Vuoi ottimizzare? Fallo con Scilab!

5 agosto

2011

In questo tutorial vi facciamo vedere come Scilab possa essere considerato a tutti gli effetti un potente software di ottimizzazione multiobiettivo e multidisciplinare.

EnginSoft SpA - Scilab Professional Partner
e-mail: info@openeering.com web: www.openeering.com
Sede legale: via della Stazione 27 - loc. Mattarello, 38123 Trento (TN)
P.Iva e C.F. IT00599320223

Vuoi ottimizzare? Fallo con Scilab!

In questo tutorial vi facciamo vedere come Scilab[1] possa essere considerato a tutti gli effetti un potente software di ottimizzazione multiobiettivo e multidisciplinare. Sappiamo oramai che Scilab è un linguaggio di alto livello con una sintassi molto simile a quella di MATLAB ®[2].

Esattamente come per MATLAB®, l'ottimizzazione è un tema importante per Scilab. Scilab ha la capacità di risolvere problemi di ottimizzazione, lineare e non lineare, mono e multiobiettivo, attraverso una vasta collezione di algoritmi disponibili.

Qui verrà fatta una presentazione molto generale degli algoritmi di ottimizzazione disponibili in Scilab, il lettore può trovare di seguito qualche riga codice che potrà essere copiato direttamente nella console Scilab.

Ottimizzazione lineare e non lineare

Come i nostri lettori di sicuro già sanno, "ottimizzare" significa selezionare la migliore opzione disponibile da una vasta gamma di scelte possibili. Questa attività può risultare un compito complesso considerando che, potenzialmente, potrebbe essere necessario un numero enorme di tentativi se si utilizza un approccio a forza bruta.

La formulazione matematica di un problema di ottimizzazione generale può essere espressa come segue:

$$\min_{x \in S^n} (f_1(x), \cdots, f_k(x))$$
$$\text{subject to} \begin{cases} g_i(x) \geq 0 \\ g_l(x) = 0 \\ x \in S^n \end{cases}$$

Con (X_1, ..., X_n) parametri liberi di variare nel dominio S. Ogni volta che k>1, ossia ogni volta che ci sono più funzioni da minimizzare, si parla di ottimizzazione multiobiettivo.

Metodi grafici

Con Scilab è possibile risolvere problemi di ottimizzazione anche solo utilizzando i grafici. Per esempio, supponiamo di voler trovare il punto di minimo della funzione Rosenbrock. Il grafico “contour plot” può essere di aiuto per individuare l'area ottimale.

Se sulla shell Scilab copiate il seguente script ottenete il grafico in Figura1.

```
function f=rosenbrockC(x1, x2)
  x = [x1 x2];
  f = 100.0 *(x(2)-x(1)^2)^2 + (1-x(1))^2;
endfunction
xdata = linspace(-2,2,100);
ydata = linspace(-2,2,100);
```

[3]

[1] Scarica Scilab gratuitamente dal sito http://www.scilab.org
[2] MATLAB è un marchio registrato di The MathWorks Inc.

contour(xdata , ydata , rosenbrockC , [1 10 100 1000])

