Università degli Studi di Firenze
Facoltà di Scienze Matematiche Fisiche e Naturali
Corso di Laurea in Informatica

GklsGUI

Un insieme di *tool* didattici per l'ottimizzazione

Anno Accademico 2004-2005

Candidato: Valerio Angelini

Relatore: Prof. Luigi Brugnano

alla Determinazione,
elemento nonsingolare.

e al determinante

Indice

Introduzione

Possiamo descrivere un problema di ottimizzazione globale come la ricerca del miglior insieme di parametri possibile che ottimizzi una data funzione obiettivo. Gran parte dei problemi, in genere, ammette soluzioni che sono ottime localmente, ma non globalmente.

Diventa così di fondamentale importanza avere a disposizione una varietà di problemi test, sui quali poter sperimentare la robustezza dei metodi risolutivi che si vogliono implementare.

M. Gaviano e D. Lera [GL98] hanno sviluppato una tecnica per generare funzioni test per problemi di questo genere a partire da una serie di parametri della funzione obiettivo stabiliti *a priori*: la dimensione del problema, il numero di minimi locali, i punti di minimo locali stessi ed i valori della funzione in questi punti.

Su queste basi è stata sviluppata `GKLS` [GKLS03], una libreria in linguaggio `C` che permette di generare una vasta gamma di funzioni test partendo da un insieme minimo di informazioni di base e generando le altre in maniera *pseudo-casuale*.

Scopo di questa tesi è di produrre una nuova libreria che permetta l'utilizzo del software `GKLS` in ambiente MATLAB, consentendo così di unire la flessibilità delle funzioni test generate, alla potenza del linguaggio di *scripting* interno a MATLAB.

Utilizzando questa libreria verrà sviluppato un software grafico interattivo, creato per testare e valutare algoritmi di ricerca di punti di minimo. Verranno inoltre implementati alcuni metodi appresi durante il corso di *metodi numerici per l'ottimizzazione*, che serviranno come algoritmi base di confronto.

La progettazione del software verrà portata avanti con l'intento di rendere il programma estendibile ed utilizzabile a fine didattico e/o di analisi dei problemi e dei relativi metodi di risoluzione.

Capitolo 1

Basi teoriche

1.1 Genesi delle funzioni test

Descriviamo le modalità per ottenere funzioni test per il problema di ottimo

$$\min f(\mathbf{x}), \qquad \mathbf{x} \in D \subseteq \mathbb{R}^n.$$

Esse verranno generate a partire da alcuni dati stabiliti *a priori*:

- la dimensione del problema;
- il numero di minimi locali;
- i punti di minimo locale;
- i valori della funzione nei punti di minimo;
- le "regioni di attrazione"[1] dei punti di minimo.

La tecnica usata per generare queste funzioni [GL98] consiste nel definire una funzione quadratica convessa Z, che d'ora in poi chiameremo "paraboloide", in un dominio fissato $D \subseteq \mathbb{R}^n$, e poi di ridefinire l'equazione in degli intorni sferici $S_i \subseteq D$, $i = 1, \ldots, m$, di raggio ρ_i tramite delle funzioni polinomiali di grado 3 (5) che "deformano" Z, in modo da avere una funzione di test di classe C^1 (C^2). La scelta dei raggi ρ_i viene effettuata secondo i seguenti vincoli:

- Gli insiemi S_i non si devono sovrapporre.
- Ogni intorno S_i deve essere interamente contenuto in D.

[1] Non possiamo parlare di una regione di attrazione vera e propria, dato che la definizione di questa si basa sulla scelta di un algoritmo di ricerca. In questo caso, come sarà chiaro in seguito, si intende il raggio ρ_i della palla in cui il paraboloide viene deformato.

1.1.1 Costruzione delle regioni di minimo locale

Esponiamo brevemente la costruzione di una di queste funzioni polinomiali di grado tre in una generica palla S. Indicheremo d'ora in avanti con la notazione $\|\cdot\|$ l'usuale norma euclidea ($\|\cdot\|_2$).

Definiamo formalmente il paraboloide Z con centro $\mathbf{T}$, di equazione $g(\mathbf{x})$:

$$Z: \quad g(\mathbf{x}) = \|\mathbf{x}-\mathbf{T}\|^2 + t, \quad \mathbf{x},\mathbf{T} \in D, \quad t \in \mathbb{R}. \tag{1.1}$$

Pertanto Z avrà punto di minimo in $\mathbf{T}$, con valore t.

Fissiamo $\mathbf{M} \equiv (y_1, \ldots, y_n)^T$, punto di minimo della deformazione polinomiale, con $\mathbf{M} \neq \mathbf{T}$. Per mantenere $\mathbf{T}$ al di fuori della zona "deformata", scegliamo il raggio ρ dell'intorno sferico S, come:

$$\rho < \|\mathbf{M}-\mathbf{T}\|$$

La palla S di centro $\mathbf{M}$ avrà quindi la forma

$$S = \{\mathbf{x} \in \mathbb{R}^n |\ \|\mathbf{x}-\mathbf{M}\| \leq \rho\}.$$

Chiamiamo B la frontiera di S. Il nostro scopo è quello di ridefinire la funzione Z all'interno della regione S.

Per far questo descriviamo innanzitutto ogni punto di S come

$$\mathbf{x} \in S \equiv (\lambda, \mathbf{d}),$$

con

$$\lambda \in [0,\rho] \subset \mathbb{R}, \qquad \mathbf{d} \in \mathbb{R}^n, \|\mathbf{d}\| = 1,$$

tale che

$$\mathbf{x} = \mathbf{M} + \lambda\mathbf{d}.$$

Consideriamo adesso il problema, ristretto ad una direzione $\widehat{\mathbf{d}}$ di trovare il polinomio $C(\lambda)$ di grado 3 che soddisfi le seguenti condizioni:

$C(0) = f$**.** Assegnamo al punto di minimo il valore f;

$C'(0) = 0$**.** Il punto di minimo deve essere un punto stazionario ;

$C(\rho) = \varphi$**.** Il punto alla frontiera ρ deve avere lo stesso valore φ del paraboloide in quel punto;

$C'(\rho) = \gamma$**.** Il punto alla frontiera ρ deve avere la stessa derivata γ del paraboloide in quel punto.

Qui lo 0 rappresenta il punto di minimo $\mathbf{M}$, ρ rappresenta il punto della frontiera intersecato dalla direzione che stiamo considerando e φ e γ rappresentano, rispettivamente, il valore della funzione e della derivata direzionale del paraboloide Z in ρ. Andiamo quindi a risolvere il sistema:

$$\begin{cases} C(\lambda) &= a\lambda^3 + b\lambda^2 + c\lambda + d, \\ C(0) &= f, \\ C'(0) &= 0, \\ C(\rho) &= \varphi, \\ C'(\rho) &= \gamma. \end{cases}$$

Si ottiene,

$$\begin{aligned} C'(\lambda) &= 3a\lambda^2 + 2b\lambda + c, \\ C(0) = d \quad &\Rightarrow \quad d = f, \\ C'(0) = c \quad &\Rightarrow \quad c = 0. \end{aligned}$$

Imponendo le ultime due condizioni,

$$\begin{cases} C(\rho) &= a\rho^3 + b\rho^2 + f = \varphi, \\ C'(\rho) &= 3a\rho^2 + 2b\rho = \gamma, \end{cases}$$

vengono ricavati i due coefficienti

$$a = -\frac{2}{\rho^3}(\varphi - f) + \frac{\gamma}{\rho^2} \qquad \text{e} \qquad b = -\frac{2}{\rho^2}(\varphi - f) + \frac{\gamma}{\rho}$$

che definiscono il polinomio

$$C(\lambda) = \left(-\frac{2}{\rho^3}(\phi - f) + \frac{\gamma}{\rho^2}\right)\lambda^3 + \left(-\frac{2}{\rho^2}(\phi - f) + \frac{\gamma}{\rho}\right)\lambda^2 + f.$$

A questo punto, al variare di $\widehat{\mathbf{d}}$ possiamo tornare alla notazione n-dimensionale. In questo caso scriveremo $\lambda \equiv \|\mathbf{x} - \mathbf{M}\|$ e $\mathbf{d} \equiv \frac{\mathbf{x}-\mathbf{M}}{\|\mathbf{x}-\mathbf{M}\|}$. Per ogni $\mathbf{x} \in S$, definiamo la funzione

$$\begin{aligned} C_\rho(\mathbf{x}) = &\left(\frac{2}{\rho^2}\frac{\langle \mathbf{x}-\mathbf{M}, \mathbf{T}-\mathbf{M}\rangle}{\|\mathbf{x}-\mathbf{M}\|} - \frac{2}{\rho^3}A\right)\|\mathbf{x}-\mathbf{M}\|^3 + \\ &+ \left(1 - \frac{4}{\rho}\frac{\langle \mathbf{x}-\mathbf{M}, \mathbf{T}-\mathbf{M}\rangle}{\|\mathbf{x}-\mathbf{M}\|} + \frac{3}{\rho^2}A\right)\|\mathbf{x}-\mathbf{M}\|^2 + f, \end{aligned}$$

dove $A = \|\mathbf{T}-\mathbf{M}\|^2 + t - f$, e $\langle \cdot, \cdot \rangle$ denota l'usuale prodotto scalare in $\mathbb{R}^n$. Definiamo quindi la funzione $\sigma : \mathbb{R}^n \to \mathbb{R}$ come:

$$\sigma(\mathbf{x}) = \begin{cases} C_\rho(\mathbf{x}) & \text{se} \quad \mathbf{x} \in S, \\ g(\mathbf{x}) & \text{se} \quad \mathbf{x} \notin S, \end{cases}$$

dove $g(\mathbf{x})$ è l'equazione del "paraboloide" definito in (1.1). Riguardo alla funzione $\sigma(\mathbf{x})$ è possibile dimostrare [GL98] i seguenti lemmi:

Lemma 1. *$\sigma(\mathbf{x})$ è una funzione di classe C^1.*

Lemma 2. **M** *è l'unico punto di minimo locale di $\sigma(\mathbf{x})$ in S.*

Con un procedimento analogo, sebbene maggiormente complesso, è possibile generare deformazioni polinomiali di grado 5, con le quali si costruisce una funzione $\tau(\mathbf{x})$, con punto di minimo **M**. Questa viene definita come:

$$\tau(\mathbf{x}) = \begin{cases} Q_\rho(\mathbf{x}) & \text{se} \quad \mathbf{x} \in S, \\ g(\mathbf{x}) & \text{se} \quad \mathbf{x} \notin S, \end{cases}$$

dove $Q_\rho(\mathbf{x})$ ha la forma:

$$\begin{aligned} Q_\rho(\mathbf{x}) = &\left[-\frac{6}{\rho^4}\frac{\langle \mathbf{x}-\mathbf{M}, t-\mathbf{M}\rangle}{\|\mathbf{x}-\mathbf{M}\|} + \frac{6}{\rho^5}A + \frac{1}{\rho^3}\left(1-\frac{\delta}{2}\right)\right]\|\mathbf{x}-\mathbf{M}\|^5 + \\ &+ \left[\frac{16}{\rho^3}\frac{\langle \mathbf{x}-\mathbf{M}, t-\mathbf{M}\rangle}{\|\mathbf{x}-\mathbf{M}\|} - \frac{15}{\rho^4}A - \frac{3}{\rho^2}\left(1-\frac{\delta}{2}\right)\right]\|\mathbf{x}-\mathbf{M}\|^4 + \\ &+ \left[-\frac{12}{\rho^2}\frac{\langle \mathbf{x}-\mathbf{M}, t-\mathbf{M}\rangle}{\|\mathbf{x}-\mathbf{M}\|} + \frac{10}{\rho^3}A + \frac{3}{\rho}\left(1-\frac{\delta}{2}\right)\right]\|\mathbf{x}-\mathbf{M}\|^3 + \\ &+ \frac{1}{2}\delta\|\mathbf{x}-\mathbf{M}\|^2 + f. \end{aligned}$$

Come prima $A = \|\mathbf{T}-\mathbf{M}\|^2 + t - f$.

Per $\tau(\mathbf{x})$ è possibile dimostrare [GL98]:

Lemma 3. *$\tau(\mathbf{x})$ è una funzione di classe C^2.*

Lemma 4. **M** *è l'unico punto di minimo locale di $\tau(\mathbf{x})$ in S.*

La procedura appena illustrata può essere reiterata per otternere un numero arbitrario di deformazioni nel dominio in questione, e di conseguenza un numero ben definito di punti di minimo locali noti.

1.2 La libreria `gkls`

A seguito dei risultati ottenuti [GL98] è stata scritta una libreria in C che permette l'effettivo utilizzo di queste classi di funzioni [GKLS03].

Abbiamo già osservato che, per definire univocamente una di queste funzioni, è necessario stabilire un certo numero di parametri tra loro correlati. All'aumentare della dimensione del problema e del numero di minimi locali desiderati, diventa sempre più macchinosa la determinazione di tali parametri. Inoltre diventa sempre più difficile la generazione di differenti funzioni aventi proprietà di base similari.

In [GKLS03], oltre ai due (`D-type` e `D2-type`) tipi di funzioni definiti in [GL98], rispettivamente di classe C^1 e C^2, viene presentato un terzo tipo di funzioni di test (`ND-type`), di classe C^0 e viene proposto un generatore per tutte e tre le tipologie.

Il generatore necessita di una serie di dati di base, richiesti all'utente, e genera casualmente, ma in modo riproducibile, i restanti parametri per definire univocamente la funzione. In particolare, per ogni *set* di dati forniti dall'utente, questo è in grado di generare 100 funzioni di test con le stesse caratteristiche di base. Al termine della generazione viene fornita all'utente una struttura dati, contenente tutte le informazioni rilevanti della classe di test.

Generazione delle classi di test

Descriviamo brevemente come viene generata una funzione `D-type`; le altre vengono prodotte in modo analogo. Le informazioni che vengono richieste all'utente per la creazione di una funzione sono:

- la dimensione N del problema, $N \geq 2$;
- il numero m di punti di minimo locale, $m \geq 2$ incluso il punto di minimo $\mathbf{T}$ del parabolide (la scelta dei punti verrà fatta casualmente);
- il valore f^* del punto di minimo globale $\mathbf{x}^*$;
- il raggio ρ^* relativo alla deformazione contenente il punto di minimo globale $\mathbf{x}^*$;
- la distanza r^* fra il vertice $\mathbf{T}$ del paraboloide ed il punto di minimo globale;
- un valore $1 \leq n \leq 100$ che specifica quale funzione scegliere tra le 100 possibili generabili a partire dai dati precedenti.

Gli altri parametri della funzione sono scelti in maniera casuale, utilizzando il generatore di numeri casuale proposto in [Knu97].

Vengono imposti alcuni vincoli nell'inserimento dei dati, con riferimento alla (1.1):

$$f^* < t \equiv g(\mathbf{T}). \tag{1.2}$$

L'equazione (1.2) impone di fissare il minimo globale distinto dal minimo del paraboloide.

Definiamo il dominio D come l'iperrettangolo

$$D = \{[a(1), b(1)] \times \ldots \times [a(N), b(N)] \subseteq \mathbb{R}^N\}.$$

Poniamo quindi il vincolo

$$0 < r^* < 0.5 \min_{1 \leq j \leq N} |b(j) - a(j)|. \tag{1.3}$$

L'equazione (1.3) mantiene il punto di minimo globale all'interno del dominio anche nel caso in cui il vertice del parabolide si trovi nel centro del dominio stesso. Infine si impone il vincolo

$$0 < \rho^* \leq 0.5 r^*, \tag{1.4}$$

per mantenere una certa distanza tra il punto di minimo globale ed il vertice del parabolide.

Per prima cosa vengono generate casualmente[2] le coordinate di $\mathbf{T}$, quelle del punto di minimo globale $\mathbf{x}^*$, e quelle degli altri punti di minimo locale $\mathbf{M}_i$. A questo punto si generano i raggi d'attrazione dei vari minimi locali, controllando sempre che le regioni di "deformazione" non si sovrappongano.

[2]Come già accenntato il termine *casuale* utilizzato da qui in avanti, assume il significato di *pseudo-casuale*. I valori sono quindi generati casualmente ma a partire da un *seed* legato al numero di test che abbiamo scelto. Questo permette la riproducibilità di un test senza pregiudicarne l'arbitrarietà dei parametri.

Capitolo 2

La libreria `gkls2matlab`

Per consentire l'interazione tra `gkls` e MATLAB è stato utilizzato l'ambiente di sviluppo `mex`, una parte del vasto *framework* di MATLAB.

Questo consente la creazione di funzioni utilizzabili direttamente nel linguaggio MATLAB, ma scritte in linguaggi di programmazione esterni, in particolare in Fortran ed in C. L'integrazione di parti di codice compilato all'interno di un linguaggio di *scripting*, permette un notevole miglioramento delle prestazioni, soprattutto se questo viene effettuato nelle parti di maggior carico di lavoro. Il `mex` permette inoltre di utilizzare librerie esplicitamente create per l'utilizzo in ambiente C o Fortran, a condizione di scrivere una serie di funzioni, nel linguaggio compilato, che servano da collegamento tra la libreria ed il linguaggio di *scripting*.

La maggiore difficoltà nel connnettere questi due ambienti è stata la loro diversa natura. Infatti, mentre `gkls` è progettato per essere parte integrante di un programma compilato, e quindi residente in memoria dall'inizio alla fine dell'esecuzione di questo, il metodo di funzionamento di `mex` è quello di una funzione, che risiede in memoria centrale solo durante la sua esecuzione.

Per permettere un facile utilizzo della libreria, è stato scelto di generare i parametri del problema ad ogni chiamata della funzione. La principale controindicazione a questo approccio è che esso genera delle operazioni ripetute in caso che vengano fatte varie chiamate allo stesso problema di test. Per ovviare a questo inconveniente è stata introdotta la possibilità di far agire la funzione su dati vettoriali, oltre che su punti sigoli.

