Università degli Studi di Padova
Ingegneria dell' Automazione

DEPARTMENT OF INFORMATION ENGINEERING
UNIVERSITY OF PADOVA

Corso di Progettazione di Sistemi di Controllo
A.A. 2011-2012

Introduzione

Consensus Newton-Raphson per l'ottimizzazione distribuita di funzioni convesse

Presentazione di

Giulio Veronesi.

Gruppo:

Andrea Bernardi, Enrico Regolin e Giulio Veronesi.

Overview

Lavoro di approfondimento dell'algoritmo:

F. Zanella, D. Varagnolo, A. Cenedese, G. Pillonetto, L. Schenato

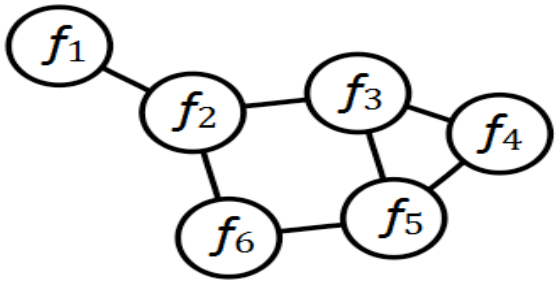
"**Newton-Raphson consensus for distributed convex optimization**"

[http://paduareserch.cab.unipd.it/ Tech Rep - 2011]

- Descrizione l'algoritmo
- Applicazioni ed implementazioni
- Studio e tuning del parametro fondamentale

Ottimizzazione distribuita convessa

- Rete di agenti che devono cooperare per trovare il punto di ottimo



- Minimizzazione distribuita di una funzione costo globale, somma di tutte le funzioni costo convesse private :

$$\overline{f} : \mathbb{R} \to \mathbb{R} \qquad \overline{f}(x) := \frac{1}{N} \sum_{i=1}^{N} f_i(x)$$

- Gli agenti devono raggiungere il consenso nel punto di ottimo:

$$x^* = \operatorname*{argmin}_x \overline{f}(x)$$

Motivazioni

- Approccio distribuito vantaggioso rispetto a quello centralizzato:
  - Minori oneri computazionali
  - Economicità in termini energetici
  - Convergenza al punto di minimo e robustezza
- Problemi di ottimizzazione convessa positivamente ed efficacemente risolvibili
- Applicazione in casi di interesse come analisi di dati, problemi di regressione, stima distribuita, allocazione di risorse, mapping, etc.

Osservazioni sulle funzioni costo

- Funzioni costo quadratiche: $$f_i(x) = \frac{1}{2} a_i (x - b_i)^2$$

  Il minimo vale: $$x^* = \frac{\frac{1}{N}\sum_{i=1}^{N} a_i b_i}{\frac{1}{N}\sum_{i=1}^{N} a_i}$$

- Possiamo generalizzare al caso di funzioni costo convesse generiche ponendo:
  (approssimazione locale con una parabola)

$$a_i b_i = f_i''(x_i) x_i - f_i'(x_i) =: g_i(x_i)$$

$$a_i = f_i''(x_i) =: h_i(x_i)$$

→ PARALLELO DI 2 AVERAGE CONSENSUS

L'algoritmo

Aggiornamento variabili con consensus

Calcolo nuovo stato

(variabili)

1: $\mathbf{x}(k), \mathbf{y}(k,m), \mathbf{z}(k,m) \in \mathbb{R}^N, m = 0, \ldots, M; k = 0, 1, \ldots$

(parametri)

2: $P \in \mathbb{R}^{N \times N}$ matrice di consenso positiva e doppiamente stocastica

3: $\varepsilon \in (0,1)$

(initializzazione)

4: $\mathbf{x}(0) = \mathbf{0}, \mathbf{g}(\mathbf{x}(-1)) = \mathbf{h}(\mathbf{x}(-1)) = \mathbf{0}$

5: $\mathbf{y}(0,M) = \mathbf{z}(0,M) = \mathbf{0}$

(algoritmo)

6: **for** $k = 1, 2, \ldots$ **do**

7: $\quad \mathbf{y}(k,0) = \mathbf{y}(k-1,M) + \mathbf{g}(\mathbf{x}(k-1)) - \mathbf{g}(\mathbf{x}(k-2))$

8: $\quad \mathbf{z}(k,0) = \mathbf{z}(k-1,M) + \mathbf{h}(\mathbf{x}(k-1)) - \mathbf{h}(\mathbf{x}(k-2))$

9: $\quad$ **for** $m = 1, 2, \ldots, M$ **do**

10: $\quad\quad \mathbf{y}(k,m) = P\mathbf{y}(k,m-1)$

11: $\quad\quad \mathbf{z}(k,m) = P\mathbf{z}(k,m-1)$

12: $\quad$ **end for**

13: $\quad \mathbf{x}(k) = (1-\varepsilon)\mathbf{x}(k-1) + \varepsilon \frac{\mathbf{y}(k,M)}{\mathbf{z}(k,M)}$

14: **end for**

- ε rende meno aggressiva la stima dello stato, opera una combinazione convessa tra stato vecchio e nuovo

Commenti

- *Primal decomposition methods*:
  - Facile da implementare
  - Poche ipotesi (convessità funzioni costo)
  - Sincronizzazione agenti non necessaria
- Grafo associato non orientato e connesso
- Utilizzata P doppiamente stocastica, convergenza nella media degli stati iniziali (*average consensus*):

$$\frac{1}{N}\sum_{i=1}^{N} x_i(0)$$

- Implementazione con P costante e gossip

Doppia scala temporale - I

- Partendo dal sistema a tempo continuo

$$\begin{cases} \varepsilon\dot{\mathbf{v}}(t) = -\mathbf{v}(t) + \mathbf{g}(\mathbf{x}(t)) \\ \varepsilon\dot{\mathbf{w}}(t) = -\mathbf{w}(t) + \mathbf{h}(\mathbf{x}(t)) \\ \varepsilon\dot{\mathbf{y}}(t) = -K\mathbf{y}(t) + (I-K)[\mathbf{g}(\mathbf{x}(t)) - \mathbf{v}(t)] \\ \varepsilon\dot{\mathbf{z}}(t) = -K\mathbf{z}(t) + (I-K)[\mathbf{h}(\mathbf{x}(t)) - \mathbf{w}(t)] \\ \dot{\mathbf{x}}(t) = \mathbf{x}(t) + \frac{\mathbf{y}(t)}{\mathbf{z}(t)} \end{cases}$$

- Discretizzando con il metodo di Eulero in avanti, $T = \varepsilon$, per ε piccoli il sistema può essere scritto nella forma ($P = I - K$):

$$\begin{cases} \mathbf{v}(k+1) = \mathbf{g}(\mathbf{x}(k)) \\ \mathbf{w}(k+1) = \mathbf{h}(\mathbf{x}(k)) \\ \mathbf{y}(k+1) = P[\mathbf{y}(k) + \mathbf{g}(\mathbf{x}(k)) - \mathbf{v}(k)] \\ \mathbf{z}(k+1) = P[\mathbf{z}(k) + \mathbf{h}(\mathbf{x}(k)) - \mathbf{w}(k)] \\ \mathbf{x}(k+1) = (1-\varepsilon)\mathbf{x}(k) + \varepsilon\frac{\mathbf{y}(k)}{\mathbf{z}(k)} \end{cases}$$

Doppia scala temporale - II

- Sistema a doppia scala temporale, regolata dal parametro ε (studio successivo)
- Sistemi con una dinamica veloce (prime 4 equazioni - Consensus) ed una lenta (ultima equazione - NR)

  → *Standard singular perturbation models* (Khalil)
- Si giunge all'approssimazione: $\mathbf{x}(t) \approx \overline{x}(t)\mathbb{1}$ $\quad \dot{\overline{x}}(t) = -\dfrac{\overline{f}'(\overline{x}(t))}{\overline{f}''(\overline{x}(t))}$
- Si tratta di un Algoritmo di Newton-Raphson che converge al punto di minimo, formula generale:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \xrightarrow{k \to +\infty} x^*$$

