Politecnico di Torino

Corso di Laurea
in Ingegneria Matematica

Tesi di Laurea Magistrale

Modellizzazione della distribuzione della popolazione tra città su reti spaziali mediante la teoria cinetica dei sistemi multiagente

Relatore
prof. Andrea Tosin

Candidato
Valerio Taralli

firma del relatore

.....

firma del candidato

.....

Anno Accademico 2025-2026

Ai miei genitori,
Elisabetta e *Marco*

SOMMARIO

La corrente tesi studia la distribuzione della popolazione mediante la teoria cinetica dei sistemi multiagente su reti spaziali, interpretando le città come agenti/vertici e rappresentando le connessioni interurbane come lati. Si propongono varie regole d'emigrazione per regolare le interazioni tra agenti, ricavando due principali risultati: l'adattamento lognormale bimodale della distribuzione della popolazione tra le città e l'adattamento di Pareto della coda. Nel complesso il modello restituisce dei risultati coerenti colla distribuzione reale dedotta dai dati ISTAT della Sardegna; perdipiú l'adattamento lognormale bimodale segue meglio la coda della distribuzione rispetto a quello di Pareto, sebbene nella prima metà coincidano.

ABSTRACT

The current thesis investigates population distribution through the kinetic theory of multi-agent systems on spatial networks, interpreting cities as agents/vertices and representing interurban links as edges. Various emigration rules have been proposed to regulate the interactions between agents, yielding two primary results: the bimodal log-normal fit of the total population distribution among cities and the Pareto fit of the tail. Overall, the model provides results consistent with the actual distribution derived from ISTAT data of Sardinia; furthermore, the bimodal log-normal fit follows the distribution tail more accurately than the Pareto one, although the two coincide in the first half.

INDICE

1 INTRODUZIONE

Lo studio della distribuzione della popolazione tra le città è stato un fenomeno già analizzato [sporadicamente] in passato. Il primo fu Auerbach [1] che all'inizio del XX secolo notò una caratteristica formalizzata successivamente da Zipf[1] [20] quasi cinquant'anni dopo, seppure inizialmente in un contesto linguistico; difatti la legge di Zipf è una legge empirica di forma

$$f \propto \frac{1}{r} \quad \text{ove} \begin{cases} f \text{ è la frequenza della parola,} \\ r \text{ è il suo rango nella classifica;} \end{cases} \tag{1.1}$$

per dare un esempio concreto, se la decima parola piú frequente compare 2000 volte allora esiste una costante C tale che $2000 \approx C/10$. La stessa legge descritta in (1.1) si può ritrovare in molt'altri contesti, tra i quali figura la distribuzione della popolazione tra le città se in luogo di f si scrive la popolazione s [20, Fig. 9-2 p. 375]; tuttavia ciò è vero solo qualora $s \gg 1$, ossia nella coda della distribuzione, contrariamente al contesto linguistico originale di Zipf ove (1.1) vale sempre.

Piú in generale la legge [empirica] di Zipf può essere descritta da una distribuzione di Pareto: sia $S \in \mathbb{R}_+$ la variabile aleatoria legata alla taglia delle città e sia $f_S(s)$ la sua funzione di densità di probabilità[2], la quale permette d'interpretare $f_S(s)ds$ come il numero di città con $s \in [s, s+dt]$; allora si dice che S segue una distribuzione di Pareto nella coda se soddisfà

$$\mathcal{R}(s) \equiv \int_s^{+\infty} f_S(t)dt \approx \frac{c}{s^\beta} \quad \text{se } s \gg 1, \tag{1.2}$$

per una certa costante $c \in \mathbb{R}_+$ e con $\mathcal{R}(s)$ uguale alla funzione di ripartizione complementare di S: essa rappresenta il numero di città con popolazione maggiore o uguale a s e, qualora l'indice di Pareto β fosse unitario, è analoga al rango r nella (1.1).

Piú recentemente [8, 9] si è scoperto che al di fuori della coda la $f_S(s)$ è ben fittata dalla distribuzione lognormale con densità di probabilità

$$f_X(x) = \frac{1}{\ln(10)} \frac{1}{x\sigma\sqrt{2\pi}} \exp\left(-\frac{(\log x - \mu)^2}{2\sigma^2}\right), \tag{1.3}$$

ove $X \sim \text{Lognormale}(\mu, \sigma)$; come suggerisce il nome, la peculiarità di tale distribuzione è che vale $Y = \log X \sim \mathcal{N}(\mu, \sigma)$ (Fig. 1.1(c)) in cui $\mathcal{N}(\mu, \sigma)$ indica la distribuzione gaussiana di media μ e deviazione σ.

Perdipiú da Gualandi *et al.* [9, 10] ci si è poi resi conto che l'intera distribuzione è ben adattata da una lognormale bimodale (qui denotata anche come bilognormale), vale a dire una combinazione convessa della (1.3)

$$f_S(s) \approx f_X(s) = \xi f_{S_1}(s) + (1-\xi) f_{S_2}(s), \tag{1.4}$$

1 In realtà, come Zipf stesso ammette [20, p. 546], non fu il primo a scoprire tale legge; in ogni caso la popolarizzò a tal punto che oggi è conosciuta col suo nome.

2 Affinché $f_S(s)$ esista bisogna rigorosamente anche supporre che S sia assolutamente continua, ma qui non è necessario entrare nei dettagli per i quali si rimanda ai §§ 3.1 e 3.3.

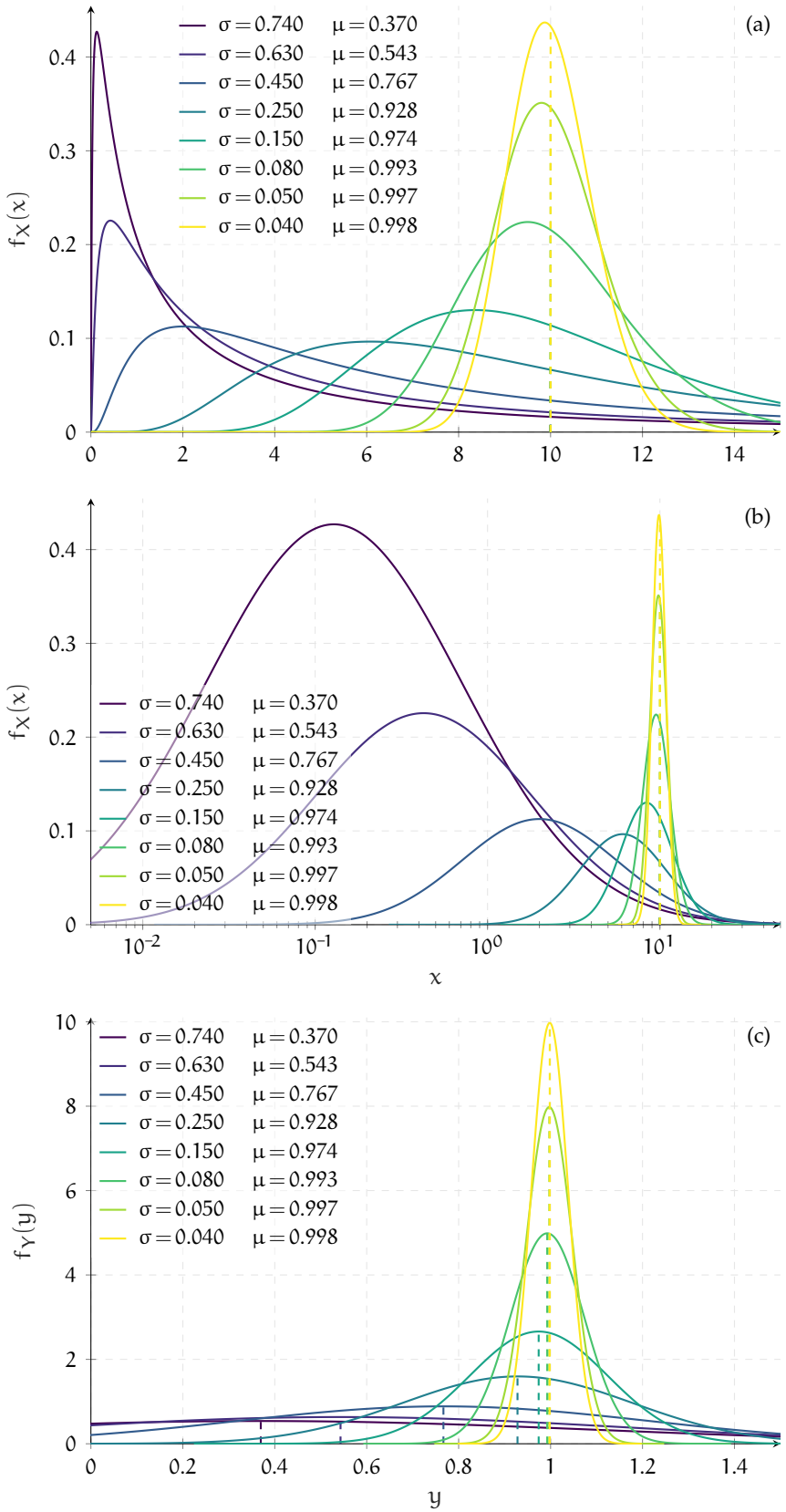

Figura 1.1: Funzione $X \sim \text{Lognormale}(\mu, \sigma)$ al variare dei parametri, ma con uguale valore atteso $\mathbb{E}[X] = 10^{\mu + \sigma^2 \ln(10)/2} = 10$, in scala lineare (a) e logaritmica (b); invece in (c) sono riportate le stesse distribuzioni in spazio logaritmico $y = \log(x)$. Le linee tratteggiate sono i valori attesi di ciascuna distribuzione.

ove $\xi \in (0,1)$, $S_1, S_2 \sim \text{Lognormale}(\mu,\sigma)$ e $X \sim \text{BiLognormale}(\xi,\mu_1,\sigma_1,\mu_2,\sigma_2)$.

Pertanto l'obbiettivo di questa tesi è il seguente: si vuole simulare attraverso la Teoria Cinetica dei Sistemi MultiAgente (TCSMA) le interazioni tra città su un grafo spaziale, tentando di riprodurre una distribuzione della popolazione con caratteristiche analoghe a quelle realmente misurate (principalmente corpo lognormale e coda di Pareto); una volta raggiunto questo scopo, s'interpreta il fenomeno della migrazione tramite le leggi d'emigrazione proposte, che sono alla base delle interazioni tra città.

Inoltre, si sottolinea che in questo lavoro le città verranno considerate come agenti caratterizzati dalla loro popolazione, le quali interagiscono attraverso la struttura sottostante di un grafo [spaziale]; tale descrizione, salvo la rete, non è affatto dissimile a quella classica delle molecole di un gas caratterizzate dalla loro velocità e posizione. Difatti in letteratura, perlomeno quella a conoscenza dell'autore, non esistono articoli che trattano contemporaneamente la TCSMA, la distribuzione della popolazione tra città e i grafi, nonostante la rappresentazione di queste su una rete sia del tutto naturale; piú nel dettaglio

- alcuni o applicano la TCSMA alle città senza considerare una naturale topologia sottostante [10][3] o, in senso opposto, considerano la struttura da grafo senza applicare la TCSMA [4];
- altri vedono i nodi piú come luoghi ove vivono e interagiscono gli agenti, creando cosí un modello descritto da un sistema di equazioni di Boltzmann elevate e accoppiate [15];
- infine, l'articolo che piú si avvicina all'obbiettivo di questa tesi applica, tuttavia, la sua teoria nel contesto delle reti sociali [17].

Questa lacuna è probabilmente attribuibile al fatto che gli sviluppi della teoria dei grafi applicata alla TCSMA sono stati piuttosto recenti e principalmente concentrati sulla prospettiva nodo-luogo anziché nodo-agente. Dunque l'aspetto piú innovativo di questo lavoro è mostrare che la seconda prospettiva è altrettanto valida quanto la prima, nonostante sia piú comune; per il resto questa tesi dev'essere considerata come un'esplorazione formale applicativa con pochi approfondimenti analitici.

Nel complesso lo scritto è cosí suddiviso: nel secondo capitolo si affronteranno brevemente e superficialmente alcuni concetti della teoria dei grafi, soprattutto quelli pertinenti alla topologia interurbana; quindi nel terzo la TCSMA è approfondita prima da un punto di vista classico, e poi mediata dai grafi sia esattamente che approssimatamente; infine negli ultimi due s'illustreranno i principali risultati concludendo con delle note finali e potenziali sviluppi futuri. Per il lettore interessato è anche presente un'appendice ove si spiega a grandi linee il codice di Python dell'implementazione.

3 In particolare [10], nonostante sia un articolo molto interessante e che giunge a conclusioni altrettanto importanti, astrae eccessivamente le interazioni tra città oltre a non definire con sufficiente chiarezza che cosa s'intende per *dintorni*.

2 NOTE DI TEORIA DEI GRAFI

In questo capitolo sono prima introdotti alcuni oggetti di base della teoria delle reti, quindi si discute la natura della rete d'interesse, infine si descriveranno molto brevemente i dati usati.

2.1 DEFINIZIONI MISCELLANEE

Innanzitutto s'inizia dando la definizione di rete:

Definizione 2.1 (Grafo)**.** Un grafo è un insieme $\mathcal{G} = (\mathcal{I}, \mathcal{E})$ formato dalla coppia dell'insieme degl'indici dei nodi $\mathcal{I}$ e dell'insieme dei lati $\mathcal{E}$, vale a dire di coppie d'indici $\mathcal{I}$: due nodi $i, j \in \mathcal{I}$ sono connessi sse $(i, j) \in \mathcal{E}$.

Dall'insieme $\mathcal{E}$ si può poi specializzare il concetto di grafo:

Definizione 2.2 (Grafo [in]diretto)**.** Un grafo è indiretto sse

$$(i, j) \in \mathcal{E} \iff (j, i) \in \mathcal{E},$$

e la direzione è trascurabile; altrimenti è detto diretto.

La trascurabilità della direzione sarà discussa poco dopo nel § 2.2, anche se è facilmente intuibile dalla Fig. 2.1.

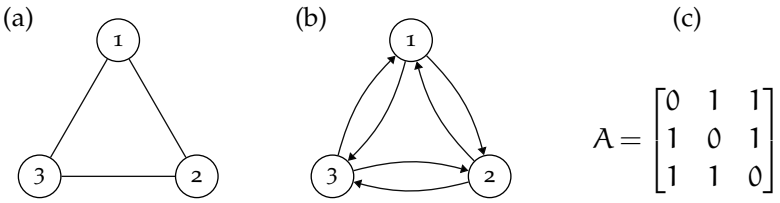

Figura 2.1: Esempio di un grafo indiretto (a), della sua equivalente forma diretta (b) e della loro [identica] matrice d'adiacenza (c).

Definizione 2.3 (Matrice d'adiacenza)**.** Sia $N \equiv |\mathcal{I}| \in \mathbb{N}_+$ la cardinalità dell'insieme degl'indici[1], ossia il numero di vertici totali, si definisce la matrice d'adiacenza $\mathbf{M} \in \mathbb{R}^{N \times N}$ pesata come:

$$m_{i,j} \equiv \begin{cases} q_{i,j} & \text{se } (i, j) \in \mathcal{E}, \\ 0 & \text{altrimenti,} \end{cases} \tag{2.1}$$

ove $q_{i,j} \in \mathbb{R}$ è il peso associato al lato (i, j); nel caso in cui $q_{i,j} \equiv 1 \ \forall i, j \in \mathcal{I}$ si ha la matrice d'adiacenza unitaria $\mathbf{A} \in \mathbb{R}^{N \times N}$:

$$a_{i,j} \equiv \begin{cases} 1 & \text{se } (i, j) \in \mathcal{E}, \\ 0 & \text{altrimenti.} \end{cases} \tag{2.2}$$

1 Per $\mathbb{N}_+$ si veda la Def. 3.1 poco dopo.

Si osserva che, se non altrimenti detto, scrivendo solo *matrice d'adiacenza* si sottintende sempre quella unitaria.

Definizione 2.4 (Matrice trasposta)**.** Data $\mathbf{M} \in \mathbb{R}^{N\times N}$, con $N \in \mathbb{N}_+$, s'indica con $\mathbf{M}^\top$ la matrice trasposta cosí definita:

$$m^\top_{i,j} = m_{j,i}, \quad \forall i,j \in \{1,2,\ldots,N\}.$$

Definizione 2.5 (Matrice simmetrica)**.** Data $\mathbf{M} \in \mathbb{R}^{N\times N}$, con $N \in \mathbb{N}_+$, essa è simmetrica sse

$$m_{i,j} = m_{j,i} = m^\top_{i,j} \iff \mathbf{M} = \mathbf{M}^\top. \tag{2.3}$$

Osservazione 2.1. Nei grafi indiretti sia $\mathbf{M}$ che $\mathbf{A}$ sono simmetriche mentre per quelli diretti non è detto lo siano.

Definizione 2.6 (Gradi e Forze)**.** In un grafo diretto e pesato si definiscono la forza entrante e uscente come rispettivamente le quantità

$$w^-_i \equiv \sum_{j=1}^{|\mathcal{I}|} m_{j,i} \quad \text{e} \quad w^+_i \equiv \sum_{j=1}^{|\mathcal{I}|} m_{i,j}, \quad \forall i \in \mathcal{I}, \tag{2.4}$$

le quali se il grafo è simmetrico coincidono: $\mathbf{w} = \mathbf{w}^- = \mathbf{w}^+$. Nel caso unitario $\mathbf{M} \equiv \mathbf{A}$ si è soliti denominare le forze come gradi uscenti ed entranti:

$$k^-_i \equiv \sum_{j=1}^{|\mathcal{I}|} a_{j,i} \quad \text{e} \quad k^+_i \equiv \sum_{j=1}^{|\mathcal{I}|} a_{i,j}, \quad \forall i \in \mathcal{I}, \tag{2.5}$$

che, come prima, in una rete simmetrica coincidono: $\mathbf{k} = \mathbf{k}^- = \mathbf{k}^+$.

2.2 RETI E CITTÀ

Dopo aver illustrato un po' di teoria sui grafi sorge successivamente il problema di come rappresentare con essi la moltitudine di collegamenti urbani; in effetti, vi sono molti modi di rappresentare le città mediante i grafi: i primi si pongono a livello *intraurbano*, o considerando le strade come lati e le loro intersezioni come nodi [3, § 3.1.3.1 p. 17], o rappresentando le reti di trasporto (tram, bus e metro) [3, § 3.2.1 p. 22]; altri si pongono piú propriamente a livello *interurbano* prendono singolarmente in considerazione varie reti dei trasporti (ferroviario [3, § 3.1.3.2 p. 17], navale [3, § 3.1.4 p. 19], aereo [3, § 3.1.2 p. 13], ecc.).

Tuttavia, da quanto detto nel Cap. 1, è chiaro che per predire la distribuzione della popolazione tra città valgono le seguenti osservazioni:

1. ogni rappresentazione intraurbana va scartata perché è troppo fine, oltre a considerare movimenti limitatamente a una sola città;
2. d'altra parte tutte quelle interurbane vanno considerate contemporaneamente e non singolarmente siccome ciascun individuo può scegliere diversi trasporti per muoversi.

Ecco perché la corretta rete da considerare è quella legata ai movimenti pendolari [3, § 3.1.3.3 p. 18; 6] tra città che mostrano olisticamente tutt'i possibili collegamenti tra le città a prescindere dal trasporto scelto; inoltre essa mostra collegamenti realistici associati a movimenti quotidiani anziché straordinari (vacanze, visite mediche, ecc.).

Una volta fissata la rappresentazione è necessario comprendere il tipo di grafo con cui si ha a che fare. Eppure anche qui la risposta è immediata dopo due ulteriori note:

1. una città può interagire con un'altra senza che quest'ultima interagisca colla prima: è necessario considerare il senso di direzione;
2. ammesso che una città possa interagire con un'altra, allora è sempre possibile l'opposto: le potenziali interazioni sono simmetriche.

Dunque il grafo in questione è diretto ma con struttura indiretta (Def. 2.2):

Ipotesi 2.1 (**A** simmetrica)**.** Il grafo $\mathcal{G}$ è diretto e simmetrico.

Inoltre in questa trattazione vale la seguente fondamentale ipotesi:

Ipotesi 2.2. Il grafo $\mathcal{G}$ è statico: $\mathcal{I}$ e $\mathcal{E}$ sono costanti nel tempo.

In altre parole si è solo interessati a prevedere come la popolazione si distribuisce rispetto a una topologia prestabilita *a priori*; naturalmente si tratta di una forte semplificazione siccome nella realtà la topologia, soprattutto le connessioni interurbane in $\mathcal{E}$, sono chiaramente coevoluta assieme allo sviluppo delle città stesse.

Infine si osserva che questo tipo di grafo è spaziale [3, 6], ovvero i suoi nodi occupano un punto nello spazio euclideo, seppure, al momento, sia una caratteristica alquanto formale; tuttavia serve tenerlo a mente quando s'interpretano alcune regole d'emigrazione nel Cap. 4. Perdipiú, contrariamente a quanto si possa pensare di primo acchito, tale grafo non è a invarianza di scala [2] proprio per la sua natura spaziale [3, 5]; ciò non esclude l'esistenza di nodi piú centrali di altri, ma solo che il massimo grado di un nodo è limitato superiormente a causa di limiti fisici.

2.3 CENNI SUI DATI

La matrice di pendolarismo costruita è quella fornita dall'ISTAT del 1991 [14] seguendo l'esempio di [6]. I dati sono di fatto un *file* di testo formato da una serie di righe di 29 numeri il cui significato è spiegato dalla Tab. 2.1.

I dati considerano tutt'i comuni italiani nel 1991 (in totale 8100), quindi è necessario estrapolare da essi le matrici di pendolarismo delle singole regioni e parallelamente quella complessiva dell'Italia. In piú, si possono definire due tipi di matrici d'adiacenza:

1. quella unitaria in cui un lato esiste se almeno un pendolare (cioè la riga ⊛ nella Tab. 2.1) si muove tra i due comuni e
2. quella pesata che è analoga alla precedente ma con peso $q_{i,j}$ uguale alla somma totale dei pendolari tra due comuni.

2 si v. il documento `trapen91.txt` per maggiori informazioni.

Tabella 2.1: Formattazione di una sola riga nei dati ISTAT[2]; le linee barrate corrispondono a dati trascurati.

	Dato	Col. iniziale	Lunghezza
	Provincia di partenza	1	3
	Comune di partenza	4	3
	~~Sesso~~	~~7~~	~~1~~
	~~Mezzo di trasporto~~	~~8~~	~~1~~
	~~Cond.professionale~~	~~9~~	~~1~~
	~~Orario di uscita~~	~~10~~	~~1~~
	~~Tempo di percorrenza~~	~~11~~	~~1~~
	Provincia di arrivo	12	3
	Comune di arrivo	15	3
⊛	Numero di persone	18	12

Si notano tuttavia due principali difetti dei dati contenuti nel documento `Pen_91It.txt`:

1. in alcune linee come comune di destinazione è segnato il codice «022008» che però non è elencato nel documento `elencom91.xls` che riassume tutti gli 8100 comuni italiani nel 1991;
2. in molte linee come comune di destinazione sono segnati dei codici incompleti contenenti solo l'ultima metà (codice comune) ma non la prima (codice provincia), e questi sono in totale 8: «215», «229», «241», «216», «203», «224», «236» e «246».

In entrambi i casi i relativi dati [a quelli che sono molto probabilmente refusi] sono stati trascurati nella corrente tesi. Infatti il primo errore è già di per sé molto raro, mentre il secondo è eccessivamente ambiguo da risolvere; per esempio, si prenda la linea successiva relativa al codice «215»:

`00200714212215_____________1,`

ove _ indica uno spazio, per la quale, tralasciando il fatto che vi sono in totale 6 città italiane collo stesso codice comunale, se ci si limita alla regione del Piemonte (vale a dire la stessa del comune di partenza «002007») esistono in realtà ben due città condividenti lo stesso codice: «001215» e «004215», eppure entrambe non hanno lo stesso codice provinciale («002») del comune di partenza. Da questo piccolo estratto, quindi, non si può nemmeno ipotizzare che il codice provinciale sia lo stesso tra il comune di partenza e d'arrivo; è allora chiaro che sia impossibile scegliere la prima metà per completare il codice del comune di destinazione.

Osservazione 2.2. A dire il vero si potrebbe scegliere, tra i comuni con uguale codice comunale, quello che minimizza la distanza euleriana, visto che l'ISTAT [13] rende disponibili le coordinate dei punti rappresentativi di ogni comune. In ogni caso il numero d'eccezioni è cosí esiguo da rendere una tale "correzione" del tutto innecessaria[3]. Il tutto senza considerare che la scelta piú saggia è banalmente quella di prendere dati piú recenti[4].

4 I risultati del Cap. 4 mostrano anche che è irrilevante.
4 Non si è fatto per riprodurre [6], com'è discusso súbito dopo.

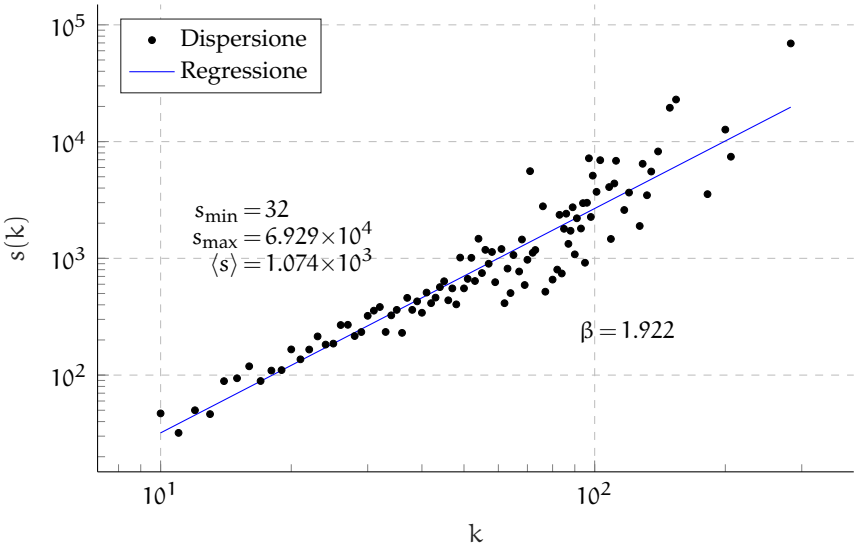

Figura 2.2: Forza contro grado della Sardegna.

Infine nelle Figg. 2.2 e 2.3 sono stati anche riprodotti i risultati di [6] anche se con alcune differenze:

1. il coefficiente d'aggregazione medio $\langle C(k) \rangle = 0.453$ è quasi il doppio di quello riportato da [6, p. 911] (0.26);
2. la Fig. 2.3(c) rispetto alla [6, Fig. 3b] presenta sia una forma leggermente diversa, seppure la tendenza a mezza luna sia la stessa, che limiti superiori e inferiori piú estesi;
3. il coefficiente d'aggregazione pesato nella Fig. 2.3(g) ha chiaramente una tendenza decrescente, anche se poco ripida per cui si può considerare comunque «circa costante» come afferma [6];
4. i primi due pesi piú elevati, riportati nella Tab. 2.2, non corrispondono, ma è probabilmente un refuso da parte di [6] poiché i collegamenti «Cagliari-Sassari» e «Sassari-Olbia» sono molto distanti geograficamente[5], per cui è ragionevole che i flussi siano bassi.

Queste discrepanze sono trascurabili perché lievi e, per quanto concerne questo elaborato, non significative. Inoltre l'unico risultato di rilievo, usato nel § 4.3.5, è la correlazione positiva tra la forza e il grado nella Fig. 2.2.

Tabella 2.2: Confronto con [6] dei pesi maggiori.

(a) Pesi maggiori correnti.

Connessione	Peso
Cagliari-Quartu SE	14709
Cagliari-Selargius	7995
Assemini-Selargius	4418
Porto Torres-Sassari	4149
Cagliari-Capoterra	3865

(b) Pesi maggiori di [6]

Connessione	Peso
Cagliari-Sassari	13953
Sassari-Olbia	7246
Cagliari-Assemini	4226
Porto Torres-Sassari	3993
Cagliari-Capoterra	3731

5 Il primo equivale a percorrere l'intera estensione dell'isola ogni giorno.

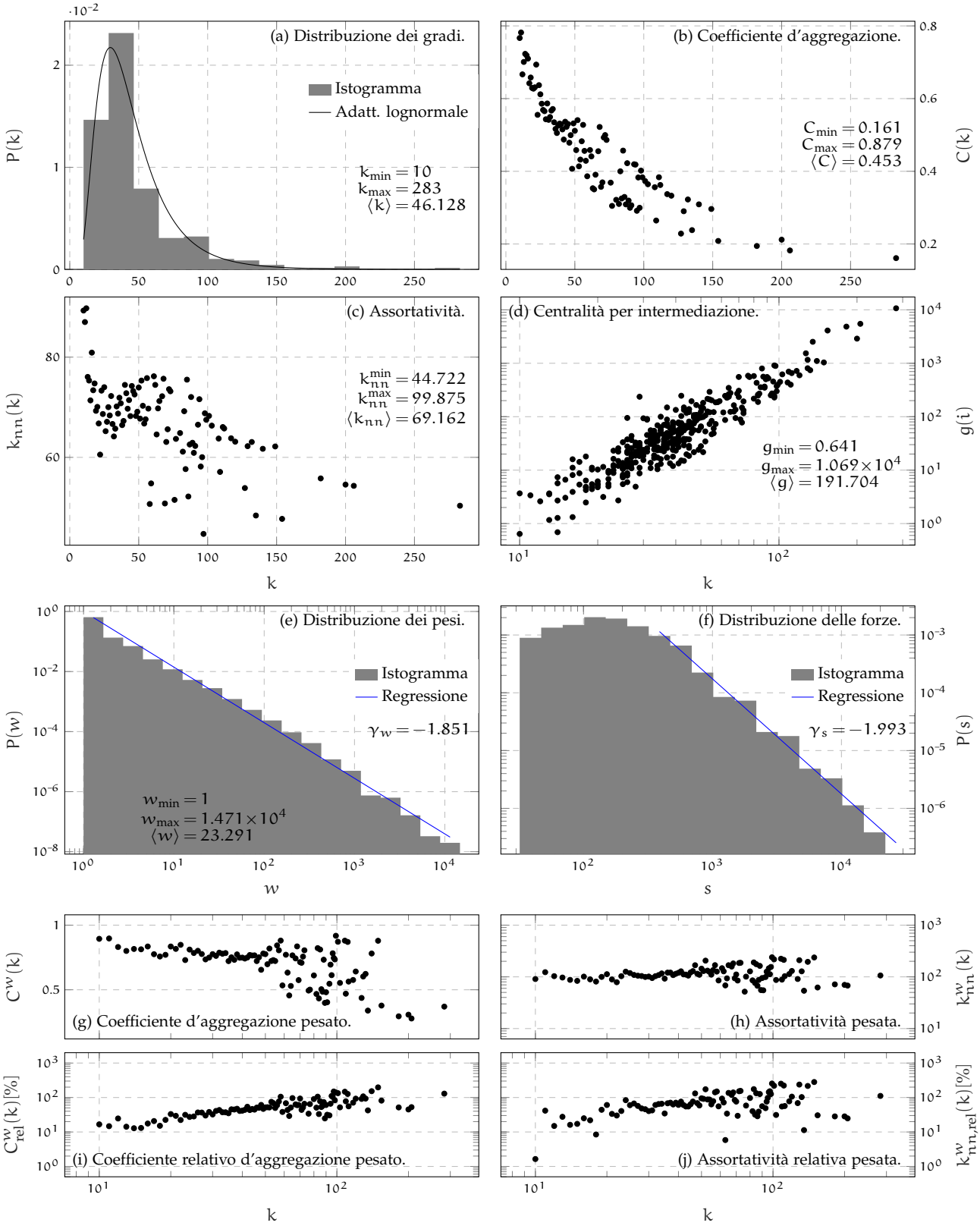

Figura 2.3: Riproduzione dei principali grafici di [6] [a cui si rimanda per maggiori dettagli sui vari coefficienti] relativi alla topologia della rete sarda del pendolarismo.

3 TEORIA CINETICA DEI SISTEMI MULTIAGENTE

In questo capitolo è discussa la TCSMA necessaria per ottenere e discutere i risultati nel Cap. 4. S'inizia prima dando delle nozioni generali di teoria della probabilità colle quali si deriva successivamente la celeberrima equazione di Boltzmann a partire da una descrizione stocastica delle particelle di un gas; quindi si passa a generalizzare tali risultati per mediare le interazioni tramite una struttura topologica sottostante gli agenti, sia in modo esatto che approssimato; si conclude analizzando le ipotesi semplificative che portano dalla presente teoria a quella classica di Boltzmann.

3.1 DEFINIZIONI PRELIMINARI DI PROBABILITÀ

Si annoverano ora alcuni concetti e risultati di teoria della probabilità avvisando, però, che per molti di essi è impossibile entrare eccessivamente nei dettagli; in ogni caso si rimanda a [7] per gl'interessati.

Definizione 3.1 (Semirette)**.** In tutto questo elaborato si fanno uso dei seguenti sottinsiemi dei numeri reali e naturali:

$$\mathbb{R}_+ \equiv \{x \in \mathbb{R} : \ x \geqslant 0\} \subset \mathbb{R},$$
$$\mathbb{N}_+ \equiv \{n \in \mathbb{N} : \ n \geqslant 1\} \subset \mathbb{N}.$$

Definizione 3.2 (Variabile aleatoria)**.** Una variabile aleatoria X è una funzione misurabile $X\colon \Omega \to \mathbb{R}$, il cui dominio è uno spazio astratto Ω di eventi elementari; quest'ultimo definisce a sua volta uno spazio di probabilità $(\Omega,\mathcal{F},\mathbb{P})$ composto da una σ-algebra $\mathcal{F}$ degli eventi e una misura $\mathbb{P}$ di massa unitaria.

Siccome Ω è uno spazio astratto poco trattabile, in questo scritto si definisce una variabile aleatoria semplicemente evidenziando l'appartenenza al suo codominio e sottintendendo il suo dominio: $X \in \mathbb{R}$.

Definizione 3.3 (Densità di probabilità di X)**.** Sia $X \in \mathbb{R}$ una variabile aleatoria assolutamente continua, allora la densità di probabilità $f_X\colon \mathbb{R} \to \mathbb{R}_+$ è una funzione misurabile che rappresenta la legge $\mathbb{P}_X$:

$$\mathbb{P}_X(A) = \mathbb{P}(X \in A) = \int_A f_X(x)\mathrm{d}x, \quad \forall A \subseteq \mathbb{R} \text{ misurabile.}$$

Osservazione 3.1. Nella definizione precedente si assume che la variabile aleatoria X sia assolutamente continua, in tal modo $f_X(x)$ è la sua derivata di Radon-Nikodym, ma la teoria sviluppata in questo paragrafo vale anche per misure piú astratte rispetto alle quali $f_X(x)\mathrm{d}x$ va inteso come $f_X(\mathrm{d}x)$.

Definizione 3.4 (Densità congiunta di probabilità di **X**)**.** Siano $n \in \mathbb{N}_+$ e

$$\mathbf{X} = (X_1, X_2, \ldots, X_n) \in \mathbb{R}^n$$

un vettore aleatorio assolutamente continuo, allora la densità congiunta di probabilità $f_{\mathbf{X}}\colon \mathbb{R}^n \to \mathbb{R}_+$ è una funzione misurabile che rappresenta la legge $\mathbb{P}_{\mathbf{X}}$:

$$\mathbb{P}_{\mathbf{X}}(A) = \mathbb{P}(\mathbf{X} \in A) = \int_A f_{\mathbf{X}}(\mathbf{x})\mathrm{d}\mathbf{x}, \quad \forall A \subseteq \mathbb{R}^n \text{ misurabile},$$

ove $\mathrm{d}\mathbf{x} \equiv \prod \mathrm{d}x_i = \mathrm{d}x_1\mathrm{d}x_2\cdots\mathrm{d}x_n$.

Definizione 3.5 (Densità marginale di probabilità di $\mathbf{X}$)**.** Siano $n \in \mathbb{N}_+$ e $\mathbf{X} \in \mathbb{R}^n$ un vettore aleatorio assolutamente continuo, allora la densità marginale di probabilità $f_{X_i}\colon \mathbb{R} \to \mathbb{R}_+$ è una funzione misurabile cosí definita:

$$f_{X_i}(x_i) = \int_{\mathbb{R}^{n-1}} f_{\mathbf{X}}(\mathbf{x})\mathrm{d}x_1\mathrm{d}x_2\cdots\mathrm{d}x_{i-1}\mathrm{d}x_{i+1}\cdots\mathrm{d}x_n, \quad \forall i \in \{1,2,\cdots,n\}; \tag{3.1}$$

essa è a tutti gli effetti la densità di probabilità della variabile aleatoria X_i secondo la Def. 3.3.

Osservazione 3.2. Per brevità, da qui in avanti s'indicheranno le precedenti densità sottintendendo che siano relative alla probabilità: f_X è dunque la densità di X, $f_{\mathbf{X}}$ la densità congiunta di $\mathbf{X}$ e f_{X_i} la densità marginale di X_i.

Proprietà 3.1 (Densità congiunta di variabili aleatorie indipendenti)**.** Siano $n \in \mathbb{N}_+$ e $\mathbf{X} \in \mathbb{R}^n$ un vettore aleatorio assolutamente continuo, allora se tutte le variabili aleatorie che lo compongono sono tra loro indipendenti, la densità congiunta di $\mathbf{X}$ equivale alla produttoria di quelle marginali:

$$X_i \perp\!\!\!\perp X_j \;\; \forall i \neq j \in \{1,2,\ldots,n\} \implies f_{\mathbf{X}} = \prod_{i=1}^{n} f_{X_i}.$$

Definizione 3.6 (Valore atteso)**.** Sia $X \in \mathbb{R}$ una variabile aleatoria assolutamente continua e data una funzione misurabile $\varphi\colon \mathbb{R} \to \mathbb{R}$ arbitraria, allora il valore atteso di $\varphi(X)$ è definito come

$$\mathbb{E}[\varphi(X)] \equiv \int_{\mathbb{R}} \varphi(x) f_X(x)\mathrm{d}x,$$

e un simile discorso vale anche per un vettore aleatorio ma con $\varphi\colon \mathbb{R}^n \to \mathbb{R}$, preso $n \in \mathbb{N}_+$.

Definizione 3.7 (σ-algebra generata da X)**.** Sia $X \in \mathbb{R}$ una variabile aleatoria, essa induce una σ-algebra definita come

$$\sigma(X) \equiv \sigma(\{X^{-1}(B) \mid B \in \mathcal{B}(\mathbb{R})\}),$$

ove $\mathcal{B}(\mathbb{R})$ è la σ-algebra di Borel relativa allo spazio dei numeri reali, mentre $\sigma(\mathcal{E})$ è la piú piccola σ-algebra che contiene $\mathcal{E} \subset \mathcal{F}$.

Un simile risultato vale anche per i vettori aleatori $\mathbf{X} \in \mathbb{R}^n$.

Definizione 3.8 (Valore atteso condizionato)**.** Sia $X \in \mathbb{R}$ una variabile aleatoria a media finita $\mathbb{E}[X] < \infty$ e sia $\mathcal{G} \subseteq \mathcal{F}$ una sotto-σ-algebra di $\mathcal{F}$, allora l'attesa di X condizionata a $\mathcal{G}$ è una variabile aleatoria $Y \in \mathbb{R}$ tale che

AC1 $\sigma(Y) \in \mathcal{G}$, ossia Y è $\mathcal{G}$-misurabile e

AC2 per ogni $A \in \mathcal{G}$ la misura di X e Y tramite $\mathbb{P}$ è la stessa:

$$\mathbb{P}(X \in A) = \int_A X d\mathbb{P} = \int_A Y d\mathbb{P} = \mathbb{P}(Y \in A)$$

L'insieme delle Y che soddisfanno le AC1 e AC2 sono indicate col simbolo $\mathbb{E}[X|\mathcal{G}]$ che ne è anche il suo rappresentante: una generica Y si può indicare anche direttamente come $\mathbb{E}[X|\mathcal{G}]$.

Un'analoga definizione vale anche per i vettori aleatori e per i casi misti in cui $\mathbf{X} \in \mathbb{R}^n$ e $\mathbf{Y} \in \mathbb{R}^m$, presi $n, m \in \mathbb{N}_+$.

Osservazione 3.3. Nel caso in cui $\mathcal{G} \equiv \sigma(Y)$, data una variabile aleatoria $Y \in \mathbb{R}$, si è soliti indicare l'attesa condizionata con $\mathbb{E}[X|Y]$ sottintendendo $\sigma(\cdot)$.

Osservazione 3.4 (Dalla teoria alla pratica)**.** Esiste un'interpretazione piú pratica per l'attesa condizionata rispetto alla Def. 3.8 alquanto teorica: la $\mathbb{E}[X|\mathcal{G}]$ si calcola mediando X *considerando* $\mathcal{G}$ *noto*; ciò implica che si sanno tutte le informazioni di $\mathcal{G}$, ovverosia che i suoi eventi si sono realizzati.

Un uso che permette di capire meglio quanto detto si può trovare nella proprietà AC2 imponendo $A = \Omega$, sempre lecito essendo $\mathcal{G}$ una σ-algebra:

$$\mathbb{E}[X] = \mathbb{E}[\mathbb{E}[X|\mathcal{G}]], \tag{3.2}$$

che a parole significa che l'attesa di X si può calcolare mediando l'attesa condizionata; tuttavia $\mathcal{G}$ continua a essere una generica σ-algebra, difficilmente rappresentabile, per cui è opportuno specializzare ulteriormente la (3.2).

Dati $n, h, k \in \mathbb{N}_+$ tali che $n = h + k$, siano

1. un vettore aleatorio $\mathbf{Z} = (\mathbf{X}, \mathbf{Y}) \in \mathbb{R}^n$, con $\mathbf{X} \in \mathbb{R}^h$ e $\mathbf{Y} \in \mathbb{R}^k$, e
2. una funzione $\varphi \colon \mathbb{R}^n \to \mathbb{R}$ misurabile e tale che $\mathbb{E}[\varphi(\mathbf{Z})] < \infty$,

allora, ipotizzando che $\mathbf{X}(\mathbf{Y})$ sia funzione di $\mathbf{Y}$ e che quest'ultimo *non* sia funzione[1] di $\mathbf{X}$, posto $\mathcal{G} \equiv \sigma(\mathbf{Y})$ la (3.2) diventa

$$\mathbb{E}[\varphi(\mathbf{X}(\mathbf{Y}), \mathbf{Y})] = \mathbb{E}[\mathbb{E}[\varphi(\mathbf{X}(\mathbf{Y}), \mathbf{Y})|\mathbf{Y}]];$$

ma ora è facile capire cosa s'intende per $\sigma(\mathbf{Y})$ nota: si assuma che $\mathbf{Y}$ abbia realizzato l'evento $\mathbf{y}$ arbitrario[2], e, siccome non è funzione di $\mathbf{X}$, ciò implica

$$\mathbb{E}[\varphi(\mathbf{X}(\mathbf{Y}), \mathbf{Y})|\mathbf{Y} = \mathbf{y}] = \mathbb{E}[\varphi(\mathbf{X}(\mathbf{y}), \mathbf{y})], \tag{3.3}$$

vale a dire che, in questo caso specifico, $\mathbb{E}[\varphi(\mathbf{X}(\mathbf{Y}), \mathbf{Y})|\mathbf{Y}]$ si svolge mediando $\varphi(\mathbf{X}(\mathbf{Y}), \mathbf{Y})$ ma considerando $\mathbf{Y}$ come un parametro anziché una variabile aleatoria, ossia essa diventa di fatto fissa nel conto perché, appunto, nota.

Lo stesso *non* si può dire invertendo il condizionamento

$$\mathbb{E}[\varphi(\mathbf{X}(\mathbf{Y}), \mathbf{Y})] = \mathbb{E}[\mathbb{E}[\varphi(\mathbf{X}(\mathbf{Y}), \mathbf{Y})|\mathbf{X}]],$$

sebbene l'identità rimanga valida; difatti in questo caso $\mathbf{X}$ è funzione di $\mathbf{Y}$: una sua realizzazione condiziona implicitamente $\mathbf{Y}$ che non può essere

1 Si noti che i due vettori $\mathbf{X}$ e $\mathbf{Y}$ sono per costruzione dipendenti.

2 L'evento $\mathbf{y}$ va visto come un punto se $\mathbf{Y}$ è discreta, come un intervallo se è assolutamente continua e una via di mezzo se è mista.

di conseguenza arbitrario, per cui un'analoga relazione alla (3.3), in cui $\mathbf{X}$ diventa questa volta un parametro, sarebbe in generale sbagliata.

Da questo esempio è chiaro che considerare $\mathcal{G}$ nota è a volte intuitivo, come nella (3.3), semplificando notevolmente il calcolo dell'attesa condizionata, e altre volte di meno nel qual caso non è banale esplicitare $\mathbb{E}[X|\mathcal{G}]$.

Definizione 3.9 (Processo stocastico)**.** Dato $n \in \mathbb{N}_+$, un processo stocastico è un insieme di vettori aleatori $\mathbf{X}_t \in \mathbb{R}^n$ parametrizzati da un indice $t \in I \subseteq \mathbb{R}$:

$$\{\mathbf{X}_t \in \mathbb{R}^n \mid t \in I\} = \{\mathbf{X}_t\}_{t\in I},$$

la cui densità congiunta è

$$\mathbb{P}_{\mathbf{X}_t}(A) = \mathbb{P}(\mathbf{X}_t \in A) = \int_A f_{\mathbf{X}_t}(\mathbf{x},t)\mathrm{d}\mathbf{x}, \quad \forall t \in I \text{ e } \forall A \subseteq \mathbb{R}^n \text{ misurabile.}$$

Comunemente s'interpreta l'indice t come il tempo ponendo $I \equiv \mathbb{R}_+$.

3.2 DESCRIZIONE CINETICA DI [TIPO] BOLTZMANN

3.2.1 Equazione di Boltzmann omogenea

Descrizione classica

Innanzitutto è necessario postulare alcune proprietà del gas da modellizzare.

Ipotesi 3.1. Si consideri un gas composto da particelle che soddisfaccia le successive importanti ipotesi:

- G1 è uniformemente distribuito nello spazio cosicché in ogni punto la distribuzione statistica delle velocità è la stessa;
- G2 è rarefatto, da cui segue che solo le interazioni binarie possono aver luogo o, piú precisamente, sono piú frequenti;
- G3 è composto da particelle indistinguibili e aventi ugual massa;
- G4 le collisioni sono elastiche per cui si ha conservazione dell'impulso e dell'energia esprimibile, grazie alle G2 e G3, come

$$\mathbf{v}' + \mathbf{v}'_* = \mathbf{v} + \mathbf{v}_*, \tag{CI}$$
$$|\mathbf{v}'|^2 + |\mathbf{v}'_*|^2 = |\mathbf{v}|^2 + |\mathbf{v}_*|^2, \tag{CE}$$

  ove $\mathbf{v}', \mathbf{v}'_* \in \mathbb{R}^3$ sono le velocità poscollisionali che collidono colle velocità precollisionali $\mathbf{v}, \mathbf{v}_* \in \mathbb{R}^3$, delle due particelle interagenti[3].

Dalla G4 si possono in realtà esprimere le velocità poscollisionali a partire da quelle precollisionali, definendo quelle che sono le regole d'interazione.

