondo. Anzi: quando ci troviamo a non poter utilizzare la rete, ci risulta difficile l'uso dello stesso computer. Senza la possibilità di condividere file ed informazioni con altri utenti della rete, sembra venir meno l'utilità di queste macchine.

Una delle innovazioni principali quindi è proprio quella costituita dalla diffusione di massa della rete Internet e dal potenziamento delle linee dati che

collegano fra loro i nodi della rete. Oggi, grazie a queste tecnologie, possiamo utilizzare indifferentemente applicazioni grafiche che operano su una macchina di una università italiana, collegandoci da un terminale (di fatto un Personal Computer o un Macintosh) dalla stanza di un albergo negli Stati Uniti. Fino a pochi anni fa eravamo al massimo in grado di leggere la posta elettronica, e non senza qualche disagio!

In questo contesto si inserisce lo sviluppo di un nuovo linguaggio di programmazione, il linguaggio Java, progettato appositamente per trarre il massimo del beneficio dall'uso della rete (sia essa una "rete locale" o anche una "rete geografica" come Internet).

In una prima fase Java è stato interpretato da molti soltanto come un linguaggio mediante il quale si potessero arricchire di componenti interattive le pagine di un sito Web. L'uso che se ne è fatto è stato principalmente quello di realizzare le cosiddette *applet Java* che effettivamente permettevano di estendere le scarse possibilità offerte dal linguaggio HTML utilizzato per codificare le pagine Web.

Oggi Java è evoluto al punto da essere considerato uno dei principali linguaggi per lo sviluppo di applicazioni in ambito industriale: tutti i prodotti più diffusi (pensiamo ad esempio ai database) consentono la realizzazione di funzionalità aggiuntive mediante il linguaggio Java. In molti contesti le applicazioni scritte in Java stanno sostituendo vecchi programmi realizzati in Cobol, in Fortran ed anche in C.

Il successo di questo linguaggio è da ricercarsi nelle sue caratteristiche intrinseche che effettivamente semplificano di molto la realizzazione di funzionalità altrimenti estremamente complesse. Quali sono queste caratteristiche? Innanzi tutto il fatto che Java è un linguaggio ad oggetti: non è certo l'unico, nè il migliore, sotto questo punto di vista, però questa caratteristica di base già da sola basterebbe a farlo preferire ad altri linguaggi, visto che consente

di ingegnerizzare in modo più elegante le applicazioni ed i programmi.

Ma non solo: Java fornisce diverse *classi* di oggetti per la realizzazione di procedure che sfruttano in pieno i servizi offerti dalla rete. Questo si traduce in una stretta integrazione con la tecnologia Web, ma soprattutto nella disponibilità di funzioni (che chiameremo *metodi* nella terminologia *object oriented* di Java) che permettono di utilizzare i *socket* del protocollo TCP/IP ed il protocollo RMI (*Remote Method Invocation*) per la progettazione di applicazioni *distribuite* di alto livello.

Una applicazione distribuita non è altro che un programma frammentato in due o più componenti distinte che possono essere eseguite su calcolatori differenti, ma collegati in rete fra loro e coordinati da una applicazione di livello superiore che stabilisce la distribuzione del carico di lavoro sulle diverse macchine. A differenza di quanto avviene nel caso delle applicazioni *parallele*, in questo caso ogni macchina esegue operazioni differenti, specializzandosi anzi, nell'esecuzione di procedure ben precise messe a disposizione degli altri calcolatori che contribuiscono all'esecuzione del "calcolo distribuito".

Questa tesi fornisce una descrizione del lavoro di progettazione ed implementazione di un modello di calcolo distribuito applicato alla risoluzione di un problema di carattere numerico classico. In particolare ci siamo occupati di verificare la praticabilità di una architettura di calcolo distribuito, realizzata in linguaggio Java sfruttando la tecnologia RMI, per la soluzione approssimata di un sistema di equazioni differenziali ordinarie (EDO) con i metodi di Eulero e di Runge Kutta. Sia il problema che i metodi utilizzati sono argomenti standard, ben consolidati e naturalmente privi di aspetti innovativi: è chiaro quindi che sono da considerarsi solo strumentali alla verifica di un modello di calcolo, quale appunto quello distribuito mediante RMI.

La tesi si articola su tre capitoli: nel primo viene presentato il linguaggio Java e la metodologia di programmazione ad oggetti (OOP, *Object Oriented Pro-*

gramming). Abbiamo approfondito i concetti di incapsulamento, ereditarietà e polimorfismo, tipici della programmazione ad oggetti. In questo capitolo viene proposta una sintesi delle caratteristiche fondamentali del linguaggio Java, presentando alcune distinzioni tra le tipologie di programmi che possono essere realizzati (es.: le *applet*, le *applications*, ecc.) e affrontando le problematiche che un approccio *object oriented* implicano nella soluzione di un problema mediante un calcolatore.

Il secondo capitolo affronta il protocollo RMI (*Remote Method Invocation*), l'aspetto del linguaggio Java di maggiore rilevanza per il nostro obiettivo di realizzare una architettura di calcolo distribuita. In questo capitolo infatti viene anche descritta la modalità con cui può essere progettata ed implementata una applicazione distribuita che sfrutti i metodi offerti da RMI. Vengono anche presentate le *applet*, una tecnica per costruire applicazioni che vengono trasferite ed eseguite automaticamente sulla macchina dell'utente prelevandole da un server presente in rete (o su Internet).

Nel terzo capitolo viene descritto il problema matematico affrontato (la risoluzione di sistemi di equazioni differenziali ordinarie), concentrando l'attenzione sugli algoritmi per il calcolo approssimato di tali sistemi. Come esempi abbiamo utilizzato l'equazione dell'oscillatore armonico e dell'oscillatore armonico smorzato, l'equazione di Van der Pol ed il modello "preda–predatore". Quindi, partendo da quanto visto nei due capitoli precedenti, viene proposta l'architettura di calcolo per la realizzazione di un software per la soluzione approssimata dei sistemi di equazioni differenziali citati in precedenza. Vengono presentate due architetture differenti in cui la frammentazione del programma avviene secondo due distinte modalità. La prima prevede di realizzare un *server* dedicato all'esecuzione di funzioni semplici a cui vengono forniti come parametri solo pochi dati. Tali funzioni vengono però richiamate numerose volte dal *client* che si occupa di coordinare l'intera procedura di calcolo. La seconda architettura proposta è stata realizzata spostando sul

server l'esecuzione dell'intero algortitmo risolutivo: in questo caso il client richiamerà una sola volta il *metodo* disponibile sul server, ricevendo da questo una grande mole di informazioni (le coordinate dei punti che definiscono la soluzione approssimata del sistema). Sperimentalmente abbiamo potuto verificare che la prima architettura è preferibile alla seconda nel caso in cui la connessione di rete è piuttosto veloce e si dispone di una macchina client di buona potenza. La seconda architettura invece, fornisce il massimo del beneficio quando viene utilizzata su client di scarsa potenza con una connessione di rete anche lenta; in questo caso però il server dovrà essere una macchina piuttosto potente.

La nostra sperimentazione non si è limitata a questo: nella parte iniziale del nostro lavoro abbiamo infatti provveduto ad implementare gli algoritmi risolutivi utilizzando linguaggi di programmazione differenti: ad una prima versione scritta in linguaggio Pascal ne abbiamo affiancate altre due scritte rispettivamente in linguaggio C ed in linguaggio Java. Abbiamo quindi confrontato (a parità di algoritmo di calcolo) l'efficienza delle tre codifiche: il risultato, peraltro abbastanza ovvio e prevedibile, ha portato alla conclusione che la codifica in C era la più efficiente, mentre la meno efficiente era proprio quella scritta in Java. Abbiamo però deciso di trascurare questo risultato, a fronte di alcune considerazioni che ci permettono comunque di preferire il linguaggio Java, nonostante sia in certi casi meno efficiente.

Queste considerazioni sono state poi messe in evidenza dal nostro lavoro e sono riportate diffusamente in questa tesi: mediante Java siamo in grado di distribuire facilmente l'applicazione su più CPU, mentre una analoga architettura di calcolo sarebbe improponibile in linguaggio Pascal e di difficile implementazione utilizzando il C. Inoltre, come vedremo più avanti, Java ci permette di scrivere applicazioni che possono essere eseguite senza alcuna modifica su macchine completamente diverse, facendo anche uso di output grafici che rendono la nostra applicazione facile da utilizzare e di grande

impatto visivo.

In appendice sono riportati i listati in linguaggio java dei due programmi che implementano le diverse architetture di calcolo distribuito descritte nel terzo capitolo ed alcune stampe dell'output prodotto dal nostro programma.

Capitolo 1

Java ed il calcolo distribuito

Lo sviluppo del linguaggio Java da parte di Sun Microsystems è iniziato nel 1991 con l'idea di ottenere un linguaggio di programmazione per l'elettronica di consumo, ossia adatto a realizzare programmi che dovevano essere integrati negli elettrodomestici. Le finalità erano quelle di avere dei programmi di piccole dimensioni, veloci, efficienti e di facile trasferimento su diverse macchine. Inizialmente il linguaggio venne utilizzato all'interno della Sun con il nome di OAK. Nel corso del 1994 tale linguaggio venne abbinato ad un *browser* sperimentale del Web di nome HotJava. Quando Netscape decise di incorporare il supporto per l'esecuzione degli *applet* Java nel suo *web-browser* la diffusione e la popolarità di Java aumentò enormemente.

In questo capitolo analizziamo la struttura del linguaggio Java, le caratteristiche fondamentali della programmazione orientata agli oggetti e le differenze sostanziali rispetto ad altri linguaggi di programmazione. Esaminiamo inoltre le tecniche della programmazione distribuita utilizzando la tecnologia RMI.

1.1 Cos'è Java

Java è stato progettato partendo da zero; non deriva direttamente da nessun altro linguaggio di programmazione, nè può essere reso compatibile con altri linguaggi. Essendoci la libertà di progettare un linguaggio senza riferimenti ad alcun modello, è stato scelto un approccio ad oggetti pratico ed essenziale; il modello ad oggetti di Java è semplice e facilmente estensibile, mentre i numeri e altri tipi semplici vengono mantenuti come "non-oggetti" ad alte prestazioni, senza quindi inutili sovrastrutture ed appesantimenti che avrebbero potuto impattare in modo negativo sulle prestazioni. Gli è stato dato un aspetto sintattico simile al C++ per permettere ai programmatori che già utilizzano questo linguaggio di passare agevolmente a Java.

Java è nato per operare in ambiente di rete, quindi dando particolare risalto ed evidenza agli aspetti relativi alla sicurezza, alla connessione mediante protocolli di rete TCP/IP, alla portabilità su piattaforme differenti e alla possibilità di progettare applicazioni che operano in modalità *distribuita*, ossia consentendo la cooperazione tra componenti software distinte che operano su più calcolatori.

Uno degli obiettivi centrali del nostro progetto è quello di verificare operativamente la possibilità di sfruttare quest'ultima caratteristica mediante la tecnologia Java-RMI (*Remote Method Invocation*) per la realizzazione di software di calcolo numerico.

Per quanto concerne la "portabilità" del software e la possibilità di eseguire gli stessi programmi su architetture hardware e sistemi operativi diversi, Java propone una architettura innovativa basata sul concetto di *macchina virtuale* e di *bytecode*.

L'idea di fondo è quella di realizzare dei programmi che simulano comple-

tamente il comportamento di un calcolatore "astratto". Tutte le macchine oggi hanno una architettura molto simile a livello macroscopico; tuttavia è proprio a livello di dettaglio che sorgono numerosissime differenze tra un modello di calcolatore ed un altro. Tali differenze sono relative alla rappresentazione delle informazioni nella memoria della macchina, alla modalità con cui la macchina sfrutta i servizi di rete, alla modalità con cui l'utente può interagire con la macchina e con il suo sistema operativo (le cosiddette "interfacce utente"). Sun, sviluppando la *Java Virtual Machine*, ha proposto un software in grado di simulare completamente il funzionamento di un computer astratto, ma con caratteristiche sempre uguali, su più architetture hardware e software. In questo modo la JVM permetterà di simulare la medesima "macchina astratta" sia su un PC Windows, che su un Macintosh o una workstation UNIX. I programmi scritti in Java non saranno eseguiti direttamente dal computer ospite, ma dalla Java Virtual Machine: quindi tali programmi potranno girare indifferentemente su piattaforme differenti, senza che il programmatore si sia dovuto preoccupare delle caratteristiche di base di ognuna delle piattaforme su cui il programma sarà poi eseguito.

Come il linguaggio macchina è l'unico linguaggio che la CPU di un computer è in grado di comprendere ed eseguire, così il formato *bytecode* è il linguaggio macchina della Java Virtual Machine. Per ottenere un programma codificato in bytecode è necessario, come per qualsiasi altro linguaggio di programmazione, compilare il programma sorgente Java mediante il compilatore Java. Per riutilizzare su macchine diverse lo stesso programma eseguibile codificato in bytecode, non sarà necessario disporre del compilatore Java, ma soltanto della Java Virtual Machine compatibile con la nostra macchina.

L'architettura "neutrale" realizzata mediante la JVM è solo una parte di un sistema veramente portabile. Java fa fare alla portabilità del software un passo in avanti, precisando e specificando la grandezza dei tipi di dato ed il comportamento degli operatori aritmetici; i programmi sono gli stessi su ogni

piattaforma, non ci sono incompatibilità dei tipi di dato attraverso diverse architetture hardware e software.

Questa è una delle novità fondamentali della struttura Java: anche prima della nascita di Java era possibile compilare lo stesso programma su piattaforme diverse in quanto erano stati creati dei compilatori per quasi tutte le piattaforme; ciò comportava che, per far eseguire lo stesso programma su piattaforme diverse, bisognava ricompilarlo per produrre un codice in linguaggio macchina che fosse compatibile con la piattaforma. La ricompilazione del programma era possibile solo se il programma era stato scritto utilizzando un linguaggio base, senza poter sfruttare la potenza e le caratteristiche di una determinata piattaforma.

La portabilità del linguaggio Java ne determina, in parte, anche un aspetto negativo: la lentezza. Java infatti è un linguaggio interpretato dalla JVM: ogni istruzione del programma Java compilato in formato bytecode viene "tradotta" dalla JVM e successivamente eseguita; mentre le istruzioni di un programma compilato in C++, ad esempio, vengono direttamente "capite" ed eseguite sulla macchina su cui gira perché sono già state tradotte in linguaggio macchina una volta per tutte dal compilatore. La maggior parte dei linguaggi di programmazione che hanno tentato di offrire questa caratteristica (la portabilità) sono stati penalizzati sotto l'aspetto delle prestazioni. Altri sistemi neutri rispetto alla piattaforma sono anch'essi linguaggi interpretati, come il BASIC e il Perl, ed infatti anche questi linguaggi sono carenti in termini di prestazioni.

Java è stato progettato per offrire buone prestazioni anche su sistemi con processori poco potenti; se da una parte è vero che Java è un linguaggio interpretato, dall'altra il bytecode Java, più vicino alla macchina di un linguaggio di alto livello, è stato progettato con cura per essere facilmente tradotto in linguaggio macchina dalle alte prestazioni.

Java limita la libertà dei programmatori in alcune aree-chiave proprio per costringerli ad individuare gli errori fin dalle prime fasi dello sviluppo del programma. Nello stesso tempo permette di non preoccuparsi di aspetti che in altri linguaggi sono fonte di errori di programmazione, il malfunzionamento dei programmi tradizionali è dato principalmente da due motivi: una errata gestione della memoria e le condizioni impreviste di errore (le cosiddette *eccezioni*). Java elimina alla radice entrambi i problemi, con una modalità avanzata di gestione della memoria e fornendo delle istruzioni di base per il trattamento delle eccezioni. La gestione della memoria viene resa più efficiente grazie alla presenza di un componente della Java Virtual Machine denominato *garbage collector* che si occupa di liberare le aree di memoria allocate precedentemente e non più utilizzate dal programma. Al tempo stesso Java maschera la struttura della memoria al programmatore, fornendo solo dei metodi di alto livello per l'allocazione delle strutture dati: scompare quindi il concetto di puntatore tipico del linguaggio C e del C++.

Java è un linguaggio molto rigoroso per quanto riguarda i tipi di dato e le dichiarazioni: grazie a questa caratteristica la maggior parte degli errori più comuni possono essere intercettati in fase di compilazione; ciò rappresenta un notevole risparmio di tempo rispetto alle metodologie tradizionali, dove per accorgersi di malfunzionamenti bisogna eseguire il programma in tutte le sue parti.

Altre caratteristiche interessanti del linguaggio Java, alcune delle quali sono state utilizzate anche nell'ambito del nostro progetto, sono le seguenti:

- *Abstract Windows Toolkit.* Java fornisce una astrazione dell'interfaccia utente a finestre, in modo tale da distaccarsi dal particolare modello di interfaccia utente proposto dal sistema operativo Microsoft Windows, o dal System della Apple o anche dall'interfaccia grafica X11 delle work-station UNIX; questo aspetto non è di secondaria importanza, visto che

uno dei limiti principali alla portabilità del software è costituito proprio dalla impossibilità di ricondurre una particolare interfaccia utente ad un'altra.

- *Multi threading.* È la caratteristica di alcuni moderni linguaggi di programmazione e sistemi operativi, che consente di scindere il flusso di un programma in più "thread", ossia in sottoprogrammi parzialmente indipendenti, in grado di eseguire compiti differenti in parallelo.

- *Remote Method Invocation.* Come abbiamo già accennato e come descriveremo meglio in seguito, è la modalità con cui Java rende possibile la realizzazione di applicazioni distribuite su più macchine, che comunicano tra di loro (per lo scambio dei dati e la sincronizzazione dei processi) mediante il protocollo RMI.

1.2 La programmazione orientata agli oggetti

I programmi per computer sono simulazioni digitali di modelli concettuali logici ed astratti, oppure fisici. Questi modelli sono spesso complessi e l'obiettivo del programmatore è di tramutare questa complessità in una forma comprensibile, presentabile agli utenti attraverso un'interfaccia.