Teorema fondamentale

- Teorema:

$f_i \in C^2$ convesse, condizioni iniziali nulle, $x^* \neq \pm\infty$, $\exists$ $w^* \in (0,1)$ tale che $\forall w < w^*$ si ha: $\lim_{k\to+\infty} \boldsymbol{x}(k) = x^* \mathbf{1}$

- Proposizione

Gli stati iniziali di $\boldsymbol{x}$ possono essere arbitrari, invece le condizioni iniziali di $\boldsymbol{v}, \boldsymbol{w}, \boldsymbol{y}, \boldsymbol{z}$ devono essere suff. piccole

$\boldsymbol{\varepsilon}$ errato:

- Troppo grande: divergenza variabili di stato, non si giunge al minimo
- Troppo piccolo: lentezza di raggiungimento del consensus

P gossip

- Algoritmi Gossip:
  - Riduzione numero trasmissioni necessarie
  - Evitano collisioni ed errori nello scambio dati
- Implementazione con *P* gossip – *probabilistic consensus*
  - Minor sincronismo tra i nodi
  - Comunicazione tra nodi distanti al più 4 lati
  - Poche modifiche all'algoritmo precedente
- La matrice *P* cambia ad ogni iterazione
- Risultati paragonabili alla versione sincrona dell'algoritmo

Applicazioni: stima distribuita

- Si vuole stimare il valore medio di una serie di misure rilevate da sensori in maniera distribuita (inquinamento, temperatura, umidità, etc.)
- Approccio distribuito molto più efficiente del classico approccio centralizzato:
  - Meno bit trasmessi
  - Minor dispendio energetico
- È un problema di ottimizzazione convessa:

$$x^* = \operatorname*{argmin}_x \overline{f}(x) = \operatorname*{argmin}_x \frac{1}{n}\sum_{i=1}^{n} f_i(x) \qquad f_i(x) = \frac{1}{m}\sum_{j=1}^{m}(x_{i,j} - x)^2$$

- Possiamo applicare l'algoritmo, si avrà convergenza nel punto di ottimo/media.

Applicazioni: regressioni

- Problema di identificazione di una funzione da una serie di dati rilevati.
- Parametrizzazione per trovare la funzione di interesse:

$$f_\theta(x) = \sum_{i=1}^{m} \theta_i h_i(x)$$

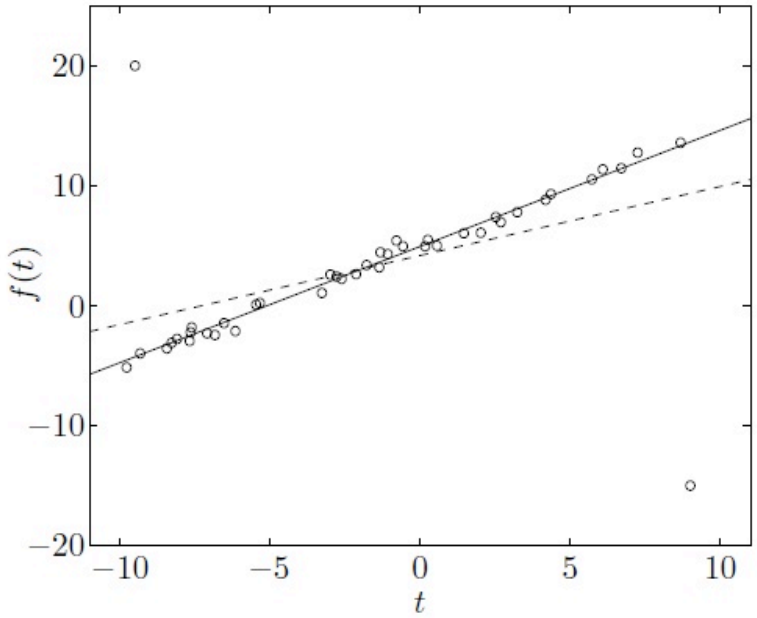
- Parametri incogniti determinabili attraverso (problema di minimizzazione) :

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \sum_{i=1}^{N} (y_i - f_\theta(x_i))^2$$

NB. È un problema di regressione lineare (utilizzo di funzioni quadratiche – *minimizzazione ai minimi quadrati*) ➡ Poca robustezza agli *outliers*

Applicazioni: regressioni

- *Outliers*: valori anomali rispetto alle altre misure.
- Metodi di regressione robusta: contrastano la presenza di *outliers* attraverso funzioni costo apposite.
- Presenza di 2 *outliers*, differenze di risultati:
  - Funzione costo quadratica (retta tratteggiata)
  - Funzione costo di *Laplace* (retta continua)
- Implementazione algoritmo con funzioni costo "robuste"



Altre applicazioni trattate:

- Calibrazione di sensori
- Allocazione di risorse
- Localizzazione di una fonte acustica

In tutte le applicazioni presentate vi è la minimizzazione distribuita di una cifra di merito

si può utilizzare l'algoritmo presentato

Consensus Newton-Raphson per l'ottimizzazione distribuita di funzioni convesse

Applicazione: Mapping

Presentazione di *Enrico Regolin*

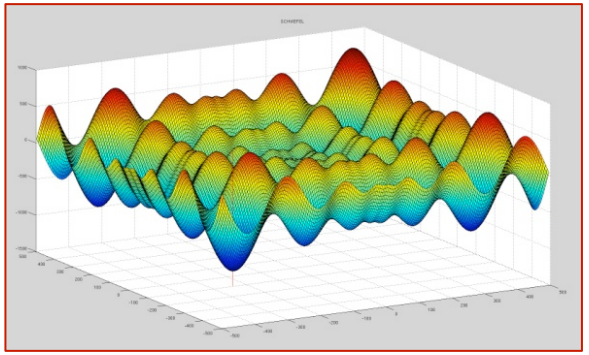
Gruppo: *Andrea Bernardi, Enrico Regolin e Giulio Veronesi.*

Mapping 1

- Perché mapping?
  - Applicazione pratica dell'algoritmo proposto in [1]
  - Utilizzo delle funzioni costo robuste viste in precedenza

→ Distributed Mapping

Ricostruzione di un profilo come somma di funzioni elementari



- Multivariable Distributed Newton-Raphson
  - Rumore gaussiano additivo
  - OUTLIERS

[1] F. Zanella, D. Varagnolo, A. Cenedese, G. Pillonetto, L. Schenato, *"Newton-Raphson consensus for distributed convex optimization"*, http://paduareserch.cab.unipd.it/, Tech Rep., 2011.

Mapping 2

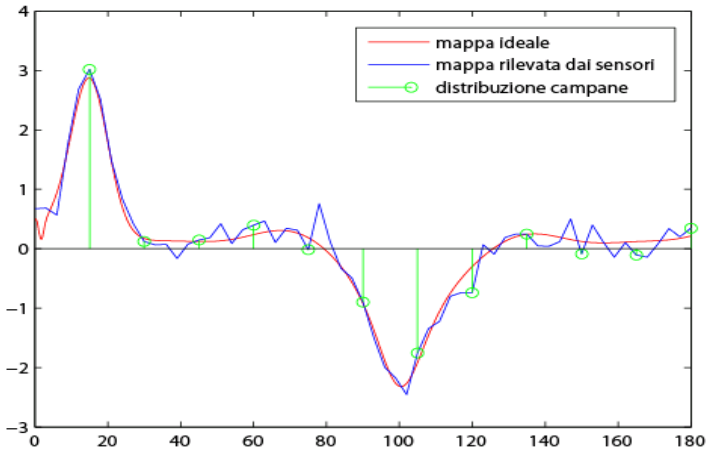
- Ricostruzione di una mappa monodimensionale
  - Temperatura/umidità lungo un corridoio
  - Livello di inquinamento di un fiume