Proposizione 3.1 (Regole d'interazione)**.** *Esistono due funzioni*

$$\psi, \psi_* \colon \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}^3$$

3 In questo contesto non è necessario distinguere quale delle due sia quella interagente e quale quella ricevente, contrariamente a quello delle città.

bilineari che soddisfanno (CI, CE) e tali che

$$\begin{aligned} \mathbf{v}' &= \psi(\mathbf{v},\mathbf{v}_*) \equiv \mathbf{v} + [(\mathbf{v}_* - \mathbf{v})\cdot n]n, \\ \mathbf{v}'_* &= \psi_*(\mathbf{v},\mathbf{v}_*) \equiv \mathbf{v}_* + [(\mathbf{v} - \mathbf{v}_*)\cdot n]n, \end{aligned} \tag{3.4}$$

ove $n \in \mathbb{S}^2$ *è un qualunque vettore appartenente alla sfera unitaria di* $\mathbb{R}^3$.

Dimostrazione. Assumendo l'*ansatz*

$$\mathbf{v}' = \mathbf{v} - \gamma n \quad \text{e} \quad \mathbf{v}'_* = \mathbf{v}_* + \gamma n, \tag{3.5}$$

in cui $\gamma \in \mathbb{R}$ è un parametro da determinare, si può notare che la (CI) è già verificata, mentre imponendo la (CE) si ha

$$\begin{aligned} |\mathbf{v}'|^2 + |\mathbf{v}'_*|^2 &= |\mathbf{v} - \gamma n|^2 + |\mathbf{v}_* + \gamma n|^2 \\ &= (\mathbf{v} - \gamma n)\cdot(\mathbf{v} - \gamma n) + (\mathbf{v}_* + \gamma n)\cdot(\mathbf{v}_* + \gamma n) \\ &= |\mathbf{v}|^2 - 2\gamma(\mathbf{v}\cdot n) + \gamma^2 + |\mathbf{v}_*|^2 + 2\gamma(\mathbf{v}_*\cdot n) + \gamma^2 \\ &= |\mathbf{v}|^2 + |\mathbf{v}_*|^2 + 2\gamma[(\mathbf{v}_* - \mathbf{v})\cdot n + 2\gamma] \\ &= |\mathbf{v}|^2 + |\mathbf{v}_*|^2 \quad \Longrightarrow \quad \gamma[(\mathbf{v}_* - \mathbf{v})\cdot n + \gamma] = 0, \end{aligned}$$

ma escludendo il caso banale di $\gamma = 0$ (nessuna interazione) si ha

$$\gamma = \Gamma(\mathbf{v},\mathbf{v}_*) \equiv (\mathbf{v} - \mathbf{v}_*)\cdot n,$$

per cui γ è in realtà una funzione delle velocità precollisionali; in tal modo la (3.5) diventa la (3.4). □

Osservazione 3.5. Nella precedente dimostrazione $n \in \mathbb{S}^2$ è lasciato arbitrario, seppure da un punto di vista fisico sia ragionevole porlo parallelo al vettore passante per i centri delle particelle collidenti.

Le regole d'interazione nella (3.4) sono anche simmetriche:

Definizione 3.10. Se le regole d'interazione del tipo (3.4) verificano

$$\begin{aligned} \mathbf{v}' &= \psi(\mathbf{v},\mathbf{v}_*) = \psi_*(\mathbf{v}_*,\mathbf{v}), \\ \mathbf{v}'_* &= \psi_*(\mathbf{v},\mathbf{v}_*) = \psi(\mathbf{v}_*,\mathbf{v}), \end{aligned}$$

$\forall \mathbf{v},\mathbf{v}_* \in \mathbb{R}^3$, allora si dicono simmetriche.

Descrizione statistica

Per proseguire è però necessaria una descrizione statistica del gas in esame: siano allora $\{\mathbf{V}_t\}_{t\in\mathbb{R}_+}$ e $\{\mathbf{V}^*_t\}_{t\in\mathbb{R}_+}$ i processi stocastici delle velocità precollisionali, dove $\mathbf{V}_t, \mathbf{V}^*_t \in \mathbb{R}^3$ sono variabili aleatorie di cui le velocità precedenti, $\mathbf{v}$ e $\mathbf{v}_*$, sono due realizzazioni al tempo t; un simile discorso vale per i processi stocastici delle velocità poscollisionali $\{\mathbf{V}'_t\}_{t\in\mathbb{R}_+}$ e $\{\mathbf{V}^{*\prime}_t\}_{t\in\mathbb{R}_+}$.

Osservazione 3.6. Le variabili aleatorie $\mathbf{V}_t$ e $\mathbf{V}^*_t$ non sono indicizzate (per es. $\mathbf{V}^i_t$) proprio per l'indistinguibilità delle particelle assunta dalla G3, da cui si evincono anche che le leggi di $\mathbf{V}_t$ e $\mathbf{V}^*_t$ sono identiche:

$$f_{\mathbf{V}_t}(\mathbf{v},t) = f_{\mathbf{V}^*_t}(\mathbf{v},t) = f(\mathbf{v},t), \quad \forall \mathbf{v} \in \mathbb{R}^3.$$

Le regole d'interazione nella (3.4) diventano

$$[\mathrm{RI}]_{\mathbf{V}} \begin{cases} \mathbf{V}_t' = \psi(\mathbf{V}_t, \mathbf{V}_t^*) \equiv \mathbf{V}_t + [(\mathbf{V}_t^* - \mathbf{V}_t)\cdot n]n, \\ \mathbf{V}_t^{*\prime} = \psi^*(\mathbf{V}_t, \mathbf{V}_t^*) \equiv \mathbf{V}_t^* + [(\mathbf{V}_t - \mathbf{V}_t^*)\cdot n]n, \end{cases}$$

in cui anche n risulta aleatoriamente distribuito; usualmente si postula che non vi siano direzioni preferenziali, ossia $n \sim \mathcal{U}(\mathbb{S}^2)$, essendo queste uniformemente distribuite sulla sfera unitaria.

Tuttavia l'interazione tra due particelle non è detto che avvenga, aspetto formulabile introducendo

$$\Theta \sim \text{Bernoulli}(B((\mathbf{V}_t^* - \mathbf{V}_t)\cdot n)\Delta t) \qquad [\mathrm{Ber}]_{\mathbf{V}}$$

che è una variabile aleatoria di Bernoulli, dipendente da $\mathbf{V}_t$ e $\mathbf{V}_t^*$, la cui probabilità è descritta da due termini:

1. la funzione $B\colon \mathbb{R} \to \mathbb{R}_+$, detta nucleo di collisione, che permette d'avere una maggiore espressività sulle collisioni molecolari[4], e
2. un passo temporale $\Delta t > 0$ per discretizzare il tempo.

Osservazione 3.7. Per definizione di Θ, deve valere $B((\mathbf{V}_t^* - \mathbf{V}_t)\cdot n)\Delta t \leqslant 1$ condizione soddisfacibile anche prendendo un passo temporale Δt adattivo; si vedrà tra poco, però, che, per quanto concerne il modello analitico, tale condizione sarà sempre verificata.

Colle $[\mathrm{RI}]_{\mathbf{V}}$ e $[\mathrm{Ber}]_{\mathbf{V}}$, lo stato dei due agenti al tempo $t+\Delta t$ successivo è dunque dato dal seguente algoritmo d'interazione:

$$[\mathrm{AR}]_{\mathbf{V}} \begin{cases} \mathbf{V}_{t+\Delta t} = (1-\Theta)\mathbf{V}_t + \Theta\mathbf{V}_t', \\ \mathbf{V}_{t+\Delta t}^* = (1-\Theta)\mathbf{V}_t^* + \Theta\mathbf{V}_t^{*\prime}, \end{cases}$$

che di fatto è una regola che descrive se gli agenti interagiscono a coppie (modellizzato dalla Θ) e, nel caso, come interagiscono (modellizzato dalle $\mathbf{V}_t'$ e $\mathbf{V}_t^{*\prime}$) modificando il loro stato al tempo successivo.

Derivazione modello

L'idea successiva, al fine di ricavare un modello dell'evoluzione per la densità f, è di mediare le $[\mathrm{AR}]_{\mathbf{V}}$ attraverso una funzione arbitraria $\varphi\colon \mathbb{R}^3 \to \mathbb{R}$, detta quantità osservabile, computabile dalle realizzazioni o di $\mathbf{V}_{t+\Delta t}$ o di $\mathbf{V}_{t+\Delta t}^*$; pertanto applicando φ alle $[\mathrm{AR}]_{\mathbf{V}}$,

$$\begin{aligned} \varphi(\mathbf{V}_{t+\Delta t}) &= \varphi((1-\Theta)\mathbf{V}_t + \Theta\mathbf{V}_t'), \\ \varphi(\mathbf{V}_{t+\Delta t}^*) &= \varphi((1-\Theta)\mathbf{V}_t^* + \Theta\mathbf{V}_t^{*\prime}), \end{aligned}$$

e mediando, si arriva a

$$\begin{aligned} \mathbb{E}[\varphi(\mathbf{V}_{t+\Delta t})] &= \mathbb{E}[\varphi((1-\Theta)\mathbf{V}_t + \Theta\mathbf{V}_t')], \\ \mathbb{E}[\varphi(\mathbf{V}_{t+\Delta t}^*)] &= \mathbb{E}[\varphi((1-\Theta)\mathbf{V}_t^* + \Theta\mathbf{V}_t^{*\prime})]; \end{aligned}$$

4 Si noti come B dipenda da $(\mathbf{V}_* - \mathbf{V})\cdot n$, termine ripreso dalla Prop. 3.1, cosa che permette d'interpretare n come la direzione lungo la quale le interazioni sono piú frequenti.

per espandere la media, siccome Θ dipende da $(\mathbf{V}_t, \mathbf{V}_t^*, n)$, bisogna avvalersi dell'attesa condizionata, e in particolare della Oss. 3.4, che porta a

$$\begin{aligned}\mathbb{E}[\varphi(\mathbf{V}_{t+\Delta t})] &= \mathbb{E}[\varphi((1-\Theta)\mathbf{V}_t + \Theta\mathbf{V}_t')],\\ &= \mathbb{E}[\mathbb{E}[\varphi((1-\Theta)\mathbf{V}_t + \Theta\mathbf{V}_t') \mid \mathbf{V}_t, \mathbf{V}_t^*, n]],\\ &= \mathbb{E}[\varphi(\mathbf{V}_t)(1 - B((\mathbf{V}_t^* - \mathbf{V}_t)\cdot n)\Delta t)]\\ &\quad + \mathbb{E}[\varphi(\mathbf{V}_t')B((\mathbf{V}_t^* - \mathbf{V}_t)\cdot n)]\Delta t,\end{aligned} \tag{3.6}$$

e similmente per $\mathbb{E}[\varphi(\mathbf{V}_{t+\Delta t}^*)]$; riordinando i termini si ricava

$$\begin{aligned}\frac{\mathbb{E}[\varphi(\mathbf{V}_{t+\Delta t})] - \mathbb{E}[\varphi(\mathbf{V}_t)]}{\Delta t} &= \mathbb{E}[B((\mathbf{V}_t^* - \mathbf{V}_t)\cdot n)(\varphi(\mathbf{V}_t') - \varphi(\mathbf{V}_t))],\\ \frac{\mathbb{E}[\varphi(\mathbf{V}_{t+\Delta t}^*)] - \mathbb{E}[\varphi(\mathbf{V}_t^*)]}{\Delta t} &= \mathbb{E}[B((\mathbf{V}_t^* - \mathbf{V}_t)\cdot n)(\varphi(\mathbf{V}_t^{*\prime}) - \varphi(\mathbf{V}_t^*))],\end{aligned}$$

e passando formalmente al tempo continuo col limite[5] $\Delta t \to 0^+$ si ha

$$\begin{aligned}\frac{d\mathbb{E}[\varphi(\mathbf{V}_t)]}{dt} &= \mathbb{E}[B((\mathbf{V}_t^* - \mathbf{V}_t)\cdot n)(\varphi(\mathbf{V}_t') - \varphi(\mathbf{V}_t))],\\ \frac{d\mathbb{E}[\varphi(\mathbf{V}_t^*)]}{dt} &= \mathbb{E}[B((\mathbf{V}_t^* - \mathbf{V}_t)\cdot n)(\varphi(\mathbf{V}_t^{*\prime}) - \varphi(\mathbf{V}_t^*))],\end{aligned}$$

che, dopo aver espanso i valori attesi secondo le loro definizioni, diventano

$$\begin{aligned}\frac{d}{dt}\int_{\mathbb{R}^3}\varphi(\mathbf{v})f(\mathbf{v},t)d\mathbf{v} &= \int_{\mathbb{R}^6}\langle B(\mathbf{v}_*,\mathbf{v},n)(\varphi(\mathbf{v}') - \varphi(\mathbf{v}))\rangle f_{\mathbf{V}}(\mathbf{v},\mathbf{v}_*,t)d\mathbf{v}d\mathbf{v}_*,\\ \frac{d}{dt}\int_{\mathbb{R}^3}\varphi(\mathbf{v}^*)f(\mathbf{v}_*,t)d\mathbf{v}_* &= \int_{\mathbb{R}^6}\langle B(\mathbf{v}_*,\mathbf{v},n)(\varphi(\mathbf{v}_*') - \varphi(\mathbf{v}_*))\rangle f_{\mathbf{V}}(\mathbf{v},\mathbf{v}_*,t)d\mathbf{v}d\mathbf{v}_*,\end{aligned} \tag{3.7}$$

in cui $B(\mathbf{v}_*,\mathbf{v},n)$ è una forma breve per $B((\mathbf{v}_* - \mathbf{v})\cdot n)$, $\langle\cdot\rangle \equiv \frac{1}{4\pi}\int_{S^2}\cdot dn$ indica la media rispetto a n, e si è posto $\mathbf{V} \equiv (\mathbf{V}_t, \mathbf{V}_t^*)$ con densità congiunta $f_{\mathbf{V}}(\mathbf{v},\mathbf{v}_*,t)$. È infine necessario sommare le (3.7); s'inizi dal primo membro:

$$\frac{d}{dt}\int_{\mathbb{R}^3}\varphi(\mathbf{v})f(\mathbf{v},t)d\mathbf{v} + \frac{d}{dt}\int_{\mathbb{R}^3}\varphi(\mathbf{v}^*)f(\mathbf{v}_*,t)d\mathbf{v}_* = 2\frac{d}{dt}\int_{\mathbb{R}^3}\varphi(\mathbf{v})f(\mathbf{v},t)d\mathbf{v}, \tag{3.8}$$

infatti dal'Oss. 3.6 le due leggi sono di fatto identiche come lo è il dominio d'integrazione, quindi basta il cambio di variabile $\mathbf{v}_* = \mathbf{v}$ nel secondo integrale per rendersi conto dell'equivalenza. Per il secondo membro sono necessarie due ulteriori fondamentali ipotesi:

Ipotesi 3.2. Si assume che il nucleo di collisione sia una funzione pari:

$$B(\mathbf{v}_*,\mathbf{v},n) = B((\mathbf{v}_* - \mathbf{v})\cdot n) = B((\mathbf{v} - \mathbf{v}_*)\cdot n) = B(\mathbf{v},\mathbf{v}_*,n), \quad \forall \mathbf{v},\mathbf{v}_* \in \mathbb{R}^3; \tag{3.9}$$

una scelta tipica è il valore assoluto $|\cdot|$: $B(\mathbf{v}_*,\mathbf{v},n) \equiv |(\mathbf{v}_* - \mathbf{v})\cdot n|$.

Ipotesi 3.3 (Caos molecolare)**.** Le particelle interagenti secondo l'[AR]$_V$ sono campionante indipendentemente. Tal'ipotesi è piú facile da giustificare matematicamente che fisicamente perché semplifica di molto i conti; in ogni caso, la G2 la corrobora poiché in un gas rarefatto è naturale che se due particelle interagiscono, prima che ricollidano, avranno perso ogni vicendevole dipendenza a causa delle innumerevoli altre interazioni colle altre particelle.

5 Ciò permette di soddisfare sempre la condizione nell'Oss. 3.7.

Osservazione 3.8. Dal'Ip. 3.3, unitamente alla Prop. 3.1, si evince che la densità congiunta del vettore aleatorio $\mathbf{V} \equiv (\mathbf{V}_t, \mathbf{V}_t^*)$ è data dal prodotto

$$f_{\mathbf{V}}(\mathbf{v}, \mathbf{v}_*, t) = f_{\mathbf{V}_t}(\mathbf{v}, t) f_{\mathbf{V}_t^*}(\mathbf{v}_*, t) = f(\mathbf{v}, t) f(\mathbf{v}_*, t), \quad \forall \mathbf{v}, \mathbf{v}_* \in \mathbb{R}^3.$$

In tal modo il termine moltiplicato a $\varphi(\mathbf{v}_*)$ nel secondo membro della seconda equazione della (3.7) può essere cosí riformulato:

$$\int_{\mathbb{R}^6} \langle (\varphi(\mathbf{v})) B(\mathbf{v}, \mathbf{v}_*, n) \rangle f(\mathbf{v}, t) f(\mathbf{v}_*, t) d\mathbf{v} d\mathbf{v}_*, \tag{3.10}$$

cambiando le variabili analogamente alla (3.8). Applicando i due risultati illustrati nelle (3.8, 3.10) si perviene finalmente all'equazione di Boltzmann omogenea asimmetrica in forma debole:

$$\frac{d}{dt} \int_{\mathbb{R}^3} \varphi f d\mathbf{v} = \frac{1}{4\pi} \int_{\mathbb{R}^6} \int_{S^2} B(\mathbf{v}, \mathbf{v}_*, n) \left(\frac{\varphi' + \varphi_*'}{2} - \varphi \right) f f_* dn d\mathbf{v} d\mathbf{v}_*, \tag{3.11}$$

ove, per brevità di notazione, gli apici $'$ e asterischi $*$ sottintendono gli argomenti[6] per tutte le funzioni, eccetto per il nucleo di collisione.

Tuttavia, qualora le regole d'interazione siano simmetriche secondo la Def. 3.10, come in questo caso data la Prop. 3.1, sempre con un cambio di variabili e sfruttando la parità del nucleo di collisione vale

$$\int_{\mathbb{R}^6} \left\langle B(\mathbf{v}, \mathbf{v}_*, n) \varphi' \right\rangle f f_* d\mathbf{v} d\mathbf{v}_*, = \int_{\mathbb{R}^6} \left\langle B(\mathbf{v}, \mathbf{v}_*, n) \varphi_*' \right\rangle f f_* d\mathbf{v} d\mathbf{v}_*,$$

da cui segue l'equazione di Boltzmann omogenea simmetrica in forma debole[7]:

$$\frac{d}{dt} \int_{\mathbb{R}^3} \varphi f d\mathbf{v} = \frac{1}{4\pi} \int_{\mathbb{R}^6} \int_{S^2} B(\mathbf{v}, \mathbf{v}_*, n) (\varphi' - \varphi) f f_* dn d\mathbf{v} d\mathbf{v}_*. \tag{EB}_{\mathbf{V}}$$

Si può anche ricavare la forma forte della $[EB]_{\mathbf{V}}$ considerando le regole d'interazione inverse delle $[RI]_{\mathbf{V}}$ [16, § 2.5, p. 15], ma il conto esula dagli scopi di questo elaborato; ciononostante si riporta come riferimento per il paragrafo a venire:

$$\frac{\partial f}{\partial t} = \frac{1}{4\pi} \int_{\mathbb{R}^3} \int_{S^2} B(\mathbf{v}, \mathbf{v}_*, n) (f' f_*' - f f_*) dn d\mathbf{v}_*. \tag{3.12}$$

3.2.2 Equazione di Boltzmann disomogenea

L'equazione originariamente ricavata da Boltzmann non è la (3.12), bensí è quella disomogenea la cui unica differenza è la presenza di un termine avvettivo $\mathbf{v} \cdot \nabla f$ a primo membro:

$$\frac{\partial f}{\partial t} + \mathbf{v} \cdot \nabla f = \frac{1}{4\pi} \int_{\mathbb{R}^3} \int_{S^2} B(\mathbf{v}, \mathbf{v}_*, n) (f' f_*' - f f_*) dn d\mathbf{v}_*, \tag{3.13}$$

dove l'operatore $\nabla \equiv (\partial_{x_1}, \partial_{x_2}, \partial_{x_3})$ è il gradiente e la densità dipende ora anche dallo spazio: $f \equiv f(\mathbf{x}, \mathbf{v}, t)$. Essa è un'equazione integro-differenziale della distribuzione statistica delle posizioni e delle velocità in ogn'istante.

Piú nel dettaglio, il primo membro non è altro che la derivata materiale di f: difatti, posto $B(\mathbf{v}, \mathbf{v}_*, t) \equiv 0$, ovvero in mancanza di collisioni, l'equazione involve in una del trasporto lineare con soluzione nota.

6 Per una spiegazione piú dettagliata si legga poco dopo nel § 3.2.3.
7 Si dice debole siccome essa vale per una funzione *test* φ arbitraria.

Proposizione 3.2. *La soluzione dell'equazione del trasporto lineare*

$$\frac{\partial f}{\partial t} + \mathbf{v} \cdot \nabla f = 0 \tag{3.14}$$

ha come soluzione $f(\mathbf{x},\mathbf{v},t) = f_0(\mathbf{x}-\mathbf{v}t,\mathbf{v})$ *in cui* $f_0 = f(x,v,0)$ *è la distribuzione iniziale.*

Dimostrazione. Si procede usando il metodo delle caratteristiche: nella (3.14) la velocità $\mathbf{v}$ nell'operatore avvettivo non dipende né dallo spazio né dal tempo, quindi si possono prendere le curve, dette appunto caratteristiche, $\hat{\mathbf{x}}(t)$ tali che

$$\frac{d\hat{\mathbf{x}}}{dt} = \mathbf{v} \quad \Longrightarrow \quad \hat{\mathbf{x}}(t) = \hat{\mathbf{x}}_0 + \mathbf{v}t \tag{3.15}$$

indicando con $\hat{\mathbf{x}}_0 \in \mathbb{R}^3$ il piede della caratteristica, ossia la posizione iniziale della traiettoria; valutando allora f lungo queste si ricava $\hat{f}(t) = f(\hat{\mathbf{x}}(t),\mathbf{v},t)$ che ha derivata totale nulla:

$$\frac{d\hat{f}}{dt} = \frac{\partial f}{\partial t} + \frac{d\mathbf{x}}{dt} \cdot \nabla f = \frac{\partial f}{\partial t} + \mathbf{v} \cdot \nabla f = 0,$$

perciò $\hat{f}(t)$ è costante $\forall t \geqslant 0$, e ciò vale per ogni caratteristica essendo $\hat{\mathbf{x}}_0$ arbitrario; eppure, preso un punto generico $(\tilde{\mathbf{x}},\mathbf{v},\tilde{t})$ dello spazio delle fasi, è possibile definire il piede $\hat{\mathbf{x}}_0 \equiv \tilde{\mathbf{x}} - \mathbf{v}\tilde{t}$ da cui

$$\hat{\mathbf{x}}(\tilde{t}) = \hat{\mathbf{x}}_0 + \mathbf{v}\tilde{t} = \tilde{\mathbf{x}},$$

che porta rapidamente alla soluzione trascurando la tilde:

$$f(\mathbf{x},\mathbf{v},t) = f(\hat{\mathbf{x}}(t),\mathbf{v},t) = \hat{f}(t) = \hat{f}(0) = f(\hat{\mathbf{x}}_0,\mathbf{v},0) = f_0(\hat{\mathbf{x}}_0,\mathbf{v}) = f_0(\mathbf{x}-\mathbf{v}t,\mathbf{v}).$$

□

Ciò significa che in assenza di collisioni la distribuzione iniziale f_0 semplicemente trasla rigidamente nello spazio allo scorrere del tempo lungo la direzione della velocità costante $\mathbf{v}$; tale movimento rappresenta la variazione media statistica delle posizioni molecolari.

D'altra parte il secondo membro della (3.13) rappresenta la variazione media statistica dalle velocità molecolari a causa delle collisioni, e viene perciò anche chiamato operatore collisionale.

Pertanto la (3.13) descrive una chiara separazione di effetti: il primo membro altera solo la posizione delle particelle per il trasporto libero, mentre il secondo comporta esclusivamente variazioni della velocità per le collisioni.

L'obbiettivo di Boltzmann era di raccordare due mondi: quello macroscopico, che all'equilibrio termodinamico restituisce quantità fisiche stazionarie e ben definite, e quello microscopico la cui descrizione molecolare implica una continua e caotica variazione delle stesse quantità anche in condizioni di equilibrio termodinamico (agitazione termica).

Ciò è stato possibile introducendo i momenti statistici, ossia quantità macroscopiche calcolabili a partire dalla distribuzione f, soluzione della (3.13); i principali sono, sottintendendo gli argomenti come nella [EB]$_V$,

$$\begin{aligned} \rho(\mathbf{x},t) &\equiv \int_{\mathbb{R}^3} f d\mathbf{v}, & E(\mathbf{x},t) &\equiv \frac{1}{\rho}\int_{\mathbb{R}^3} \|\mathbf{v}\|^2 f d\mathbf{v}, \\ \mathbf{u}(\mathbf{x},t) &\equiv \frac{1}{\rho}\int_{\mathbb{R}^3} \mathbf{v} f d\mathbf{v}, & e(\mathbf{x},t) &\equiv \frac{1}{\rho}\int_{\mathbb{R}^3} \|\mathbf{v}-\mathbf{u}\|^2 f d\mathbf{v}, \end{aligned} \tag{3.16}$$

rispettivamente densità, energia totale, velocità massica ed energia interna[8]. Tali quantità sono definite a partire dalla f, seppure questa sia in genere infattibile da ricavare dalla (3.13), vista la difficile trattabilità analitica dell'equazione; eppure, tramite una sua attenta riformulazione in determinati regimi limite [16, 18], è possibile svincolarsi del tutto dall'evoluzione puntuale della f ricavando un sistema chiuso dei soli momenti (3.16). La convergenza di questi a un valore stazionario è il nesso tra i due mondi attraverso una descrizione mesoscopica indotta dalla distribuzione f.

3.2.3 Equazione di tipo Boltzmann omogenea

La derivazione dettagliata nel § 3.2.1 è valida in realtà per una classe di problemi molto piú generali, detti *di tipo* Boltzmann omogenei, la cui teoria è stata sviluppata da Pareschi *et al.* [18]; sono chiamati cosí per due ragioni:

1. sono formulati per mezzo di equazioni integro-differenziali analoghe strutturalmente alla $[\mathrm{EB}]_V$ e
2. sono applicati a contesti[9] molto distanti da quello originale di una popolazione di particelle di un gas.

Dunque in questo paragrafo si generalizzano molti risultati analoghi a quelli già visti nel § 3.2.1 senza però riscrivere tutt'i passaggi.

Descrizione e derivazione

Sia $\{\mathbf{X}_t\}_{t\in\mathbb{R}_+}$ il processo stocastico relativo allo stato microscopico dell'agente di un sistema, dove $\mathbf{X}_t \in I \subseteq \mathbb{R}^n$ è il vettore aleatorio[10] che descrive i suoi $n \in \mathbb{N}_+$ stati microscopici al tempo t; allora le Ipp. 3.1 e 3.3 diventano:

Ipotesi 3.4. Gli agenti sono caratterizzati dalle seguent'ipotesi:

- B1 la distribuzione statistica dei microstati è uniforme nello spazio;
- B2 le interazioni non binarie tra agenti sono trascurabili;
- B3 gli agenti sono indistinguibili e ciascuno ne rappresenta l'insieme;
- B4 due agenti che interagiscono sono statisticamente indipendenti.

D'altra parte gli stati poscollisionali (postati), $\mathbf{X}'_t$ e $\mathbf{X}^{*\prime}_t$, si legano con quelli precollisionali (prestati), $\mathbf{X}_t$ e $\mathbf{X}^*_t$, tramite

$$[\mathrm{RI}]_X \begin{cases} \mathbf{X}'_t = \psi(\mathbf{X}_t, \mathbf{X}^*_t, \mathbf{y}), \\ \mathbf{X}^{*\prime}_t = \psi_*(\mathbf{X}_t, \mathbf{X}^*_t, \mathbf{y}_*), \end{cases}$$

ove ora

$$\psi \colon \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^h \to \mathbb{R}^n$$
$$\psi_* \colon \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^{h_*} \to \mathbb{R}^n$$

sono generiche regole d'interazione, non necessariamente lineari o simmetriche (Def. 3.12), mentre $\mathbf{y} \in \mathbb{R}^h$ e $\mathbf{y}_* \in \mathbb{R}^{h_*}$, con $h, h_* \in \mathbb{N}$, rappresentano dei coefficienti potenzialmente stocastici.

8 Un'altra quantità macroscopica di rilievo legata a e è la temperatura $\theta(\mathbf{x},t) \equiv \frac{1}{3}e(\mathbf{x},t)$.

9 Esempi importati sono quello sociofisico, come i vari modelli sulle opinioni, ed econofisico, come quello qui considerato delle città, sebbene contenga elementi anche del primo.

10 Si noti che il dominio I degli stati non coincide necessariamente coll'intero spazio reale $\mathbb{R}^n$.

L'interazione tra di essi avviene secondo una variabile aleatoria di Bernoulli del tipo

$$\Theta \sim \text{Bernoulli}(\mu(\mathbf{X}_t, \mathbf{X}_t^*, \mathbf{w}, t)\Delta t), \qquad [\text{Ber}]_\chi$$

con tasso/nucleo d'interazione

$$\mu \colon \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^k \times \mathbb{R}_+ \to \mathbb{R}_+, \qquad [\text{NI}]_\chi$$

per controllare quanto frequentemente due agenti interagiscano, e coefficienti $\mathbf{w} \in \mathbb{R}^k$, con $k \in \mathbb{N}$, potenzialmente stocastici.

Osservazione 3.9. Se non si sa o non si vuole toccare quest'aspetto modellistico si può semplicemente porre unitario:

$$\mu \equiv 1, \quad \forall (\mathbf{x}, \mathbf{x}_*, \mathbf{w}, t) \in \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^k \times \mathbb{R}_+$$

sottintendendo in tal modo un'interazione uniforme tra gli agenti.

Nel complesso i due agenti s'interfacciano mediante l'algoritmo d'interazione

$$[\text{AR}]_\chi \begin{cases} \mathbf{X}_{t+\Delta t} = (1-\Theta)\mathbf{X}_t + \Theta \mathbf{X}_t', \\ \mathbf{X}_{t+\Delta t}^* = (1-\Theta)\mathbf{X}_t^* + \Theta \mathbf{X}_t^{*\prime}, \end{cases}$$

dove $\Delta t \in \mathbb{R}_+$ è il passo temporale atto a discretizzare il tempo.

Osservazione 3.10. La sigla $[\text{AR}]_\chi$ sta per «Azione-Reazione» (Fig. 3.1) poiché si suppone che ogn'interazione (azione), ossia $\Theta = 1$, necessariamente modifica entrambi gli stati degli agenti coinvolti (reazione); in alcuni contesti [17, (2.5) p. 5] questa regola può essere alterata dimodoché solo l'agente che interagisce sia modificato portando alle interazioni «Azione-Azione».

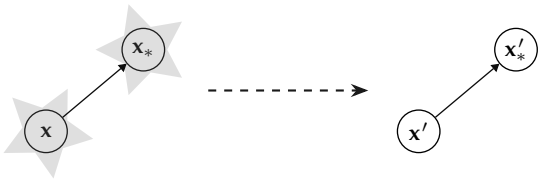

Figura 3.1: Rappresentazione grafica dell'$[\text{AR}]_\chi$.

Infine, riapplicando un ragionamento analogo a quello svolto nel § 3.2.1 colla quantità osservabile $\varphi \colon \mathbb{R}^n \to \mathbb{R}$ per ricavare la $[\text{EB}]_V$, si perviene all'equazione di tipo Boltzmann omogenea

$$\frac{d}{dt}\int_{\mathbb{R}^n} \varphi f \, d\mathbf{x} = \int_{\mathbb{R}^{2n}} \left\langle \mu(\mathbf{x}, \mathbf{x}_*, \mathbf{w}, t) \frac{\varphi' + \varphi_*' - \varphi - \varphi_*}{2} \right\rangle f f_* \, d\mathbf{x} d\mathbf{x}_*, \qquad [\text{EtB}]_\chi$$

nella quale $\langle \cdot \rangle$ rappresenta la media rispetto a tutt'i coefficienti potenzialmente stocastici ($\mathbf{y}$, $\mathbf{y}_*$ e $\mathbf{w}$), mentre il termine $-\varphi - \varphi_*$ discende dal fatto che non si è ipotizzato che il tasso d'interazione μ sia pari.

Definizione 3.11 ($[\text{NI}]_\chi$ pari)**.** Se il tasso d'interazione verifica

$$\mu(\mathbf{x}, \mathbf{x}^*, \mathbf{w}, t) = \mu(\mathbf{x}^*, \mathbf{x}, \mathbf{w}, t) \quad \forall \mathbf{x}, \mathbf{x}^* \in I \subseteq \mathbb{R}^n \text{ e } \forall \mathbf{w} \in \mathbb{R}^k,$$

allora μ è detto pari.

Definizione 3.12 ([RI]$_X$ simmetriche)**.** Se le regole d'interazione verificano

$$\mathbf{x}_t' = \psi(\mathbf{x},\mathbf{x}^*,\mathbf{y}) = \psi_*(\mathbf{x}^*,\mathbf{x},\mathbf{y}),$$
$$\mathbf{x}_t^{*\prime} = \psi_*(\mathbf{x},\mathbf{x}^*,\mathbf{y}_*) = \psi(\mathbf{x}^*,\mathbf{x},\mathbf{y}_*),$$

$\forall \mathbf{x},\mathbf{x}_* \in I \subseteq \mathbb{R}^n$ e $\forall \mathbf{y},\mathbf{y}_* \in \mathbb{R}^h$, allora si dicono simmetriche.

Osservazione 3.11 ([EtB]$_X$ simmetrica)**.** Qualora l'[NI]$_X$ sia pari e le [RI]$_X$ siano simmetriche, la [EtB]$_X$ diventa

$$\frac{d}{dt}\int_{\mathbb{R}^n} \varphi f d\mathbf{x} = \int_{\mathbb{R}^{2n}} \left\langle \mu(\mathbf{x},\mathbf{x}_*,w,t)(\varphi' - \varphi)\right\rangle f f_* d\mathbf{x} d\mathbf{x}_*, \tag{3.17}$$

con un cambio di variabili per invertire $\mathbf{x}$ e $\mathbf{x}_*$ nella differenza $\varphi_*' - \varphi_*$.

Notazione breve

Nei previ paragrafi si è fatto spesso uso, specie per le varie equazioni [di tipo] Boltzmann (3.11–3.13), [EB]$_V$ e [EtB]$_X$, di una notazione abbreviata per tutti gli oggetti salvo il tasso d'interazione μ.

Si definiscono adesso esplicitamente queste abbreviazioni facendo riferimento al contesto generale appena sviluppato. Innanzitutto la densità congiunta f da sola sottintende la variabile dell'agente interagente $\mathbf{x}$ mentre con un pedice f_* quella dell'agente ricevente $\mathbf{x}_*$:

$$f \equiv f(\mathbf{x},t) \qquad \text{e} \qquad f_* \equiv f(\mathbf{x}_*,t);$$

dopodiché le quantità osservabili φ seguono la medesima logica di prima ma coll'aggiunta di un apice per distinguere i postati:

$$\varphi \equiv \varphi(\mathbf{x}) \qquad \text{e} \qquad \varphi_* \equiv \varphi(\mathbf{x}_*),$$
$$\varphi' \equiv \varphi(\mathbf{x}') \qquad \text{e} \qquad \varphi_*' \equiv \varphi(\mathbf{x}_*').$$

Ovviamente questa notazione può essere generalizzata a una qualunque funzione scalare $g\colon \mathbb{R}^n \to \mathbb{R}$ funzione del microstato degli agenti:

$$g \equiv g(\mathbf{x}) \qquad \text{e} \qquad g_* \equiv g(\mathbf{x}_*),$$
$$g' \equiv g(\mathbf{x}') \qquad \text{e} \qquad g_*' \equiv g(\mathbf{x}_*').$$

Infine si osserva che qualora vi siano altri pedici o apici nelle funzioni a venire, essi *non* fanno riferimento a questa notazione abbreviata, se non altrimenti specificato. Per esempio nel prossimo paragrafo si lavora con delle densità f_i, in cui i è l'indice del nodo del grafo associato all'agente; in aggiunta è comunque possibile, anche per questi casi, usare questa notazione riposizionando lievemente i pedici e gli apici:

$$f_i \equiv f_i(\mathbf{x},t) \qquad \text{e} \qquad f_i^* \equiv f_i(\mathbf{x}_*,t).$$

Analisi dimensionale

Tutte le equazioni finora analizzate fanno riferimento a un tempo adimensionale, sebbene l'introduzione di una dimensione sia una modifica alquanto indolore e lasci trasparire un'interessante interpretazione di Δt.

Sia dunque $\tilde{t} = \tau t$ il tempo dimensionale scomposto nel prodotto della dimensione τ col tempo adimensionale t.

Osservazione 3.12. Seguendo la notazione ISO [11, § 6.2] $\tilde{t}$ dev'essere scritto $\tilde{t} = \{t\} \times [t]$ ma, per leggerezza di notazione, si omettono le parentesi ponendo $\tau \equiv [t]$: l'importante è vedere t come il valore numerico variabile e adimensionale, mentre τ come la dimensione costante e fissa.

Detto ciò, mediante la derivata dimensionale

$$\frac{d\cdot}{dt} = \tau \frac{d\cdot}{d\tilde{t}}$$

e le riformulazioni

$$\tilde{f}(\mathbf{x}, \tilde{t}) \equiv f(\mathbf{x}, \tilde{t}/\tau), \quad \text{e} \quad \tilde{f}_*(\mathbf{x}_*, \tilde{t}) \equiv f(\mathbf{x}_*, \tilde{t}/\tau),$$

abbreviate con $\tilde{f}$ e $\tilde{f}_*$ rispettivamente, la $[EtB]_\chi$ diventa

$$\frac{d}{d\tilde{t}} \int_{\mathbb{R}^n} \varphi \tilde{f} d\mathbf{x} = \frac{1}{\tau} \int_{\mathbb{R}^{2n}} \left\langle \mu(\mathbf{x}, \mathbf{x}_*, w, \tilde{t}/\tau) \frac{\varphi' + \varphi'_* - \varphi - \varphi_*}{2} \right\rangle \tilde{f}\tilde{f}_* d\mathbf{x} d\mathbf{x}_*. \tag{3.18}$$

Per capire come varia il tasso d'interazione μ è sufficiente dimensionalizzare la $[Ber]_\chi$:

$$\Theta \sim \text{Bernoulli}\left(\mu(\mathbf{X}_t, \mathbf{X}_t^*, \mathbf{w}, \tilde{t}/\tau) \frac{\Delta\tilde{t}}{\tau}, \right),$$

dove $\Delta\tilde{t} = \tau \Delta t$, da cui si deduce il tasso d'interazione dimensionale $\tilde{\mu}$:

$$\tilde{\mu}(\mathbf{x}, \mathbf{x}_*, w, \tilde{t}) \equiv \frac{1}{\tau} \mu(\mathbf{x}, \mathbf{x}_*, w, \tilde{t}/\tau),$$

che trasforma la (3.18) nell'equazione di tipo Boltzmann dimensionale

$$\frac{d}{d\tilde{t}} \int_{\mathbb{R}^n} \varphi \tilde{f} d\mathbf{x} = \int_{\mathbb{R}^{2n}} \left\langle \tilde{\mu}(\mathbf{x}, \mathbf{x}_*, w, \tilde{t}) \frac{\varphi' + \varphi'_* - \varphi - \varphi_*}{2} \right\rangle \tilde{f}\tilde{f}_* d\mathbf{x} d\mathbf{x}_*. \tag{3.19}$$

Osservazione 3.13 (frequenza d'interazione)**.** Da $\Delta\tilde{t} = \tau\Delta t$ si può interpretare Δt come la frazione della dimensione τ dopo la quale può avvenire un'interazione, da cui si evince [banalmente] che $\Delta\tilde{t}$ è l'intervallo di tempo tra due interazioni. Di conseguenza si può vedere l'$[AR]_\chi$ come un fenomeno periodico di periodo $\Delta\tilde{t}$ e frequenza

$$\tilde{\nu} \equiv \frac{1}{\Delta\tilde{t}} = \frac{1}{\tau\Delta t} \equiv \frac{1}{\tau}\nu,$$

in cui, come per il tasso d'interazione, $1/\tau$ è la dimensione della frequenza adimensionale ν. Ecco che il reciproco del passo temporale Δt prende il significato della frequenza d'interazione del fenomeno, nell'unità di tempo sia adimensionale che dimensionale mediante $1/\tau$.

Osservazione 3.14 (Scelta di τ)**.** La precedente osservazione permette anche di esplicitare un metodo con cui decidere la dimensione temporale τ: si consideri $\Delta t = 0.01$, allora la frequenza è di 100 interazioni nell'unità di tempo. In un fenomeno come quello urbano si possono allora escludere due scale:

- ◇ quella dei mesi o superiori, perché è irrealistico che in un arco temporale cosí lungo vi siano solo 100 interazioni tra le città, e
- ◇ parimenti quella dei secondi o inferiori per un ragionamento simile a quello di prima ma opposto.

È pertanto naturale che la scala ideale in tal contesto sia quella dei giorni.

Cionnonostante, in questo scritto non si studia un'equazione di tipo Boltzmann dimensionale analoga alla (3.19) per tre principali motivazioni:

1. la dimensione va considerata solo se si è interessati a studiare il transitorio, non la distribuzione stazionaria;
2. lo studio di quest'ultimo è incompatibile coll'Ip. 2.2 di grafo statico;
3. mancano i dati storici, sia attendibili che su finestre temporali adeguate, per poter confrontare il transitorio simulato con quello reale.

Tali sono le ragioni per cui il tempo sarà sempre considerato adimensionale da qui in poi: l'obbiettivo è studiare la distribuzione stazionaria raggiunta, non in quanto tempo si raggiunge (tempo di convergenza).

3.3 DESCRIZIONE CINETICA URBANA SU RETI

Si affronta adesso il caso del tessuto interurbano descritto nel § 2.2, considerando le città sia come agenti che come nodi di una rete che ne regola le interazioni: due città possono interagire sse sono connesse; è anche ovvio che l'interazione modifichi, in qualche modo da definire, la loro popolazione.

A grandi linee in questo paragrafo si applicano i concetti della teoria sviluppata nei §§ 2.1 e 3.2.3 e, in particolare, la notazione definita. S'inizia mostrando una derivazione tramite la topologia esatta; quindi si approfondisce una variante nella quale si rilassa la topologia tramite una sua approssimazione; infine, si studiano i collegamenti tra la derivazione esatta e quella di tipo Boltzmann sotto determinate ipotesi.

3.3.1 Equazione di tipo Boltzmann esatta

Descrizione e derivazione

Visto che si assume che valgono le Ip. 3.4[11], dalla B3 sorge un problema non di poco conto: gli agenti sono indistinguibili ma la rete sottostante li rende distinti per via degl'indici $\mathcal{I}$. Per dipanare la questione è sufficiente considerare l'indice come una variabile aleatoria $I \in \mathcal{I}$ cosí da definire il processo stocastico come

$$\{\mathbf{X}_t\} \equiv \{(I, S_t)\}_{t \in \mathbb{R}_+},$$

in cui $S_t \in \mathbb{R}_+$ è la variabile aleatoria relativa alla popolazione dell'agente rappresentativo al tempo t ed è l'unica componente del processo stocastico a variare nel tempo per l'Ip. 2.2.

Osservazione 3.15 (Sulla natura numerica della popolazione)**.** È naturale che non esistano frazioni di persone e che quindi rigorosamente $S_t \in \mathbb{N}$ ma non sempre la scelta piú realistica è piú modellisticamente agevole; infatti trattare S_t come una variabile aleatoria discreta impone un codominio, appunto, discreto che è in genere piú difficile da manipolare matematicamente

11 Perdipiú la B2 è valida *per un passo* Δt, ma nell'arco temporale τ le interazioni si possono considerate anche come non binarie, come accennato nell'Oss. 3.14.

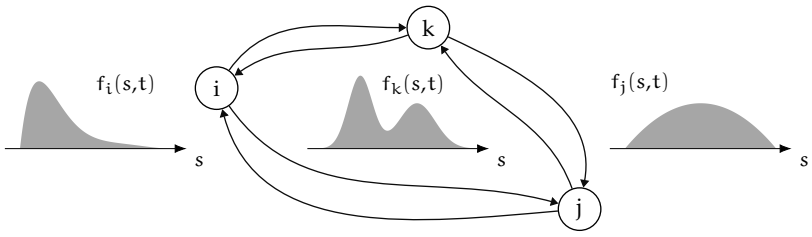

Figura 3.2: Rappresentazione grafica della TCSMA su reti.

in confronto a un intervallo continuo. In ogni caso, ammesso sia necessario, si può poi stimare il numero di abitanti effettivo approssimando i risultati per difetto; l'errore cosí commesso è di al piú una persona.

Cosí ragionando si può nel complesso descrivere statisticamente lo stato microscopico $\mathbf{X}_t \equiv (I, S_t)$ dell'agente rappresentativo colla densità congiunta

$$f\colon \mathcal{I} \times \mathbb{R}_+ \times \mathbb{R}_+ \to \mathbb{R}_+,$$

che è discreta in $i \in \mathcal{I}$ e continua in $s \in \mathbb{R}_+$, uguale a

$$f(i,s,t) \equiv \frac{1}{N} \sum_{j \in \mathcal{I}} f_j(s,t) \otimes \delta(i-j), \tag{3.20}$$

dove $N \equiv |\mathcal{I}|$ è il numero totale d'agenti del grafo mentre $\delta(\cdot)$ denota la delta di Dirac centrata all'origine; d'altra parte

$$f_i : \mathbb{R}_+ \times \mathbb{R}_+ \to \mathbb{R}_+$$

è la densità di S_t relativa all'i-esimo agente (Fig. 3.2).

In tal modo la densità congiunta f è effettivamente indistinguibile rispetto a tutti gli agenti, soddisfacendo la B3, e integra sul suo dominio a uno:

$$\begin{aligned}\mathbb{P}_t(\mathcal{I} \times \mathbb{R}_+) &= \int_{\mathcal{I}} \int_{\mathbb{R}_+} f(i,s,t)\,ds\,di = \frac{1}{N} \sum_{i \in \mathcal{I}} \int_{\mathcal{I}} \int_{\mathbb{R}_+} f_i(s,t)\,ds \otimes \delta(i-j)\,di \\ &= \frac{1}{N} \sum_{i \in \mathcal{I}} \int_{\mathcal{I}} \delta(i-j)\,di = \frac{1}{N} \sum_{i \in \mathcal{I}} 1 = \frac{N}{N} = 1, \quad \forall t \geqslant 0,\end{aligned}$$

ma permette anche di preservare la naturale distinguibilità degli agenti indotta dalla topologia sottostante tramite le singole densità f_i.

Sempre come conseguenza del grafo, la $[\mathrm{Ber}]_X$ deve dipendere dalla matrice d'adiacenza $\mathbf{A}$:

$$\Theta \sim \mathrm{Bernoulli}(A(I, I_*)\Delta t), \tag{$[\mathrm{Ber}]_S^{\mathbf{A}}$}$$

ove $A\colon \mathcal{I} \times \mathcal{I} \to \mathbb{R}$ è una funzione che a ogni coppia d'indici (I, I_*) associa il relativo elemento della matrice d'adiacenza

$$A(I, I_*) \equiv a_{I,I_*} = \begin{cases} 1 & \text{se } (I, I_*) \in \mathcal{E}, \\ 0 & \text{altrimenti,} \end{cases}$$

secondo la (2.2); cosí definita la $[\mathrm{Ber}]_S^{\mathbf{A}}$ inibisce *a priori* l'interazione se i due agenti non sono connessi.

Osservazione 3.16. Rispetto al caso generale $[\mathrm{Ber}]_{\chi}$ il tasso d'interazione ha forma

$$\mu(i,i_*) \equiv A(i,i_*), \tag{3.21}$$

esso, cioè, è per definizione costante sia rispetto agli stati S_t e S_t^* che rispetto al tempo, oltre a non avere coefficienti[12] $\mathbf{w}$.