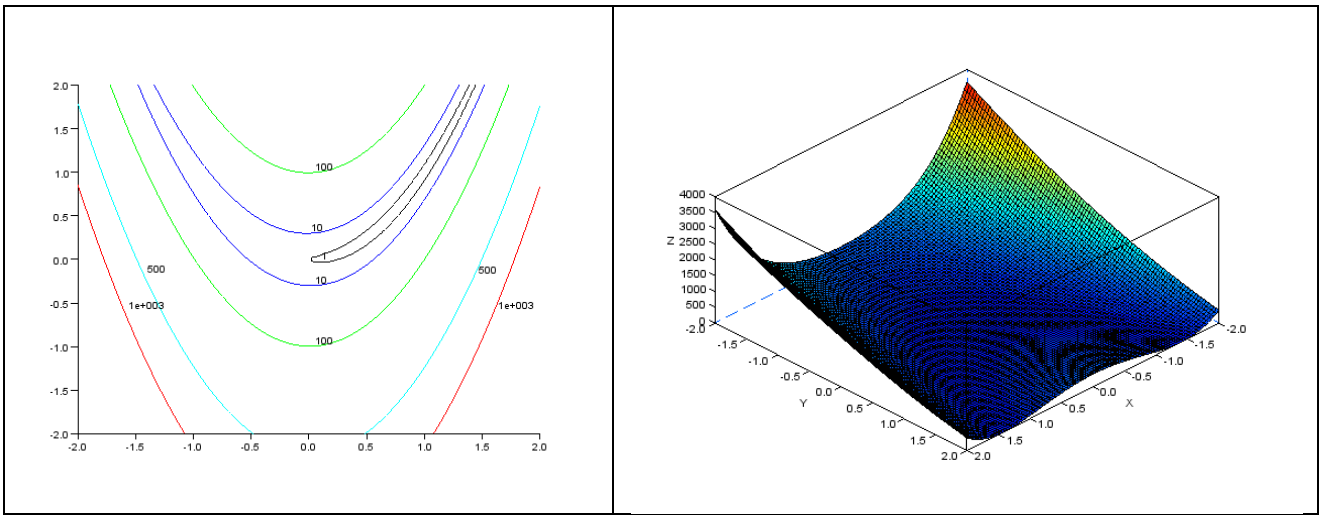

Figura 1: curve di livello (a sinistra) e grafico 3D (a destra) della funzione Rosenbrock. Possiamo in questo modo identificare il minimo che si trova nella regione del contorno nero. Questo ci suggerisce che un buon punto di partenza per ulteriori ricerche potrebbe essere x = (0.5,0.5)

Questo grafico può essere certamente di aiuto per identificare una soluzione ottimale. Purtroppo però, risolvere un problema di ottimizzazione con metodi grafici è possibile solo quando abbiamo un numero limitato di variabili di ingresso (al massimo 2 o 3). In tutti gli altri casi, e qualora si cerchino soluzione più precise, è necessario procedere oltre e utilizzare dei metodi numerici.

Algoritmi di ottimizzazione

Ci sono decine e decine di algoritmi di ottimizzazione in Scilab e ogni metodo può essere utilizzato per risolvere un problema specifico tenendo conto, ad esempio, del numero di obiettivi, del numero e il tipo di variabili *x*, oppure del numero e tipo di vincoli.

Alcuni metodi possono essere più adatti per risolvere problemi di ottimizzazione vincolata, altri possono essere più efficaci per problemi convessi, altri possono essere personalizzati per risolvere dei problemi a variabili discrete.

In Scilab sono disponibili metodi specifici per la risoluzione di programmazione quadratica, problemi non lineari, ottimizzazione multiobiettivo o di ricerca operativa.

La tabella 1 fornisce una panoramica degli algoritmi di ottimizzazione disponibili in Scilab. Molti altri metodi di ottimizzazione vengono messi a disposizione dalla comunità di giorno in giorno come moduli esterni e sono accessibili attraverso il portale ATOM, http://atoms.scilab.org/ .

[3] Contatta l'autore per la versione originale e completa degli script descritti in questo tutorial.

Objective	Bounds	Equality	Inequalities	Problem size	Gradient needed	Solver
Linear	y	l	l	m	-	linpro
Quadratic	y	l	l	m	-	quapro
						qld
				l	-	qpsolve
Nonlinear	y			l	y	optim
				s	n	neldermead
						optim_ga
				s	n	fminsearch
						optim_sa
Nonlinear Least Squares				l	optional	lsqrsolve
						leastsq
Min-Max	y			m	y	optim/nd
Multi-Obj.	y			s	n	optim_moga
			l*	l		semidef
Semi-Def.		l*	l*	l	n	lmisolve

Tabella 1: Questa tabella fornisce una panoramica degli algoritmi di ottimizzazione disponibili in Scilab e il tipo di problemi di ottimizzazione che può essere risolto. Per la colonna "Equality", la "l" lettera significa "lineare". Per la dimensione del problema ("Problem Size") s, m, l indicano piccole, medie e grandi dimensioni rispettivamente, il che significa circa meno di dieci, decine o centinaia di variabili.