Comunemente siamo abituati a superare la complessità dei problemi che intendiamo risolvere attraverso l'astrazione: ad esempio non si pensa alle automobili come ad un elenco di decine di migliaia di "oscuri" componenti, ma si considera un'auto come un oggetto ben delineato, con un suo proprio ed unico comportamento. Questa nozione dell'astrazione ci permette di utilizzare la macchina senza essere sopraffatti dalla complessità di quell'algoritmo che è l'automobile: si ignorano i dettagli del funzionamento del motore, della trasmissione e dei freni, ma si considera invece una nozione astratta del comportamento dell'oggetto nel suo insieme. È utile raffinare l'abilità di

trattare efficacemente queste astrazioni in un modo gerarchico, abilità che ci permette di stratificare la semantica dei sistemi complessi. Questo modo di pensare alle astrazioni gerarchiche per i sistemi complessi del mondo reale, è applicabile anche ai programmi per computer.

Utilizzando un linguaggio di programmazione "tradizionale", come il C o il Pascal, siamo portati a definire una serie di procedure e funzioni che applicano un determinato algoritmo (secondo le ben note metodologie della programmazione strutturata e della progettazione top-down) a delle strutture dati definite spesso al di fuori della stessa procedura o funzione. Nei casi peggiori viene anche fatto uso di costanti o variabili globali definite ed istanziate al di fuori della procedura o della funzione. Questo conduce a definire una struttura del programma che difficilmente ci permetterà di riutilizzare in altri contesti, senza alcuna modifica, le funzioni e le subroutine definite nel programma stesso.

Negli ultimi anni, soprattutto in ambito industriale, dove l'esigenza del "riuso delle componenti software" è particolarmente sentita, sono state raffinate ulteriormente le metodologie di programmazione *Object Oriented* (OOP). Java è oggi, insieme al C++, il più diffuso linguaggio di programmazione ad oggetti.

L'idea alla base di questa nuova modalità di progettazione e di programmazione è quella di costruire una serie di componenti software autonome (gli *oggetti*), tali da poter essere riutilizzate con grande facilità in diversi contesti. Gli oggetti non sono altro che un insieme di strutture dati e di procedure (i *metodi*) che implementano algoritmi per operare su tali strutture. Alcuni di questi metodi saranno "pubblici" e dunque saranno visibili al programma che utilizza l'oggetto e potranno essere quindi richiamati per eseguire determinate operazioni sull'oggetto stesso (e sui dati che esso contiene). Viene meno quindi la necessità di definire variabili o costanti globali; cade anche l'esi-

genza di costruire funzioni a cui debbano essere "passate" lunghe sequenze di parametri. L'attività del programmatore si modifica: è necessario progettare tanti piccoli oggetti che potranno poi essere assemblati fra loro in un programma principale.

Naturalmente le tecniche di programmazione strutturata e top-down continuano ad essere utilizzate per la realizzazione degli algoritmi implementati dai metodi degli oggetti. I programmi algoritmici tradizionali possono essere resi più astratti negli oggetti componenti che il procedimento sta elaborando. Una sequenza di passi di un procedimento può diventare una collezione di "messaggi" tra oggetti autonomi; ognuno di questi oggetti contiene il proprio comportamento specifico. Gli oggetti, anche quelli astratti, vengono trattati come entità reali e concrete che rispondono ai messaggi che dicono loro di "fare qualcosa".

Questa è l'essenza della programmazione orientata agli oggetti. Dotiamo gli oggetti di una "personalità" e contiamo che si comportino sempre secondo le loro specifiche ben definite. L'autoradio non provocherà mai l'accelerazione dell'automobile, mentre schiacciando il pedale del freno non si azioneranno i tergicristalli; la complessità nella vita reale viene gestita in questo modo e le stesse regole valgono per il software. I concetti della programmazione orientata agli oggetti formano il cuore di Java proprio come nel nostro modo di ragionare costituiscono le basi per la comprensione, è importante capire come questi concetti della metodologia orientata agli oggetti si traducono nella programmazione.

I linguaggi orientati agli oggetti forniscono meccanismi che rinforzano il modello *object-oriented*, mettendo a disposizione del programmatore degli strumenti per evitare a priori i problemi tipici della programmazione tradizionale. I meccanismi fondamentali sono conosciuti come: *incapsulamento*, *ereditarietà* e *polimorfismo*.

1.2.1 Incapsulamento

Tutti i programmi consistono di due cose: istruzioni e dati. Nel modello di programmazione tradizionale, i dati vengono collocati nella memoria e manipolati dalle istruzioni contenute in subroutine o funzioni. La chiave della programmazione orientata agli oggetti è l'incapsulamento del codice che manipola i dati con la dichiarazione e la memorizzazione di quei dati. Si può pensare all'incapsulamento come un "involucro" che avvolge sia le istruzioni, che i dati che si stanno manipolando; questo involucro definisce il comportamento e "protegge" le istruzioni ed i dati da accessi arbitrari da parte di un altro programma. Per rapportare questo all'esempio pratico dell'automobile, la trasmissione automatica dell'auto incapsula milioni di bit di informazioni riguardanti il motore, come il grado di accelerazione, la pendenza della salita e la leva del cambio. Il guidatore può influenzare questo complesso incapsulamento solo in un modo: attraverso la leva del cambio; non si può quindi agire sula trasmissione azionando la freccia o il tergicristallo. Il pregio del codice incapsulato è che ognuno sa come accedervi, quindi può utilizzarlo senza preoccuparsi dei dettagli dell'implementazione. In Java la base dell'incapsulamento è una *classe*. Si crea una classe che rappresenta un'astrazione per un gruppo di oggetti che hanno in comune la stessa struttura e lo stesso comportamento. Un *oggetto* è una singola istanza di una classe, che mantiene la struttura ed il comportamento definiti dalla classe, come se fosse stato ricavato da uno stampo con la sagoma della classe. A volte ci si riferisce a questi oggetti come *istanze di una classe*. La struttura individuale o la rappresentazione dei dati di una classe, vengono definite da una serie di *variabili di istanza*. Queste variabili contengono lo stato dinamico di ogni istanza di una classe. Il comportamento e l'interfaccia di una classe sono definiti dai metodi che operano su quei dati di istanza. Un *metodo* è un messaggio che dice di fare qualcosa su un oggetto; questi messaggi appaiono molto simili alle chiamate di subroutine presenti nei linguaggi procedurali. Siccome lo

scopo è incapsulare la complessità, ci sono dei meccanismi per nascondere la complessità dell'implementazione all'interno della classe. Ogni metodo della classe può essere contrassegnato come "privato" o "pubblico". L'interfaccia *pubblica* di una classe rappresenta ogni cosa che gli utenti esterni della classe devono conoscere o, potrebbero conoscere. È possibile dichiarare metodi *privati* e istanze, dati che non possono essere accessibili ad ogni altro codice al di fuori dell'implementazione di quella classe. L'interfaccia pubblica deve essere selezionata con prudenza per non esporre troppo i meccanismi interni di una classe.

1.2.2 Ereditarietà

In genere, nell'esperienza di tutti i giorni, consideriamo la realtà che ci circonda come un insieme di "oggetti" che sono messi in relazione tra di loro in un modo gerarchico, ad esempio: animali → mammiferi → cani. Se si vogliono descrivere gli animali in modo astratto, si potrebbe dire che essi hanno attributi come le dimensioni, l'intelligenza e il tipo di struttura scheletrica. Gli animali hanno anche certi aspetti comportamentali: mangiano, respirano e dormono. Questa descrizione della struttura e del comportamento è la defnizione della *classe* per gli animali. Se si desidera descrivere una classe di animali più specifica, come i mammiferi, questi dovrebbero avere più attributi specifici, come il tipo di dentatura e il periodo di gestazione. Questa è conosciuta come una *sottoclasse* di animali, mentre gli animali sono considerati come una *superclasse* dei mammiferi. Siccome i mammiferi non sono altro che animali specificati con maggiore precisione, ereditano tutti i loro attributi dagli animali. Una sottoclasse con molti livelli di eredità, riporta gli attributi di ognuno dei suoi antenati nella *gerarchia* di classe.

Nella programmazione orientata agli oggetti l'ereditarietà interagisce anche con l'incapsulamento: se una certa classe incapsula alcuni attributi, qualche

sottoclasse avrà gli stessi attributi *più* qualcuno che si aggiunge come parte della sua specializzazione. Questo è un concetto chiave che permette ai programmi *object oriented* di crescere di complessità con una progressione lineare piuttosto che geometrica. Una nuova sottoclasse include tutto il comportamento e le specifiche di tutti i suoi antenati; non ha interazioni imprevedibili con la maggior parte degli altri programmi presenti nel sistema.

1.2.3 Polimorfismo

I metodi riguardanti gli oggetti sono informazioni passate sotto forma di *parametri* al momento dell'invocazione del metodo. Questi parametri rappresentano i valori di input verso una funzione che il metodo deve eseguire. Per eseguire due compiti differenti, nei linguaggi di programmazione funzionale, è necessario avere due funzioni separate con nomi differenti. Il *polimorfismo*, intendendo un solo oggetto con molte forme, è un concetto semplice che permette ad un metodo di avere implementazioni multiple, selezionate basandosi sul tipo di oggetto passato nell'invocazione del metodo. Ciò viene detto *overloading* del metodo. L'overloading dei metodi è un modo per una classe singola di trattare con tipi diversi in modo uniforme. È statico perché l'implementatore di una classe ha la necessità di conoscere in anticipo tutti i tipi che incontrerà prima di scrivere i metodi. In alcuni casi ciò è desiderabile e porta a istruzioni di programma che sono pulite e dal comportamento prevedibile; è comunque poco flessibile, perché spesso si vorrebbe estendere un ambiente dopo che la maggior parte del programma sorgente è stato congelato o è andato smarrito. Il ricorso alle sottoclassi permette di sfruttare un polimorfismo più dinamico in fase di *run-time*.

L'esempio dell'automobile riassume la potenza del progetto orientato agli oggetti: tutti gli automobilisti fanno affidamento sull'ereditarietà per guidare tipi differenti (sottoclassi) di veicoli; ognuno di noi può individuare o far fun-

zionare il volante, i freni e l'acceleratore, a prescindere dal fatto che il veicolo sia un camion, una berlina o una Ferrari. Dopo qualche "grattata" con il cambio, si può anche immaginare la differenza tra un cambio manuale ed uno automatico perché si comprende la loro superclasse comune, la trasmissione. Sulle automobili si interfacciano caratteristiche incapsulate in ogni momento. I pedali del freno e dell'acceleratore nascondono un incredibile ordinamento di complessità, con un'interfaccia così semplice che sembra li faccia funzionare con il solo utilizzo dei piedi. L'attributo finale, il polimorfismo, è rispecchiato nell'abilità dei costruttori di automobili di offrire un ampia gamma di opzioni su quello che fondamentalmente è lo stesso veicolo; ognuna di queste opzioni rappresenta un modo differente di interfacciarsi ottenendo lo stesso risultato finale.

1.3 Java ed il C++

Il linguaggio Java mostra una chiara simiglianza con il C++, tuttavia ne differisce in molti aspetti sostanziali. Il C++ è stato preso come punto di riferimento grazie alla sua popolarità, ed in questo modo Sun ha ridotto le difficoltà nell'apprendimento del suo nuovo linguaggio: i programmatori che conoscono il C++ devono faticare poco per imparare la sintassi di Java; non bisogna però pensare che Java è solo un "C++ migliorato". Prima di tutto Java non è solo un linguaggio, ma è un intero ambiente di sviluppo. Volendo comunque analizzare Java soltanto come un linguaggio, Java e C++ non sono due entità intercambiabili; anche se Java consente lo sviluppo di applicazioni molto complesse, non può sostituire, attualmente, il C++ (e Sun non lo ha inventato per questo scopo). Java non permette programmazione di sistema a basso livello, nè manipolazione diretta dell'I/O (per fare queste cose Java richiede l'intervento del C). Non è possibile attualmente scrivere un sistema operativo come Unix o Windows in Java puro, mentre la stragrande

maggioranza dei sistemi operativi esistenti sono scritti in C/C++. Il C/C++ mantiene uno stretto contatto con l'hardware su cui gira, mentre Java crea volutamente un livello di astrazione intermedio.

Java e la Java Virtual Machine costituiscono certamente un progetto più omogeneo, elegante e moderno del C++. Mentre il C++ è sostanzialmente un C con il supporto della *Programmazione Orientata agli Oggetti (OOP)*, ma conserva tutte le caratteristiche originali del C, Java è pienamente orientato agli oggetti. A parte i tipi numerici primitivi, tutti i tipi di Java sono oggetti e viene sempre garantita la discendenza da un antenato comune, la classe Object (in C++ si potevano avere più radici di gerarchie indipendenti). Non esistono neanche più le funzioni, almeno formalmente (esistono le funzioni matematiche come `sin()` che non sono altro che metodi statici della classe `Math`), ma solo metodi di classi; di fatto questo è un modo elegante per estendere anche alle funzioni il paradigma delle gerarchie di nomi. I tipi primitivi invece non sono oggetti, proprio come in C++. La novità di Java è che il numero di bit di ciascun tipo primitivo è indipendente dalla piattaforma hardware. Così, per esempio, per memorizzare un `int` (un dato di tipo intero) vengono sempre allocati 32 bit, per uno `short` (un intero "corto") sempre 16, e così via. In C/C++, invece, la dimensione in byte di variabili di tipo `int` dipende strettamente dall'architettura della CPU. Non esiste il tipo di dato inetro senza segno (in C il tipo `unsigned`) e l'intero a 64 bit (in C il tipo `long`) coincide con l'intero standard. Una curiosità è data dai caratteri, che sono a 16 bit anziché i consueti 8. Questi bit extra permettono la codifica UNICODE che supporta praticamente tutti gli alfabeti nazionali del mondo; i *floating point* (da 32 e 64 bit) sono sempre conformi allo standard IEEE e non esistono più specifiche lasciate libere all'implementazione; tutte le variabili vengono sempre inizializzate a zero in modo automatico, a meno che non si specifichi un valore alternativo nel momento in cui vengono definite. L'inizializzazione si può fare dovunque sia consentito dichiarare una variabile, quindi anche

dentro la dichiariazione di una classe. Tutte queste caratteristiche conducono ad una perfetta portabilità del codice.

Capitolo 2

Calcolo distribuito in Java

Abbiamo accennato nel capitolo precedente al fatto che con Java possiamo facilmente progettare una applicazione che sfrutti a fondo la rete a cui è collegata la macchina, per poter distribuire su altri computer porzioni dello stesso programma. La tecnica principale per far questo è costituita da RMI (*Remote Method Invocation*), con cui Java rende possibile la realizzazione di applicazioni distribuite. Nella prima parte di questo capitolo vedremo quali componenti entrano a far parte dell'architettura RMI e quale è il suo funzionamento.

Un altra tecnica con cui Java ci consente di eseguire su calcolatori differenti uno stesso programma è costituita dalle *applet*: piccoli programmi integrati nelle pagine Web che vengono automaticamente trasferiti ed eseguiti sulla macchina dell'utente. Nella seconda parte di questo capitolo descriveremo la modalità con cui può essere realizzata un'applet. Entrambi questi aspetti tecnologici, tipici del linguaggio Java e non immediatamente riscontrabili in altri linguaggi di programmazione, sono stati utilizzati nella realizzazione del progetto software descritto nel terzo capitolo.

2.1 L'architettura RMI

RMI consente di definire delle classi di oggetti che potranno essere istanziate su una macchina differente da quella che sta eseguendo il programma: in questo modo si viene a creare una relazione di tipo client/server tra un computer (che chiameremo *client*) che si occupa di eseguire il programma "principale", eventualmente gestendo anche l'interfaccia utente, ed un altro computer (che chiameremo *server*) che avrà il compito di gestire degli oggetti e quindi di eseguire i metodi che operano su di essi. La relazione client/server non deve necessariamente essere "esclusiva": uno stesso server può fornire servizi a più di un client e al tempo stesso, un client può utilizzare oggetti differenti messi a disposizione da più server.

Questo modello di architettura distribuita è estremamente utile nel caso in cui si intenda specializzare determinati server per l'erogazione di specifici servizi di calcolo. Al tempo stesso se ne può trarre vantaggio quando le macchine client a disposizione degli utenti hanno una limitata potenza di calcolo.

In maggiore dettaglio possiamo dire che l'architettura RMI richiede la presenza di tre componenti fondamentali:

- RMI Server: è l'applicazione Java che mette a disposizione gli oggetti e i metodi remoti;
- RMI Registry: è un componente del server RMI che fornisce informazioni ai client relativamente ai metodi remoti pubblicati dal server ed alla modalità con cui tali metodi devono essere invocati;
- RMI client: è il programma che richiama ed esegue i metodi remoti disponibili sul server.

Nel momento in cui si potesse disporre di una libreria di algoritmi di calcolo ben strutturata ed implementata attraverso oggetti e metodi istanziati su un server RMI, realizzare applicazioni di cacolo scientifico in Java risulterebbe essere un compito meno oneroso, visto che dovremmo limitarci a costruire l'applicazione client che utilizza i metodi generici presenti sul server (metodi per l'esecuzione di operazioni standard sulle matrici – prodotti, inversioni, ecc. –, metodi per il calcolo di determinate funzioni, ecc.).

In fig. 2.1 è riportata una schematizzazione delle componenti dell'architettura RMI e dei flussi di comunicazione esistenti.

Il server è costituito, come abbiamo visto, da due componenti che operano parallelamente: RMI registry ed RMI server. Il primo è un componente che gira su ogni computer su cui è attivo un server RMI e si occupa di mantenere la lista degli oggetti che sono attivi su quel computer. Tramite questo componente, il client è in grado di ottenere, da ciascun computer che esegue un server RMI, l'elenco degli oggetti che sono disponibili su quella macchina.

Una delle caratteristiche peculiari di RMI è infatti la possibilità di scaricare dinamicamente il codice delle classi che non sono già installate sulla macchina chiamante, rendendo possibile l'estensione delle funzionalità dell'applicazione in modo dinamico. Un'altra caratteristica fondamentale di RMI è la possibilità di attivare gli oggetti remoti in funzione di una richiesta del client. L'attivazione remota comporta che, se un processo server non è in esecuzione, ma un client ne richiede i servizi, RMI è in grado di lanciare l'esecuzione del server in modo automatico.

Sul client operano le seguenti componenti:

1. il sistema operativo della macchina (Windows, Linux, Mac OS, ecc.);

Figura 2.1: Architettura RMI

2. la Java Virtual Machine relativa al sistema operativo della macchina;
3. il programma Java compilato in formato bytecode.