Osservazione 3.17. Per definizione della distribuzione di Bernoulli, è necessario imporre $A(I,I^*)\Delta t \leqslant 1$, ma dato che $A(I,I^*) \in \{0,1\}$ ciò si traduce nella naturale condizione che $\Delta t \leqslant 1$.

D'altro canto le $[\mathrm{RI}]_{\chi}$ diventano

$$[\mathrm{RI}]_S \begin{cases} S_t' = \psi(S_t,I,S_t^*,I_*,\gamma), \\ S_t^{*\prime} = \psi_*(S_t,I,S_t^*,I_*). \end{cases}$$

Osservazione 3.18. Confrontate alle $[\mathrm{RI}]_{\chi}$, solo la prima funzione della città interagente ψ presenta un coefficiente stocastico $y \equiv \gamma \in \mathbb{R}$, mentre quella relativa alla città ricevente ψ_* non dipende da potenziali coefficienti[13] $\mathbf{y}_*$.

Unendo la $[\mathrm{Ber}]_S^{\mathbf{A}}$ e le $[\mathrm{RI}]_S$, l'$[\mathrm{AR}]_{\chi}$ si scrive

$$[\mathrm{AR}]_S \begin{cases} S_{t+\Delta t} = (1-\Theta)S_t + \Theta S_t', \\ S_{t+\Delta t}^* = (1-\Theta)S_t^* + \Theta S_t^{*\prime}, \end{cases}$$

coerentemente col fatto che in un contesto urbano, qualora una città-nodo interagisca con un'altra, entrambe devono variare il loro stato (Oss. 3.10).

Osservazione 3.19. Paragonato all'$[\mathrm{AR}]_{\chi}$ ci si potrebbe chiedere perché non si considera l'intero vettore aleatorio $\mathbf{X}_t$, come pure nelle $[\mathrm{RI}]_S$: la ragione è che I è una componente statica che non varia nel tempo.

Volendo però includere anche quelle componenti, ossia considerando I_t e I_t^* momentaneamente dinamiche, si possono definire le regole d'interazione

$$[\mathrm{RI}]_I \begin{cases} I_t' = \psi(S_t,I,S_t^*,I_*) \equiv I_t, \\ I_t^{*\prime} = \psi_*(S_t,I,S_t^*,I_*) \equiv I_t^*, \end{cases}$$

da cui segue l'algoritmo d'interazione

$$[\mathrm{AR}]_I \begin{cases} I_{t+\Delta t} = (1-\Theta)I_t + \Theta I_t' = (1-\Theta)I_t + \Theta I_t = I_t, \\ I_{t+\Delta t}^* = (1-\Theta)I_t^* + \Theta I_t^{*\prime} = (1-\Theta)I_t^* + \Theta I_t^* = I_t^*, \end{cases}$$

che soddisfà la staticità di I e I_* e completa, assieme a $[\mathrm{AR}]_S$, la formulazione dell'$[\mathrm{AR}]_{\chi}$ piú generale.

Sia ora $\Phi\colon \mathcal{I} \times \mathbb{R}_+ \to \mathbb{R}$ un'arbitraria quantità osservabile, dalla $[\mathrm{Ber}]_S^{\mathbf{A}}$ l'equazione di tipo Boltzmann omogenea $[\mathrm{EtB}]_{\chi}$ ha forma

$$\frac{d}{dt}\int_{\mathcal{I}}\int_{\mathbb{R}_+} \Phi f dv di = \int_{\mathcal{I}^2}\int_{\mathbb{R}_+^2} A(i,i_*)\frac{\langle \Phi' + \Phi_*' - \Phi - \Phi_* \rangle}{2} f f_* ds ds_* di di_*, \quad [\mathrm{EtB}]_S^{\mathbf{A}}$$

ove si è usata la notazione abbreviata illustrata nel § 3.2.3, mentre $\langle\cdot\rangle$ indica il valore atteso rispetto alla variabile aleatoria γ nelle $[\mathrm{RI}]_S$.

12 Questa situazione si può simbolicamente rappresentare ponendo $k = 0$.
13 Come nell'Oss. 3.16, tale condizione si può simbolicamente rappresentare ponendo $h_* = 0$.

Osservazione 3.20. Anche se la derivazione della $[\mathrm{EtB}]^{\mathbf{A}}_{S}$ è già stata spiegata nel dettaglio nel § 3.2.1, rimane stimolante riportare il passaggio piú difficile, ossia la media condizionata (3.6):

$$\begin{aligned}\mathbb{E}[\Phi(I,S_{t+\Delta t})] &= \mathbb{E}[\mathbb{E}[\Phi(I,(1-\Theta)S_t+\Theta\psi(S_t,I,S_t^*,I_*,\sigma)A(I,I_*)\Delta t)|I,I_*]]\\ &= \mathbb{E}[\mathbb{E}[\Phi(I,S_t)(1-A(I,I_*)\Delta t)\\ &\qquad +\Phi(I,\psi(S_t,I,S_t^*,I_*,\sigma))A(I,I_*)\Delta t]].\end{aligned}$$

Si noti come il condizionamento sia solo rispetto a I a I_*, proprio poiché Θ in $[\mathrm{Ber}]^{\mathbf{A}}_{S}$, tramite il tasso d'interazione μ nella (3.21), dipende solo da essi.

Analisi delle regole d'interazione

Innanzitutto, passando alle realizzazioni di tutte le variabili aleatorie considerate, vale a dire scrivendole in minuscolo, e omettendo la dipendenza dal tempo, le $[\mathrm{RI}]_S$ sono cosí definite[14]

$$[\mathrm{REI}]\begin{cases}\psi(s,i,s_*,i_*\gamma) = s\big(1-E(s,i,s_*,i_*)+\gamma\big)\\ \psi_*(s,i,s_*,i_*) = s_*+sI(s,i,s_*,i_*)\end{cases}$$

ove

$$E,I\colon \mathbb{R}_+\times\mathcal{I}\times\mathbb{R}_+\times\mathcal{I}\to\mathbb{R}_+,$$

sono rispettivamente i tassi d'emigrazione e d'immigrazione, i cui argomenti sono per brevità sottintesi da qui in poi, mentre γ rappresenta fluttuazioni stocastiche. Si caratterizzano ulteriormente E e γ:

Ipotesi 3.5 (Vincoli di positività)**.** Se $\gamma = 0$, per avere un postato s fisicamente sensato, ossia non negativo, si assume che E sia superiormente limitato dalla costante $\lambda \equiv \sup E \in (0,1)$, e inferiormente limitato dall'origine:

$$0 \leqslant E \leqslant \lambda < 1, \quad \forall s,\ \forall s_*,\ \forall i,\ \forall i_*.$$

D'altra parte il postato s'_* non si considera poiché è per definizione sempre positivo qualunque sia il tasso d'immigrazione $I \in \mathbb{R}_+$.

Invece se $\gamma \neq 0$, la perturbazione va limitata inferiormente:

$$s' = s(1-E+\gamma) > 0 \implies \gamma > E-1, \quad \forall E \in [0,\lambda]. \tag{3.22}$$

La scelta di $>$ anziché $\geqslant$ nella (3.22) è ben fondata: di fatto si sta supponendo che le fluttuazioni non possano spopolare una città; tale condizione è necessaria perché nei tassi d'emigrazione venturi, proposti nel § 4.3, compare il rapporto s_*/s o, equivalentemente, s/s_*. Ciò non significa che il modello non possa prevedere un tale fenomeno, poiché la taglia può arbitrariamente avvicinarsi a zero, ma semplicemente che una città non potrà mai avere popolazione nulla colle $[\mathrm{REI}]$; il valore nullo è raggiunto solo *a posteriori* dopo l'approssimazione dai numeri reali a quelli naturali (Oss. 3.15).

Passando alle fluttuazioni γ, esse rappresentano a grandi linee quei fenomeni complessivi di nascita e di morte che vengono considerati ma non direttamente modellati; pertanto valgono le seguenti ipotesi:

14 Sono state in parte ispirate dalla [10, (2.1) p. 223].

Ipotesi 3.6. La variabile aleatoria γ deve soddisfare tre caratteristiche:

- F1 può assumere sia valori positivi che negativi, ma non minori del vincolo imposto dalla (3.22);
- F2 la media è scelta arbitrariamente posta a zero, ossia $\langle\gamma\rangle = 0$;
- F3 seppure non vi siano limiti superiori per l'entità della fluttuazione, è chiaro che piú grande questa è meno è probabile.

Con queste si possono scrivere alcune considerazioni:

1. la distribuzione normale $\mathcal{N}(\mu,\gamma)$ non soddisfà la F1 poiché può assumere valori reali arbitrari con probabilità non nulla;
2. la distribuzione uniforme $\mathcal{U}([a,b])$ è adeguata solo per intervalli finiti e diventa degenere quando un suo estremo diverge, per cui non soddisfà né F2 né F3, mentre F1 sí;
3. la distribuzione esponenziale $\mathrm{Exp}(\lambda)$ è quella piú promettente perché riflette sia F2, dopo un'opportuna traslazione dei valori campionati, che F3, ma sfortunatamente non F1 perché la densità è non nulla al valore estremo $\gamma = E - 1$;
4. l'unica distribuzione che può soddisfare tutt'e tre le caratteristiche ricercate è proprio la distribuzione $\mathrm{Gamma}(\alpha,\beta)$.

Si assuma allora $\hat{\gamma} \sim \mathrm{Gamma}(\alpha,\beta)$ con densità

$$f_\gamma(x) = \frac{1}{\beta^\alpha \Gamma(\alpha)} x^{\alpha-1} \exp\left(-\frac{x}{\beta}\right),$$

dove α e β sono i parametri ordinatamente di forma e di scala, mentre

$$\Gamma(\alpha)\colon \mathbb{R}_+ \to \mathbb{R}_+$$

è la funzione gamma

$$\Gamma(\alpha) \equiv \int_0^{+\infty} y^{\alpha+1} e^{-y}\,dy.$$

Prima di tutto la F2 si può soddisfare semplicemente traslando i valori campionanti di una qualunque $\hat{\gamma}$ di $-\langle\hat{\gamma}\rangle$, ossia si considera la distribuzione $\gamma \sim \hat{\gamma} - \langle\hat{\gamma}\rangle$:

$$\langle\gamma\rangle = \langle\hat{\gamma} - \langle\hat{\gamma}\rangle\rangle = \langle\hat{\gamma}\rangle - \langle\hat{\gamma}\rangle = 0.$$

D'altro canto per i momenti si può imporre

$$\langle\hat{\gamma}\rangle \equiv 1 - E = \alpha\beta \qquad \text{e} \qquad \mathrm{var}(\hat{\gamma}) \equiv \sigma^2 = \alpha\beta^2,$$

in cui σ indica la deviazione, da cui

$$\alpha = \frac{(1-E)^2}{\sigma^2} \qquad \text{e} \qquad \beta = \frac{\sigma^2}{1-E},$$

cosí da soddisfare F1 una volta applicata la traslazione $-\langle\hat{\gamma}\rangle$.

Tuttavia bisogna salvaguardarsi dai casi degeneri della distribuzione gamma: essa infatti se $\alpha = 1$ diventa $\mathrm{Exp}(\beta)$, e se $\alpha < 1$ diverge all'origine; per avere quindi densità nulla all'origine è necessario porre

$$\alpha > 1 \implies \frac{(1-E)^2}{\sigma^2} \geqslant \frac{(1-\lambda)^2}{\sigma^2} > 1 \implies \sigma < |1-\lambda| = 1-\lambda.$$

che implica come non sia possibile avere una deviazione arbitraria per poter soddisfare la F1, ma che questa è limitata superiormente dal massimo tasso d'emigrazione λ: piú grande è λ piú piccola è σ, e viceversa.

In realtà la σ non può davvero assumere valori arbitrari tra 0 e $1-\lambda$: difatti se $\sigma \to 1-\lambda$ vale $\alpha \to 1$ e la distribuzione gamma tende a un'esponenziale; questo ha l'effetto d'introdurre una pericolosa asimmetria nella densità, come si può notare nella Fig. 3.3: una volta centrata rispetto alla media la moda (il valore piú frequente) ha valore negativo portando, in un orizzonte temporale finito, le perturbazioni negative a essere quelle piú frequenti.

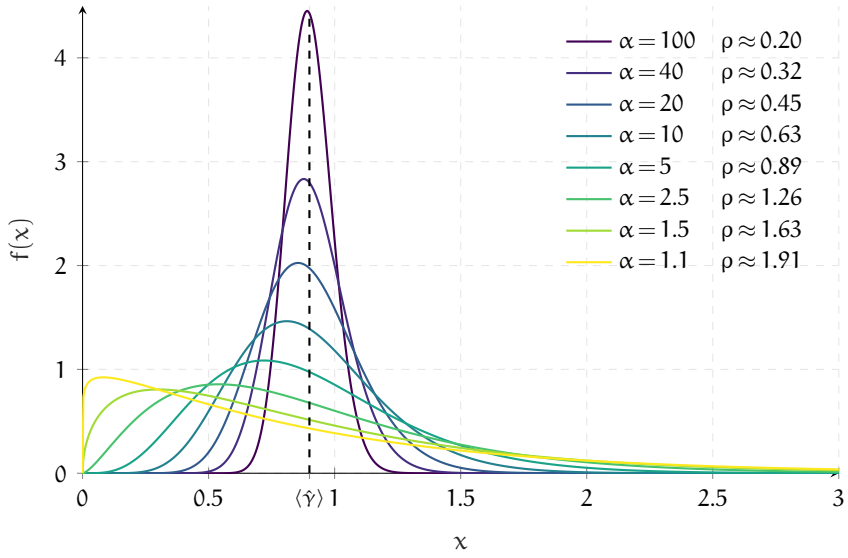

Figura 3.3: Transizione della densità gamma verso l'asimmetria con $\lambda = 0.1$ ed $E = \lambda$, da cui $\langle\hat{\gamma}\rangle = 0.9$ e $\beta = \langle\hat{\gamma}\rangle/\alpha$.

Il risultato è che con σ molto elevato le città tendono a spopolarsi perché v'è una preferenza per fluttuazioni negative. Per ovviare al problema bisogna ulteriormente vincolare σ affinché l'indice di asimmetria $\rho \equiv 2/\sqrt{\alpha}$ sia sufficientemente piccolo; dalla Fig. 3.3 si può perciò imporre $\rho \leqslant 0.2$, cioè

$$\alpha \geqslant 100 \implies \frac{(1-E)^2}{\sigma^2} \geqslant \frac{(1-\lambda)^2}{\sigma^2} \geqslant 100 \implies \sigma \leqslant \frac{1-\lambda}{10}. \tag{3.24}$$

Dunque il vero limite superiore, per preservare la simmetria, è di un ordine di grandezza inferiore a quello precedentemente stimato.

Osservazione 3.21. Quello qui mostrato è solo un possibile modello stocastico perturbativo; infatti, modificando le Ip. 3.6, se ne può scegliere un altro: ad esempio si può imporre una distribuzione uniforme, postulando una massima fluttuazione ammissibile, oppure si può troncare una gaussiana affinché soddisfaccia il vincolo di positività (3.22).

Non manca che caratterizzare il tasso d'immigrazione I:

Ipotesi 3.7 (Conservazione della popolazione media)**.** Si postula che le [REI] conservino [in media] la popolazione totale:

$$s + s_* = \langle s + s_* \rangle = \langle s' + s'_* \rangle \overset{F2}{=} s - sE + s_* + sI,$$

da cui $E \equiv I$, che è ragionevole siccome l'emigrazione e l'immigrazione sono fenomeni relativi (invertendo s ed s_* sarebbe l'opposto).

Osservazione 3.22. Le [REI] non sono ancora complete siccome non si è esplicitata la regola d'emigrazione, la cui scelta, tuttavia, si può dunque vedere come la regola d'interazione stessa: è l'ultimo tassello del mosaico. Pertanto si lascia quest'ultima definizione al § 4.3 nel quale se ne propongono varie.

3.3.2 Equazione di tipo Boltzmann approssimata

L'approssimazione si fonda sulla seguente matrice:

Definizione 3.13 (Matrice dei gradi **B**)**.** Sia $\mathbf{B} \in \mathbb{R}^{N\times N}$ la matrice di rango uno

$$\mathbf{B} \equiv \frac{\mathbf{k}^+(\mathbf{k}^-)^\top}{D_N}$$

definita tramite il prodotto diadico dei vettori

$$\mathbf{k}^+ \equiv (k_1^+, k_2^+, \dots, k_N^+)^\top = \mathbf{A}\mathbf{1} \in \mathbb{R}^N,$$
$$\mathbf{k}^- \equiv (k_1^-, k_2^-, \dots, k_N^-)^\top = \mathbf{A}^\top\mathbf{1} \in \mathbb{R}^N,$$

rispettivamente dei gradi uscenti ed entranti, e la costante[15]

$$D_N \equiv \sum_{i\in\mathcal{I}} k_i^+ = \mathbf{1}^\top\mathbf{k}^+ = \sum_{i\in\mathcal{I}} k_i^- = \mathbf{1}^\top\mathbf{k}^- = \mathbf{1}^\top\mathbf{A}\mathbf{1} = \|\mathbf{A}\|_1. \tag{3.25}$$

Allora **B** approssima la matrice d'adiacenza **A** poiché per definizione vale

$$\left.\begin{aligned} \mathbf{B}\mathbf{1} &= \frac{1}{D_N}\mathbf{k}^+(\mathbf{k}^-)^\top\mathbf{1} = \mathbf{k}^+\frac{D_N}{D_N} = \mathbf{k}^+ = \mathbf{A}\mathbf{1} \\ \mathbf{B}^\top\mathbf{1} &= \frac{1}{D_N}\mathbf{k}^-(\mathbf{k}^+)^\top\mathbf{1} = \mathbf{k}^-\frac{D_N}{D_N} = \mathbf{k}^- = \mathbf{A}^\top\mathbf{1} \end{aligned}\right\} \implies \mathbf{B} \approx \mathbf{A}. \tag{3.26}$$

Inoltre, poiché **A** è simmetrica (Ip. 2.1) vale $\mathbf{k}^+ = \mathbf{k}^- = \mathbf{k}$ e quindi

$$\mathbf{B} = \frac{\mathbf{k}\mathbf{k}^\top}{D_N}.$$

Tramite la Def. 3.13, la $[\text{Ber}]_S^{\mathbf{A}}$ diventa

$$\Theta \sim \text{Bernoulli}(B(I, I^*)\Delta t), \tag{$[\text{Ber}]_S^{\mathbf{B}}$}$$

da cui la $[\text{EtB}]_S^{\mathbf{A}}$ si legge

$$\frac{d}{dt}\int_{\mathcal{I}}\int_{\mathbb{R}_+} \Phi g\, dv\, di = \int_{\mathcal{I}^2}\int_{\mathbb{R}_+^2} B(i, i_*)\frac{\langle \Phi' + \Phi'_* - \Phi - \Phi_*\rangle}{2} g g_*\, ds\, ds_*\, di\, di_*, \tag{$[\text{EtB}]_S^{\mathbf{B}}$}$$

dove le densità approssimate g e g_* sono definite come la (3.20) ma vengono indicate diversamente per distinguerle da quelle esatte f ed f_*.

Sorge però spontanea una domanda non di poco conto: perché non si può scegliere un'altra matrice $\mathbf{C} \in \mathbb{R}^{N\times N}$ per approssimare la **A**, secondo altri criteri analoghi o dissimili alla (3.26)? La risposta non è immediata ma discende in essenza da come la $[\text{EtB}]_S^{\mathbf{B}}$ può essere riformulata. A tal scopo si necessita di un osservazione:

15 La $\|\cdot\|$ è la norma uno applicata alle matrici: $\|\mathbf{A}\|_1 \equiv \sum_{i,j\in\mathcal{I}} |a_{i,j}|$.

Osservazione 3.23 (Teoria e Pratica)**.** Come detto nel Cap. 1, l'interesse è di studiare la distribuzione della popolazione tra città; perciò, non a torto, si può vedere la densità f nella (3.20) come eccessivamente dettagliata, contenendo informazioni legate ai vertici: ai fini pratici è quindi sufficiente ricavare in qualche modo la densità marginale $\bar{f}(s,t) \equiv \int_{\mathcal{I}} f(i,s,t)di$, $\forall t \geqslant 0$, che è esattamente quanto fatto nel § 4.1 per i risultati delle venture simulazioni.

Dall'Oss. 3.23 si possono pertanto specializzare gli osservabili:

Ipotesi 3.8 (Osservabili puntuali)**.** Si considera la seguente classe di osservabili: $\Phi(i,s) \equiv \delta(i-j)\varphi(s)$, $\forall j \in \mathcal{I}$, dove $\varphi : \mathbb{R}_+ \to \mathbb{R}$ è una funzione arbitraria.

Con tale scelta è possibile recuperare un sistema di equazioni debole per le g_i: usare, infatti, l'Ip. 3.8 nella $[\text{EtB}]^{\mathbf{B}}_{S}$ porta a

$$\begin{aligned}\frac{d}{dt}\int_{\mathbb{R}_+} \varphi g_j ds = &\frac{1}{N}\int_{\mathbb{R}_+^2} \frac{\langle \varphi' - \varphi \rangle}{2} g_j \sum_{i\in\mathcal{I}} B(j,i) g_i^* ds ds_* \\ &+ \frac{1}{N}\int_{\mathbb{R}_+^2} \frac{\langle \varphi_*' - \varphi_* \rangle}{2} g_j^* \sum_{i\in\mathcal{I}} B(i,j) g_i ds ds_*, \quad \forall j \in \mathcal{I},\end{aligned}$$

ma essendo $\mathbf{B}$ simmetrica vale

$$\frac{d}{dt}\int_{\mathbb{R}_+} \varphi g_j ds = \frac{1}{N}\int_{\mathbb{R}_+^2} \frac{\langle \varphi' + \varphi_*' - \varphi - \varphi_* \rangle}{2} g_j \sum_{i\in\mathcal{I}} B(j,i) g_i^* ds ds_* \quad \forall j \in \mathcal{I},$$

che in forma matriciale, introducendo le densità vettoriali

$$\mathbf{g} \equiv (g_i)_{i\in\mathcal{I}} \quad \text{e} \quad \mathbf{g}_* \equiv (g_i^*)_{i\in\mathcal{I}},$$

si legge

$$\frac{d}{dt}\int_{\mathbb{R}_+} \varphi \mathbf{g} ds = \frac{1}{N}\int_{\mathbb{R}_+^2} \frac{\langle \varphi' + \varphi_*' - \varphi - \varphi_* \rangle}{2} \mathbf{g} \odot \mathbf{B}\mathbf{g}_* ds ds_*, \tag{3.27}$$

ove $\odot$ indica il prodotto di Hadamard (prodotto per elementi).

Il prossimo passo è introdurre la densità globale delle taglie, ossia la densità marginale della (3.20) rispetto alla popolazione s:

$$\bar{g}_N(s,t) \equiv \int_{\mathcal{I}} g(i,s,t)di = \frac{1}{N}\sum_{i\in\mathcal{I}} g_i = \frac{1}{N}\mathbf{1}^\top \mathbf{g}, \tag{3.28}$$

ove $\mathbf{1} \equiv (1,1,\ldots,1) \in \mathbb{R}^N$, che altro non è che la densità media di S_t fra tutt'i vertici[16]; quindi premoltiplicando la (3.27) per $\frac{1}{N}\mathbf{1}^\top$ e usando la (3.28) si ha

$$\frac{d}{dt}\int_{\mathbb{R}_+} \varphi \bar{g}_N ds = \frac{1}{N^2}\int_{\mathbb{R}_+^2} \frac{\langle \varphi' + \varphi_*' - \varphi - \varphi_* \rangle}{2} \mathbf{g}^\top \mathbf{B}\mathbf{g}_* ds ds_*, \tag{E$\bar{g}$}$$

Osservazione 3.24. La [E$\bar{g}$] si può anche ottenere usando, in luogo del'Ip. 3.8, direttamente un osservabile indipendente dall'indice i: $\Phi(i,s) \equiv \varphi(s)$, confermando quanto premesso nell'Oss. 3.23, cioè che la [E$\bar{g}$] si ricava perdendo le informazioni legati agl'indici negli osservabili.

16 La medesima definizione vale anche per $\bar{f}_N$ della $[\text{EtB}]^{\mathbf{A}}_{S}$.

Eppure la [Eḡ] non è ancora un'equazione chiusa per $\bar{g}_N$ per via del secondo membro nel quale $\mathbf{g}^\top \mathbf{B} \mathbf{g}_*$ richiede sia di conoscere nel dettaglio le connessioni sottostanti del grafo che le distribuzioni della popolazione di ogni città. Tuttavia, avvalendosi della definizione della **B**, la [Eḡ] diventa

$$\frac{d}{dt}\int_{\mathbb{R}_+} \varphi \bar{g}_N ds = \frac{1}{N^2}\int_{\mathbb{R}_+^2} \frac{\langle \varphi' + \varphi'_* - \varphi - \varphi_* \rangle}{2D_N} (\mathbf{k}^\top \mathbf{g})(\mathbf{k}^\top \mathbf{g}_*) ds ds_*, \tag{3.29}$$

che richiede d'introdurre una nuova densità di probabilità:

Definizione 3.14 (Densità dei gradi **k**)**.** Siano $\mathcal{K} \equiv \{0,1,\dots,N\}$ l'insieme dei gradi e

$$d_N : \mathbb{R}_+ \times \mathcal{K} \times \mathbb{R}_+ \to \mathbb{R}_+,$$

la densità di probabilità dell'evento che al tempo t un agente abbia popolazione s e grado k, definizione che la lega alle g_i dalla relazione

$$d_N(s,k,t) = \frac{1}{N}\sum_{i \in \mathcal{I}_k} g_i(s,t),$$

ove

$$\mathcal{I}_k \equiv \{i \in \mathcal{I} \mid k_i = k\} \subseteq \mathcal{I} \tag{3.30}$$

è l'insieme dei nodi con grado k; da essa ne seguono altre due:

$$\begin{aligned} \bar{g}_N(s,t) &= \frac{1}{N}\sum_{i\in\mathcal{I}} g_i(s,t) = \sum_{k\in\mathcal{K}} \frac{1}{N}\sum_{i\in\mathcal{I}_k} g_i(s,t) = \sum_{k\in\mathcal{K}} d_N(s,k,t), \\ \mathbf{k}^\top \mathbf{g}(s,t) &= \sum_{i\in\mathcal{I}} k_i g_i(s,t) = \sum_{k\in\mathcal{K}} k \sum_{i\in\mathcal{I}_k} g_i(s,t) = N \sum_{k\in\mathcal{K}} k d_N(s,k,t). \end{aligned} \tag{3.31}$$

E dalla Def. 3.14 segue la sua normalizzazione:

Definizione 3.15 (Densità dei gradi **k** normalizzati)**.** Sia

$$\hat{\mathcal{K}} \equiv \left\{ \frac{k}{N} \mid k = 0,1,\dots,N \right\} \subseteq [0,1]$$

l'insieme discreto dei gradi normalizzati, allora la sua densità

$$\hat{d}_N : \mathbb{R}_+ \times \hat{\mathcal{K}} \times \mathbb{R}_+ \to \mathbb{R}_+,$$

è cosí definita

$$\hat{d}_N(s,\hat{k},t) \equiv N d_N(s,N\hat{k},t). \tag{3.32}$$

Osservazione 3.25. Visto che due elementi consecutivi dell'insieme $\hat{\mathcal{K}}$ sono separati da un passo costante e uguale a $\Delta\hat{k} \equiv 1/N$, che dalla (3.32) implica

$$\sum_{\hat{k}\in\hat{\mathcal{K}}} \int_{\mathbb{R}_+} \hat{d}_N(s,\hat{k},t) ds \Delta\hat{k} = \sum_{k\in\mathcal{K}} \int_{\mathbb{R}_+} d_N(s,k,t) ds = 1, \quad \forall t \geqslant 0,$$

cosa che permette di vedere la $\hat{d}_N$ come funzione costante a tratti dei gradi normalizzati $\hat{k}$, e similmente per d_N con $\Delta k \equiv 1$.

Osservazione 3.26. L'Oss. 3.25 dà la possibilità anche d'interpretare la (3.31) come un uguaglianza tra densità marginali: $\bar{g}_N = \bar{d}_N = \bar{\hat{d}}_N$.

$$
\begin{array}{rcl}
\bar{\hat{d}}_N & \xrightarrow{N\to\infty} & \bar{\hat{d}} \qquad \text{Topologia rilassata } (\hat{k}) \\
\text{Teo. 3.1} \quad \| & & \| \\
\bar{\hat{g}}_N & \xrightarrow{N\to\infty} & \bar{\hat{g}} \qquad \text{Topologia approssimata } (\mathbf{B}) \\
\text{Def. 3.13} \quad \wr\!\wr & & \wr\!\wr \\
\bar{\hat{f}}_N & \xrightarrow{N\to\infty} & \bar{\hat{f}} \qquad \text{Topologia esatta } (\mathbf{A})
\end{array}
$$

Figura 3.4: Schema riassuntivo del Teo. 3.1.

Con tutte queste definizioni e osservazioni, si può tornare alla (3.29) che può essere riformulare in termini di d_N:

$$\frac{d}{dt}\sum_{k\in\mathcal{K}}\int_{\mathbb{R}_+}\varphi d_N ds = \sum_{k,k_*\in\mathcal{K}}^{N}\int_{\mathbb{R}_+^2}\frac{kk_*}{D_N}\frac{\langle\varphi'+\varphi'_*-\varphi-\varphi_*\rangle}{2}d_N d_N^* ds ds_*,$$

dove k_* è il grado associato a g_N^*, e poi in termini di $\hat{d}_N$ moltiplicando e dividendo per N il primo membro e per N^2 il secondo:

$$\underbrace{\frac{d}{dt}\sum_{\hat{k}\in\hat{\mathcal{K}}}\int_{\mathbb{R}_+}\varphi\hat{d}_N ds\Delta\hat{k}}_{\text{(I)}} = \underbrace{\sum_{\hat{k},\hat{k}_*\in\hat{\mathcal{K}}}\int_{\mathbb{R}_+^2}\frac{\hat{k}\hat{k}_*}{D_N}\frac{\langle\varphi'+\varphi'_*-\varphi-\varphi_*\rangle}{2}\hat{d}_N\hat{d}_N^* ds ds_*}_{\text{(II)}}.$$

Il secondo membro si può ulteriormente manipolare rimoltiplicando e ridividendo per N^2 e definendo il grado normalizzato medio

$$\bar{D}_N \equiv \frac{D_N}{N^2} = \sum_{\hat{k}\in\hat{\mathcal{K}}}\int_{\mathbb{R}_+}\hat{k}\hat{d}_N ds\Delta\hat{k}, \tag{3.33}$$

da cui

$$\text{(II)} = \sum_{\hat{k},\hat{k}_*\in\hat{\mathcal{K}}}\int_{\mathbb{R}_+^2}\frac{\hat{k}\hat{k}_*}{\bar{D}_N}\frac{\langle\varphi'+\varphi'_*-\varphi-\varphi_*\rangle}{2}\hat{d}_N\hat{d}_N^* ds ds_*\Delta\hat{k}\Delta\hat{k}_*.$$

Valutando il limite $N\to\infty$ alla (3.33) e ai membri (I) e (II)

$$(3.33)\xrightarrow{N\to\infty}\bar{D}\equiv\int_0^1\int_{\mathbb{R}_+}\hat{k}\hat{d}\,ds d\hat{k}, \qquad \text{(I)}\xrightarrow{N\to\infty}\frac{d}{dt}\int_0^1\int_{\mathbb{R}_+}\varphi\hat{d}\,ds d\hat{k}$$
$$\text{(II)}\xrightarrow{N\to\infty}\int_0^1\int_0^1\int_{\mathbb{R}_+^2}\frac{\hat{k}\hat{k}_*}{\bar{D}}\frac{\langle\varphi'+\varphi'_*-\varphi-\varphi_*\rangle}{2}\hat{d}\hat{d}^*\,ds ds_* d\hat{k} d\hat{k}_*,$$

si perviene infine a un'equazione di tipo Boltzmann analoga alla $[EtB]_{\mathbf{X}}$:

$$\frac{d}{dt}\int_0^1\int_{\mathbb{R}_+}\varphi\hat{d}\,ds d\hat{k} = \int_0^1\int_0^1\int_{\mathbb{R}_+^2}\frac{\hat{k}\hat{k}_*}{\bar{D}}\frac{\langle\varphi'+\varphi'_*-\varphi-\varphi_*\rangle}{2}\hat{d}\hat{d}^*\,ds ds_* d\hat{k} d\hat{k}_*. \qquad [EtB]_S^k$$

Si è dunque dimostrato il seguente teorema:

Teorema 3.1 (di rilassamento della topologia)**.** *La $[EtB]_S^{\mathbf{B}}$ è formalmente equivalente nel limite $N\to\infty$ a un'equazione di tipo Boltzmann di forma $[EtB]_{\mathbf{X}}$ con microstato $\mathbf{X}_t\equiv\{\hat{K},S_t\}$, ove $\hat{K}\in[0,1]$ è la variabile aleatoria dei gradi normalizzati, osservabili $\varphi(s)$ indipendenti da $\hat{k}$ e nucleo d'interazione $\mu(\hat{k},\hat{k}_*)\equiv(\hat{k}\hat{k}_*)/\bar{D}$.*

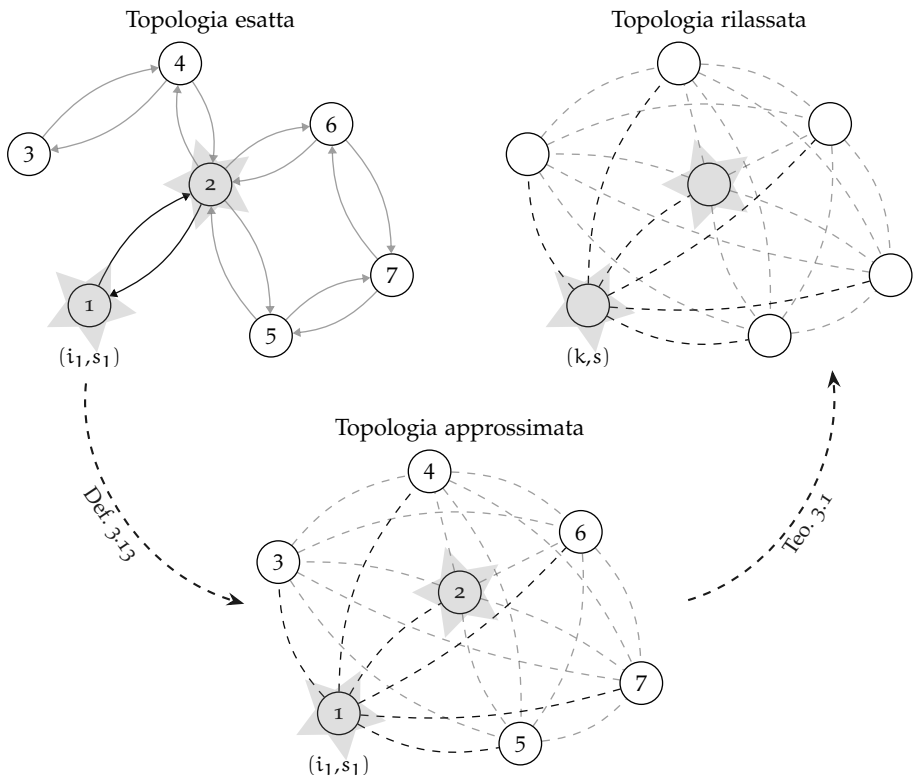

Figura 3.5: Rappresentazione grafica del Teo. 3.1.

Il significato del Teo. 3.1 è profondo: approssimare $[\mathrm{EtB}]_S^{\mathbf{A}}$ con $[\mathrm{EtB}]_S^{\mathbf{B}}$ coincide col perdere la distinguibilità indotta dal grafo rilassando in tal modo la topologia, la quale non scompare ma rimane solo come distribuzione dei gradi; tale riduzione è come sfocare i dettagli delle connessioni tra i vertici: si passa da uno specifico grafo a una classe di grafi aventi la stessa distribuzione dei gradi. Nella Fig. 3.4 è mostrato uno schema riassuntivo del Teo. 3.1 mentre nella Fig. 3.5 è raffigurata una sua rappresentazione piú grafica.

Osservazione 3.27. Il Teo. 3.1 motiva anche il perché non si possa considerare una generica matrice $\mathbf{C}$ che approssima la $\mathbf{A}$: non è detto che rilassi la topologia portando a un'equazione analoga alla $[\mathrm{EtB}]_S^k$.

Osservazione 3.28. La Fig. 3.4 contiene al suo interno $\hat{f}$ che ha un'analoga definizione alla Def. 3.15, ma normalizzando gl'indici: sia

$$\hat{\mathcal{I}} \equiv \left\{ \frac{i}{N} \mid i = 0,1,\dots,N \right\} \subseteq [0,1]$$

l'insieme discreto degl'indici normalizzati, allora la (3.20) normalizzata, per analogia colla (3.32), si può definire come

$$\hat{f}(\hat{\imath},s,t) = Nf(N\hat{\imath},s,t) = \sum_{j\in\mathcal{I}} f_j(s,t) \otimes \delta(N\hat{\imath} - j),$$

ma per la condizione di normalità $\mathbb{P}([0,1]\times\mathbb{R}_+) = 1$, $\forall t \geqslant 0$, deve valere[17]

$$f_{\hat{\jmath}}(s,t) = Nf_j(s,t), \quad \text{ove } j = N\hat{\jmath},$$

17 Si ricordi che l'indice j non è un argomento della densità f_j, quindi può essere cambiato con un altro indice purché sia distinto da tutti gli altri secondo la trasformazione scelta.

anch'essa analoga alla (3.32), da cui

$$\underset{\wedge}{f}(\underset{\wedge}{i},s,t) = \frac{1}{N}\sum_{\underset{\wedge}{j}\in\underset{\wedge}{\mathcal{I}}} f_{\underset{\wedge}{j}}(s,t)\otimes\delta(N(\underset{\wedge}{i}-\underset{\wedge}{j})) = \frac{1}{N}\sum_{\underset{\wedge}{j}\in\underset{\wedge}{\mathcal{I}}} f_{\underset{\wedge}{j}}(s,t)\otimes\delta(\underset{\wedge}{i}-\underset{\wedge}{j}).$$

Attraverso $\underset{\wedge}{f}$ e $i = N\underset{\wedge}{i}$ la densità marginale $\bar{f}_N$ diventa

$$\bar{f}_N = \int_{\mathcal{I}} f(i,s,t)di = \int_{\underset{\wedge}{\mathcal{I}}} Nf(\underset{\wedge}{i},s,t)d\underset{\wedge}{i} = \int_{\underset{\wedge}{\mathcal{I}}} \underset{\wedge}{f}(\underset{\wedge}{i},s,t)d\underset{\wedge}{i} = \bar{\underset{\wedge}{f}}_N = \frac{1}{N}\sum_{\underset{\wedge}{i}\in\underset{\wedge}{\mathcal{I}}} f_{\underset{\wedge}{i}}(s,t),$$

e, detto $\Delta\underset{\wedge}{i} \equiv 1/N$ il passo tra due indici normalizzati consecutivi, si arriva a

$$\bar{f}_N = \bar{\underset{\wedge}{f}}_N = \sum_{\underset{\wedge}{i}\in\underset{\wedge}{\mathcal{I}}} f_{\underset{\wedge}{i}}(s,t)\Delta\underset{\wedge}{i} \xrightarrow{N\to\infty} \bar{f} = \bar{\underset{\wedge}{f}} = \int_0^1 f_{\underset{\wedge}{i}}(s,t)d\underset{\wedge}{i},$$

mentre la $\underset{\wedge}{f}$ si legge

$$\underset{\wedge}{f}(\underset{\wedge}{i},s,t) = \int_0^1 f_{\underset{\wedge}{j}}(s,t)\otimes\delta(\underset{\wedge}{i}-\underset{\wedge}{j})d\underset{\wedge}{j} = f_{\underset{\wedge}{i}}(s,t).$$

Un simile discorso vale per $\underset{\wedge}{g}$.

Osservazione 3.29 (Sulla bontà di $\mathbf{A}\approx\mathbf{B}$)**.** Su un aspetto al momento non ci si può esprimere: quanto bene la $\mathbf{B}$ approssima la matrice $\mathbf{A}$ in generale? Sarebbe necessario elaborare ulteriormente la teoria per trovare una risposta, obbiettivo, come già detto, non di questa tesi. Tuttavia, euristicamente, si può riflettere in questo modo: con $N \gg 1$ le singole connessioni sono meno importati, per cui vengono meno i dettagli, mentre la panoramica può essere ragionevolmente colta da $\mathbf{B}$; ciò suggerirebbe che piú sono i nodi migliore è l'approssimazione $\mathbf{A}\approx\mathbf{B}$. Tale breve riflessione è però solo un intuito, *non* una dimostrazione rigorosa, e dunque anche potenzialmente falsa.

Osservazione 3.30. Si veda [17] per la dimostrazione del Teo. 3.1 in un contesto delle reti sociali e nel caso di un grafo diretto, non necessariamente simmetrico, oltre che con delle $[RI]_\chi$ lineari e simmetriche.

3.3.3 Altri nessi distinguibile-indistinguibile

Il Teo. 3.1 dimostra come si possa rilassare la struttura sottostante indotta dal grafo passando a una densità dei gradi normalizzati, recuperando cosí un'equazione classica di tipo Boltzmann. Risulta perciò stimolante esplorare se sussitano altre ipotesi che permettano di trasformare la $[E\bar{g}]$ in una forma analoga alla $[EtB]_\chi$, approfondendo cosí i legami tra la teoria retale appena sviluppata e la teoria di tipo Boltzmann.

Ipotesi semplificative

Ipotesi 3.9. Esistono due principali ipotesi semplificative della $[E\bar{g}]$:

S1 Il grafo è completamente connesso con matrice d'adiacenza unitaria:

$$\mathbf{B} \equiv \mathbb{1} \iff b_{i,j} \equiv 1, \quad \forall i,j \in \mathcal{I}.$$

S2 Gli agenti sono indistinguibili rispetto al grafo:

$$g_i(s,t) = g_j(s,t) = g(s,t) \quad \forall i,j \in \mathcal{I},$$

Analisi della S1

Notando che $\mathbf{B} = \mathbb{1} = \mathbf{1}\mathbf{1}^\top$, la [Eḡ] diventa

$$\begin{aligned}\frac{d}{dt}\int_{\mathbb{R}_+} \varphi \bar{g}_N ds &= \frac{1}{N^2}\int_{\mathbb{R}_+^2} \frac{\langle \varphi' + \varphi'_* - \varphi - \varphi_* \rangle}{2} \mathbf{g}^\top \mathbf{1}\mathbf{1}^\top \mathbf{g}_* ds ds_* \\ &= \frac{1}{N^2}\int_{\mathbb{R}_+^2} \frac{\langle \varphi' + \varphi'_* - \varphi - \varphi_* \rangle}{2} (\mathbf{1}^\top \mathbf{g})(\mathbf{1}^\top \mathbf{g}_*) ds ds_* \\ &= \int_{\mathbb{R}_+^2} \frac{\langle \varphi' + \varphi'_* - \varphi - \varphi_* \rangle}{2} \left(\frac{1}{N}\mathbf{1}^\top \mathbf{g}\right)\left(\frac{1}{N}\mathbf{1}^\top \mathbf{g}_*\right) ds ds_*,\end{aligned}$$

ma ricordando (3.28) si arriva a

$$\frac{d}{dt}\int_{\mathbb{R}_+} \varphi \bar{g}_N ds = \int_{\mathbb{R}_+^2} \frac{\langle \varphi' - \varphi + \varphi'_* - \varphi_* \rangle}{2} \bar{g}_N \bar{g}^*_N ds ds_*, \tag{3.34}$$

che è affine alla $[\text{EtB}]_\chi$ ma con microstato $\mathbf{X}_t \equiv S_t$ scalare, regole d'interazione $[\text{RI}]_S$ e nucleo d'interazione unitario[18] $\mu \equiv 1$ nella $[\text{Ber}]_\chi$. Ciò significa che con S1 gli agenti, nonostante siano distinti per il grafo, si possono vedere come indistinguibili purché si consideri la distribuzione media (3.28).

Analisi della S2

Sotto tal'ipotesi, che equivale coll'assumere $\mathbf{g} = \mathbf{1}g$ e $\mathbf{g}_* = \mathbf{1}g_*$, la [Eḡ] diventa

$$\begin{aligned}\frac{d}{dt}\int_{\mathbb{R}_+} \varphi \bar{g} ds &= \frac{1}{N^2}\int_{\mathbb{R}_+^2} \frac{\langle \varphi' + \varphi'_* - \varphi - \varphi_* \rangle}{2} g \mathbf{1}^\top \mathbf{B}\mathbf{1} g_* ds ds_* \\ &= \frac{\mathbf{1}^\top \mathbf{B}\mathbf{1}}{N^2}\int_{\mathbb{R}_+^2} \frac{\langle \varphi' + \varphi'_* - \varphi - \varphi_* \rangle}{2} g g_* ds ds_*\end{aligned}$$

e rimembrando le (3.25, 3.26) si ottiene

$$\frac{d}{dt}\int_{\mathbb{R}_+} \varphi g ds = \int_{\mathbb{R}_+^2} \frac{D_N}{N^2} \frac{\langle \varphi' - \varphi + \varphi'_* - \varphi_* \rangle}{2} g g_* ds ds_*. \tag{3.35}$$

In questo contesto il rapporto $D_N/N^2 \in [0,1]$, che si ricorda essere il grado medio normalizzato (3.33), rappresenta quanto la rete è topologicamente simile a una completamente connessa; difatti D_N è altrettanto interpretabile come il numero di lati presenti in un grafo diretto con limite superiore proprio N^2, ossia il numero totale di coppie estraibili da N nodi.

Per il resto l'equazione è analoga alla $[\text{EtB}]_\chi$ con microstato $\mathbf{X}_t \equiv S_t$ scalare, regole d'interazione $[\text{RI}]_S$ e nucleo d'interazione $\mu \equiv D_N/N^2 = \bar{D}_N$ nella $[\text{Ber}]_\chi$ costante. Dunque l'indistinguibilità degli agenti ha una notevole conseguenza sulla [Eḡ]: essa permette di attribuire l'effetto complessivo del grafo al solo coefficiente $\bar{D}_N$ il quale, dunque, ne rappresenta gli ultimi bagliori prima di una sua totale scomparsa per la S1.

18 Tale risultato implica anche che un nucleo di collisione unitario nella $[\text{EtB}]_\chi$ è il corrispettivo di una matrice unitaria nella $[\text{EtB}]_S^{\mathbf{B}}$.

Analisi delle S1 e S2

Per la S2 la densità media (3.28) diventa

$$\bar{g}_N(s,t) = \frac{1}{N}\sum_{i\in\mathcal{I}} g_i(s,t) \overset{S2}{=} \frac{1}{N}\sum_{i\in\mathcal{I}} g(s,t) = g(s,t),$$

ossia $\bar{g}_N$ coincide con quella di tutti gli agenti g, essendo questi, appunto, indistinguibili. Allora, dalla S1, la (3.34) diventa

$$\frac{d}{dt}\int_{\mathbb{R}_+} \varphi g ds = \int_{\mathbb{R}_+^2} \frac{\langle \varphi' - \varphi + \varphi'_* - \varphi_* \rangle}{2} g g_* ds ds_*, \tag{3.36}$$

che è uguale in tutto e per tutto alla $[EtB]_X$ con microstato $\mathbf{X}_t \equiv S_t$ scalare, regole d'interazione $[RI]_S$ e nucleo d'interazione unitario $\mu \equiv 1$ nella $[Ber]_X$. In effetti, contrariamente alle (3.34), con $\bar{g}_N$, e (3.35), con $\bar{D}_N$, il grafo è completamente scomparso dalla (3.36).