Per mostrare le potenzialità di Scilab come strumento di ottimizzazione, possiamo cominciare dalla funzione di ottimizzazione più utilizzata, ossia la funzione **optim.** Questo comando di fatto contiene al proprio interno una serie di algoritmi di ottimizzazione non lineare e non vincolata.

Vediamo cosa succede se si usa la funzione **optim** per sul problema precedente:

```
function [ f , g, ind ] = rosenbrock ( x , ind )
    f = 100*(x(2)-x(1)^2)^2+(1-x(1))^2;
   g(1) = - 400. * ( x(2) - x(1)^2 ) * x(1) -2. * ( 1. - x(1) )
   g(2) = 200. * ( x(2) - x(1)^2 )
endfunction
x0 = [-1.2 1];
[f, x] = optim(rosenbrock, x0);
// Display results[4]
mprintf("x = %s\n", strcat(string(x)," "));
mprintf("f = %e\n", f);
```

Se usiamo x0 = [-1.2 1] come punto iniziale, la funzione converge facilmente al punto ottimale x *= [1,1] con f = 0.0.

L'esempio precedente calcola sia il valore della funzione Rosenbrock e la sua derivata g dato che il calcolo del gradiente è richiesto dal metodo di ottimizzazione.

In molte applicazioni reali, il gradiente può essere troppo complesso da calcolare o più semplicemente non disponibile in quanto la funzione non è nota ed è disponibile solo come una

[4] Il simbolo // indica l'inizio di un commento

“black-box” (scatola nera), in altre parole come una funzione esterna della quale non conosciamo le equazioni. Per questo motivo, Scilab ha la possibilità di calcolare le derivate con differenze finite utilizzando la funzione **derivative** oppure la funzione **numdiff.**

Per esempio, il codice che segue definisce una funzione f e ne calcola la derivata su un punto specifico x.

```
function f=myfun(x)
  f=x(1)*x(1)+x(1)*x(2)
endfunction
x=[5 8]
g=numdiff(myfun,x)
```

Queste due funzioni (**derivative** and **numdiff**) possono essere utilizzate insieme ad **optim** per cercare le soluzioni ottime di problemi dove la derivata è troppo complicato da programmare.

La funzione **optim** utilizza un metodo quasi-Newton basato sul **BFGS,** un algoritmo rapido e accurato per l'ottimizzazione locale. Sull’esempio precedente, possiamo anche provare ad applicare un approccio completamente diverso come il Nelder-Mead **Simplex** [1] che non richiede il calcolo delle derivate ed è implementato nella funzione **fminsearch**.

Per fare questo, rispetto all’esempio precedente, dobbiamo solo sostituire la riga:

```
[F, x] = optim (Rosenbrock, x0);
```

Con

```
[X, f] = fminsearch (Rosenbrock, x0);
```

Questo metodo, partendo dallo stesso punto iniziale, converge molto vicino al punto ottimale in x *= [1.000022 1.0000422] con f = 8.177661e-010 ma non esattamente a zero. Questo dimostra che il secondo approccio è meno accurato di quello precedente. Nulla di cui stupirsi, questo è il prezzo da pagare per avere un approccio più robusto rispetto agli ottimi locali e alle possibili perturbazioni imprevedibili della funzione (funzioni rumorose).

La figura 2 mostra la convergenza del metodo Nelder-Mead simplex sulla funzione Rosenbrock.