In questo modo, le operazioni più onerose, come ad esempio la valutazione della funzione in una *mesh* di dati per la visualizzazione, possono essere svolte con un'unica chiamata di funzione, con la conseguenza di ridurre al minimo le operazioni ridondanti. Ad esempio, in `GklsGUI`, ogni volta che viene visualizzata la funzione, questa vene valutata in una *mesh* di 200×200 (40.000) punti, ma il problema viene istanziato una sola volta.

Oltre alla funzione `gkls2matlab`, che genera problemi di test in $\mathbb{R}^n$, è stata sviluppata una libreria specifica per generare classi di test in $\mathbb{R}^2$, `gkls2matlab2`. Quest'ultima si differenzia dalla precedente per i parametri di ingresso e di uscita che può trattare.

2.1 Funzionamento di base

Le librerie `gkls2matlab` seguono le specifiche di utilizzo della libreria `gkls`, aggiungendo i dovuti controlli sugli *input*, in modo da prevenire eventuali *memory leaks* e restituire il controllo delle eccezioni all'utente MATLAB.

In maggior dettaglio, nell'ordine specificato:

- effettuano controlli sul numero e sul tipo degli argomenti di ingresso;
- forniscono a `gkls` i parametri che individuano una specifica classe di test;
- generano il problema test;
- effettuano le operazioni richieste dall'utente sulla funzione;
- preparano i dati di *output* in una forma intelleggibile da MATLAB;
- terminano la propria esecuzione.

2.2 Interfaccia di `gkls2matlab`

La sintassi della funzione è:

```
res = gkls2matlab( dim, num_minima, global_value, global_dist,
                   global_radius, action, test_number
                   [, target_points [, der_var1 [, der_var2]]])
```

2.2.1 Input

L'interfaccia di ingresso della funzione è composta da un primo gruppo di parametri **fisso** che specifica il problema di test, e da un secondo gruppo di parametri **variabile** che indica l'operazione da eseguire ed i relativi dati. Accanto ad ogni parametro viene specificata la "dimensione" della matrice (mxArray è il tipo di dato standard di MATLAB) che deve essere fornita alla funzione.

Parametri di ingresso:

- `dim` (mxArray 1×1) dimensione del problema, deve essere `dim` ≥ 2;
- `num_minima` (mxArray 1×1) numero di minimi locali, compresi quello globale e il minimo del parabolide;
- `global_value` (mxArray 1×1) valore desiderato della funzione nel punto di minimo globale;
- `global_dist` (mxArray 1×1) distanza desiderata tra il punto di minimo globale ed il punto corrispondente al vertice del parabolide;
- `global_radius` (mxArray 1×1) raggio di attrazione del punto di minimo globale;
- `action` (mxArray 1×1) operazione da svolgere, da scegliere tra:
  - `'info'` ottenere le informazioni sui punti di minimo;
  - `'ND_func'` valutare i `target_points` nella funzione **non derivabile**;
  - `'D_func'` valutare i `target_points` nella funzione **derivabile**;
  - `'D2_func'` valutare i `target_points` nella funzione **derivabile due volte**;
  - `'D_deriv'` valutare i `target_points` nella derivata; funzione **derivabile**
  - `'D2_deriv1'` valutare i `target_points` nella derivata prima della funzione **derivabile due volte**;
  - `'D2_deriv2'` valutare i `target_points` nella derivata seconda della funzione **derivabile due volte**;
  - `'D_gradient'` ottenere il vettore gradiente della funzione **derivabile**, calcolato nel punto `target_points`;
  - `'D2_gradient'` ottenere il vettore gradiente della funzione **derivabile due volte**, calcolato nel punto `target_points`;
  - `'D2_hessian'` ottenere la matrice hessiana della funzione **derivabile due volte**, calcolata nel punto `target_points`;
- `test_number` (mxArray 1×1) valore tra 1 e 100 per scegliere il test;
- `target_points` (mxArray $dim \times n$) array di n punti in cui valutare la funzione, ogni punto deve occupare una riga;
- `der_var1` (mxArray 1×1) indice della variabile secondo cui si deve effettuare la prima derivata parziale;

- `der_var2` (mxArray 1×1) indice della variabile secondo cui si deve effettuare la seconda derivata parziale;

2.2.2 Output

A seconda del tipo di operazione eseguita l'*output* della funzione avrà un diverso formato:

- nel caso di una richesta di tipo `'info'` si otterrà una matrice con m righe e n colonne, con m pari al numero di punti di minimo locali e n pari alla dimensione del problema + 2.
  Ogni riga specifica informazioni su un punto di minimo $\mathbf{M}_i$: nelle colonne in ordine troviamo il valore del punto $f(\mathbf{M}_i)$, il suo raggio d'attrazione ρ_i, e le coodinate, una per colonna.
  In particolare nella prima riga si troveranno le informazioni relative al vertice del paraboloide, mentre nella seconda quelle relative al punto di minimo globale;

- nel caso di una richiesta di valutazioni della funzione o di una sua derivata parziale in n punti, la libreria restituisce un vettore di dimensione n contente i valori richiesti;

- nel caso di una richiesta della valutazione del gradiente in un punto, viene restituito un vettore di dimensione n, con n dimensione del problema;

- nel caso della richiesta della valutazione dell'hessiana in un punto, viene restituita una matrice di dimensione $n \times n$, con n dimensione del problema.

2.3 Interfaccia di `gkls2matlab2`

La differenza sostanziale di questa funzione è che assume essere 2 la dimensione del problema. In questo modo è possibile trattare in modo più comodo le quantità in `input` e in `output`, avvicinandosi alla notazione standard di MATLAB.

Questa caratteristica risulta particolarmente utile in un ottica didattica della libreria.

La sintassi della funzione è:

```
res = gkls2matlab2( num_minima, global_value, global_dist,
                    global_radius, action, test_number
```

```
[, target_points_x, target_points_y
[, der_var1 [, der_var2]]])
```

2.3.1 Input

L'interfaccia di ingresso della funzione è del tutto simile alla precedente, ad eccezione del modo in cui vengono passati i punti.

Questa volta abbiamo 2 parametri in cui inserire i punti `target_points_x` e `target_points_y`, contenenti le rispettive coordinate x e y.

Le funzionalità della libreria rimangono invariate, ma adesso è possibile valutare input di ogni formato, punti, vettori di punti, o matrici di punti che siano.

Accostandosi in questo modo alla notazione "vettoriale" standard di MATLAB, alcune operazioni diventano immediate, come ad esempio la valutazione di una funzione in una *mesh* di dati.

2.3.2 Output

Allo stesso modo, l'interfaccia di uscita della `gkls2matlab2` è diversa dalla precedente solo nel caso di una richiesta di valutazioni della funzione o di una sua derivata parziale in n punti. Infatti, in questo caso viene restituita una matrice che ha le stesse dimensioni dei parametri `target_points_x` e `target_points_y`.

2.4 Esempi di utilizzo

Essendo librerie scritte in `mex`, queste sono dipendenti dalla piattaforma di utilizzo. È quindi innanzitutto necessario assicurarsi di possedere il file adeguato al proprio sistema. MATLAB utilizza estensioni diverse per i files di libreria a seconda delle varie piattaforme (Si veda la tabella 2.1).

Ad esempio, in ambiente Windows, con una versione di Matlab precedente alla 7.0.4, saranno necessari i files `gkls2matlab.dll` e `gkls2matlab2.dll`. In caso che non li si possegga sarà necessario compilarli.

Per compilare le librerie è necessario innanzitutto creare una *directory* per i files sorgenti. Qui andranno inseriti sia i files della libreria `gkls`:

- `gkls.c`;
- `rnd_gen.c`;

sia quelli sviluppati in quest'ambito:

- `gkls2matlab.c`;
- `glks2matlab2.c`.

All'interno dell'ambiente MATLAB a questo punto, dopo aver scelto come *directory* di esecuzione quella appena creata, si potrà procedere alla compilazione mediante i comandi

```
mex gkls2matlab.c
mex gkls2matlab2.c
```

In ambiente *Windows*, è necessario utilizzare il compilatore proprietario *VisualC++*, dato che `LCC`, il compilatore interno a MATLAB, non riesce a generare correttamente la libreria `gkls`.

Tabella 2.1: Estensioni dei `mex` files nei vari sistemi operativi

Sistema Operativo	**Estensione Mex file**
Sun Solaris	.mexsol
HP-UX	.mexhpux
Linux su x86	.mexglx
Linux su AMD Opteron	.mexa64
Linux su Intel Itanium2	.mexi64
MacOS X su PowerPC	.mexmac
Windows (MATLAB $>$ 7.0.4)	.mexw32
Windows (MATLAB $\leq$ 7.0.4)	.dll

Mostriamo adesso come è possibile utilizzare le 2 librerie per creare il grafico di una funzione di test.

Innanzitutto generiamo la *mesh* dei punti in cui vogliamo valutare la funzione.

```
>> [a,b]=meshgrid([-1:.4:1])

a =

   -1.0000   -0.6000   -0.2000    0.2000    0.6000    1.0000
   -1.0000   -0.6000   -0.2000    0.2000    0.6000    1.0000
   -1.0000   -0.6000   -0.2000    0.2000    0.6000    1.0000
   -1.0000   -0.6000   -0.2000    0.2000    0.6000    1.0000
   -1.0000   -0.6000   -0.2000    0.2000    0.6000    1.0000
   -1.0000   -0.6000   -0.2000    0.2000    0.6000    1.0000


b =

   -1.0000   -1.0000   -1.0000   -1.0000   -1.0000   -1.0000
   -0.6000   -0.6000   -0.6000   -0.6000   -0.6000   -0.6000
   -0.2000   -0.2000   -0.2000   -0.2000   -0.2000   -0.2000
    0.2000    0.2000    0.2000    0.2000    0.2000    0.2000
    0.6000    0.6000    0.6000    0.6000    0.6000    0.6000
    1.0000    1.0000    1.0000    1.0000    1.0000    1.0000
```

A questo punto usamo la libreria `gkls2matlab2` per valutare la *mesh*, e possiamo subito farne il grafico (Figura 2.1).

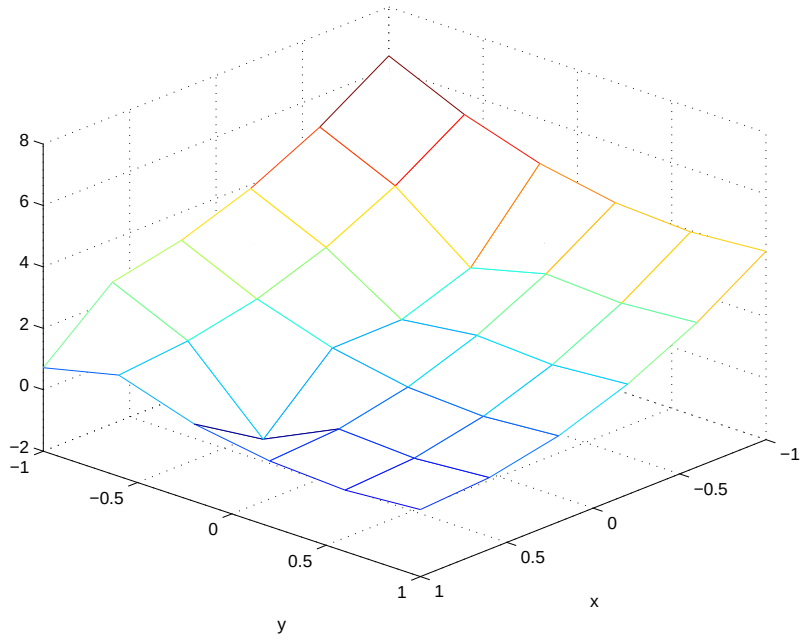

Figura 2.1: Esempio di grafico di una funzion di test

```
>> c=gkls2matlab2(5,-1,.8,.4,'D2_func',38,a,b)

c =

    6.4140    4.9841    3.8743    3.0844    2.6104    0.7154
    5.3181    3.8882    2.7784    1.9885    1.5187    1.2923
    4.5421    2.0365    1.2440    1.2126   -0.8692    0.5212
    4.0862    2.6564    1.5465    0.7567    0.2834    0.1370
    3.9503    2.5205    1.4106    0.6208    0.1509    0.0011
    4.1344    2.7046    1.5947    0.8049    0.3350    0.1852

>> mesh(a,b,c)
```

Possiamo eseguire la stessa operazione con `gkls2matlab`, ma con delle adeguate trasformazioni delle matrici.

```
>> ab=[reshape(a,36,1) reshape(b,36,1)]

ab =

   -1.0000   -1.0000
   -1.0000   -0.6000
   -1.0000   -0.2000
```

```
    -1.0000    0.2000
    -1.0000    0.6000
    -1.0000    1.0000
    -0.6000   -1.0000
    -0.6000   -0.6000
    -0.6000   -0.2000
    -0.6000    0.2000
    -0.6000    0.6000
    -0.6000    1.0000
    -0.2000   -1.0000
    -0.2000   -0.6000
    -0.2000   -0.2000
    -0.2000    0.2000
    -0.2000    0.6000
    -0.2000    1.0000
     0.2000   -1.0000
     0.2000   -0.6000
     0.2000   -0.2000
     0.2000    0.2000
     0.2000    0.6000
     0.2000    1.0000
     0.6000   -1.0000
     0.6000   -0.6000
     0.6000   -0.2000
     0.6000    0.2000
     0.6000    0.6000
     0.6000    1.0000
     1.0000   -1.0000
     1.0000   -0.6000
     1.0000   -0.2000
     1.0000    0.2000
     1.0000    0.6000
     1.0000    1.0000

>> d=gkls2matlab(2,5,-1,.8,.4,'D2_func',38,ab)

d =

    6.4140
    5.3181
    4.5421
    4.0862
    3.9503
    4.1344
    4.9841
    3.8882
    2.0365
    2.6564
    2.5205
    2.7046
    3.8743
    2.7784
    1.2440
    1.5465
    1.4106
    1.5947
    3.0844
    1.9885
    1.2126
    0.7567
    0.6208
    0.8049
```

```
    2.6104
    1.5187
   -0.8692
    0.2834
    0.1509
    0.3350
    0.7154
    1.2923
    0.5212
    0.1370
    0.0011
    0.1852
```

```
>> reshape(d,6,6)

ans =

    6.4140    4.9841    3.8743    3.0844    2.6104    0.7154
    5.3181    3.8882    2.7784    1.9885    1.5187    1.2923
    4.5421    2.0365    1.2440    1.2126   -0.8692    0.5212
    4.0862    2.6564    1.5465    0.7567    0.2834    0.1370
    3.9503    2.5205    1.4106    0.6208    0.1509    0.0011
    4.1344    2.7046    1.5947    0.8049    0.3350    0.1852
```

```
>> mesh(a,b,reshape(d,6,6))
```

Mostriamo infine i dati del problema. Ogni riga descrive un punto di minimo. Le colonne esprimono il valore del punto, il suo raggio d'attrazione e le sue coordinate.

```
>> info=gkls2matlab(2,5,-1,.8,.4,'info',38)

info =

         0    0.3960    0.9873    0.5699
   -1.0000    0.4000    0.6752   -0.1667
    1.0808    0.2050   -0.1700   -0.2560
    1.9987    0.2050   -0.5771   -0.1804
    0.5520    0.4060    0.9717   -0.9206
```

Capitolo 3

GklsGUI

Sulla base di `gkls2matlab` è stata sviluppata una applicazione che ha come obiettivo quello di creare un "banco di lavoro" nel quale sia possibile testare vari algoritmi di ottimizzazione su una vasta gamma di funzioni obiettivo. Il funzionamento di `GklsGUI` è riassumibile in quattro operazioni fra loro distinte:

- la generazione di un problema, che consta a sua volta nella definizione di una funzione da analizzare e delle condizioni iniziali di ricerca;
- la scelta di un algoritmo di risoluzione;
- l'applicazione dell'algoritmo al problema prescelto;
- l'analisi dei risultati.

L'intento è stato quello di mantenere una accentuata modularità delle varie parti dell'applicazione, per consentirne una facile estendibilità anche senza la conoscenza approfondita di tutta l'infrastruttura esistente.

In questo modo risulta molto semplice per l'utente aggiungere dei nuovi algoritmi di minimizzazione a quelli già implementati.

È stata inoltre utilizzata quanto più possibile l'estensione *object oriented* del linguaggio MATLAB, per la realizzazione della classe astratta `problem`.

Questo consente di aggiungere con semplicità nuovi tipi di funzioni su cui sperimentare gli algoritmi. Ne è un esempio la funzione "banana" di Rosembrock, che è stata aggiunta al programma solo alla fine del ciclo di sviluppo dello stesso.

3.1 La struttura del *tester* di problemi

Il compito di seguire l'evoluzione del problema in esame durante la sua permanenza all'interno di `GklsGUI` è affidato alla classe `problem`. Questa si occupa quindi della generazione del problema, della sua valutazione ai fini di visualizzazione e analisi, di registrare la quantità di operazioni che vengono richieste, in modo da consentire una successiva analisi della risoluzione.

A causa del metodo con cui sono implementati gli oggetti (per valore), e grazie alla totale trasparenza dell'utilizzo del linguaggio *Java* all'interno degli script MATLAB, è stata inserita all'interno di `problem` una classe *Java*, `AlgExaminer`.
Questo permette di rendere trasparente all'implementatore di algoritmi l'immagazzinamento delle operazioni da lui svolte sul problema, consentendo quindi una corretta analisi del costo computazionale del metodo utilizzato.

3.1.1 La classe `problem` ed i suoi "figli"

Non implementando MATLAB completamente il modello *Object Oriented*, non esistono ad esempio le **classi astratte** o le **interfacce**. Per questo la classe `problem` ha richiesto alcuni artifici inusuali per ottenere un buon compromesso tra facilità di utilizzo ed estendibilità.