4 SIMULAZIONI

In questo capitolo si applica tutta la teoria affrontata in quello precedente. Innanzitutto si descrive l'algoritmo con cui sono stati svolte le simulazioni; quindi si spiegano le formule che definiscono i grafici dei risultati per poi motivare rapidamente il perché le fluttuazioni possono [e anzi devono] essere trascurate; successivamente si propongono varie regole d'emigrazione, analizzate studiando una configurazione di riferimento, e s'interpreta con questi risultati, assieme a un paio di studi parametrici, il fenomeno della migrazione; infine si mostrano alcune varianti delle regole d'emigrazione previamente proposte, e due simulazioni per l'Italia e la Valle d'Aosta.

4.1 METODO MONTE CARLO

Nel contesto delle TCSMA l'obbiettivo, com'è chiaro dal Cap. 3, è quello di ricavare la densità $f(\mathbf{x},t)$ nella $[\mathrm{EtB}]_X$ al variare del tempo. Si potrebbe allora pensare di discretizzare quest'ultima equazione mediante il Metodo delle Differenze Finite, degli Elementi Finiti o dei Volumi Finiti; tuttavia, vi sono due principali problemi:

1. l'impossibilità, in generale, di ricavare la forma forte della $[\mathrm{EtB}]_X$, specie nel caso delle $[\mathrm{RI}]_X$ non lineari;
2. anche ipotizzando di trovare la forma forte a cui applicare i precedenti metodi, la sua natura integro differenziale la rende complessa da manipolare dato che l'operatore collisionale[1], come

$$Q(f,f)(v,t) \equiv \frac{1}{4\pi}\int_{\mathbb{R}^3}\int_{S^2} B(\mathbf{v},\mathbf{v}_*,n)\left(f(v',t)f(v'_*,t)-f(v,t)f(v_*,t)\right)dn d\mathbf{v}_*$$

nella (3.13), dipende dalla densità medesima.

Per tali ragioni si procede in maniera piú semplice: conoscendo l'$[\mathrm{AR}]_S$, che governa le interazioni binarie tra agenti, si possono quindi direttamente simulare tutte le molteplici collisioni mediante un metodo di Monte Carlo di tipo Nanbu-Babovsky, descritto nel dettaglio nell'Alg. 1 nel quale $T>0$ è il tempo finale di simulazione, mentre P è la popolazione totale iniziale. Infatti usare un metodo di Monte Carlo è un approccio piú fisico perché si recupera la fisica particellare stocastica soggiacente all'equazione di Boltzmann.

Osservazione 4.1. Nella linea 18, $\bar{q}_N$ è una densità arbitraria: va intesa come $\bar{f}_N$ se l'approccio è esatto e come $\bar{g}_N$ se è approssimato.

Osservazione 4.2. Piú in generale, nella linea 1, $\mathcal{S}_0$ può essere campionato da una densità $\bar{q}_0$ iniziale; tuttavia non conoscendone alcuna per la distribu-

1 Vale a dire la parte integrale della forma forte dell'equazione di tipo Boltzmann, in genere scritta a secondo membro, legata agli effetti collisionali piuttosto che di trasporto.

Algoritmo 1: Algoritmo $[AR]_S$ di tipo Nanbu-Babovsky

Dati: $N \in \mathbb{N}_+$, (numero d'agenti), $\Delta t \leqslant 1$, (passo temporale), σ, (fluttuazione), $T > 0$, (tempo finale), P, (popolazione totale iniziale), $\mathbf{A}$ (matrice d'adiacenza) e $\mathbf{B}$ (matrice d'adiacenza approssimata);

$\mathcal{S}^0 \leftarrow (s_1^0, s_2^0, \ldots, s_N^0) \equiv (P/N)\mathbf{1} \in \mathbb{R}_+^N$;
per $n = 0,1,2,\ldots,\lfloor T/\Delta t \rfloor - 1$ **fai**
  $p \leftarrow$ permutazione indipendente di $\{1,2,\ldots,N\}$;
  per $c = 1,2,\ldots,\lfloor N/2 \rfloor$ **fai** `# numero di coppie`
    $i \leftarrow p(c)$; $r \leftarrow p(\lfloor N/2 \rfloor + c)$;
    se *esatto* **allora**
      $\Theta \leftarrow \text{Bernoulli}(A(i,r)\Delta t)$;
    altrimenti `# è approssimato`
      $\Theta \leftarrow \text{Bernoulli}(B(i,r)\Delta t)$;
    se $\Theta = 1$ **allora**
      $E \leftarrow E(s_i^n, i, s_r^n, r)$;
      $\gamma \leftarrow \text{Gamma}((1-E)^2/\sigma^2, \sigma^2/(1-E))$;
      $s_i^{n+1} \leftarrow s_i^n(1 - E + \gamma)$; `# città interangente`
      $s_r^{n+1} \leftarrow s_r^n + s_i^n E$; `# città ricevente`
    altrimenti
      $s_i^{n+1} \leftarrow s_i^n$; $s_r^{n+1} \leftarrow s_r^n$;
  $\mathcal{S}^{n+1} \leftarrow (s_1^{n+1}, s_2^{n+1}, \ldots, s_N^{n+1})$;
  $\bar{q}_N(s, (n+1)\Delta t) \leftarrow$ istogramma di $\mathcal{S}^{n+1}$;

zione della popolazione, si è deciso di definire $\mathcal{S}^0$ come un vettore uniforme rispetto a una popolazione totale P iniziale.

4.2 RAPPRESENTAZIONE DEI RISULTATI

Per analizzare i risultati delle simulazioni è opportuno descrivere nel dettaglio le funzioni usate per descriverli; per farlo, però, si devono prima approfondire le conseguenze della natura stocastica dell'Alg. 1 e la struttura dei risultati. Verso la fine si motiva anche l'omissione delle fluttuazioni γ.

4.2.1 Intervalli di confidenza

Innanzitutto simulando la $[EtB]_S^A$ mediante Alg. 1, e quindi l'algoritmo $[AR]_S$, si sta introducendo nei risultati un rumore di natura stocastica: piú simulazioni daranno risultati diversi per cui la singola non ha piú rilevanza statistica; bisogna cioè calcolarne molteplici e valutare gl'intervalli di confidenza per conoscere l'incertezza sulla stima della media.

Sia $R \in \mathbb{N}_+$ il numero di simulazioni eseguite e $\mathbf{S} \in \mathbb{R}_+^R$ il vettore aleatorio della popolazione di una città relativa a ciascuna simulazione.

Ipotesi 4.1 (Numero di simulazioni)**.** In questo elaborato si assume $R = 100$ per avere un numero statisticamente significativo di dati.

Per l'Alg. 1 le componenti $(S_1, S_2, \ldots, S_R)$ di $\mathbf{S}$ sono indipendenti e identicamente distribuite dalla densità $\bar{l}_N$, proprietà che permette di definire

$$\bar{S} \equiv \frac{1}{R}\sum_{r=1}^{R} S_r \quad \text{e} \quad V \equiv \frac{1}{R-1}\sum_{r=1}^{R}(S_r - \bar{S})^2,$$

rispettivamente la media e la varianza campionarie. Allora, data μ la *vera*[2] media della densità $\bar{l}_N$, la distribuzione T di Student con parametro $R-1$ si scrive

$$T = \frac{\bar{S}-\mu}{\sqrt{V/R}} \sim \text{Student}(R-1).$$

Da questa si può definire l'intervallo di confidenza [simmetrico] con livello di confidenza α ponendo

$$\mathbb{P}\left(-t_{R-1}^{\alpha/2} \leqslant T \leqslant t_{R-1}^{\alpha/2}\right) = 1-\alpha \tag{4.1}$$

ove $t_{R-1}^{\alpha/2} \in \mathbb{R}$ è quel valore reale tale che

$$\mathbb{P}\left(T < t_{R-1}^{\alpha/2}\right) = 1 - \frac{\alpha}{2}.$$

Esplicitando la T nella (4.1) e isolando la media μ si ha

$$\mathbb{P}\left(\bar{S} - t_{R-1}^{\alpha/2}\sqrt{\frac{V}{R}} \leqslant \mu \leqslant \bar{S} + t_{R-1}^{\alpha/2}\sqrt{\frac{V}{R}}\right) = 1-\alpha,$$

da cui

$$IC_R^{\alpha}(\mathbf{S}) \equiv \left[\bar{S} - t_{R-1}^{\alpha/2}\sqrt{\frac{V}{R}}, \bar{S} + t_{R-1}^{\alpha/2}\sqrt{\frac{V}{R}}\right]$$

è la stimatore intervallare che definisce l'intervallo di confidenza ricercato, esprimibile anche piú compattamente come

$$IC_R^{\alpha}(\mathbf{S}) \equiv \bar{S} \pm t_{R-1}^{\alpha/2}\sqrt{V/R}. \tag{4.2}$$

Ipotesi 4.2 (livello di confidenza)**.** Si sceglie $\alpha = 0.05$ cosí d'avere un intervallo con livello di confidenza 0.95.

Osservazione 4.3. È naturale che lo stimatore intervallare (4.2) appena ricavato vale tanto per il vettore aleatorio $\mathbf{S}$ che per le sue realizzazioni[3] $\mathbf{s}$, che sono proprio il risultato delle R simulazioni.

Il significato dell'intervallo di confidenza in essenza è l'errore statistico commesso: esso valuta quanto è probabile che lo stimatore intervallare (4.2) contenga il parametro μ; in altre parole IC_R^{α} misura l'incertezza sulla stima della media: preso un campione $\mathbf{s}$, piú l'intervallo di confidenza è esteso piú la media campionaria $\bar{s}$ è una stima incerta dell'effettiva media μ; viceversa piú è stretto, piú la stima $\bar{s}$ è precisa nel senso che μ si trova in un intorno piccolo della media campionaria. Sotto questo punto di vista è concettualmente analogo alla precisione di uno strumento di misura.

2 Vale a dire μ *non* è una variabile aleatoria ma l'esatto parametro della media della $\bar{q}_N$.
3 In tal caso la (4.2) si dice stima intervallare.

4.2.2 Struttura dei dati

I dati hanno una struttura di un tensore del quart'ordine $\mathbf{s} \in \mathbb{R}_+^{N_f \times 2 \times N \times R}$ i cui indici hanno i seguenti significati

$$s_{i,r}^{n,a} \begin{cases} n = \text{istante temporale}, & a = \text{tipo di simulazione}, \\ i = \text{indice della città}, & r = \text{numero della simulazione}. \end{cases}$$

L'istante temporale è definito tramite tre parametri in $\mathbb{N}_+$:

- $N_t \equiv \lfloor T/\Delta t \rfloor$ è il numero totale di tempi simulati;
- $N_s \ll N_t$ è il numero di catture dai tempi simulati;
- $N_f < N_s$ è il numero di tempi ridotti dalle catture.

Sia le catture che le riduzioni sono campionate in intervalli equispaziati con passi

$$\Delta s = N_t/N_s \quad \text{e} \quad \Delta f = N_s/N_f$$

ove si suppone, per semplicità, che N_s e N_f siano divisori ordinatamente di N_t e N_s, ovvero

$$N_t/N_s - \lfloor N_t/N_s \rfloor = 0 \quad \text{e} \quad N_s/N_f - \lfloor N_s/N_f \rfloor = 0,$$

e ugualmente si assume fra Δt e T.

Tramite l'Alg. 1 si simulano in totale N_t tempi con passo Δt di cui N_s sono salvati nel tensore $\underset{\sim}{s} \in \mathbb{R}_+^{N_s \times 2 \times N \times R}$ delle catture:

$$\underset{\sim}{s}_{\cdot,r}^{n,a} \equiv \mathcal{S}_{\cdot,r}^{n\Delta s,a}, \quad \forall n, \forall a, \forall r,$$

dove $\mathcal{S}_{\cdot,r}^{n\Delta s,a}$ va inteso come il vettore delle taglie predette al passo $n\Delta s$-esimo, nell'r-esima simulazione esatta, se $a = 1$, o approssimata, se $a = 2$; successivamente si convolve $\underset{\sim}{s}$ rispetto all'indice del tempo:

$$s_{i,r}^{n,a} \equiv \frac{1}{\Delta f} \sum_{j=1}^{\Delta f} \underset{\sim}{s}_{i,r}^{(n-1)\Delta f + j,a}, \quad \forall n, \forall a, \forall i, \forall r,$$

vale a dire si mediano ogni Δf elementi delle N_s catture. Tale mollificazione è necessaria per rendere $\mathbf{s}$ meno rumoroso rispetto a $\underset{\sim}{s}$ e quindi piú leggibile una volta raffigurato.

Per quanto riguarda gl'istanti temporali considerati, si hanno tre forme a seconda di come s'intende l'indice n:

$$\begin{aligned} t_n &= n\Delta t, & \forall n \in \{1,2,\ldots,N_t\}, \\ t_n^s &= t_n \Delta s, & \forall n \in \{1,2,\ldots,N_s\}, \\ t_n^f &= t_n^s \Delta f, & \forall n \in \{1,2,\ldots,N_f\}, \end{aligned}$$

rispettivamente per i tempi discretizzati, campionanti e ridotti; a prescindere vale comunque $t_{N_t} = t_{N_s}^s = t_{N_f}^f = T$.

Osservazione 4.4. L'indice n in $\mathbf{s}$ non include 0 perché fa riferimento alla distribuzione iniziale, sempre uguale a $\mathcal{S}^0$ dall'Alg. 1; se si volesse però estendere il tensore, si può scrivere $\mathbf{s}_{\cdot,r}^{0,a} \equiv \mathcal{S}^0$, $\forall a, \forall r$.

Ipotesi 4.3. In questa trattazione si considerano $\Delta t = 0.01$, $N_s = 1000$, $N_f = 50$ mentre N_t viene scelto per avere t_n poco superiore al tempo di convergenza, se i risultati convergono, ma sicuramente è un multiplo di 10 per garantire che N_s sia un suo divisore.

Osservazione 4.5. Per la maggior parte dei grafici si considera esclusivamente la distribuzione al tempo finale T *senza* convoluzione temporale, cosicché, per leggerezza di notazione, si può impropriamente denotare con $\mathbf{s}^T \in \mathbb{R}_+^{2\times N\times R}$ il tensore del terz'ordine $\mathbf{s}^T \equiv \underset{\sim}{\mathbf{s}}^{N_s}$.

4.2.3 Definizione dei grafici

Istogrammi

I primi grafici considerano gl'istogrammi [normalizzati] della distribuzione al tempo finale T. Si inizia considerando il massimo e il minimo elemento del tensore $\mathbf{s}^T$,

$$s_{max} \equiv \max_{a,i,r} s_{i,r}^{T,a} \quad \text{e} \quad s_{min} \equiv \min_{a,i,r} s_{i,r}^{T,a},$$

coi quali si può definire una griglia comune equispaziata su cui costruire le classi degl'istogrammi. Sia N_c il numero di intervalli/classi della griglia, allora definendo

$$\mathcal{B}\colon \mathbb{R}^N \to \mathbb{R}^{N_c}$$

come la funzione che restituisce i valori [normalizzati] delle classi dell'istogramma, il tensore $\mathbf{b} \in \mathbb{R}_+^{2\times N_c\times R}$ si scrive

$$\mathbf{b}_{\cdot,r}^a \equiv \mathcal{B}(\mathbf{s}_{\cdot,r}^{T,a}), \ \ \forall a \text{ e } \forall r. \tag{4.3}$$

Valutati gl'intervalli di confidenza

$$IC_R^\alpha(\mathbf{b}_c^a) = \bar{b}_c^a \pm t_{R-1}^{\alpha/2}\sqrt{\frac{\nu_c^a}{R}}, \ \ \forall a \text{ e } \forall c,$$

si può rappresentare con $\bar{\mathbf{b}}^a$ l'istogramma medio della simulazione esatta e approssimata, e con $\pm t_{R-1}^{\alpha/2}\sqrt{\nu_c^a/R}$ l'errore stocastico sulle classi.

Lognormale bimodale

La logica è simile agl'istogrammi: siano

$$\mathcal{L}(\cdot;\mathbf{s}_{\cdot,r}^{T,a})\colon \mathbb{R}_+ \to \mathbb{R}_+,$$

la densità di una distribuzione bilognormale (1.4) fittata dal vettore $\mathbf{s}_{\cdot,r}^{T,a}$, e

$$\mathcal{L}(\cdot;\mathbf{s}^{T,a})\colon \mathbb{R}_+ \to \mathbb{R}_+^R$$

la funzione vettoriale tale che

$$\mathcal{L}(x;\mathbf{s}^{T,a}) \equiv \left[\mathcal{L}(x;\mathbf{s}_{\cdot,1}^{T,a}) \quad \mathcal{L}(x;\mathbf{s}_{\cdot,2}^{T,a}) \quad \cdots \quad \mathcal{L}(x;\mathbf{s}_{\cdot,R}^{T,a})\right]^\top,$$

che in essenza raccoglie puntualmente tutti gli adattamenti in un unico vettore. Allora gl'intervalli di confidenza hanno forma

$$IC_R^\alpha(\mathcal{L}(x;\mathbf{s}^{T,a})) = \bar{\mathcal{L}}^a(x) \pm t_{R-1}^{\alpha/2}\sqrt{\frac{\nu^a(x)}{R}}, \ \ \forall x \in [s_{min}, s_{max}] \text{ e } \forall a,$$

in cui, come prima, $\bar{\mathcal{L}}^a(x)$ è la funzione media mentre $\pm t_{R-1}^{\alpha/2}\sqrt{v^a(x)/R}$ la sua incertezza stocastica.

Osservazione 4.6. Essendo x continuo, l'insieme degl'intervalli di confidenza forma per le funzioni un fascio di confidenza.

Osservazione 4.7. L'intervallo di confidenza è applicato tanto all'adattamento intero della bilognormale quanto ai suoi cinque parametri ξ, μ_1, σ_1, μ_2 e σ_2, di cui si riporta la forma solo del primo per brevità: sia allora $\boldsymbol{\xi} \in \mathbb{R}^{2\times R}$ la matrice di tutte le frazioni ricavate nelle simulazioni, allora vale

$$\mathrm{IC}_R^{\alpha}(\boldsymbol{\xi}) = \bar{\xi}^a \pm t_{R-1}^{\alpha/2}\sqrt{\frac{v^a}{R}}.$$

Funzione di ripartizione complementare empirica

La funzione di ripartizione complementare (FRC) empirica è definita nella coda della distribuzione della popolazione, la quale però va definita: in questo caso si sceglie come soglia l'ultimo quartile, ossia quel valore $s_{3/4}$ tale che $\mathbb{P}(S \leqslant s_{3/4}) = 3/4$, data una qualunque variabile aleatoria S; ciò corrisponde nella pratica a trovare quell'elemento $s_{3/4}$ del tensore $\mathbf{s}_{\cdot,r}^{T,a}$ tale che

$$\left|\left\{ s_{i,r}^{T,a} \geqslant s_{3/4} \mid i \in \{1,2,\ldots,N\}\right\}\right| = \left\lceil \frac{N}{4} \right\rceil, \quad \forall a \text{ e } \forall r,$$

che a parole vuol dire che *almeno* $1/4$ di tutt'i valori di $\mathbf{s}_{\cdot,r}^{T,a}$ sono superiori a $s_{3/4}$. Allora l'intervallo $[s_{3/4}, s_{\max}]$ caratterizza la coda.

Con ciò la FRC empirica,

$$\mathcal{R}_{\mathcal{E}}(\cdot;\mathbf{s}_{\cdot,r}^{T,a}) \colon \mathbb{R}_+ \to \mathbb{R}_+,$$

che approssima la vera FRC $\mathbb{P}(S > x)$, si scrive

$$\mathcal{R}_{\mathcal{E}}(x;\mathbf{s}_{\cdot,r}^{T,a}) \equiv 1 - \frac{1}{N}\sum_{i=1}^{N} \chi_{[0,x]}\left(s_{i,r}^{T,a}\right), \quad \forall a \text{ e } \forall r, \tag{4.4}$$

dove $\chi_{[0,x]} \colon \mathbb{R} \to \{0,1\}$ è la funzione indicatrice dell'insieme $[0,x]$:

$$\chi_{[0,x]}(w) \equiv \begin{cases} 1 & \text{se } w \in [0,x], \\ 0 & \text{altrimenti.} \end{cases}$$

Tuttavia, siccome la (4.4) dev'essere valutata in $x \in [s_{3/4}, s_{\max}]$, e in particolare nel tensore $\mathbf{p} \in \mathbb{R}^{2\times N_{1/4}\times R}$ di $\mathbf{s}$ ristretto alla coda

$$p_{i,r}^{a} \equiv s_{N-N_{1/4}+i,r}^{T,a}, \quad \forall a, \forall i \in \{1,2,\ldots,N_{1/4}\} \text{ e } \forall r$$

nel quale $N_{1/4} \equiv \lceil N/4 \rceil$, vale il seguente riscalamento:

$$\begin{aligned}\mathcal{R}_{\mathcal{E}}(x;\mathbf{s}_{\cdot,r}^{T,a}) &= 1 - \frac{N - N_{1/4}}{N} - \frac{1}{N}\sum_{i=1}^{N_{1/4}} \chi_{[0,x]}\left(s_{N-N_{1/4}+i,r}^{T,a}\right) \\ &= \frac{N_{1/4}}{N}\left[1 - \frac{1}{N_{1/4}}\sum_{i=1}^{N_{1/4}} \chi_{[0,x]}\left(p_{i,r}^{T,a}\right)\right] = \frac{N_{1/4}}{N}\mathcal{R}_{\mathcal{E}}(x;\mathbf{p}_{\cdot,r}^{a}), \quad \forall a \text{ e } \forall r,\end{aligned}$$

Vale a dire che $\mathcal{R}_{\mathcal{E}}(x;\mathbf{p}_{\cdot,r}^{a})$ è uguale a $\mathcal{R}_{\mathcal{E}}(x;\mathbf{s}_{\cdot,r}^{T,a})$ a meno di un riscalamento $N_{1/4}/N$. Tuttavia l'indice di Pareto è invariante rispetto ai riscalamenti:

Osservazione 4.8. In scala logaritmica la (1.2) si scrive

$$\log(\mathcal{R}(s)) \approx -\beta \log(s) + \log(c),$$

che equivale a una retta con pendenza β e intercetta $\log(c)$; indicando con $\mathcal{R}'(s)$ il rapporto $\mathcal{R}(s)/c$ segue

$$\log(\mathcal{R}'(s)) \equiv \log(\mathcal{R}(s)) - \log(c) \approx -\beta \log(s),$$

per la quale l'indice di Pareto β è invariato.

Da quest'osservazione si può pertanto scegliere $\mathcal{R}_{\mathcal{E}}(x; \mathbf{s}^{T,a}_{\cdot,r})$ in scala logaritmica per maggiore leggibilità, proprio perché ciò a cui si è piú interessati non sono i valori bensí la loro tendenza racchiusa nel parametro β. L'uso di tale scala introduce però un lieve problema: l'ultimo elemento $p^a_{N_{1/4},r}$ ha ordinata nulla ed è dunque non visibile; per ovviare il problema si può traslare la FRC empirica in scala lineare

$$\mathcal{R}_{\mathcal{E}}(x; \mathbf{p}^a_{\cdot,r}) + \frac{1}{2N_{1/4}}, \quad \forall a \text{ e } \forall r, \tag{4.5}$$

ciò corrisponde in scala logaritmica a una traslazione non uniforme dei valori che quindi vengono falsati, seppure di poco essendo $1/2N_{1/4}$ una quantità piccola; per tale ragione questa modifica è valida *solo graficamente* e non per il calcolo dell'indice di Pareto, ricavato sempre dal tensore $\mathbf{p}$.

Di conseguenza, nel complesso, la FRC empirica è raffigurata in scala logaritmica dal grafico a dispersione delle coppie

$$\left(\frac{N}{N_{1/4}} \mathcal{R}_{\mathcal{E}}(\bar{p}^a_i; \mathbf{s}^{T,a}_{\cdot,r}) + \frac{1}{2N_{1/4}}, \bar{p}^a_i \right) = \left(\mathcal{R}_{\mathcal{E}}(\bar{p}^a_i; \mathbf{p}^a_{\cdot,r}) + \frac{1}{2N_{1/4}}, \bar{p}^a_i \right), \quad \forall a \text{ e } \forall i,$$

ove $\bar{p}^a_i$ viene dagl'intervalli di confidenza

$$IC^{\alpha}_R(\mathbf{p}^a_i) = \bar{p}^a_i \pm t^{\alpha/2}_{R-1} \sqrt{\frac{\nu^a_i}{R}}, \quad \forall a \text{ e } \forall i.$$

Osservazione 4.9. Si noti che, contrariamente ai casi precedenti, l'intervallo di confidenza è orizzontale e non verticale; difatti l'immagine della FRC empirica è invariata rispetto alle simulazioni che modificano solo le popolazioni dei centri maggiori, ossia l'ascissa della FRC empirica.

FRC della Pareto e relativo indice

Per quanto riguarda il fittaggio della distribuzione di Pareto il ragionamento è analogo a quello visto per la lognormale bimodale: sia

$$\mathcal{R}_{\mathcal{P}}(\cdot; \mathbf{p}^a_{\cdot,r}) \colon \mathbb{R}_+ \to \mathbb{R}_+, \quad \forall a \text{ e } \forall r,$$

la FRC della densità della distribuzione di Pareto, analoga alla (1.2), fittata dal vettore $\mathbf{p}^a_{\cdot,r}$ e sia

$$\mathcal{R}_{\mathcal{P}}(\cdot; \mathbf{p}^a) \colon \mathbb{R}_+ \to \mathbb{R}_+$$

la funzione vettoriale tale che

$$\mathcal{R}_{\mathcal{P}}(x; \mathbf{p}^a) \equiv \left[\mathcal{R}_{\mathcal{P}}(x; \mathbf{p}^a_{\cdot,1}) \quad \mathcal{R}_{\mathcal{P}}(x; \mathbf{p}^a_{\cdot,2}) \quad \cdots \quad \mathcal{R}_{\mathcal{P}}(x; \mathbf{p}^a_{\cdot,R}) \right]^{\top}.$$

Allora gl'intervalli di confidenza hanno forma

$$\mathrm{IC}_R^\alpha\left(\frac{N}{N_{1/4}}\mathcal{R}_{\mathcal{P}}(x;\mathbf{p}^a)\right)=\bar{\mathcal{R}}^a_{\mathcal{P}}(x)\pm t^{\alpha/2}_{R-1}\sqrt{\frac{\nu^a(x)}{R}},\quad \forall x\in[s_{3/4},s_{max}]\text{ e }\forall a,$$

ove $N/N_{1/4}$ è il fattore di riscalamento descritto poco fa ed è necessario per confrontare l'adattamento colla FRC empirica.

Osservazione 4.10. Come nell'Oss. 4.7, l'intervallo di confidenza è applicato tanto alla FRC della Pareto quanto al suo parametro β: sia allora $\boldsymbol{\beta}\in\mathbb{R}^{2\times R}$ la matrice di tutti gl'indici di Pareto ricavati nelle simulazioni, allora vale

$$\mathrm{IC}_R^\alpha(\boldsymbol{\beta})=\bar{\beta}^a\pm t^{\alpha/2}_{R-1}\sqrt{\frac{\nu^a}{R}}.$$

FRC della lognormale bimodale

La bilogonormale non è solo necessaria per l'intera distribuzione ma anche per la coda: siano

$$\mathcal{R}_{\mathcal{L}}(\cdot;\mathbf{p}^a_{\cdot,r}),\ \mathcal{R}_{\mathcal{B}}(\cdot;\mathbf{p}^a_{\cdot,r})\colon\mathbb{R}_+\to\mathbb{R}_+,\quad\forall a\text{ e }\forall r,$$

le FRC rispettivamente della densità della distribuzione lognormale e bilognormale adattate al vettore $\mathbf{p}^a_{\cdot,r}$; allora per linearità della FRC, indotta dalla linearità dell'integrale, dalla (1.4) vale

$$\mathcal{R}_{\mathcal{B}}(x;\mathbf{p}^a_{\cdot,r})=\xi\mathcal{R}_{\mathcal{L}}(x;\mathbf{p}^a_{\cdot,r})+(1-\xi)\mathcal{R}_{\mathcal{L}}(x;\mathbf{p}^a_{\cdot,r}),\quad\forall a\text{ e }\forall r.$$

Pertanto, come prima, si può definire

$$\mathcal{R}_{\mathcal{B}}(\cdot;\mathbf{p}^a)\colon\mathbb{R}_+\to\mathbb{R}_+$$

la funzione vettoriale tale che

$$\mathcal{R}_{\mathcal{B}}(x;\mathbf{p}^a)\equiv\begin{bmatrix}\mathcal{R}_{\mathcal{B}}(x;\mathbf{p}^a_{\cdot,1}) & \mathcal{R}_{\mathcal{B}}(x;\mathbf{p}^a_{\cdot,2}) & \cdots & \mathcal{R}_{\mathcal{B}}(x;\mathbf{p}^a_{\cdot,R})\end{bmatrix}^\top,$$

che porta agl'intervalli di confidenza

$$\mathrm{IC}_R^\alpha\left(\frac{N}{N_{1/4}}\mathcal{R}_{\mathcal{B}}(x;\mathbf{p}^a)\right)=\bar{\mathcal{R}}^a_{\mathcal{B}}(x)\pm t^{\alpha/2}_{R-1}\sqrt{\frac{\nu^a(x)}{R}},\quad\forall x\in[s_{3/4},s_{max}]\text{ e }\forall a.$$

Trasformazione dei dati

Come riportato nella Fig. 1.1, se dei dati $\mathbf{s}$ sono distribuiti seguendo una lognormale allora la loro riformulazione $\mathbf{h}\in\mathbb{R}_+^{N_f\times 2\times N\times R}$ in spazio logaritmico

$$h^{n,a}_{i,r}\equiv\log(s^{n,a}_{i,r}),\quad\forall n,\ \forall a,\ \forall i,\ \forall r,$$

è distribuita secondo una normale; il vantaggio di usare $\mathbf{h}$ anziché $\mathbf{s}$ si trova nella piú facile interpretabilità dei dati, soprattutto quando si variano i parametri, e quindi negli studi parametrici.

Questa trasformazione viene pertanto applicata ai grafici finora definiti i quali cambiano nel seguente modo: innanzitutto gl'istogrammi si calcolano come prima ma coi dati riformulati, per cui la (4.3) diventa

$$\mathbf{b}^a_{\cdot,r}\equiv\mathcal{B}(\mathbf{h}^{T,a}_{\cdot,r}),\quad\forall a\text{ e }\forall r,$$

e similmente per gl'intervalli di confidenza. È altrettanto rilevante far notare che nello spazio $h \equiv \log(s)$ anche la densità $\bar{f}_N(s) \equiv \bar{f}_N(s,T)$ si trasforma[4] pur preservando la probabilità totale; il che si traduce matematicamente in

$$dP = \bar{f}_N(s)ds = \bar{l}_N(h)dh = \bar{l}_N(h)\frac{ds}{s\ln(10)} \implies \bar{l}_N(h) = 10^h \ln(10)\bar{f}_N(10^h),$$

dove dP è la variazione di probabilità relativa alle variazioni ds/dh in punti generici s/h, mentre $\bar{l}_N(h)$ è la trasformazione di $\bar{f}_N(s)$ nel nuovo spazio h.

D'altra parte, per quanto riguarda le FRC empirica, di Pareto e bilognormale, esse rimangono invariate e vengono semplicemente rimappate dall'intervallo $[s_{min}, s_{max}]$ ad $[h_{min}, h_{max}]$, con $h_{min} = \log(s_{min})$ e $h_{max} = \log(s_{max})$. Ciò deriva dal fatto che il logaritmo è una funzione strettamente crescente, dunque preserva l'ordine dei dati trasformati:

$$\mathbb{P}(X < x) = \mathbb{P}(\log(X) < \log(x)), \quad \forall x \in \mathbb{R}_+,$$

data una generica variabile aleatoria $X \in \mathbb{R}_+$, vale

Osservazione 4.11. Dall'Oss. 4.8, trasformando s in h, la retta diventa

$$\log(\mathcal{R}(h)) \approx -\beta h + \log(c) \quad \text{o} \quad \log(\mathcal{R}'(h)) \approx -\beta h,$$

la quale è la stessa relazione ma in scala semilogaritmica, che quindi sarà quella usata per tutt'i grafici relativi alle FRC.

Evoluzione della taglia media

Esprimendo con $\bar{\mathbf{s}} \in \mathbb{R}_+^{N_f \times 2 \times R}$ il tensore delle taglie medie rispetto ai nodi

$$\bar{s}_r^{n,a} \equiv \frac{1}{N}\sum_{i=1}^{N} s_{i,r}^{n,a}, \quad \forall n, \forall a \text{ e } \forall r,$$

dove gl'intervallo di confidenza di $\bar{\mathbf{s}}$ sono

$$IC_R^{\alpha}(\bar{\mathbf{s}}^{n,a}) = \bar{\bar{s}}^{n,a} \pm t_{R-1}^{\alpha/2}\sqrt{\frac{v^{n,a}}{R}}, \quad \forall n \text{ e } \forall a,$$

allora l'andamento della taglia media totale è raffigurato interpolando linearmente le coppie

$$\left(t_n^f, \bar{\bar{s}}^{n,a}\right), \quad \forall n \text{ e } \forall a,$$

in istanti temporali contingui con fascio di confidenza $\pm t_{R-1}^{\alpha/2}\sqrt{v^{n,a}/R}$.

Taglie medie contro gradi

S'indichi ora con $\bar{\mathbf{s}} \in \mathbb{R}_+^{2 \times R}$ il tensore delle taglie medie rispetto alle simulazioni

$$\bar{s}_i^a \equiv \frac{1}{R}\sum_{r=1}^{R} s_{i,r}^{T,a}, \quad \forall a \text{ e } \forall i,$$

dunque dalle coppie

$$(k_i, \bar{s}_i^a), \quad \forall a \text{ e } \forall i,$$

si può definire il grafico a dispersione che lega la taglia media tra tutte le simulazioni di un nodo al suo grado.

4 Piú nel dettaglio, come già accennato all'inizio, tende a una densità di una gaussiana.

Osservazione 4.12. In questo caso non si disegnano gl'intervalli di confidenza perché il grafico è già molto denso e aggiungere ulteriori barre verticali/orizzontali lo renderebbe difficilmente leggibile.

Evoluzioni delle taglie medie delle classi dei gradi

Si denoti con $\bar{s} \in \mathbb{R}^{N_f \times 2 \times |\mathcal{K}|}$ il tensore

$$\bar{s}_k^{n,a} \equiv \frac{1}{|\mathcal{I}_k|} \sum_{i \in \mathcal{I}_k} \frac{1}{R} \sum_{r=1}^{R} s_{i,r}^{n,a}, \quad \forall n, \forall a \text{ e } \forall k,$$

ove $\mathcal{K} \subset \{1,2,\ldots,N\}$ è l'insieme dei gradi distinti di $\mathbf{k}$, mentre $\mathcal{I}_k$ dalla (3.30) rappresenta l'insieme degl'indici con grado k; pertanto interpolando linearmente le coppie

$$\left(t_n^f, \bar{s}_k^{n,a}\right), \quad \forall n, \forall a \text{ e } \forall k,$$

a istanti temporali contigui, si può rappresentare l'andamento temporale della popolazione media della classe k-esima fra tutte le simulazioni. Questo grafico viene quindi usato per accertarsi di non essere piú nel transitorio e di aver quindi raggiunto uno stato stazionario.

Si conclude notando che non si disegnano gl'intervalli di confidenza per la medesima ragione chiarita nell'Oss. 4.12.

4.2.4 Sulle fluttuazioni γ

Nelle [REI] è stato introdotto per completezza il termine γ legato alle fluttuazioni, ossia a quei fenomeni di morte e nascita che caratterizzano la naturale variazione di una popolazione intorno a una media. Si argomenta adesso che è possibile trascurarle per due ragioni:

1. si è interessati solo alla predizione della distribuzione stazionaria da confrontare con quella reale del 1991 e
2. dal vincolo (3.24) le uniche perturbazioni simmetriche sono per loro natura piccole, essendo concentrate in un intorno dell'origine.

Nella Fig. 4.1 sono riportati dei risultati simulati coi parametri nella Tab. 4.1 e colla regola d'interazione

$$E(s,s_*,i,i_*) \equiv (1-\zeta)\lambda \frac{(s_*/s)(w_{i_*}/w_i)}{\alpha + (s_*/s)(w_{i_*}/w_i)} + \zeta\lambda \frac{(s/s_*)(w_i/w_{i_*})}{\alpha + (s/s_*)(w_i/w_{i_*})}, \tag{4.6}$$

la quale è spiegata implicitamente nel corso del § 4.3 ed è brevemente discussa piú nel dettaglio nel § 4.4.1. Nel complesso confermano che γ ha solo due principali effetti: aumenta leggermente l'incertezza stocastica e la varianza; perciò introduce solo del lieve rumore che, sebbene abbia il nobile scopo di rendere piú realistico il modello, non aiuta a interpretare i risultati se confrontati con quelli reali.

Tabella 4.1: Parametri della Fig. 4.1.

	λ	α	ζ	σ	N_t
Senza γ	0.1	1	0.1	–	10^6
Con γ	0.1	1	0.1	0.05	10^6

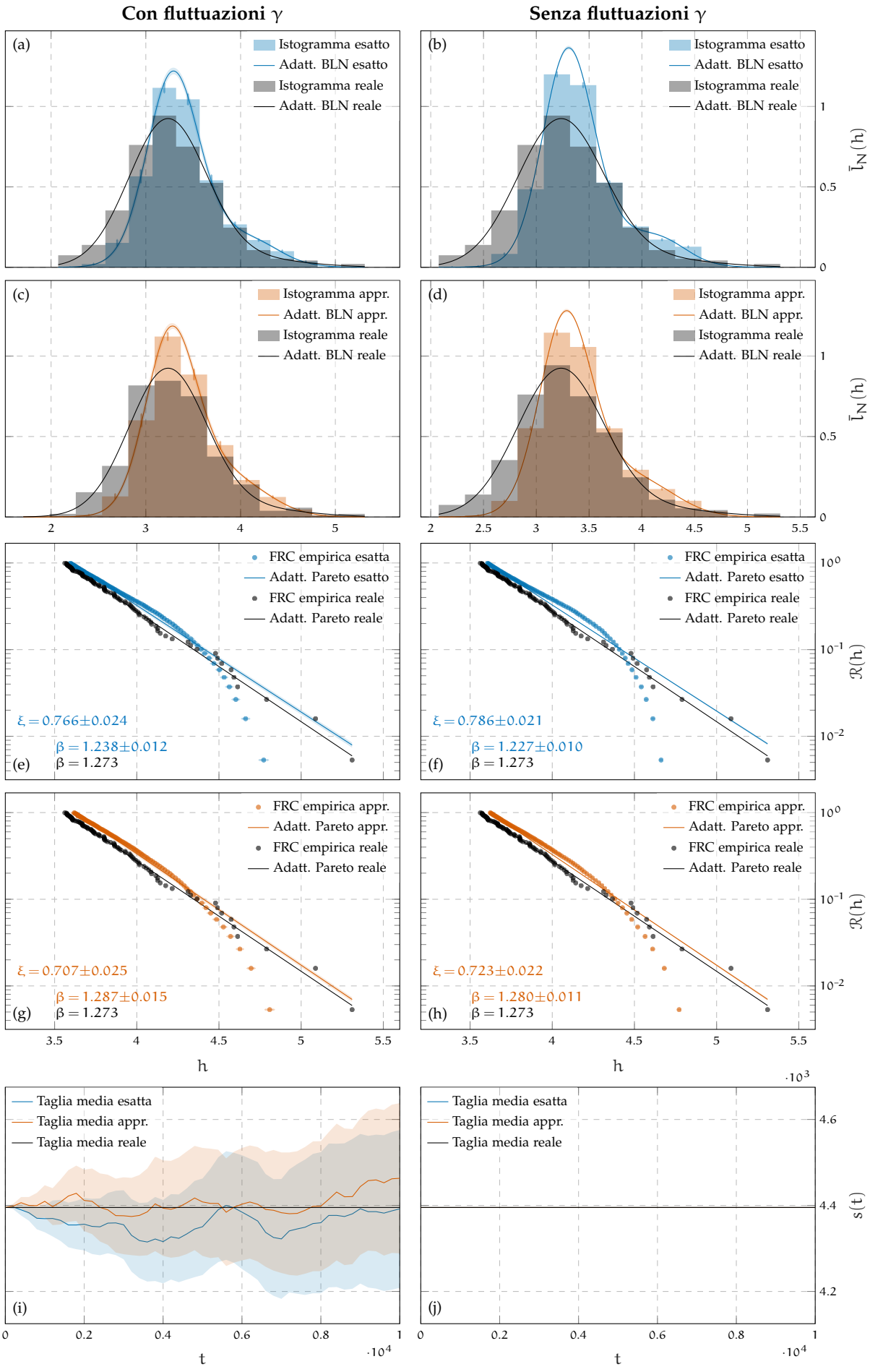

Figura 4.1: Confronto di una simulazione con e senza fluttuazioni; la regola d'emigrazione è la (4.6) mentre i parametri sono illustrati nella Tab. 4.1.

4.3 REGOLE D'EMIGRAZIONE

In questo paragrafo si specializza la regola d'emigrazione E definita nelle [REI] per completare le $[RI]_S$; tutte le forme qui presenti condividono comunque tre aspetti: sono non simmetriche, non lineari e vedono i microstati (i,s) e (i_*,s_*) rispettivamente come la città interagente e ricevente.

Per brevità si limita lo studio a venire su tre aspetti:

1. si considera solo la regione della Sardegna sia perché molti risultati di [6] sono già stati riprodotti nel § 2.3, sia poiché per le altre 19 regioni sono stati trovati risultati analoghi;
2. si analizza esclusivamente la configurazione che meglio adatta i dati reali, almeno tra quelle esplorate dall'autore, e
3. si approfondiscono, vista la loro lunghezza, due soli studi parametrici per l'ultima regola d'emigrazione proposta.

Pertanto, data l'importanza della Sardegna, si elencano nella Tab. 4.2 i dati principali della regione assieme ai parametri fittati della Pareto e della lognormale bimodale.

Tabella 4.2: Dati [12, 14] e parametri della Sardegna nel 1991.

N	P	β	ξ	μ_1	σ_1	μ_2	σ_2
375	$\approx 1.648\times10^6$	1.273	0.929	3.227	0.408	4.099	0.556

Cionnonostante, nel § 4.4.2 si studiano cursoriamente comunque la Valle d'Aosta e l'Italia essendo casi estremi rispetto al numero di nodi.

Osservazione 4.13. Nel corrente paragrafo con minori, medi e maggiori si fa riferimento ai centri rispettivamente con grado piccolo, medio e alto, come mostrato nelle sottofigure (j)-(l) delle Figg. 4.2–4.6.

4.3.1 Regola taglia

Una prima possibilità[5] consiste nella

$$E(s,s_*) \equiv \lambda \frac{(s_*/s)^\alpha}{1+(s_*/s)^\alpha}, \qquad [RE]_S$$

in cui $\lambda \in (0,1)$ (Ip. 3.5) e $\alpha \in \mathbb{R}^+$. La logica è che il tasso d'emigrazione verso la città ricevente è tanto maggiore quanto piú grande è la sua popolazione relativa, data dal rapporto s_*/s, rispetto alla città interagente.

In totale sono state studiate due configurazioni di riferimento descritte dai parametri nella Tab. 4.3, mentre i risultati sono illustrati nelle Figg. 4.2 e 4.3; quest'ultimi descrivono due comportamenti:

- ◇ uno stabile ma non distante dalla distribuzione iniziale uniforme e
- ◇ uno instabile in cui *in media* si spopolano i centri medi.

5 È una funzione di Hill di ordine α, parzialmente ispirata dalle [10, (2.2), § 2, p. 223, e (4.5), § 4, p. 228], ampiamente applicata nel campo della Biomedicina.

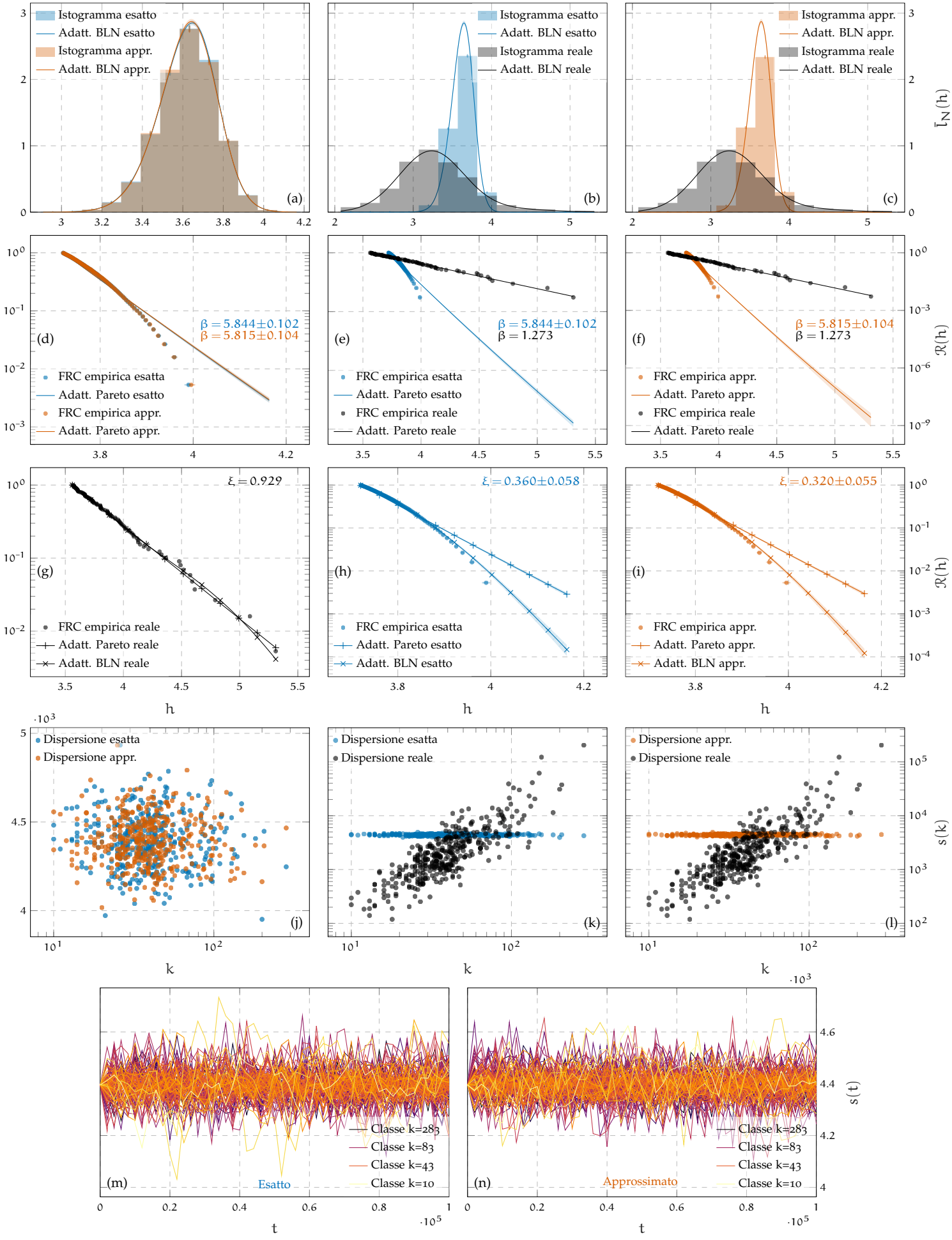

Figura 4.2: Studio della configurazione di riferimento stabile della $[RE]_S$ con parametri dalla Tab. 4.3; per la spiegazione dei grafici si veda il § 4.2.3.