Su ogni server sono presenti le seguenti componenti software:

1. il sistema operativo della macchina (Windows, Linux, Mac OS, ecc.);
2. la Java Virtual Machine relativa al sistema operativo del server;
3. il programma RMI Registry;

4. il programma RMI Server, configurato per attendere connessioni da parte dei client su una determinata porta del protocollo di rete TCP/IP.

RMI Server esegue le classi Java che implementano gli oggetti pubblicati attraverso RMI; al tempo stesso l'applicazione che opera sul client è stata compilata includendo anche una classe che espone l'interfaccia dei metodi remoti; in altre parole l'applicazione client deve conoscere a priori la modalità di chiamata dei metodi remoti.

Il flusso di comunicazione tra il client ed il server RMI prevede i seguenti passi, che sono resi del tutto trasparenti al programmatore: egli non dovrà occuparsi di eseguirli perché è la stessa invocazione dei metodi remoti mediante RMI che produce l'attivazione delle seguenti operazioni da parte della Java Virtual Machine del client:

1. il programma Java (client) invoca un metodo che è stato dichiarato "remoto";
2. la Java Virtual Machine apre una connessione (grazie ai servizi di rete offerti dal sistema operativo ospite) con la macchina che ospita il server RMI; per far questo utilizza il protocollo di comunicazione TCP/IP ed una "porta" inutilizzata da altre applicazioni;
3. una volta stabilita una connessione con il server RMI, viene interrogato RMI Registry per ottenere i riferimenti interni relativi all'oggetto e al metodo invocato dal programma client;
4. con i riferimenti così ottenuti viene finalmente invocato il metodo sul server RMI;
5. la Java Virtual Machine del server RMI istanzia l'oggetto ed esegue il metodo su di esso, quindi restituisce i riferimenti all'oggetto istanziato o i dati di output del metodo invocato;

6. la Java Virtual Machine del client riceve l'output dal server e lo restituisce al programma client che lo utilizzerà opportunamente.

2.2 Oggetti Remoti

Lo sviluppo in RMI avviene per interfacce. Ogni oggetto remoto definito in una applicazione distribuita, deve essere specificato tramite una interfaccia che rispetti alcune regole. L'interfaccia contiene i metodi che possono essere invocati dal client. Questa costituisce una specie di contratto tra il server ed il client. Gli oggetti remoti che implementano queste interfacce possono contenere, oltre ai metodi che implementato l'interfaccia, anche metodi aggiuntivi di supporto. Ovviamente, questi non potranno essere utilizzati dal client in quanto non presenti nell'interfaccia remota. Questa deve:

- estendere `java.rmi.Remote`;
- ogni metodo deve lanciare l'eccezione `java.rmi.RemoteException` oltre alle eccezioni specifiche delle applicazioni.

L'eccezione `RemoteException` serve ad indicare eventuali errori di comunicazione di rete al client. Questo potrà intercettare questa eccezione e gestire eventualmente il "recupero" dell'applicazione.

Come detto, RMI ha la capacità di trasferire oggetti tramite la rete. Questa funzionalità discrimina tra gli oggetti cosiddetti "normali" e gli oggetti remoti, ossia quelli che implementano l'interfaccia `Remote`. I primi vengono serializzati e passati sul canale di comunicazione alla Java Virtual Machine chiamante. Gli oggetti remoti non vengono passati: al loro posto viene inviato il loro *stub*, una specie di *proxy* locale. Questo significa che il ritornare o passare oggetti a metodi remoti comporta il passaggio di soli riferimenti e non di oggetti reali.

Vediamo quali sono i passi da compiere per costruire un'applicazione basata su RMI; in questo caso, a puro titolo di esempio, presentiamo un programmino per il calcolo della media aritmetica fra due numeri letti in input: la lettura e la stampa del risultato avvengono sul client, mentre il calcolo viene eseguito sul server mediante un oggetto remoto invocato con RMI.

2.2.1 Definizione dell'oggetto server

Abbiamo già detto che la classe dell'oggetto remoto deve implementare una interfaccia che estende l'interfaccia `java.rmi.Remote`. Questa interfaccia deve definire i metodi che potranno essere invocati per operare sull'oggetto remoto; ogni metodo deve gestire l'eccezione `java.rmi.RemoteException`. La classe dell'oggetto remoto deve anche estendere la classe `RemoteServer` (o una sua classe derivata, tipicamente `UnicastRemotObject`).

Il programma lato server si articola su tre file distinti: il primo, che chiameremo *implementazione dell'oggetto*, costituisce il corpo del programma; è su questo file che viene codificato l'algoritmo risolutore. Il secondo file, che chiameremo *interfaccia dell'oggetto*, serve esclusivamente a definire la modalità con cui dovranno essere invocati i metodi (sostanzialmente il tipo di dato restituito dai metodi ed i parametri da fornire). Il terzo file definisce il server RMI per l'oggetto remoto contenuto nella classe, ossia la modalità con cui il server RMI si collega al Registry.

Di seguito riportiamo il programma relativo all'implementazione dell'oggetto `MediaRmi`.

```
import java.io.*;

public class MediaRmiObjectImpl extends
  java.rmi.server.UnicastRemoteObject implements MediaRmiObject {
```

```
  public MediaRmiObjectImpl() throws java.rmi.RemoteException {
    super();
  }

  public float media(int A, int B) {
    return (float)((A+B)/2);
  }
}
```

Il metodo `media` riceve come argomento due numeri interi e restituisce un numero *floating point*. È un metodo pubblico (la sua definizione è preceduta dalla parola chiave `public`) e quindi potrà essere invocato da classi esterne a quella in cui è stato definito.

Di seguito riportiamo il codice Java dell'interfaccia dell'oggetto. Come abbiamo già anticipato è sufficiente definire la modalità con cui dovrà essere invocato il metodo (in linguaggio C si direbbe il *prototipo* della funzione) e non la sua implementazione, ossia le istruzioni di programma che ne definiscono il comportamento.

```
import java.rmi.*;

public interface MediaRmiObject extends java.rmi.Remote {
  public float media(int A, int B) throws
    java.rmi.RemoteException;
}
```

Infine riportiamo il programma che definisce il server dell'oggetto `MediaRmi`.

```
import java.rmi.Naming;

public class MediaRmiServer {
  public static void main(String args[]) {
    try {
      MediaRmiObjectImpl  oggetto = new MediaRmiObjectImpl();
      Naming.rebind("rmi://"+args[0]+"/MediaRmiService", oggetto);
    }
```

```
    catch (Exception e) {
      System.out.println("Trouble: " + e);
      System.exit(0);
    }
  }
}
```

Il codice presente nel metodo `main` non deve essere considerato implementazione dell'oggetto remoto. Il suo scopo è quello di collegare l'oggetto remoto al Registry in funzione sulla macchina, in modo da comunicarne la disponibilità ad eventuali client. L'operazione viene eseguita tramite il metodo `rebind` dell'oggetto `Naming`.

2.2.2 Programmazione del client RMI

Una volta che il server RMI è in esecuzione, è possibile scrivere dei client che utilizzino gli oggetti remoti contenuti.

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.net.MalformedURLException;
import java.rmi.NotBoundException;

public class MediaRmiClient {
  public static void main(String[] args) {
    try {
      MediaRmiObject m = (MediaRmiObject)
      Naming.lookup("rmi://"+args[0]+"/MediaRmiService");
      int X = Integer.parseInt(args[1]);
      int Y = Integer.parseInt(args[2]);
      float media = m.media(X, Y);
      System.out.println("media="+media);
    }
    catch (MalformedURLException murle) {
      System.out.println();
      System.out.println("MalformedURLException");
      System.out.println(murle);
    }
```

```
    catch (RemoteException re) {
      System.out.println();
      System.out.println("RemoteException");
      System.out.println(re);
    }
    catch (NotBoundException nbe) {
      System.out.println();
      System.out.println("NotBoundException");
      System.out.println(nbe);
    }
    catch (java.lang.Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

Il client è tutto contenuto nel metodo `main` della classe `MediaRmiClient`. Vengono eseguite le seguenti operazioni: per prima cosa viene contattato il Registry tramite l'oggetto `Naming` e viene richiesto l'oggetto remoto. Una volta ottenuto un riferimento all'oggetto remoto, viene invocato il metodo dell'interfaccia remota e l'applicazione client presenterà l'output sul video.

2.2.3 Compilazione del codice

Per compilare il server ed il client in ambiente Windows bisogna procedere attraverso i seguenti passi:

1. compilazione dell'interfaccia:

```
javac MediaRmiObject.java
```

2. compilazione dell'oggetto remoto:

```
javac MediaRmiObjectImpl.java
```

3. compilazione dell'implementazione dell'oggetto remoto:

```
javac MediaRmiServer.java
```

4. creazione dei file di supporto per RMI:

```
rmic MediaRmiObjectImpl
```

5. compilazione del client:

```
javac MediaRmiClient.java
```

L'oggetto remoto `MediaRmi` compilato con il programma `rmic` (*RMI Compiler*) sul server, consente di produrre due classi (denominate *stub* e *skeleton*) indispensabili per l'esecuzione del processo RMI sia sul client che sul server. La classe *stub* emula l'oggetto remoto sulla JVM del client: a questo oggetto il client invia richieste "locali" che lo *stub* si occupa di inoltrare al server, o meglio, all'oggetto *skeleton* che si trova sulla stessa JVM del'oggetto server. La comunicazione tra *stub* e *skeleton* è invisibile agli oggetti client e server, viene eseguita dallo strato software RMI della Java Virtual Machine.

2.2.4 Esecuzione del codice

Anche l'esecuzione del programma, vista l'architettura "distribuita" dell'applicazione, richiede alcuni step aggiuntivi rispetto a quanto è necessario fare con un programma tradizionale (*stand alone*). In particolare sarà necessario eseguire le seguenti operazioni:

1. attivare il registry RMI sul server con il comando

   ```
   rmiregistry 1050
   ```

   dove 1050 è il numero identificativo della porta TCP/IP su cui RMI Registry attenderà delle richieste;

2. eseguire il server RMI con il comando

```
java MediaRmiServer servername:1050
```

   anche in questo caso 1050 indica la porta su cui avviene il colloquio in modalità RMI, mentre "`servername`" è il nome della macchina server su cui viene pubblicato l'oggetto remoto;

3. eseguire l'applicazione client con il comando

```
java MediaRmiClient servername:1050 24 257
```

   dove "`servername`" è il nome della macchina che pubblica l'oggetto server, 1050 è la porta TCP/IP ed i due numeri interi 24 e 257 sono i dati di input su cui vogliamo eseguire il calcolo della media aritmetica (in questo particolare esempio).

Il client, naturalmente, può essere attivato su una macchina fisicamente distinta da quella che ospita il server.

2.3 Le applet

Il World Wide Web (sinteticamente WWW o Web) è costituito da una serie di risorse (documenti ed applicazioni) disponibili sulla rete Internet ed accessibili mediante un protocollo di comunicazione denominato HTTP[1] che consente di prelevare e visualizzare informazioni codificate mediante la sintassi prevista dal linguaggio HTML[2]. Per produrre un output facilmente comprensibile ed impaginato opportunamente, si utilizza un interprete del linguaggio HTML costituito generalmente da un programma denominato *web browser* (Microsoft Internet Explorer o Netscape Navigator). I moderni web browser

[1] HTTP è l'acronimo di *HyperText Transfer Protocol*.
[2] HTML è l'acronimo di *HyperText Markup Language*.

contengono una Java Virtual Machine, in grado quindi di eseguire programmi Java.

Un'applet è un programma (letteralmente *applet* significa piccola applicazione) che per poter funzionare deve essere inglobata all'interno di un documento codificato in HTML. Sarà il *browser* (o meglio, la Java Virtual Machine contenuta al suo interno) ad interpretare il bytecode e trasformarlo nel codice adatto per il microprocessore utilizzato dalla macchina. Per motivi inerenti la sicurezza in Internet, le applet non possono copiare, modificare, cancellare i file del sistema sul quale vengono eseguite.

Come già detto in precedenza, in ambito Java distingueremo quindi i programmi denominati *application* da quelli denominati *applet*. Le *application* sono programmi autonomi eseguibili dalla Java Virtual Machine, mentre le *applet* sono programmi che devono essere eseguiti dalla JVM di un web browser. Nell'esempio della media aritmetica abbiamo visto una *application*, mentre invece nel programma che presenteremo in questo capitolo, abbiamo utilizzato la tecnologia delle *applet*.

Se dal punto di vista utente si distinguono le applet dalle application sulla base della modalità operativa con cui devono essere eseguite, dal punto di vista del programmatore la differenza è ben più sostanziale, ma, grazie anche alla modalità *object oriented* di Java, si riduce a poche differenze tecniche individuabili per lo più nella presenza del metodo `main()` nelle application (assente invece nelle applet) e nel fatto che l'implementazione di una applet estende la classe `Applet`. Per il resto i due tipi di programma sono molto simili, tanto da poter facilmente trasformare una applet in una application e viceversa.

Le applet, essendo distribuite sulle pagine web di Internet, sono molto pericolose, perché vengono scaricate ed eseguite su milioni di computer senza poterle controllare. Di conseguenza la Sun ha disabilitato tutte quelle

funzionalità adatte alla creazione di virus.

Le cose più importanti che le applet non possono fare sono:

1. Leggere e soprattutto scrivere dati nel computer dell'utente, a meno che quest'ultimo non specifichi delle directory a cui le applet possono accedere. Comunque i browser normalmente non concedono l'accesso a nessuna directory e quindi nella progettazione di una applet non si può supporre che gli utenti abbiano abilitato tale possibilità.

2. Eseguire programmi sul computer degli utenti.

3. Comunicare con server diversi da quello su cui è stata scaricata l'applet stessa: con le funzioni di rete è possibile stabilire un contatto tra un'applet ed il server su cui è stata scaricata, per scambiare dati, ma non con altri server nella rete. Se l'applet è in una pagina di `www.sito.com`, essa non potrà comunicare con altri server.

Compiliamo ed eseguiamo un applet generica; vediamo come poter trasformare una application in una applet. Innanzitutto occorre fare in modo che la classe principale sia una figlia derivata dala classe `Applet`:

```
//
// Importo la class Applet dal package java.applet
//
import java.applet.Applet

  public class Applet1 extends Applet {
    // Questa e' un'applet
    ...
}
```

A questo punto si possono definire dei metodi fondamentali dell'applet:

```
public void init() {
```

```
  ...
}
```

Il metodo `init` è, come il metodo *main* in una application, quello che viene invocato per primo quando l'applet viene caricata nella Java Virtual Machine.

```
public void start() {
  ...
}
```

Il metodo `start` viene chiamato dopo `init` e come punto di partenza dopo che è stata bloccata l'esecuzione di un'applet. Mentre `init` viene chiamato una sola volta, la prima volta che l'applet viene caricata, `start` è chiamato ogni volta che il documento HTML in cui si trova l'applet è visualizzato a schermo. In questo modo se la pagina Web viene abbandonata e poi riaperta, l'applet inizia l'esecuzione con `start` e non più con `init`.

```
public void stop() {
  ...
}
```

Il metodo `stop` viene chiamato quando viene interrotta l'esecuzione dell'applet. È possibile utilizzare `stop` per sospendere i processi che non è necessario eseguire quando l'applet non è visibile.

```
public void destroy() {
  ...
}
```

Questo metodo, eseguito automaticamente un'unica volta dalla JVM è chiamato nel momento in cui l'applet viene eliminata completamente dalla memoria. A questo punto saranno eliminate tutte le risorse del sistema che l'applet sta utilizzando.

Quando un'applet viene attivata viene eseguito `init()`, poi `start()`, che però può essere richiamato ogni volta che l'applet "riparte" dopo una qualche pausa (se per esempio si era chiamato stop o si era visualizzata un'altra pagina, poi si torna indietro). Le istruzioni presenti in `start()` e `stop()` dunque sono eseguite quando la pagina html contenente l'applet è attivata o disattivata. Sono metodi molto usati per la gestione dei *thread*, infatti quando un'applet che esegue continuamente istruzioni (come un'animazione) non è più visibile, perché l'utente è passato ad un'altra pagina, occorre fermare le operazioni, divenute inutili: ciò si fa scrivendo le routine di interruzione di ogni operazione nel metodo `stop()`. Se l'utente torna indietro col browser e l'applet torna visibile, viene rieseguito `start()`, in cui risiederanno le routine di riabilitazione complementari a quelle definite in `stop()`. Se non si devono eseguire particolari istruzioni all'inizializzazione e all'avvio (start) o alla disattivazione (stop), questi metodi si possono omettere.

Oltre ad `init`, `start` e `stop`, la Java Virtual Machine prevede che l'applet contenga la definizione di altri metodi dal significato particolare, come ad esempio il metodo `paint`, che fa parte della classe `Graphics`, che ci fornisce una serie di oggetti e metodi per la visualizzazione delle informazioni in formato grafico. `Graphics` è a sua volta un componente della classe `awt`, che implementa il cosiddetto *Abstract Windows Toolkit* (vedi pag. 13), ossia uno strato software della JVM che fornisce una astrazione di una generica interfaccia utente grafica a finestre, dotata di tutti quegli elementi tipici di questo genere di interfacce (bottoni, menù a tendina, *dialog box*, ecc.).

```
import java.awt.Graphics; // Occorre importare la classe Graphics

public void paint(Graphics g) {
  ...
}
```

Al metodo `paint` viene passato l'oggetto "`g`" di tipo `Graphics`, che non è

altro che l'area dell'applet su cui disegneremo tramite i metodi della classe `Graphics`. Il metodo `paint` viene eseguito all'inizio, ma anche tutte le volte che l'area dell'applet viene spostata, coperta con un'altra finestra e poi riscoperta, ecc., ossia quando occorre ridisegnarla, tutta o in parte. Per fare un esempio pratico introduciamo il metodo `drawString` della classe `Graphics`, che serve a stampare stringhe di testo (tipo la funzione `printf` del C o l'istruzione `writeln` del Pascal, ma per aree grafiche). L'applet e' la seguente:

```
import java.applet.*;
import java.awt.*;

public class Applet1 extends Applet {
  public void paint(Graphics g) {

    // l'oggetto g, di tipo Graphics, e' l'area dell'applet
    // su cui possiamo scrivere, tramite il metodo drawString(),
    // ad esempio:
    // Stampa "Ciao" a partire dalle coordinate x=50, y=50

    g.drawString("Ciao.", 50, 50);
  }
}
// Fine classe Applet1
```

Una volta compilato questo sorgente abbiamo la classe, che però dovrà essere inclusa in una pagina HTML, mettendo un opportuno riferimento all'applet. Nell'applet manca il metodo `main`, per cui non si può eseguire come applicazione: se provassimo ad eseguire il programma come una application otterremmo un messaggio di errore.