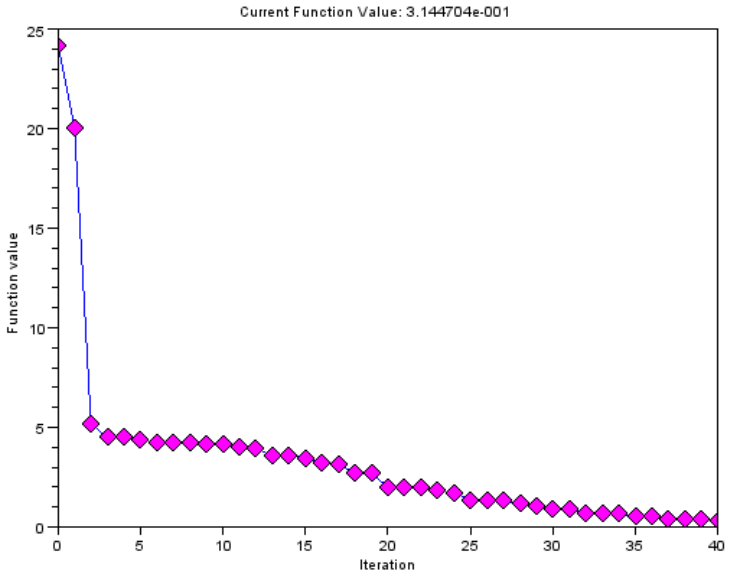

Figura 2: La convergenza dell'algoritmo Nelder-Mead Simplex (funzione fminsearch) sull'esempio Rosenbrock.

E ' importante dire che, nell'esempio precedente, la funzione è data per mezzo di uno script di Scilab ma questo è stato fatto in questo tutorial solo per semplicità. E' infatti sempre possibile valutare la funzione f come "funzione esterna" (o "black-box"), ossia come ad esempio un programma in C, in Fortran, in Java o un qualunque altro solutore commerciale.

Identificazione di parametri

In questo breve paragrafo mostriamo un problema specifico di ottimizzazione molto comune in ingegneria. Faremo vedere come è facile e veloce l'identificazione di parametri per sistemi non lineari a partire da una tabella di dati input/output.

Supponiamo per esempio di disporre di un certo numero di misure nella matrice **X** con valore di output nel vettore Y. Supponiamo di conoscere la funzione che descrive il modello **(FF)** a meno di un certo numero di parametri **p**. Il nostro scopo è appunto dare un valore a questi parametri. Per farlo, è sufficiente scrivere poche righe di codice Scilab:

```
//modello con parametri
function y=FF(x, p)
    y=p(1)*(x-p(2))+p(3)*x.*x;
endfunction

Z=[Y;X];
//Criterio di valutazione dell'errore
function e=G(p, z)
   y=z(1),x=z(2);
   e=y-FF(x,p),
endfunction

//Risolve il problema partendo da un punto iniziale p0
p0=[1;2;3]
```

```
[p,err]=datafit(G,Z,p0);
```

Questo metodo è molto veloce, efficiente è può trovare un elevato numero di parametri. E' inoltre possibile, con questo metodo, dare un intervallo di validità per i parametri cercati e attribuire dei pesi alle diverse misure in modo da dare una maggiore importanza ad un punto piuttosto che ad altri.

Algoritmi evolutivi: metodi genetici e multiobiettivo

Gli algoritmi genetici [2] sono metodi di ricerca ispirati al principio di evoluzione e selezione naturale. Questi metodi sono ampiamente utilizzati per risolvere problemi reali altamente non-lineari per la loro capacità di essere robusti e di non lasciarsi ingannare dagli ottimi locali.

Gli algoritmi genetici sono largamente utilizzati nel mondo reale e, ovviamente, anche in un certo numero di ambiti ingegneristici in cui i metodi "classici" falliscono o non forniscono risultati soddisfacenti.