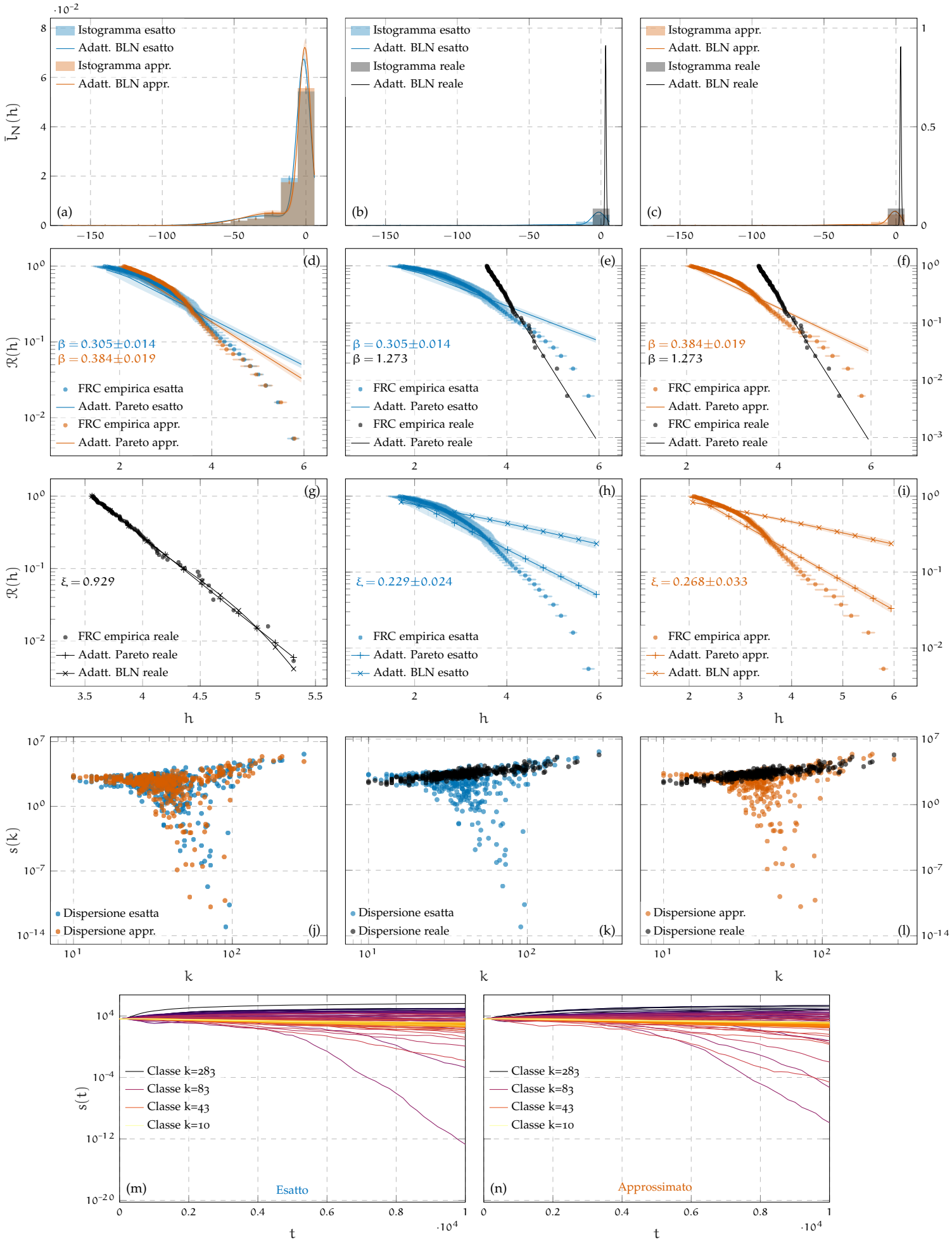

Figura 4.3: Studio della configurazione di riferimento instabile della $[\mathrm{RE}]_S$ con parametri dalla Tab. 4.3; per la spiegazione dei grafici si veda il § 4.2.3.

Tabella 4.3: Parametri delle Figg. 4.2 e 4.3.

	λ	α	N_t	R
Stabile	0.1	0.5	10^7	100
Instabile	0.1	1.5	10^6	10

Si sottolinea in media poiché nelle singole realizzazioni è molto piú probabile che si spopolino i centri maggiori mentre su piú realizzazioni si manifesta almeno una con taglia elevata, bilanciando cosí la media; questa è anche la ragione per cui si sono considerate solo 10 iterazioni anziché le usuali 100 nella Tab. 4.3 visto che la media tende a mascherare lo spopolamento, il quale però è inevitabile ma semplicemente piú lento. Oltre alle Figg. 4.3(j)-4.3(l), gl'istogrammi nelle Figg. 4.3(a)-4.3(c) confermano anche questa tendenza sotto forma di una coda che allunga la distribuzione nei valori negativi, corrispondenti agli esponenti delle città che si spopolano.

Piú in generale si è verificato, seppure solo empiricamente, che per ogni λ esiste un $\bar{\alpha} \in \mathbb{R}_+$ tale che $\alpha < \bar{\alpha}$ risulta in una configurazione stabile mentre $\alpha \geqslant \bar{\alpha}$ ne restituisce una instabile.

4.3.2 Regola taglia-grado

Al fine di correggere la $[\mathrm{RE}]_S$ si può introdurre anche il rapporto relativo dei gradi:

$$E_{SK}(s,s_*,i,i_*) \equiv \lambda \frac{[(s_*/s)(k_{i_*}/k_i)]^\alpha}{1+[(s_*/s)(k_{i_*}/k_i)]^\alpha}. \qquad [\mathrm{RE}]_{SK}$$

Cosí la logica diventa che il tasso d'emigrazione verso la città ricevente è tanto maggiore quanto piú popolosa e connessa è rispetto alla città interagente.

Tabella 4.4: Parametri della Fig. 4.4.

λ	α	N_t
0.15	0.63	3×10^6

La configurazione di riferimento studiata è descritta dai parametri nella Tab. 4.4, mentre i risultati sono illustrati nella Fig. 4.4. Nel complesso sono migliori di quelli ottenuti mediante la $[\mathrm{RE}]_S$:

- le Figg. 4.4(a), 4.4(d) e 4.4(j) illustrano che l'approssimazione corrisponde quasi perfettamente alla simulazione esatta;
- nelle Figg. 4.4(b) e 4.4(c) le distribuzioni simulate sono abbastanza simili a quelle reale, mentre nelle Figg. 4.4(e) e 4.4(f) gl'indici di Pareto sono ragionevolmente vicini;
- a eccezione del caso reale in cui coincidono, nelle Figg. 4.4(g)-4.4(i) la bilognormale segue meglio la coda verso la fine, sebbene coincida colla Pareto nella prima metà;
- nelle Figg. 4.4(k) e 4.4(l) le dispersioni presentano un'analoga correlazione positiva ma piú stretta rispetto alla reale.

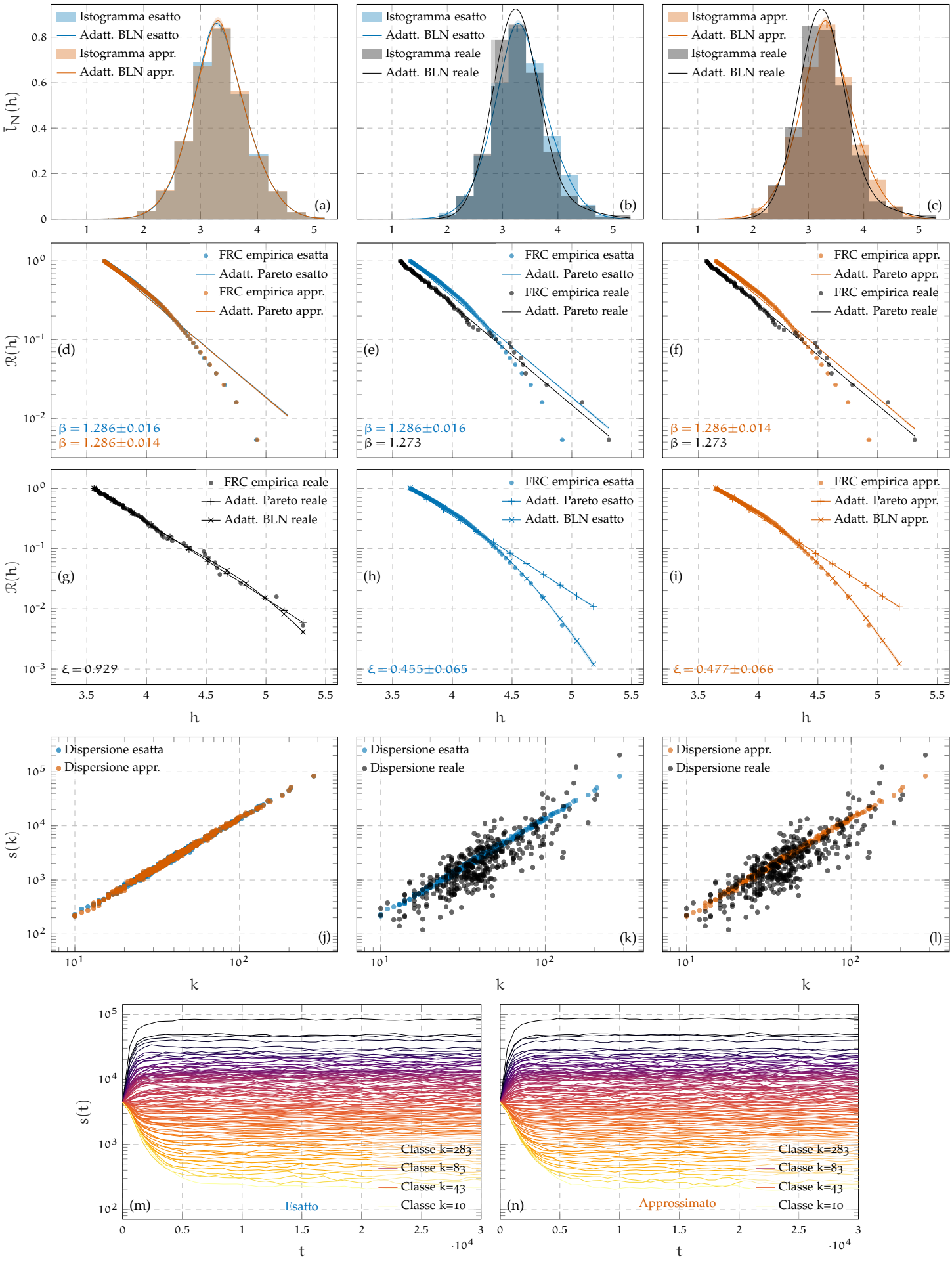

Figura 4.4: Studio della configurazione di riferimento della $[\mathrm{RE}]_{SK}$ con parametri dalla Tab. 4.4; per la spiegazione dei grafici si veda il § 4.2.3.

4.3.3 Regola frazionata

La $[\mathrm{RE}]_{SK}$ si può poi modificare introducendo un ulteriore parametro ζ

$$E^{f}_{SK}(s,s_*,i,i_*) \equiv (1-\zeta)E_{SK}(s,s_*,k_i,k_{i_*}) + \zeta E_{SK}(s_*,s,k_{i_*},k_i), \qquad [\mathrm{RE}]^{f}_{SK}$$

con $\zeta \in (0,1)$, che in essenza descrive la frazione di popolazione che si comporta seguendo una regola inversa alla $[\mathrm{RE}]_{SK}$, vale a dire che manifesta un tasso d'emigrazione verso la città ricevente tanto maggiore quanto piú piccola e sconnessa è rispetto alla città interagente.

Tabella 4.5: Parametri della Fig. 4.5.

λ	α	ζ	N_t
0.15	0.67	0.01	3×10^6

La configurazione di riferimento analizzata è descritta dai parametri nella Tab. 4.5, mentre i relativi risultati sono mostrati nella Fig. 4.4. Questi migliorano sotto alcuni aspetti la $[\mathrm{RE}]_{SK}$:

- le Figg. 4.5(a), 4.5(d) e 4.5(j) confermano di nuovo come l'approssimazione corrisponda quasi perfettamente alla simulazione esatta;
- nelle Figg. 4.5(b) e 4.5(c) le distribuzioni simulate si adattano meglio a quella reale, e non alterano la buona corrispondenza degl'indici di Pareto nelle Figg. 4.5(e) e 4.5(f);
- le Figg. 4.5(h) e 4.5(i) non si discostano in modo significativo dalla precedente configurazione;
- nelle Figg. 4.5(k) e 4.5(l) le dispersioni sono compatte quanto prima e manifestano ora un lieve assestamento verso le città minori.

4.3.4 Regola taglia-forza

Dalla riga ⊛ della Tab. 2.1 si potrebbe anche pensare di sostituire nella $[\mathrm{RE}]_{SK}$ le forze in luogo dei gradi:

$$E_{SW}(s,s_*,i,i_*) \equiv \lambda \frac{[(s_*/s)(w_{i_*}/w_i)]^{\alpha}}{1+[(s_*/s)(w_{i_*}/w_i)]^{\alpha}}, \qquad [\mathrm{RE}]_{SW}$$

intendendo cosí che si ha un tasso d'emigrazione verso la città ricevente tanto maggiore quanto piú popolosa e trafficata è rispetto alla città interagente.

Tabella 4.6: Parametri della Fig. 4.6.

λ	α	N_t
0.18	0.44	3×10^6

La configurazione di riferimento analizzata è descritta dai parametri nella Tab. 4.6, mentre i relativi risultati sono mostrati nella Fig. 4.6. Essi sono di fatto i migliori:

- le Figg. 4.6(a), 4.6(d) e 4.6(j) confermano che l'approssimazione corrisponde quasi perfettamente alla simulazione esatta;

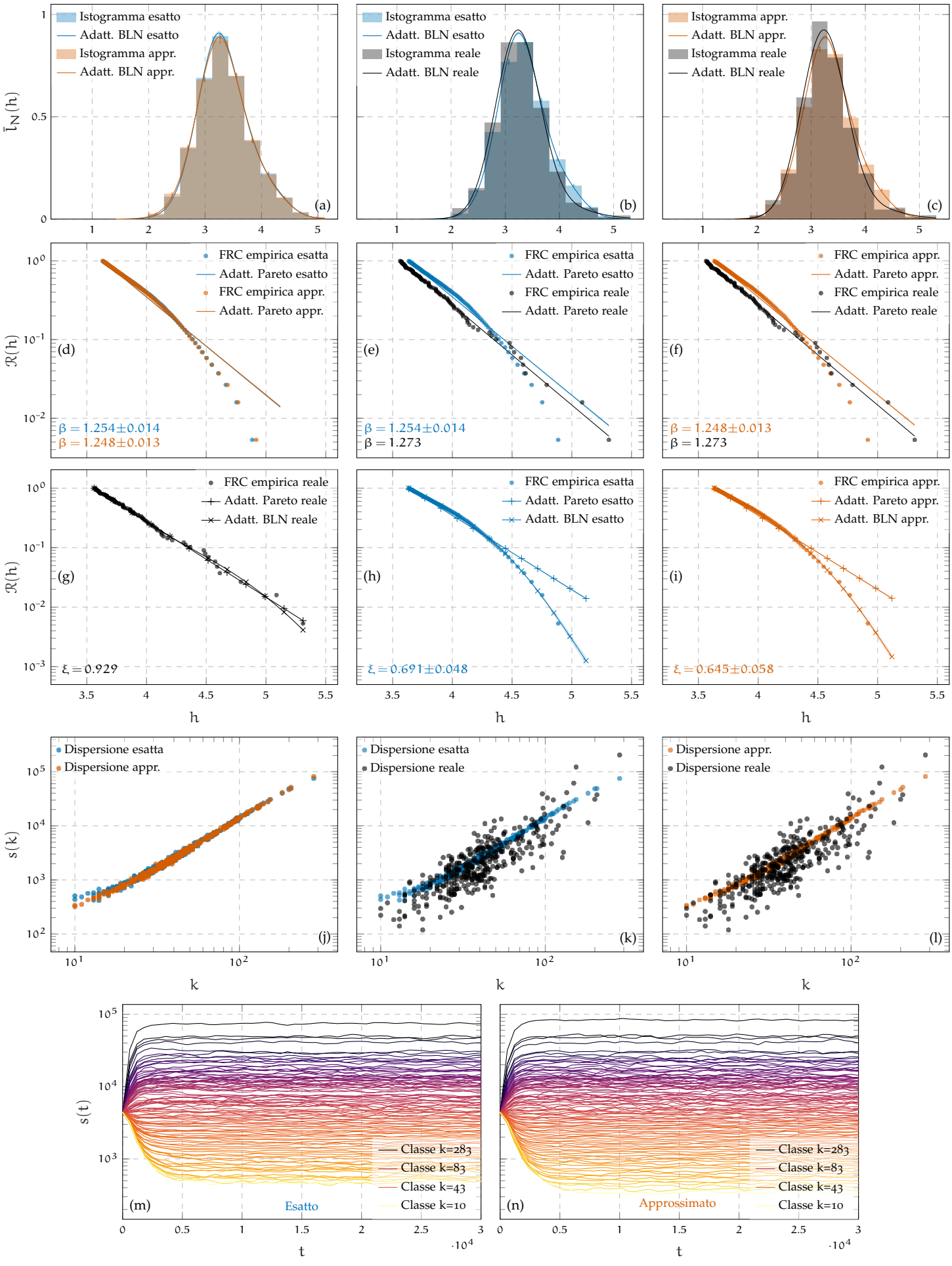

Figura 4.5: Studio della configurazione di riferimento della $[\mathrm{RE}]^{\mathrm{f}}_{\mathrm{SK}}$ con parametri dalla Tab. 4.5; per la spiegazione dei grafici si veda il § 4.2.3.

- ◇ nelle Figg. 4.6(b) e 4.6(c) le distribuzioni simulate sono quas'identiche a quella reale, e pure i relativi indici di Pareto nelle Figg. 4.6(e) e 4.6(f) corrispondono;
- ◇ le Figg. 4.6(h) e 4.6(i) non si discostano in modo significativo dalle precedenti due configurazioni;
- ◇ nelle Figg. 4.6(k) e 4.6(l) le dispersioni, oltre ad avere la stessa correlazione positiva rispetto alla reale, manifestano maggiore respiro cogliendo la differenziazione di taglia a parità di grado.

L'unico aspetto negativo si trova nelle Figg. 4.6(k) e 4.6(l): la regola d'emigrazione non riesce a prevedere correttamente in media le taglie della porzione inferiore della dispersione reale; in altre parole la previsione sovrastima la taglia dei centri minori e sottostima leggermente quella dei centri maggiori. Eppure, ripensandoci, questa è una caratteristica condivisa anche dagli altri casi nelle Figg. 4.4(k) e 4.4(l) e Figg. 4.5(k) e 4.5(l), come si può notare osservando la pendenza della retta descritta dalla dispersione.

4.3.5 Interpretazioni

Innanzitutto bisogna delucidare una caratteristica che accomuna tutte le regole d'emigrazione precedenti: esse sono da un punto di vista interpretativo miope, vale a dire non vedono i dettagli ma solo i fattori comuni che potrebbero governare le migrazioni. Pertanto il modello non dice il perché una città piú popolata, connessa o trafficata è piú attraente ma solo che quella misura viene presa in considerazione nei movimenti migratori; certamente, però, si può *a posteriori* motivare queste scelte: per esempio, i precedenti fattori possono essere legati a piú opportunità di lavoro, a maggiore vicinanza coi propri familiari, minor costo della vita, eccetera. Poi, sempre *a posteriori*, si può decidere quale fattore sia il piú realistico e da li trarre le proprie conclusioni in merito al fenomeno migratorio.

Vista dunque l'arbitrarietà delle potenziali interpretazioni si è deciso in questo sottoparagrafo di non esprimersi nel dettaglio, ma solo di evidenziare quali fattori influiscano piú probabilmente sulla migrazione alla luce dei confronti coi dati reali nelle Figg. 4.2–4.6.

Regola taglia $[RE]_S$

Dai risultati della $[RE]_S$ è chiaro che considerare solo la popolazione relativa non è sufficiente a descrivere la distribuzione delle taglie tra città. La ragione della sua degenerazione si può attribuire a un comportamento binario: se $s \leqslant s_*$ allora s_* sarà sempre un polo attraente per s [e viceversa]; quindi non appena una città perde anche solo un po' di popolazione rispetto a tutt'i loro nodi vicini, il loro destino è segnato a spopolarsi. Con tale logica si può motivare lo spopolamento *in media* dei centri medi:

1. i centri maggiori abbattono le popolazioni dei centri medi rendendoli relativamente poli repellenti sin dall'inizio della simulazione;
2. nel breve termine i centri medi si spopolano a tal punto da diventare repellenti anche rispetto ai centri minori;

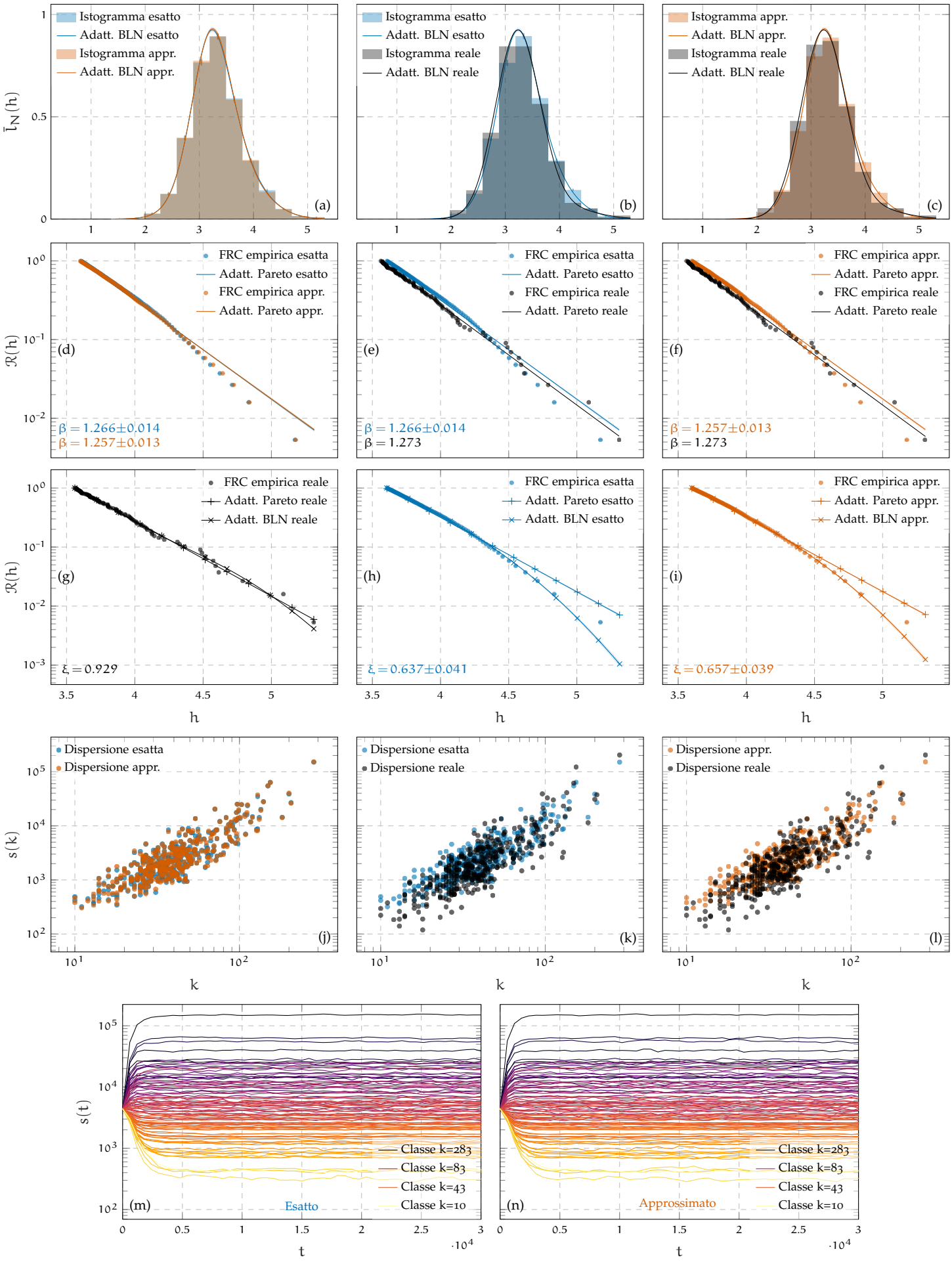

Figura 4.6: Studio della configurazione di riferimento della $[\mathrm{RE}]_{SW}$ con parametri dalla Tab. 4.6; per la spiegazione dei grafici si veda il § 4.2.3.

3. i centri minori non s'interfacciano mai direttamente con quelli maggiori per cui non si spopolano inizialmente come quelli medi.

Il risultato di questa dinamica è quello mostrato nelle Figg. 4.3(k) e 4.3(l) almeno nel caso instabile. Invece, in quello stabile non è chiaro a cosa si possa attribuire questo comportamento: di certo α non è abbastanza piccolo da rendere la funzione di Hill nella $[RE]_S$ costante e uguale $\lambda/2$, specialmente considerando che la $[RE]_{SW}$ nella configurazione di riferimento dalla Tab. 4.6 presenta pure un valore di α minore; di conseguenza l'unica conclusione che si può trarre è che il problema sia la regola d'emigrazione in sé.

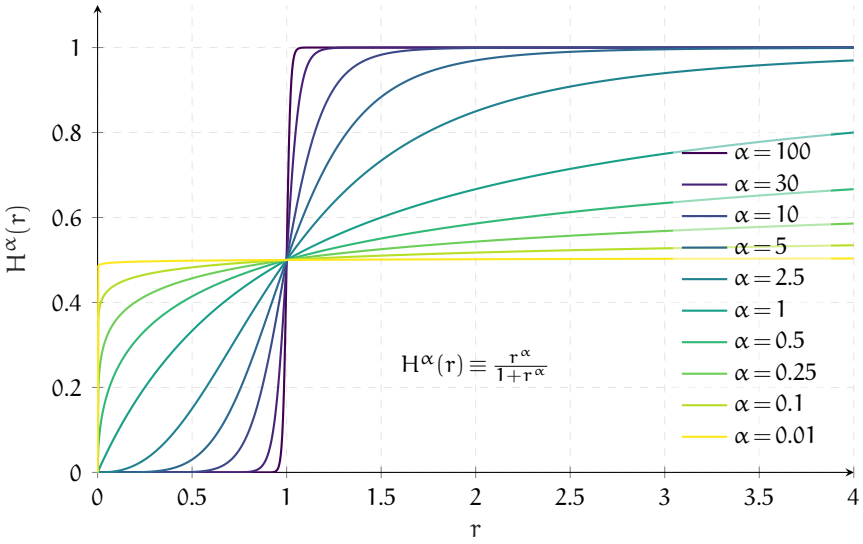

Figura 4.7: Funzione di Hill al variare del grado α.

Nel complesso, quindi, da questi risultati si può dedurre che la migrazione molto probabilmente non dipende solo dalla popolazione relativa.

Regola taglia-grado $[RE]_{SK}$

Questi risultati sono decisamente piú realistici, tuttavia, contrariamente alla popolazione relativa, l'introduzione del grado relativo tramite il rapporto k_{i_*}/k_i non è intuitiva; in altra parole ci si potrebbe chiedere perché sia piú ragionevole postulare che una città con grado basso, cioè meno connessa, abbia anche popolazione bassa. Il timore è che si stia forzando il modello a dei risultati che, seppure corrispondenti a quelli reali, non sono realistici nella loro logica piú profonda. Cionnonostante, per motivare quest'ipotesi è sufficiente far emergere due vincoli naturalmente osservabili.

V1 Il primo è di natura fisica, indotto dal grafo spaziale, e discende dal fatto che per loro natura i lati occupano spazi tra città e si collegano a esse prendendo una porzione del loro bordo; perciò l'unico modo che una città ha di aumentare i suoi gradi è quella di espandersi per avere un bordo piú esteso, ma ciò implica necessariamente un aumento di popolazione perché questa la deve mantenere.

V2 Il secondo è di natura sociale, indotto dai flussi pendolari, e viene dal fatto che una connettività maggiore sottintende una rete di trasporto piú complessa che quindi può essere giustificata se vi sono

abbastanza pendolari che la usano; questi, però, possono essere attratti da una città se essa presenta una popolazione sufficientemente elevata che motiva naturalmente tali flussi.

Il secondo vincolo è difatti confermato dalla Fig. 2.2 che riporta una correlazione positiva tra il grado e la forza: maggiore è il primo altrettanto lo sarà il secondo, vale a dire il flusso di pendolari, da cui segue V2.

Dunque il vincolo fisico giustifica a prescindere questo tipo d'interazione perché il grafo è necessariamente spaziale; parimenti il vincolo sociale motiva l'interazione se e solo se v'è una correlazione positiva tra grado e forza[6]. Tutto ciò implica la relazione $k \gg 1 \implies s \gg 1$, che rende realistica l'introduzione di k_{i_*}/k_i da un punto di vista modellistico.

È anche notevole soffermarsi, sebbene senza una risposta risolutiva, sul perché la regola d'emigrazione converga affatto: ci si potrebbe chiedere, difatti, come mai non si manifesti il flusso

$$\text{centri minori} \quad \to \quad \text{centri medi} \quad \to \quad \text{centri maggiori}.$$

L'unica ragione della mancata degenerazione del modello va ricondotta alla presenza di un retroflusso naturalmente indotto dalla $[\mathrm{RE}]_{SK}$ che bilancia quello verso i centri relativamente piú popolati e connessi; in altre parole, non solo l'introduzione dei gradi nella $[\mathrm{RE}]_S$ rende i centri medi attrattivi rispetto a quelli minori nel breve termine, ma stabilizza anche le interazioni a lungo termine, ossia stazionarie. Eppure l'esatta dinamica che instaura un flusso inverso stabilizzante non è chiara, siccome il rapporto r nella funzione di Hill (Fig. 4.7) dovrebb'essere piú piccolo nella $[\mathrm{RE}]_{SK}$ che nella $[\mathrm{RE}]_S$.

In conclusione, da tali risultati la $[\mathrm{RE}]_{SK}$ implica che la migrazione è probabilmente governata da *almeno* due fattori: la popolazione e la connettività relative fra due città.

Regola frazionata $[RE]^f_{SK}$

Innanzitutto l'introduzione di $\zeta \in (0,1)$ preserva l'Ip. 3.5 essendo una combinazione convessa[7] di due $[\mathrm{RE}]_{SK}$:

$$E^f_{SK} = (1-\zeta)E_{SK} + \zeta E_{SK} \leqslant (1-\zeta)\lambda + \zeta\lambda = \lambda,$$

ove gli argomenti sono sottintesi. Per il resto valgono le stesse considerazioni precedenti, essendo costruita a partire dalla $[\mathrm{RE}]_{SK}$, mentre i risultati, come già notato nel § 4.3.3, in generale sono migliori. Nella dispersione, però, si può notare l'emergenza di una dimensione minima ideale che le città raggiungono anche con bassa connettività; inoltre, per altri parametri o regole d'interazione (§ 4.4.1), lo stesso si può dire anche per la dimensione massima ideale, anche se nella Figg. 4.5(k) e 4.5(l) è lieve ragion per cui si nota poco. In ogni caso, il confronto colla dispersione reale implica che la città minima ideale non è realistica, ma non è detto che non possa descrivere correttamente altre dispersioni al di fuori di quella sarda.

6 Si conferma tale caratteristica anche per ogni regione italiana, oltre che per l'Italia medesima.

7 Tale logica si può generalizzare con una combinazione affine di tassi d'emigrazione, ove ogni peso è associato a una frazione della popolazione con un certo tipo di comportamento.

Perciò la $[\mathrm{RE}]^{f}_{SK}$ non è intrinsecamente sbagliata e, anzi, riesce comunque a perfezionare molti aspetti della regola precedente; semplicemente è chiaro che nel panorama sardo non esistano frazioni di popolazioni con tendenze opposte alla $[\mathrm{RE}]_{SK}$.

Regola taglia-forza $[RE]_{SW}$

L'introduzione della forza si motiva attraverso gli stessi vincoli descritti per motivare l'introduzione del grado nella $[\mathrm{RE}]_{S}$. In effetti, il vincolo sociale vale direttamente poiché la forza coincide proprio col flusso di pendolari; d'altro canto quello fisico vale poiché la forza non è che un'estensione delle informazioni puramente binarie del grado, in cui ora un valore positivo implica sia la presenza di un lato che l'intensità del suo flusso pendolare.

Cosí ragionando è possibile ora distinguere città che hanno ugual grado ma non necessariamente ugual traffico; tale naturale distinzione permette proprio di corregge la relazione quasi biunivoca tra il grado e la popolazione nelle Figg. 4.4(k) e 4.4(l) trasformandola in una piú realistica correlazione positiva descritta dalle Figg. 4.6(k) e 4.6(l).

Nel complesso, essendo questi risultati i migliori tra quelli visti, la $[\mathrm{RE}]_{SW}$ implica che molto probabilmente il fenomeno della migrazione è governato da due fattori: la popolazione e il traffico relativi fra due città.

Parametri λ *e* α

In tutte le regole d'emigrazione analizzate vi sono due principali parametri: λ e α; per cercare di comprendere il loro significato si può procedere mediante due studi parametrici: entrambi prendono la configurazione di riferimento della $[\mathrm{RE}]_{SW}$ nella Tab. 4.6 e perturbano solo uno dei due parametri lasciando invariati gli altri; la Tab. 4.7 riassumere quanto detto dove $N_p = 5$ indica il numero di valori campionati uniformemente nei due intervalli.

Tabella 4.7: Parametri degli studi parametrici delle Figg. 4.8–4.17

	N_p	λ	α	N_t
Studio parametrico λ	5	[0.01,0.3]	0.44	10^7
Studio parametrico α	5	0.18	[0.3,0.6]	3×10^6

Prima di procedere è necessario tenere a mente una proprietà:

Osservazione 4.14. Per la conservazione di massa (Ip. 3.7) la media campionaria[8] P/N è costante qualunque sia la coppia parametrica (λ,α) scelta. Tuttavia ciò *non* implica che il parametro μ in una $X \sim \text{Lognormale}(\mu,\sigma)$ sia anch'esso costante; infatti quello che si preserva è il suo valore atteso di forma

$$\mathbb{E}[X] = 10^{\mu+\sigma^2\ln(10)/2} = \frac{P}{N},$$

dal quale si comprende immediatamente che se la varianza σ^2 dovesse aumentare, la media diminuirebbe e viceversa.

Come si può vedere nella Fig. 1.1, questo comportamento è poco evidente se si ragiona direttamente con una lognormale in spazio lineare, sia in scala

8 Per la Sardegna il valore è $\approx 4.395\times10^3$ dalla Tab. 4.2.

lineare che logaritmica; invece in spazio logaritmico, tutto cambia perché μ e σ corrispondono proprio colla media e varianza dell'equivalente gaussiana.

Analoghe considerazioni valgono pure per l'adattamento bilognormale (1.4) essendo una combinazione convessa di due lognormali.

Per quanto riguarda il primo studio parametrico, esso varia λ:

- nella Fig. 4.8 si può notare come la varianza aumenti allargando la distribuzione per λ crescente, seppure leggermente;
- nella Fig. 4.9 l'indice di Pareto di fatto rimane pressoché inalterato, variando tra gli estremi solo di ≈ 0.05 nella media;
- nella Fig. 4.10 la relazione tra Pareto e bilognormale rimane invariata: conferma in ogni caso che la seconda segue meglio la fine della coda rispetto alla prima, mentre coincidono all'inizio;
- nella Fig. 4.11 si può osservare che la dispersione è inalterata al variare di λ: il comportamento medio è il medesimo;
- la Fig. 4.12 conferma infine la convergenza di tutt'i casi considerati, oltre che indicare un minore tempo di convergenza per λ crescente.

Si nota che tutte queste variazioni sono comunque lievi considerando che λ aumenta di un ordine di grandezza; ciò significa che λ influenza poco i risultati finali, a eccezione del tempo di convergenza. Tuttavia, per $\lambda > 0.3$ si arriva comunque a una configurazione instabile.

Per quanto riguarda il secondo studio parametrico, esso varia α:

- nella Fig. 4.13 si può notare come al crescere di λ la varianza incrementi notevolmente, diminuendo di conseguenza la media (Oss. 4.14) e allargando la distribuzione;
- nella Fig. 4.14 l'indice di Pareto varia in modo significativo diminuendo quasi di un ordine di grandezza;
- nella Fig. 4.15 la relazione tra Pareto e bilognormale continua a rimanere invariata, come nello studio parametrico precedente;
- nella Fig. 4.16 la dispersione inizialmente sovrastima i centri minori-medi e sottostima quelli maggiori, per poi invertire tale relazione all'aumentare di α;
- la Fig. 4.17 conferma la convergenza di tutt'i casi considerati, oltre che indicare un maggior tempo di convergenza per α crescente.

Contrariamente allo studio parametrico λ la variazione dei risultati è significativa; in particolare l'indice di Pareto β sembra in larga parte dipendere principalmente dal valore di α. Similmente, per valori $\alpha > 0.6$ si perviene a una configurazione instabile.

Nel complesso, di fronte a questi studi, λ e α sono cosí interpretabili:

- λ è l'attrazione: maggiore è il suo valore piú i centri relativamente maggiori riescono ad attrarre, in una singola interazione, frazioni elevate della popolazione dei centri relativamente minori;
- α è la repulsione: maggiore è il suo valore piú i centri relativamente maggiori sono respinti da quelli relativamente minori; essa è chiaramente inversamente legata all'entità del retroflusso stabilizzante.

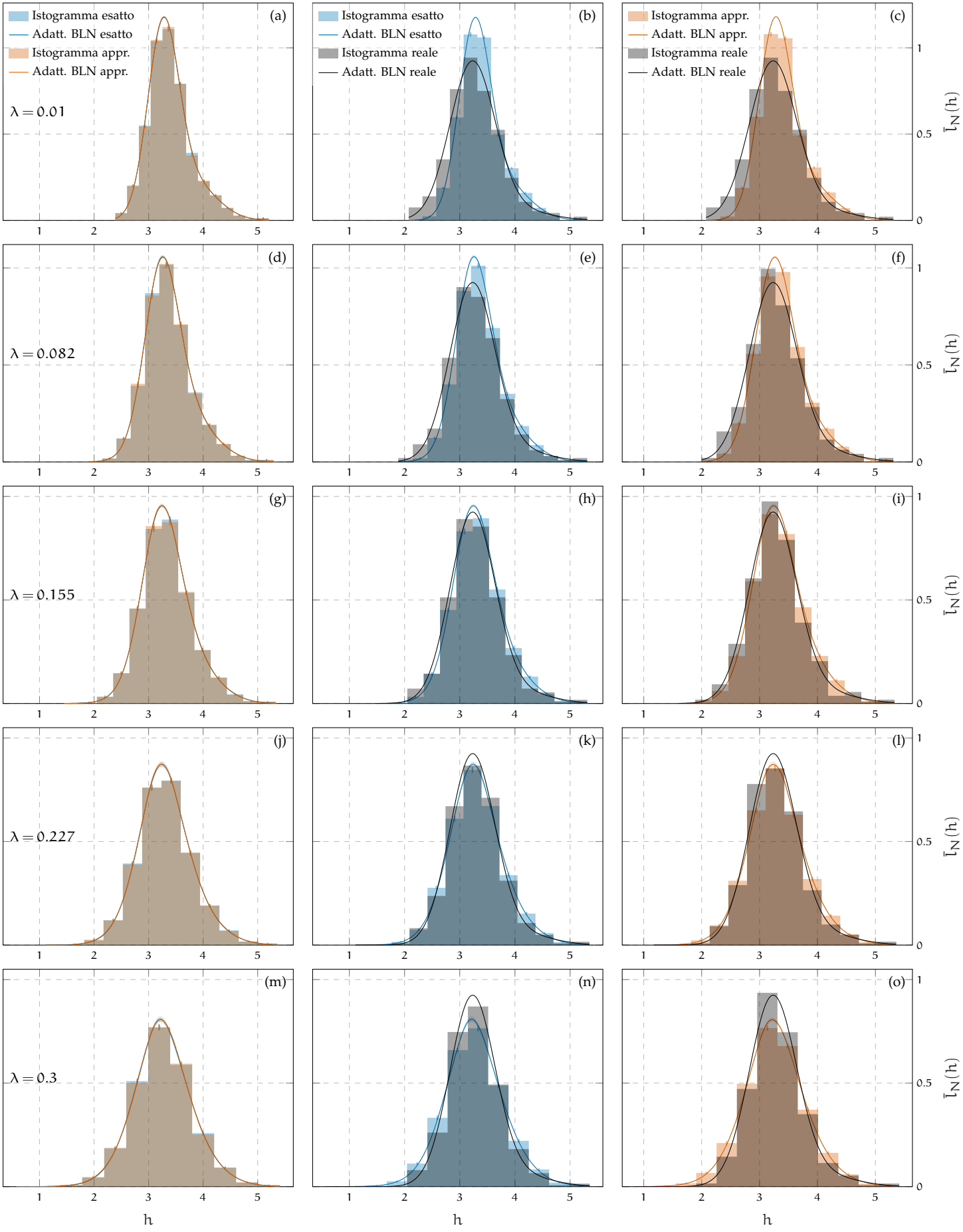

Figura 4.8: Distribuzioni dello studio parametrico λ con $[RE]_{SW}$ e parametri dalla Tab. 4.7; per la spiegazione dei grafici si veda il § 4.2.3.

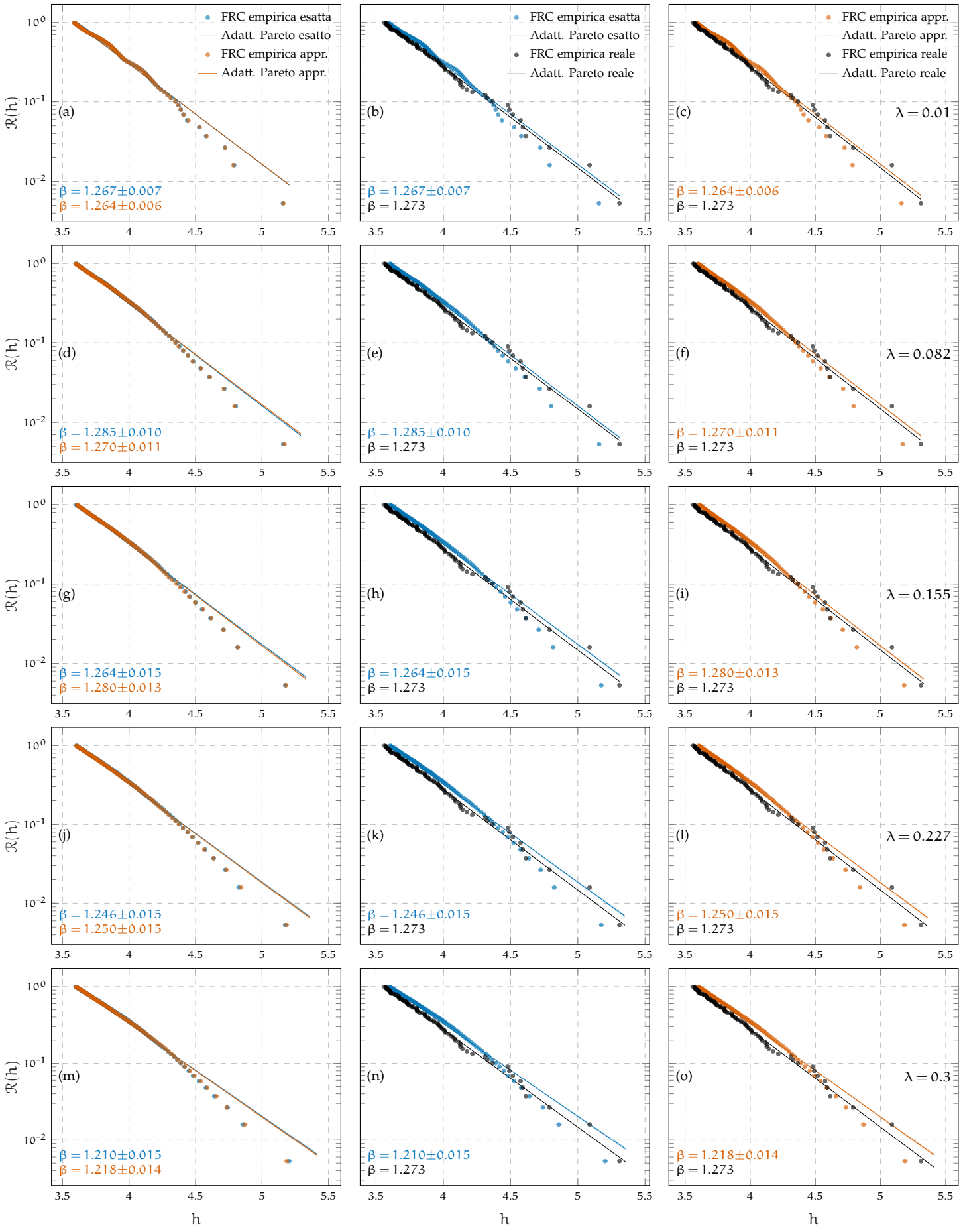

Figura 4.9: Adattamenti di Pareto dello studio parametrico λ con $[RE]_{SW}$ e parametri dalla Tab. 4.7; per la spiegazione dei grafici si veda il § 4.2.3.

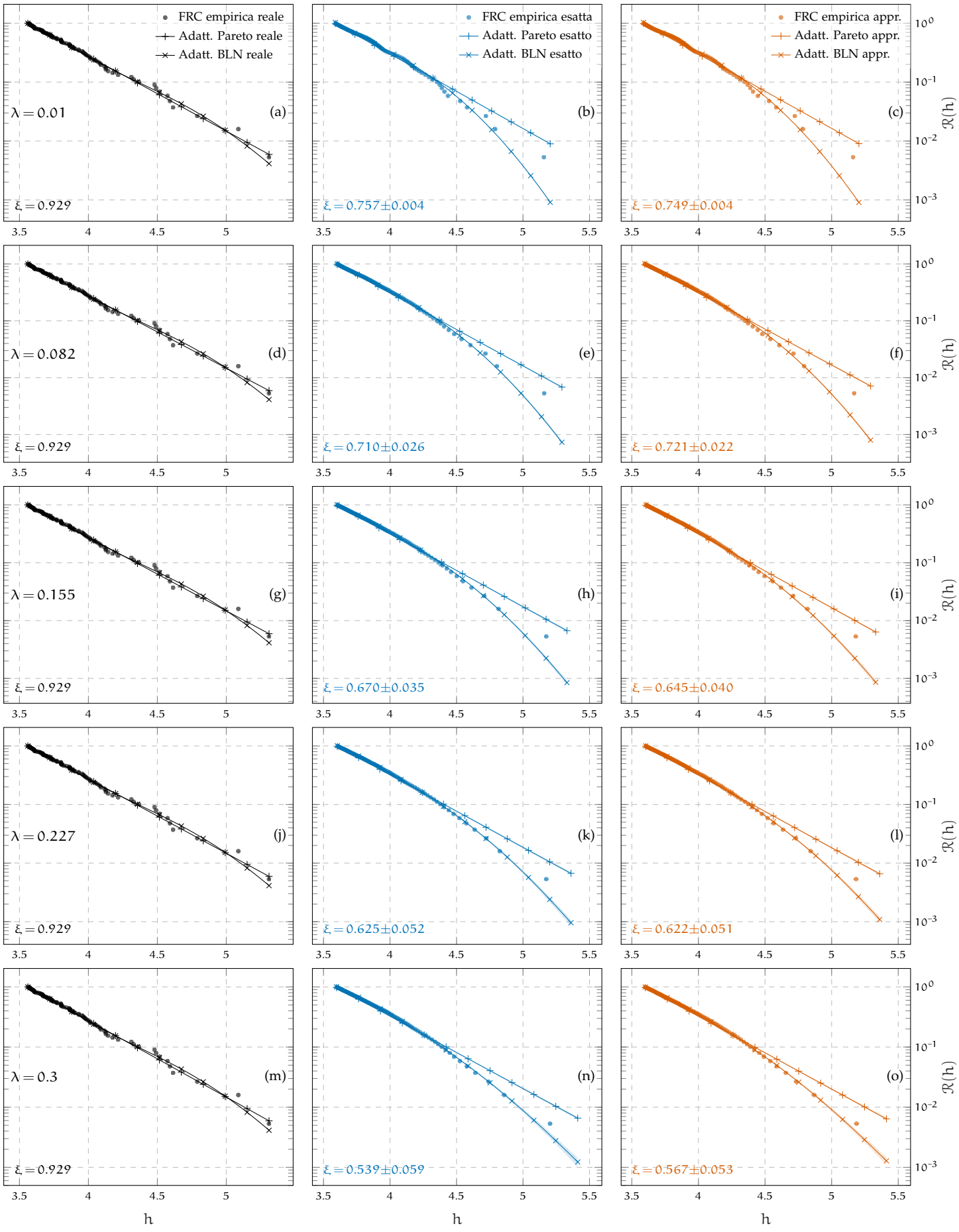

Figura 4.10: Adattamenti di Pareto e lognormali bimodali dello studio parametrico λ con $[RE]_{SW}$ e parametri dalla Tab. 4.7; per la spiegazione dei grafici si veda il § 4.2.3.