Per includere una applet all'interno di un documento HTML si deve utilizzare il *tag* `<applet>`, come nel seguente esempio:

```
<HTML>
 <BODY>
```

```
  <APPLET CODE ="Applet1.class" WIDTH="160" HEIGHT="100">
  </APPLET>
 </BODY>
</HTML>
```

All'interno del tag `<applet>` l'attributo `CODE` specifica il nome della classe, e i parametri obbligatori `WIDTH` (larghezza) e `HEIGHT` (altezza) indicano la dimensione dell'area grafica utilizzata dall'applet. Se il file "`.class`" non è nella stessa directory del documento HTML, occorre specificarne il *path* (eventualmente comprendente anche l'URL) con il tag `CODEBASE`.

Ora possiamo finalmente eseguire l'applet caricando con un web browser il file `Applet1.html`, naturalmente dopo aver generato `Applet1.class` compilando `Applet1.java`.

Capitolo 3

Applicazione per il calcolo delle EDO

Nei capitoli precedenti abbiamo descritto le caratteristiche della programmazione distribuita in Java e dell'architettura RMI, specificando la costruzione, la compilazione e l'esecuzione di un programma che utilizza questa struttura.

In questo capitolo presenteremo finalmente una applicazione in grado di eseguire dei calcoli numerici classici utilizzando l'architettura distribuita ed il linguaggio Java. Nel nostro programma, che si occupa di produrre come output il grafico della traiettoria di un'equazione differenziale ordinaria, la parte del calcolo viene finalizzata ad una uscita grafica eseguita da un *applet*; ciò significa che l'utente avrà un interfaccia grafica sulla quale inserire e modificare i parametri, scegliere l'equazione ed il metodo da utilizzare per il calcolo della soluzione approssimata. Premendo poi un bottone si avrà il grafico.

L'applicazione è dunque suddivisa in due componenti: la prima si occupa dell'esecuzione dell'algoritmo che implementa il metodo selezionato dall'utente; la seconda si occupa dell'interazione con l'utente (acquisizione dei parametri di input e visualizzazione dell'output). In particolare la modalità scelta per

l'implementazione del client consiste nella realizzazione di una *applet*. Si tratta di un particolare tipo di applicazione prevista da Java, tale da garantire un elevatissimo grado di "portabilità" e di integrazione con il Web.

Il capitolo si suddivide in due parti: nella prima presentiamo il problema numerico della soluzione approssimata di sistemi di equazioni differenziali ordinarie (EDO); nella seconda parte, iniziando con una descrizione delle applet (utilizzate poi nell'implementazione del programma), presenteremo finalmente l'architettura di calcolo distribuito proposta per la soluzione del problema numerico mediante Java ed RMI.

3.1 Problema di valori iniziali

Il problema

$$\begin{cases} y' = f(t,y), \quad t \in [a,b], \quad y \in \mathbf{R} \\ y(t_0) = y_0 \end{cases} \tag{3.1}$$

costituisce quello che si chiama problema di valori iniziali o problema di Cauchy, per una equazione differenziale ordinaria del primo ordine in forma esplicita.

Si supponga che $f(t,y)$ sia continua in (a,b) e che valga la seguente diseguaglianza

$$\mid f(t,y) - f(t,y^*) \mid \leq L \mid y - y^* \mid, \qquad \forall t \in a,b \quad e \quad \forall y, y^*$$

dove L è la cosiddetta costante di Lipschitz.

Queste condizioni sono sufficienti per l'esistenza e l'unicità della soluzione $y = \Phi(t)$ in $[a,b]$. In tali circostanze ha senso porsi il problema di ricercare una approssimazione numerica della soluzione. L'integrazione di problemi di valori iniziali è alla base dello studio approssimato dei problemi riguardanti

le e.d.o. La maniera tradizionale di ottenere una approssimazione della soluzione di un problema differenziale è quella di costruire una approssimazione del problema in forma di equazioni non differenziali e di risolvere quest'ultimo con tecniche numeriche. Il fatto che l'approssimazione sia buona non garantisce di per sè che la soluzione che si ottiene sia altrettanto buona, anzi non garantisce nemmeno che una soluzione approssimata sia ottenibile: per comprendere questo fatto è necessario introdurre i concetti di *convergenza* e *stabilità numerica*.

Al fine di studiare la stabilità numerica del problema di Cauchy, consideriamo il seguente problema

$$\begin{cases} z' = f(t, z(t)) + \delta(t), \quad t \in [a, b] \\ z(t_0) = y_0 + \delta_0 \end{cases} \tag{3.2}$$

dove $\delta_0 \in \mathbf{R}$ e δ è una funzione continua in (a, b), ottenuto perturbando nel problema di Cauchy (3.1) sia il dato iniziale y_0, sia la funzione f.

Il problema di Cauchy (3.1) si dice *totalmente stabile* se, per ogni perturbazione $(\delta_0, \delta(t))$ che soddisfa

$$|\delta_0| < \varepsilon, \qquad |\delta(t)| < \varepsilon \quad \forall t \in (a, b)$$

con $\varepsilon > 0$, la soluzione z del problema perturbato (3.2) verifica la seguente proprietà:

$$\exists C > 0 : \quad |y(t) - z(t)| < C\varepsilon, \quad \forall t \in (a, b). \tag{3.3}$$

Nel caso in cui $b = \infty$, si dice che (3.1) è *asintoticamente stabile* se, oltre alla (3.3), si ha anche

$$|y(t) - z(t)| \to 0, \quad \textit{per } t \to \infty. \tag{3.4}$$

La richiesta che il problema di Cauchy sia stabile equivale a richiedere che esso

sia ben posto. L'ipotesi di Lipschitzianità della f, è sufficiente a garantire la stabilità del problema di Cauchy.

3.2 Metodi numerici ad un passo

Analizziamo l'approssimazione numerica del problema di Cauchy (3.1). Fissato l'intervallo di integrazione $I = (t_0, t_0 + T)$, sia $h > 0$ il passo di discretizzazione dell'intervallo I, che viene così suddiviso nei punti $t_n = t_0 + nh$, con $n = 0, 1, 2, ..., N_h$. N_h è il massimo intero per il quale risulti $t_{N_h} \leq t_0 + T$. Indichiamo con u_j l'approssimazione nel nodo t_j della soluzione esatta $y(t_j)$ che denoteremo con y_j. Analogamente, f_j indicherà il valore di $f(t_j, u_j)$; in generale si porrà $u_0 = y_0$.

Un metodo numerico per l'approssimazione del problema di Cauchy si dice *ad un passo* se $\forall n \geq 0$, u_{n+1} dipende solo da u_n. In caso contrario si dirà a *più passi* o *multistep*. Vediamo ora alcuni metodi ad un passo.

1. Il *metodo di Eulero in avanti* (o *Eulero esplicito*):

$$u_{n+1} = u_n + hf_n \tag{3.5}$$

2. Il *metodo di Eulero all'indietro* (o *implicito*):

$$u_{n+1} = u_n + hf_{n+1} \tag{3.6}$$

In entrambi i casi la derivata prima di y è stata approssimata con un rapporto incrementale: in avanti nel metodo (3.5), all'indietro nel metodo (3.6).

Un metodo si dice *esplicito* se u_{n+1} si ricava direttamente in funzione dei valori nei soli punti precedenti. Un metodo è *implicito* se u_{n+1} dipende implicitamente da se stessa attraverso la f.

I metodo impliciti, come Eulero all'indietro, richiedono ad ogni passo la risoluzione di un problema non lineare se f dipende in modo non lineare dal secondo argomento.

3.3 Analisi dei metodo ad un passo

Ogni metodo esplicito ad un passo per l'approssimazione di (3.1) si può scrivere nella forma compatta:

$$u_{n+1} = u_n + h\Phi(t_n, y_n, f(t_n, y_n); h), \quad 0 \leq n \leq N_h - 1, \quad u_0 = y_0 \tag{3.7}$$

dove $\Phi(\cdot, \cdot, \cdot; \cdot)$ è detta *funzione incremento*. Ponendo come al solito $y_n = y(t_n)$, in analogia alla (3.7) possiamo scrivere:

$$y_{n+1} = y_n + h\Phi(t_n, y_n, f(t_n, y_n); h) + \varepsilon_{n+1}, \quad 0 \leq n \leq N_h - 1 \tag{3.8}$$

dove ε_n è il residuo che si genera nel punto t_n avendo preteso di "far verificare" alla soluzione esatta lo schema numerico. Riscriviamo il residuo nella forma seguente

$$\varepsilon_{n+1} = h\tau_{n+1}(h)$$

La quantità $\tau_{n+1}(h)$ è detta *errore di troncamento locale* (nel nodo t_{n+1}). Definiamo allora *errore di troncamento globale* la quantità

$$\tau(h) = \max_{0 \leq n \leq N_h - 1} |\tau_{n+1}(h)|$$

Si noti che $\tau(h)$ dipende dalla funzione y, soluzione del problema di Cauchy (3.1). Il metodo di Eulero in avanti è un caso particolare della (3.7), dove si pone $\Phi(t_n, y_n, f(t_n, y_n); h) = f(t_n, y_n)$. Uno schema esplicito ad un passo, potendosi rappresentare nella forma (3.7) è completamente caratterizzato

dalla sola funzione di incremento Φ. La funzione di incremento, nei due metodi ad un passo considerati è tale che:

$$\lim_{h\to 0} \Phi(t_n, y_n, f(t_n, y_n); h) = f(t_n, y_n), \quad \forall t_n \geq t_0 \tag{3.9}$$

La proprietà (3.9), unita alla proprietà del rapporto incrementale:

$$\lim_{h\to 0} \frac{y_{n+1} - y_n}{h} = y'(t_n), \quad \forall n \geq 0$$

assicura che dalla (3.8) segua $\lim_{h\to 0} \tau_n(h) = 0$, $0 \leq n \leq N_h - 1$. A sua volta questa condizione garantisce che

$$\lim_{h\to 0} \tau(h) = 0,$$

proprietà che esprime la *consistenza* del metodo numerico (3.7) con il problema di Cauchy (3.1). In generale un metodo si dice *consistente* quando l'errore di troncamento ad esso corrispondente è infinitesimo rispetto ad h. Inoltre, uno schema ha *ordine di consistenza* p se, $\forall t \in I$, la soluzione $y(t)$ del problema di Cauchy (3.1) soddisfa la condizione

$$\tau(h) = O(h^p) \quad per \quad h \to 0. \tag{3.10}$$

I metodi di Eulero hanno ordine di consistenza uno.

3.3.1 La zero-stabilità

Il metodo numerico (3.7) per la soluzione del problema (3.1) è *zero-stabile* se

$$\exists h_0 > 0, \quad C > 0 : \quad \forall h \in (0, h_0], \ |z_n^{(h)} - u_n^{(h)}| \leq C\varepsilon, \quad 0 \leq n \leq N_h, \tag{3.11}$$

dove $N_h = \max\{n : t_n \leq t_f\}$ e $z_n^{(h)}$, $u_n^{(h)}$ sono rispettivamente le soluzioni dei problemi

$$\begin{cases} z_{n+1}^{(h)} = z_n^{(h)} + h[\Phi(t_n, z_n^{(h)}, f(t_n, z_n^{(h)}); h) + \delta_{n+1}], & 0 \leq n \leq N_h - 1, \\ z_0 = y_0 + \delta_0, \end{cases}$$

$$\begin{cases} u_{n+1}^{(h)} = u_n^{(h)} + h\Phi(t_n, u_n^{(h)}, f(t_n, u_n^{(h)}); h), & 0 \leq n \leq N_h - 1, \\ u_0 = y_0, \end{cases}$$

con $|\delta_k| \leq \varepsilon$ per ogni $0 \leq k \leq N_h$.

La zero-stabilità richiede dunque che in un intervallo limitato valga la proprietà (3.11) per ogni valore di $h \leq h_0$. Essa riguarda, in particolare, il comportamento del metodo numerico nel caso limite $h \to 0$ ed è perciò una proprietà di quest'ultimo, non del problema di Cauchy (il quale è stabile grazie alla lipschitzianità di f). La proprietà (3.11) assicura che il metodo numerico è poco sensibile alle piccole perturbazioni ed è dunque stabile.

Si osservi inoltre che la costante C nella (3.11) è indipendente da h (e dunque da N_h), ma può dipendere dall'ampiezza dell'intervallo di integrazione; infatti la (3.11) non esclude a priori che la costante C diventi tanto più grande quanto maggiore è l'ampiezza dell'intervallo. La richiesta di stabilità per il metodo numerico è suggerita dalla necessità di tenere sotto controllo gli errori che l'aritmetica finita di ogni calcolatore introduce sia nei dati iniziali, sia nella valutazione di f. Se il metodo numerico non fosse zero-stabile, tali errori renderebbero la soluzione calcolata priva di significato.

Teorema 1. (Zero-stabilità) *Consideriamo il generico metodo numerico esplicito ad un passo (3.7) per la risoluzione del problema di Cauchy (3.1). Supponiamo che la funzione di incremento* Φ *sia lipschitziana di costante* Λ *rispetto al secondo argomento, uniformenmente rispetto ad* h *e* $t_j \in [t_0, t_0 + T]$*. Allora il metodo (3.7) è zero-stabile.*

Omettiamo per brevità la dimostazione del teorema.

3.3.2 Analisi di convergenza

Un metodo si dice *convergente* se

$$\forall 0 \leq n \leq N_h - 1, \quad |u_n - y_n| \leq C(h) \tag{3.12}$$

dove $C(h)$ è un infinitesimo rispetto ad h, mentre si dice *convergente di ordine* p se $\exists C > 0$ tale che:

$$\forall 0 \leq n \leq N_h - 1, \quad |u_n - y_n| \leq Ch^p \tag{3.13}$$

ovvero se è convergente con $C(h) = Ch^p$ nella (3.12). Possiamo dimostare il seguente teorema:

Teorema 2. (Convergenza) *Nelle stesse ipotesi del teorema (1) si ha*

$$|y_n - u_n| \leq (|y_0 - u_0| + nh\tau(h))e^{nh\Lambda}, \quad 1 \leq n \leq N_h. \tag{3.14}$$

Pertanto se vale l'ipotesi di consistenza (3.9) e $|y_0 - u_0| \to 0$ *per* $h \to 0$*, allora il metodo è convergente. Inoltre se il metodo ha ordine di consistenza* p *e* $|y_0 - u_0| = O(h^p)$*, allora vale la (3.13).*

Dimostrazione. Posto $w_j = y_j - u_j$, sottraendo la (3.7) dalla (3.8) si ottiene:

$$w_{j+1} = w_j + h[\Phi(t_j, y_j, f(t_j, y_j); h) - \Phi(t_j, u_j, f(t_j, u_j); h)] + h\tau_{j+1}(h)$$

$j = 0, ..., N_h - 1$. Sommando su j si ottiene:

$$w_n \leq w_0 + h \sum_{j=0}^{(n-1)} \tau_{j+1}(h) + h \sum_{j=0}^{(n-1)} (\Phi(t_j, y_j, f(t_j, y_j); h) - (\Phi(t_j, u_j, f(t_j, u_j); h))$$

per $1 \leq n \leq N_h$, e dunque:

$$|w_n| \leq |w_0| + h \sum_{j=0}^{(n-1)} |\tau_{j+1}(h)| + h\Lambda \sum_{j=0}^{(n-1)} |w_j|, \quad 1 \leq n \leq N_h \tag{3.15}$$

dove $w_0 = y_0 - u_0$, applicando il lemma di Gronwall nel caso discreto, si ottiene la (3.14):

$$|w_n| \leq (|w_0| + nh\tau(h))e^{nh\Lambda}, \quad 1 \leq n \leq N_h$$

mentre la (3.13) segue dalle ipotesi di convergenza dei dati iniziali e dalla (3.10) osservando che $nh \leq T$.

Un metodo consistente e zero-stabile è dunque convergente; questa proprietà è nota come *teorema di equivalenza*, vale anche il viceversa:“ un metodo convergente è zero stabile”.

3.3.3 L’assoluta stabilità

Un metodo si dice *assolutamente stabile* se, per h fissato, u_n si mantiene limitata per $t_n \to +\infty$. Tale proprietà riguarda il comportamento asintotico di u_n. Consideriamo il seguente problema di Cauchy:

$$\begin{cases} y'(t) = \lambda y(t), \quad t > 0 \\ y(0) = 1 \end{cases} \tag{3.16}$$

con $\lambda \in \mathbf{C}$, la cui soluzione è $y(t) = e^{\lambda t}$. Se $\Re(\lambda) < 0$ allora $\lim_{t\to+\infty} |y(t)| = 0$. Un metodo numerico per l’approssimazione di (3.16) è *assolutamente stabile* se

$$|u_n| \to 0 \qquad \text{per} \qquad t_n \to \infty \tag{3.17}$$

essendo u_n funzione di λh, possiamo definire *regione di assoluta stabilità* del metodo numerico, la regione del piano complesso

$$\mathcal{A} \equiv \{z = h\lambda \in \mathbf{C} : \text{la (3.17) sia soddisfatta}\}. \tag{3.18}$$

$\mathcal{A}$ è l’insieme dei valori del prodotto λh per i quali il metodo numerico produce soluzioni che tendono a zero quando t_n tende all’infinito. Verifichiamo l’assoluta stabilità dei metodi di Eulero.

1. *Metodo di Eulero in avanti*: applicando la (3.5) alla (3.16) si ottiene $u_{n+1} = u_n + h\lambda u_n$ per $n \geq 0$, con $u_0 = 1$. Iterando ricorsivamente il metodo rispetto a n si ricava:

$$u_n = (1 + h\lambda)^n, \quad n \geq 0$$

da cui si ottiene che la condizione (3.18) è verificata se e soltanto se $|1 + h\lambda| < 1$, ovvero se $h\lambda$ appartiene al cerchio di raggio unitario e centro $(-1, 0)$. Tale richiesta equivale a

$$h\lambda \in \{z \in \mathbf{C} : \Re(z) < 0\} \quad \text{e} \quad 0 < h < \frac{2}{|\lambda|}. \tag{3.19}$$

2. *Metodo di Eulero indietro*: procedendo in modo analogo al caso precedente, si ha:

$$u_n = \frac{1}{(1 - h\lambda)^n}, \quad n \geq 0$$

in questo caso la proprietà di assoluta stabilità (3.17) è soddisfatta *per ogni valore* di $h\lambda$ che non appartiene al cerchio del piano complesso di centro $(1, 0)$ e raggio unitario.