Il suo modo di utilizzo è quello tipico di una classe astratta. Il programmatore che voglia realizzare un nuovo tipo di problema non deve fare altro che creare una classe "figlia" che implementi i metodi astratti per `problem`. La parte di lavoro relativa al salvataggio delle operazioni svolte è implementata direttamente in `problem`.

A questo punto l'utente ha a disposizione un nuovo tipo di problema da risolvere con la stessa interfaccia di utilizzo generica di `problem`.

In ordine alfabetico descriviamo i metodi pubblici della classe, che ne delineano la sua interfaccia.

Il primo parametro di ogni metodo è **p**, l'oggetto `problem` su cui deve essere eseguito il metodo.

r = evalMesh(p,a,b,type)

Funzione di servizio per la valutazione del problema in una *mesh* di dati ai fini della sua visualizzazione. Queste operazioni non vengono conteggiate ai fini dell'analisi dell'algoritmo.
a e **b** sono le due *mesh* secondo le variabili x e y.
type descrive il tipo di valutazione da effettuare:

1 funzione;

21 derivata parziale in x;

22 derivata parziale in y;

31 seconda derivata parziale in xx;

32 seconda derivata parziale in xy;

33 seconda derivata parziale in yy.

r = getGrad(p,x)

Restitusce il vettore gradiente del problema **p** calcolato nel punto **x**.

r = getHess(p,x)

Restitusce la matrice hessiana del problema **p** calcolata nel punto **x**.

r = getMinInfo(p)

Restituisce una matrice $3 \times n$, con n numero di minimi di **p**, contenente in ogni riga il valore di ogni punto di minimo nella prima colonna, e la sua posizione nelle restanti.

[x,y,z] = getPath(p)

Restituisce i tre vettori **x y z** di dimensione n, con n uguale alla dimensione del problema **p**, che rappresentano il percorso seguito dall'algoritmo di minimo utilizzato.

x = getStart(p)

Restituisce il punto iniziale, specificato dall'utente, da cui deve partire l'algoritmo di ricerca.

x = getTol(p)

Restituisce la tolleranza richiesta dall'utente.

r = getValue(p,x)

Restitusce il valore della funzione obiettivo per il problema **p**, calcolata nel punto **x**.

r = nextPoint(p,x)

Salva il punto **x** come nuovo punto intermedio del percorso dell'algoritmo di minimo. Restituisce il numero di punti del percorso aggiornato al passo corrente.

3.1.2 La classe *Java* `AlgExaminer`

`AlgExaminer` è una semplice classe *Java* che consente alla classe `problem` di memorizzare dati di esecuzione del programma.

Essa permette infatti l'archiviazione delle risorse richieste da un algoritmo durante il suo ciclo di esecuzione ed è in grado di tenere traccia del "percorso" che questo segue.

L'utilizzo di un oggetto *Java* permette in questo caso di sollevare colui che scrive un algoritmo da ogni preoccupazione relativa al conteggio delle risorse utilizzate. Questo sarebbe stato impossibile utilizzando solamente un oggetto MATLAB, che avrebbe richiesto un assegnamento supplementare dopo ogni chiamata di un metodo.

In ordine alfabetico descriviamo i metodi pubblici della classe, che ne delineano la sua interfaccia.

public void addEval()

Incrementa di uno il numero delle valutazioni di funzione eseguite.

public void addGrad()

Incrementa di uno il numero delle valutazioni del vettore gradiente eseguite.

public void addHess()

Incrementa di uno il numero delle valutazioni della matrice hessiana funzione eseguite.

public int addPoint(double x, double y, double z)

Aggiunge il punto (x, y, z) al cammino percorso. Restituisce la nuova lunghezza del cammino.

public int getEval()

Restituisce il numero delle valutazioni di funzione eseguite.

public int getGrad()

Restituisce il numero delle valutazioni del vettore gradiente eseguite.

public int getHess()

Restituisce il numero delle valutazioni della matrice hessiana funzione eseguite.

public double[] getPoint(int index)

Restituisce il punto del cammino avente indice `index`

public int length()

Restituisce la lunghezza del cammino percorso.

public String toString()

Restituisce una breve descrizione dello stato corrente dell'oggetto.

3.1.3 Implementazione di un problema: `gklsProblem`

Come già esposto, l'implementazione di un nuovo problema richiede la creazione di una classe. Questa erediterà da `problem` il funzionamento di base, e, tramite la definizione di alcuni metodi, specificherà le sue caratteristiche peculiari.

Esponiamo questa costruzione, utilizzando come esempio `gklsProblem`.

Il costruttore

È necessario innanzitutto scrivere il costruttore della classe. Questo, come possiamo osservare nel codice, avrà 2 argomenti di input obbligatori, ed i restanti saranno relativi al problema in questione.

In particolare `firstPoint` e `tol` indicano rispettivamente il punto iniziale dell'algoritmo e la tolleranza richiesta.

Gli altri argomenti sono invece relativi alla generazione di un problema di tipo `gkls`:

mn numero totale di minimi locali

gmv valore della funzione nel punto di minimo globale

dgp distanza del punto di minimo globale dal vertice del paraboloide

ar raggio della regione d'attrazione del punto di minimo locale

cn classe del problema da generare

Nel costruttore viene quindi generato un oggetto `problem`, viene specificato che la nuova classe sarà discendente di `problem` e vengono inizializzati i campi dell'oggetto.

```
function  p  = gklsProblem(tol,maxiter,mn,gmv,dgp,ar,cn)
parent = problem(tol,maxiter);
p.minNum=mn;
p.globMinVal=gmv;
p.distGlobPar=dgp;
p.attrReg=ar;
p.classNum=cn;
p.tol=tol;
p.maxiter=maxiter;
info=gkls2matlab2(mn,gmv,dgp, ar,'info',cn);
p.parabVertex=[info(1,3); info(1,4)];
p = class(p, 'gklsProblem', parent);
```

problemDeriv1

Il metodo `problemDeriv1` restituisce i valori della derivata parziale prima rispetto alla variabile `d` ($d \in \{1, 2\}$) nei punti di coordinate `x` e `y`.

```
function [r] = problemDeriv1(p,x,y,d)

domain=inDomain(p,x,y);
r=domain.*gkls2matlab2(p.minNum,p.globMinVal,p.distGlobPar, ...
                p.attrReg,'D2_deriv1',p.classNum,domain.*x,domain.*y,d);
if d==1
    r=r+~domain.*(2*x-2*p.parabVertex(1));
else
    r=r+~domain.*(2*y-2*p.parabVertex(2));
end
```

problemDeriv2

Il metodo `problemDeriv2` restituisce i valori della derivata parziale seconda rispetto alle variabili `d1` e `d2`, nei punti di coordinate `x` e `y`.

```
function [r] = problemDeriv2(p,x,y,d1,d2)

domain=inDomain(p,x,y);
r=domain.*gkls2matlab2(p.minNum,p.globMinVal,p.distGlobPar, ...
                p.attrReg,'D2_deriv2',p.classNum,domain.*x,domain.*y,d1,d2);

sum=d1+d2;
if sum==2
    r=r+~domain.*(2*x.^0);
end
if sum==4
    r=r+~domain.*(2*y.^0);
end
```

problemGrad

Il metodo `problemGrad` restituisce il vettore gradiente relativo al punto di coordinate `x` e `y`.

```
function [ r ] = problemGrad(p,x,y)

if inDomain(p,x,y)
    r=gkls2matlab2(p.minNum,p.globMinVal,p.distGlobPar, ...
                p.attrReg,'D2_gradient',p.classNum,x,y);
else
    r=[2*x-2*p.parabVertex(1); 2*y-2*p.parabVertex(2)];
end
```

problemHess

Il metodo `problemHess` restituisce la matrice hessiana relativa al punto di coordinate `x` e `y`.

```
function [ r ] = problemHess(p,x,y)

if inDomain(p,x,y)
    r=gkls2matlab2(p.minNum,p.globMinVal,p.distGlobPar, ...
                p.attrReg,'D2_hessian',p.classNum,x,y);
else
    r=[2 0; 0 2];
end

end
```

problemMinInfo

Il metodo `problemMinInfo` restituisce una matrice contenente le informazioni relative ai punti di minimo del problema.

Ogni riga sarà relativa ad un punto; le colonne dovranno indicare per primo il valore del punto di minimo, poi le coordinate del punto stesso, una per colonna.

```
function [ r ] = problemMinInfo(p)
r=gkls2matlab2(p.minNum,p.globMinVal,p.distGlobPar, ...
                p.attrReg,'info',p.classNum);
[a,b]=size(r);
r=r(:,[1 3:b]); %elimino i raggi di attrazione
```

problemValue

Il metodo `problemValue` restituisce i valori della funzione obiettivo, calcolata nei punti con coordinate `x` e `y`.

```
function [ r ] = problemValue(p,x,y)

domain=inDomain(p,x,y);
r=domain.*gkls2matlab2(p.minNum,p.globMinVal,p.distGlobPar, ...
```

```
                p.attrReg,'D2_func',p.classNum,domain.*x,domain.*y);
r=r+~domain.*((x-p.parabVertex(1)).^2+(y-p.parabVertex(2)).^2);
```

3.2 Gli algoritmi di ottimizzazione

L'interfaccia per la comunicazione tra gli algoritmi di minimizzazione e l'interfaccia grafica, è stata matenuta più semplice possibile, in modo da facilitare la scrittura di nuovi algoritmi da parte di un utilizzatore esterno.

Ogni algoritmo ha infatti un solo parametro di ingresso ed uno di uscita, entrambi appartenenti alla classe `problem`.

Durante l'esecuzione, l'algoritmo prende i dati iniziali del problema dall'istanza di `problem`, ed ogni volta che necessita di nuove informazioni si rivolge sempre a questa.

A ricerca conclusa viene restituita come output la stessa istanza dell'oggetto, ma questa volta arricchita di tutte le informazioni necessarie all'analisi dell'algoritmo: il numero di valutazioni effettuate, il percorso dell'algoritmo, i risultati ottenuti.

Per inserire un nuovo algoritmo di minimizzazione nel programma è necessario inserire nella directory `multidimAlg` il file MATLAB relativo. Il programma terrà conto della nuova funzione al suo riavvio.

Oltre all'implementazione di alcuni metodi di minimizzazione standard, sono state definite alcune funzioni che permettono di utilizzare gli algoritmi di ricerca del pacchetto di ottimizzazione di MATLAB all'intereno di GklsGUI. Queste funzioni, `FMinValue` ed `FMinGrad`, utilizzano entrambi il comando `fminunc`, ma si distinguono per le diverse informazioni che forniscono al comando: la prima fornisce solamente il valore della funzione nei punti che il comando richiede, mentre la seconda mette a disposizione anche il valore del gradiente.

3.3 Il problema della ricerca di un punto di minimo

Possiamo in generale descrivere il problema della ricerca di un punto di minimo come

$$\min f(\mathbf{x})$$

$$f : D \subseteq \mathbb{R}^n \to \mathbb{R}$$

dove $\mathbf{x} \in D$, il dominio della funzione.

Grazie alla capacità dei moderni computer di poter svolgere un enorme numero di operazioni ripetitive efficientemente, la maggior parte degli algoritmi studiati per la risoluzione di problemi di ottimizzazione sono di tipo iterativo.

Tipicamente, nella ricerca di un vettore $\mathbf{x}^*$ soluzione del problema, viene innanzitutto scelto un punto iniziale $\mathbf{x}_0$, partendo dal quale l'algoritmo genera un vettore $\mathbf{x}_1$, più vicino alla soluzione. Il processo viene ripetuto, generando una successione di approssimazioni, $\mathbf{x}_2, \mathbf{x}_3, \ldots, \mathbf{x}_n, \ldots$ che, sotto opportune condizioni, convergono a $\mathbf{x}^*$.

L'iterazione viene terminata quando viene generato un punto sufficientemente vicino alla soluzione.

3.3.1 Line Search

Gran parte degli algoritmi implementati hanno una struttura di base in comune. Dal punto iniziale si sceglie una **direzione di ricerca**, scelta da ciascun metodo in modo diverso, in base a varie considerazioni.

Lungo la direzione prescelta viene effettuata una minimizzazione unidimensionale, in modo da ridurre la complessità del problema. Si parla, in questo caso, di metodi *line search*.

Possiamo descrivere questa tecnica come la ricerca di

$$\alpha_k = \arg\min_{\alpha \geq 0} f(\mathbf{x}_k + \alpha \mathbf{d}_k),$$

dove f è la funzione obiettivo, $\mathbf{x}_k$ indica l'approssimazione al passo k e $\mathbf{d}_k$ è la direzione di ricerca. La nuova approssimazione della soluzione viene espressa come

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k.$$

Una volta ottenuta un'approssimazione *accettabile* del punto di minimo lungo la direzione scelta, si genera una nuova direzione di ricerca e la procedura viene iterata.

Esistono vari metodi che implementano la ricerca unidimensionale, differenti per i tipi di dati che utilizzano, velocità di convergenza e robustezza rispetto al problema in esame.

In questa implementazione viene utilizzato il **metodo di Armijo**.

3.3.2 Il metodo di Armijo

Il metodo in questione è per costruzione *inaccurato* nel determinare il punto di minimo, ma questo fatto non pregiudica la convergenza del metodo multidimensionale sovrastante. Inoltre richiede solamente la valutazione della fun-

zione in un numero esiguo di punti, al contrario di altri metodi che richiedono successive valutazioni del vettore gradiente o della matrice hessiana.

L'idea essenziale del metodo è quella di scegliere un α che non sia nè troppo grande nè troppo piccolo (Figura 3.1, pag. 26).

A questo scopo definiamo la funzione

$$\varphi(\alpha) = f(\mathbf{x}_k + \alpha \mathbf{d}_k).$$

Scegliamo quindi due coefficienti ε e η, tali che $0 < \varepsilon < 1$ e $\eta > 1$. L'algoritmo parte da un valore di α fissato e va a cercare, modificando questo parametro tramite moltiplicazioni o divisioni per η, il massimo valore di α per cui valga la relazione

$$\varphi(\alpha) \leq \varphi(0) + \varepsilon \varphi'(0)\alpha. \tag{3.1}$$

La ricerca avviene in questo modo:

- se α soddisfa la (3.1), viene ripetutamente moltiplicato per η finché la diseguaglianza non è più soddisfatta. A questo punto viene scelto il penultimo valore di α;
- se α non soddisfa la (3.1) viene diviso per η finché non la soddisfa. Viene quindi scelto il valore corrente di α.

Per l'implementazione del metodo si veda pagina 62.

3.3.3 Il metodo del gradiente (*Steepest Descent*)

Uno tra i più noti metodi di ottimizzazione è certamente il **metodo del gradiente**.

La sua semplicità ne consente una accurata analisi teorica, che risulta assai più difficile, e talora impraticabile, con algoritmi più complessi. È infatti possibile dimostrarne la convergenza globale ed anche stimarne la velocità in base al tipo di problema in esame.

L'algoritmo iterativo può essere riassunto come

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \mathbf{g}_k$$

dove $\mathbf{g}_k = \nabla f(\mathbf{x})^T$, il gradiente di f trasposto, ed α_k è uno scalare non negativo che minimizza $f(\mathbf{x}_k - \alpha \mathbf{g}_k)$, per $\alpha \geq 0$. In pratica si esegue una *line search* nella direzione del vettore gradiente in $\mathbf{x}_k$, ma con verso contrario.

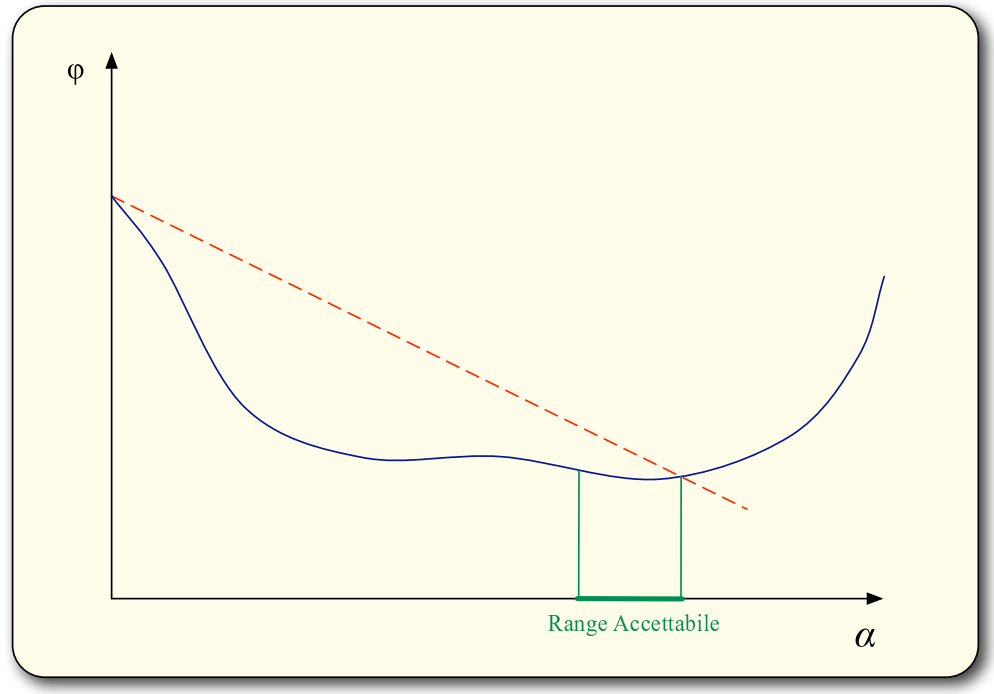

Figura 3.1: Metodo di Armijo: La linea blu rappresenta la funzione $\varphi(\alpha)$, la linea rossa tratteggiata rappresenta $\varphi(0) + \varepsilon\varphi'(0)\alpha$

Il caso quadratico

Particolarmente interessante risulta l'analisi del metodo del gradiente nel caso della ricerca del minimo per una funzione quadratica. Se poinamo

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T Q\mathbf{x} - \mathbf{x}^T\mathbf{b},$$

con Q matrice SDP, derivando otteniamo rispettivamente il gradiente e la matrice hessiana

$$\nabla f(\mathbf{x})^T = Q\mathbf{x} - \mathbf{b} \equiv \mathbf{g}$$
$$\nabla^2 f(\mathbf{x}) = Q.$$

In questo contesto è dunque possibile risolvere esplicitamente la minimizzazione unidimensionale

$$f(\mathbf{x}_k - \alpha\mathbf{g}_k) = \frac{1}{2}(\mathbf{x}_k - \alpha\mathbf{g}_k)^T Q(\mathbf{x}_k - \alpha\mathbf{g}_k) - (\mathbf{x}_k - \alpha\mathbf{g}_k)^T\mathbf{b};$$

derivando rispetto ad α e imponendo la derivata uguale a zero otteniamo:

$$a_k = \frac{\mathbf{g}_k^T \mathbf{g}_k}{\mathbf{g}_k^T Q \mathbf{g}_k}.$$

Il passo del gradiente nel caso quadratico risulta quindi essere dederminato dall'equazione

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \left(\frac{\mathbf{g}_k^T \mathbf{g}_k}{\mathbf{g}_k^T Q \mathbf{g}_k} \right) \mathbf{g}_k.$$

È inoltre possibile dimostrare [Lue89] il seguente teorema.