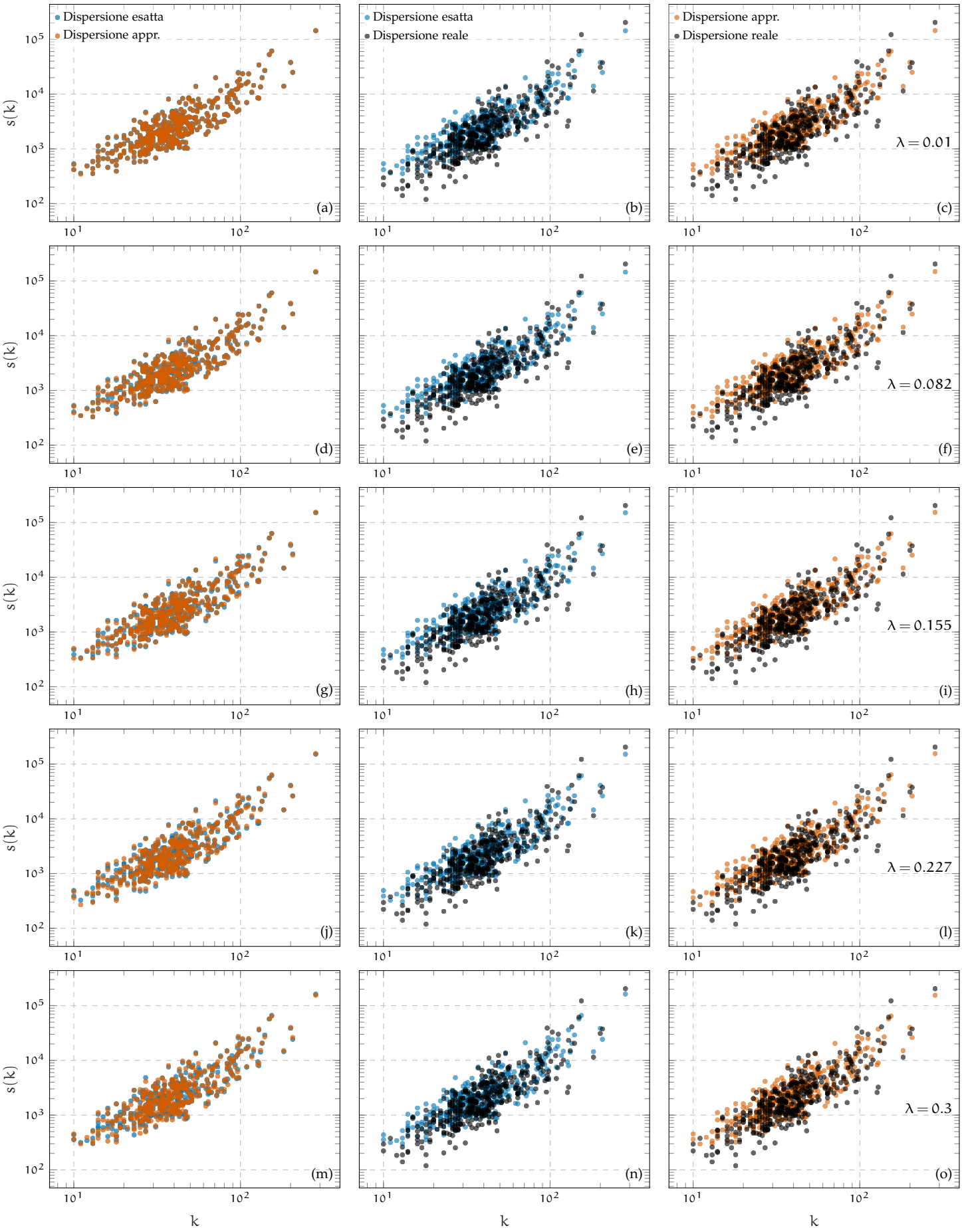

Figura 4.11: Taglie contro gradi dello studio parametrico λ con $[RE]_{SW}$ e parametri dalla Tab. 4.7; per la spiegazione dei grafici si veda il § 4.2.3.

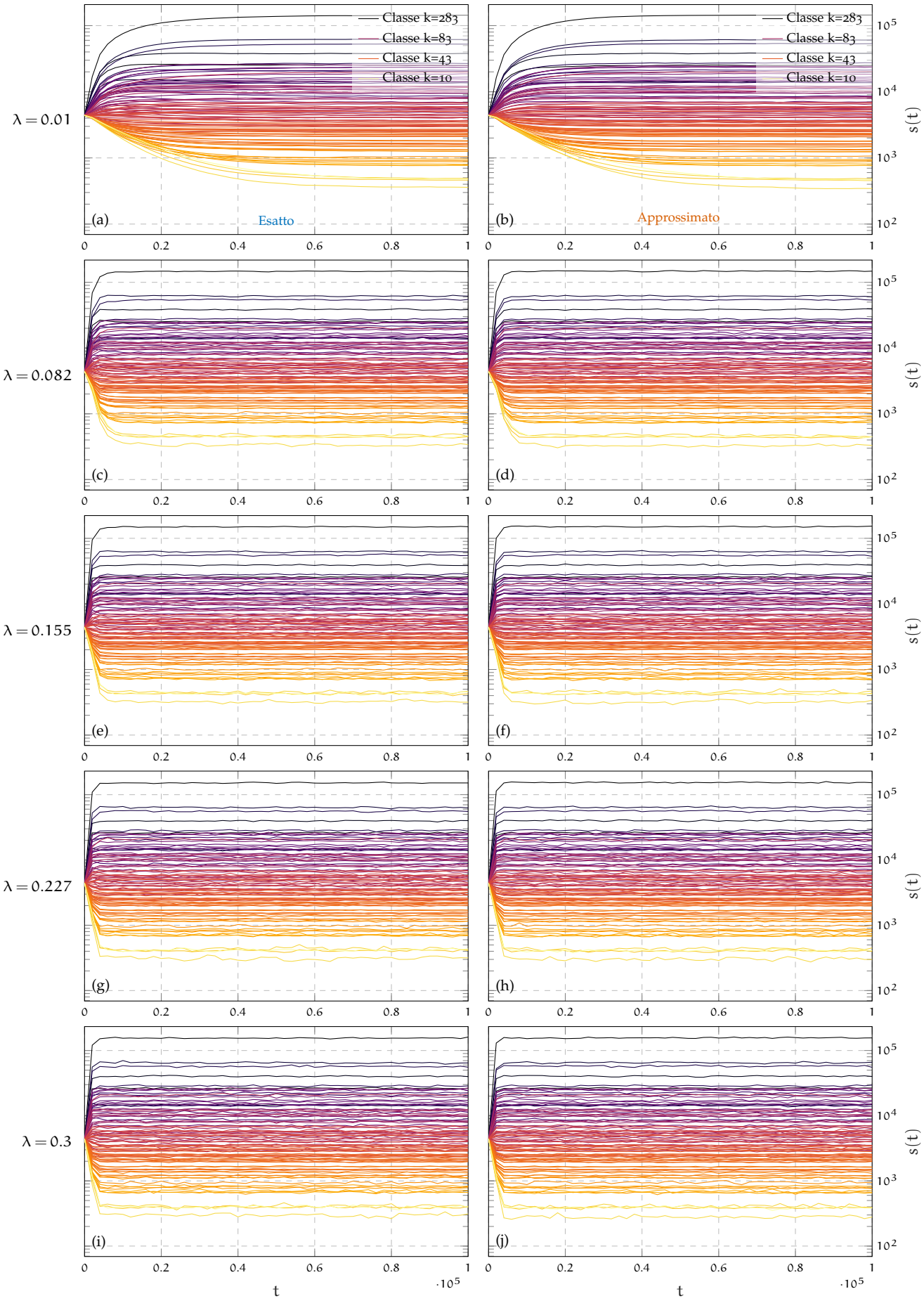

Figura 4.12: Evoluzioni delle taglie medie delle classi dei gradi dello studio parametrico λ con $[RE]_{SW}$ e parametri dalla Tab. 4.7; per la spiegazione dei grafici si veda il § 4.2.3.

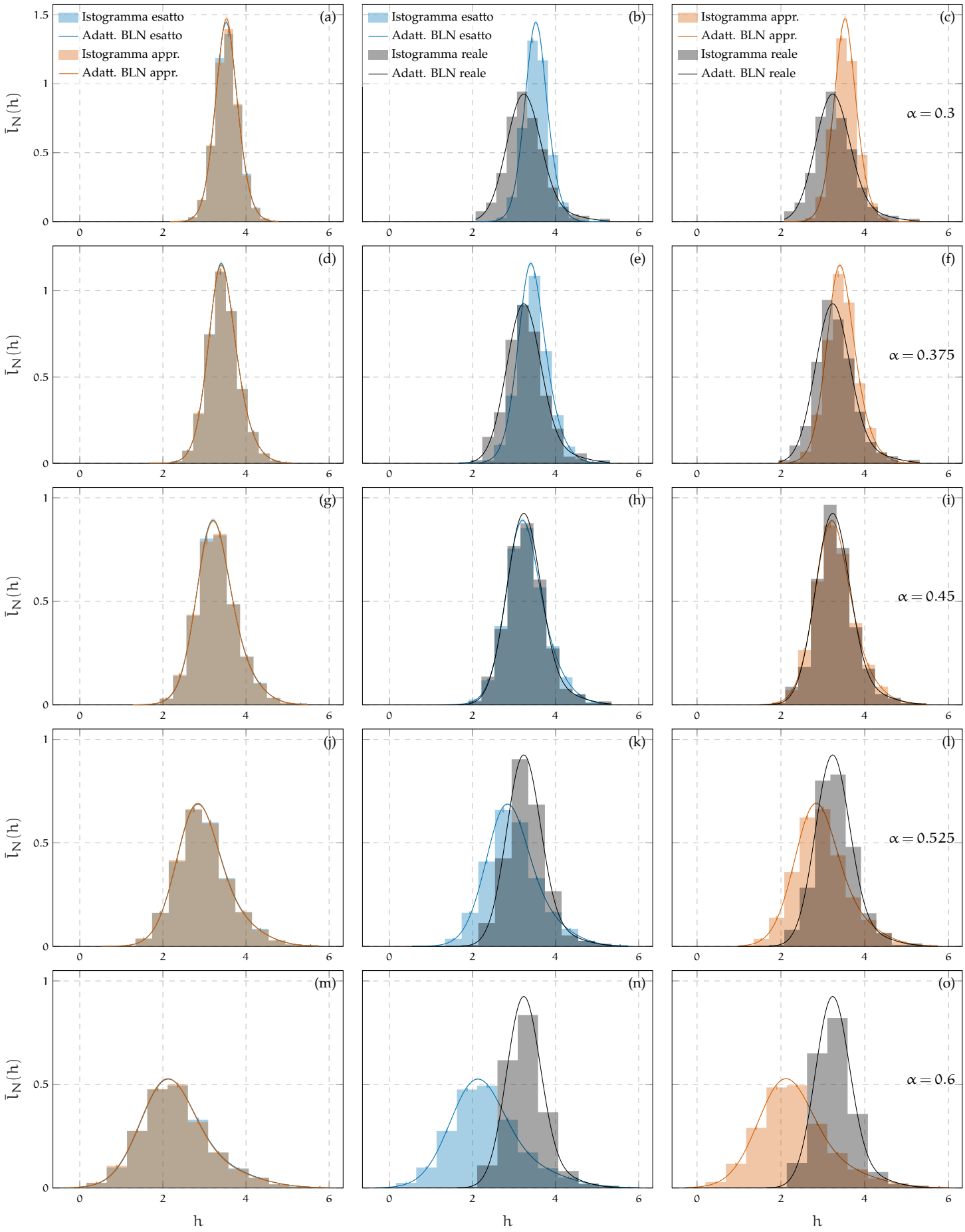

Figura 4.13: Distribuzioni dello studio parametrico α con $[RE]_{SW}$ e parametri dalla Tab. 4.7; per la spiegazione dei grafici si veda il § 4.2.3.

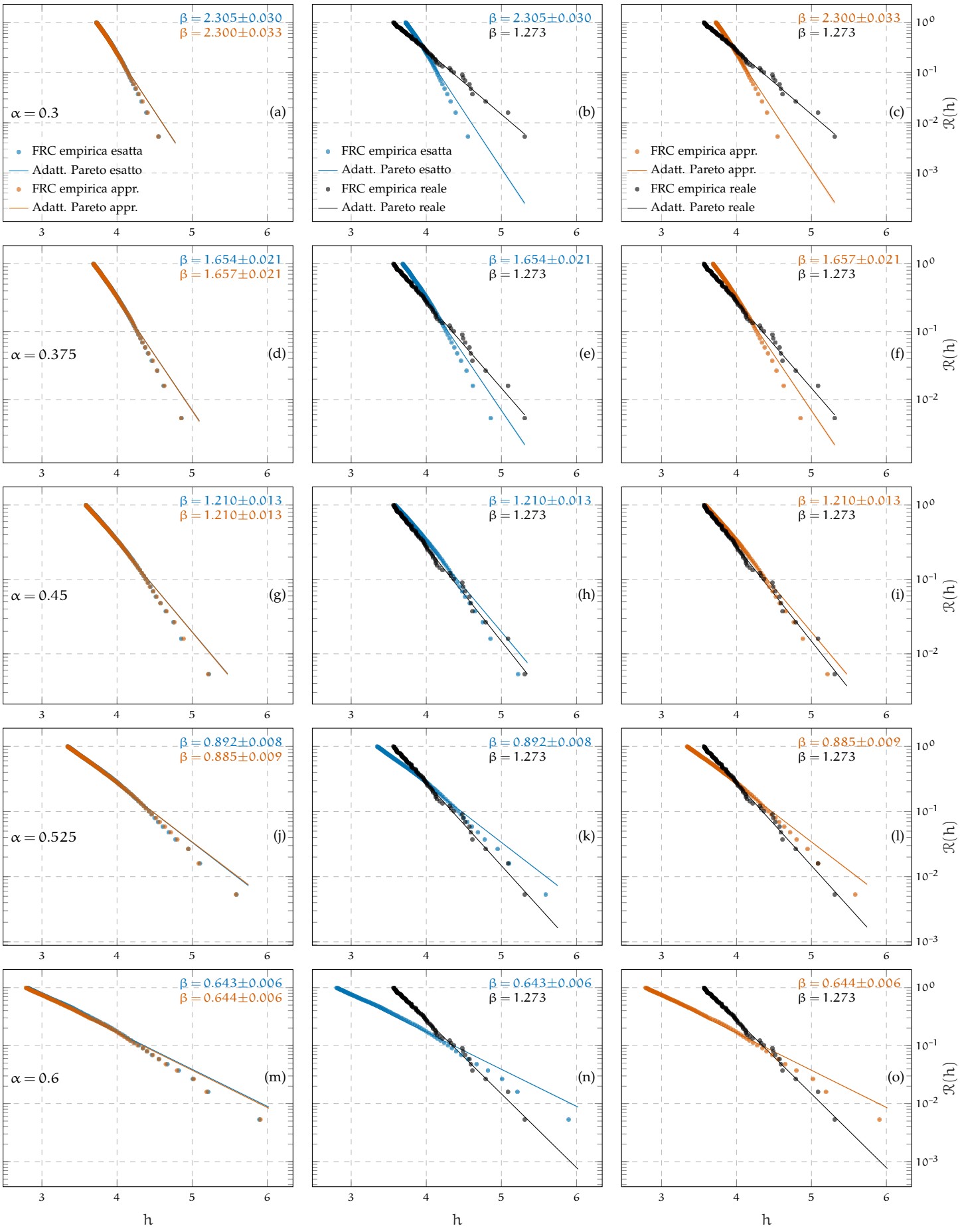

Figura 4.14: Adattamenti di Pareto dello studio parametrico α con $[RE]_{SW}$ e parametri dalla Tab. 4.7; per la spiegazione dei grafici si veda il § 4.2.3.

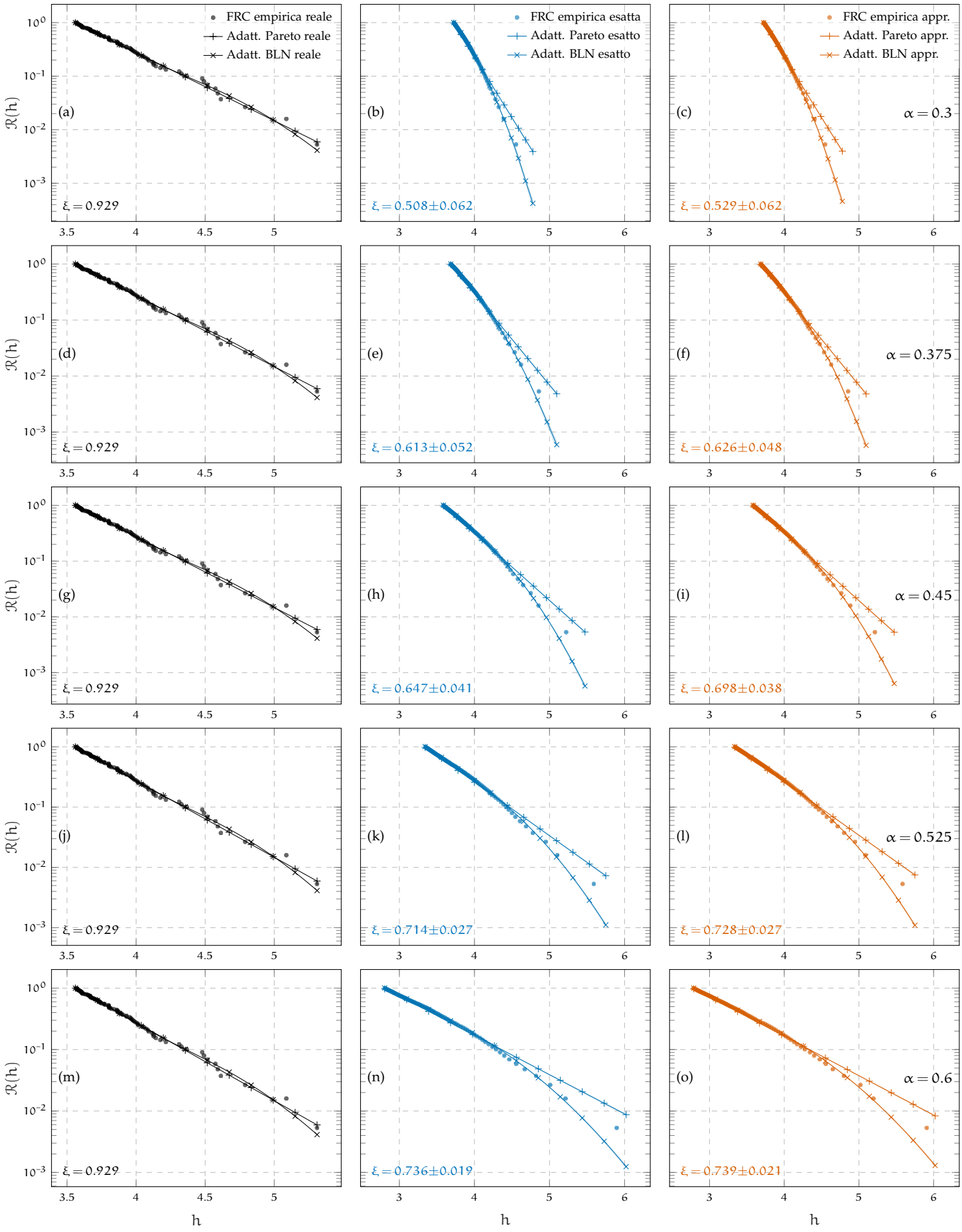

Figura 4.15: Adattamenti di Pareto e lognormali bimodali dello studio parametrico α con $[RE]_{SW}$ e parametri dalla Tab. 4.7; per la spiegazione dei grafici si veda il § 4.2.3.

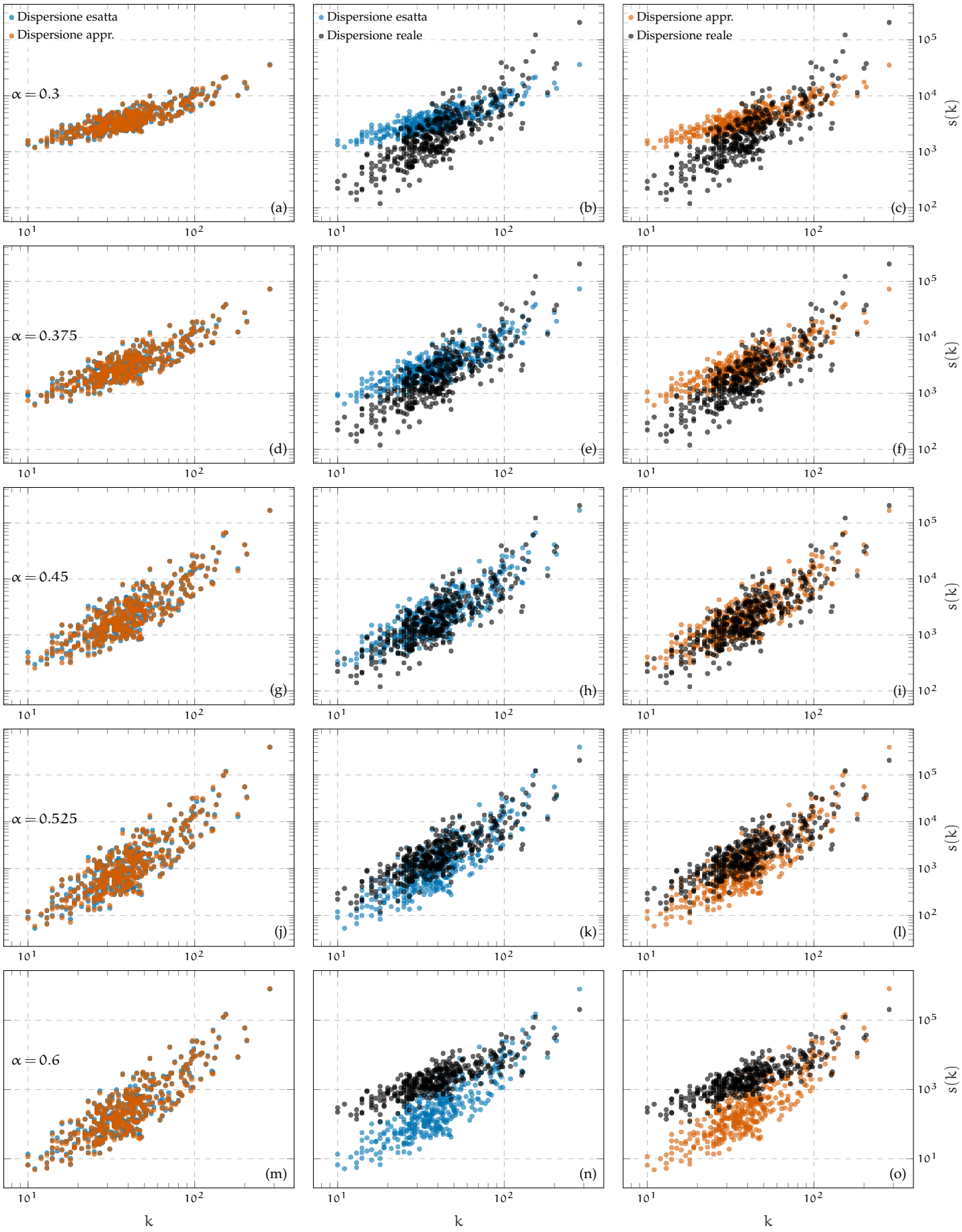

Figura 4.16: Taglie contro gradi dello studio parametrico α con $[RE]_{SW}$ e parametri dalla Tab. 4.7; per la spiegazione dei grafici si veda il § 4.2.3.

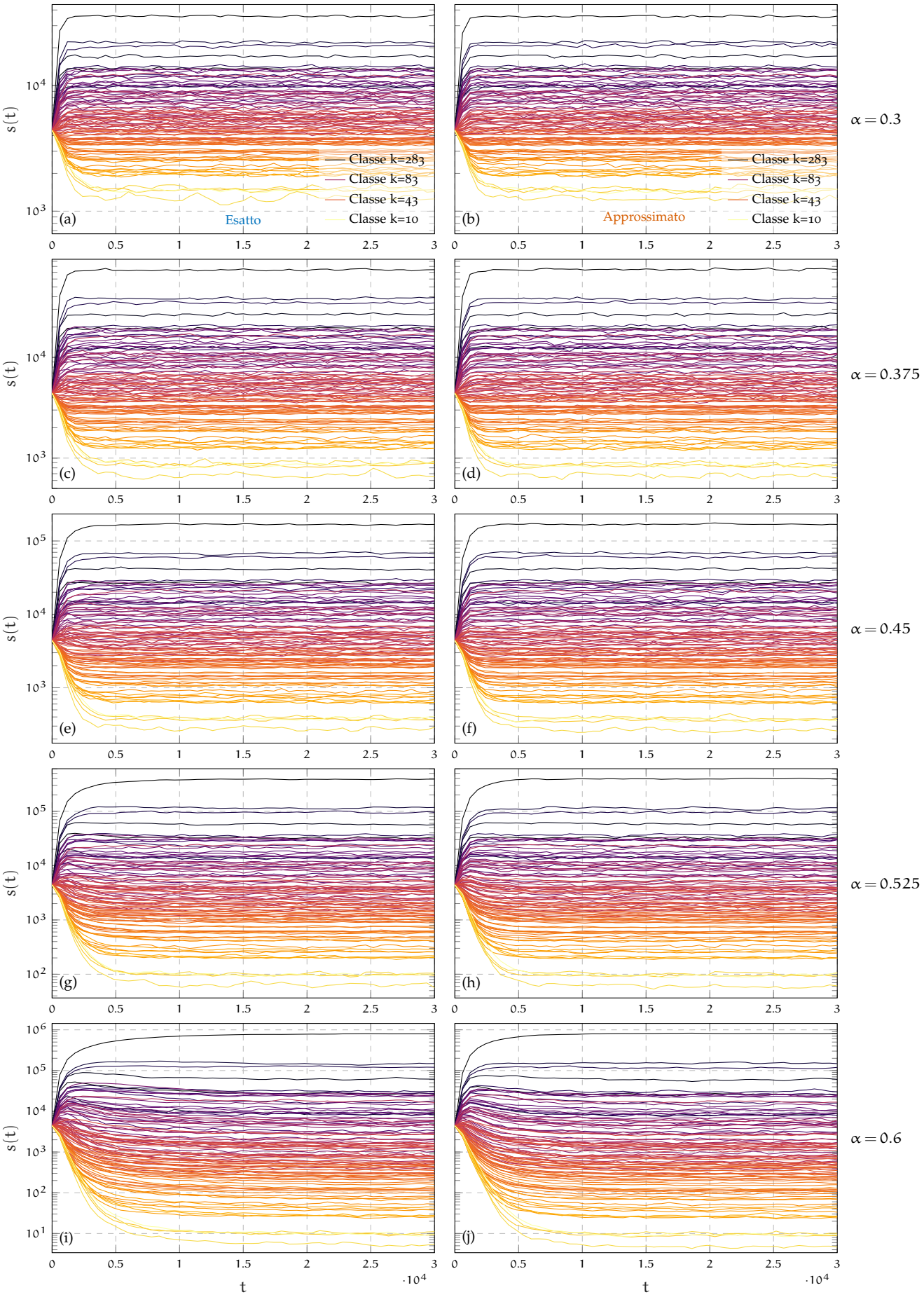

Figura 4.17: Evoluzioni delle taglie medie delle classi dei gradi dello studio parametrico α con $[\mathrm{RE}]_{SW}$ e parametri dalla Tab. 4.7; per la spiegazione dei grafici si veda il § 4.2.3.

Un altro punto di vista è il seguente: λ influenza principalmente la porzione della funzione di Hill (Fig. 4.7) oltre l'unità ($r > 1$), e quindi l'attrazione, mentre α condiziona maggiormente la porzione sotto l'unità ($r < 1$), e dunque la repulsione.

4.4 ALTRI CASI NOTEVOLI

4.4.1 Varianti delle regole d'interazione

La prima variante consiste nel dividere il rapporto per la repulsione α anziché valutarne la potenza:

$$E(s,s_*,i,i_*) = \lambda \frac{\frac{(s_*/s)(w_{i_*}/w_i)}{\alpha}}{1+\frac{(s_*/s)(w_{i_*}/w_i)}{\alpha}} = \lambda \frac{(s_*/s)(w_{i_*}/w_i)}{\alpha+(s_*/s)(w_{i_*}/w_i)}, \tag{4.7}$$

in tal modo si può scegliere α per avere tra città simili, ossia

$$(s_*/s)(w_{i_*}/w_i) \approx 1 \implies s \approx s_* \text{ e } w_{i_*} \approx w_i,$$

scambi intorno all'1% [o una generica percentuale p] della loro taglia:

$$1\% = \frac{\lambda}{\alpha+1} \implies \alpha = \frac{\lambda}{1\%} - 1 = 4.$$

Purtroppo, nonostante la bontà di questa logica, questa regola non riesce a convergere e spopola lentamente molte città, analogamente alla $[RE]_S$; ciò è almeno valido per i parametri sperimentati.

Osservazione 4.15. Proprio per i pessimi risultati si è deciso di non analizzarla a fondo, ragion per cui potrebbero esservi dei buoni valori che fanno convergere anche la (4.7).

Osservazione 4.16. La (4.6) non è altro che la versione frazionata della (4.7).

La seconda variante si ottiene prendendo la $[RE]_{SW}$ ed elevando rispetto ad α l'intero rapporto moltiplicato a λ:

$$E(s,s_*,i,i_*) = \lambda \left(\frac{(s_*/s)(w_{i_*}/w_i)}{1+(s_*/s)(w_{i_*}/w_i)} \right)^{\alpha}. \tag{4.8}$$

Le principali differenze rispetto a $[RE]_{SW}$, illustrate nella Fig. 4.18, sono due: tende a λ per $\alpha \to 0$ ed è molto piú dolce nelle variazioni in contrapposizione alla Fig. 4.7.

Per il resto i risultati ottenuti sono analoghi a quelli della $[RE]_{SW}$, considerando, ovviamente, dei parametri diversi da quelli nella Tab. 4.6.

4.4.2 L'Italia e la Valle d'Aosta

Come menzionato all'inizio del § 4.3, è interessante studiare la regione della Valle d'Aosta e l'intera Italia, specialmente per il numero di nodi. Per brevità,

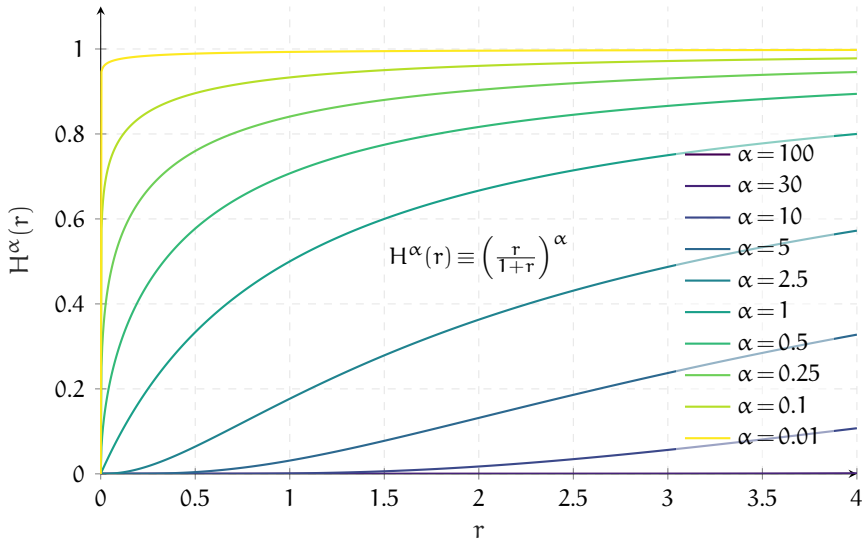

Figura 4.18: Funzione di Hill modificata al variare del grado α.

tuttavia, non si ripropongono tutte le regole d'emigrazione viste previamente ma si considera solo la configurazione migliore della Sardegna, vale a dire la $[RE]_{SW}$ con parametri dalla Tab. 4.6.

Perdipiú, si varia il numero di simulazioni: per la Valle d'Aosta si aumentano a $R = 1000$ e per l'Italia si diminuiscono a 10; difatti come si può notare nella Tab. 4.8 la prima è composta da soli 74 nodi e invece la seconda da ben 8100: due ordini di grandezza superiori alla prima e uno rispetto alla Sardegna; di conseguenza, mentre i risultati della Valle d'Aosta nella Fig. 4.19 hanno richiesto meno di 15 minuti, quelli dell'Italia nella Fig. 4.20 hanno richiesto piú di 14 ore; per dare un ulteriore confronto, i risultati della Sardegna hanno richiesto meno di 30 minuti per configurazione. Di fatto è dunque proibitivo considerare piú di 10 simulazioni per l'Italia.

I risultati si commentano parimenti a prima: per la Valle d'Aosta

- le Figg. 4.19(a), 4.19(d) e 4.19(j) illustrano che l'approssimazione corrisponde quasi perfettamente alla simulazione esatta;
- nelle Figg. 4.19(b) e 4.19(c) le distribuzioni simulate approssimano bene quella reale al di fuori del picco, segnalando una maggiore varianza, cosa che però non sembra influenzare gl'indici di Pareto nelle Figg. 4.19(e) e 4.19(f) essendo ragionevolmente simili;
- nelle Figg. 4.19(g)-4.19(i) Pareto pare fittare meglio la coda rispetto alla bilognormale, probabilmente a causa del basso valore di N;
- nelle Figg. 4.19(k) e 4.19(l) le dispersioni ben approssimano quella reale, sovrastimando di poco i centri minori-medi.

Per quanto riguarda l'Italia

- le Figg. 4.20(a), 4.20(d) e 4.20(j) mostrano che l'approssimazione corrisponde quasi perfettamente alla simulazione esatta;
- nelle Figg. 4.20(b) e 4.20(c) le distribuzioni simulate riproducono tutto sommato la forma di quella reale, ma sovrastimano leggermente la media come pure negl'indici di Pareto nelle Figg. 4.20(e) e 4.20(f);

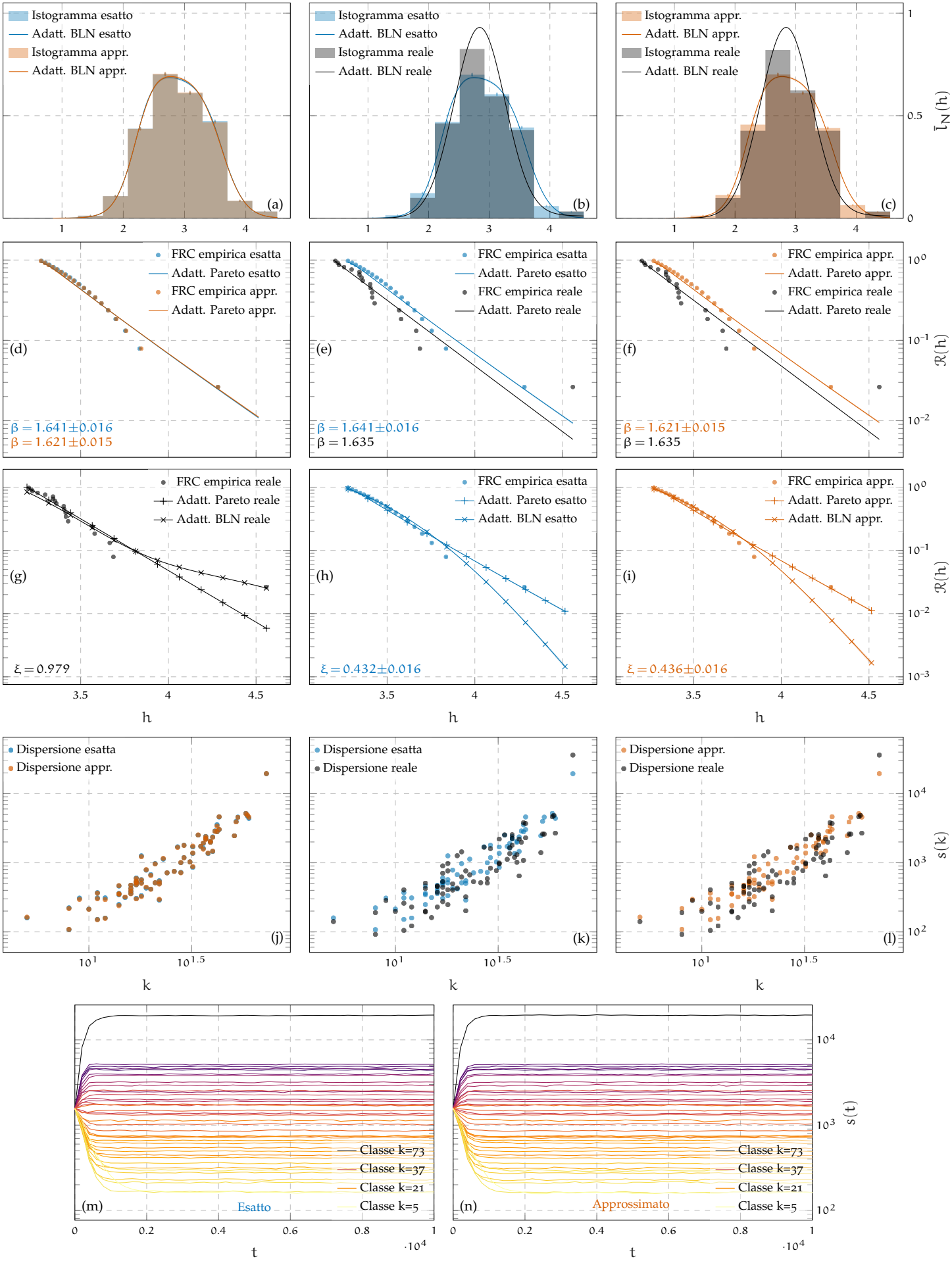

Figura 4.19: Studio della configurazione di riferimento della $[RE]_{SW}$ per la Valle d'Aosta con parametri dalla Tab. 4.8; per la spiegazione dei grafici si veda il § 4.2.3.

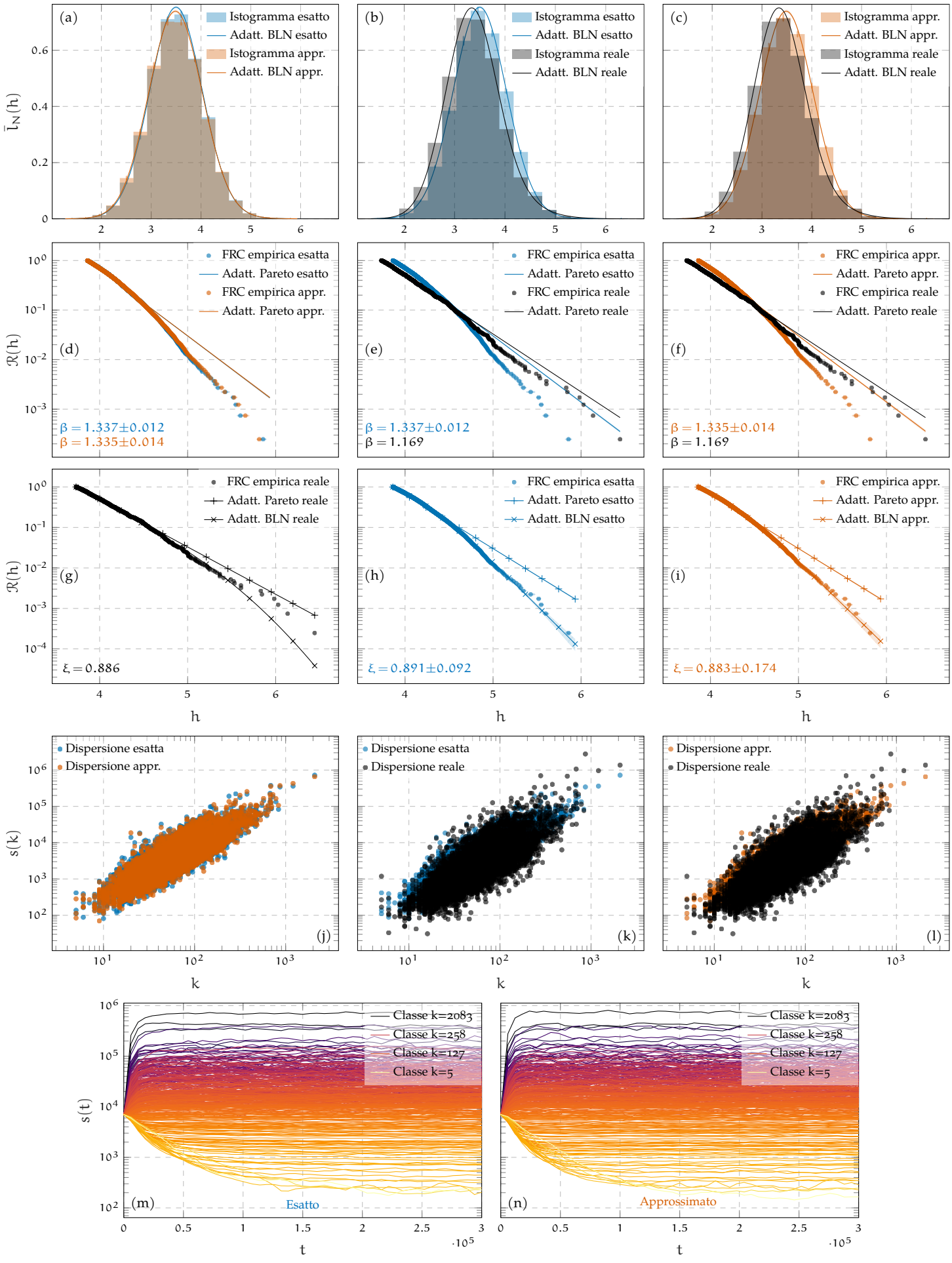

Figura 4.20: Studio della configurazione di riferimento della $[\mathrm{RE}]_{SW}$ per l'Italia con parametri dalla Tab. 4.8; per la spiegazione dei grafici si veda il § 4.2.3.

Tabella 4.8: Dati [12, 14] e parametri della Valle d'Aosta e dell'Italia nel 1991.

	N	P	β	ξ	μ_1	σ_1	μ_2	σ_2
VdA	74	$\approx 1.159\times10^5$	1.635	0.979	2.841	0.421	4.215	0.707
It	8100	$\approx 5.678\times10^7$	1.169	0.886	3.318	0.502	3.908	0.674

- ◇ nelle Figg. 4.20(g)-4.20(i) la bilognormale bimodale segue meglio la coda verso la fine, sebbene coincida colla Pareto nella prima metà;
- ◇ nelle Figg. 4.20(k) e 4.20(l) le dispersioni ben approssimano quella reale ma ripropongono in scala maggiore i difetti riscontrati nella dispersione della $[\mathrm{RE}]_{SW}$ nelle Figg. 4.6(k) e 4.6(l).

Nel complesso i parametri migliori della Sardegna vanno bene per la Valle d'Aosta se si è interessati esclusivamente ad approssimare l'indice di Pareto e non la distribuzione, al contrario per l'Italia.

Si conclude osservando che i parametri adattati della bilognormale italiana nella Tab. 4.8 corrispondono abbastanza bene a quelli ottenuti da [10, Tab. 6.2 p. 230]: le differenze sono imputabili o ai dati del 1991 o all'algoritmo usato per fittare la bilognormale (List. A.5).

5 CONCLUSIONI

Tutt'i risultati illustrati nel precedente capitolo corroborano la prospettiva nodo-agente in un contesto urbano per descrivere sia la distribuzione della taglia tra città che il loro indice di Pareto.

L'approssimazione di rango unitario dalla Def. 3.13 produce dei risultati corrispondenti quasi perfettamente a quelli ottenuti esattamente sia con N limitato (Valle d'Aosta) che elevato (Italia), confermando [17].

Tutte le regole d'interazione, vale a dire d'emigrazione (Oss. 3.22), riescono a riprodurre fedelmente l'indice di Pareto della coda della distribuzione reale, mentre la distribuzione in sé è stata complessivamente ben riprodotta con quasi ogni regola considerata. In ogni caso la lognormale bimodale segue meglio la coda rispetto alla Pareto confermando [9, 10]; tuttavia entrambi gli adattamenti hanno pregi e difetti: da un punto di vista ingegneristico la Pareto è piú facile da trattare avendo solo un parametro, sebbene sia al tempo stesso peggiore matematicamente; viceversa, la lognormale bimodale è piú corretta a livello matematico ma è composta da ben 5 parametri, rendendo piú difficile un eventuale confronto.

In conclusione, è naturale che il lavoro qui presentato non sia esaustivo e conduca a molteplici sviluppi futuri:

1. sperimentare con regole d'interazione contenenti altri coefficienti della teoria dei grafi come, per esempio, la centralità per intermediazione o il coefficiente d'aggregazione;
2. ricavare l'equazione di Fokker-Planck della $[\mathrm{EtB}]^{\mathbf{A}}_{\mathbf{S}}$ per studiare analiticamente la densità stazionaria, sebbene la natura non lineare e simmetrica delle regole d'emigrazione qui proposte rendano tale obbiettivo alquanto difficile da raggiungere;
3. abbandonare l'ipotesi semplificativa della rete statica per descrivere una rete dinamica che evolva parallelamente alla popolazione con un possibile confronto con dati reali [se esistenti];
4. applicare la presente teoria anche a reti europee o internazionali per confermare la sua validità sia in altri contesti che a scale superiori.

APPENDICE

A CODICE

Il codice è reperibile in questo repositorio [19] ed è stato scritto interamente in Python 3.13.7 mentre le simulazioni sono state eseguite su una macchina con sistema operativo Windows e processore Intel Core i5-7500 (quadri-nucleo, 3.40 GHz).

Innanzitutto `main.py` nel List. A.1 è il *file* principale contenente quasi tutta la logica del programma:

- `clsGUI.GatherParameters()` raccoglie tutt'i parametri attraverso un'interfaccia grafica mostrata nella Fig. A.1;
- `libD.ExtractRegionData()` estrae e analizza i dati ISTAT [12–14];
- `libD.LoadRegionData()` carica i dati appena o già estratti;
- `libN.NetworkAnalysis()` è la classe contenente le funzioni per riprodurre i risultati di [6];
- `libK.KineticSimulation()` è la classe contenente le funzioni per simulare l'Alg. 1 e trattare i risultati;
- `libK.ParametricStudy()` è la classe chiamata in alternativa alla precedente se si è interessati a uno studio parametrico[1].