Figura 3.1: Regione di assoluta stabilità per i metodi di Eulero in avanti e di Eulero all'indietro

3.4 Metodi di tipo Runge-Kutta

Nel processo di evoluzione dal metodo di Eulero in avanti (3.5), $u_{n+1} = u_n + hf_n$, $n \geq 0$ con $f_n = f(t_n, u_n)$, verso metodi di ordine più elevato del primo, i metodi Runge-Kutta si ispirano a criteri opposti. Il metodo di Eulero è lineare in u_n e f_n, richiede una sola valutazione funzionale per ogni passo temporale e guadagna accuratezza incrementando il numero dei passi. I metodi Runge-Kutta, al contrario, guadagnano accuratezza conservando la struttura ad un passo, ma sacrificando la linearità a prezzo di un aumento del numero di valutazioni funzionali per ogni passo. In questo modo è facile modificare il passo di integrazione, ma si perde la possibilità di valutare in modo semplice l'errore locale. Nella forma più generale un metodo Runge-Kutta può essere scritto nel modo seguente:

$$u_{n+1} = u_n + hF(t_n, u_n, h; f), \quad n \geq 0 \tag{3.20}$$

dove F è la funzione di incremento definita nel modo seguente

$$\begin{aligned} &F(t_n, u_n, h; f) \equiv \textstyle\sum_{i=1}^{s} b_i K_i, \\ &K_i = f(t_n + c_i h, u_n + h \textstyle\sum_{j=1}^{s} a_{ij} K_j), \quad i = 1, 2, \cdots, s \end{aligned} \tag{3.21}$$

dove s indica il numero degli *stadi* del metodo. I coefficienti $\{a_{ij}\}$, $\{c_i\}$ e $\{b_j\}$ caratterizzano completamente un metodo Runge-Kutta e vengono raccolti nella *matrice di Butcher*

$$\begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \dots & a_{1s} \\ c_2 & a_{21} & a_{22} & \dots & a_{2s} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s1} & a_{s2} & \dots & a_{ss} \\ \hline & b_1 & b_2 & \dots & b_s \end{array} \qquad \text{o} \qquad \begin{array}{c|c} c & A \\ \hline & b^T \end{array}$$

essendo $A \equiv (a_{ij}) \in \mathbf{R}^{s \times s}$, $b \equiv (b_1, \dots, b_s)^T \in \mathbf{R}^s$ e $c \equiv (c_1, \dots, c_s)^T \in \mathbf{R}^s$. Supporremo inoltre che valga la seguente condizione

$$c_i = \sum_{j=1}^{s} a_{ij} \quad i = 1, \dots, s \tag{3.22}$$

Se in A gli a_{ij} sono nulli per $j \geq i$, con $i = 1, 2, \ldots, s$, allora ogni K_i può essere calcolato esplicitamente in funzione dei soli $i-1$ coefficienti $K_1, \ldots, K_{i-1}$ già precedentemente calcolati; in tal caso lo schema viene detto *esplicito*. In caso contrario, lo schema Runge-Kutta è *implicito* ed il calcolo dei K_i richiede la risoluzione di un sistema non lineare accoppiato di dimensione s. Per quanto riguarda la *consistenza*, definiamo implicitamente l'errore di troncamento locale $\tau_{n+1}(h)$ nel nodo t_{n+1} utilizzando l'equazione del residuo:

$$h\tau_{n+1}(h) = y_{n+1} - y_n - hF(t_n, y_n, h; f), \tag{3.23}$$

essendo $y(t)$ la soluzione del problema di Cauchy (3.1). Il metodo (3.20) è allora *consistente* se $\tau(h) = \max_n |\tau_n(h)| \to 0$ per $h \to 0$. Si verifica che ciò accade se e soltanto se

$$\sum_{i=1}^{s} b_i = 1$$

Si dirà inoltre che un metodo è consistente di ordine p ($p \geq 1$) rispetto ad h se $\tau(h) = O(h^p)$ per $h \to 0$. Per quanto riguarda la *convergenza* di un metodo di Runge-Kutta si può osservare che, essendo metodi ad un passo, la consistenza implica la stabilità e quindi la convergenza. Un esempio del metodo di Runge-Kutta del 4° ordine è fornito dal seguente schema a quattro passi esplicito:

$$u_{n+1} = u_n + \tfrac{h}{6}(K_1 + 2K_2 + 2K_3 + K_4)$$

$$K_1 = f_n,$$

$$K_2 = f(t_n + \tfrac{h}{2}, u_n + \tfrac{h}{2}K_1),$$

$$K_3 = f(t_n + \tfrac{h}{2}, u_n + \tfrac{h}{2}K_2),$$

$$K_4 = f(t_{n+1}, u_n + hK_3).$$

La descrizione dei metodi per la risoluzione delle e.d.o. è stata fatta per una singola equazione differenziale del primo ordine. I programmi implementati in

Java e descritti nelle pagine seguenti, riguardano invece equazioni differenziali di ordine superiore al primo. In effetti sia la trattazione teorica che i metodi numerici descritti si estendono ad un equazione di ordine N:

$$y^{(N)}(t) = f(y(t), y'(t), y''(t), ..., y^{(N-1)}(t))$$

$$y(0) = y_0, y'(0) = y_1, ..., y^{(N-1)}(0) = y_{N-1}.$$

Questo problema può essere ricondotto ad un sistema di N equazioni del primo ordine del tipo:

$$\begin{cases} y_1'(t) = f_1(y(t)) \\ y_2'(t) = f_2(y(t)) \\ \vdots \\ y_N'(t) = f_N(y(t)) \\ y_1(0) = x_1 \\ y_2(0) = x_2 \\ \vdots \\ y_N(0) = x_N \end{cases} \tag{3.24}$$

Introducendo $N-1$ nuove funzioni incognite $z_1, ..., z_{N-1}$. Si verifica infatti che il sistema (3.24) è equivalente al sistema:

$$\begin{cases} y_1'(t) = z_1(t) \\ z_1'(t) = z_2(t) \\ \vdots \\ z_{N-2}'(t) = z_{N-1}(t) \\ z_{N-1}'(t) = f(y(t), z_1(t), z_2(t), ..., z_{N-1}(t)) \end{cases}$$

In questo modo le equazioni del secondo ordine affrontate nell'implementazione dei programmi quali l'equazione dell'oscillatore armonico, l'equazione di Van der Pol, il modello "preda-predatore", possono essere descritte da un sistema di due equazioni differenziali del primo ordine.

3.5 Esempi e algoritmi

Di seguito riportiamo alcuni esempi classici di applicazioni dei metodi di Eulero e Runge-Kutta.

Studio del movimento del pendolo senza attrito. Consideriamo il pendolo senza attrito il cui movimento è governato dal seguente sistema di e.d.o:

$$y_1''(t) = -k \sin(y_1(t))$$

per $t > 0$, dove k è una costante positiva dipendente dai parametri geometrico-meccanici del pendolo, nel nostro programma $k = 1$. Utilizzando la trasformazione (3.24) descritta nelle pagine precedenti si ottiene il sistema equivalente:

$$\begin{cases} y_1'(t) = y_2(t) \\ y_2'(t) = -k \sin(y_1(t)) \end{cases} \tag{3.25}$$

Il campo di vettori f che regola questa sistema ha le componenti

$$\begin{aligned} f_1(y_1, y_2) &= y_2 \\ f_2(y_1, y_2) &= -k \sin(y_1) \end{aligned}$$

dove $y_1(t)$ e $y_2(t)$ sono rispettivamente la posizione e la velocità angolare del pendolo; consideriamo le condizioni iniziali: $y_1(0) = \Theta_0$, $y_2(0) = 0$. Indicando con y$=(y_1, y_2)^T$ la soluzione di (3.25), quest'ultimo ammette infinite posizioni di equilibrio della forma y$=(n\pi, 0)^T$ per $n \in \mathbf{Z}$, corrispondenti alle posizioni in cui il pendolo è fermo sulla verticale. Per n pari l'equilibrio è stabile, mentre per n dispari l'equilibrio è instabile. A questa conclusione si perviene attraverso lo studio del sistema linearizzato che assume le seguenti espressioni

$$\mathbf{y}' = \mathbf{A_p y} = \begin{bmatrix} 0 & 1 \\ -K & 0 \end{bmatrix} \mathbf{y} \qquad \mathbf{y}' = \mathbf{A_d y} = \begin{bmatrix} 0 & 1 \\ K & 0 \end{bmatrix} \mathbf{y}$$

Nel caso in cui n è pari, la matrice A_p ha autovalori complessi coniugati $\lambda_{1,2} = \pm i\sqrt{K}$ ed autovettori associati y$_{1,2} = (\mp i/\sqrt{K}, 1)^T$, mentre per

n dispari, A_d ha autovalori reali ed opposti $\lambda_{1,2} = \pm\sqrt{K}$ ed autovettori $\mathrm{y}_{1,2} = (1/\sqrt{K}, \mp 1)^T$. Consideriamo due diversi insiemi di dati iniziali $\mathrm{y}^{(0)} = (\Theta_0, 0)^T$ e $\mathrm{y}^{(0)} = (\pi + \Theta_0, 0)^T$, essendo $|\Theta_0| \ll 1$. Le soluzioni del sistema linearizzato sono date rispettivamente da

$$\begin{cases} y_1(t) = \Theta_0 \cos(\sqrt{K}t) \\ y_2(t) = -\sqrt{K}\Theta_0 \sin(\sqrt{K}t) \end{cases} \qquad \begin{cases} y_1(t) = (\pi + \Theta_0) \cosh(\sqrt{K}t) \\ y_2(t) = \sqrt{K}(\pi + \Theta_0) \sinh(\sqrt{K}t) \end{cases} \tag{3.26}$$

Il primo caso verrà detto *stabile*, il secondo *instabile*.

Modello preda-predatore. Supponiamo che nello stesso ambiente convivano due sole specie di animali. Quelli della prima specie sono erbivori, mentre quelli della seconda, i *predatori*, sono carnivori e si nutrono degli animali appartenenti ala prima specie: le *prede*. Il nostro sudio è legato all'interazione tra le due specie: come varia nel tempo il numero degli animali appartenenti alle due specie? Possono le due specie convivere nello stesso habitat oppure una delle due è destinata ad estinguersi? Supponiamo, ad esempio, che il cibo della prima specie sia disponibile sul territorio senza limitazioni; nella costruzione del modello viene isolato l'aspetto che più ci interessa:l'interazione tra preda e predatore. Supponiamo che sia le prede che i predatori si riproducano con tassi di natalità costanti e si estinguano con tassi di mortalità costanti. Indichiamo con $y_1(t)$ e $y_2(t)$ rispettivamente il numero delle prede e dei predatori presenti sul territorio al tempo t. La variazione della grandezza delle due popolazioni, ovvero l'evoluzione nel tempo delle due specie è descritta allora dalle derivate rispetto al tempo delle funzioni y_1 e y_2, che indichiamo con $y_1'(t)$ e $y_2'(t)$. In assenza di predazione l'evoluzione nel tempo delle due specie è descritta dalle due equazioni differenziali:

$$y_1'(t) = ay_1(t)$$

$$y_2'(t) = by_2(t)$$

con a e b positivi. Le soluzioni di queste equazioni differenziali sono

$$y_1(t) = y_1(0)e^{at}$$

$$y_2(t) = y_2(0)e^{-bt}$$

questo vuol dire che il numero delle prede tende a crescere indefinitamente, coerentemente con l'ipotesi che il loro cibo sia illimitato, mentre il numero dei predatori decresce verso zero poiché non si è tenuto conto della attività di predazione, che è l'unica forma di sostentamento della seconda specie. Per descrivere l'attività di predazione supponiamo che il numero delle prede diminuisca di un fattore proporzionale a $y_1(t)y_2(t)$ (questo numero lo interpretiamo come il numero di "incontri" fra le due specie) e che il numero dei predatori cresca in maniera analoga. Il sistema che descrive questo tipo di interazione è dunque:

$$\begin{cases} y_1'(t) = ay_1(t) - cy_1(t)y_2(t) \\ y_2'(t) = -by_2(t) + dy_1(t)y_2(t) \end{cases} \tag{3.27}$$

descritto dal seguente campo vettoriale

$$\begin{aligned} f_1(y_1, y_2) &= ay_1 - cy_1y_2 \\ f_2(y_1, y_2) &= -by_2 + dy_1y_2 \end{aligned}$$

con c e d positivi.

La determinazione esplicita della soluzione di questo sistema non è facile, a causa del legame dato dal *termine non lineare* $y_1(t)y_2(t)$, possiamo però effettuare unaanalisi qualitativa che fornisce qualche informazione sull'evoluzione delle due specie.Osserviamo che il sistema di equazioni non lineari

$$ay_1(t) - cy_1(t)y_2(t) = 0$$

$$by_2(t) + dy_1(t)y_2(t) = 0$$

ha le due soluzioni $E_1 = (0, 0)$ ed $E_2 = (b/d, a/c)$ che sono due punti di equilibrio per il sistema (3.27).Dunque se il numero iniziale delle prede e dei

predatori è dato dalle coordinate di E_i, le due popolazioni conservano nel tempo lo stesso numero di membri. Questa situazione è però assai improbabile , quello che ci interessa è sapere se, partendo da un arbitraria situazione iniziale $(y_1(0), y_2(0))$, l'evoluzione converga o meno algi equilibri. Se si parte da un punto sull'asse y_1 le prede tenderanno a moltiplicarsi perché non ci sono predatori sul territorio. Se invece la condizione iniziale è del tipo $(0, y_2(0))$, allora i predati tenderanno ad estinguersi in assenza di prede e dunque la soluzione del sistema (3.27) convergerà verso l'equilibrio E_1. Si osserva inoltre che non appena y_1 o y_2 sono uguali a zero anche le corrispondenti derivate $y_1'(t)$ e $y_2'(t)$ si annullano: se il numero di una delle due popolazioni è nullo ad n certo tempo, rimarrà nullo per tutti i tempi successivi. Supponiamo ora, che l'evoluzione della popolazione delle prede sia influenzata anche da un fenomeno di competizione interna dovuto alla limitatezza del cibo e al sovrappopolamento; questo fenomeno è tanto più forte quanto più alto è il numero delle prede; in particolare supponiamo che sia proporzionale a y_1^2. Prendiamo quindi come modello il sistema seguente:

$$\begin{cases} y_1^{'}(t) = ay_1(t) - cy_1(t)y_2(t) - ey_1(t)^2 \\ y_2^{'}(t) = -by_2(t) + dy_1(t)y_2(t) \end{cases}$$

descritto dal seguente campo vettoriale

$$\begin{aligned} f_1(y_1, y_2) &= ay_1 - cy_1y_2 - ey_1^2 \\ f_2(y_1, y_2) &= -by_2 + dy_1y_2 \end{aligned}$$

Siano $a, b, c, d, e > 0$ dove:

- a rappresenta il tasso di natalità della preda.
- d rappresenta il tasso di mortalità del predatore.
- c rappresenta il *termine collisione*: ogni volta che una preda ed un predatore si incontrano diminuiscono le prede ed aumentano i predatori.

- e rappresenta il *termine di attrito* nella popolazione delle prede; relativo alla collisione sulle fonti di nutrimento.

Equazione di Van der Pol. Prendiamo in considerazione l'equazione di Van der Pol poiché rappresenta lo studio di un' equazione differenziale ordinaria non lineare ed il primo esempio storico di un'orbita periodica intorno ad un equilibrio che è l'origine.

È un caso particolare dell'equazione dei circuiti elettrici di Lienard descritta dal seguente sistema:

$$\begin{cases} y_1^{'}(t) = y_2(t) - f(y_1(t)) \\ y_2^{'}(t) = -y_1(t) \end{cases} \tag{3.28}$$

dove $f : \mathbf{R}^n \to \mathbf{R}^n$, se assumiamo $f(y_1(t)) = y_1(t)^3 - y_1(t)$ otteniamo proprio l'equazione di Van der Pol descritta dal seguente campo vettoriale:

$$\begin{array}{l} f_1(y_1, y_2) = y_2 - y_1^3 + y_1 \\ f_2(y_1, y_2) = -y_1 \end{array} \tag{3.29}$$

Analizziamo le soluzioni dell'equazione di Lienard prima per il caso in cui f è lineare e dunque:$f(y_1(t) = Ky_1(y)$, $K > 0$. Allora (3.28) assume la forma:

$$\mathbf{y}' = \mathbf{A}\mathbf{y}, \qquad \mathbf{A} = \begin{bmatrix} -K & 1 \\ -1 & 0 \end{bmatrix}, \qquad \mathbf{y} = (\mathbf{y_1}, \mathbf{y_2})$$

Gli autovalori di A sono dati da:

$$\lambda = \frac{1}{2}[-K \pm (K^2 - 4)^{1/2}]$$

poiché spesso λ ha la parte reale negativa, allora l'origine $(0, 0)$ è un punto di equlibrio stabile cioè un *pozzo*. Ogni condizione tende a zero; dal punto di vista fisico questo è dovuto agli effetti dissipativi della resistenza del circuito. Quando $K < 2$ l'origine è un pozzo a spirale. Se consideriamo una generica funzione continua $f \in C^1$, esiste un unico equilibrio E_1 del sistema (3.28) ottenuto ponendo:

$$y_2 - f(y_1) = 0$$

$$-y_1 = 0$$

oppure

$$E_1 = (0, f(0)).$$

La matrice delle derivate prime parziali di (3.28) in E_1 è

$$\begin{bmatrix} -f'(0) & 1 \\ -1 & 0 \end{bmatrix}$$

e gli autovalori sono dati da

$$\lambda = \frac{1}{2}[-f'(0) \pm (f'(0)^2 - 4)^{1/2}]$$

Possiamo caratterizzare l'equilibrio E_1 in questo modo:

- E_1 è un *pozzo* se $f'(0) > 0$
- E_1 è una *sorgente se* $f'(0) < 0$

Nel caso particolare dell'equazione di Van der Pol ($f(y_1) = y_1^3 - y_1$) l'unico equilibrio è una sorgente; abbiamo inoltre il seguente

Teorema 3. *Esiste una soluzione periodica non banale dell'equazione di Van der Pol ed ogni soluzione di "non equilibrio' tende a questa soluzione periodica. "Il sistema oscilla".*

Per quanto abbiamo visto sopra, il sistema

$$\begin{cases} y_1'(t) = y_2(t) - y_1(t)^3 + y_1(t) \\ y_2'(t) = -y_1(t) \end{cases} \tag{3.30}$$

ha un unico punto di equlibrio nell'origine $(0, 0)$ che è una sorgente, si dimostra inoltre che ogni soluzione di non equlibrio "ruota" intorno all'equilibrio in senso orario.