- Somma di "campane" equispaziate $c_i(x) = a_i e^{\frac{(x-w_i)}{R}^2}$
  - Larghezza e distanza tra le campane dipendono dal tipo di mappa che ci si aspetta
  - Il contributo di ciascuna campana è pressoché nullo nei punti distanti
  - Numero di sensori N diverso dal numero di campane M
  - Mappa ricostruita:

$$C(x, \mathbf{a}(k)) = \sum_{i=1}^{M} c_i(x, a_i(k))$$

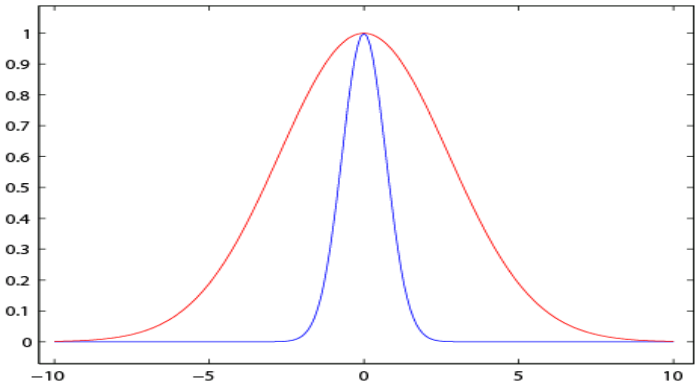
Mapping 3

- Caratteristiche mappa
  - Lunghezza: 180
  - Distanza tra 2 sensori: 3 (61 totali)



- Caratteristiche campane
  - Distanza tra due campane: 15 (13 totali)
  - Fattore R=15

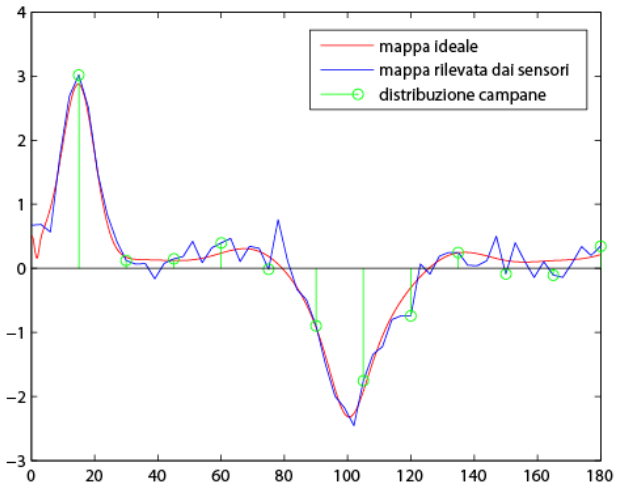
$$c_i(x) = a_i e^{\frac{(x-w_i)^2}{R}}$$



Mapping: Newton-Raphson

- Soluzione del problema di mapping come ricerca del minimo di una funzione costo
  - Somma delle distanze pesate tra il valore acquisito e la ricostruzione della mappa in quel punto.
  - Funzione costo globale:
  - Il valore della singola funzione costo dipende dalla funzione peso che si usa per l'errore!

$$F(\mathbf{a}) = \sum_{i-1}^{N} f_i(\mathbf{a}), \quad F : \mathbb{R}^M \rightarrow \mathbb{R}$$



- Soluzione
  - Newton-Raphson classico:

$$\mathbf{a}(n+1) = \mathbf{a}(n) - [H_F(\mathbf{a}(n))]^{-1} \nabla F(\mathbf{a}(n))$$

  - Distributed Newton-Raphson, caso multivariabile

Newton-Raphson centralizzato

- Bisogna calcolare ad ogni ciclo i valori del gradiente e dell'Hessiana nell'espressione: $$\mathbf{a}(n+1) = \mathbf{a}(n) - [H_F(\mathbf{a}(n))]^{-1} \nabla F(\mathbf{a}(n))$$
  - Definiamo la funzione costo $$F(\mathbf{a}) = \sum_{i=1}^{N} g(y_i - C(x_i, \mathbf{a}))$$
  - lasciando indeterminata la funzione *g(·)*, che "pesa" l'*i*-esimo errore di approssimazione $$z_i := y_i - C(x_i, \mathbf{a})$$
  - Allora calcolando il gradiente si trova:

$$\begin{bmatrix} \frac{\partial F(}{\partial a} \\ \vdots \\ \frac{\partial F(}{\partial a_1} \end{bmatrix} \nabla F(a(n)) = -S \begin{bmatrix} g'(z_1) \\ \vdots \\ g'(z_N) \end{bmatrix} \begin{matrix} \text{N}) \frac{\partial z_N}{\partial a_1} \\ \\ z_N) \frac{\partial z_N}{\partial a_M} \end{matrix}$$

Newton-Raphson centralizzato

- Espressione della matrice costante S che interviene nel calcolo del gradiente

$$S = \begin{bmatrix} e^{-(x_1-w_1)^2} & \dots & e^{-(x_N-w_1)^2} \\ e^{-(x_1-w_2)^2} & \dots & e^{-(x_N-w_2)^2} \\ \vdots & \ddots & \vdots \\ e^{-(x_1-w_M)^2} & \dots & e^{-(x_N-w_M)^2} \end{bmatrix}$$

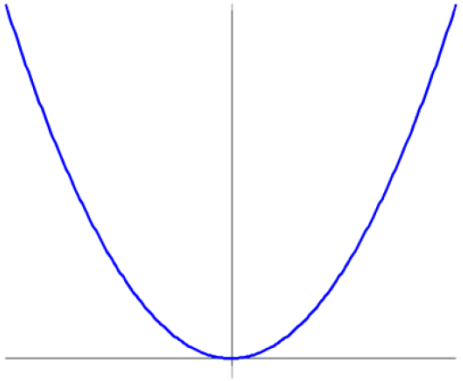
  - S è determinata univocamente dalla posizione di sensori e ”campane”.
- La matrice hessiana delle derviate parziali seconde è

$$H_F(a) = SG''S^T \qquad G'' = \begin{bmatrix} g''(z_1) & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & g''(z_N) \end{bmatrix}$$

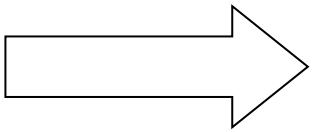
  - Per sicurezza si somma una matrice diagonale δI_M prima dell’inversione della matrice Hessiana.

Funzioni costo

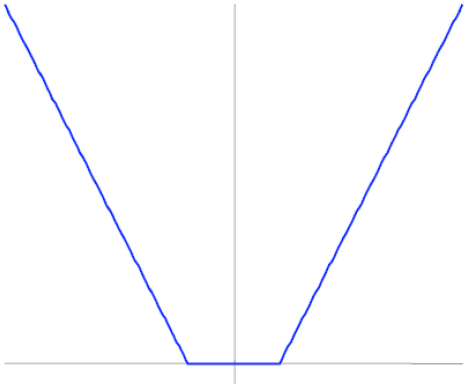
- Scelta della funzione "peso" dell'errore.
  - Quadratica (calcoli piú semplici)
  - Laplace (contrasto degli outliers)
    - In presenza di outliers conviene adottare una funzione costo di Laplace



- Problema: per applicare Newton-Raphson è necessario che la funzione sia derivabile 2 volte