Listato A.1: Codice `main.py`.

```
from multiprocessing import freeze_support

def main():
    import libGUIs
    import libData as libD
    import libNetworks as libN
    import libKTMAS as libK


    ### Graphic User Interface ###
    clsGUI = libGUIs.ParametersGUI()
    clsPrm = clsGUI.GatherParameters() # Parameters


    if clsPrm.simFlag:

        ### Data extraction ###
        if clsPrm.extraction: libD.ExtractRegionData()


        ### Matrices extraction ###
        clsReg = libD.LoadRegionData(clsPrm.region)


        ### Network analysis ###
        if clsPrm.analysis:
```

1 Di fatto non è altro che `libK.KineticSimulation()` tante volte quanti sono le variazioni del parametro studiato.

```
            clsNA = libN.NetworkAnalysis(clsPrm,clsReg)

            clsNA.DegreeDistributionFig()
            clsNA.WeightDistributionFig()
            clsNA.StrengthDistributionFig()

            clsNA.BetweennessCentralityFig()
            clsNA.StrengthVsDegreeFig()

            clsNA.AClusteringCoefficientFig()
            clsNA.WClusteringCoefficientFig()

            clsNA.AAssortativityFig()
            clsNA.WAssortativityFig()

            # clsNA.ShowFig()


        ### Kinetic simulation ###
        if clsPrm.parametricStudy:
            clsKS = libK.ParametricStudy(clsPrm,clsReg)
        else:
            clsKS = libK.KineticSimulation(clsPrm,clsReg)

        clsKS.MonteCarloSimulation()

        clsKS.SizeDistrFittingsFig()
        clsKS.AverageSizeFig()
        clsKS.SizeVsDegreeFig()
        clsKS.SizeDistrEvolutionFig()
        clsKS.SizeEvolutionsFig()

        # clsKS.ShowFig()

if __name__ == "__main__":
    freeze_support()
    main()
```

Invece i rimanenti listati formano nel complesso 6 moduli ognuno contenente delle funzioni, classi o anche solo informazioni, che svolgono principalmente quanto appena descritto.

Il primo modulo è `libParameters.py` nel List. A.2 che semplicemente contiene tutt'i parametri[2] come, per esempio, i percorsi relativi dei dati estratti oppure i dati degli studi del caso.

Listato A.2: Modulo `libParameters.py`.

```
# Library to more easily define/change parameters

from dataclasses import dataclass
from typing import Any

from copy import deepcopy
import numpy as np

from pathlib import Path


### Module attributes ###

```

2 Piccola curiosità: è servito anche a evitare, durante la scrittura del codice, un piccolo problema di ricorsione infinita durante i caricamenti dei moduli.

```
mainFolder     = Path(__file__).resolve().parent
projectFolder  = mainFolder.parent
dataFolder     = projectFolder/'Dati'

matrixZipPath  = dataFolder/'MatriciPendolarismo1991.zip'
sizeZipPath    = dataFolder/'CensimentoRegioni1991.zip'
coordZipPath   = dataFolder/'LimitiRegioni1991.zip'
shpFilePath    = f"zip://{coordZipPath}!Limiti1991_g/Com1991_g/Com
↪ 1991_g_WGS84.shp"

regDataZipPath = dataFolder/'DatiRegioni1991.zip'
simDataZipFile = dataFolder/'DatiSimulazione.zip'

parameters = {
    "population": {
        "text": "S",
        "val": 1
    },
    "attractivity": {
        "text": "λ",
        "val": 1.0
    },
    "convincibility": {
        "text": "α",
        "val": 1.0
    },
    "deviation": {
        "text": "σ",
        "val": 1.0
    },
    "region": {
        "text": "Region selected",
        "val": "region"
    },
    "zetaValue": {
        "text": "ζ",
        "val": 1.0
    },
    "timestep": {
        "text": "Δt",
        "val": 1.0
    },
    "timesteps": {
        "text": "Nt",
        "val": 1
    },
    "iterations": {
        "text": "Ni",
        "val": 1
    },
    "progressBar": {
        "text": "Progress Bar",
        "val": True
    },
    "extraction": {
        "text": "Extract data",
        "val": False
    },
    "analysis": {
        "text": "Network analysis",
        "val": False
    },
    "edgeWeights": {
        "text": "Edge weights",
        "val": False
    },
    "fluctuations": {
```

```
            "text": "Fluctuations",
            "val": True
        },
        "zetaFraction": {
            "text": "Fraction",
            "val": False
        },
        "interactionRule": {
            "text": "Interaction rule",
            "val": "law"
        },
        "pdfPopUp": {
            "text": "Open PDF",
            "val": False
        },
        "TikZConversion": {
            "text": "TikZ Conversion",
            "val": False
        },
        "snapshots": {
            "text": "Ns",
            "val": 100
        },
        "smoothingFactor": {
            "text": "Sf",
            "val": 10
        },
        "parametricStudy": {
            "text": "Parametric study",
            "val": False
        },
        "studiedParameter": {
            "text": "Studied parameter",
            "val": "study"
        },
        "startValuePrmStudy": {
            "text": "Start",
            "val": 1.0
        },
        "endValuePrmStudy": {
            "text": "End",
            "val": 1.0
        },
        "numberPrmStudy": {
            "text": "Nv",
            "val": 1
        }
    }

    regionList = [
        'Piemonte',
        "Valle d'Aosta",
        'Lombardia',
        'Trentino-Alto Adige',
        'Veneto',
        'Friuli-Venezia Giulia',
        'Liguria',
        'Emilia-Romagna',
        'Toscana',
        'Umbria',
        'Marche',
        'Lazio',
        'Abruzzo',
        'Molise',
        'Campania',
        'Puglia',
        'Basilicata',
```

```
    'Calabria',
    'Sicilia',
    'Sardegna',
    'Italia'
]

intRuleList = [
    'λ(rs^α)/(1+rs^α)',         # 0
    'λ(rsk/α)/(1+rsk/α)',       # 1
    'λ(rsk^α)/(1+rsk^α)',       # 2
    'λ[rsk/(1+rsk)]^α',         # 3
]

caseStudies = {
    "selected": "Default",
    "list": {
        "Default": {
            "attractivity": .18,
            "convincibility": .44,
            "deviation": 0.05,
            "region": 20,
            "zetaValue": 0.01,
            "timestep": 0.01,
            "timesteps": int(3e7),
            "iterations": 10,
            "progressBar": False,
            "extraction": False,
            "analysis": False,
            "edgeWeights": True,
            "fluctuations": False,
            "zetaFraction": False,
            "interactionRule": 2,
            "pdfPopUp": False,
            "TikZConversion": True,
            "snapshots": 1000,
            "smoothingFactor": 50,
            "studiedParameter": 1,
            "startValuePrmStudy": .3,
            "endValuePrmStudy": .6,
            "numberPrmStudy": 5,
            "parametricStudy": False
        },
        "λ(rsk/α)/(1+rsk/α)": {
            "attractivity": 0.05,
            "deviation": 0.05,
            "region": 19,
            "zetaValue": 0.1,
            "timestep": 0.01,
            "timesteps": int(5e5),
            "iterations": 15,
            "progressBar": False,
            "extraction": False,
            "analysis": False,
            "edgeWeights": False,
            "fluctuations": True,
            "zetaFraction": False,
            "interactionRule": 1,
            "pdfPopUp": False,
            "TikZConversion": False,
            "snapshots": 100,
            "smoothingFactor": 10,
            "studiedParameter": 0,
            "startValuePrmStudy": 1.0,
            "endValuePrmStudy": 1.0,
            "numberPrmStudy": 1,
            "parametricStudy": False
        },
```

```
        "(1-ζ)efl_k/α+ζefs_k": {
            "attractivity": 0.05,
            "deviation": 0.05,
            "region": 19,
            "zetaValue": 0.1,
            "timestep": 0.01,
            "timesteps": int(1e7),
            "iterations": 15,
            "progressBar": False,
            "extraction": False,
            "analysis": False,
            "edgeWeights": False,
            "fluctuations": True,
            "zetaFraction": True,
            "interactionRule": 1,
            "pdfPopUp": False,
            "TikZConversion": False,
            "snapshots": 100,
            "smoothingFactor": 10,
            "studiedParameter": 0,
            "startValuePrmStudy": 0.01,
            "endValuePrmStudy": 0.1,
            "numberPrmStudy": 5,
            "parametricStudy": True
        },
        "λ(rsk^α)/(1+rsk^α)": {
            "attractivity": 0.1,
            "convincibility": 0.5,
            "deviation": 0.8,
            "region": 19,
            "zetaValue": 0.1,
            "timestep": 0.01,
            "timesteps": int(4e5),
            "iterations": 100,
            "progressBar": False,
            "extraction": False,
            "analysis": False,
            "edgeWeights": True,
            "fluctuations": True,
            "zetaFraction": False,
            "interactionRule": 2,
            "pdfPopUp": False,
            "TikZConversion": False,
            "snapshots": 1000,
            "smoothingFactor": 50,
            "studiedParameter": 1,
            "startValuePrmStudy": 0.1,
            "endValuePrmStudy": 1,
            "numberPrmStudy": 10,
            "parametricStudy": False
        },
        "λ[rsk/(1+rsk)]^α": {
            "attractivity": 0.05,
            "convincibility": 0.3,
            "deviation": 0.05,
            "region": 19,
            "zetaValue": 0.1,
            "timestep": 0.01,
            "timesteps": int(1e7),
            "iterations": 15,
            "progressBar": False,
            "extraction": False,
            "analysis": False,
            "edgeWeights": False,
            "fluctuations": True,
            "zetaFraction": False,
            "interactionRule": 3,
```

```
            "pdfPopUp": False,
            "TikZConversion": False,
            "snapshots": 100,
            "smoothingFactor": 10,
            "studiedParameter": 1,
            "startValuePrmStudy": 0.3,
            "endValuePrmStudy": 1,
            "numberPrmStudy": 3,
            "parametricStudy": False
        }
    }
}

prmStudyList = [
    parameters['attractivity']['text'],
    parameters['convincibility']['text'],
    parameters['zetaValue']['text']
]

regPopDict = { # Italian region sizes in 1991
    'Piemonte':int(4302565),
    "Valle d'Aosta":int(115938),
    'Lombardia':int(8856074),
    'Trentino-Alto Adige':int(890360),
    'Veneto':int(4380797),
    'Friuli-Venezia Giulia':int(1197666),
    'Liguria':int(1676282),
    'Emilia-Romagna':int(3909512),
    'Toscana':int(3529946),
    'Umbria':int(811831),
    'Marche':int(1429205),
    'Lazio':int(5140371),
    'Abruzzo':int(1249054),
    'Molise':int(330900),
    'Campania':int(5630280),
    'Puglia':int(4031885),
    'Basilicata':int(610528),
    'Calabria':int(2070203),
    'Sicilia':int(4966386),
    'Sardegna':int(1648248),
    'Italia':int(56778031)
} # See Table 6.1 on p. 488 of «ISTAT Popolazione e abitazioni 1991
↪ {04-12-2025}.pdf»

workersShMTemplate = {
    'handles':[],
    'parameters':{
        'Mdt': None,
        'wOdt': None,
        'wI': None,
        'di': None,
        'idi': None,
        'ns': None
    },
    'gui':{
        'progress': np.int64,
        'elapsed': np.float64,
        'done': np.int8
    }
}


### Main classes and function ###

@dataclass(eq=False)
class Parameter:
    text: Any = None
```

```
347         val: Any = None
348         var: Any = None
349         lbl: Any = None
350         wid: Any = None
351         frame: Any = None
352         list: Any = None
353         cbid: Any = None
354 
355     class Parameters():
356         def __init__(self,**kwargs):
357             for text,value in kwargs.items():
358                 setattr(self,text,value)
359 
360     class ComboBoxList():
361         def __init__(self,attribute,list):
362             attribute.list = list
363             self.list = list
364             self.code = {
365                 r:i for i,r in enumerate(list)
366             }
367 
368     def CopyWorkerShMTemplate(): return deepcopy(workersShMTemplate)
```

Il secondo modulo è `libGUIs.py` nel List. A.3 che definisce due principali interfacce grafiche:

1. una per raccogliere i parametri (Fig. A.1) e
2. l'altra per le barre di progressione (Fig. A.2).

Listato A.3: Modulo `libGUIs.py`.

```
1   # Library to create Graphics Users Interfaces
2 
3   import tkinter as tk
4   from tkinter import ttk
5 
6   import numpy as np
7 
8   import libParameters as libP
9   import libData as libD
10 
11 
12  ### Main class ###
13 
14  class ParametersGUI(tk.Tk):
15      def __init__(self):
16          #region Window
17          # Resets the default root before creating the GUI
18          tk._default_root = None
19 
20          super().__init__() # Main window
21          self.title('City Size Model')
22          self.resizable(False,False) # Disable resizing the window
23 
24          self.bind('<Escape>',lambda e:
            ↪  self.SetSimulationState(False))
25          self.bind('<space>',lambda e: self.SetSimulationState(True))
26          #endregion
27 
28          #region Parameters
29          libD.SetParameters(self)
30 
31          self.regPopDict = libP.regPopDict
```

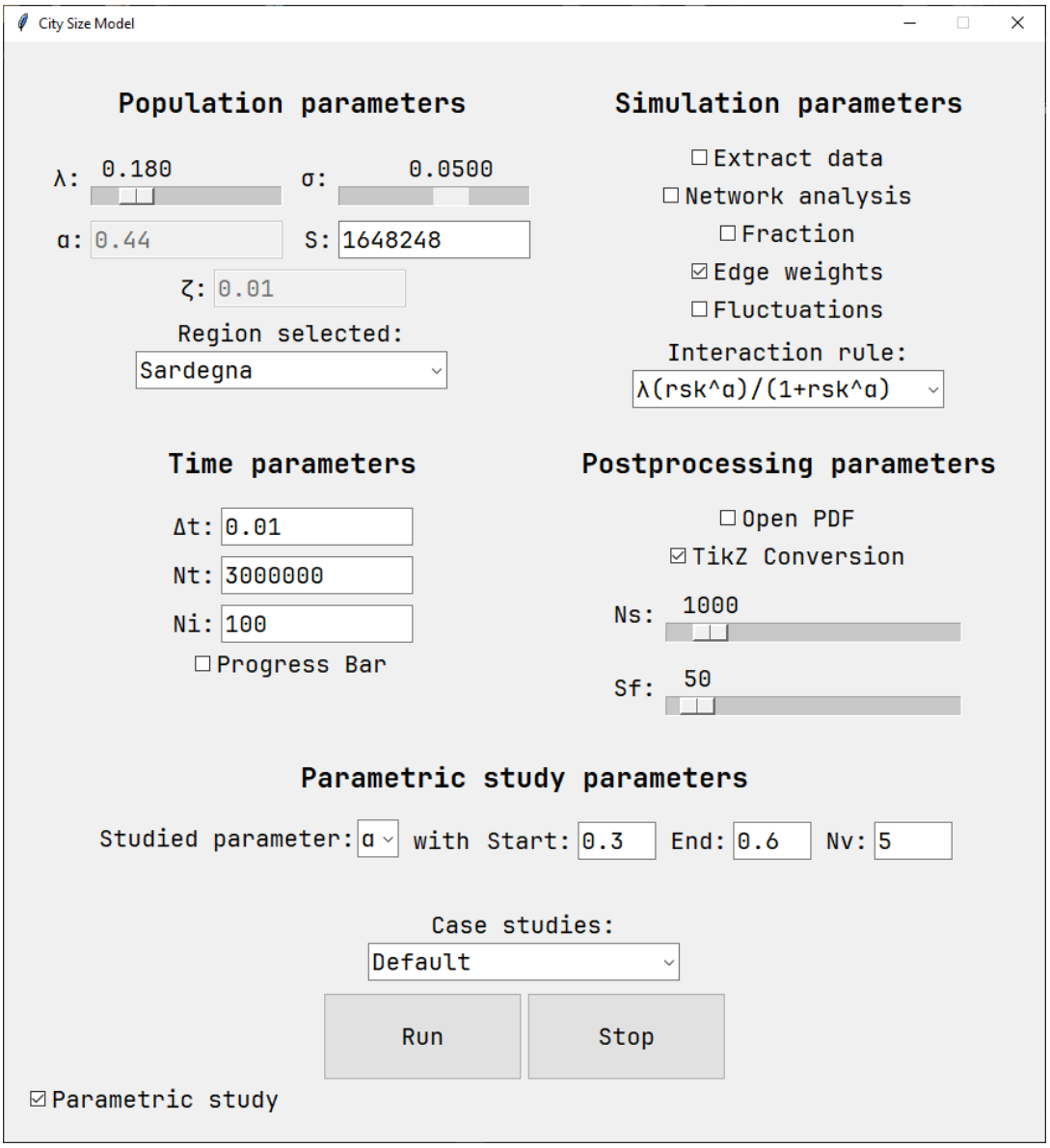

Figura A.1: Interfaccia grafica per raccogliere i parametri.

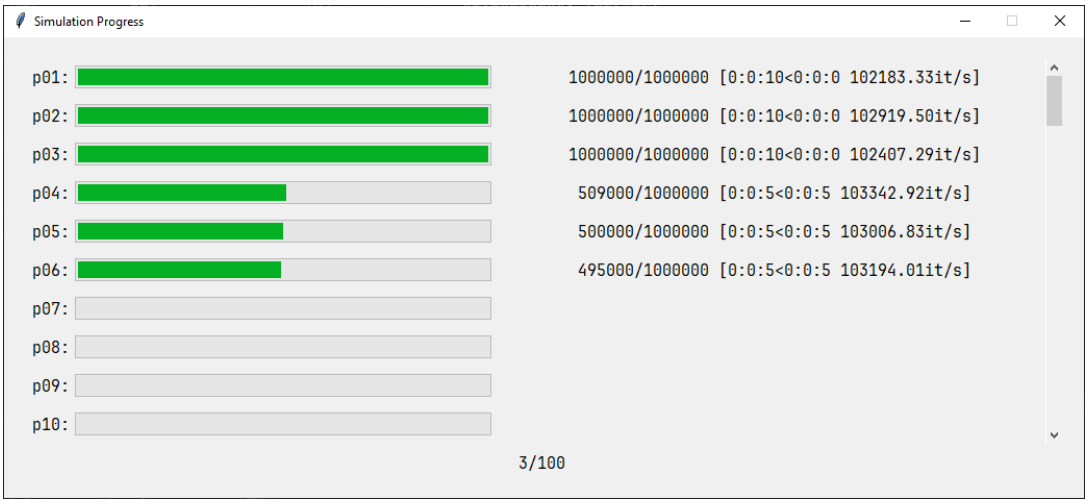

Figura A.2: Interfaccia grafica per le barre di progressione.

```
32            self.regionList = libP.ComboBoxList(
33                self.region,
34                libP.regionList
35            )
36            self.intRuleList = libP.ComboBoxList(
37                self.interactionRule,
38                libP.intRuleList
39            )
40            self.studiedPrmList = libP.ComboBoxList(
41                self.studiedParameter,
42                libP.prmStudyList
43            )
44
45            self.simFlag =
              ↪  libP.Parameter(var=tk.BooleanVar(value=False))
46            #endregion
47
48            #region Parameter frames
49            pad = 20; pad = (pad,pad)
50            mainFrame = Frame(self,pad=pad,sticky='nsew')
51            pad = 15; pad = (1.25*pad,pad)
52
53            popPrmFrame = Frame(
54                mainFrame,
55                pad=pad,
56                title='Population parameters'
57            )
58            simPrmFrame = Frame(
59                mainFrame,
60                pad=pad,
61                title='Simulation parameters'
62            )
63            timePrmFrame = Frame(
64                mainFrame,
65                pad=pad,
66                title='Time parameters'#,labelWidth=3
67            )
68            ppcPrmFrame = Frame(
69                mainFrame,
70                pad=pad,
71                title='Postprocessing parameters'
72            )
73            pspPrmFrame = Frame(
74                mainFrame,
75                pad=pad,
76                title='Parametric study parameters',
77                colSpan=mainFrame.nCol,
78                nCol=5,
79                removed=True
80            ); self.pspPrmFrame = pspPrmFrame
81            buttonFrame = Frame(
82                mainFrame,
83                pad=(pad[0],(1.5*pad[1],0)),
84                colSpan=mainFrame.nCol
85            )
86            parametricStudyFrame = Frame(
87                mainFrame,
88                colSpan=mainFrame.nCol,
89                sticky='sw'
90            )
91            #endregion
92
93            #region Population parameters
94            popPrmFrame.LabelSlider(
95                self.attractivity,1e-3,(0,1),
96                extremes=(False,False)
97            )
```

```
98          popPrmFrame.LabelSlider(
99              self.deviation,1e-4,(1e-4,1),
100             extremes=(True,False)
101         )
102         self.SetDeviationUpperLimit()
103         popPrmFrame.LabelEntry(self.convincibility)
104
105         popPrmFrame.LabelEntry(self.population)
106         self.attractivity.var.trace_add('write',self.SetDeviationU
            ↪ pperLimit)
107
108         popPrmFrame.LabelEntry(self.zetaValue,colSpan=2)
109
110         popPrmFrame.LabelComboBox(self.region)
111         self.region.var.trace_add('write',self.SetPopulation)
112         #endregion
113
114         #region Simulation parameters
115         simPrmFrame.CheckBox(self.extraction)
116         simPrmFrame.CheckBox(self.analysis)
117         simPrmFrame.CheckBox(self.zetaFraction)
118         simPrmFrame.CheckBox(self.edgeWeights)
119         simPrmFrame.CheckBox(self.fluctuations)
120         self.zetaFraction.var.trace_add('write',self.SetZetaFracti
            ↪ onState)
121         self.fluctuations.var.trace_add('write',self.SetDeviationS
            ↪ tate)
122
123         simPrmFrame.LabelComboBox(self.interactionRule)
124         self.interactionRule.var.trace_add('write',self.Interactin
            ↪ gRuleCallBack)
125         #endregion
126
127         #region Time parameters
128         timePrmFrame.LabelEntry(self.timestep,colSpan=timePrmFrame
            ↪ .nCol)
129         timePrmFrame.LabelEntry(self.timesteps,colSpan=timePrmFram
            ↪ e.nCol)
130         timePrmFrame.LabelEntry(self.iterations,colSpan=timePrmFra
            ↪ me.nCol)
131         timePrmFrame.CheckBox(self.progressBar)
132         #endregion
133
134         #region Postprocessing parameters
135         ppcPrmFrame.CheckBox(self.pdfPopUp)
136         ppcPrmFrame.CheckBox(self.TikZConversion)
137         ppcPrmFrame.LabelSlider(
138             self.snapshots,1,
139             (1,self.timesteps.var.get()),
140             sliderLength=250,
141             colSpan=ppcPrmFrame.nCol
142         ) # Number of screenshots [not considering the initial
            ↪ state]
143         self.timesteps.var.trace_add(
144             'write',lambda *args: self.SetSliderUpperLimit(
145                 self.snapshots,self.timesteps
146             )
147         )
148         ppcPrmFrame.LabelSlider(
149             self.smoothingFactor,1,
150             (1,self.snapshots.var.get()),
151             sliderLength=250,
152             colSpan=ppcPrmFrame.nCol
153         )
154         self.snapshots.var.trace_add(
155             'write',lambda *args: self.SetSliderUpperLimit(
```

```
156                 self.smoothingFactor,self.snapshots
157             )
158         )
159         #endregion
160 
161         #region Parametric study parameters
162         pspPrmFrame.LabelComboBox(
163             self.studiedParameter,
164             colSpan=1,
165             cbWdith=1,
166             # pad=((5,0),(0,0))
167         )
168         self.studiedParameter.var.trace_add(
169             'write',self.SetStudiedParameterState
170         )
171         width = 5
172         pspPrmFrame.Label(libP.Parameter('with'),pad=pspPrmFrame.p
            ↪ ad)
173         pspPrmFrame.LabelEntry(self.startValuePrmStudy,entryWidth=
            ↪ width)
174         pspPrmFrame.LabelEntry(self.endValuePrmStudy,entryWidth=wi
            ↪ dth)
175         pspPrmFrame.LabelEntry(self.numberPrmStudy,entryWidth=width)
176 
177         parametricStudyFrame.CheckBox(self.parametricStudy)
178         self.parametricStudy.var.trace_add(
179             'write',self.ShowParametricStudyFrame
180         )
181         #endregion
182 
183         #region Buttons and case studies
184         self.dictCS, selectedCS, listCS = libD.LoadCaseStudies(self)
185         self.caseStudy = libP.Parameter(
186             'Case studies',
187             selectedCS,
188             list=listCS,
189         )
190 
191         buttonFrame.LabelComboBox(
192             self.caseStudy,
193             colSpan=buttonFrame.nCol,
194             pad=((0,0),(0,10))
195         )
196         self.caseStudy.var.trace_add('write',self.SetCaseStudy)
197 
198         buttonFrame.Button('Run',lambda:
            ↪ self.SetSimulationState(True))
199         buttonFrame.Button('Stop',lambda:
            ↪ self.SetSimulationState(False))
200         #endregion
201 
202         #region GUI call
203         self.SetCaseStudy()
204 
205         CentreGUI(self)
206         self.deiconify()
207 
208         self.mainloop()
209         #endregion
210 
211     # Callbacks
212     def SetDeviationState(self,*args):
213         checked = self.fluctuations.var.get()
214         self.deviation.wid['state'] = 'normal' if checked else
            ↪ 'disabled'
215         self.deviation.wid['sliderrelief'] = 'raised' if checked
            ↪ else 'flat'
```

```
216
217         def SetDeviationUpperLimit(self,*args):
218             try:
219                 l = self.attractivity.var.get()
220                 if isinstance(l,float):
221                     res = self.deviation.wid['resolution']
222                     if l >= 1:
223                         l=1-res
224                         self.attractivity.var.set(l)
225                     self.deviation.wid['to'] = ((1-l)/10//res+1)*res
226             except Exception:
227                 pass
228
229         def SetPopulation(self,*args):
230             nameReg = self.region.var.get()
231             popReg = self.regPopDict[nameReg]
232             self.population.var.set(popReg)
233
234         def SetConvincibility(self,*args):
235             if self.intRuleList.code[self.interactionRule.var.get()] ==
                ↪ 1:
236                 l = self.attractivity.var.get()
237                 self.convincibility.var.set(np.round(l/.01-1,decimals=
                    ↪ 2))
238             else:
239                 l = self.attractivity.var.get()
240                 self.convincibility.var.set(
241                     np.round(np.log10(.01/l)/np.log10(.5),decimals=2)
242                 )
243
244         def SetSimulationState(self,state):
245             self.simFlag.var.set(state)
246             self.destroy() # Close the window after any button is
                ↪ pressed
247
248         def InteractingRuleCallBack(self,*args):
249             if self.intRuleList.code[self.interactionRule.var.get()] in
                ↪ (1,3):
250                 self.EnableCallBack(self.attractivity,self.SetConvinci
                    ↪ bility)
251             else:
252                 self.DisableCallBack(self.attractivity)
253
254         def SetZetaFractionState(self,*args):
255             checked = self.zetaFraction.var.get()
256
257             if checked:
258                 self.zetaValue.wid['state'] = 'normal'
259                 self.studiedParameter.wid['values'] =
                    ↪ self.studiedPrmList.list
260             else:
261                 self.zetaValue.wid['state'] = 'disabled'
262                 self.studiedParameter.wid['values'] =
                    ↪ self.studiedPrmList.list[:-1]
263
264             if self.parametricStudy.var.get():
                ↪ self.SetStudiedParameterState()
265
266         def SetSliderUpperLimit(self,slider,ref):
267             try:
268                 val = ref.var.get()
269             except Exception:
270                 return
271             slider.wid["to"] = val if val<1e4 else 1e4
272
273         def ShowParametricStudyFrame(self,*args):
274             frame = self.pspPrmFrame
```

```
275
276             if self.parametricStudy.var.get(): frame.grid()
277             else: frame.grid_remove()
278             self.SetStudiedParameterState()
279
280             self.ResizeWindow()
281
282         def SetStudiedParameterState(self,*args):
283             state = 'normal'
284             self.attractivity.wid['state'] = state
285             self.attractivity.wid['sliderrelief'] = 'raised'
286             self.convincibility.wid['state'] = state
287
288             value = self.studiedPrmList.code[self.studiedParameter.var
                ↪  .get()]
289             if self.zetaFraction.var.get() == 0:
290                 if value == 2: self.studiedParameter.var.set(self.stud
                    ↪  iedParameter.list[0])
291             else:
292                 self.zetaValue.wid['state'] = state
293
294             checked = self.parametricStudy.var.get()
295             state = 'disabled' if checked else 'normal'
296
297             if checked:
298                 match value:
299                     case 0:
300                         self.attractivity.wid['state'] = state
301                         self.attractivity.wid['sliderrelief'] = 'flat'
302                     case 1:
303                         self.convincibility.wid['state'] = state
304                     case 2:
305                         self.zetaValue.wid['state'] = state
306
307         def SetCaseStudy(self,*args):
308             caseStudy = self.caseStudy.var.get()
309             for prm,val in self.dictCS[caseStudy].items():
310                 prm.var.set(val)
311
312             self.InteractingRuleCallBack()
313             if self.intRuleList.code[self.interactionRule.var.get()] ==
                ↪  1:
314                 self.SetConvincibility()
315             self.ShowParametricStudyFrame()
316             self.SetStudiedParameterState()
317
318         # Functions
319         def GatherParameters(self):
320             parameters = {}
321             for attribute,value in self.__dict__.items():
322                 if isinstance(value,libP.Parameter):
323                     parameters[attribute] = value.var.get()
324                     # if value.var is not None:
325                     # setattr(self,name,value.var.get())
326                     # value.val = value.var.get()
327
328             parameters = libP.Parameters(**parameters)
329
330             parameters.region = self.regionList.code[
331                 parameters.region
332             ]+1
333             parameters.interactionRule = self.intRuleList.code[
334                 parameters.interactionRule
335             ]
336             parameters.studiedParameter = self.studiedPrmList.code[
337                 parameters.studiedParameter
338             ]
```

```
339
340             return parameters
341
342         def ResizeWindow(self,center=True):
343             self.update_idletasks()
344
345             ww = self.winfo_reqwidth()
346             wh = self.winfo_reqheight()
347
348             if center:
349                 sw = self.winfo_screenwidth()
350                 sh = self.winfo_screenheight()
351
352                 x = max(0,(sw-ww)//2)
353                 y = max(0,(sh-wh)//2)
354
355                 # self.after(100,lambda:
                    ↪  self.geometry(f"{ww}x{wh}+{x}+{y}"))
356                 self.geometry(f"{ww}x{wh}+{x}+{y}")
357             else:
358                 # self.after(100,lambda: self.geometry(f"{ww}x{wh}"))
359                 self.geometry(f"{ww}x{wh}")
360
361         def EnableCallBack(self,prm,clb):
362             if prm.cbid is None:
363                 prm.cbid = prm.var.trace_add('write',clb)
364
365         def DisableCallBack(self,prm):
366             if prm.cbid is not None:
367                 prm.var.trace_remove('write',prm.cbid)
368                 prm.cbid = None
369
370     class ProgressBarsGUI(tk.Tk):
371         def __init__(
372             self,Ni,Nt,
373             sid=None,Nv=None,
374             shmPrm=None,
375             LoadShM=None
376         ):
377             #region Window
378             super().__init__()
379
380             self.resizable(False,False)
381             self.grid_rowconfigure(0,weight=1)
382             self.grid_columnconfigure(0,weight=1)
383
384             self.title("Simulation Progress")
385             self.iconify() # Start minimized
386             #endregion
387
388             #region Parameters
389             self.shm = []
390             for key in libP.workersShMTemplate['gui'].keys():
391                 shm, arr = LoadShM(shmPrm[key])
392                 self.shm.append(shm)
393                 setattr(self,key,arr)
394
395             self.Ni = Ni
396             self.Nt = Nt
397             # self.d = [False]*Ni
398             #endregion
399
400             #region Frames
401             shown = 10
402
403             padmf = 20; pad = (padmf,padmf)
404             if Ni>shown:
```

```
405                 mainFrame = Frame(
406                     self,
407                     nCol=2,
408                     pad=pad,
409                     normalFontStyle=('JetBrains Mono',11),
410                     sticky="nsew"
411                 )
412                 barsFrame = mainFrame.ScrollFrame(
413                     nCol=1,
414                     normalFontStyle=('JetBrains Mono',11)
415                 )
416             else:
417                 mainFrame = Frame(
418                     self,
419                     nCol=1,
420                     pad=pad,
421                     normalFontStyle=('JetBrains Mono',11)
422                 )
423                 barsFrame = mainFrame
424 
425             padpb = 5; pad = (padpb,padpb)
426             self.frames = []; self.bars = []; self.status = []
427             for i in range(Ni):
428                 progressBarFrame = Frame(barsFrame,nCol=3,pad=pad)
429 
430                 label = f"p{'0' if i+1<10 else ''}{i+1}" # Process
431                 setattr(self,label,libP.Parameter(text=label+':'))
432                 progressBarFrame.Label(getattr(self,label),width=4)
433 
434                 progressBarFrame.ProgressBar(getattr(self,label),Nt)
435 
436                 status = f"s{'0' if i+1<10 else ''}{i+1}"
437                 setattr(self,status,libP.Parameter(text=''))
438                 progressBarFrame.Label(
439                     getattr(self,status),
440                     width=60,
441                     pad=((0,0),(0,0)),
442                     anchor='c'
443                 )
444 
445                 self.frames.append(progressBarFrame)
446                 self.bars.append(getattr(self,label).wid)
447                 self.status.append(getattr(self,status).lbl)
448 
449             padcf = padpb
450             completionFrame = Frame(
451                 mainFrame,
452                 colSpan=mainFrame.nCol,
453                 # sticky='se',
454                 pad=((0,0),(padcf,0))
455             )
456             self.completion = libP.Parameter(
457                 text=f'/{Ni}' if Nv is None else fr'/{Ni} {sid}/{Nv}'
458             )
459             completionFrame.Label(self.completion)
460 
461             if Ni>shown:
462                 self.update_idletasks()
463                 guiw = barsFrame.winfo_width()+2*padmf
464                 guih = (
465                     (progressBarFrame.winfo_height()+2*padpb)*shown
466                     +2*padmf+completionFrame.winfo_height()+padcf
467                 )
468                 self.geometry(f'{guiw}x{guih}')
469             #endregion
470 
471             #region GUI [after]call
```

```
            self.centered = False
            self.bind("<Map>",lambda e:self.CenterWhenActive())

            self.after(100,self.PoolInfo)
            #endregion

        def PoolInfo(self):
            for p in range(self.Ni):
                nt = int(self.progress[p])
                el = float(self.elapsed[p])

                if el != 0:
                    self.bars[p]['value'] = nt
                    ips = max(1,nt)/el # Iterations per second

                    self.status[p]['text'] = (
                        f'{nt}/{self.Nt} ['
                        f'{TimeFormatter(nt/ips)}<'
                        f'{TimeFormatter((self.Nt-nt)/ips)} '
                        # f'[{nt/ips:.2f}<{(self.Nt-nt)/ips:.2f}, '
                        f'{ips:.2f}it/s]'
                    )

            self.completion.lbl['text']=(
                f'{np.sum(self.done)}'+self.completion.text
            )

            if np.all(self.done):
                for shm in self.shm: shm.close()
                self.destroy()
            else:
                self.after(100,self.PoolInfo)

        def CenterWhenActive(self):
            if self.state() != "normal":
                return

            if not self.centered:
                # self.update_idletasks()
                CentreGUI(self)
                self.centered=True


    ### Auxiliary classes ###

    class Frame(ttk.Frame):
        def __init__(
            self,parent,
            nCol=2,
            colSpan=1,
            title=None,
            sticky='n',
            pos=None,
            pad=((0,0),(0,0)),
            labelWidth=None,
            normalFontStyle=None,
            titleFontStyle=None,
            removed=False
        ):
            super().__init__(parent,borderwidth=0)

            self.columnconfigure(nCol)
            self.nCol = nCol

            self.cCol = 0
            self.cRow = 0
```

```
539             self.pCol = [0,0]
540             self.pRow = [0,0]
541             self.pad = (self.pCol,self.pRow)
542 
543             self.labelWidth = labelWidth
544             self.entryWidth = 13
545             self.comboBoxWidth = 20
546 
547             if normalFontStyle is None:
548                 if isinstance(parent,Frame):
549                     self.normalFontStyle = parent.normalFontStyle
550                 else:
551                     self.normalFontStyle = ('JetBrains Mono',15)
552             else:
553                 self.normalFontStyle = normalFontStyle
554 
555             if titleFontStyle is None:
556                 if isinstance(parent,Frame):
557                     self.titleFontStyle = parent.titleFontStyle
558                 else:
559                     self.titleFontStyle = ('JetBrains Mono',17,'bold')
560             else:
561                 self.titleFontStyle = titleFontStyle
562 
563             if title is not None:
564                 self.SetTitle(title)
565 
566             if isinstance(parent,Frame):
567                 if pos is None:
568                     pos = (parent.cRow,parent.cCol)
569                 parent.NextColumn(colSpan)
570             else:
571                 pos = (0,0) # Main frame position
572 
573             self.grid(
574                 row=pos[0],
575                 column=pos[1],
576                 columnspan=colSpan,
577                 padx=pad[0],
578                 pady=pad[1],
579                 sticky=sticky
580             )
581             if removed: self.grid_remove()
582 
583         # Functions
584         def NextRow(self):
585             self.cRow +=1
586 
587             if self.cRow == 0:
588                 self.pRow[0] = 0
589             else:
590                 self.pRow[0] = 5
591 
592         def SetRow(self,row): self.cRow = row
593 
594         def NextColumn(self,colSpan=1):
595             if self.cCol+colSpan < self.nCol:
596                 self.cCol += colSpan
597             else:
598                 self.cCol = 0
599                 self.NextRow()
600 
601             if self.cCol == 0:
602                 self.pCol[1] = 0
603             else:
604                 self.pCol[1] = 10
605 
```

```
606         def SeteColumn(self,column): self.cCol = column
607 
608         def SetTitle(self,title):
609             label = ttk.Label(
610                 self,
611                 text=f'{title}',
612                 font=self.titleFontStyle,
613                 anchor='s',
614             )
615             label.grid(
616                 row=self.cRow,
617                 column=self.cCol,
618                 columnspan=self.nCol,
619                 pady=(0,12.5)
620             )
621             self.NextRow()
622 
623         def CheckDataType(self,data):
624             if isinstance(data.val,float):
625                 data.var = tk.DoubleVar(value=data.val)
626             elif isinstance(data.val,int):
627                 data.var = tk.IntVar(value=data.val)
628             elif isinstance(data.val,str):
629                 data.var = tk.StringVar(value=data.val)
630             else:
631                 data.var = tk.BooleanVar(value=data.val)
632 
633         # Default widgets
634         def Label(
635             self,
636             data,
637             width=None,
638             anchor='e',
639             sticky=None,
640             colSpan=1,
641             pos=None,
642             pad=((0,3),2)
643         ):
644             if width is None: width=self.labelWidth
645             data.lbl = ttk.Label(
646                 master=self,
647                 text=data.text,
648                 font=self.normalFontStyle,
649                 anchor=anchor,
650                 width=width
651             )
652 
653             row = self.cRow if pos is None else pos[0]
654             col = self.cCol if pos is None else pos[1]
655 
656             data.lbl.grid(
657                 row=row,
658                 column=col,
659                 columnspan=colSpan,
660                 padx=pad[0],pady=pad[1],
661                 sticky=sticky
662             )
663             self.NextColumn(colSpan)
664 
665         def Entry(self,data,width=None):
666             self.CheckDataType(data)
667 
668             if width is None: width = self.entryWidth
669 
670             data.wid = ttk.Entry(
671                 master=self,
672                 textvariable=data.var,
```

```
673                 font=self.normalFontStyle,
674                 width=width,
675             )
676             data.wid.grid(
677                 row=self.cRow,
678                 column=self.cCol,
679                 pady=2
680             )
681             self.NextColumn()
682 
683         def Slider(
684             self,
685             data,
686             bounds,
687             res,
688             extremes,
689             length
690         ):
691             if length is None:
692                 # Measure the character length in pixels
693                 fieldInput = ttk.Entry(
694                     master=self,
695                     font=self.normalFontStyle,
696                     width=self.entryWidth
697                 )
698                 fieldInput.grid(row=0,column=1)
699                 l = fieldInput.winfo_reqwidth()
700                 fieldInput.destroy()
701             else:
702                 l = length
703 
704             # Create the slider widget
705             self.CheckDataType(data)
706             data.wid = tk.Scale(
707                 master=self,
708                 variable=data.var,
709                 from_=bounds[0]+(not extremes[0])*res,
710                 to=bounds[1]-(not extremes[0])*res,
711                 resolution=res,
712                 orient="horizontal",
713                 font=self.normalFontStyle,
714                 length=l
715             )
716             data.wid.grid(
717                 row=self.cRow,
718                 column=self.cCol,
719                 pady=2
720             )
721             self.NextColumn()
722 
723         def ComboBox(
724             self,
725             data,
726             colSpan=1,
727             state='readonly',
728             width=None
729         ):
730             self.CheckDataType(data)
731 
732             if width is None: width = self.comboBoxWidth
733             data.wid = ttk.Combobox(
734                 master=self,
735                 values=data.list,
736                 textvariable=data.var,
737                 state=state,
738                 font=self.normalFontStyle,
739                 width=width
```

```
740             )
741             data.wid.grid(
742                 row=self.cRow,
743                 column=self.cCol,
744                 columnspan=colSpan
745             )
746             self.NextColumn(colSpan)
747 
748         def CheckBox(self,data,colSpan=2):
749             style = ttk.Style()
750             style.configure(
751                 'ckb.TCheckbutton',
752                 font=self.normalFontStyle
753             )
754 
755             self.CheckDataType(data)
756             data.wid = ttk.Checkbutton(
757                 master=self,
758                 text=data.text,
759                 variable=data.var,
760                 style='ckb.TCheckbutton',
761             )
762 
763             # frameWidth = self.winfo_width()
764             data.wid.grid(
765                 row=self.cRow,
766                 column=self.cCol,
767                 columnspan=colSpan,
768                 # sticky='w',
769                 # padx=(frameWidth/3,0)
770             )
771 
772             self.NextColumn(colSpan)
773 
774         def Button(self,text,callBack):
775             style = ttk.Style()
776             style.configure(
777                 'btt.TButton',
778                 padding=(20,20),
779                 width=10,
780                 font=self.normalFontStyle
781             )
782             button = ttk.Button(
783                 self,
784                 text=text,
785                 command=callBack,
786                 style="btt.TButton"
787             )
788             button.grid(
789                 row=self.cRow,
790                 column=self.cCol,
791                 padx=2
792             )
793             self.NextColumn()
794 
795         def ProgressBar(self,data,max,width=400):
796             data.wid = ttk.Progressbar(
797                 master=self,
798                 maximum=max,
799                 length=width
800             )
801             data.wid.grid(
802                 row=self.cRow,
803                 column=self.cCol,
804             )
805             self.NextColumn()
806 
```

```
807         # Compound widgets
808         def LabelEntry(
809             self,data,
810             colSpan=1,
811             pos=None,
812             lblWidth=None,
813             pad=None,
814             entryWidth=None,
815         ):
816             if lblWidth is None: lblWidth = self.labelWidth
817             if pad is None: pad = (self.pCol,self.pRow)
818 
819             data.frame = Frame(
820                 self,colSpan=colSpan,
821                 pos=pos,
822                 pad=pad,
823                 labelWidth=lblWidth
824             )
825 
826             data.text += ':'
827             data.frame.Label(data,lblWidth)
828             data.text = data.text[:-1]
829             data.frame.Entry(data,entryWidth)
830 
831         def LabelSlider(
832             self,data,res,bounds,
833             extremes=(True,True),
834             sliderLength=None,
835             pos=None,
836             labelWidth=None,
837             colSpan=1
838         ):
839             data.frame = Frame(
840                 self,
841                 colSpan=colSpan,
842                 pos=pos,
843                 pad=(self.pCol,self.pRow)
844             )
845 
846             data.text += ':'
847             data.frame.Label(
848                 data,
849                 width=labelWidth
850             )
851             data.text = data.text[:-1]
852             data.frame.Slider(
853                 data,
854                 bounds,
855                 res,
856                 extremes,
857                 sliderLength
858             )
859 
860         def LabelComboBox(
861             self,
862             data,
863             colSpan=2,
864             pos=None,
865             # pad=((0,0),(10,0)),
866             pad=None,
867             cbWdith=None
868         ):
869             if pad is None: pad = (self.pCol,self.pRow)
870 
871             data.frame = Frame(
872                 self,
873                 pos=pos,
```

```
874                 colSpan=colSpan,
875                 pad=pad
876             )
877 
878             data.text += ':'
879             data.frame.Label(
880                 data,
881                 colSpan=colSpan,
882                 pad=((0,0),(0,0)),
883                 anchor='s'
884             )
885             data.text = data.text[:-1]
886             data.frame.ComboBox(data,colSpan,width=cbWdith)
887 
888         def ScrollFrame(self,*args,**kwargs):
889             canvas = tk.Canvas(self,highlightthickness=0)
890             scrollbar = ttk.Scrollbar(
891                 self,
892                 orient='vertical',
893                 command=canvas.yview
894             )
895             canvas.configure(yscrollcommand=scrollbar.set)
896 
897             scrollbar.grid(row=0,column=1,sticky="ns")
898             canvas.grid(row=0,column=0,sticky="nsew")
899             self.NextRow()
900 
901             self.columnconfigure(0,weight=1)
902             self.rowconfigure(0,weight=1)
903 
904             frame = Frame(canvas,*args,**kwargs)
905 
906             window = canvas.create_window(
907                 (0,0),
908                 window=frame,
909                 anchor='nw'
910             )
911 
912             frame.bind(
913                 "<Configure>",
914                 lambda e:self.FrameConfiguration(canvas)
915             )
916             canvas.bind(
917                 "<Configure>",
918                 lambda e:self.CanvasConfiguration(e,canvas,window)
919             )
920 
921             canvas.bind_all("<MouseWheel>",lambda
                ↪  e:self.ScrollMotion(e,canvas))
922 
923             return frame
924 
925         # CallBacks
926         def FrameConfiguration(self,canvas):
927             canvas.configure(scrollregion=canvas.bbox("all"))
928 
929         def CanvasConfiguration(self,event,canvas,window):
930             canvas.itemconfig(window,width=event.width)
931 
932         def ScrollMotion(self,event,canvas):
933             canvas.yview_scroll(int(-event.delta/120),"units")
934 
935     def CentreGUI(window):
936         # Get screen width and height
937         sw = window.winfo_screenwidth()
938         sh = window.winfo_screenheight()
939 
```

```
940         # Update window size
941         window.update_idletasks()
942 
943         # Measure window size
944         ww = window.winfo_width()
945         wh = window.winfo_height()
946 
947         # Calculate new centred coordinates
948         x = max(0,(sw-ww)//2)
949         y = max(0,(sh-wh)//2)
950 
951         # Set geometry
952         window.geometry(f"{ww}x{wh}+{x}+{y}")
953 
954     def TimeFormatter(seconds):
955         m, s = divmod(seconds,60)
956         h, m = divmod(m,60)
957         return f"{h:.0f}:{m:.0f}:{s:.0f}"
```

Il terzo modulo è `libData.py` nel List. A.4 che serve specialmente per estrarre i dati ISTAT [12–14]; in particolare è qui che si costruiscono le matrici d'adiacenza unitarie e pesate.