Algoritmo per il metodo di Eulero esplicito. Per implementare il metodo di Eulero occorre assegnare in input i seguenti dati:

1. f la velocità, nel nostro caso i campi vettoriali $f_1(y_1, y_2); f_2(y_1, y_2)$;
2. y il dato iniziale, nel nostro caso $y_1(0), y_2(0)$;
3. h il passo di integrazione;
4. N il numero di iterazioni.

L'algoritmo procede secondo i seguenti passi:

1. Considerata la matrice $w_{i,j}$ con $1 \leq i,\ j \leq N$, pone:

$$w_{0,1} = y_1(0)$$

$$w_{1,1} = y_2(0)$$

2. Calcola $y(n+1) = y(n) + h * f(y(n))$
3. Se $n < N$ sostituisce $y(n+1)$ a $y(n)$ e ritorna al passo 2 altrimenti si arresta.

Algoritmo di Runge-Kutta. Per implementare il metodo di Runge-Kutta, assegnamo in input gli stessi dati del metodo di Eulero:

1. f la velocità $(f_1(y_1, y_2); f_2(y_1, y_2))$;
2. y il dato iniziale $(y_1(0), y_2(0))$;
3. h il passo di integrazione;
4. N il numero di iterazioni.

L'algoritmo del quarto ordine procede secondo i seguenti passi:

1. Considerata la matrice $w_{i,j}$ con $1 \leq i,\ j \leq N$, pone:

$$w_{0,1} = y_1(0)$$

$$w_{1,1} = y_2(0)$$

2. Calcola la funzione $g(y(n), h)$ usando la seguente formula:

$$g(y(n), h) = (k_1 + 2k_2 + 2k_3 + k_4)/6$$

con

$$\begin{aligned} k_1 &= f(y) \\ k_2 &= f(y + hk_1/2) \\ k_3 &= f(y + hk_2/2) \\ k_4 &= f(y + hk_3) \end{aligned}$$

3. Calcola $y(n+1) = y(n) + hg(y(n), h)$

4. Se $n < N$ sostituisce $y(n+1)$ a $y(n)$ e ritorna al passo 2, altrimenti si arresta.

3.6 Il programma

Nelle pagine precedenti abbiamo visto la struttura delle applicazioni RMI e la costruzione di un'applet su piccoli programmi dimostativi.

Vediamo ora come questa struttura sia stata inserita all'interno dei nostri programmi per il calcolo della soluzione approssimata di un'equazione differenziale ordinaria.

Il programma richiede in input attraverso un'interfaccia grafica (l'applet) l'inserimento dei dati che permettono il calcolo del metodo numerico; una volta inseriti i dati iniziali, il numero dei nodi su cui iterare la funzione ed il

passo; si può scegliere, attraverso un menu a tendina, l'equazione da risolvere ed l'algoritmo di calcolo.

Abbiamo sviluppato due programmi che forniscono lo stesso risultato utilizzando la tecnoclogia RMI in maniera diversa. Nel primo programma, il calcolo delle funzioni, a cui poi verranno applicati gli algoritmi di Eulero o Runge-Kutta, viene lasciato al server, mentre la lettura dei dati, il calcolo degli algoritmi e l'output grafico viene fatto sul client.

Nell'interfaccia remota `EdoRmiObject` vengono definiti i due metodi `fx` e `fy` che restituiscono un valore *floating point* da passare dal server al client. L'interfaccia remota rende disponibile l'insieme dei metodi utilizzabili per l'invocazione a distanza.

Per definire un'interfaccia remota è necessario estendere `java.rmi.Remote`: questa è un' interfaccia vuota e serve solo per verificare, durante l'esecuzione, che le operazioni di invocazione remota siano plausibili. La gestione delle eccezioni `java.rmi.RemoteException` è obbligatoria: infatti, a causa della distribuzione in rete, oltre alla gestione di eventuali problemi derivanti dalla normale esecuzione del codice, si deve anche proteggere tutta l'applicazione da anomalie derivanti dall'utilizzo di risorse remote. Ad esempio potrebbe venire a mancare improvvisamente la connessione fisica verso l'`host` dove è in esecuzione il server RMI.

```
//
// EdoRmiObject.java
//
import java.rmi.*;

public interface EdoRmiObject extends java.rmi.Remote {

  public float fx(float x, float y, int f,int met)
    throws java.rmi.RemoteException;
  public float fy(float x, float y, int f,int met)
    throws java.rmi.RemoteException;
```

```
}
```

Nella classe che implenta l'interfaccia remota `EdoRmiObjectImpl` vengono implementate le equazioni ed i due metodi che permettono la scelta di una di queste; questo metodo `EdoRmiObjectImpl`, più il costruttore che richiama `super()`, costituiscono l'implementazione dell'oggetto remoto (si definisce remoto un oggetto che implementi l'interfaccia `Rmote` ed i cui metodi possano essere eseguiti da un'applicazione client non residente sulla stessa macchina virtuale. La classe `EdoRmiObjectImpl extends UnicastRemoteObject` che contiene l'implementazione di base di un oggetto server RMI. Il termine *extends* significa ereditare o derivare dalla classe `UnicastRemoteObject`. Ogni classe in Java deriva da qualche altra classe e ogni classe in cui non sia definito esplicitamente un padre, eredita automaticamente dalla classe `Object`. Accanto alla parola `extends` troviamo un altra parola chiave del linguaggio Java: `implements` che permette alla classe `EdoRmiOjectImpl` di implementare l'interfaccia `EdoRmiObject`

```
//
// EdoRmiObjectImpl.java
//
import java.io.*;

public class EdoRmiObjectImpl extends java.rmi.server.
  UnicastRemoteObject implements EdoRmiObject {

  public EdoRmiObjectImpl() throws java.rmi.RemoteException {
    super();
  }

  //  fx(x,y)
  //
  //  restituisce il valore della prima componente della
  //  funzione calcolata in (x,y), secondo il metodo n. f
  //

  public float fx(float x, float y, int f,int met) {
```

```
  ...
}
//  fy(x,y)
//
//  restituisce il valore della seconda componente della
//  funzione calcolata in (x,y), secondo il metodo n. f
//

public float fy(float x, float y, int f,int met) {
  ...
}

//
// Oscillatore armonico
//

static float armonico_x(float x, float y) {
  return(float)(y);
}

static float armonico_y(float x, float y) {
  return(float)(-x);
}

//
// Equazione di Van der Pol
//

static float pol_x(float x, float y) {
  return(float)(y-(x*x*x)+x);
}

static float pol_y(float x, float y) {
  return(float)(-x);
}

//
//  Modello "preda-predatore"
//

static float preda_x(float x, float y) {
  float a=(float)0.5, c=(float)0.03, e=(float)0.02;
  return(float)(a*x-c*x*y-e*x*x);
```

```
  }

  static float preda_y(float x, float y) {
    float b=(float)0.2, d=(float)0.06;
    return(float)(-b*y + d*x*y);
  }
}



//
// EdoRmiServer.java
//

import java.rmi.Naming;

public class EdoRmiServer {
  public static void main(String args[]) {
    try {
      EdoRmiObjectImpl  dro  = new EdoRmiObjectImpl();
      Naming.rebind("rmi://localhost:1050/edoRmiService", dro);
    }
    catch (Exception e) {
      System.out.println("Trouble: " + e);
      System.exit(0);
    }
}
```

Nel file `EdoRmiServer` viene definita la modalità con cui il server si collega al registry. Il codice presente nel metodo main non deve essere considerato implementazione dell'oggetto remoto: il suo scopo quello di collegare l'oggetto remoto al registry in funzione sulla macchina, in modo da comunicarne la disponibilità ad eventuali client. L'operazione viene eseguita tramite il metodo `rebind` dell'oggetto `Naming`.

Questi sono i metodi definiti sul server, la compilazione e l'esecuzione corretta di questi metodi indica che la creazione dell'oggetto remoto ed il suo collegamento al `registry` è avvenuto in modo corretto. Una volta che il

server RMI è in esecuzione, è possibile scrivere il client che utilizzi gli oggetti remoti contenuti.

```
//
// EdoRmiClient.java
//
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.net.MalformedURLException;
import java.rmi.NotBoundException;

import java.applet.*;
import java.io.*;
import java.awt.*;

public class EdoRmiClient extends Applet {

  //
  //  Definizione delle variabili,
  //  delle costanti e delle etichette
  //
  //  Metodo per la scelta dell'algoritmo
  //

  public int algoritmo (int m) {
    ...
  }

  //
  //  metodo di inizializzazione dell'applet;
  //

  public void init() {
    ...
  }

  //
  //  eulero(), metodo che esegue il ciclo per il calcolo
  //  dei valori della funzione F utilizzando l'algoritmo
  //  di Eulero
  //

  public int eulero() {
```

```
  ...

  //
  //  viene inizializzato RMI con la dichiarazioni
  //  delle eccezioni
  //
  try {
    EdoRmiObject edo = (EdoRmiObject) Naming.lookup
      ("rmi://localhost:1050/edoRmiService");
      ...

    for (i=1; i<n ;i++) {
      k1 = edo.fx(w[0][i], w[1][i], funzione_scelta,
        scegli_metodo);
      k2 = edo.fy(w[0][i], w[1][i], funzione_scelta,
        scegli_metodo);
      ...
    }
  }

  catch (MalformedURLException murle) {
    System.out.println();
    System.out.println("MalformedURLException");
    System.out.println(murle);
  }
  catch (RemoteException re) {
    System.out.println();
    System.out.println("RemoteException");
    System.out.println(re);
  }
  catch (NotBoundException nbe) {
    System.out.println();
    System.out.println("NotBoundException");
    System.out.println(nbe);
  }
  catch (java.lang.Exception  ex) {
    ex.printStackTrace();
  }

return(n);
}

//
```

```
//  runge_kutta(), metodo che esegue il ciclo per il
//  calcolo dei valori della funzione F utilizzando
//  l'algoritmo di runge_kutta,si procede nelle stesso
//  modo del metodo di Eulero
//

public int runge_kutta() {
  ...
  return(n);
}

public void start() {

}

public void stop() {

}

public void destroy() {

}

//
//  trasforma(nodi);
//  il metodo calcola i valori della matrice wgraf che
//  contiene i valori della matrice w, ma ricolalcolati
//  sulla base delle "coordinate grafiche".
//

public void trasforma (int nodi){
  ...
}

public void paint(Graphics g) {
  ...
}

//
//  action(evt, arg);
//
// metodo invocato automaticamente dalla JVM quando si
// verifica un evento. In particolare noi intercettiamo
```

```
  // soltanto l'evento corrispondente  alla pressione del
  // bottone.
  //

  public boolean action( Event evt, Object arg) {
    ...
  }
}
```

Il client è tutto contenuto nel metodo main della classe `EdoRmiClient`. Le operazioni eseguite sono le seguenti: per prima cosa viene contattato il `registry` tramite l'oggetto `Naming` e viene richiesto l'oggetto remoto. Una volta ottenuto un riferimento all'oggetto remoto, viene chiamato il metodo dell'interfaccia remota, ed il suo risultato viene dato in output attraverso l'applet.

Come accennavamo all'inizio del paragrafo questa versione del programma è basata su una implementazione dell'oggetto remoto tale da fornire come metodi le sole funzioni `fx` ed `fy`, il calcolo degli algoritmi viene lasciato al client: in questo modo pero' la struttura RMI viene sfruttata solo in parte; i due metodi remoti restituiscono un valore *floating point* che servirà poi nel client per conoscere la funzione da implementare. Abbiamo così molti passaggi in rete di dati di piccole dimensioni. Nella seconda versione abbiamo voluto invece "appesantire" il programma facendo svolgere al server anche l'implementazione degli algoritmi, sul client avremo dunque solo ciò che riguarda l' interfaccia grafica. In questo caso il tipo di dato restituito dai metodi remoti non e' più un numero *floating point* ma è un vettore, cio' comporta una grande spostamento di dati in rete con due soli passaggi; i metodi remoti infatti vengono richiesti due volte dal client: nella scelta del'algoritmo da implementare, mentre nella prima implementazione erano inseriti all'interno di un ciclo che veniva eseguito `n` volte.

Per riuscire ad implementare correttamente il programma client abbiamo do-

vuto definire degli oggetti remoti che restituiscono un vettore, a cui vengono passati come parametri tutti i valori necessari per il calcolo dell'algoritmo che il client non conosce.Vediamo come sono stati definiti i metodi, gli oggetti e l'interfaccia per l'oggetto remoto in maniera diversa dalla prima versione:

```
//
//  server/EdoRmiObject.java
//
//  Interfaccia dei metodi pubblicati dall'oggetto EdoRmiObject
//

import java.rmi.*;

public interface EdoRmiObject extends java.rmi.Remote {

  public float[][] eulero(int equazione, int n, float h,
    float x_in, float y_in, float w[][])
      throws java.rmi.RemoteException;
  public float[][] runge_kutta(int equazione, int n,float h,
    float x_in, float y_in, float w[][])
      throws java.rmi.RemoteException;
}
```

In questo caso nell'interfaccia remota vengono definiti i metodi di Eulero e Runge-Kutta, gli vengono passati tutti i parametri necessari per il calcolo dell'algoritmo, eseguito nell'implementazione dell'interfaccia remota,e gli viene chiesto di restituire un vettore a due dimensioni di numeri *floating point*. Nell'implementazione dell'interfaccia remota `EdoRmiObjectImpl` differentemente dalla prima versione vengono implementati i due algoritmi risolutivi:

```
//
//  EdoRmiObjectImpl.java
//
```

```
import java.io.*;

public class EdoRmiObjectImpl extends java.rmi.server.
  UnicastRemoteObject implements EdoRmiObject {

  public EdoRmiObjectImpl() throws java.rmi.RemoteException {
    super();
  }

  public float[][] eulero(int equazione, int n, float h,
    float x_in, float y_in, float w[][]) {

    float k1, k2;
    int i;

    w[0][1] = x_in;
    w[1][1] = y_in;

    for (i=1; i<n ;i++) {
      k1 = fx(w[0][i], w[1][i], equazione);
      k2 = fy(w[0][i], w[1][i], equazione);
      w[0][i+1] = w[0][i]+h*k1;
      w[1][i+1] = w[1][i]+h*k2;
    }
    w[0][0] = n;
    return(w);
  }


  public float[][] runge_kutta(int equazione, int n, float h,
    float x_in, float y_in, float w[][]) {

    float k11,k12,k21,k22,k31,k32,k41,k42;
    int i;

    w[0][1] = x_in;
    w[1][1] = y_in;

    for (i=1; i<n+1 ;i++) {
      k11 = fx(w[0][i], w[1][i], equazione);
      k12 = fy(w[0][i], w[1][i], equazione);
      k21 = fx(w[0][i]+((h/2)*k11),w[1][i]+((h/2)*k12), equazione);
      k22 = fy(w[0][i]+((h/2)*k11),w[1][i]+((h/2)*k12), equazione);
```

```
      k31 = fx(w[0][i]+((h/2)*k21),w[1][i]+((h/2)*k22), equazione);
      k32 = fy(w[0][i]+((h/2)*k21),w[1][i]+((h/2)*k22), equazione);
      k41 = fx(w[0][i]+((h/2)*k31),w[1][i]+((h/2)*k32), equazione);
      k42 = fy(w[0][i]+((h/2)*k31),w[1][i]+((h/2)*k32), equazione);
      w[0][i+1] = w[0][i]+((h/6)*(k11+2*k21+2*k31+k41));
      w[1][i+1] = w[1][i]+((h/6)*(k12+2*k22+2*k32+k42));
    }
    w[0][0] = n;
    return(w);
  }

  //
  // il resto del metodo rimane invariato
  //

  static float fx(float x, float y, int f) {
    ...
    return(rc);
  }

  static float fy(float x, float y, int f) {
    ...
    return(rc);
  }

  //
  //  Definizione delle funzioni
  //
  ...
}
```

Il metodo `EdoRmiServer` rimane lo stesso. Vediamo allora cosa cambia nell'implementazione del client:

```
//
//  EdoRmiClient.java
//
//

import java.rmi.Naming;
```

```
import java.rmi.RemoteException;
import java.net.MalformedURLException;
import java.rmi.NotBoundException;

import java.applet.*;
import java.io.*;
import java.awt.*;


public class EdoRmiClient extends Applet {

  //
  //  Variabili globali
  //
  ...

  //
  // scelta del algoritmo risolutore
  //

  public int algoritmo (int m) {
    int funzione_scelta= new Integer(menu_equazione.
      getSelectedIndex()).intValue();
    int n=new Integer(textField4.getText()).intValue();
    float h=new Float(textField3.getText()).floatValue();
    float x_in = new Float(textField1.getText()).floatValue();
    float y_in = new Float(textField2.getText()).floatValue();
    int rm = 0, i;

    switch (m) {
      case 0:
        //
        //inizializzazione RMI
        //

        try {
          EdoRmiObject edo = (EdoRmiObject)Naming.lookup
            ("rmi://localhost:1050/EdoRmiService");
          w = edo.eulero(funzione_scelta, n, h, x_in, y_in, w);
          rm = (int)w[0][0];
        }
        catch (MalformedURLException murle) {
          System.out.println();
```

```
      System.out.println("MalformedURLException");
      System.out.println(murle);
    }
    catch (RemoteException re) {
      System.out.println();
      System.out.println("RemoteException");
      System.out.println(re);
    }
    catch (NotBoundException nbe) {
      System.out.println();
      System.out.println("NotBoundException");
      System.out.println(nbe);
    }
    catch (java.lang.Exception  ex) {
      ex.printStackTrace();
    }
    break;
  case 1:
    try {
      EdoRmiObject edo = (EdoRmiObject)Naming.lookup
        ("rmi://localhost:1050/EdoRmiService");
      w = edo.runge_kutta(funzione_scelta,n,h,x_in,y_in,w);
      rm = (int)w[0][0];
    }
    catch (MalformedURLException murle) {
      System.out.println();
      System.out.println("MalformedURLException");
      System.out.println(murle);
    }
    catch (RemoteException re) {
      System.out.println();
      System.out.println("RemoteException");
      System.out.println(re);
    }
    catch (NotBoundException nbe) {
      System.out.println();
      System.out.println("NotBoundException");
      System.out.println(nbe);
    }
    catch (java.lang.Exception  ex) {
      ex.printStackTrace();
    }
    break;
```

```
  }
return (rm);
}

//
//  Inizializzazione dell'applet
//
public void init() {
  ...
}

//
//  Definizione dei metodi necessari all'esecuzione
//  dell'applet
//
public void start() {

}

public void stop() {

}

public void destroy() {

}

//
//  trasforma(nodi);
//
//  Il metodo rimane invariato rispetto a quello
//  definito nella prima versione
//
public void trasforma (int nodi){
  ...
}

//
//  paint(g);
//
//  E' lo stesso metodo definito nella prima versione
//
public void paint(Graphics g) {
```

```
    ...
  }

  //
  //  action(evt, arg);
  //
  //  E' lo stesso metodo definito nella prima versione
  //
  public boolean action( Event evt, Object arg) {
    ...
  }
}
```

3.7 Conclusioni

Lo scopo di questa tesi era la presentazione e la costruzione di un software numerico attraverso una tecnologia diversa da quella utilizzata tradizionalmente con i linguaggi di programmazione procedurali: la tecnica della programmazione distribuita in Java mediante il protocollo RMI.