Utilizzare gli algoritmi genetici in Scilab è molto semplice: in poche righe è possibile impostare i parametri richiesti come il numero di generazioni, la dimensione della popolazione, la probabilità di cross-over e la mutazione. La Fig. 3 mostra il comportamento della funzione **optim_ga** sulla funzione Rosenbrock. Venti punti iniziali random (in giallo) evolvono attraverso 50 generazioni, verso il punto ottimale. La generazione finale è in rosso.

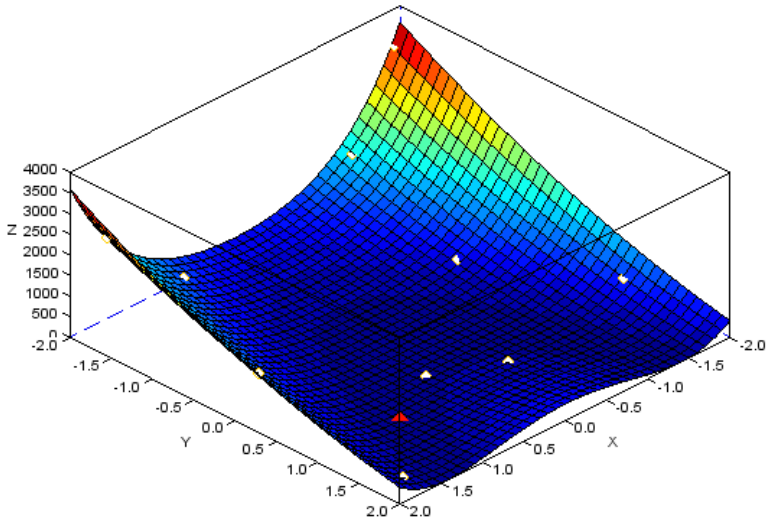

Figura 3: L'ottimizzazione della funzione Rosenbrock con un algoritmo genetico. La popolazione iniziale casuale è in giallo, la popolazione finale è in rosso e converge verso la vera soluzione ottimale.

Ottimizzazione Multiobiettivo

Scilab non solo è in grado di risolvere problemi con un singolo obiettivo ma può anche facilmente trattare problemi di ottimizzazione multiobiettivo.

Giusto per fare un esempio, , gli utenti Scilab possono usare il metodo NSGA-II direttamente all'interno del sistema. NSGA-II è la seconda versione del famoso " Non-dominated Sorting Genetic Algorithm " basato sul lavoro del Prof. Kalyanmoy Deb [3].

La Figura 4 mostra il risultato di una ottimizzazione multiobiettivo con NSGA-II sulla funzione test ZDT1.