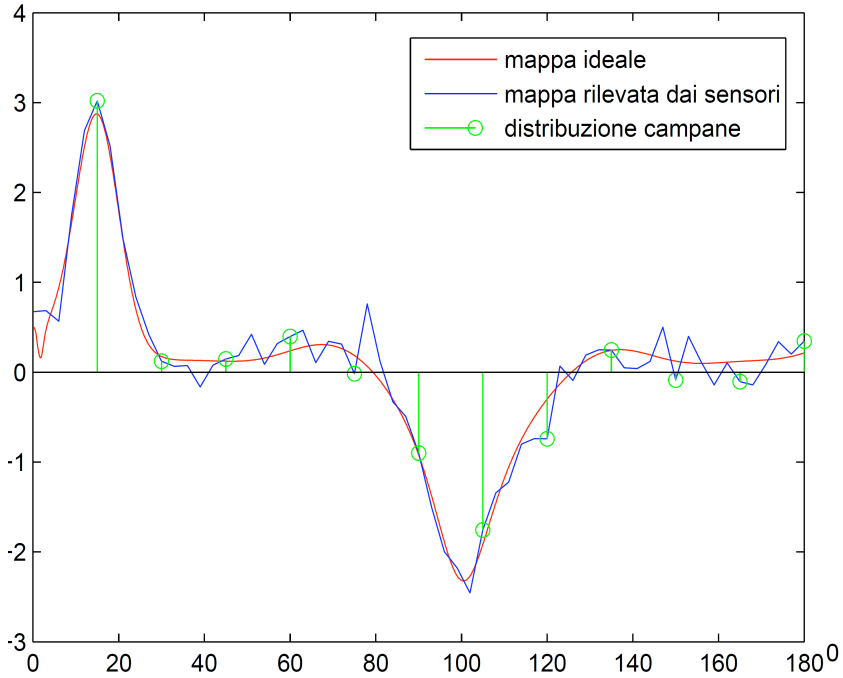


Ramo di iperbole con asintoti coincidenti con i rami della funzione di Laplace

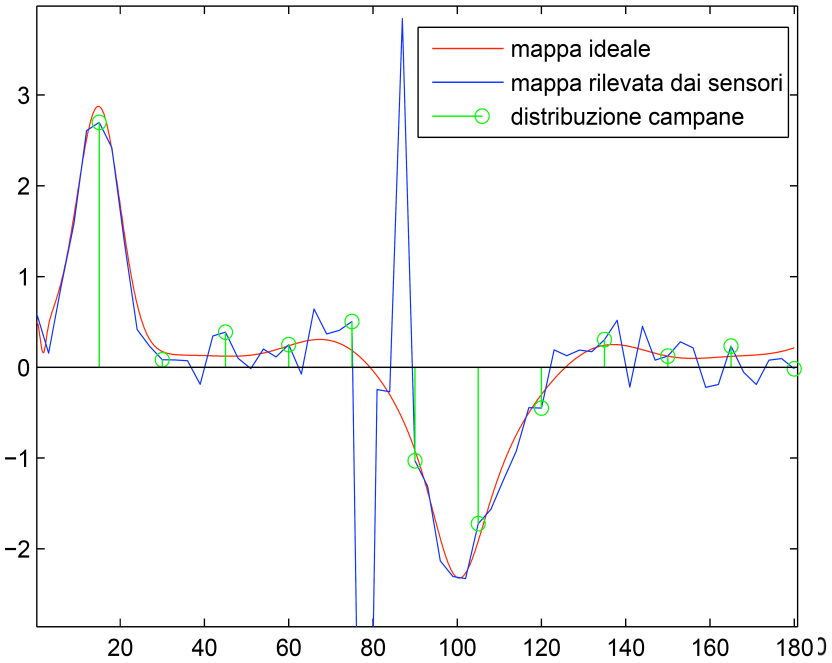


Newton-Raphson centralizzato

- No outliers



- Outliers



- A regime:
  - Iperbolica: < 50 iterazioni
  - Quadratica: < 15 iterazioni

Distributed Newton-Raphson

- Distributed Newton Raphson: scalare e multivariabile

(algoritmo)

9: **for** $n = 1, 2, \ldots$ **do**

(aggiornamenti locali)

10: **for** $i = 1$ to N **do**

11: $g_i(n) = \nabla^2 f_i(x_i(n))x_i(n) - \nabla f_i(x_i(n))$

12: $H_i(n) = \nabla^2 f_i(x_i(n))$

13: $y_i(n) = y_i(n-1) + g_i(n-1) - g_i(n-2)$

14: $Z_i(n) = Z_i(n-1) + H_i(n-1) - H_i(n-2)$

15: **end for**

("average consensus" multidimensionale)

16: $Y(n) = (P \otimes I_M)Y(n)$

17: $\mathbf{Z}(n) = (P \otimes I_M)\mathbf{Z}(n)$

(aggiornamenti locali)

18: **for** $i = 1$ to N **do**

19: $x_i(n) = (1-\varepsilon)x_i(n-1) + \varepsilon(Z_i(n))^{-1}y_i(n)$

20: **end for**

21: **end for**

Fase 1

Fase 2

Fase 3

Distributed Mapping: Fase 1

- Fase 1 (ciclo di Newton-Raphson locale)
  - Calcolo dell'errore di predizione presso ciascun sensore secondo il proprio vettore di coefficienti :

$$z_i(\mathbf{a}^{(i)}(n)) = y_i - G(x_i, \mathbf{a}^{(i)}(n))$$

  - (righe 11 e 12) Calcolo dei valori $H_i(z_i)$ e $g_i(z_i)$ a partire da gradiente e hessiana della funzione peso dell'errore locale $f(z_i)$.
    Scrivendo l'errore z_i nella forma:

$$y_i - a_1^{(i)} e^{-(x_i - w_1)^2} - \cdots - a_M^{(i)} e^{-(x_i - w_M)^2}$$

    le derivate rispetto ai coefficienti sono molto semplici da calcolare e conducono al gradiente locale:

$$\nabla f(z_i) = f'(z_i) \begin{bmatrix} \frac{\partial z_i}{\partial a_1} \\ \vdots \\ \frac{\partial z_i}{\partial a_M} \end{bmatrix} = -f'(z_i) S_i$$

Continua...

Distributed Mapping: Fase 1

- Il vettore S_i , così come la matrice S vista nel caso centralizzato, è costante.
  - É univocamente determinato dalla posizione dell'*i*-esimo sensore rispetto ai centri delle campane.

$$S_i = \begin{bmatrix} e^{-(x_i - w_1)^2} \\ \vdots \\ e^{-(x_i - w_M)^2} \end{bmatrix}$$

- La matrice delle derivate seconde è data da:

$$\frac{\partial^2 f(z_i)}{\partial a_j \partial a_k} = f''(z_i)\frac{\partial z_i}{\partial a_j}\frac{\partial z_i}{\partial a_k} + f'(z_i)\frac{\partial^2 z_i}{\partial a_j \partial a_k}$$

$$= f''(z_i)\frac{\partial z_i}{\partial a_k}\frac{\partial z_i}{\partial a_j}$$

$$= f''(z_i)e^{-(x_i - w_k)^2 - (x_i - w_j)^2}$$

$$\boxed{\nabla^2 f(z_i) = f''(z_i) S_i S_i^T}$$

Distributed Mapping: Fasi 2/3

- Fase 2 (ciclo di consenso)
  - Ogni agente scambia con i propri vicini l'informazione
  - Nella simulazione Matlab, questo avviene con il prodotto di Kronecker

$$A \otimes B = \begin{bmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{bmatrix}$$

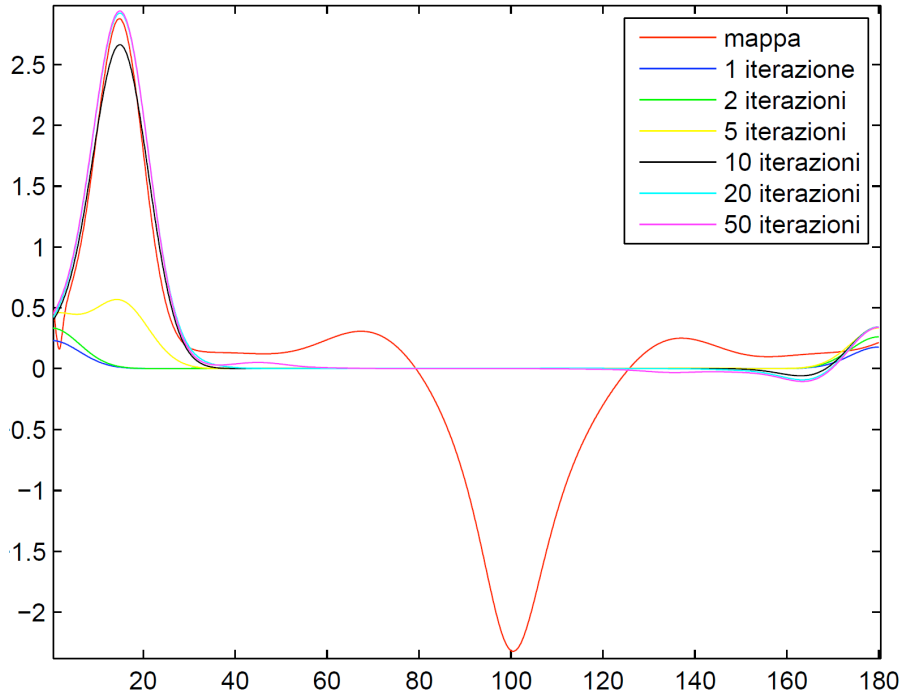
- Fase 3 (aggiornamento parametri)