Nella parte iniziale del lavoro abbiamo imparato e studiato le caratteristiche fondamentali di Java che ci potessero portare a capire perché Java viene oggi utilizzato non solo per costruire le applet grafiche nelle pagine web, ma è considerato uno dei principali linguaggi di programmazione per lo sviluppo di applicazioni in ambito industriale; nonostante non sia particolarmente efficiente dal punto di vista della velocità computazionale, rispetto a linguaggi consolidati per l'implementazione di algoritmi di calcolo numerico, come il C o il Fortran.

Le carattestiche intrinseche del linguaggio, quali la portabilità, la struttutra *Object Oriented*, ci hanno permesso di costruire programmi semplici, facilmente leggibili, eleganti e sopratutto utilizzabili su qualsiasi macchina (su cui si installata una Java Virtual Machine); il protocollo RMI ci ha permesso

invece di distribuire componenti dello stesso programma su macchine diverse che eseguono operazioni differenti e forniscono come risultato un output grafico facilmente utilizzabile dall'utente.

L'uso di un linguaggio ad oggetti ci ha permesso anche di verificare come questo modello di programmazione consenta di progettare applicazioni in modo pulito ed elegante, potendo strutturare il codice sorgente in modo da rendere le applicazioni facilmente comprensibili e manutenibili e quindi ampliabili.

Appendice A: Output grafico dei programmi

Di seguito riportiamo la stampa della finestra grafica (client) del programma. Sono riprodotti i grafici ottenuti come traiettoria della soluzione per i diversi modelli, utilizzando il metodo di Runge-Kutta.

Figura 3.2: Traiettoria della soluzione dell'equazione dell'oscillatore armonico

Figura 3.3: Traiettoria della soluzione dell'equazione dell'oscillatore armonico smorzato

Figura 3.4: Traiettoria della soluzione dell'equazione di Van der Pol

Figura 3.5: Traiettoria della soluzione dell'equazione del modello "preda-predatore"

Figura 3.6: Grafici sovrapposti

Appendice B: Listati dei programmi

Riportiamo di seguito i listati delle due versioni del programma per il calcolo della soluzione di un sistema di equazioni differenziali ordinarie. La prima versione, come descritto nelle pagine precedenti, è basata su una implementazione dell'oggetto remoto tale da fornire come metodi le sole funzioni `fx` ed `fy`. In questo modo buona parte del calcolo, compresa l'iterazione del ciclo principale dell'algoritmo, viene eseguita sul client. La seconda versione propone invece una architettura di calcolo distribuito un po' diversa, spostando sul server l'intero algoritmo che implementa i metodi di Eulero e Runge-Kutta; il client ne risulta "alleggerito", ma è necessario passare mediante il protocollo RMI una grande mole di informazioni (la matrice `w`) per poter fornire al client il risultato dell'intera elaborazione del server.

Prima versione

Il programma server

EdoRmiObject.java

```
//
//  server/EdoRmiObject.java
//
//  Interfaccia dei metodi pubblicati dall'oggetto
//  EdoRmiObject
```

```
//
import java.rmi.*;

public interface EdoRmiObject extends java.rmi.Remote {

  public float fx(float x, float y, int f,int met)
    throws java.rmi.RemoteException;
  public float fy(float x, float y, int f,int met)
    throws java.rmi.RemoteException;
}
```

EdoRmiObjectImpl.java

```
//
//  server/EdoRmiObjectImpl.java
//
//  Implementazione dei metodi pubblicati dall'oggetto
//  EdoRmiObject
//
import java.io.*;

public class EdoRmiObjectImpl extends java.rmi.server.
  UnicastRemoteObject implements EdoRmiObject {

  public EdoRmiObjectImpl() throws java.rmi.RemoteException {
    super();
  }

  // fx(x,y)
  //
  // restituisce il valore della prima componente della
  // funzione calcolata in (x,y), secondo il metodo n. f
  //

  public float fx(float x, float y, int f,int met) {
    float rc=(float)0.0;

      switch (f) {
        case 0:
          rc = armonico_x(x, y);
          break;
        case 1:
          rc = pol_x(x, y);
```

```
        break;
      case 2:
        rc = preda_x(x,y);
        break;
    }
    return(rc);
}

// fy(x,y)
//
// restituisce il valore della seconda componente della
// funzione calcolata in (x,y), secondo il metodo n. f
//

public float fy(float x, float y, int f,int met) {
  float rc=(float)0.0;

    switch (f) {
      case 0:
        rc = armonico_y(x, y);
        break;
      case 1:
        rc = pol_y(x, y);
        break;
      case 2:
        rc = preda_y(x, y);
        break;
    }
    return(rc);
}

//
// Oscillatore armonico
//

static float armonico_x(float x, float y) {
  return(float)(y);
}

static float armonico_y(float x, float y) {
  return(float)(-x);
}
```

```
  //
  // Equazione di Van der Pol
  //

  static float pol_x(float x, float y) {
    return(float)(y-(x*x*x)+x);
  }

  static float pol_y(float x, float y) {
    return(float)(-x);
  }

  //
  // Modello "preda-predatore"
  //

  static float preda_x(float x, float y) {
    float a=(float)0.5, c=(float)0.03, e=(float)0.02;
    return(float)(a*x-c*x*y-e*x*x);
  }

  static float preda_y(float x, float y) {
    float b=(float)0.2, d=(float)0.06;
    return(float)(-b*y + d*x*y);
  }
}
```

EdoRmiServer.java

```
//
//  server/EdoRmiObject.java
//
//
import java.rmi.Naming;

public class EdoRmiServer {

  public static void main(String args[]) {
    try {
      EdoRmiObjectImpl  dro  = new EdoRmiObjectImpl();
      Naming.rebind("rmi://localhost:1050/edoRmiService", dro);
    }
    catch (Exception e) {
```

```
      System.out.println("Trouble: " + e);
      System.exit(0);
    }
  }
}
```

Il programma client

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.net.MalformedURLException;
import java.rmi.NotBoundException;

import java.applet.*;
import java.io.*;
import java.awt.*;

public class EdoRmiClient extends Applet {


  //
  // Variabili globali
  //
  int nmax   = 2000;
  int puntiX =  600;
  int puntiY =  500;
  Dimension D;
  float w[][]       = new float[2][nmax];
  int   wgraf[][][][] = new int[2][nmax][4][3];
  float h;
  int   n, equazione;
  Label label1, label2, label3, label4, label5, label6, label7,
    label8; //etichette dei campi
  TextField textField1,textField2,textField3,textField4,
    textField5,textField6,textField7,textField8;
    //campi in cui scrivere
  Choice menu_equazione;
    // menu' a tendina da cui scegliere
    // la funzione per il calcolo
  Choice menu_metodo;
    // menu' a tendina da cui scegliere
```

```
  // il metodo per il calcolo
Button button1;

//
// scelta del algoritmo
//
public int algoritmo (int m){
  int rm = 0;
    switch (m) {
      case 0:
        rm = eulero();
        break;
      case 1:
        rm = runge_kutta();
        break;
    }
    return (rm);
}


//
// init()
//
//  metodo di inizializzazione dell'applet; viene richiamato
//  una sola volta, automaticamente dalla Java Virtual
//  Machine, nel momento in cui l'applet viene avviato.
//
public void init() {

  this.setLayout(null);// inizializzo la finestra

  label1=new Label("Cond.Iniz. X0");
  label1.reshape(25,25,100,10);
  this.add(label1);
  textField1=new TextField();
  textField1.reshape(25,40,100,20);
  this.add(textField1);

  label2=new Label("Cond.Iniz. Y0");
  label2.reshape(150,25,100,10);
  this.add(label2);
  textField2=new TextField();
  textField2.reshape(150,40,100,20);
```

```
  this.add(textField2);

  label3=new Label("PASSO");
  label3.reshape(275,25,100,10);
  this.add(label3);
  textField3=new TextField();
  textField3.reshape(275,40,100,20);
  this.add(textField3);

  label4=new Label("NODI");
  label4.reshape(400,25,100,10);
  this.add(label4);
  textField4=new TextField();
  textField4.reshape(400,40,100,20);
  this.add(textField4);

  int width = Integer.parseInt(getParameter("width"));
  int height = Integer.parseInt(getParameter("height"));

  menu_equazione = new Choice();
  menu_equazione.addItem("Oscillatore armonico");
  menu_equazione.addItem("Equazioni di Van der pol");
  menu_equazione.addItem("Modello preda-predatore");
  menu_equazione.reshape(25,90,220,100);
  this.add(menu_equazione);

  menu_metodo = new Choice();
  menu_metodo.addItem("Metodo di Eulero");
  menu_metodo.addItem("Metodo di Runge Kutta");
  menu_metodo.reshape(260,90,160,100);
  this.add(menu_metodo);

  button1=new Button("GRAFICO");
  button1.reshape(450,90,100,30);
  this.add(button1);

  show();

}


//
//  eulero()
```

```
//
//  esegue il ciclo per il calcolo dei valori della funzione
//  F utilizzando il metodo di Eulero. I valori calcolati
//  vengono memorizzati nella matrice w che ha due righe ed
//  un numero di colonne pari al numero dei nodi: nelle due
//  righe di ogni colonna memorizziamo le componenti della
//  funzione calcolate al passo precedente. Nella prima colonna
//  della matrice memorizziamo le condizioni iniziali.
//
public int eulero() {
  float k1, k2;
  int i;

  int funzione_scelta = new Integer(menu_equazione.
    getSelectedIndex()).intValue();
  int scegli_metodo = new Integer(menu_metodo.
    getSelectedIndex()).intValue();
  int n=new Integer(textField4.getText()).intValue();
  float h=new Float(textField3.getText()).floatValue();
  float x_in = new Float(textField1.getText()).floatValue();
  float y_in = new Float(textField2.getText()).floatValue();

  //
  // inizializzazioni di RMI
  //

 try {
    EdoRmiObject edo = (EdoRmiObject) Naming.lookup
      ("rmi://localhost:1050/edoRmiService");

    w[0][1] = x_in;
    w[1][1] = y_in;

    for (i=1; i<n ;i++) {
      k1 = edo.fx(w[0][i], w[1][i], funzione_scelta,scegli_metodo);
      k2 = edo.fy(w[0][i], w[1][i], funzione_scelta,scegli_metodo);
      w[0][i+1] = w[0][i]+h*k1;
      w[1][i+1] = w[1][i]+h*k2;
    }
  }
  catch (MalformedURLException murle) {
    System.out.println();
    System.out.println("MalformedURLException");
```

```
    System.out.println(murle);
  }
  catch (RemoteException re) {
    System.out.println();
    System.out.println("RemoteException");
    System.out.println(re);
  }
  catch (NotBoundException nbe) {
    System.out.println();
    System.out.println("NotBoundException");
    System.out.println(nbe);
  }
  catch (java.lang.Exception  ex) {
    ex.printStackTrace();
  }
return(n);
}

//
//  runge_kutta()
//
//  esegue il ciclo per il calcolo dei valori della funzione F
//  utilizzando  il metodo di Rnge Kutta. I valori calcolati
//  vengono memorizzati nella matrice w che ha due righe ed un
//  numero di colonne pari al numero dei nodi: nelle due righe
//  di ogni colonna memorizziamo le componenti della funzione
//  calcolate al passo precedente. Nella prima colonna della
//  matrice memorizziamo le condizioni iniziali.
//

public int runge_kutta() {

  float k11,k12,k21,k22,k31,k32,k41,k42;
  int i;

  int funzione_scelta =
    new Integer(menu_equazione.getSelectedIndex()).intValue();
  int scegli_metodo =
    new Integer(menu_metodo.getSelectedIndex()).intValue();
  int n=new Integer(textField4.getText()).intValue();
  float h=new Float(textField3.getText()).floatValue();
  float x_in = new Float(textField1.getText()).floatValue();
  float y_in = new Float(textField2.getText()).floatValue();
```

```
//
// inizializzazioni di RMI
//

try {
  EdoRmiObject edo =(EdoRmiObject)Naming.
    lookup("rmi://localhost:1050/edoRmiService");

  w[0][1] = x_in;
  w[1][1] = y_in;

  for (i=1; i<n+1 ;i++) {
    k11 = edo.fx(w[0][i],w[1][i],funzione_scelta,scegli_metodo);
    k12 = edo.fy(w[0][i],w[1][i],funzione_scelta,scegli_metodo);
    k21 = edo.fx(w[0][i]+((h/2)*k11),w[1][i]+((h/2)*k12),
      funzione_scelta,scegli_metodo);
    k22 = edo.fy(w[0][i]+((h/2)*k11),w[1][i]+((h/2)*k12),
      funzione_scelta,scegli_metodo);
    k31 = edo.fx(w[0][i]+((h/2)*k21),w[1][i]+((h/2)*k22),
      funzione_scelta,scegli_metodo);
    k32 = edo.fy(w[0][i]+((h/2)*k21),w[1][i]+((h/2)*k22),
      funzione_scelta,scegli_metodo);
    k41 = edo.fx(w[0][i]+((h/2)*k31),w[1][i]+((h/2)*k32),
      funzione_scelta,scegli_metodo);
    k42 = edo.fy(w[0][i]+((h/2)*k31),w[1][i]+((h/2)*k32),
      funzione_scelta,scegli_metodo);
    w[0][i+1] = w[0][i]+((h/6)*(k11+2*k21+2*k31+k41));
    w[1][i+1] = w[1][i]+((h/6)*(k12+2*k22+2*k32+k42));
  }
}
catch (MalformedURLException murle) {
  System.out.println();
  System.out.println("MalformedURLException");
  System.out.println(murle);
}
catch (RemoteException re) {
  System.out.println();
  System.out.println("RemoteException");
  System.out.println(re);
}
catch (NotBoundException nbe) {
  System.out.println();
```

```
    System.out.println("NotBoundException");
    System.out.println(nbe);
  }
  catch (java.lang.Exception  ex) {
    ex.printStackTrace();
  }
  return(n);
}

//
// start()
//
//  funzione eseguita automaticamente un'unica volta dalla
//  Java Virtual Machine nel momento in cui viene eseguito
//  l'applet.
//

public void start() {

}

//
// stop()
//
//  funzione eseguita automaticamente dalla Java Virtual
//  Machine nel momento in cui viene interrotta l'esecuzione
//  dell'applet.
//
public void stop() {

}

//
// destroy()
//
//  funzione eseguita automaticamente un'unica volta dalla
//  Java Virtual Machine nel momento in cui viene eliminato
//  l'applet dalla Java Virtual Machine.
//
public void destroy() {

}
```

```
//
// trasforma(nodi);
//
// il metodo calcola i valori della matrice wgraf che
// contiene i valori della matrice w, ma ricolalcolati
// sulla base delle "coordinate grafiche".
//
public void trasforma (int nodi){
  int i,k,xs,ys;
  float c,scalax,scalay,sup=(float)1.0,inf=(float)1.0,
    min=(float)1.0,max=(float)1.0,dx=(float)1.0,dy=(float)1.0;
  int funzione_scelta =
    new Integer(menu_equazione.getSelectedIndex()).intValue();
  int scegli_metodo =
    new Integer(menu_metodo.getSelectedIndex()).intValue();

  inf=w[0][1];
  sup=w[0][1];
  min=w[1][1];
  max=w[1][1];

  //
  // calcolo i valori massimi e minimi per entrambe
  //  le componenti della funzione
  //

  for (i=1; i<=nodi; i++) {
    if (w[0][i]>sup) {
      sup = w[0][i];
    }
    if (w[0][i] < inf) {
      inf = w[0][i];
    }
    if (w[1][i]>max) {
      max = w[1][i];
    }
    if (w[1][i] < min) {
      min = w[1][i];
    }
  }
```

```
  //
  // cotruisce le coordinate grafiche in wgraf
  //

  for(i=1;i<=nodi;i++)  {

    wgraf[0][i][funzione_scelta][scegli_metodo]=20+(int)((float)
      ((w[0][i]-inf) * ((puntiX-20)-20))/(float)(sup-inf));
    wgraf[1][i][funzione_scelta][scegli_metodo]=140-(int)((float)
      ((w[1][i]-min) *(140-(puntiY-140)))/(float)(max-min));
  }
  n = nodi;
}

//
//  paint(g);
//
//  viene richiamato automaticamente dalla JVM quando c'e'
//  necessita' di ridisegnare la finestra grafica.
//

public void paint(Graphics g) {

  int i, k;
  int m =new Integer(menu_metodo.getSelectedIndex()).intValue();

  D = size();
  puntiX = D.width;
  puntiY = D.height;

  g.setColor(Color.black);
  g.drawRect(10,130,puntiX-20,puntiY-140);
  g.clearRect(11,131,puntiX-22,puntiY-142);

  for (k=0; k<4; k++) {
    switch (k) {
      case 0:
        g.setColor(Color.blue);
        break;
      case 1:
        g.setColor(Color.red);
        break;
      case 2:
```