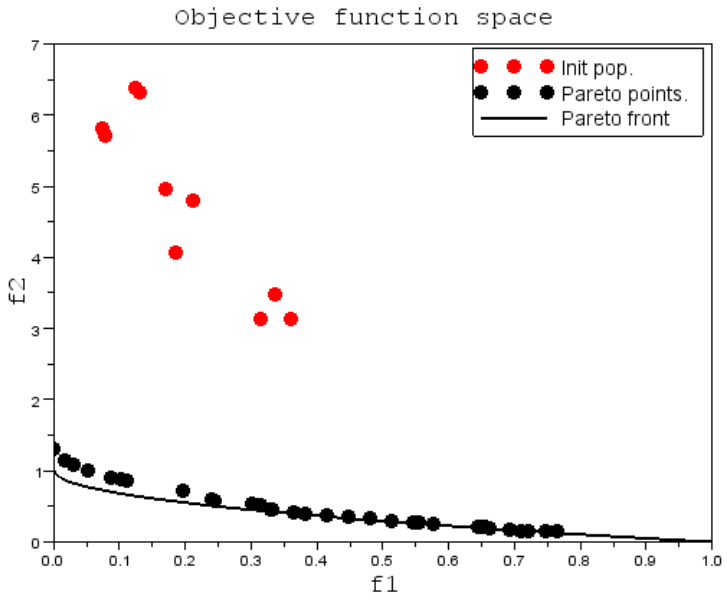

Figura 4: Problema ZDT1 risolto con il famoso metodo di ottimizzazione NSGA-II. I punti rossi indicano la popolazione iniziale, i punti neri le soluzioni finali di Pareto. La linea rappresenta la vera frontiera di Pareto che, in questo particolare caso, è continua e convessa.

```
function f=zdt1(x)

f1_x1 = x(:,1);
g_x2  = 1 + 9 * ((x(:,2)-x(:,1)).^2);
h     = 1 - sqrt(f1_x1 ./ g_x2);

f(:,1) = f1_x1;
f(:,2) = g_x2 .* h;
endfunction
```

Con il ZDT1 vogliamo minimizzare contemporaneamente sia il valore di f1 che quello di f2. Questo approccio ci indica che siamo di fronte ad un problema multiobiettivo in cui la nozione di soluzioni ottimali cambia.

Una ottimizzazione multiobiettivo non produce una soluzione unica, ma un insieme di soluzioni ottimali. Queste soluzioni si chiamano soluzioni **non-dominate**[5] o punti di **Pareto,** l'insieme delle soluzioni può essere chiamato **frontiera di Pareto.**

La Figura 4 mostra le soluzioni calcolate in Scilab con l'algoritmo NSGA-II sul problema ZDT1. I punti rossi sulla parte superiore sono la popolazione iniziale (punti casuali), i punti neri sul fondo sono i punti di Pareto della popolazione finale. La linea continua rappresenta la vera frontiera di Pareto che, in questo esempio specifico, rappresenta un fronte continuo e convesso ed è nota.

Questo esempio ci può chiarire il concetto di dominanza di Pareto. I punti rossi sulla parte superiore sono certamente dominati dai punti neri sul fondo perché tutti i punti rossi sono peggiori dei punti neri sia per quanto riguarda l'obiettivo f1 che per quanto riguarda l'obiettivo f2. Al contrario, tutti i punti neri sulla figura non si dominano tra loro, possiamo dire in questo caso che tutti i punti neri rappresentano l'insieme delle soluzioni efficienti.

Per capire come Scilab riconosca l'importanza dell'ottimizzazione multiobiettivo, si può anche notare che ha una funzione interna chiamata **pareto_filter** che è in grado di filtrare automaticamente l'insieme delle funzioni non dominate da una tabella di dati.

```
X_in=rand(1000,2);
F_in=zdt1(X_in);
[F_out,X_out,Ind_out] = pareto_filter(F_in,X_in)
drawlater;
plot(F_in(:,1),F_in(:,2),'.r')
plot(F_out(:,1),F_out(:,2),'.b')
drawnow;
```

Il codice precedente genera 1000 valori di input casuali, valuta la funzione zdt1 e identifica l'insieme delle soluzioni non-dominate. Le ultime quattro righe del codice generano il seguente grafico con tutti i punti in rosso ed i punti di Pareto in blu.

[5] Per definizione diciamo che una soluzione a domina una soluzione b se [f1(a)<=f1(b) AND f2(a)<=f2(b) ... AND fk(a)<=fk(b)], per ogni f AND [f1(a)<f1(b) OR f2(a)<f2(b) ... OR fk(a)< fk(b)] per almeno un obiettivo f