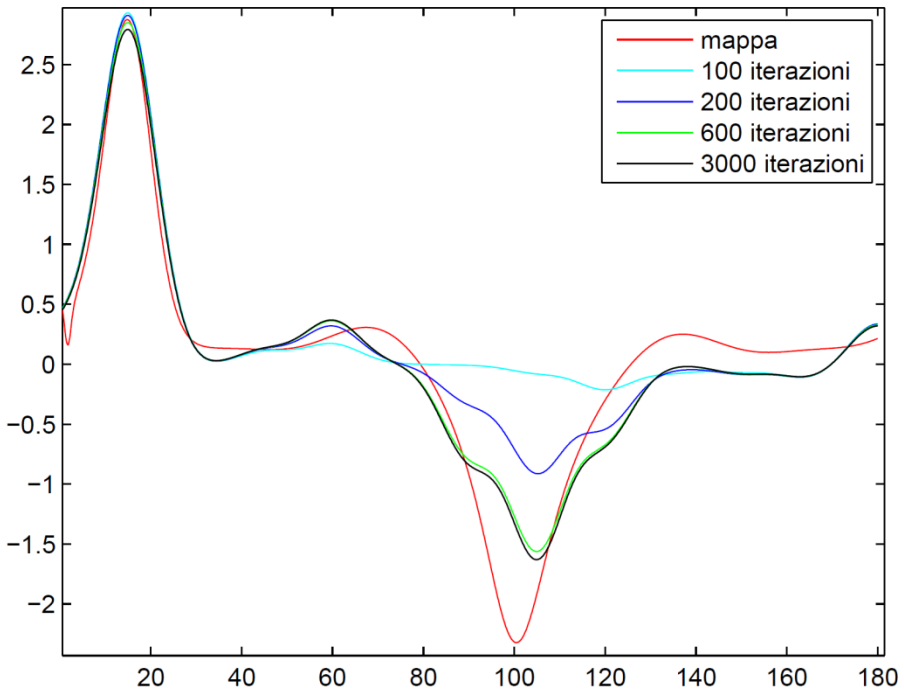
$$x_i(n) = (1-\varepsilon)x_i(n-1) + \varepsilon(Z_i(n))^{-1}y_i(n)$$

  - Attenzione all'eventuale singolarità della matrice $Z_i(n)$!
  - La pesantezza dell'algoritmo dipende dalla dimensione di $Z_i(n)$.

Distributed Mapping:Risultati 1

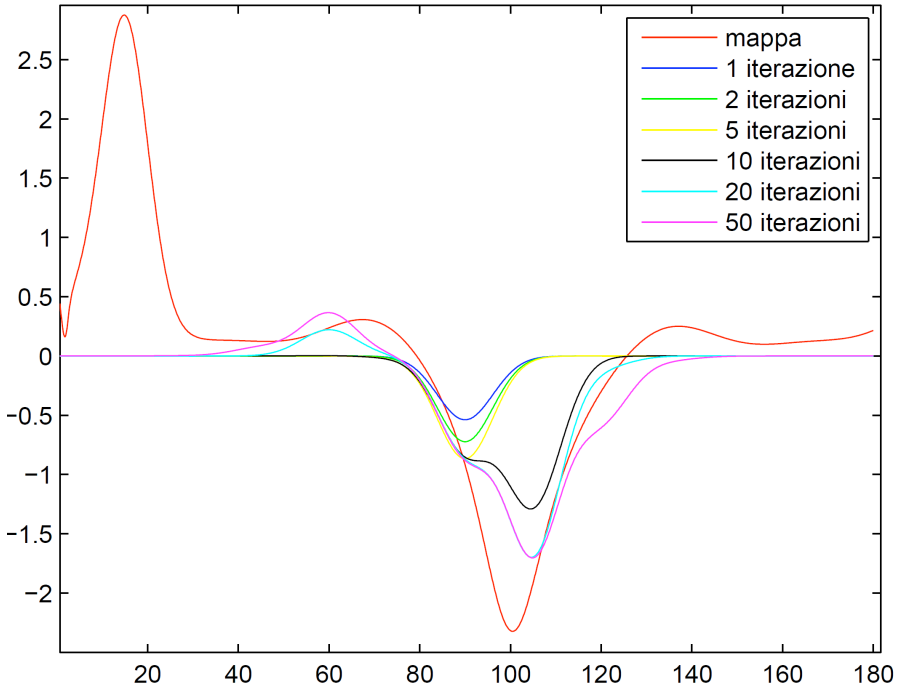
- Mappa disponibile presso l'agente 1 (su 61)

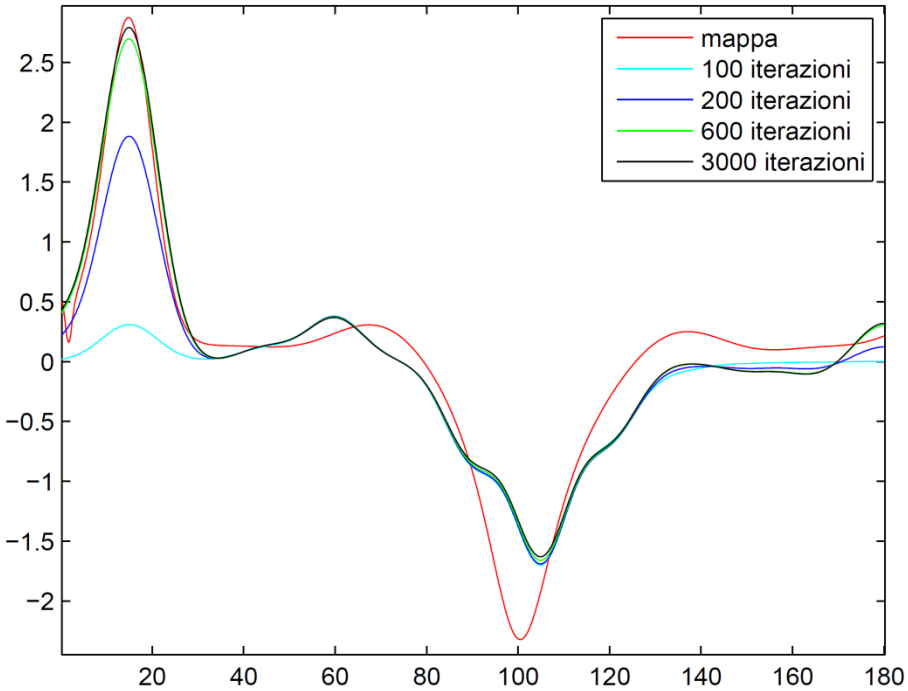




Distributed Mapping:Risultati 2

- Mappa disponibile presso l'agente 30 (su 61)





- Consensus dopo circa 500 iterazioni

Mapping: Conclusioni

- Newton-Raphson Distribuito
  - Algoritmo funzionante
  - Possibile miglioramento: ogni agente contribuisce solo ad aggiornare i coefficienti relativi a campane vicine
- Outliers
  - Funzioni costo iperboliche riducono l'influenza degli outliers