Teorema 1. *Per ogni $\mathbf{x}_0 \in \mathbb{R}^n$ il metodo del gradiente converge all'unico punto di minimo $\mathbf{x}^*$ della funzione quadratica f. Inoltre, posto $E(\mathbf{x}) = \frac{1}{2}(\mathbf{x} - \mathbf{x}^*)^T Q(\mathbf{x} - \mathbf{x}^*)$, si ha ad ogni passo k:*

$$E(\mathbf{x}_{k+1}) \leq \left(\frac{A - a}{A + a} \right)^2 E(\mathbf{x}_k)$$

dove A ed a sono rispettivamente l'autovalore massimo e minimo di Q.

Questo teorema, stabilisce la convergenza lineare del metodo, e la dipendenza della costante asintotica dell'errore dal numero di condizionamento $K = \frac{A}{a}$ della matrice Q.

Criterio di arresto (Aitken δ^2-process)

Nell'implementazione di questo metodo è stato possibile utilizzare un criterio di arresto basato su una stima del valore f^* del minimo globale. Questa stima segue dalla conoscenza della convergenza lineare del metodo:

$$\frac{E(\mathbf{x}_{k+1})}{E(\mathbf{x}_k)} \equiv \frac{f(\mathbf{x}_{k+1}) - f^*}{f(\mathbf{x}_k) - f^*} \approx c.$$

Scriviamo la stessa equazione al passo precedente

$$\frac{f(\mathbf{x}_k) - f^*}{f(\mathbf{x}_{k-1}) - f^*} \approx c$$

ottenendo, pertanto,

$$\frac{f(\mathbf{x}_{k+1}) - f^*}{f(\mathbf{x}_k) - f^*} \approx \frac{f(\mathbf{x}_k) - f^*}{f(\mathbf{x}_{k-1}) - f^*}.$$

Tramite semplici passaggi algebrici è possibile ricavare

$$f^* \approx \frac{f(\mathbf{x}_{k+1})f(\mathbf{x}_{k-1}) - f(\mathbf{x}_k)^2}{f(\mathbf{x}_{k+1}) - 2f(\mathbf{x}_k) + f(\mathbf{x}_{k-1})} \equiv f_k^*.$$

Grazie a questa stima e possibile stabilire un criterio d'arresto della forma

$$|f(\mathbf{x}_{k+1}) - f_k^*| \leq tol$$

con $tol > 0$, tolleranza stabilita a priori.

Per l'implementazione del metodo del gradiente si veda pagina 63.

3.3.4 Il metodo di Newton con smorzamento *(Newton Damped)*

L'idea di base del **metodo di Newton** è quella di generare una approssimazione quadratica locale $\widetilde{f}$ della funzione obiettivo f, e scegliere come passo duccessivo dell'algoritmo il minimo, ricavabile analiticamente, della $\widetilde{f}$. L'approssimazione viene generata attraverso uno sviluppo in serie di Taylor nell'intorno di x_k, troncato al secondo ordine:

$$\widetilde{f}(\mathbf{x}) = f(\mathbf{x}_k) + \nabla f(\mathbf{x}_k)(\mathbf{x} - \mathbf{x}_k) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_k)^T F(\mathbf{x}_k)(\mathbf{x} - \mathbf{x}_k),$$

dove con $F(\mathbf{x}_k) \equiv \nabla^2 f(\mathbf{x}_k)$ indichiamo la matrice hessiana nel punto $\mathbf{x}_k$. La $\widetilde{f}$ è minimizzata analiticamente nel punto

$$x_{k+1} = x_k - [F(\mathbf{x}_k)]^{-1} \nabla f(\mathbf{x}_k)^T.$$

Un risultato è dato dal seguente [Lue89]:

Teorema 2. *Sia $f : \mathbb{R}^n \to \mathbb{R}$, $f \in C^3$, e sia assunto che, nel punto di minimo locale $\mathbf{x}^*$, la matrice hessiana $F(\mathbf{x}^*)$ sia definita positiva. Allora, se si parte sufficientemente vicini ad $\mathbf{x}^*$, la successione di punti generata dal metodo di Newton converge a $\mathbf{x}^*$. L'ordine di convergenza è almeno 2.*

Nel programma è stata implementata una variante di questo metodo, chiamata **Newton con smorzamento**. Questa introduce il criterio di ricerca *line search* lungo la direzione indicata dal metodo di Newton. L'algoritmo sarà quindi della forma

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k [F(\mathbf{x}_k)]^{-1} \nabla f(\mathbf{x}_k)^T.$$

Il parametro α_k introdotto, risulterà circa uguale ad 1 per punti vicini ad $\mathbf{x}^*$, mantenendo quindi la convergenza locale quadratica. Inoltre, così facendo, si

previene il caso in cui, per il metodo di Newton, si ottiene $f(\mathbf{x}_{k+1}) > f(\mathbf{x}_k)$, che potrebbe verificarsi per funzioni non quadratiche.

Inoltre, per punti distanti dal minimo la matrice $[F(\mathbf{x}_k)]^{-1}$ potrebbe non essere **definita positiva**, e potrebbe quindi individuare direzioni di ricerca non di discesa. In questo caso, l'algoritmo di ricerca del parametro α_k che minimizza il problema unidimensionale, potrebbe determinare $\alpha_k = 0$. Di conseguenza tutto l'algoritmo verrebbe bloccato.

Prendiamo ad esempio il caso di $F(\mathbf{x}_k)$ **definita negativa**, quindi $f(\mathbf{x}_k)$ localmente concava. Questo darà:

$$f(\mathbf{x}_{k+1}) = f(\mathbf{x}_k - \alpha_k[F(\mathbf{x}_k)]^{-1}\nabla f(\mathbf{x}_k)^T) > f(\mathbf{x}_k)$$

per α_k abbastanza piccolo e positivo.

Per l'implementazione del metodo si veda pagina 63.

3.3.5 Il metodo di Levemberg-Marquardt

Una differente tipologia di metodi nasce con l'intento di raggiungere un proficuo equilibrio tra la desiderabile proprietà di convergenza globale del metodo del gradiente e l'ottima velocità di convergenza (locale) tipica del metodo di Newton. Possiamo descriverli con la formula generale

$$\mathbf{x}_{k+1} = \mathbf{x}_x - \alpha M_k \mathbf{g}_k,$$

dove al solito $\mathbf{g}_k = \nabla f(\mathbf{x}_k)^T$, α è un parametro di ricerca positivo ed $M_k \in \mathbb{R}^{n\times n}$.

Osserviamo che se poniamo $M_k = I$ otteniamo il metodo del gradiente, mentre se poniamo $M_k = [F(\mathbf{x}_k)]^{-1}$, otteniamo il metodo di Newton Damped.

Un modo di definire questa matrice è

$$M_k = [\varepsilon_k I + F(\mathbf{x}_k)]^{-1}$$

In questa maniera per ε_k piccoli ci troviamo vicino al metodo di Newton, mentre per ε_k grandi ci avvicinamo al metodo del gradiente.

E necessario però effettuare la scelta del parametro ε_k in modo da garantire la definita positività della matrice M_k. Fissando una soglia $\delta > 0$, vorremmo dunque imporre la condizione

$$\lambda_i > \delta \qquad \forall \lambda_i \in \sigma(M_k)).$$

Potremmo soddisfare questo vincolo scegliendo il parametro ε_k come

$$\varepsilon_k \geq \delta + \max(0, \lambda_{min}), \qquad \lambda_{min} = \min(\sigma(F(\mathbf{x}_k))).$$

Un possibile algoritmo per la determinazione del parametro potrebbe quindi avere la struttura:

- calcolare lo spettro di $F(\mathbf{x}_k)$. Questo ha un costo computazionale dell'ordine di $O(n^3)$;
- se $\lambda_{min} \leq 0$ si pone $\varepsilon_k = -\lambda_{min} + \delta$;
- se $\lambda_{min} > 0$ si pone $\varepsilon_k = 0$.

Sarebbe quindi necessario prima il calcolo dello spettro e poi una fattorizzazione per calcolare l'inversa della matrice M_k.

L'idea del metodo di Levemberg-Marquardt è invece quella di ricavare informazioni sullo spettro della matrice imponendo direttamente una fattorizzazione di tipo LDL^T, realizzabile solo su matrici SDP. In questo modo, incrementando il parametro ε_k ad ogni fattorizzazione fallita, si arriva ad avere una matrice M_k SDP già fattorizzata.

In particolare, nell'implementazione di questo metodo (pag. 64), viene effettuato un incremento del parametro ε_k pari a $\frac{\|F(\mathbf{x}_k)\|_\infty}{n}$.

Al passo k-esimo si pone inizialmente

$$\varepsilon_k = \max\left(0, \varepsilon_{k-1} - \frac{\|F(\mathbf{x}_k)\|_\infty}{n}\right), \tag{3.2}$$

e si tenta la fattorizzazione sull'hessiana non modificata. Se questa non riesce si prova con ε_k e, se necessario, si incrementa ancora fino al suo completamento.

In questo modo, se è stato raggiunto un punto con hessiana SDP si prosegue con un passo equivalente al metodo di Newton Damped ($\varepsilon_k = 0$), in caso contrario viene sfruttata l'informazione raccolta al passo precedente.

La (3.2) consente di ridurre gradualmente la ε nel caso di iterazioni successive in cui l'hessiana risulti SDP.

Per i dettagli implementativi si confronti pagina 64.

3.3.6 I metodi *quasi-newtoniani*

Nella pratica la valutazione e l'uso della matrice hessiana risulta una scelta costosa dal punto di vista computazionale. Da questa osservazione nasce la necessità di studiare dei metodi che risultino meno onerosi sotto questo aspetto, ma che si avvicinino alle proprietà di convergenza del metodo di Newton. L'idea fondamentale alla base dei metodi *quasi-newtoniani* è infatti quella di costruire l'inversa della matrice hessiana, o una sua approssimazione, utilizzando informazioni raccolte duarante l'esecuzione dell'algoritmo.

Si osservi un generico algoritmo del tipo

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k S_k \nabla f(\mathbf{x}_k)^T, \tag{3.3}$$

dove S_k è una matrice simmetrica $n \times n$, ed α_k è il parametro che minimizza $f(\mathbf{x}_{k+1})$. Per definire un metodo che sia di discesa anche per piccoli valori di α_k è necessario innanzitutto che S_k sia definita positiva.

Analizzando una funzione quadratica

$$f(x) = \frac{1}{2}\mathbf{x}^T Q\mathbf{x} - \mathbf{b}^T\mathbf{x} \tag{3.4}$$

e svolgendo considerazioni simili a quelle fatte per il metodo del gradiente, è possibile ricavare l'espressione esplicita del parametro

$$\alpha_k = \frac{\mathbf{g}_k^T S_k \mathbf{g}_k}{\mathbf{x}_k^T S_k Q S_k \mathbf{x}_k}.$$

È possibile inoltre dimostrare [Lue89] il seguente.

Teorema 3. *Definiamo $\mathbf{x}^*$ unico punto di minimo di f (3.4), e poniamo $E(\mathbf{x}) = \frac{1}{2}(\mathbf{x} - \mathbf{x}^*)^T Q(\mathbf{x} - \mathbf{x}^*)$. Allora per un algoritmo del tipo (3.3) è vero*

$$E(\mathbf{x}_{k+1}) \leq \left(\frac{B_k - b_k}{B_k + b_k}\right)^2 E(\mathbf{x}_k),$$

dove b_k e B_K sono, rispettivamente, il più piccolo ed il più grande autovalore della matrice $S_k Q$.

Per avere una buona convergenza nel caso quadratico sarebbe quindi auspicabile che $S_k \approx Q^{-1}$, condizione che renderebbe B_k e b_k vicini all'unità. Nel caso di una funzione obiettivo non quadratica l'analogo di Q è la matrice hessiana $F(\mathbf{x})$. Questo giustifica la scelta di una matrice $S_k \approx F(\mathbf{x}_k)^{-1}$.

È quindi necessario sviluppare una tecnica che consenta di trovare un'approssimazione dell'inversa dell'hessiana senza ricorrere al suo calcolo esplicito.

Definiamo due vettori $\mathbf{p}_k$ e $\mathbf{q}_k$, a partire da due punti $\mathbf{x}_k$ e $\mathbf{x}_{k+1}$, e dai rispettivi vettori gradiente $\mathbf{g}_{k+1}$ e $\mathbf{g}_k$, come

$$\mathbf{p}_k = \mathbf{x}_{k+1} - \mathbf{x}_k,$$

$$\mathbf{q}_k = \mathbf{g}_{k+1} - \mathbf{g}_k.$$

Consideriamo la relazione

$$\mathbf{g}_{k+1} - \mathbf{g}_k \approx F_k(\mathbf{x}_{k+1} - \mathbf{x}_k);$$

se consideriamo il caso quadratico (hessiana costante), otteniamo

$$\mathbf{q}_k = F\mathbf{p}_k.$$

In questa circostanza se conoscessimo n vettori $\mathbf{p}_1, \ldots, \mathbf{p}_n$ linearmente indipendenti, e i relativi $\mathbf{q}_1, \ldots, \mathbf{q}_n$, la matrice F risulterebbe univocamente determinata. In particolare se consideriamo $\mathbf{P}$ e $\mathbf{Q}$, le matrici formate dai vettori $\mathbf{p}_k$ e $\mathbf{q}_k$, possiamo ricavare

$$\mathbf{Q} = F\mathbf{P} \quad \Rightarrow \quad F = \mathbf{Q}\mathbf{P}^{-1}.$$

Nel nostro caso vogliamo generare una matrice H_{k+1} che approssimi l'inversa dell'hessiana. Si imporrà al generico passo k,

$$H_{k+1}\mathbf{q}_k = \mathbf{p}_k.$$

Il metodo DFP

Il metodo DFP, acronimo dei suoi autori Davidon, Fletcher e Powell, genera la matrice H_{k+1} ad ogni passo, sommando due matrici simmetriche di rango uno alla precedente approssimazione H_k. Per questo fatto il metodo viene spesso denominato *correzione di rango due*.

In particolare, se poniamo

$$\mathbf{p}_k = \mathbf{x}_{k+1} - \mathbf{x}_k,$$

$$\mathbf{q}_k = \mathbf{g}_{k-1} - \mathbf{g}_k,$$

si ha:

$$H_{k+1} = H_k + \frac{\mathbf{p}_k\mathbf{p}_k^T}{\mathbf{p}_k^T\mathbf{q}_k} - \frac{H_k\mathbf{q}_k\mathbf{q}_k^T H_k}{\mathbf{q}_k^T H_k \mathbf{q}_k}.$$

È possibile dimostrare [Lue89] i seguenti

Teorema 4. *Se H_k è una matrice simmetrica definita positiva, allora H_{k+1} è simmetrica definita positiva.*

Teorema 5. *Se $F_k \equiv F \equiv$ cost (funzione quadratica) allora:*

1. $\mathbf{p}_i^T F \mathbf{p}_j = 0, \qquad 0 \le i < j \le k;$
2. $H_{k+1} F \mathbf{p}_i = \mathbf{p}_i, \qquad 0 \le i \le k.$

La F-ortogonalità dei vettori $\mathbf{p}_0, \ldots, \mathbf{p}_{n-1}$ ci assicura la nonsingolarità della matrice $P = (\mathbf{p}_0, \ldots, \mathbf{p}_{n-1})$. Possiamo quindi derivare che

$$H_n F P = P \quad \Rightarrow \quad H_n F = I \quad \Rightarrow \quad H_n = F^{-1}.$$

L'implementazione del metodo pone $H_0 = I$, generando quindi alla prima iterazione un passo del metodo del gradiente.

Alle iterazioni successive la direzione $\mathbf{d}_k$ di ricerca della procedura *line search* è descritta come

$$\mathbf{d}_k = -H_k\mathbf{g}_k.$$

Per l'implementazione del metodo si veda pagina 64.

3.3.7 I metodi Trust Region

L'approccio *trust region* nasce dall'osservazione che l'approssimazione quadratica di una funzione f al passo k, in generale, è valida solamente in un opportuno intorno di $\mathbf{x}_k$.