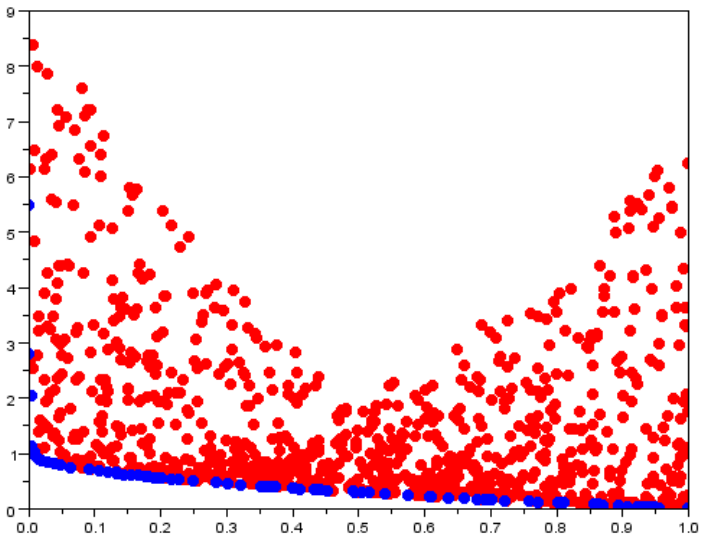

Figura 5: 1.000 punti casuali sulla funzione di test zdt1. I punti blu sono le soluzioni non-dominate (di Pareto). Il codice per la selezione automatica delle soluzioni è riportato nel testo. La funzione principale da usare è "pareto_filter".

Risolvere un problema di taglio: ridurre gli scarti.

Il problema di taglio è un problema di ottimizzazione molto comune nelle industrie, ed è economicamente significativo. Il problema consiste nel trovare il modo ottimale per tagliare un semilavorato in diverse dimensioni al fine di soddisfare una serie di richieste dei clienti , utilizzando il materiale nel modo più efficiente possibile.

Questo tipo di problema è molto frequente e può comportare una serie di obiettivi diversi. La riduzione dei costi può avvenire riducendo al minimo il numero dei tagli, riducendo al minimo lo spreco di materiale, e così via. Qualunque sia il target, piccoli miglioramenti nel layout di taglio possono comportare notevoli risparmi di materiale e sensibile riduzione dei costi di produzione.

In questa sezione mostreremo come risolvere un problema unidimensionale (1D) di taglio con Scilab. Risolvere un problema di taglio 1D è certamente meno complesso di risolvere un problema in 2D in cui devono essere tagliati dei rettangoli da un foglio. Tuttavia problemi di taglio 1D sono molto comuni e rappresentano un problema interessante.

Questo genere di problemi 1D possono sorgere, ad esempio, nell'edilizia dove sono necessarie barre di acciaio in specifiche quantità e lunghezze e queste barre sono tagliate da barre con lunghezze standard.

Supponiamo ora che lavoriamo per una azienda produttrice di tubi, che di solito hanno una lunghezza fissa, e sono ora in attesa di essere tagliati. Questi tubi devono essere tagliati in diverse lunghezze per soddisfare le richieste dei clienti. Come possiamo tagliare i tubi in modo da minimizzare la quantità totale di scarti?

La formulazione matematica per i problemi di taglio 1D si scrive in questo modo:

$$\min_x \sum_{i=1}^{n} c_i x_i$$

$$con \sum_{i=1}^{n} a_{ij} x_i = q_j \quad e \quad j = 1, \dots, m$$

Dove, *i* è l'indice dei **pattern (modelli di taglio)**, *j* l'indice delle lunghezze, x_i sono il numero di modelli di taglio *i* (variabili decisionali) e c_i sono i costi legati al modello di taglio *i*. $A = (a_{ij})$ la matrice di tutti i modelli possibili e q_j richieste dei clienti. Possiamo dire che il valore a_{ij} indica il numero di pezzi di *j* lunghezza all'interno di un tubo tagliato col modello di taglio i.

Lo scopo è quello di minimizzare la funzione obiettivo ossia i totale dei costi di taglio. Se c_i è uguale a 1 per tutti i modelli di taglio, l'obiettivo corrisponde a minimizzare il numero totale di tubi necessari per soddisfare tutte le richieste dei clienti.