Università degli Studi di Padova
Ingegneria dell' Automazione

DEPARTMENT OF INFORMATION ENGINEERING
UNIVERSITY OF PADOVA

Consensus Newton-Raphson per l'ottimizzazione distribuita di funzioni convesse

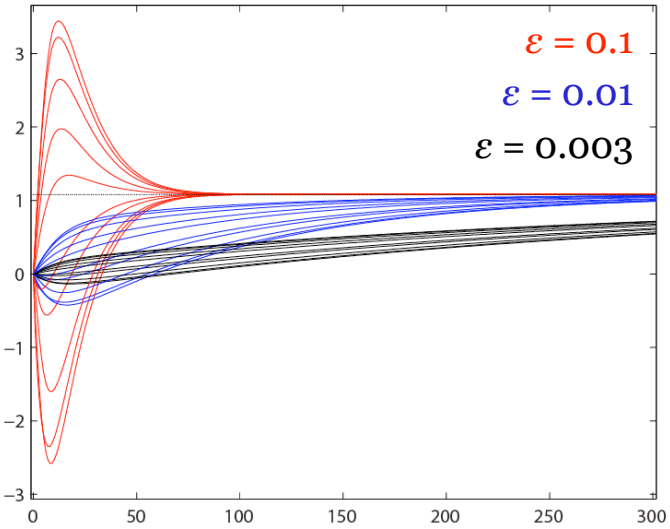
Tuning del fattore di scala temporale ε

Presentazione di *Andrea Bernardi.*

Gruppo: *Andrea Bernardi, Enrico Regolin e Giulio Veronesi.*

Ruolo del parametro ε

- Il parametro ε gioca un ruolo fondamentale per
  - la stabilità
  - la velocità di convergenza
- Teorema
  - Esiste ε^* tale che per ogni $\varepsilon < \varepsilon^*$ è garantita la stabilità esponenziale



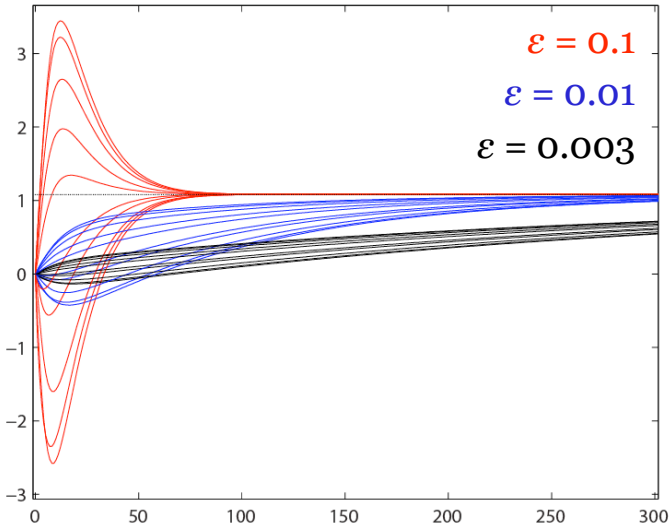
$$f_i(x) = c_i e^{a_i x} + d_i e^{-b_i x}, \quad i = 1, ..., N$$
$$a_i, b_i \sim \mathcal{U}[0, 0.2], \quad c_i, d_i \sim \mathcal{U}[0, 1]$$

[1] F. Zanella, D. Varagnolo, A. Cenedese, G. Pillonetto, L. Schenato, *Newton-Raphson consensus for distributed convex optimization"*, http://paduareserch.cab.unipd.it/, Tech Rep., 2011.

[2] H. K. Khalil, *"Nonlinear Systems (third edition)"*, Prentice Hall, 2002.

Autotuning centralizzato di ε

- Algoritmo adattativo
  - Retroazione dagli stati a ε
  - Distanza dal punto di minimo ignota



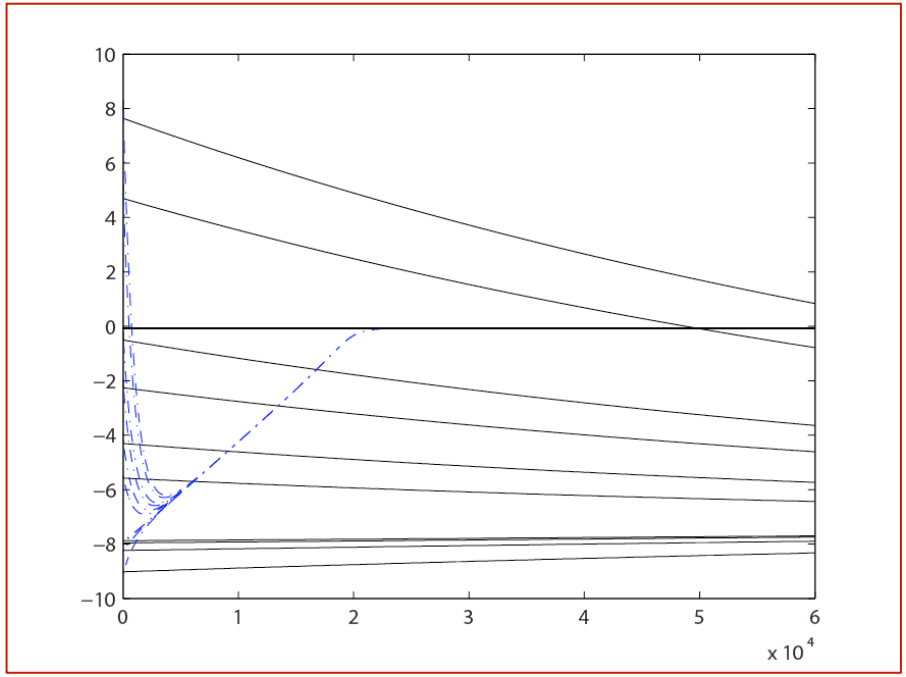
- Incrementi delle variabili di stato $\delta(k) = |x(k) - x(k-1)|$
  - Se crescono ⟶ divergenza
  - Se decrescono ⟶ convergenza

Algoritmo per il tuning di ε

1: $\boldsymbol{\delta}(k-1) = \left| \frac{\mathbf{y}(k-1)}{\mathbf{z}(k-1)} - \frac{\mathbf{y}(k-2)}{\mathbf{z}(k-2)} \right|$
2: $\boldsymbol{\delta}(k) = \left| \frac{\mathbf{y}(k)}{\mathbf{z}(k)} - \frac{\mathbf{y}(k-1)}{\mathbf{z}(k-1)} \right|$
3: **if** $\sum_{i=1}^{N} \delta_i(k-1) > \sum_{i=1}^{N} \delta_i(k)$ **then** ▷ aumenta ε
4: **if** $\varepsilon < 0.5$ **then**
5: $\varepsilon = 1.5\varepsilon$
6: **else**
7: $\varepsilon = \varepsilon + (1-\varepsilon)/2.5$
8: **end if**
9: **else** ▷ riduci ε
10: $\varepsilon = \varepsilon/2$
11: **end if**

- Ad ogni iterazione si valuta la media degli incrementi
- Evitare la divergenza è più importante che velocizzare l'algoritmo
- ε deve appartenere all'intervallo [0,1]