I metodi *trust region* stabiliscono un intorno del punto corrente all'interno del quale "si fidano" dell'approssimazione quadratica della funzione obiettivo. Il nuovo passo dell'algoritmo viene infatti scelto come il punto che minimizza l'approssimazione quadratica, sia essa m_k, all'interno della *trust region*.

$$m_k(\mathbf{p}) = f_k + \nabla f_k^T \mathbf{p} + \frac{1}{2}\mathbf{p}^T B_k \mathbf{p}. \tag{3.5}$$

In simboli, si porrà

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{p}_k, \tag{3.6}$$

dove

$$\mathbf{p}_k = \arg\min m_k(\mathbf{p}), \qquad \|\mathbf{p}\| \leq \Delta_k, \tag{3.7}$$

in cui Δ_k indica il raggio della *trust region* e B_k è l'hessiana di f in $\mathbf{x}_k$, ovvero una sua approssimazione.

Fondamentale per la buona esecuzione di questi metodi è la scelta di Δ_k. Se infatti la regione è troppo piccola l'algoritmo perde l'opportunità di avvicinarsi significativamente al punto di minimo, se è troppo grande il punto di minimo della funzione obiettivo all'interno della regione potrebbe essere molto distante da quello dell'approssimazione quadratica.

Nella pratica la dimensione della *trust region* viene calcolata ad ogni passo in base alle prestazioni dell'algoritmo nei passi precedenti. Dato un passo $\mathbf{p}_k$ definiamo il parametro

$$\rho_k = \frac{f(\mathbf{x}_k) - f(\mathbf{x}_k + \mathbf{p}_k)}{m_k(\mathbf{0}) - m_k(\mathbf{p}_k)}. \tag{3.8}$$

Il numeratore è la *riduzione effettiva* della funzione obiettivo, mentre il denominatore è la sua *riduzione prevista* (dall'approssimazione quadratica).

A seconda del valore che assumerà ρ_k, sarà possibile dedurre informazioni sulla *trust region*. Se questo è circa 1 significa che c'è una buona concordanza tra il modello e la funzione f, e sarà quindi possibile ampliare la regione. Al contrario se questo è vicino a zero o negativo sarà necessario diminuire il raggio della *trust region*.

Per realizzare l'algoritmo è necessario scegliere un metodo per trovare il minimo dell'approssimazione quadratica della funzione obiettivo all'interno della *trust region* prescelta.

Sono stati implementati due diversi algoritmi di tipo *trust region*.

Il primo (`TrustRegCauchy.m`, pag. 66) genera l'approssimazione direttamente con la matrice hessiana e sceglie come nuovo passo il *punto di Cauchy*. Il secondo (`TrustRegDogleg.m`, pag. 67) utilizza la matrice generata dal metodo Levemberg-Marquardt, SDP per costruzione, e minimizza la regione tramite il metodo *Dogleg*.

Il punto di Cauchy

Per determinare il punto di Cauchy dobbiamo innanzitutto trovare il vettore che risolve l'equazione lineare

$$\mathbf{p}_k^{\mathrm{S}} = \arg\min f_k + \nabla f_k \mathbf{p}, \qquad \|\mathbf{p}\| \leq \Delta_k. \tag{3.9}$$

Osserviamo che questo corrisponde a

$$\mathbf{p}_k^{\mathrm{S}} = -\Delta_k \frac{\nabla f_k^T}{\|\nabla f_k\|},$$

ovvero il vettore gradiente con modulo uguale a Δ_k e verso opposto.

A questo punto cerchiamo lo scalare $0 < \tau_k \leq 1$ che minimizza l'approssimazione quadratica m_k:

$$\tau_k = \arg\min m_k(\tau \mathbf{p}_k^{\mathrm{S}}). \tag{3.10}$$

Per individuarlo dobbiamo considerare separatamente due varianti del problema: il caso $\nabla f_k^T B_k \nabla f_k < 0$ e quello $\nabla f_k^T B_k \nabla f_k > 0$.

Nel primo caso la (3.10) è una funzione strettamente decrescente. Scegliamo quindi $\tau_k = 1$.

Nel secondo caso la (3.10) è una parabola convessa con punto di minimo in

$$-\frac{\|\nabla f_k\|^2}{\nabla f_k B_k \nabla f_k^T} \nabla f_k^T. \tag{3.11}$$

È necessario quindi scegliere questo punto se la sua norma risulta minore di Δ_k. In definitiva possiamo esprimere il punto di Cauchy $\mathbf{p}_k$ come:

$$\mathbf{p}_k = -\tau_k \nabla f_k^T, \tag{3.12}$$

con

$$\tau_k = \begin{cases} \frac{\Delta_k}{\|\nabla f_k\|} & \text{se } \nabla f_k B_k \nabla f_k^T < 0; \\ \min\left(\frac{\|\nabla f_k\|^2}{\nabla f_k B_k \nabla f_k^T}, \frac{\Delta_k}{\|\nabla f_k\|} \right) & \text{altrimenti.} \end{cases} \tag{3.13}$$

In pratica il punto di Cauchy indica il punto che minimizza la funzione m_k lungo la direzione del metodo del gradiente, all'interno della *trust region*.

Questo metodo eredita dal metodo del gradiente la convergenza globale, ed è adatto a trattare anche punti in cui l'hessiama risulti non definita positiva.

Il metodo *Dogleg*

Questo metodo nasce con l'intento di migliorare l'accuratezza della ricerca del minimo dell'approssimazione quadratica m_k (3.5), sfruttando l'informazione di definita positività dell'approssimazione dell'hessiana B_k.

Il punto di minimo non vincolato per la funzione m_k, con hessiana SDP, è il passo del metodo di Newton $\mathbf{p}^{\mathrm{B}} = -B_k^{-1}\nabla f_k^T$. Quindi col vincolo della *trust region* la soluzione $\mathbf{p}^*$ della 3.5 sarà:

$$\mathbf{p}^*(\Delta) = \mathbf{p}^{\mathrm{B}}, \qquad \text{per } \Delta \geq \|\mathbf{p}^{\mathrm{B}}\|. \tag{3.14}$$

Allo stesso modo, per Δ piccoli, in cui quindi il termine quadratico è meno rilevante, possiamo supporre:

$$\mathbf{p}^*(\Delta) \approx \Delta\frac{\mathbf{p}^{\mathrm{U}}}{\|\mathbf{p}^{\mathrm{U}}\|}, \quad \text{con} \quad \mathbf{p}^{\mathrm{U}} = -\frac{\|\nabla f_k\|^2}{\nabla f_k^T B_k \nabla f_k}\nabla f_k, \qquad \text{per } \Delta \approx 0. \tag{3.15}$$

Il metodo *Dogleg* parte da questi presupposti e approssima la traiettoria di $\mathbf{p}^*(\Delta)$ tramite la spezzata:

$$\widetilde{\mathbf{p}}(\tau) = \begin{cases} \tau\mathbf{p}^{\mathrm{U}}, & 0 \leq \tau \leq 1, \\ \mathbf{p}^{\mathrm{U}} + (\tau - 1)(\mathbf{p}^{\mathrm{B}} - \mathbf{p}^{\mathrm{U}}), & 1 \leq \tau \leq 2. \end{cases} \tag{3.16}$$

Si cerca quindi il minimo della m_k lungo questo percorso, compatibilmente con il vincolo della *trust region*. Riguardo a $\widetilde{\mathbf{p}}(\tau)$ è possibile dimostrare [Lue89]:

Teorema 6. *Sia B_k simmetrica definita positiva. Si ha:*

1. *$\|\widetilde{\mathbf{p}}(\tau)\|$ è una funzione crescente rispetto a τ, ovvero l'intersezione con la frontiera della* trust region *è al più un punto;*

2. *$\|m_k(\widetilde{\mathbf{p}}(\tau))\|$ è una funzione decrescente rispetto a τ.*

Possiamo dunque schematizzare l'algoritmo in questo modo:

1. se $\|\mathbf{p}^{\mathrm{B}}\| \leq \Delta$, pongo $\mathbf{p}^* := \mathbf{p}^{\mathrm{B}}$;

2. altrimenti, se $\|\mathbf{p}^{\mathrm{U}}\| \geq \Delta$, pongo $\mathbf{p}^* := \Delta_k\frac{\mathbf{p}^{\mathrm{U}}}{\|\mathbf{p}^{\mathrm{U}}\|}$;

3. altrimenti risolvo l'equazione scalare $\|\mathbf{p}^{\mathrm{U}} + \tau(\mathbf{p}^{\mathrm{U}} - \mathbf{p}^{\mathrm{B}})\|^2 = \Delta^2$, ottenendo un valore $0 < \tau < 1$, e pongo $\mathbf{p}^* := \mathbf{p}^{\mathrm{P}} + \tau(\mathbf{p}^{\mathrm{B}} - \mathbf{p}^{\mathrm{P}})$.

Questo metodo risulta più efficiente rispetto alla ricerca del punto di Cauchy, ma richiede la definita positività dell'approssimazione dell'hessiana B_k. Nell'implemantazione, per garantire questa condizione, la matrice B_k è stata ricavata tramite il metodo di Levemberg-Marquardt.

3.4 L'interfaccia grafica

Per permettere un facile utilizzo delle librerie sviluppate, è stata preparata un'interfaccia grafica realizzata attraverso le funzioni grafiche dell'ambiente MATLAB.

Mediante questo strumento è possibile generare una vasta gamma di problemi test tramite la libreria `gkls`, visualizzarli, scegliere un algoritmo di risoluzione ed analizzarne i risultati. È inoltre possibile confrontare gli algoritmi su una funzione di test classica, la funzione "banana" di Rosembrock.

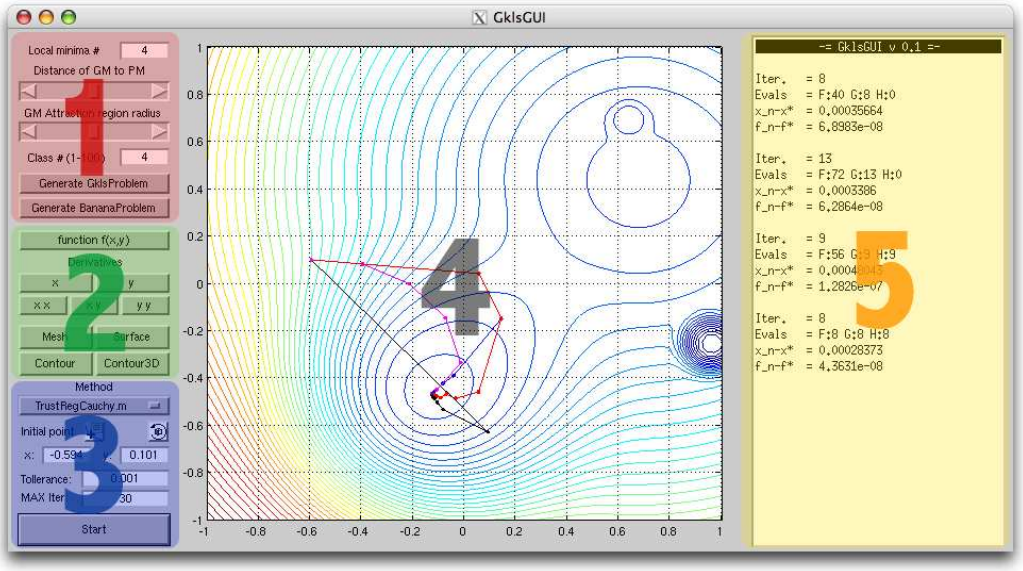

Figura 3.2: Le 5 parti in cui è suddivisa la GUI.

Come possiamo osservare in figura 3.2, l'interfaccia è suddivisa in 5 parti fondamentali:

- 1 Elementi necessari alla generazione di un problema di test.
- 2 Pulsanti relativi alla modalità di visualizzazione del problema in questione.
- 3 Elementi necessari all'impostazione dell'algoritmo di risoluzione.
- 4 Area di visualizzazione del problema.
- 5 Area di analisi dei risultati degli algoritmi.

3.4.1 Generazione del problema

Funzioni di tipo Gkls

In questa parte stabiliamo i parametri che verranno passati alla libreria `gkls` per generare il problema.

L'area di testo *"Local minima #"* indica il numero di minimi locali del problema; è necessario inserire un numero intero maggiore di 1.

Gli *slider "Distance of GM to PM"* e *"GM attraction region radius"* regolano la distanza tra il vertice del paraboloide ed il punto di minimo globale, ed il raggio della regione di attrazione del punto minimo globale.

L'area di testo *"Class #"* seleziona la classe di test desiderata tra le 100 disponibili.

Il pulsante *"Generate GklsProblem"* avvia la costruzione del problema selezionato. Questo elimina anche tutti i dati relativi ad algoritmi eventualmente testati in precedenza.

Funzione di Rosenbrock

Il pulsante *"Generate BananaProblem"* genera una funzione di Rosembrock centrata nell'origine, con punto di minimo in (1,1). La produzione di questo tipo di funzione è del tutto indipendente dai parametri sovrastanti, relativi solamente alle funzioni `Gkls`.

3.4.2 Visualizzazione

I primi sei pulsanti di questa sezione permettono di stabilire il tipo di informazione che si desidera visualizzare relativamente al problema. In ordine è possibile scegliere di visualizzare la funzione, le sue derivate parziali prime (figura 3.4) o le sue derivate parziali seconde, rispetto alle variabili desiderate.

Indipendentemente dal tipo di informazioni è possibile ottenere quattro diversi stili di visualizzazione:

Mesh Visualizza le informazioni in un grafico 3D poligonale.

Surface Visualizza le informazioni in una superficie uniforme (figure 3.3 e 3.6).

Contour Visualizza le informazioni in forma di "curve di livello" (figure 3.2 e 3.5).

Contour3D Visualizza le informazioni in forma di "curve di livello" tridimensionali (figura 3.4).

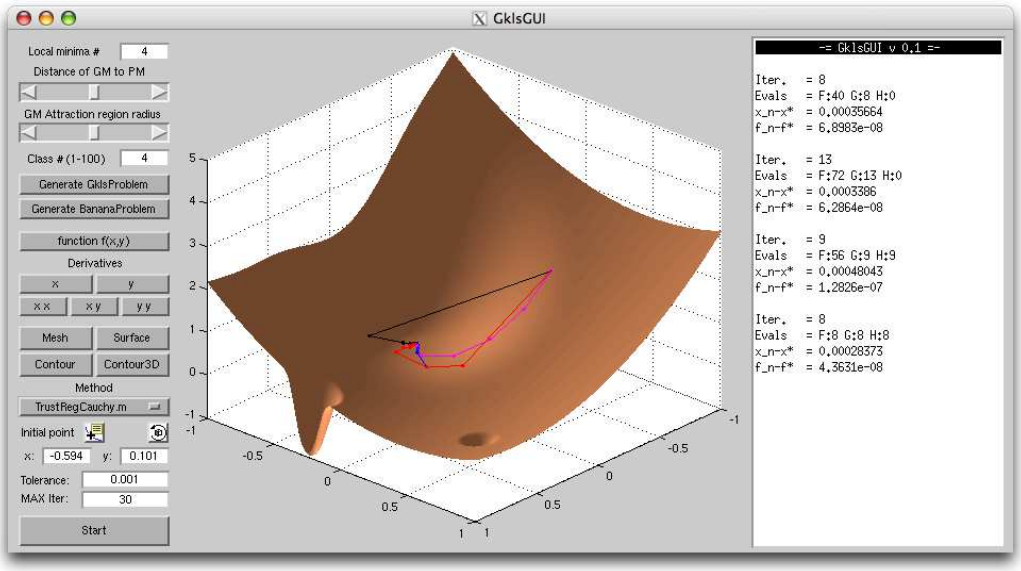

Figura 3.3: Una veduta tridimensionale del problema.

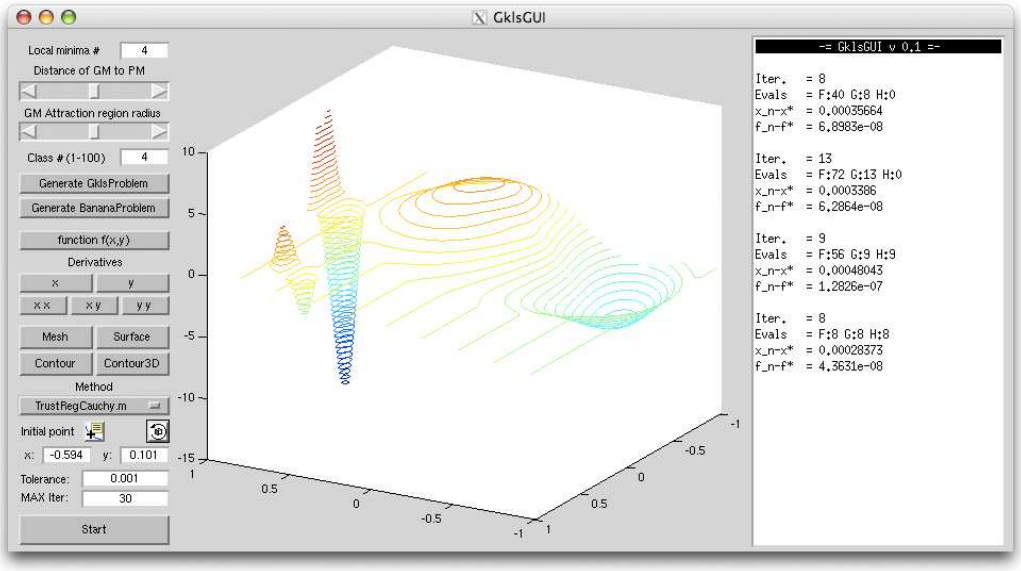

Figura 3.4: Grafico della derivata parziale rispetto alla variabile x.