```
          g.setColor(Color.green);
          break;
        case 3:
          g.setColor(Color.cyan);
          break;
      }
      for (i=1;i<n-1;i++) {
        g.drawLine(wgraf[0][i][k][m],wgraf[1][i][k][m],
          wgraf[0][i+1][k][m],wgraf[1][i+1][k][m]);
      }
    }
  }

  //
  //  action(evt, arg);
  //
  //  metodo invocato automaticamente dalla JVM quando si
  //  verifica un evento. In particolare noi intercettiamo
  //  soltanto l'evento corrispondente alla pressione del
  //  bottone.
  //

  public boolean action( Event evt, Object arg) {
    int n_nodi;
    int scegli_metodo =
      new Integer(menu_metodo.getSelectedIndex()).intValue();

    if ( evt.target == button1 ) {
      n_nodi = algoritmo(scegli_metodo);
      trasforma(n_nodi);
      repaint();
    }
    return(true);
  }
}
```

EdoRmiClient.html

```
<html>
  <head>
    <title>Edo RMI Client</title>
  </head>

  <body>
```

```
    <applet code="EdoRmiClient.class" width="600" height="500">
    </applet>
  </body>
</html>
```

Seconda versione

Il programma server

EdoRmiObject.java

```
//
//  server/EdoRmiObject.java
//
//  Interfaccia dei metodi pubblicati dall'oggetto
//  EdoRmiObject
//

import java.rmi.*;

public interface EdoRmiObject extends java.rmi.Remote {

  public float[][] eulero(int equazione, int n, float h,
    float x_in, float y_in, float w[][])
      throws java.rmi.RemoteException;
  public float[][] runge_kutta(int equazione, int n,float h,
    float x_in,float y_in, float w[][])
      throws java.rmi.RemoteException;

}
```

EdoRmiObjectImpl.java

```
//
//  server/EdoRmiObjectImpl.java
//
//  Implementazione dei metodi pubblicati dall'oggetto
//  EdoRmiObject
//

import java.io.*;
```

```
public class EdoRmiObjectImpl extends java.rmi.server.
  UnicastRemoteObject implements EdoRmiObject {

  public EdoRmiObjectImpl() throws java.rmi.RemoteException {
    super();
  }

  //
  //  eulero()
  //
  //  esegue il ciclo per il calcolo dei valori della funzione
  //  F utilizzando il metodo di Eulero. I valori calcolati
  //  vengono memorizzati nella matrice w che ha due righe ed
  //  un numero di colonne pari al numero dei nodi: nella due
  //  righe di ogni colonna memorizziamo le componenti della
  //  funzione calcolate al passo precedente. Nella prima colonna
  //  della matrice memorizziamo le condizioni iniziali.
  //

  public float[][] eulero(int equazione, int n, float h,
    float x_in, float y_in, float w[][]) {
      float k1, k2;
      int i;

      w[0][1] = x_in;
      w[1][1] = y_in;

      for (i=1; i<n ;i++) {
        k1 = fx(w[0][i], w[1][i], equazione);
        k2 = fy(w[0][i], w[1][i], equazione);
        w[0][i+1] = w[0][i]+h*k1;
        w[1][i+1] = w[1][i]+h*k2;
      }
      w[0][0] = n;
      return(w);
  }

  //
  //  runge_kutta()
  //
  //  esegue il ciclo per il calcolo dei valori della funzione F
  //  utilizzando il metodo di Runge Kutta. I valori calcolati
  //  vengono memorizzati nella matrice w che ha due righe ed un
```

```
//  numero di colonne pari al numero dei nodi: nelle due righe
//  di ogni colonna memorizziamo le componenti della funzione
//  calcolate al passo precedente. Nella prima colonna della
//  matrice memorizziamo le condizioni iniziali.
//

public float[][] runge_kutta(int equazione, int n, float h,
  float x_in, float y_in, float w[][]) {

  float k11,k12,k21,k22,k31,k32,k41,k42;
  int i;

  w[0][1] = x_in;
  w[1][1] = y_in;

  for (i=1; i<n+1 ;i++) {
    k11 = fx(w[0][i], w[1][i], equazione);
    k12 = fy(w[0][i], w[1][i], equazione);
    k21 = fx(w[0][i]+((h/2)*k11),w[1][i]+((h/2)*k12), equazione);
    k22 = fy(w[0][i]+((h/2)*k11),w[1][i]+((h/2)*k12), equazione);
    k31 = fx(w[0][i]+((h/2)*k21),w[1][i]+((h/2)*k22), equazione);
    k32 = fy(w[0][i]+((h/2)*k21),w[1][i]+((h/2)*k22), equazione);
    k41 = fx(w[0][i]+((h/2)*k31),w[1][i]+((h/2)*k32), equazione);
    k42 = fy(w[0][i]+((h/2)*k31),w[1][i]+((h/2)*k32), equazione);
    w[0][i+1] = w[0][i]+((h/6)*(k11+2*k21+2*k31+k41));
    w[1][i+1] = w[1][i]+((h/6)*(k12+2*k22+2*k32+k42));
  }
  w[0][0] = n;
  return(w);
}


//  fx(x,y)
//
//  restituisce il valore della prima componente della
//  funzione calcolata in (x,y), secondo il metodo n. f
//

static float fx(float x, float y, int f) {
  float rc=(float)0.0;

  switch (f) {
    case 0:
```

```
      rc = armonico_x(x, y);
      break;
    case 1:
      rc = pol_x(x, y);
      break;
    case 2:
      rc = preda_x(x,y);
      break;
  }
  return(rc);
}

//  fy(x,y)
//
//  restituisce il valore della seconda componente della
//  funzione calcolata in (x,y), secondo il metodo n. f
//

static float fy(float x, float y, int f) {
  float rc=(float)0.0;

  switch (f) {
    case 0:
      rc = armonico_y(x, y);
      break;
    case 1:
      rc = pol_y(x, y);
      break;
    case 2:
      rc = preda_y(x, y);
      break;
  }
  return(rc);
}

//
// Oscillatore armonico
//

static float armonico_x(float x, float y) {
  return(float)(y);
}
```

```
  static float armonico_y(float x, float y) {
    return(float)(-x);
  }

  //
  // Equazione di Van der Pol
  //

  static float pol_x(float x, float y) {
    return(float)(y-(x*x*x)+x);
  }

  static float pol_y(float x, float y) {
    return(float)(-x);
  }

  //
  //  Modello "preda-predatore"
  //

  static float preda_x(float x, float y) {
    float a=(float)0.5, c=(float)0.03, e=(float)0.02;
    return(float)(a*x-c*x*y-e*x*x);
  }

  static float preda_y(float x, float y) {
    float b=(float)0.2, d=(float)0.06;
    return(float)(-b*y + d*x*y);
  }
}
```

EdoRmiServer.java

```
//
//  server/EdoRmiServer.java
//
//  Server per la pubblicazione dell'oggetto EdoRmiObject
//

import java.rmi.Naming;

public class EdoRmiServer {
  public static void main(String args[]) {
    try {
```

```
      EdoRmiObjectImpl  oggetto  = new EdoRmiObjectImpl();
      Naming.rebind("rmi://localhost:1050/EdoRmiService",oggetto);
    }
    catch (Exception e) {
      System.out.println("Trouble: " + e);
      System.exit(0);
    }
  }
}
```

Il programma client

EdoRmiClient.java

```
//
//  client/EdoRmiClient.java
//
//  Applet client che si interfaccia con i metodi Eulero e
//  Runge_Kutta pubblicati dall'oggetto remoto EdoRmiObject
//

import java.rmi.Naming;
import java.rmi.RemoteException;
import java.net.MalformedURLException;
import java.rmi.NotBoundException;

import java.applet.*;
import java.io.*;
import java.awt.*;


public class EdoRmiClient extends Applet {

  //
  //  Variabili globali
  //

  int nmax   = 2000;
  int puntiX =  600;
  int puntiY =  500;

  Dimension D;
```

```
float w[][]       = new float[2][nmax];
int   wgraf[][][][] = new int[2][nmax][4][3];
float h;
//float inf, sup, min, max, dx, dy;
int   n, equazione;
Label label1, label2, label3, label4, label5, label6,
      label7, label8; //etichette dei campi
TextField textField1,textField2,textField3,textField4,
      textField5, textField6, textField7, textField8;
      //campi in cui scrivere
Choice menu_equazione;
      // menu' a tendina da cui scegliere la
      // funzione per il calcolo
Choice menu_metodo;
      // menu' a tendina da cui scegliere il metodo
Button button1;

//
//  scelta del metodo
//

public int algoritmo (int m) {
  int funzione_scelta = new Integer(menu_equazione.
      getSelectedIndex()).intValue();
  int n=new Integer(textField4.getText()).intValue();
  float h=new Float(textField3.getText()).floatValue();
  float x_in = new Float(textField1.getText()).floatValue();
  float y_in = new Float(textField2.getText()).floatValue();

  int rm = 0, i;
  switch (m) {
    case 0:
      try {
        EdoRmiObject edo = (EdoRmiObject) Naming.lookup
            ("rmi://localhost:1050/EdoRmiService");
        w = edo.eulero(funzione_scelta, n, h, x_in, y_in, w);
        rm = (int)w[0][0];
      }
      catch (MalformedURLException murle) {
        System.out.println();
        System.out.println("MalformedURLException");
        System.out.println(murle);
      }
```

```
    catch (RemoteException re) {
      System.out.println();
      System.out.println("RemoteException");
      System.out.println(re);
    }
    catch (NotBoundException nbe) {
      System.out.println();
      System.out.println("NotBoundException");
      System.out.println(nbe);
    }
    catch (java.lang.Exception  ex) {
      ex.printStackTrace();
    }
    break;
  case 1:
    try {
      EdoRmiObject edo = (EdoRmiObject) Naming.lookup
          ("rmi://localhost:1050/EdoRmiService");
      w = edo.runge_kutta(funzione_scelta, n, h, x_in, y_in, w);
      rm = (int)w[0][0];
    }
    catch (MalformedURLException murle) {
      System.out.println();
      System.out.println("MalformedURLException");
      System.out.println(murle);
    }
    catch (RemoteException re) {
      System.out.println();
      System.out.println("RemoteException");
      System.out.println(re);
    }
    catch (NotBoundException nbe) {
      System.out.println();
      System.out.println("NotBoundException");
      System.out.println(nbe);
    }
    catch (java.lang.Exception  ex) {
      ex.printStackTrace();
    }
    break;
}

return (rm);
```

```
}


//
//  init()
//
//  metodo di inizializzazione dell'applet; viene richiamato
//  una sola volta, automaticamente dalla Java Virtual
//  Machine, nel momento in cui l'applet viene avviato.
//
public void init() {

  this.setLayout(null);// inizializzo la finestra

  // X0 e Y0 sono le condizioni iniziali
  label1=new Label("Cond.Iniz. X0");
  label1.reshape(25,25,100,10);
  this.add(label1);
  textField1=new TextField();
  textField1.reshape(25,40,100,20);
  this.add(textField1);

  label2=new Label("Cond.Iniz. Y0");
  label2.reshape(150,25,100,10);
  this.add(label2);
  textField2=new TextField();
  textField2.reshape(150,40,100,20);
  this.add(textField2);

  label3=new Label("PASSO");
  label3.reshape(275,25,100,10);
  this.add(label3);
  textField3=new TextField();
  textField3.reshape(275,40,100,20);
  this.add(textField3);

  label4=new Label("NODI");
  label4.reshape(400,25,100,10);
  this.add(label4);
  textField4=new TextField();
  textField4.reshape(400,40,100,20);
  this.add(textField4);
```

```
  int width = Integer.parseInt(getParameter("width"));
  int height = Integer.parseInt(getParameter("height"));

  menu_equazione = new Choice();
  menu_equazione.addItem("Oscillatore armonico");
  menu_equazione.addItem("Equazioni di Van der pol");
  menu_equazione.addItem("Modello preda-predatore");
  menu_equazione.reshape(25,90,220,100);
  this.add(menu_equazione);

  menu_metodo = new Choice();
  menu_metodo.addItem("Metodo di Eulero");
  menu_metodo.addItem("Metodo di Runge Kutta");
  menu_metodo.reshape(260,90,160,100);
  this.add(menu_metodo);

  button1=new Button("GRAFICO");
  button1.reshape(450,90,100,30);
  this.add(button1);

  show();

}

//
//  start()
//
//  funzione eseguita automaticamente un'unica volta dalla Java
//  Virtual Machine nel momento in cui viene eseguito l'applet.
//
public void start() {

}

//
//  stop()
//
//  funzione eseguita automaticamente dalla Java Virtual Machine
//  nel momento in cui viene interrotta l'esecuzione dell'applet.
//
public void stop() {

}
```

```
//
//  destroy()
//
//  funzione eseguita automaticamente un'unica volta dalla Java
//  Virtual Machine nel momento in cui viene eliminato l'applet
//  dalla Java Virtual Machine.
//
public void destroy() {

}

//
//  trasforma(nodi);
//
//  il metodo calcola i valori della matrice wgraf che
//  contiene i valori della matrice w, ma ricolalcolati
//  sulla base delle "coordinate grafiche".
//
public void trasforma (int nodi){
  int i,k,xs,ys;
  float c,scalax,scalay, sup=(float)1.0, inf=(float)1.0,
    min=(float)1.0, max=(float)1.0, dx=(float)1.0,dy=(float)1.0;
  int funzione_scelta = new Integer(menu_equazione.
    getSelectedIndex()).intValue();
  int scegli_metodo = new Integer(menu_metodo.
    getSelectedIndex()).intValue();

   inf=w[0][1];
   sup=w[0][1];
   min=w[1][1];
   max=w[1][1];

  //
  //  calcolo i valori massimi e minimi per entrambe
  //  le componenti della funzione
  //
   for (i=1; i<=nodi; i++) {
     if (w[0][i]>sup) {
       sup = w[0][i];
     }
     if (w[0][i] < inf) {
       inf = w[0][i];
```

```
      }
      if (w[1][i]>max) {
        max = w[1][i];
      }
      if (w[1][i] < min) {
        min = w[1][i];
      }
    }

  //
  //  cotruisce le coordinate grafiche in wgraf
  //
  for(i=1;i<=nodi;i++)  {
    wgraf[0][i][funzione_scelta][scegli_metodo] = 20+(int)
      ((float)((w[0][i]-inf) * ((puntiX-20)-20))/
        (float)(sup-inf));
    wgraf[1][i][funzione_scelta][scegli_metodo] = 140-(int)
      ((float)((w[1][i]-min) *(140-(puntiY-140)))/
        (float)(max-min));
  }
  n = nodi;
}


//
//  paint(g);
//
//  viene richiamato automaticamente dalla JVM quando
//  c'e' necessita' di ridisegnare la finestra grafica.
//

public void paint(Graphics g) {

  int i, k;
  int m =new Integer(menu_metodo.getSelectedIndex()).intValue();

  D = size();
  puntiX = D.width;
  puntiY = D.height;

  g.setColor(Color.black);
  g.drawRect(10,130,puntiX-20,puntiY-140);
  g.clearRect(11,131,puntiX-22,puntiY-142);
```

```
  for (k=0; k<4; k++) {
    switch (k) {
      case 0:
        g.setColor(Color.blue);
        break;
      case 1:
        g.setColor(Color.red);
        break;
      case 2:
        g.setColor(Color.green);
        break;
      case 3:
        g.setColor(Color.cyan);
        break;
    }
    for (i=1;i<n-1;i++) {
      g.drawLine(wgraf[0][i][k][m],wgraf[1][i][k][m],
        wgraf[0][i+1][k][m],wgraf[1][i+1][k][m]);
    }
  }
}


//
//  action(evt, arg);
//
//  metodo invocato automaticamente dalla JVM quando si verifica
//  un evento. In particolare noi intercettiamo soltanto l'evento
//  corrispondente  alla pressione del bottone.
//

public boolean action( Event evt, Object arg) {
  int n_nodi;
  int scegli_metodo = new Integer(menu_metodo.getSelectedIndex()).
    intValue();

  if ( evt.target == button1 ) {
    n_nodi = algoritmo(scegli_metodo);
    trasforma(n_nodi);
    repaint();
  }
  return(true);
```

```
  }
}
```

EdoRmiClient.html

```
<html>
  <head>
    <title>Edo RMI Client</title>
  </head>

  <body>
    <applet code="EdoRmiClient.class" width="600" height="500">
    </applet>
  </body>
</html>
```

Bibliografia

[1] G. Ausiello, A. Marchetti-Spaccamela, M. Protasi, *Teoria e progetto di algoritmi fondamentali*, Franco Angeli, 1988.

[2] B. Blount, S.Chatterjee, *An evaluation of Java for numerical computing*, preprint, 1999.[1]

[3] L. Comi, *Java flash*, Apogeo, 1996.

[4] D. Flanagan, *Java in a Nutshell*, O'Reilly & Associates, 1996.

[5] M.Hirsch, S. Smale, *Differential equations, Dynamical System and Linear Algebra*, Academic Press, 1974.

[6] B.W. Kernighan, R. Pike, *Practical programming*, Addison-Wesley, 1999.

[7] M. Snir, J. Moreira, M. Gupta, L. Haibt, S. Midkiff, *Floating-point performance in Java*, preprint, "T.J. Watson Research Center" IBM, 1998.

[8] P. Naughton, *Il manuale Java*, Mc Graw-Hill, 1996.

[9] R.A. Plastock, G. Kalley, *Teoria e problemi di Computer Grafica*, Etas Libri, 1989.

[1] I *preprint* riportati in questa bibliografia sono tratti dagli atti della conferenza annuale *Java Grande Conference*, promossa dalla ACM, tenutasi nel Novembre del 1998 alla Stanford University di Palo Alto e nel Giugno 1999 a San Francisco.

[10] A. Quarteroni, R. Sacco, F. Saleri, *Matematica numerica*, Springer, 1998.

[11] P. van der Linden, *Just Java and Beyond* , Practice-Hall, 1998.

[12] P. Wu, S. Midkiff, J. Moreira, M. Gupta, *Efficient support for complex numbers in Java*, preprint, 1999.

[13] AAVV, *MokaBook*, a cura di G. Puliti, Hoops, 2001.