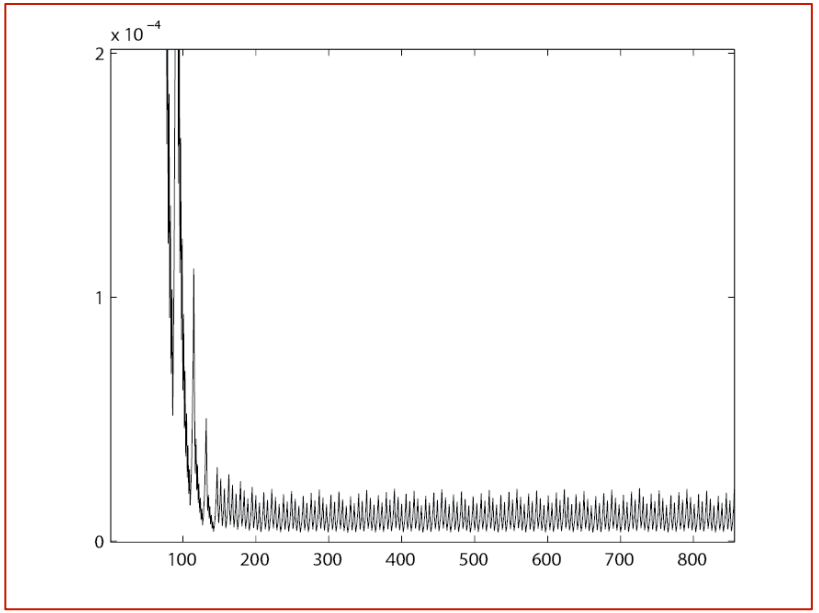
Simulazione



$$f_i(x) = c_i e^{a_i x} + d_i e^{-b_i x}, \quad i = 1, ..., N$$

$$a_i, b_i \sim \mathcal{U}[0,3], \quad c_i, d_i \sim \mathcal{U}[0,1]$$

- Si notano troppe oscillazioni su ε
- Campionare meno frequentemente
- Che valore assegnare al periodo τ ?

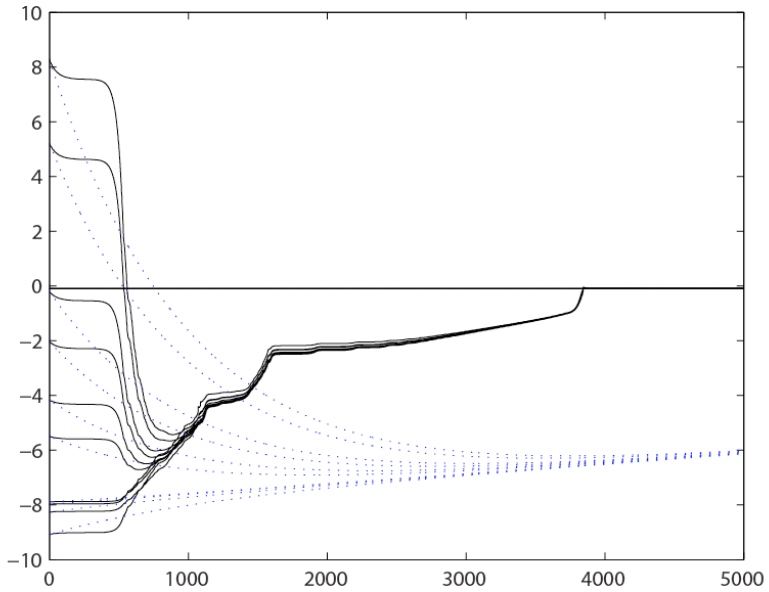


Introduzione del periodo τ

1: $\boldsymbol{\delta}(h-\tau) = \left| \frac{\mathbf{y}(h-\tau)}{\mathbf{z}(h-\tau)} - \frac{\mathbf{y}(h-\tau-\bar{\tau})}{\mathbf{z}(h-\tau-\bar{\tau})} \right|$
2: $\boldsymbol{\delta}(h) = \left| \frac{\mathbf{y}(h)}{\mathbf{z}(h)} - \frac{\mathbf{y}(h-\tau)}{\mathbf{z}(h-\tau)} \right|$
3: **if** $\sum_{i=1}^{N} \delta_i(h-\tau) > \sum_{i=1}^{N} \delta_i(h)$ **then** ▷ aumenta ε
4: **if** $\varepsilon < 0.5$ **then**
5: $\varepsilon = 1.5\varepsilon$
6: **else**
7: $\varepsilon = \varepsilon + (1-\varepsilon)/2.5$
8: **end if**
9: $\tau = \tau/2$ ▷ e riduci il periodo
10: **if** $\tau < 1$ **then**
11: $\tau = 1$
12: **end if**
13: **else** ▷ riduci ε
14: $\varepsilon = \varepsilon/2$
15: $\tau = \tau * 2$ ▷ e aumenta il periodo
16: **end if**

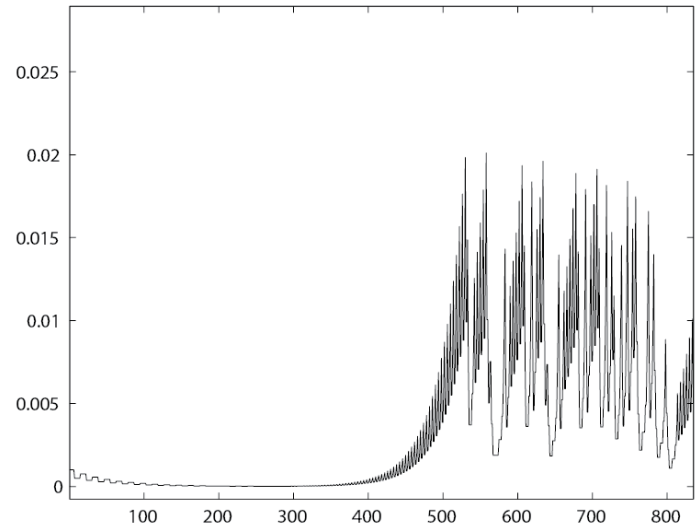
- Si dimezza τ in fase di convergenza
- Si raddoppia τ in fase di divergenza

Simulazione



- L'algoritmo cerca di raggiungere un determinato valore, senza riuscirci

- Buone prestazioni in termini di iterazioni totali
- Le oscillazioni di ε sono diminuite, ma ancora presenti