Facciamo un esempio pratico e risolviamolo con Scilab. Supponiamo di avere 3 misure possibili, 55mm, 26mm e 24mm da tagliare da un tubo originale di 100 mm. I possibili modelli di taglio sono:

1. Un taglio di tipo uno, uno di tipo due, e zero del tipo tre [1 1 0]
2. Un taglio di tipo uno e uno di tipo tre [1 0 1]
3. Due tagli di tipo due e due di tipo tre [0 2 2]

Questi modelli di taglio definiscono la matrice A. Poi ci sono i costi che sono 4, 3 e 1 per il modello 1, 2 e 3 rispettivamente. La richiesta totale da parte dei clienti sono: 150 pezzi di lunghezza 55 mm, 200 con lunghezza pari a pezzi 26mm e 300 con lunghezza 24 mm.

Per risolvere questo problema in Scilab possiamo usare questo script.

```
//pattern
aij=[ 1 1 0;
      1 0 1;
      0 2 2];
//costs
ci=[4; 3; 1];
//request
qj=[150; 200; 300];

xopt = karmarkar(aij',qj,ci)
```

Con Scilab otteniamo xopt = [25, 125, 87,5]. Questo significa che per soddisfare le richieste dei clienti e ridurre al minimo numero totale di tubi dobbiamo tagliare 25 volte il modello di taglio (1), 125 il tempo con il modello di taglio (2) e 87,5 volte il modello di taglio (3).

Questo caso potrebbe sembrare semplice con solo tre richieste diverse e tre diversi modelli. Il problema può essere molto più complicato, con molte più opzioni, molte dimensioni diverse, costi e richieste. Il problema potrebbe includere un numero massimo di tagli su un unico pezzo, e potrebbe richiedere un po' di sforzo nel generare l'elenco di tutti i possibili modello di taglio (ossia

la nostra matrice A). Tutte queste difficoltà possono essere codificate con Scilab e la logica che sottende l'approccio rimane la stessa.

Lo script precedente utilizza l'algoritmo di Karmarkar [4] per risolvere questo problema lineare. Il risultato non è una soluzione intera, quindi abbiamo bisogno di approssimare perché non possiamo tagliare 87,5 tubi con il terzo modello. Questa soluzione approssimata può essere migliorata con un altro algoritmo di ottimizzazione diverse, ad esempio valutando la più vicina soluzioni intere o utilizzando un algoritmo genetico più robusto. Ma anche se ci fermiamo con il primo passo e prendendo una soluzione arrotondata abbiamo sicuramente una buona riduzione degli scarti.

Conclusioni

Come la soluzione del problema di taglio dimostra, Scilab non è uno strumento educativo, ma un vero e proprio prodotto in grado di risolvere dei veri problemi industriali. Il problema di taglio è un problema comune nelle industrie, e una buona soluzione può portare a notevoli risparmi.

Per amor di semplicità, questo tutorial mostra solo funzioni molto semplici che sono state utilizzate con lo scopo di fare una breve e generale introduzione. Ovviamente queste funzioni semplici possono essere sostituite da altre più complesse come ad esempio solutori FEM o codici di simulazione esterni.

Gli utenti MATLAB avranno di sicuro riconosciuto le similitudini tra il software commerciale e Scilab. Ci auguriamo che tutti gli altri lettori non siano stati spaventati dal fatto che per risolvere problemi di ottimizzazione si debbano utilizzare degli script. Una volta che la logica è chiara, scrivere degli script potrà sembrare un'attività semplice ed divertente.

Per ulteriori informazioni
Silvia Poles - EnginSoft
s.poles@enginsoft.it

Referenze

[1] Nelder, John A.; R. Mead (1965). "A simplex method for function minimization". Computer Journal 7: 308–313

[2] David E. Goldberg. Genetic Algorithms in Search, Optimization & Machine Learning. Addison-Wesley, 1989.

[3] N. Srinivas and Kalyanmoy Deb. Multiobjective optimization using nondominated sorting in genetic algorithms. Evolutionary Computation, 2:221--248, 1994

[4] Narendra Karmarkar (1984). "A New Polynomial Time Algorithm for Linear Programming", Combinatorica, Vol 4, nr. 4, p. 373–395.