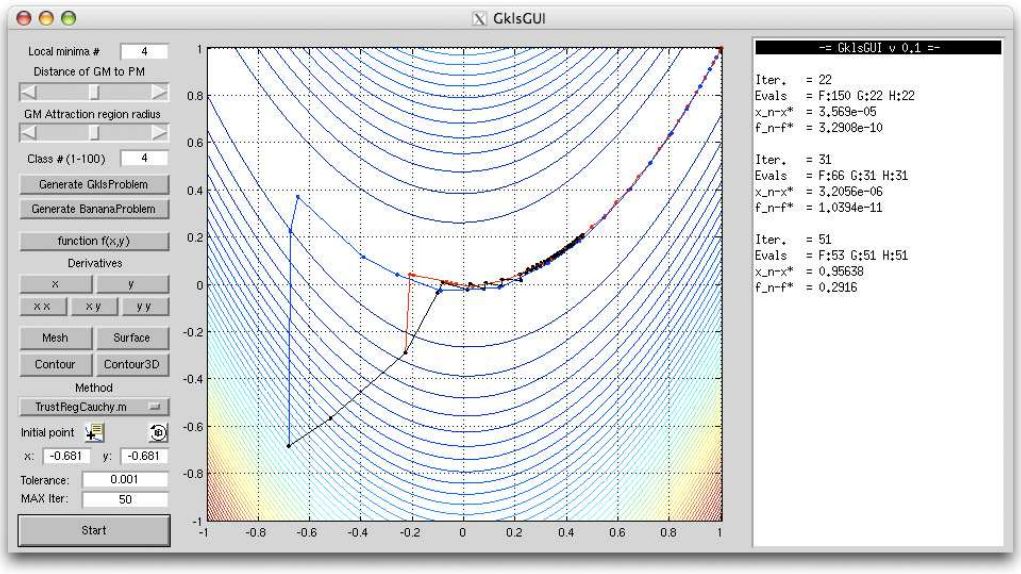

Figura 3.5: Esecuzione di alcuni algoritmi sulla funzione "banana" di Rosenbrock.

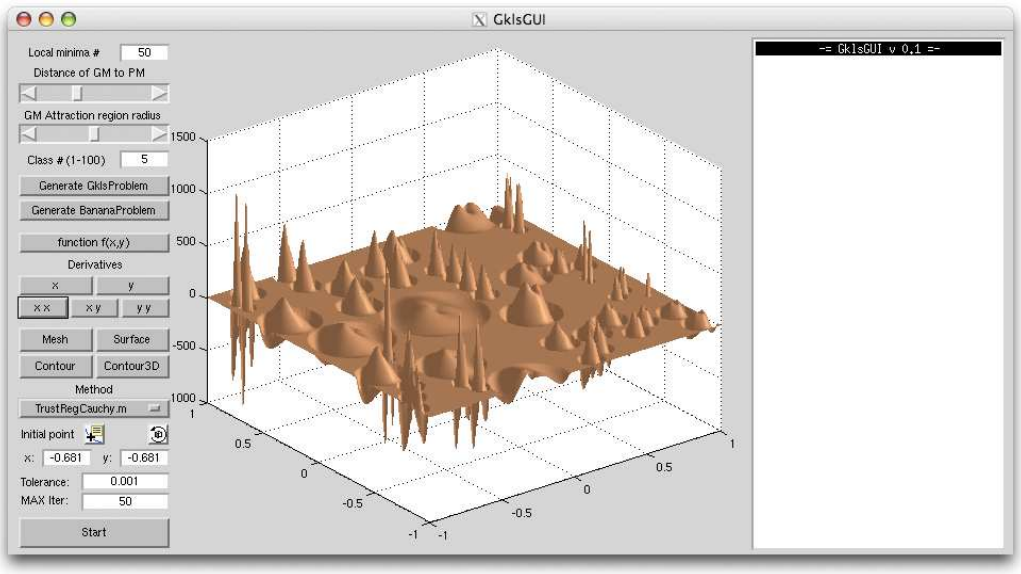

Figura 3.6: Grafico della seconda derivata parziale rispetto alla variabile x di un problema con 50 punti di minimo.

3.4.3 Risoluzione

Per determinare un algoritmo di risoluzione sarà necessario innanzitutto sceglierne la tipologia tramite il *popup "Method"*.

Successivamente è possibile scegliere il punto iniziale sia graficamente tramite il pulsante *"Initial Point"*, sia manualmente inserendolo nelle apposite aree di testo *"x"* e *"y"*.

La casella di testo *"Tolerance"* specifica la tolleranza dei criteri d'arresto.

La casella di testo *"MAX Iter"* specifica il massimo numero di iterazioni che un algoritmo deve eseguire.

Il pulsante *"Start"* determina l'esecuzione dell'algoritmo.

Per tutti i campi sono specificati dei valori di *default* generici, adatti ad una prima analisi dell'algoritmo.

3.4.4 Analisi

Al termine dell'esecuzione di ogni algoritmo, nell'area di testo alla destra del grafico vengono visualizzate alcune informazioni utili alla sua analisi.

In particolare:

`Iter.` Numero di iterazioni dell'algoritmo eseguite.

`Evals` Numero di valutazioni eseguite, suddivise in valutazioni della funzione (`F`), del gradiente (`G`) e dell'hessiana (`H`).

`x_n-x*` Distanza tra il punto di minimo trovato e quello effettivo.

`f_n-f*` Distanza tra il minimo trovato e quello effettivo.

Conclusioni

Il risultato di questo lavoro consiste nella realizzazione di due software, utili allo studio di metodi di minimizzazione.

La libreria `gkls2matlab`, progettata e sviluppata nel lavoro di tesi, risulta uno strumento utile sotto molti aspetti. Infatti, pur mantenendo la velocità ereditata dalla libreria `gkls`, permette lo studio di questa classe di problemi col valore aggiunto della potenza di un linguaggio interpretato.

In questo modo, grazie alla versatilità dell'ambiente MATLAB, è possibile testare metodi di risoluzione, analizzarne i risultati, visualizzarne ogni possibile aspetto in modo estremamente comodo. Tutto questo attraverso la forma compatta, e *standard de facto* del linguaggio MATLAB.

`GklsGUI` include questa libreria, prefigurandosi come un semplice strumento da utilizzare in ambito didattico.

Il software permette la generazione di una vasta gamma di "problemi" di minimo e l'immediata visualizzazione della funzione in esame, insieme ad alcune sue proprietà analitiche. Con semplici comandi è possibile eseguire e confrontare alcuni algoritmi di risoluzione standard, variando a piacere le condizioni iniziali e quelle di arresto.

Grazie all'attenzione posta all'estensibilità del codice, è inoltre possibile un più specifico utilizzo del programma. È infatti molto semplice includere nuovi algoritmi progettati dall'utente che vadano ad aggiungersi ai metodi di minimizzazione standard proposti. Questi potranno essere quindi subito analizzati e confrontati gli uni con gli altri in modo semplice ed intuitivo.

È stata inoltre considerata la possibilità di includere nuovi classi di problemi oltre a quelle già presenti.

Durante la fase di testing della libreria, a causa di uno strano comportamento della visualizzazione della derivata parziale seconda dei problemi di test, è stato possibile mettere in evidenza un *bug* che affliggeva la libreria `gkls`. Analizzando il codice sorgente è stato possibile risalire all'equazione in [GL98] che conteneva l'errore (pag. 216).

Reso noto questo fatto agli autori, è stato individuato un parametro errato nella stampa dell'articolo [GL98] ed è stata resa disponibile una nuova versione corretta della libreria [GKLS03].

Appendice A

Codici sorgente: le librerie

A.1 gkls2matlab.c

```
0
#define malloc  mxMalloc
#define free  mxFree

#include "gkls.c"
#include "rnd_gen.c"
#include <math.h>
#include <string.h>
#include "mex.h"

/* Operation types*/

#define INFO     0
#define ND_func    1
#define D_func     2
#define D2_func    3
#define D_deriv    4
#define D2_deriv1  5
#define D2_deriv2  6
#define D_gradient   7
#define D2_gradient  8
#define D2_hessian   9


/* Input Arguments */

#define DIM_IN          prhs[0]
#define NUM_MINIMA_IN     prhs[1]
#define GLOBAL_VALUE_IN     prhs[2]
#define GLOBAL_DIST_IN      prhs[3]
#define GLOBAL_RADIUS_IN    prhs[4]
#define ACTION_IN       prhs[5]
#define TEST_NUMBER_IN      prhs[6]
#define TARGET_POINTS_IN    prhs[7]
#define DERIV_VAR1_IN     prhs[8]
#define DERIV_VAR2_IN     prhs[9]

/* Output Arguments */
```

```
#define RESULT  plhs[0]

int isSingleReal(const mxArray *mxptr)
{
  return (mxGetM(mxptr)*mxGetN(mxptr)) == 1 && mxIsDouble(mxptr) && !
      mxIsComplex(mxptr);
}

void mexFunction( int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]
     )
{
    int test_number, action, deriv_var1, deriv_var2, i, j, m, n;

//  Check for proper number of arguments

    if (nrhs > 10) {
    mexErrMsgTxt("Too many input arguments.");
    }
    if (nrhs < 7) {
        mexErrMsgTxt("Too few input arguments.");
    }
    if (nlhs > 1) {
    mexErrMsgTxt("Too many output arguments.");
    }

//  Action check
    if(mxIsChar(ACTION_IN)){
        int strlen=mxGetN(ACTION_IN) + 1;
    char *actionStr=(char*)mxMalloc(strlen*sizeof(char));
    mxGetString(ACTION_IN, actionStr, strlen);

    if(strcmp(actionStr,"info")==0)
    {
      action=INFO;
    } else if(strcmp(actionStr,"ND_func")==0)
    {
      action=ND_func;
    } else if(strcmp(actionStr,"D_func")==0)
    {
      action=D_func;
    } else if(strcmp(actionStr,"D2_func")==0)
    {
      action=D2_func;
    } else if(strcmp(actionStr,"D_deriv")==0)
    {
      action=D_deriv;
    } else if(strcmp(actionStr,"D2_deriv1")==0)
    {
      action=D2_deriv1;
    } else if(strcmp(actionStr,"D2_deriv2")==0)
    {
      action=D2_deriv2;
    } else if(strcmp(actionStr,"D_gradient")==0)
    {
      action=D_gradient;
    } else if(strcmp(actionStr,"D2_gradient")==0)
    {
      action=D2_gradient;
    } else if(strcmp(actionStr,"D2_hessian")==0)
    {
      action=D2_hessian;
```

```
    } else {
      mexErrMsgTxt("Wrong action string");
100   }
    } else if (!isSingleReal(ACTION_IN) || (action=floor(*mxGetPr(ACTION_IN)
        ))<0 || action>9 ){
    mexErrMsgTxt("Wrong action number");
    }

105 if( ((action==D_deriv || action==D2_deriv1) && nrhs!=9) || (action==
        D2_deriv2 && nrhs!=10))
    {
        mexErrMsgTxt("Wrong input arguments number");
    }

110 //  Arguments content check

  if(!isSingleReal(DIM_IN) || (GKLS_dim=floor(*mxGetPr(DIM_IN)))<2 ){
    mexErrMsgTxt("Wrong dimension: must be a scalar >= 2");
    }
115 if (!isSingleReal(NUM_MINIMA_IN) || (GKLS_num_minima=floor(*mxGetPr(
        NUM_MINIMA_IN)))<2 ){
    mexErrMsgTxt("Wrong minima number: must be a scalar >=2");
    }
    if (!isSingleReal(GLOBAL_VALUE_IN)){
    mexErrMsgTxt("Wrong global minimizer value: must be a scalar");
120 } else {
        GKLS_global_value=*mxGetPr(GLOBAL_VALUE_IN);
      }
    if (!isSingleReal(GLOBAL_DIST_IN) || (GKLS_global_dist=*mxGetPr(
        GLOBAL_DIST_IN))<=0 ){
    mexErrMsgTxt("Wrong distance: must be a scalar > 0");
125 }
    if (!isSingleReal(GLOBAL_RADIUS_IN) || (GKLS_global_radius=*mxGetPr(
        GLOBAL_RADIUS_IN))<=0 ){
    mexErrMsgTxt("Wrong radius: must be a scalar > 0");
    }
    if (!isSingleReal(TEST_NUMBER_IN) || (test_number=floor(*mxGetPr(
        TEST_NUMBER_IN)))<1 || test_number>100){
130 mexErrMsgTxt("Wrong test number: must be an integer 0 < i <= 100");
    }
    if (nrhs>7 && GKLS_dim!=(n=mxGetN(TARGET_POINTS_IN))){
    mexErrMsgTxt("Wrong target points matrix dimension: must be n x dim");
    }
135 if (nrhs>7 && (m=mxGetM(TARGET_POINTS_IN))!=1 && action>6){
        mexErrMsgTxt("Wrong target points matrix dimension for specified
            action: must be a sigle point");
    }
    if ((action==D_deriv || action==D2_deriv1 || action==D2_deriv2) && ((
        deriv_var1=floor(*mxGetPr(DERIV_VAR1_IN)))>GKLS_dim || !isSingleReal
        (DERIV_VAR1_IN))){
        mexErrMsgTxt("Wrong first partial derivative index");
140 }
    if (action==D2_deriv2 && ((deriv_var2=floor(*mxGetPr(DERIV_VAR2_IN)))>
        GKLS_dim || !isSingleReal(DERIV_VAR2_IN))){
        mexErrMsgTxt("Wrong second partial derivative index");
    }

145
  //  Function domain allocation
    if (GKLS_domain_alloc() != GKLS_OK){
      mexErrMsgTxt("Error allocating domain");
    }
```

```
150
    if(GKLS_arg_generate(test_number) != GKLS_OK) {
      mexErrMsgTxt("Internal error generating the function: check input data
          to be consistent");
    }

155 double *x, *res_matr;
      if(action!=INFO)
          x=mxGetPr(TARGET_POINTS_IN);

    if(action==INFO){
160       RESULT=mxCreateDoubleMatrix(GKLS_num_minima, GKLS_dim+2, mxREAL);
      res_matr=mxGetPr(RESULT);
          for(i=0; i<GKLS_num_minima; i++)
          {
              res_matr[i]=GKLS_minima.f[i];
165           res_matr[i+GKLS_num_minima]=GKLS_minima.rho[i];
              for(j=0; j<GKLS_dim; j++)
              {
                  res_matr[i+GKLS_num_minima*(j+2)]=GKLS_minima.local_min[i][j
                      ];
              }
170       }

    } else if(action < 7) // The function returns a matrix m*GKLS_dim
    {
          double *temp;
175       temp=(double *)mxMalloc(GKLS_dim*sizeof(double));
      RESULT=mxCreateDoubleMatrix(m, 1, mxREAL);
      res_matr=mxGetPr(RESULT);

      for(i=0; i<m; i++)
180   {
        for(j=0; j<GKLS_dim; j++)
        {
          temp[j]=x[j*m+i];
        }
185
        switch (action)
        {
          case ND_func:
            res_matr[i]=GKLS_ND_func(temp);
190         break;
          case D_func:
            res_matr[i]=GKLS_D_func(temp);
            break;
          case D2_func:
195         res_matr[i]=GKLS_D2_func(temp);
            break;
          case D_deriv:
            res_matr[i]=GKLS_D_deriv(deriv_var1, temp);
            break;
200       case D2_deriv1:
            res_matr[i]=GKLS_D2_deriv1(deriv_var1, temp);
            break;
          case D2_deriv2:
            res_matr[i]=GKLS_D2_deriv2(deriv_var1, deriv_var2, temp);
205         break;
        }
      }

    } else if(action <9) // The function returns a vector
```

```
210   {
        RESULT=mxCreateDoubleMatrix(GKLS_dim, 1, mxREAL);
        res_matr=mxGetPr(RESULT);

        switch (action)
215     {
          case D_gradient:
            GKLS_D_gradient(x,res_matr);
            break;
          case D2_gradient:
220         GKLS_D2_gradient(x,res_matr);
            break;
        }
      } else        // The function returns a matrix dim*dim
      {
225     RESULT=mxCreateDoubleMatrix(GKLS_dim, GKLS_dim, mxREAL);
        res_matr=mxGetPr(RESULT);

        for(i=0; i<GKLS_dim; i++)
          for(j=0; j<GKLS_dim; j++)
230       {
            res_matr[i+GKLS_dim*j]=GKLS_D2_deriv2(i+1,j+1,x);
          }
      }

235   return;

    }
```