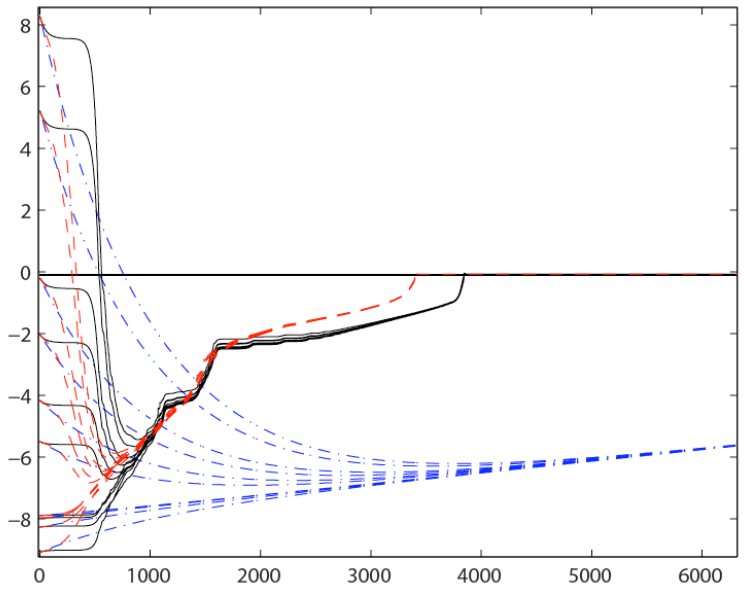


Perfezionamento dell'algoritmo

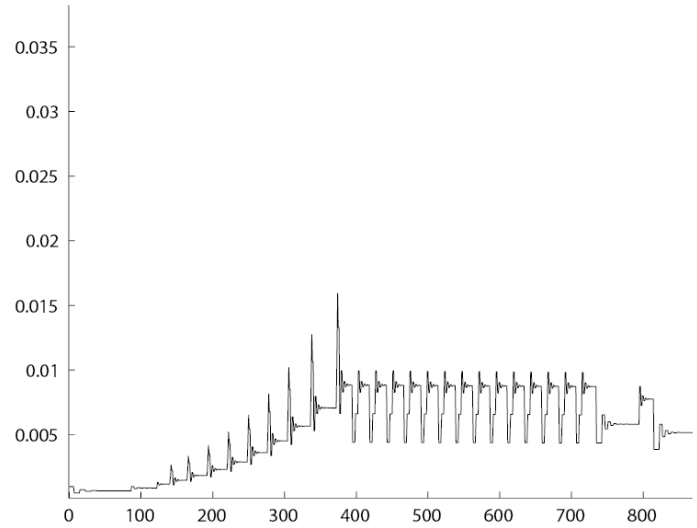
1: $\boldsymbol{\delta}(h-\tau) = \left| \frac{\mathbf{y}(h-\tau)}{\mathbf{z}(h-\tau)} - \frac{\mathbf{y}(h-\tau-\bar{\tau})}{\mathbf{z}(h-\tau-\bar{\tau})} \right|$
2: $\boldsymbol{\delta}(h) = \left| \frac{\mathbf{y}(h)}{\mathbf{z}(h)} - \frac{\mathbf{y}(h-\tau)}{\mathbf{z}(h-\tau)} \right|$
3: **if** $\sum_{i=1}^{N} \delta_i(h-\tau) > \sum_{i=1}^{N} \delta_i(h)$ **then**
4: **if** ε è stato aumentato precedentemente **then**
5: $\bar{\varepsilon} = \varepsilon$ ▷ continua ad aumentarlo
6: **if** $\varepsilon < 0.5$ **then**
7: $\varepsilon = 1.5\varepsilon$
8: **else**
9: $\varepsilon = \varepsilon + (1-\varepsilon)/2.5$
10: **end if**
11: **else** ▷ se in precedenza è diminuito,
12: $\Delta_\varepsilon = |\bar{\varepsilon} - \varepsilon|$
13: $\bar{\varepsilon} = \varepsilon$
14: $\varepsilon = \varepsilon + \Delta_\varepsilon/2$ ▷ riportalo a un valore intermedio
15: **end if**
16: $\tau = \tau/2$
17: **if** $\tau < 1$ **then**
18: $\tau = 1$
19: **end if**
20: **else**
21: **if** ε è stato aumentato precedentemente **then**
22: $\Delta_\varepsilon = |\bar{\varepsilon} - \varepsilon|$
23: $\bar{\varepsilon} = \varepsilon$
24: $\varepsilon = \varepsilon + \Delta_\varepsilon/2$ ▷ riducilo ad un valore intermedio
25: **else** ▷ se in precedenza è diminuito,
26: $\varepsilon = \varepsilon/2$ ▷ continuna a diminuirlo
27: $\bar{\varepsilon} = \varepsilon$
28: **end if**
29: $\tau = \tau * 2$
30: **end if**

- Procedimento dicotomico di avvicinamento asintotico al valore di ε

Simulazione



- Ulteriore riduzione delle iterazioni totali
- Migliore andamento del parametro ε



- Ottenuto un algoritmo indipendente dal valore iniziale di ε, più veloce di quello con ε costante

Versione distribuita

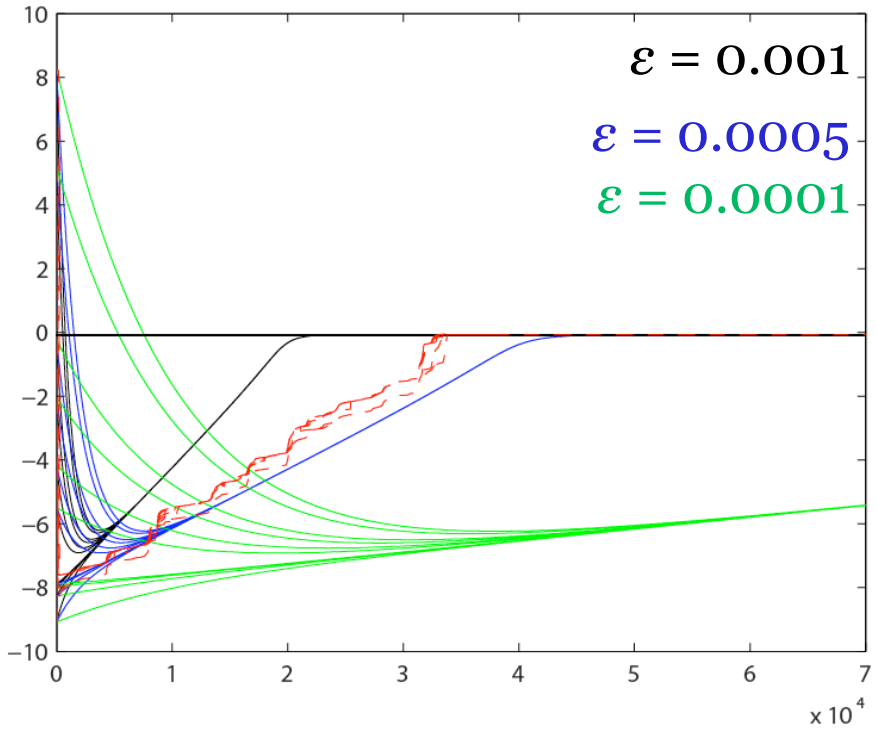
- Un diverso parametro ε per ogni agente
  - Vantaggio: si ottiene un algoritmo distribuito
  - Svantaggio: ε distinti creano problemi perché ogni agente ha una velocità di convergenza diversa
- Applicazione alla versione con matrice di *consensus gossip*
  - Si ottiene un algoritmo asincrono

Algoritmo distribuito

1: $\delta_i(h-\tau_i) = \left| \frac{yi(h-\tau_i)}{zi(h-\tau_i)} - \frac{yi(h-\tau_i-\bar{\tau}_i)}{zi(h-\tau_i-\bar{\tau}_i)} \right|$
2: $\delta_i(h) = \left| \frac{yi(h)}{zi(h)} - \frac{yi(h-\tau_i)}{zi(h-\tau_i)} \right|$
3: **if** $\delta_i(h-\tau) > \delta_i(h)$ **then**
4: **if** ε_i è stato aumentato precedentemente **then**
5: $\bar{\varepsilon}_i = \varepsilon_i$
6: **if** $\varepsilon_i < 0.5$ **then**
7: $\varepsilon_i = 1.5\varepsilon_i$
8: **else**
9: $\varepsilon_i = \varepsilon_i + (1-\varepsilon_i)/2.5$
10: **end if**
11: **else**
12: $\Delta_{\varepsilon_i} = |\bar{\varepsilon}_i - \varepsilon_i|$
13: $\bar{\varepsilon}_i = \varepsilon_i$
14: $\varepsilon_i = \varepsilon_i + \Delta_{\varepsilon_i}/2$
15: **end if**
16: $\tau_i = \tau_i/2$
17: **if** $\tau_i < 1$ **then**
18: $\tau_i = 1$
19: **end if**
20: **else**
20: **else**
21: **if** ε è stato aumentato precedentemente **then**
22: $\Delta_{\varepsilon_i} = |\bar{\varepsilon}_i - \varepsilon_i|$
23: $\bar{\varepsilon}_i = \varepsilon_i$
24: $\varepsilon_i = \varepsilon_i + \Delta_{\varepsilon_i}/2$
25: **else**
26: $\varepsilon_i = \varepsilon_i/2$
27: $\bar{\varepsilon}_i = \varepsilon_i$
28: **end if**
29: $\tau_i = \tau_i * 2$
30: **if** $\tau_i > \tau_{max}$ **then** ▷ non superare il periodo massimo
31: $\tau_i = \tau_{max}$
32: **end if**
33: **end if**

- Ad ogni iterazione si valuta l'incremento di stato del singolo agente
- Per evitare che i periodi τ crescano troppo si impone un τ_{max}

Simulazione



- Confrontando con i risultati che si ricavano tenedo fisso il parametro ε, si nota che le prestazioni non sono le migliori possibili, ma sono comunque buone

Conclusioni

- Abbiamo ottenuto un algoritmo che
  - permette di raggiungere buone prestazioni
  - è indipendente dal valore di partenza di ε
  - è in versione distribuita
  - è applicabile al caso asincrono
- Eventuale lavoro futuro
  - Affinamento del tuning del periodo τ