A.2 gkls2matlab2.c

```
0
  #define malloc  mxMalloc
  #define free  mxFree

  #include "gkls.c"
5 #include "rnd_gen.c"
  #include <math.h>
  #include <string.h>
  #include "mex.h"

10 /* Operation types*/

  #define INFO     0
  #define ND_func   1
  #define D_func    2
15 #define D2_func   3
  #define D_deriv   4
  #define D2_deriv1 5
  #define D2_deriv2 6
  #define D_gradient  7
20 #define D2_gradient 8
  #define D2_hessian  9


  /* Input Arguments */
25
  #define NUM_MINIMA_IN    prhs[0]
  #define GLOBAL_VALUE_IN    prhs[1]
  #define GLOBAL_DIST_IN     prhs[2]
```

```
#define GLOBAL_RADIUS_IN  prhs[3]
#define ACTION_IN      prhs[4]
#define TEST_NUMBER_IN     prhs[5]
#define TARGET_X1_IN     prhs[6]
#define TARGET_X2_IN     prhs[7]
#define DERIV_VAR1_IN    prhs[8]
#define DERIV_VAR2_IN    prhs[9]

/* Output Arguments */

#define RESULT  plhs[0]

int isSingleReal(const mxArray *mxptr)
{
  return (mxGetM(mxptr)*mxGetN(mxptr)) == 1 && mxIsDouble(mxptr) && !
      mxIsComplex(mxptr);
}

void mexFunction( int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]
     )
{
    int test_number, action, deriv_var1, deriv_var2, i, j, m, n;
    GKLS_dim=2;

//  Check for proper number of arguments

    if (nrhs > 10) {
    mexErrMsgTxt("Too many input arguments.");
    }
    if (nrhs < 6) {
        mexErrMsgTxt("Too few input arguments.");
    }
    if (nlhs > 1) {
    mexErrMsgTxt("Too many output arguments.");
    }

//  Action check
    if(mxIsChar(ACTION_IN)){
        int strlen=mxGetN(ACTION_IN) + 1;
    char *actionStr=(char*)mxMalloc(strlen*sizeof(char));
    mxGetString(ACTION_IN, actionStr, strlen);

    if(strcmp(actionStr,"info")==0)
    {
      action=INFO;
    } else if(strcmp(actionStr,"ND_func")==0)
    {
      action=ND_func;
    } else if(strcmp(actionStr,"D_func")==0)
    {
      action=D_func;
    } else if(strcmp(actionStr,"D2_func")==0)
    {
      action=D2_func;
    } else if(strcmp(actionStr,"D_deriv")==0)
    {
      action=D_deriv;
    } else if(strcmp(actionStr,"D2_deriv1")==0)
    {
      action=D2_deriv1;
    } else if(strcmp(actionStr,"D2_deriv2")==0)
    {
```

```
      action=D2_deriv2;
90    } else if(strcmp(actionStr,"D_gradient")==0)
      {
        action=D_gradient;
      } else if(strcmp(actionStr,"D2_gradient")==0)
      {
95      action=D2_gradient;
      } else if(strcmp(actionStr,"D2_hessian")==0)
      {
        action=D2_hessian;
      } else {
100     mexErrMsgTxt("Wrong action string");
      }
      } else if (!isSingleReal(ACTION_IN) || (action=floor(*mxGetPr(ACTION_IN)
          ))<0 || action>9 ){
      mexErrMsgTxt("Wrong action number");
      }
105
      if( ((action==D_deriv || action==D2_deriv1) && nrhs!=9) || (action==
          D2_deriv2 && nrhs!=10))
      {
          mexErrMsgTxt("Wrong input arguments number");
      }
110
  //  Arguments content check

      if (!isSingleReal(NUM_MINIMA_IN) || (GKLS_num_minima=floor(*mxGetPr(
          NUM_MINIMA_IN)))<2 ){
      mexErrMsgTxt("Wrong minima number: must be a scalar >=2");
115   }
      if (!isSingleReal(GLOBAL_VALUE_IN)){
      mexErrMsgTxt("Wrong global minimizer value: must be a scalar");
      } else {
          GKLS_global_value=*mxGetPr(GLOBAL_VALUE_IN);
120     }
      if (!isSingleReal(GLOBAL_DIST_IN) || (GKLS_global_dist=*mxGetPr(
          GLOBAL_DIST_IN))<=0 ){
      mexErrMsgTxt("Wrong distance: must be a scalar > 0");
      }
      if (!isSingleReal(GLOBAL_RADIUS_IN) || (GKLS_global_radius=*mxGetPr(
          GLOBAL_RADIUS_IN))<=0 ){
125   mexErrMsgTxt("Wrong radius: must be a scalar > 0");
      }
      if (!isSingleReal(TEST_NUMBER_IN) || (test_number=floor(*mxGetPr(
          TEST_NUMBER_IN)))<1 || test_number>100){
      mexErrMsgTxt("Wrong test number: must be an integer 0 < i <= 100");
      }
130   if (nrhs > 7 && ( (n=mxGetN(TARGET_X1_IN)) != mxGetN(TARGET_X2_IN) || (m
          =mxGetM(TARGET_X1_IN)) != mxGetM(TARGET_X2_IN))){
        mexErrMsgTxt("Wrong target points matrix dimension: must have the same
            dimension");
      }
      if (action>6 && (n*m)!=1){
          mexErrMsgTxt("Wrong target points matrix dimension for specified
              action: must be a sigle point");
135   }
      if ((action==D_deriv || action==D2_deriv1 || action==D2_deriv2) && ((
          deriv_var1=floor(*mxGetPr(DERIV_VAR1_IN)))>GKLS_dim || !isSingleReal
          (DERIV_VAR1_IN))){
          mexErrMsgTxt("Wrong first partial derivative index");
      }
      if (action==D2_deriv2 && ((deriv_var2=floor(*mxGetPr(DERIV_VAR2_IN)))>
```

```
        GKLS_dim || !isSingleReal(DERIV_VAR2_IN))){
140         mexErrMsgTxt("Wrong second partial derivative index");
    }


//  Function domain allocation
145   if (GKLS_domain_alloc() != GKLS_OK){
      mexErrMsgTxt("Error allocating domain");
    }

    if(GKLS_arg_generate(test_number) != GKLS_OK) {
150   mexErrMsgTxt("Internal error generating the function: check input data
          to be consistent");
    }

    double *x1,*x2, *res_matr, *temp;

155   if(action!=INFO)
      {
          temp=(double *)mxMalloc(GKLS_dim*sizeof(double));
          x1=mxGetPr(TARGET_X1_IN);
          x2=mxGetPr(TARGET_X2_IN);
160   }

    if(action==INFO){
          RESULT=mxCreateDoubleMatrix(GKLS_num_minima, GKLS_dim+2, mxREAL);
          res_matr=mxGetPr(RESULT);
165       for(i=0; i<GKLS_num_minima; i++)
          {
              res_matr[i]=GKLS_minima.f[i];
              res_matr[i+GKLS_num_minima]=GKLS_minima.rho[i];
              for(j=0; j<GKLS_dim; j++)
170           {
                  res_matr[i+GKLS_num_minima*(j+2)]=GKLS_minima.local_min[i][j
                      ];
              }
          }
    } else if(action < 7) // The function returns a matrix m*n
175 {
      RESULT=mxCreateDoubleMatrix(m, n, mxREAL);
      res_matr=mxGetPr(RESULT);

      for(i=0; i<(m*n); i++)
180   {
        temp[0]=x1[i];
        temp[1]=x2[i];
        switch (action)
        {
185       case ND_func:
            res_matr[i]=GKLS_ND_func(temp);
            break;
          case D_func:
            res_matr[i]=GKLS_D_func(temp);
190         break;
          case D2_func:
            res_matr[i]=GKLS_D2_func(temp);
            break;
          case D_deriv:
195         res_matr[i]=GKLS_D_deriv(deriv_var1, temp);
            break;
          case D2_deriv1:
            res_matr[i]=GKLS_D2_deriv1(deriv_var1, temp);
```

```
            break;
          case D2_deriv2:
            res_matr[i]=GKLS_D2_deriv2(deriv_var1, deriv_var2, temp);
            break;
        }
      }

    } else if(action <9) // The function returns a vector
    {
      temp[0]=x1[0];
      temp[1]=x2[0];
      RESULT=mxCreateDoubleMatrix(GKLS_dim, 1, mxREAL);
      res_matr=mxGetPr(RESULT);

      switch (action)
      {
        case D_gradient:
          GKLS_D_gradient(temp,res_matr);
          break;
        case D2_gradient:
          GKLS_D2_gradient(temp,res_matr);
          break;
      }
    } else         // The function returns a matrix dim*dim
    {
      temp[0]=x1[0];
      temp[1]=x2[0];
      RESULT=mxCreateDoubleMatrix(GKLS_dim, GKLS_dim, mxREAL);
      res_matr=mxGetPr(RESULT);

      for(i=0; i<GKLS_dim; i++)
        for(j=0; j<GKLS_dim; j++)
        {
          res_matr[i+GKLS_dim*j]=GKLS_D2_deriv2(i+1,j+1,temp);
        }
    }

      return;

}
```

Appendice B

Codici sorgente: GklsGUI

B.1 AlgExaminer.java

```
0 import java.util.*;

  public class AlgExaminer{
      private double tol=0;
    private int evaluations=0;
5   private int gradient=0;
    private int hessian=0;
      private int maxiter=0;
    private ArrayList points = new ArrayList();

10    public double getTol(){
          return tol;
      }

      public int getMaxiter(){
15        return maxiter;
      }

      public void setTol(double aTol){
          tol=aTol;
20    }

      public void setMaxiter(int aMaxiter){
          maxiter=aMaxiter;
      }
25
    public int getEval(){
      return evaluations;
    }

30  public int getGrad(){
      return gradient;
    }

    public int getHess(){
35    return hessian;
    }
```

```
  public void addEval(){
    evaluations++;
40  }


  public void addGrad(){
    gradient++;
45  }

  public void addHess(){
    hessian++;
  }
50
  public void reset(){
        evaluations=0;
      gradient=0;
        hessian=0;
55      points = new ArrayList();
    }

    public int addPoint(double x, double y, double z){
    points.add(new Point(x,y,z));
60      return points.size();
  }

    public double[] getPoint(int index)
    {
65      return ((Point) points.get(index)).getArray();
    }

    public int length()
    {
70      return points.size();
    }

  public String toString(){
      Iterator i = points.iterator();
75  String result;
    result = "Valutazioni: "+evaluations+
             "\nGradienti: "+gradient+
                 "\nHessiane: "+hessian+
             "\n\nPunti:";
80  while(i.hasNext())
        {
      result=result+"\t"+i.next()+"\n";
            }
        return result;
85  }

  private class Point{
    private double[] value;

90  public Point(double x,double y,double z){
      value = new double[3];
      value[0]=x;
      value[1]=y;
            value[2]=z;
95  }

        public double[] getArray()
        {
            double[] result={getX(),getY(),getZ()};
```

```
100             return result;
          }

        public String toString()
        {
    return getX()+" "+getY()+" "+getZ();
        }

      public double getX(){
        return value[0];
      }

      public double getY(){
        return value[1];
      }

          public double getZ(){
        return value[2];
      }
    }
}
```

B.2 L'interfaccia grafica

```
function varargout = GklsGUI(varargin)
% GKLSGUI M-file for GklsGUI.fig

% Last Modified by GUIDE v2.5 25-Mar-2006 23:52:08

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @GklsGUI_OpeningFcn, ...
                   'gui_OutputFcn',  @GklsGUI_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT


% --- Executes just before GklsGUI is made visible.
function GklsGUI_OpeningFcn(hObject, eventdata, handles, varargin)

% Choose default command line output for GklsGUI
handles.output = hObject;

funcs=dir(fullfile('multidimAlg','*.m'));
set(handles.methodsPopup,'String',char({funcs.name}));
clearOutput(handles);

% Update handles structure
```

```
guidata(hObject, handles);


% --- Outputs from this function are returned to the command line.
function varargout = GklsGUI_OutputFcn(hObject, eventdata, handles)
% varargout  cell array for returning output args (see VARARGOUT);

% Get default command line output from handles structure
varargout{1} = handles.output;


% --- Executes on slider movement.
function slider1_Callback(hObject, eventdata, handles)

set(handles.slider1,'TooltipString',strcat('Value:  ',num2str(get(handles.
    slider1,'Value'),2)));
slider2_Callback(hObject, eventdata, handles);


% --- Executes during object creation, after setting all properties.
function slider1_CreateFcn(hObject, eventdata, handles)

if isequal(get(hObject,'BackgroundColor'), get(0,'
    defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor',[.9 .9 .9]);
end


function editClass_Callback(hObject, eventdata, handles)


% --- Executes during object creation, after setting all properties.
function editClass_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
    defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on slider movement.
function slider2_Callback(hObject, eventdata, handles)

distance=get(handles.slider1,'Value');
set(handles.slider2,'TooltipString',strcat('Value: ',num2str(distance*get(
    handles.slider2,'Value')/2,2)));


% --- Executes during object creation, after setting all properties.
function slider2_CreateFcn(hObject, eventdata, handles)

if isequal(get(hObject,'BackgroundColor'), get(0,'
    defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor',[.9 .9 .9]);
end


% --- Executes on button press in generateButton.
function generateButton_Callback(hObject, eventdata, handles)

tol=eval(get(handles.editTol,'String'));
maxiter=eval(get(handles.editMaxiter,'String'));
```

```
nmin=eval(get(handles.editMinima,'String'));
class=eval(get(handles.editClass,'String'));
distance=get(handles.slider1,'Value');
radius=distance*get(handles.slider2,'Value')/2;
handles.problem=gklsProblem(tol,maxiter,nmin,-1,distance,radius,class);
handles.minInfo=getMinInfo(handles.problem);

enableAll(handles);
handles.solutions={};
handles.drawType=1;
handles.drawAspect=1;
handles.view=view(3);

guidata(hObject, handles);
draw(hObject, handles);
clearOutput(handles);


function editMinima_Callback(hObject, eventdata, handles)

% --- Executes during object creation, after setting all properties.
function editMinima_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
    defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on selection change in methodsPopup.
function methodsPopup_Callback(hObject, eventdata, handles)


% --- Executes during object creation, after setting all properties.
function methodsPopup_CreateFcn(hObject, eventdata, handles)

%if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
    defaultUicontrolBackgroundColor'))
%    set(hObject,'BackgroundColor','white');
%end



function editStartX_Callback(hObject, eventdata, handles)


% --- Executes during object creation, after setting all properties.
function editStartX_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
    defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


function editStartY_Callback(hObject, eventdata, handles)


% --- Executes during object creation, after setting all properties.
```

```
function editStartY_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
    defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on button press in initialPointButton.
function initialPointButton_Callback(hObject, eventdata, handles)


if 1==get(hObject,'Value')
    disableAll(handles);
    set(handles.initialPointButton,'Enable','on');
    datacursormode on;
else
    datacursormode off;
    dc=datacursormode(gcf);
    is=getCursorInfo(dc);
    set(handles.editStartX,'String',round(is.Position(1)*10^3)/10^3);
    set(handles.editStartY,'String',round(is.Position(2)*10^3)/10^3);
    enableAll(handles);
end

% --- Executes on button press in startButton.
function startButton_Callback(hObject, eventdata, handles)
x=eval(get(handles.editStartX,'String'));
y=eval(get(handles.editStartY,'String'));
reset(handles.problem);
setTol(handles.problem,eval(get(handles.editTol,'String')));
setMaxiter(handles.problem,eval(get(handles.editMaxiter,'String')));
getMaxiter(handles.problem);
nextPoint(handles.problem,[x y]);

% Creo la stringa della funzione e la eseguo
funcN=get(handles.methodsPopup,'Value');
func=cellstr(get(handles.methodsPopup,'String'));
func=func{funcN};
func=func([1:length(func)-2]);
eval(strcat(func,'(handles.problem);'));

[x1,x2,x3]=getPath(handles.problem);
n=length(handles.solutions);
handles.solutions{n+1}={x1,x2,x3};

% Cerco il punto di minimo piu' vicino alla soluzione trovata
solSteps=length(x1);
minFound=[x1(solSteps) x2(solSteps)];
minPoints=handles.minInfo(:,[2 3]);
[m,n]=size(minPoints);
temp=[ones(m,1)*minFound(1) ones(m,1)*minFound(2)];
distance=temp-minPoints;
distanceNormQuad=distance(:,1).^2+distance(:,2).^2;
[a,index]=min(distanceNormQuad);

% TODO cercare un modo di visualizzare sta roba!!!!!
distanza=sqrt(distanceNormQuad(index));
differenza=abs(x3(solSteps)-handles.minInfo(index,1));
[evals,grads,hesss]=getOpsDone(handles.problem);
println(handles,['Iter.   = ' num2str(solSteps)]);
println(handles,['Evals   = F:' num2str(evals) ...
```

```
                            ' G:' num2str(grads) ...
                            ' H:' num2str(hesss) ]);
println(handles,['x_n-x* = ' num2str(distanza)]);
println(handles,['f_n-f* = ' num2str(differenza)]);
println(handles,' ');


guidata(hObject, handles);
draw(hObject, handles);

%--- Executes on button press in rotateButton.
function rotateButton_Callback(hObject, eventdata, handles)

axes(handles.plotAxes);
rotate3d

function draw(hObject, handles)
axes(handles.plotAxes);
[a,b]=meshgrid(linspace(-1,1,70));
[p,q]=view;
if p~=0 || q~=90
    handles.view=[p,q];
end

switch handles.drawAspect
    case 1
        meshc(a,b,evalMesh(handles.problem,a,b,handles.drawType));
        colormap(cool);
        view(handles.view);
        grid on;
    case 2
        contour(a,b,evalMesh(handles.problem,a,b,handles.drawType),50);
        colormap(jet);
        grid on;
    case 3
        contour3(a,b,evalMesh(handles.problem,a,b,handles.drawType),50);
        colormap(jet);
        view(handles.view);
        grid off;
    case 4
        [a,b]=meshgrid(linspace(-1,1,200));
        surfl(a,b,evalMesh(handles.problem,a,b,handles.drawType));
        shading interp;
        colormap(copper);
        view(handles.view);
        grid on;
end

if handles.drawType == 1
    hold on
    colors={'b','r','k','m','y','c','g'};
    for i=1:length(handles.solutions)
        plot3(handles.solutions{i}{1},handles.solutions{i}{2},handles.
            solutions{i}{3},'Color',colors{mod(i-1,7)+1},'Marker','.');
    end
    hold off
end
guidata(hObject, handles);

function generate(hObject, handles)
tol=eval(get(handles.editTol,'String'));
maxiter=eval(get(handles.editMaxiter,'String'));
```

```
    nmin=eval(get(handles.editMinima,'String'));
    class=eval(get(handles.editClass,'String'));
    distance=get(handles.slider1,'Value');
    radius=distance*get(handles.slider2,'Value')/2;
    handles.problem=gklsProblem(tol,maxiter,nmin,-1,distance,radius,class);
    handles.minInfo=getMinInfo(handles.problem);
    guidata(hObject, handles);


    % --- Executes on button press in meshButton.
    function meshButton_Callback(hObject, eventdata, handles)

    handles.drawAspect=1;
    guidata(hObject, handles);
    draw(hObject, handles);


    % --- Executes on button press in contourButton.
    function contourButton_Callback(hObject, eventdata, handles)

    handles.drawAspect=2;
    guidata(hObject, handles);
    draw(hObject, handles);


    % --- Executes on button press in contour3DButton.
    function contour3DButton_Callback(hObject, eventdata, handles)

    handles.drawAspect=3;
    guidata(hObject, handles);
    draw(hObject, handles);

    % --- Executes on button press in surfButton.
    function surfButton_Callback(hObject, eventdata, handles)

    handles.drawAspect=4;
    guidata(hObject, handles);
    draw(hObject, handles);


    % --- Executes on button press in drawFunButt.
    function drawFunButt_Callback(hObject, eventdata, handles)

    handles.drawType=1;
    guidata(hObject, handles);
    draw(hObject, handles);

    % --- Executes on button press in drawDxButt.
    function drawDxButt_Callback(hObject, eventdata, handles)

    handles.drawType=21;
    guidata(hObject, handles);
    draw(hObject, handles);


    % --- Executes on button press in drawDyButt.
    function drawDyButt_Callback(hObject, eventdata, handles)

    handles.drawType=22;
    guidata(hObject, handles);
    draw(hObject, handles);
```

```
% --- Executes on button press in drawDxxButt.
function drawDxxButt_Callback(hObject, eventdata, handles)

handles.drawType=31;
guidata(hObject, handles);
draw(hObject, handles);


% --- Executes on button press in drawDxyButt.
function drawDxyButt_Callback(hObject, eventdata, handles)

handles.drawType=32;
guidata(hObject, handles);
draw(hObject, handles);


% --- Executes on button press in drawDyyButt.
function drawDyyButt_Callback(hObject, eventdata, handles)

handles.drawType=33;
guidata(hObject, handles);
draw(hObject, handles);


function outputEdit_Callback(hObject, eventdata, handles)


% --- Executes during object creation, after setting all properties.
function outputEdit_CreateFcn(hObject, eventdata, handles)

%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
    defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


function println(handles, string)
old=get(handles.outputEdit,'String');
set(handles.outputEdit,'String',strvcat(old,string));
[m,n]=size(old);
set(handles.outputEdit,'ListboxTop',m);

function clearOutput(handles)
set(handles.outputEdit,'String',strvcat('          -= GklsGUI v 0.1 =-',' ')
    );

function enableAll(handles)
set(handles.rotateButton,'Enable','on');
set(handles.initialPointButton,'Enable','on');
set(handles.generateButton,'Enable','on');
set(handles.drawFunButt,'Enable','on');
set(handles.drawDxButt,'Enable','on');
set(handles.drawDyButt,'Enable','on');
set(handles.drawDxxButt,'Enable','on');
set(handles.drawDxyButt,'Enable','on');
set(handles.drawDyyButt,'Enable','on');
set(handles.initialPointButton,'Enable','on');
set(handles.rotateButton,'Enable','on');
```

```
    set(handles.startButton,'Enable','on');
    set(handles.meshButton,'Enable','on');
    set(handles.surfButton,'Enable','on');
    set(handles.contourButton,'Enable','on');
400 set(handles.contour3DButton,'Enable','on');

    function disableAll(handles)
    set(handles.rotateButton,'Enable','off');
    set(handles.initialPointButton,'Enable','off');
405 set(handles.generateButton,'Enable','off');
    set(handles.drawFunButt,'Enable','off');
    set(handles.drawDxButt,'Enable','off');
    set(handles.drawDyButt,'Enable','off');
    set(handles.drawDxxButt,'Enable','off');
410 set(handles.drawDxyButt,'Enable','off');
    set(handles.drawDyyButt,'Enable','off');
    set(handles.initialPointButton,'Enable','off');
    set(handles.rotateButton,'Enable','off');
    set(handles.startButton,'Enable','off');
415 set(handles.meshButton,'Enable','off');
    set(handles.surfButton,'Enable','off');
    set(handles.contourButton,'Enable','off');
    set(handles.contour3DButton,'Enable','off');

420


    function editTol_Callback(hObject, eventdata, handles)
    % hObject    handle to editTol (see GCBO)
    % eventdata  reserved - to be defined in a future version of MATLAB
425 % handles    structure with handles and user data (see GUIDATA)

    % Hints: get(hObject,'String') returns contents of editTol as text
    %        str2double(get(hObject,'String')) returns contents of editTol as a
        double


430
    %--- Executes during object creation, after setting all properties.
    function editTol_CreateFcn(hObject, eventdata, handles)
    % hObject    handle to editTol (see GCBO)
    % eventdata  reserved - to be defined in a future version of MATLAB
435 % handles    empty - handles not created until after all CreateFcns called

    % Hint: edit controls usually have a white background on Windows.
    %       See ISPC and COMPUTER.
    if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
        defaultUicontrolBackgroundColor'))
440     set(hObject,'BackgroundColor','white');
    end


    %--- Executes on button press in genBananaButton.
445 function genBananaButton_Callback(hObject, eventdata, handles)
    % hObject    handle to genBananaButton (see GCBO)
    % eventdata  reserved - to be defined in a future version of MATLAB
    % handles    structure with handles and user data (see GUIDATA)

450 tol=eval(get(handles.editTol,'String'));
    maxiter=eval(get(handles.editMaxiter,'String'));
    handles.problem=bananaProblem(tol,maxiter);

    handles.minInfo=getMinInfo(handles.problem);
455
```

```
enableAll(handles);
handles.solutions={};
handles.drawType=1;
handles.drawAspect=1;
handles.view=view(3);

guidata(hObject, handles);
draw(hObject, handles);
clearOutput(handles);


function editMaxiter_Callback(hObject, eventdata, handles)
% hObject    handle to editMaxiter (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of editMaxiter as text
%        str2double(get(hObject,'String')) returns contents of editMaxiter
    as a double


% --- Executes during object creation, after setting all properties.
function editMaxiter_CreateFcn(hObject, eventdata, handles)
% hObject    handle to editMaxiter (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
    defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```

B.3 La classe `gklsProblem`

B.4 Gli algoritmi di ottimo

B.4.1 armijo.m

```
function res = armijo(p,x,d,gx)
%function res = armijo(p,x,d,gx)
%
% Funzione per la minimizzazione line search con il metodo Armijo
%
% res e' un cell array della forma {a,b} dove a e' il punto
% trovato, e b=f(a)

epsilon=.5;
eta=2;
alpha=0.1;

phi0=getValue(p,x);
epsilondphi0=d'*gx*epsilon;
xalpha=x+alpha*d;
phiAlpha=getValue(p,xalpha);

while phiAlpha <= phi0+epsilondphi0*alpha;
```

```
        res={xalpha , phiAlpha};
        alpha=eta*alpha;
        xalpha=x+alpha*d;
        phiAlpha=getValue(p,xalpha);
    end

    while phiAlpha > phi0+epsilondphi0*alpha;
        alpha=alpha/eta;
        xalpha=x+alpha*d;
        phiAlpha=getValue(p,xalpha);
        res={xalpha , phiAlpha};
    end
```

B.4.2 SteepestDescent.m

```
function p = SteepestDescent(p)

x=getStart(p); %Prendo il punto iniziale

gx=getGrad(p,x);% calcolo il gradiente nel punto

tol=getTol(p);

maxiter=getMaxiter(p);
n=0;
fx1=0;
fx2=0;
fx3=0;

% faccio almeno 2 iterazioni per poter stimare il minimo
% con il metodo di delta^2-Aitken

while n<3 || (abs(fx3-(fx1*fx3-fx2*fx2)/(fx3+fx1-2*fx2)) > tol && n <
    maxiter)
    fx3=fx2;
    fx2=fx1;

    % minimizzo la funzione x0+a g(x)
    res=armijo(p,x,-gx,gx);
    x=res{1};
    fx1=res{2};
    nextPoint(p,x);
    n=n+1;
    gx=getGrad(p,x); %riaggiorno il gradiente
end
```

B.4.3 NewtonDamped.m

```
function p = NewtonDamped(p)

tol=getTol(p);
tol2=tol^2;
maxiter=getMaxiter(p);

x=getStart(p); %Prendo il punto iniziale
gx=getGrad(p,x);% calcolo il gradiente nel punto
hx=getHess(p,x);

[d,A]=ldlt_fact_sys(hx,-gx);
n=0;
```

```
while n<maxiter && gx'*gx>tol2
    if(all(d==0))
        warning('La matrice non e' SDP!!! mi fermo');
        return;
    end
    res=armijo(p,x,d,gx);
    x=res{1};
    nextPoint(p,x);
    n=n+1;
    gx=getGrad(p,x); %riaggiorno il gradiente
    hx=getHess(p,x);
    d=ldlt_fact_sys(hx,-gx);
end
```

B.4.4 LevMarq.m

```
function p = LevMarq(p)

tol=getTol(p);
tol2=tol^2;
maxiter=getMaxiter(p);

x=getStart(p); %Prendo il punto iniziale
gx=getGrad(p,x);% calcolo il gradiente nel punto
hx=getHess(p,x);% calcolo l'hessiana nel punto

%eseguo una prima volta la fattorizzazione LDL^T
d=-ldlt_fact_sys(hx,gx);

n=0;
e=0;

while n < maxiter && gx'*gx > tol2
    adder=norm(hx,inf)/length(hx);
    e=max(0,e-adder);
    %controllo che la fattorizzazione sia andata a buon fine
    %altrimenti aggiorno la hx e fattorizzo nuovamente
    while(all(d==0))
        e=e+adder;
        d=-ldlt_fact_sys(e*eye(2)+hx,gx);
    end
    res=armijo(p,x,d,gx);

    x=res{1};
    fx1=res{2};
    nextPoint(p,x);
    n=n+1;
    gx=getGrad(p,x); %riaggiorno il gradiente
    hx=getHess(p,x);
    d=-ldlt_fact_sys(hx,gx);
end
```

B.4.5 DFP.m

```
function prob = DFP (prob)
% Davidon-Fletcher-Powell

tol=getTol(prob);
tol2=tol^2;
```

```
maxiter=getMaxiter(prob);

x=getStart(prob); %Prendo il punto iniziale

gx=getGrad(prob,x);% calcolo il gradiente nel punto

dim=length(x);
d=-gx;

n=0;
while n<maxiter && gx'*gx>tol2
    res=armijo(prob,x,d,gx);
    p=res{1}-x;
    x=res{1};
    nextPoint(prob,x);
    oldg=gx;
    gx=getGrad(prob,x);
    q=gx-oldg;
    if mod(n,3)==0
        H=eye(dim);
    else
        Hq=H*q;
        H=H+p*p'/(p'*q)-Hq*Hq'/(q'*Hq);
    end
    d=-H*gx;
    n=n+1;
end
```

B.4.6 DFPScaling.m

```
function prob = DFPScaling (prob)
% Davidon-Fletcher-Powell con autoscaling

tol=getTol(prob);
tol2=tol^2;
maxiter=getMaxiter(prob);

x=getStart(prob); %Prendo il punto iniziale

gx=getGrad(prob,x);% calcolo il gradiente nel punto

dim=length(x);
d=-gx;

n=0;

while n<maxiter && gx'*gx>tol2
    res=armijo(prob,x,d,gx);
    p=res{1}-x;
    x=res{1};
    nextPoint(prob,x);
    oldg=gx;
    gx=getGrad(prob,x);
    q=gx-oldg;
    if mod(n,3)==0
        H=eye(dim);
    else
        Hq=H*q;
        gamma=p'*q/(q'*Hq);
        H=gamma*(H-Hq*Hq'/(q'*Hq))+p*p'/(p'*q);
    end
```

```
    d=-H*gx;
    n=n+1;
end
```

B.4.7 TrustRegionCauchy.m

```
function p = TrustRegionCauchy(p)
%trust region Newton e Cauchy point

tol=getTol(p);
tol2=tol^2;
maxiter=getMaxiter(p);

maxDelta=.5;
delta=.2;
eta=1/5;

x=getStart(p); %Prendo il punto iniziale
fx=getValue(p,x);% valuto la funzione nel punto
gx=getGrad(p,x);% calcolo il gradiente nel punto
hx=getHess(p,x);% calcolo l' hessiana nel punto

n=0;

while n<maxiter && gx'*gx>tol2
    %--- trovo il punto di cauchy (cp)
    d2=gx'*hx*gx;
    gradnorm=norm(gx);
    if d2 > 0
        tau=min([1/gradnorm gradnorm^2/delta/d2]);
    else
        tau=1/gradnorm;
    end

    %--- ottimizzata con 1/gradnorm
    cp=-tau*delta*gx;
    %cp=-tau*delta*gk/gradnorm;

    fxkpk=getValue(p,x+cp);

    %--- aggiorno la trust region
    mkpk=fx+gx'*cp+cp'*hx*cp/2;
    rhok=(fx-fxkpk)/(fx-mkpk);

    if rhok < 1/4
        delta=norm(cp)/4;
    elseif rhok > 3/4 && norm(cp)==delta
        delta=min(2*delta,maxDelta);
    end

    %--- se
    if rhok > eta
        x=x+cp;
        fx=fxkpk;
        nextPoint(p,x);
        n=n+1;
        gx=getGrad(p,x); %riaggiorno il gradiente
        hx=getHess(p,x); %riaggiorno l'hessiana
    end
end
```

B.4.8 TrustRegionDogleg.m

```
function p = TrustRegDogleg(p)
%trust region con matrice di Levemberg-Marquardt e
%metodo dogleg

tol=getTol(p);
tol2=tol^2;
maxiter=getMaxiter(p);
debug= false;

maxDelta=.5;
delta=.2;
eta=1/5;

x=getStart(p);  % Prendo il punto iniziale
gx=getGrad(p,x);% Calcolo il gradiente nel punto
hx=getHess(p,x);% Calcolo l'hessiana nel punto

pb=-ldlt_fact_sys(hx,gx);

n=0;
e=0;

while n<maxiter && gx'*gx>tol2
    adder=norm(hx,inf)/length(hx);
    e=max(0,e-adder);

    Bk=hx;

    fx=getValue(p,x);
    while(all(pb==0))
        e=e+adder;
        Bk=e*eye(2)+hx;
        pb=-ldlt_fact_sys(Bk,gx);
    end

    %--- metodo Dogleg

    pu=-(gx'*gx/(gx'*Bk*gx))*gx;

    %--- determino il punto di minimo sulla
    %--- spezzata
    if norm(pb)<= delta
        pk=pb;
        if debug
            graphix(n+1,3)=1;
        end
    elseif norm(pu) >= delta
        if debug
            graphix(n+1,3)=2;
        end
        pk=-gx*delta/norm(gx);
    else
        if debug
            graphix(n+1,3)=3;
        end
        a=(pb-pu)'*(pb-pu);
        b=2*(pb-pu)'*pu;
        c=pu'*pu-delta^2;
        d=sqrt(b^2-4*a*c);
```

```
            tau=(-b+d)/(2*a);
            if ~(tau >=0 && tau <= 1)
                tau=(-b-d)/(2*a);
            end
            pk=pu+tau*pb;
        end
        fxkpk=getValue(p,x+pk);

        %--- aggiorno la trust region
        mkpk=fx+gx'*pk+pk'*Bk*pk/2;
        rhok=(fx-fxkpk)/(fx-mkpk);
        if debug
            graphix(n+1,1) = delta;
            graphix(n+1,2)= rhok;
        end
        if rhok < 1/4
            delta=norm(pk)/4;
        elseif rhok > 3/4 && abs(norm(pk)-delta)<eps*10
            boh=norm(pk)-delta;
            delta=min(2*delta,maxDelta);
        end

        %--- scelgo se accettare o meno il nuovo passo
        if rhok > eta
            x=x+pk;
            fx=fxkpk;
            nextPoint(p,x);
            gx=getGrad(p,x); %riaggiorno il gradiente
            hx=getHess(p,x);
            pb=-ldlt_fact_sys(hx,gx);
        else
            %--- se non cambio punto, mi ricordo la matrice
            %--- calcolata
            hx=Bk;
            pb=0;
        end
        n=n+1;
    end

    if debug
        figure
        plot(graphix)
        legend('delta','rho','tipo')
        graphix
    end
```

B.4.9 FMinGrad.m

```
function p = FMinGrad(p)

x0=getStart(p);
maxiter=getMaxiter(p);
n=0;

option=optimset('OutputFcn',@outfun,'GradObj','on', ...
                'TolFun',getTol(p),'LargeScale','off');

fminunc(@objfun,x0,option);

function [f,g] = objfun(x)
    f = getValue(p,x);
```

```
    if nargout > 1
        g = getGrad(p,x);
    end
end

function stop = outfun(x, optimValues, state)
    n=n+1;
    nextPoint(p,x);
    if n < maxiter
        stop = false;
    else
        stop = true;
    end

end

end
```

Bibliografia

[BMS05] Luigi Brugnano, Cecilia Magherini, and Alessandra Sestini, *Calcolo numerico*, Master Università e Professioni, Firenze, 2005.

[GKLS03] Marco Gaviano, Dimitri E. Kvasov, Daniela Lera, and Yaroslav D. Sergeyev, *Software for generation of classes of test functions with known local and global minima for global optimization*, ACM Transactions on Mathematical Software **24** (2003), no. 4, 469–480.

[GL98] Marco Gaviano and Daniela Lera, *Test functions with variable attraction regions for global optimization problems*, Journal of Global Optimization **13** (1998), no. 2, 207–223.

[Knu97] Donald Knuth, *The art of computer programming, vol. 2: Seminumerical algorithms*, vol. 2, Addison-Wesley, Stanford, 1997.

[Lue89] David G. Luenberger, *Linear and nonlinear programming*, Addison-Wesley, Stanford, 1989.

[NW99] Jorge Nocedal and Stephen J. Wright, *Numerical optimization*, Springer, New York, 1999.

[Pj02] Meilir Page-jones, *Progettazione a oggetti con uml*, APOGEO, Milano, 2002.