Listato A.4: Modulo `libData.py`.

```
1   # Library to handle [especially writing and reading] data
2 
3   import zipfile as zf
4   from zipfile import ZipFile as ZF
5 
6   from io import TextIOWrapper as tiow
7   from io import BytesIO as bio
8   from io import StringIO as sio
9 
10  import pandas as pd
11  import csv,json
12  import geopandas as gpd
13 
14  import numpy as np
15 
16  import libParameters as libP
17 
18 
19  ### Main functions ###
20 
21  def ExtractRegionData():
22      dictMun,dictReg = ReadMunRegCodes()
23 
24      BuildAdjacencyMatrices(dictMun,dictReg)
25      BuildSizeDistributions(dictReg)
26 
27      WriteRegionData(dictReg)
28 
29  def LoadRegionData(code):
30      el = {
31          'A':'.txt',
32          'W':'.txt',
33          'li2Name':'.json',
34          'li2Coord':'.txt',
35          'name2li':'.json',
36          'sizeDistr':'.txt',
37          'Nc':'.json'
38      }
39      parameters = {}
```

```
40
41      with ZF(libP.regDataZipPath) as z:
42          for data,ext in el.items():
43              path = f'{code:02d}/{data}{ext}'
44              with z.open(path,'r') as f:
45                  match data:
46                      case 'A' | 'W' | 'li2Coord' | 'sizeDistr':
47                          parameters[data] = np.loadtxt(
48                              tiow(f,encoding='utf-8'),
49                              delimiter=",",dtype=int
50                          )
51
52                      case 'li2Name':
53                          parameters[data] = {
54                              int(k): v for k, v in
                                ↪  json.load(f).items()
55                          } # This is necessary since the keys of a
                            ↪  «.json» file are always strings, hence
                            ↪  they have to be converted to integer if
                            ↪  originally they were such
56
57                      case 'Nc' | 'name2li':
58                          parameters[data] = json.load(f)
59
60      return libP.Parameters(**parameters)
61
62
63  ### Auxiliary functions ###
64
65  def xls2csv(
66      xlsFile,
67      zipFile=None
68  ):
69      if zipFile is None:
70          b = xlsFile
71      else:
72          with ZF(zipFile) as z:
73              b = z.read(xlsFile)  # Extract raw bytes from
                ↪  «elencom91.xls»
74
75      streamXls = bio(b)  # Create a binary stream in RAM from the
        ↪  bytes «b»
76      df = pd.read_excel( # Read the Excel content
77          streamXls,      # from «streamXLS»
78          dtype=str,      # treating columns as strings
79          engine="xlrd",  # using «xlrd»
80          sheet_name=0    # for only the first worksheet
81      ) # «df» stands for «DataFrame» from «pandas.DataFrame»
82      streamCsv = sio()   # Allocate a text buffer in RAM that
        ↪  behaves like a writable text file (UTF-8 encoding)
83      df.to_csv(          # Serialize «df» as «.csv»
84          streamCsv,      # into «streamCSV» and
85          index=False     # without the DataFrame's row numbers
86      )
87      streamCsv.seek(0)   # Reset the text buffer cursor to the start
        ↪  so it can be read from the beginning
88
89      return streamCsv
90
91  def ReadMunRegCodes(): # Read Municipality-Region Codes
92      # Conversion of «fileName» from the old format «.xls» to a more
        ↪  manageable «.csv»
93      matrixCSV = xls2csv('elencom91.xls',libP.matrixZipPath)
94
```

```
95      # These two dictionary are necessary to link muicipalities and
        ↪  regions via their codes defined in «file», which will be
        ↪  useful later on to extract the actual data for the
        ↪  adjacency matrices
96      dictMun = {} # Dictionary to link municipality codes with
        ↪  region codes
97      dictReg = {
98          i+1:{
99              'li2Name':{},  # Dictionary to link local indices with
                ↪  the municipality name
100             'li2Coord':{}, # Dictionary to link local indices with
                ↪  the municipality representative point
101             'name2li':{},  # Dictionary to link local indices with
                ↪  the municipality name
102             'code2li':{},  # Dictionary to link municipality codes
                ↪  with local indices
103             'Nc':0  # Number of cities in a region
104         } for i in range(21)
105     } # The index 21 is arbitrarily associated to Italy viewed as
        ↪  the 21th region, hence its local index is actually the
        ↪  global one
106
107     gdf = gpd.read_file(libP.shpFilePath).set_index('PRO_COM_T').g⌋
        ↪  eometry.representative_point()
108     reader = csv.reader(matrixCSV)
109     next(reader)  # Skip header line containing metadata labels
110     for row in reader:
111         try: # If the code is not empty
112             codeReg = int(row[0]) # Region code
113             codeMun = row[3]      # Municipality code
114             nameMun = row[4]      # Municipality name
115
116             dictMun[codeMun] = codeReg
117             # In reality «codeMun» it's more like «(Province
                ↪  code)+(Municipality code)»
118
119             UpdateDictionary(
120                 dictReg,
121                 codeReg,
122                 nameMun,
123                 codeMun,
124                 gdf
125             )
126
127             UpdateDictionary(
128                 dictReg,
129                 21,
130                 nameMun,
131                 codeMun,
132                 gdf
133             )
134
135         except ValueError:
136             continue # Ignore it otherwise
137
138     return dictMun, dictReg
139
140 def UpdateDictionary(
141     dictReg,
142     codeReg,
143     nameMun,
144     codeMun,
145     gdf
146 ):
147     lgi = dictReg[codeReg]['Nc'] # Local/Global index
148     dictReg[codeReg]['li2Name'][lgi] = nameMun
```

```
149         dictReg[codeReg]['li2Coord'][lgi] =
            ↪  [gdf[codeMun].x,gdf[codeMun].y]
150         dictReg[codeReg]['name2li'][nameMun] = lgi
151         dictReg[codeReg]['code2li'][codeMun] = lgi
152
153         dictReg[codeReg]['Nc'] = lgi+1 # Update local/global number of
            ↪  cities
154         # Local (codeReg=!=21) index
155         # Global (codeReg==21) index
156
157     def BuildAdjacencyMatrices(
158         dictMun,
159         dictReg
160     ):
161
162         for r in dictReg:
163             Nc = dictReg[r]['Nc']
164
165             for (M,numTyp) in [
166                 ('A',np.uint8), # 'A' == [Unitary] Adjacency matrix
167                 ('W',np.int64)  # 'W' == Weighted adjacency matrix
168             ]:
169                 dictReg[r][M] = np.zeros((Nc,Nc),dtype=numTyp)
170
171             li2Coord = np.empty((Nc,2),np.float64)
172             for i in range(Nc):
173                 li2Coord[i,0] = dictReg[r]['li2Coord'][i][0]
174                 li2Coord[i,1] = dictReg[r]['li2Coord'][i][1]
175             dictReg[r]['li2Coord'] = li2Coord
176
177         with ZF(libP.matrixZipPath) as z, z.open('Pen_91It.txt') as f:
178             for line in tiow(f,encoding="utf-8"):
179                 oMun = line[:6]         # Origin municipality
180                 dMun = line[11:17]      # Destination municipality
181                 commuters = int(line[17:-1]) # Edge weight (commuters)
182
183                 if oMun != dMun and ' ' not in dMun: # and ' ' not in
                    ↪  oMun
184                     try:
185                         oReg = dictMun[oMun] # Origin region
186                         dReg = dictMun[dMun] # Destination region
187
188                         if oReg == dReg: # and commuters!=0
189                             UpdateMatrices(
190                                 dictReg,
191                                 commuters,
192                                 oReg,dReg,
193                                 oMun,dMun
194                             )
195
196                         UpdateMatrices(
197                             dictReg,
198                             commuters,
199                             21,21,
200                             oMun,dMun
201                         )
202                     except Exception:
203                         continue
204
205                 # else:
206                 #     if oMun != dMun:
207                 #         prova = f'{oMun[:3]}{dMun[:3]}'
208                 #         try:
209                 #             dictMun[prova]
210                 #         except KeyError:
211                 #             print(line)
212
```

```
213         # There are in total two majors typos in the data:
214         #    1. There is a municipality of code «'022008'» which isn't
            ↪  listed in «elencom91.xls», raising always a KeyError
215         #    2. Some lines in «Pen_91It.txt» contain a dMun which is
            ↪  incomplete, lacking in particular the first three numbers
            ↪  for the province code; these are in total:
216         #          '215   ','229   ','241   ','216   ','203   ','224
            ↪  ','236   ','246   '
217         #       Moreover, nothing can be done about it, not even
            ↪  assuming that dReg shares the same province code of oMun,
            ↪  as some of these are ambiguous, even in the same region
218         #       For instance, the very first one '215   ' appears in
            ↪  the line «'00200714212215              1\n'»; leaving aside
            ↪  the fact that there are in total 6 cities with the same
            ↪  municipality code accross the italian peninsula, even just
            ↪  considering the Piedmont region, i.e. the same one of
            ↪  «oMun='002007'» («ASIGLIANO VERCELLESE»), there exists
            ↪  actually two cities with such a code: «'001215'» («RIVA
            ↪  PRESSO CHIERI») and «'004215'» («SAVIGLIAN»O), and both do
            ↪  not have the same province code («'002'») of oMun. In plain
            ↪  words it's impossible which one has to be chosen
219
220         # Therefore the if statement excludes municipalities whose
            ↪  codes are only partially written due to, I presume, typos
            ↪  from ISTAT
221         # while, the try statement catches the few cases where the code
            ↪  does not match any municipality (only one: «022008»)
222
223     def UpdateMatrices(
224         dictReg,commuters,
225         oReg,dReg,
226         oMun,dMun
227     ):
228         oI = dictReg[oReg]['code2li'][oMun] # Local/Global origin index
229         dI = dictReg[dReg]['code2li'][dMun] # Local/Global destination
            ↪  index
230         # The index is considered global iff «oReg=dReg=21»
231
232         dictReg[oReg]['A'][oI,dI] = 1
233         dictReg[dReg]['A'][dI,oI] = 1
234         dictReg[oReg]['W'][oI,dI] += commuters
235         dictReg[dReg]['W'][dI,oI] += commuters
236         # The sum in «matricesReg[oReg]['W'][oI,dI] += commuters» is
            ↪  necessary as there are repeating origin-destination links
            ↪  in the dataset
237
238     def BuildSizeDistributions(
239         dictReg
240     ):
241         sizeDistribution = {}
242
243         with ZF(libP.sizeZipPath) as z:
244             Nc = dictReg[21]['Nc']
245             sizeDistribution[21] = np.zeros((Nc,),dtype=np.int64)
246
247             for r in range(1,21):
248                 Nc = dictReg[r]['Nc']
249                 sizeDistribution[r] = np.zeros((Nc,),dtype=np.int64)
250
251                 # Conversion of the relative «.xls» file into a more
                    ↪  manageable «.csv» one
252                 sizeDistrFile =
                    ↪  xls2csv(z.read(f'R{r:02d}_DatiCPA_1991.xls'))
253                 reader = csv.reader(sizeDistrFile)
254
255                 next(reader) # Skip header line containing metadata
                    ↪  labels
```

```
256             for row in reader:
257                 try: # If the code is not empty and it is a key
258                     # In reality «codeMun» it's more like
                        ↪ «(Province code)+(Municipality code)»
259                     codeMun = f'{int(row[2]):006d}' # Municipality
                        ↪ code
260                     secPop  = int(row[5]) # Section population
261 
262                     li = dictReg[r]['code2li'][codeMun]
263                     sizeDistribution[r][li] += secPop
264 
265                     gi = dictReg[21]['code2li'][codeMun]
266                     sizeDistribution[21][gi] += secPop
267                 except ValueError:
268                     continue # Ignore it otherwise
269 
270             dictReg[r]['sizeDistr'] = sizeDistribution[r]
271 
272         dictReg[21]['sizeDistr'] = sizeDistribution[21]
273 
274 def WriteRegionData(
275     dictReg
276 ):
277     with ZF(
278         libP.regDataZipPath,'w',
279         compression=zf.ZIP_DEFLATED, # Enable compression
280         compresslevel=9              # Max compression for
        ↪ «ZIP_DEFLATED»
281         # https://docs.python.org/3/library/zipfile.html#zipfile-o
        ↪ bjects
282     ) as z:
283         el = {
284             'A':'.txt',
285             'W':'.txt',
286             'li2Name':'.json',
287             'li2Coord':'.txt',
288             'name2li':'.json',
289             'sizeDistr':'.txt',
290             'Nc':'.json'
291         }
292 
293         for r in range(21):
294             folder = f'{'0' if r+1<10 else ''}{r+1}'
295 
296             for data,ext in el.items():
297                 path = f'{folder}/{data}{ext}'
298                 buf = sio()
299 
300                 match ext:
301                     case '.txt':
302                         np.savetxt(
303                             buf,
304                             dictReg[r+1][data],
305                             fmt="%d",
306                             delimiter=","
307                         )
308 
309                     case '.json':
310                         json.dump(
311                             dictReg[r+1][data],
312                             buf,
313                             indent=3 if data != 'Nc' else 0
314                         )
315 
316                 z.writestr(path,buf.getvalue())
317 
318 def WriteSimulationData(
```

```
319         vrtState,
320         snapshots,
321         siVrtState,
322         typ,
323         lbl,
324         li2Name,
325         Ni,
326         sid
327     ):
328         lbl = [t.replace('.','') for t in lbl]
329 
330         mode = 'a' if sid is not None and sid != 1 else 'w'
331         with ZF(
332             libP.simDataZipFile,mode,
333             compression=zf.ZIP_DEFLATED,
334             compresslevel=9
335         ) as z:
336             for r in range(Ni):
337                 dicName2SortedPop = {
338                     t:{
339                         li2Name[i]:vrtState[r,i,t] for i in
                            ↪  siVrtState[r,::-1,t]
340                     } for t in typ
341                 }
342 
343                 folder = (
344                     f'{'' if sid is None else f's{sid}/'}'
345                     f'{0 if r+1<10 else ""}{r+1}'
346                 )
347                 for t in typ:
348                     buf = bio()
349                     path = f'{folder}/{lbl[t]}CitySizesFinal.npy'
350                     np.save(buf,vrtState[r,:,t])
351                     z.writestr(path,buf.getvalue())
352 
353                     buf = sio()
354                     path = f'{folder}/{lbl[t]}CitySizesSorted.json'
355                     json.dump(dicName2SortedPop[t],buf,indent=3)
356                     z.writestr(path,buf.getvalue())
357 
358                     buf = bio()
359                     path = f'{folder}/{lbl[t]}Snapshots.npy'
360                     np.save(buf,snapshots[r,:,:,t])
361                     z.writestr(path,buf.getvalue())
362 
363     def SetParameters(cls):
364         parameters = libP.parameters
365         for name,kwargs in parameters.items():
366             setattr(cls,name,libP.Parameter(**kwargs))
367 
368     def LoadCaseStudies(cls):
369         data = libP.caseStudies
370 
371         caseStudies = data['list']
372         selectedCS = data['selected']
373 
374         listCS = list(caseStudies.keys())
375         dictCS = {}
376 
377         for name,study in caseStudies.items():
378             dictCS[name] = {}
379 
380             for key,val in study.items():
381                 prm = getattr(cls,key)
382                 dictCS[name][prm] = val
383 
384             # After all parameters have been loaded the external lists
                ↪  are saved as well
```

```
385             for (prmName,prmlist) in [
386                 ('region','regionList'),
387                 ('interactionRule','intRuleList'),
388                 ('studiedParameter','studiedPrmList')
389             ]:
390                 prm = getattr(cls,prmName)
391                 value = dictCS[name][prm]
392                 dictCS[name][prm] = getattr(cls,prmlist).list[value]
393 
394         return dictCS, selectedCS, listCS
```

Il quarto modulo è `libFigures.py` nel List. A.5 che racchiude a grandi linee tutte le funzioni necessarie per i grafici; in particolare qui è contenuta la logica degli adattamenti della distribuzione di Pareto e bilognormale.

Listato A.5: Modulo `libFigures.py`.

```
1   # Library to create figures
2 
3   # import os, sys
4   from pathlib import Path
5   import subprocess
6 
7   import numpy as np
8 
9   from scipy.io import savemat
10  from scipy import stats
11  from scipy.optimize import minimize
12 
13  import matplotlib.pyplot as plt
14  from matplotlib.pyplot import savefig
15  # import mplcursors
16 
17  from libParameters import projectFolder
18 
19 
20  ### Main class and functions ###
21 
22  # Figure data
23  class FigData():
24      def __init__(self,clsPrm,folder):
25          self.folder = folder
26          self.projectFolder = projectFolder
27 
28          self.regCode = str(clsPrm.region)
29          self.pdfPopUp = clsPrm.pdfPopUp
30          self.TikZConversion = clsPrm.TikZConversion
31 
32          for ext in ('.pdf','.mat','.tex'):
33              if ext=='.pdf' or self.TikZConversion:
34                  setattr(self,f'{ext[1:]}FolderPath',self.CreateFol
                    ↪  ders(ext))
35                  # for p in folder.iterdir():
36                  #     if p.is_file():
37                  #         p.unlink()
38 
39      def SetFigs(self,nRow=1,nCol=1,size=None):
40          nFig = nRow*nCol; self.nFig = nFig
41 
42          if not isinstance(nFig,int) or nFig <= 0:
43              raise ValueError('The number of figures must be a
                ↪  integer positive number')
44 
45          for i in range(nFig):
```

```
46              setattr(self,f'fig{i+1}',{'plots':{},'style':{}})
47
48          if nFig == 1:
49              self.fig = self.fig1
50              return plt.figure()
51          else:
52              return plt.subplots(
53                  nRow,nCol,
54                  figsize=size,
55                  gridspec_kw=dict(
56                      wspace=0.2,
57                      hspace=0.4
58                  )
59              )
60
61      def SaveFig(self,name):
62          self.SaveFig2pdf(name)
63
64          self.names = [
65              f'{name}fig{f+1}' for f in range(self.nFig)
66          ] if self.nFig>1 else [name]
67
68          if self.TikZConversion:
69              for f in range(self.nFig):
70                  self.SaveFig2mat(f)
71                  self.SaveFig2tex(f)
72
73      def SaveFig2pdf(self,name):
74          format = '.pdf'
75          pdfFilePath = self.FigPath(name,format)
76
77          savefig(
78              pdfFilePath,
79              dpi=300,
80              bbox_inches='tight'
81          )
82
83          if self.pdfPopUp:
84              sumatraPath =
                ↪  self.projectFolder/'.vscode'/'SumatraPDF.lnk'
85              cmd = f'"{str(sumatraPath)}" "{str(pdfFilePath)}"'
86              subprocess.Popen(cmd,shell=True)
87
88      def SaveFig2mat(self,f):
89          format = '.mat'
90          self.matFilePath = self.FigPath(self.names[f],format)
91
92          savemat(self.matFilePath,getattr(self,f'fig{f+1}'))
93
94      def SaveFig2tex(self,f):
95          format = '.tex'
96          self.texFilePath = self.FigPath(self.names[f],format)
97
98          cmd = self.cmdTeX()
99          subprocess.run(cmd,shell=True)
100
101     def CreateFolders(self,format):
102         folderPath = self.projectFolder/'Figure'/format/self.regCo⌋
            ↪  de/self.folder
103         Path(folderPath).mkdir(parents=True,exist_ok=True)
104         # Define the folder path and create all the missing ones if
            ↪  necessary
105         return folderPath
106
107     def FigPath(self,figName,ext):
108         return (getattr(self,f'{ext[1:]}FolderPath')/figName).with⌋
            ↪  _suffix(ext)
```

```
109
110         def cmdTeX(self):
111             m = self.matFilePath.as_posix()
112             t = self.texFilePath.as_posix()
113
114             cmd = f"addpath('../Elaborato/Convertitore');
                ↪  mat2tex('{m}','{t}')"
115             return f'matlab -batch "{cmd}"'
116
117     # Primary/Basic plots
118     def CreateFunctionPlot(
119         x,y,
120         figData,
121         Ni=1,
122         ta=None,
123         yErr=True,
124         label='',
125         linestyle='-',
126         linewidth=1,
127         marker='None',
128         hatch='None',
129         color='black',
130         alpha=1,
131         idx='',
132         ax=None
133     ):
134         if ax is None: ax = plt.gca()
135
136         if Ni == 1:
137             ax.plot(
138                 x,y if y.ndim == 1 else y.ravel(),
139                 label=label,
140                 color=color,
141                 linestyle=linestyle,
142                 linewidth=linewidth,
143                 marker=marker,
144                 markevery=x.size//10-1,
145                 mfc=color,
146                 alpha=alpha
147             )
148             figData['plots'][f'functionPlot{idx}'] = {
149                 't':'function','x':x,'y':y,
150                 'l':label,'c':color,'m':marker
151             }
152
153         else:
154             if yErr:
155                 avrFunc, err95Func = EvaluateConfidenceInterval(y,ta,Ni)
156
157                 # To avoid [a possible] lower negative confidence
                    ↪  interval, the error has to be clipped
158                 lowerEstimate = np.maximum(avrFunc-err95Func,0)
159                 yerr95 = np.array([
160                     avrFunc-lowerEstimate,
161                     err95Func
162                 ]); xerr95 = None; y = avrFunc
163
164                 if (y-yerr95[0]).min()<0: raise("Negative values")
165                 if (y+yerr95[1]).min()<0: raise("Negative values")
166
167             else:
168                 avrFunc, err95Func = EvaluateConfidenceInterval(x,ta,Ni)
169
170                 # To avoid [a possible] lower negative confidence
                    ↪  interval, the error has to be clipped
171                 lowerEstimate = np.maximum(avrFunc-err95Func,0)
172                 xerr95 = np.array([
```

```
                    avrFunc-lowerEstimate,
                    err95Func
                ]); yerr95 = None; x = avrFunc

            ax.plot(
                x,y,
                label=label,
                color=color,
                linestyle=linestyle,
                linewidth=linewidth,
                marker=marker,
                markevery=x.size//10-1,
                mfc=color,
                alpha=alpha[0],
                zorder=4
            )
            figData['plots'][f'functionPlot{idx}'] = {
                't':'function','x':x,'y':y,
                'l':label,'c':color,'m':marker
            }

            CreateConfidenceIntervalPlot(
                x,y,
                figData,
                typ='functionfill',
                xerr95=xerr95,
                yerr95=yerr95,
                color=color,
                hatch=hatch,
                alpha=alpha[1],
                ax=ax,
                idx=idx
            )

        # mplcursors.cursor(fPlot,hover=False).connect(
        #     "add",lambda sel: sel.annotation.set_text(
        #         f"({sel.target[0]:.3f},{sel.target[1]:.3f})"
        #     )
        # )

    def CreateScatterPlot(
        x,y,
        figData,
        size=16,
        Ni=1,
        ta=None,
        yErr=True,
        label='Scatter',
        color='black',
        idx='',
        ax=None,
        alpha=1
    ):
        if ax is None: ax=plt.gca()

        if Ni == 1:
            sct = ax.scatter(
                x,y if y.ndim == 1 else y.ravel(),
                label=label,
                color=color,
                s=size,
                edgecolor='none',
                alpha=alpha
            )
            figData['plots'][f'scatterPlot{idx}'] = {
                't':'scatter','x':x,'y':y,'l':label,
                'c':color,'a':alpha,'s':size
```

```
        }
    else:
        if yErr:
            avrSct, err95Sct = EvaluateConfidenceInterval(y,ta,Ni)

            sct = ax.scatter(
                x,
                avrSct,
                label=label,
                color=color,
                s=size,
                edgecolor='none',
                alpha=alpha[0],
                zorder=5
            )
            figData['plots'][f'scatterPlot{idx}'] = {
                't':'scatter','x':x,'y':avrSct,'l':label,
                'c':color,'a':alpha[0],'s':size
            }

            CreateConfidenceIntervalPlot(
                x,
                avrSct,
                figData,
                typ='errorbar',
                yerr95=np.array([err95Sct,err95Sct]),
                color=color,
                alpha=alpha[1],
                ax=ax,
                idx=idx
            )

        else:
            avrSct, err95Sct = EvaluateConfidenceInterval(x,ta,Ni)

            sct = ax.scatter(
                avrSct,
                y,
                label=label,
                color=color,
                s=size,
                edgecolor='none',
                alpha=alpha[0],
                zorder=5
            )
            figData['plots'][f'scatterPlot{idx}'] = {
                't':'scatter','x':avrSct,'y':y,'l':label,
                'c':color,'a':alpha[0],'s':size
            }

            CreateConfidenceIntervalPlot(
                avrSct,
                y,
                figData,
                typ='errorbar',
                xerr95=np.array([err95Sct,err95Sct]),
                color=color,
                alpha=alpha[1],
                ax=ax,
                idx=idx
            )

    # mplcursors.cursor(sct,hover=True).connect(
    #     "add",lambda sel: sel.annotation.set_text(
    #         f"({x[sel.index]},{y[sel.index]})"
    #     )
    # )
```

```
307
308 def CreateHistogramPlot(
309     x,nBins,
310     figData,
311     limits=None,
312     xScale='lin',
313     Ni=1,
314     ta=None,
315     label='Histogram',
316     color='#808080',
317     norm=True,
318     alpha=1,
319     idx='',
320     ax=None
321 ):
322     def HistogramPlot(
323         binEdges,
324         binMidPoints,
325         binAverages,
326         alpha=alpha
327     ):
328         hgPlot = ax.hist(
329             binMidPoints,
330             bins=binEdges, # 'auto'
331             weights=binAverages,
332             # density=norm,
333             color=color,
334             edgecolor="none", # "black"
335             label=label,
336             alpha=alpha
337         )
338
339         figData['plots'][f'histogramPlot{idx}'] = {
340             't':'histogram','l':label,'c':color,'a':alpha,
341             'x':binMidPoints,'b':binEdges,'w':binAverages
342         }
343
344         # hgPlot[0] = heights,
345         # hgPlot[1] = bin edges,
346         # hgPlot[2] = patches (Rectangle objects)
347
348         # mplcursors.cursor(hgPlot[2],hover=False).connect(
349         #     "add",lambda sel: sel.annotation.set_text(
350         #         f"{hgPlot[0][sel.index]:.3f}"
351         #     )
352         # )
353
354     if ax is None: ax = plt.gca()
355
356     if limits is None:
357         xMin = np.min(x); xMax=np.max(x)
358     else:
359         xMin = limits[0]; xMax=limits[1]
360
361     if xScale == 'lin':
362         binPoints = np.linspace(
363             xMin,
364             xMax,
365             num=2*nBins+1
366         )
367     else:
368         binPoints = np.logspace(
369             np.log10(xMin),
370             np.log10(xMax),
371             num=2*nBins+1
372         )
373
```

```
374         binEdges = binPoints[0::2]
375         binMidPoints = binPoints[1::2]
376         # binMidPoints = (binEdges[:-1]+binEdges[1:])/2 # It only works
            ↪  in the linear case
377 
378         if Ni == 1:
379             binAverages, _ = np.histogram(
380                 x if x.ndim == 1 else x.ravel(),
381                 binEdges,density=norm
382             )
383             HistogramPlot(binEdges,binMidPoints,binAverages,alpha)
384 
385             return binMidPoints, binAverages
386         else:
387             hgData = [None]*Ni
388             for r in range(Ni):
389                 hgData[r] = np.histogram(x[r,:],binEdges,density=norm)
390 
391             binAverages, hgErr95 = EvaluateConfidenceInterval(
392                 [hgData[r][0] for r in range(Ni)],ta,Ni
393             )
394 
395             HistogramPlot(binEdges,binMidPoints,binAverages,alpha[0])
396 
397             # To avoid [a possible] lower negative confidence interval
                ↪  in the lower tail, the error has to be clipped
398             lowerBinEstimate = np.maximum(binAverages-hgErr95,0)
399             binErr95 = np.array([
400                 binAverages-lowerBinEstimate,
401                 hgErr95
402             ])
403 
404             CreateConfidenceIntervalPlot(
405                 binMidPoints,
406                 binAverages,
407                 figData,
408                 typ='errorbar',
409                 yerr95=binErr95,
410                 color=color,
411                 ax=ax,
412                 idx=idx,
413                 alpha=alpha[1]
414             )
415 
416             return binMidPoints, binAverages
417 
418     # Secondary/Compound plots
419     def CreateLogRegressionPlot(
420         x,y,
421         figData,
422         l='Regression',
423         color='blue',
424         idx='',
425         ax=None
426     ):
427         if ax is None: ax=plt.gca()
428 
429         xp = x[y>0]; yp = y[y>0]
430         logx  = np.log10(xp); logy = np.log10(yp)
431 
432         slope, intercept, _, _, _ = stats.linregress(logx,logy)
433         regression = 10**(intercept+slope*logx)
434 
435         CreateFunctionPlot(
436             xp,regression,
437             figData,
438             label=l,
```

```
            color=color,
        )

        # fPlot = ax.plot(
        # figData['plots'][f'regressionPlot{idx}'] = {
        #     't':'function','x':x,'y':regression,'l':l,'c':color
        # }

        # mplcursors.cursor(fPlot,hover=False).connect(
        #     "add",lambda sel: sel.annotation.set_text(
        #         f"({sel.target[0]:.3f},{sel.target[1]:.3f})"
        #     )
        # )

        return slope

    def CreateLognormalFitPlot(
        v,
        figData,
        limits=None,
        xScale='lin',
        Ni=1,
        ta=None,
        label='Lognormal fit (ML)', # Maximum likelihood
        color='black',
        alpha=1,
        marker='None',
        bimodal=False,
        idx='',
        ax=None
    ):
        if ax is None: ax=plt.gca()

        if limits is None:
            vMin = np.min(v); vMax=np.max(v)
        else:
            vMin = limits[0]; vMax=limits[1]

        if xScale == 'lin':
            xF = np.linspace(
                vMin,
                vMax,
                1000
            )
        else:
            xF = np.logspace(
                np.log10(vMin),
                np.log10(vMax),
                1000
            )

        yFData = [None]*Ni; xiData = [None]*Ni
        m1Data = [None]*Ni; s1Data = [None]*Ni
        m2Data = [None]*Ni; s2Data = [None]*Ni

        for r in range(Ni):
            if bimodal:
                xi,m1,s1,m2,s2 = FitBiLognormalData(v[r,:],scale=xScale)
                if xScale == 'log':
                    yFData[r] = (
                        xi*stats.lognorm.pdf(xF,s1*np.log(10),scale=10**m1) +
                        (1-xi)*stats.lognorm.pdf(xF,s2*np.log(10),scale=10**m2)
                    )#; print(xi,m1,s1,m2,s2)
```

```
                # Since «stats.lognorm» is defined with the natural
                ↪ lognormal but all the parameters are estimated
                ↪ in base 10, they have to be converted
                # To confirm that be mindful that, while doing the
                ↪ conversion using
                ↪ https://docs.scipy.org/doc/scipy/reference/gen
                ↪ erated/scipy.stats.lognorm.html, the derivative
                ↪ of «ln(x)» is «1/x» while of «log(x)» is
                ↪ «1/(xln(10))»
            elif xScale == 'lin':
                yFData[r] = (
                    xi*stats.norm.pdf(xF,loc=m1,scale=s1) +
                    (1-xi)*stats.norm.pdf(xF,loc=m2,scale=s2)
                )#; print(xi,m1,s1,m2,s2)

            xiData[r] = xi
            m1Data[r] = m1#*np.log10(np.exp(1))
            s1Data[r] = s1#*np.log10(np.exp(1))
            m2Data[r] = m2#*np.log10(np.exp(1))
            s2Data[r] = s2#*np.log10(np.exp(1))
            # Base 10 conversion if «np.log()» had been used

        else:
            shape,loc,scale = stats.lognorm.fit(v[r,:],floc=0)
            yFData[r] = stats.lognorm.pdf(xF,shape,scale=scale)
            # The average is «μ=np.log(scale)» (base e) while the
            ↪ standard deviation is «σ=shape»

            xiData[r] = 1
            m1Data[r] = np.log10(scale) # Base 10 conversion
            s1Data[r] = shape*np.log10(np.exp(1))
            m2Data[r] = 0; s2Data[r] = 0

    if Ni>1:
        yF = yFData; xi = xiData
        m1 = m1Data; s1 = s1Data
        m2 = m2Data; s2 = s2Data
    else:
        yF = yFData[0]; xi = xiData[0]
        m1 = m1Data[0]; s1 = s1Data[0]
        m2 = m2Data[0]; s2 = s2Data[0]

    # Fit plot
    CreateFunctionPlot(
        xF,yF,
        figData,
        Ni=Ni,
        ta=ta,
        label=label,
        # linewidth=1,
        color=color,
        alpha=alpha,
        marker=marker,
        idx=idx,
        ax=ax
    )

    return xi,m1,s1,m2,s2

def CreateParetoFitPlot(
    v,
    figData,
    upperbound=None,
    yScale='lin',
    Ni=1,
    ta=None,
    label=(
```

```
561             'Scatter',
562             'Pareto fit (ML)' # Minimum likelihood
563         ),
564         color=(
565             'black',
566             'black'
567         ),
568         alpha=1,
569         bimodal=False,
570         idx='',
571         ax=None
572     ):
573         if ax is None: ax=plt.gca()
574 
575         if upperbound is None:
576             xF = np.linspace(
577                 np.log10(np.quantile(v,0.75)),
578                 np.log10(v.max()),
579                 100
580             )
581         else:
582             xF = np.linspace(
583                 np.log10(np.quantile(v,0.75)),
584                 np.log10(upperbound),
585                 100
586             )
587 
588         vTails  = [None]*Ni
589         b       = [None]*Ni
590         ccdfFit = [None]*Ni
591 
592         for r in range(Ni):
593             # Select the last quarter of city sizes
594             vQuarter = np.quantile(v[r,:],.75)
595             vTail = v[r,v[r,:] >= vQuarter]
596             vTails[r] = np.log10(np.sort(vTail)) # Ascending values
597 
598             # Fitted CCDF from a Pareto function
599             b[r], loc, scale =
                ↪  stats.pareto.fit(vTail,floc=0,fscale=vQuarter)
600             ccdfFit[r] =
                ↪  stats.pareto.sf(10**xF,b[r],loc=loc,scale=scale) #
                ↪  Survival function
601 
602         # Empirical CCDF on the tail
603         n = vTails[0].size
604         ccdfEmp = (
605             n-np.arange(1,n+1,dtype=float)
606             +(0.5 if yScale == 'log' else 0)
607         )/n # P(X⩾x)
608         yS = ccdfEmp
609 
610         """
611             The survival function is the Complementary Cumulative
                ↪  Distribution Function (CCDF)
612 
613             Formally it should be
614 
615                     1-np.arange(1,n+1,dtype=float)/Nc-(Nc-n)/Nc=
616                     (n-np.arange(1,n+1,dtype=float))/Nc=
617             (*)     [(n-np.arange(1,n+1,dtype=float))/n][n/Nc]
618 
619             hence the correct variables are:
620 
621                 xS = vTails
622                 yS = (n-np.arange(1,n+1,dtype=float))/n*(n/Nc)
623                 yF = [ccdfFit[r]*(n/Nc) for r in range(Ni)]
```

```
624
625             without the constant translation 0.5, which is just
                ↪ necessary to avoid having a zero value at the tail end
                ↪ in a log-log plot.
626
627             However, since the only relevant information is the Pareto
                ↪ index, the exact probability value can be omitted:
                ↪ indeed, by dividing (*) by [n/Nc] the CCDF is rescaled
                ↪ in a linear scale while translated upward in a
                ↪ logarithmic one; both transformations do not affect the
                ↪ actual trend of the CCDF from which the Pareto index is
                ↪ calculated
628         """
629
630         if Ni>1:
631             xS = vTails
632             yF = ccdfFit
633         else:
634             xS = vTails[0]
635             yF = ccdfFit[0]
636             b = b[0]
637
638         CreateScatterPlot(
639             xS,yS,
640             figData,
641             Ni=Ni,
642             ta=ta,
643             size=10,
644             yErr=False,
645             label=label[0],
646             color=color[0],
647             alpha=alpha[0],
648             idx=idx,
649             ax=ax
650         )
651
652         CreateFunctionPlot(
653             xF,yF,
654             figData,
655             Ni=Ni,
656             ta=ta,
657             label=label[1],
658             color=color[1],
659             alpha=alpha[1],
660             marker='+' if bimodal else 'None',
661             hatch='|' if bimodal else 'None',
662             idx=idx,
663             ax=ax
664         )
665
666         if bimodal:
667             # xFl = np.log10(xF)
668             yFData = [None]*Ni
669             Nc = v.shape[1]
670
671             for r in range(Ni):
672                 xi,m1,s1,m2,s2 = FitBiLognormalData(v[r,:] if Ni>1 else
                    ↪ v)
673
674                 # ccdfFit1 = stats.norm.sf(xFl,loc=m1,scale=s1)
675                 # ccdfFit2 = stats.norm.sf(xFl,loc=m2,scale=s2)
676                 ccdfFit1 = stats.norm.sf(xF,loc=m1,scale=s1)
677                 ccdfFit2 = stats.norm.sf(xF,loc=m2,scale=s2)
678
679                 yFData[r] = xi*ccdfFit1+(1-xi)*ccdfFit2
680                 # yFData[r] /= yFData[r][0] # Normalization
681                 yFData[r] /= n/Nc # Rescaling
```

```
682             # It's necessary to effectively compare it with the
                ↪  Pareto fitting
683
684         yF = yFData if Ni>1 else yFData[0]
685
686         CreateFunctionPlot(
687             xF,yF,
688             figData,
689             Ni=Ni,
690             ta=ta,
691             label=label[2],
692             color=color[1],
693             alpha=alpha[1],
694             marker='x',
695             hatch='/',
696             idx=idx+1,
697             ax=ax
698         )
699
700     return b
701
702
703 ### Auxiliary functions ###
704
705 def SetTextStyle():
706     plt.rcParams["mathtext.fontset"] = "stix"
707     plt.rcParams["font.family"] = "serif"
708     plt.rcParams["font.serif"] = ["Times New Roman"]
709     plt.rcParams["font.size"] = 13
710 SetTextStyle()
711
712 def SetFigStyle(
713     xLabel=None,
714     yLabel=None,
715     xDom=None,
716     yDom=None,
717     xScale='lin',
718     yScale='lin',
719     xNotation='plain',
720     yNotation='plain',
721     ax=None,data=None
722 ):
723     if ax is None: ax = plt.gca()
724
725     if xLabel: ax.set_xlabel(xLabel)
726     if yLabel: ax.set_ylabel(yLabel)
727
728     if xDom: ax.set_xlim(xDom)
729     if yDom: ax.set_ylim(yDom)
730
731     ax.set_xscale('linear' if xScale == 'lin' else xScale)
732     ax.set_yscale('linear' if yScale == 'lin' else yScale)
733
734     if xScale == 'lin':
735         ax.ticklabel_format(style=xNotation,axis='x',scilimits=(0,
            ↪  0))
736     if yScale == 'lin':
737         ax.ticklabel_format(style=yNotation,axis='y',scilimits=(0,
            ↪  0))
738
739     ax.grid(True,linestyle=":",linewidth=1)
740     ax.set_axisbelow(True)
741
742     _, labels = ax.get_legend_handles_labels()
743     if any(labels):
744         ax.legend()
745         data['style']['legend'] = True
```

```
    else:
        data['style']['legend'] = False
    # Create a legend iff there are labels connected to graphs

    data['style']['scale'] = {'x':xScale,'y':yScale}
    data['style']['labels'] = {'x':xLabel,'y':yLabel}

def CentreFig():
    fig = plt.gcf()
    manager = plt.get_current_fig_manager()
    manager.window.update_idletasks()

    # Get screen size
    screenW = manager.window.winfo_screenwidth()
    screenH = manager.window.winfo_screenheight()

    # Save figure size in pixels
    figW, figH = fig.get_size_inches()*fig.dpi

    # Centre coordinates
    x = int((screenW-figW)/2)
    y = int((screenH-figH)/2)

    # Move the figure window
    manager.window.geometry(f"+{x}+{y}")

def EvaluateConfidenceInterval(
    data,
    ta,
    Ni
):
    if Ni>1:
        values = np.array(data)
        averages = np.mean(values,axis=0)
        err95 = ta*np.std(values,axis=0,ddof=1)/np.sqrt(Ni)
        return (averages,err95)
    else:
        if isinstance(data,np.ndarray):
            return (data[0],None)
        else:
            return (data,None)

def CreateConfidenceIntervalPlot(
    x,y,
    figData,
    typ='errorbar',
    yerr95=None,
    xerr95=None,
    color='black',
    fmt='none',
    hatch='None',
    alpha=0.5,
    ax=None,
    idx=''
):
    if ax is None: ax = plt.gca()

    match typ:
        case 'errorbar':
            ax.errorbar(
                x,y,
                xerr=xerr95,
                yerr=yerr95,
                ecolor=color,
                fmt=fmt,
                linestyle='none',
                elinewidth=1.2,
```

```
                capsize=8,
                capthick=1.6,
                markersize=0,
                markerfacecolor=color,
                markerfacecoloralt=color,
                markeredgecolor=color,
                markeredgewidth=0,
                alpha=alpha,
                zorder=1
            )
            if xerr95 is None:
                figData['plots'][f'{typ}{idx}'] = {
                    't':typ,'x':x,'y':y,
                    'e':'y','ye':yerr95,
                    'l':'','c':color,'a':alpha,
                }
            else:
                figData['plots'][f'{typ}{idx}'] = {
                    't':typ,'x':x,'y':y,
                    'e':'x','xe':xerr95,
                    'l':'','c':color,'a':alpha,
                }

        case 'functionfill':
            if xerr95 is None:
                ax.fill_between(
                    x,y-yerr95[0],y+yerr95[1],
                    facecolor=color,
                    alpha=alpha,
                    linewidth=0,
                    hatch= None if hatch=='None' else hatch,
                    edgecolor= None if hatch=='None' else color,
                    zorder=2
                )
                figData['plots'][f'{typ}{idx}'] = {
                    't':typ,'x':x,'y':y,
                    'e':'y','ye':yerr95,
                    'l':'','c':color,
                    'a':alpha,'h':hatch,
                }
            else:
                ax.fill_betweenx(
                    y,x-xerr95[0],x+xerr95[1],
                    facecolor=color,
                    alpha=alpha,
                    linewidth=0,
                    hatch= None if hatch=='None' else hatch,
                    edgecolor= None if hatch=='None' else color,
                    zorder=2
                )
                figData['plots'][f'{typ}{idx}'] = {
                    't':typ,'x':x,'y':y,
                    'e':'x','xe':xerr95,
                    'l':'','c':color,
                    'a':alpha,'h':hatch,
                }

def FitBiLognormalData(v,scale='log'):
    # https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.lognorm.html
    # https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.norm.html
    # https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.rv_continuous.fit.html#scipy.stats.rv_continuous.fit

    if scale == 'log':
        vp = v[v>0]
```

```
        vl = np.log10(vp)
    else:
        vl = v
    # The option «scale» refers to the x scale: if the data is meant to be plotted on a logarithmic scale, then it hasn't been transformed; on the other hand, if the scale is linear it means that the logarithm has already been applied to it

    # The objective function to minimize is the negative log-likelihood
    # This is equivalent to maximize the log-likelihood, just reversed
    def NegativeLogLikelihood(params):
        # xi is fraction between Gaussian
        # mi and si are the mean and standard deviation of the ith Gaussian
        xi,m1,s1,m2,s2 = params

        # Calculate Gaussian PDF for both components on log data
        pdf1 = stats.norm.pdf(vl,loc=m1,scale=s1)
        pdf2 = stats.norm.pdf(vl,loc=m2,scale=s2)

        mixture = xi*pdf1+(1-xi)*pdf2 # Mixture sum

        epsilon = 1e-10 # Prevent log(0) errors
        return -np.sum(np.log10(mixture+epsilon))

    # Initial heuristic guesses
    median = np.median(vl)
    leftvl = vl[vl<=median]
    rightvl = vl[vl>median]

    prm0 = [
        0.5, # Equal initial weigth
        np.mean(leftvl),np.std(leftvl),
        np.mean(rightvl),np.std(rightvl)
    ] # As the initial guesses the data is split in two cluters: the first before the meadian and the second after it; in this way the starting values are based on the actual data and are not fixed by design

    s0 = np.std(vl)
    mins = 0.1*s0
    bounds = ( # xi must not reach its bounds
        (.01,.99),   # xi
        (None,None), # m1
        (mins,None), # s1
        (None,None), # m2
        (mins, None) # s2
    ) # m has no bounds while s>mins

    # Optimization
    minimization = minimize(
        NegativeLogLikelihood,
        prm0,
        bounds=bounds,
        method='L-BFGS-B',
        options={'ftol':1e-9}
    )

    # Extract results
    xi,m1,s1,m2,s2 = minimization.x

    # Check for degeneration
    if s2<=mins+1e-5: return 1,m1,s1,m2,s2 # First dominant component
```

```
        if s1<=mins+1e-5: return 0,m1,s1,m2,s2 # Second dominant component
        # Indeed, if the fit is too unbalanced it means that there is one component not significant enough to jusfity a whole Gaussian (it's like trying to fit it with one point)

        return xi,m1,s1,m2,s2

    def DataString(
        data,
        Ni=1,
        ta=None,
        head='',
        formatVal='.3f',
        formatErr='.3f',
        space=True
    ):
        (value,error) = EvaluateConfidenceInterval(data,ta,Ni)

        space = r'\qquad' if space else ''
        if error is None:
            return fr'${head}={value:{formatVal}}{space}$'
        else:
            return fr'${head}={value:{formatVal}}\pm{error:{formatErr}}{space}$'

    def Text(
        target,
        pos,
        string,
        ha='center',
        color=None
    ):
        if hasattr(target,"transAxes"):
            target.text(
                pos[0],
                pos[1],
                string,
                color=color,
                ha=ha,
                transform=target.transAxes
            )
        else:
            target.text(
                pos[0],
                pos[1],
                string,
                color=color,
                ha=ha
            )

    def TextBlock(
        ax,
        list,
        p=(0,0),
        dp=(0,0),
        offset=0,
        **kwargs
    ):
        nR = len(list); nC = len(list[0])-offset
        for r,y in enumerate(np.linspace(p[1]+dp[1]/2,p[1]-dp[1]/2,nR)):
            for c,x in enumerate(
                np.linspace(p[0]-dp[0]/2,p[0]+dp[0]/2,nC),
                start=offset
            ):
                Text(ax,(x,y),list[r][c],**kwargs)
                # x moves from left to right
```

```
            # y moves from top to bottom

    # An alternative is to do what linspace does manually with «x0+(i-(n-1)/2)*dx» where dx is actually the space between strings rather than the block length
```

Il quinto modulo è `libNetworks.py` nel List. A.6 che serve per riprodurre i principali risultati di [6].

Listato A.6: Modulo `libNetworks.py`.

```
# Library to analyse networks [also known as graphs]

import numpy as np
import networkx as nx

import libFigures as libF


### Main class ###

class NetworkAnalysis():
    def __init__(self,clsPrm,clsReg):
        self.li2Name = clsReg.li2Name
        self.name2li = clsReg.name2li
        self.Nc = clsReg.Nc

        A = clsReg.A; self.A = A
        W = clsReg.W; self.W = W

        # Degrees
        di = np.sum(A,axis=0); self.di = di # Degree vector
        self.dk, self.Nk = np.unique(di,return_counts=True)
        # Unique degrees and corresponding frequencies
        # Pk = counts/N

        # [Nonzero] Weights
        wi = W[W>0]; self.wi = wi
        self.wk, self.wNk = np.unique(wi,return_counts=True)
        # Unique weights and corresponding frequencies

        # Strenghts
        si = np.sum(W,axis=0); self.si = si
        self.sk, self.sNk = np.unique(si,return_counts=True)
        # Unique strenghts and corresponding frequencies

        self.figData = libF.FigData(clsPrm,'NA')

    def DegreeDistributionFig(self):
        li2Name = self.li2Name
        di = self.di

        figData = self.figData
        fig = figData.SetFigs()


        kAvr = np.mean(di)
        libF.TextBlock(
            fig,[
                [libF.DataString(di.size,head='N',space=False,form
                ↪ atVal='')],
                [libF.DataString(int(np.sum(di)/2),head='E',space=
                ↪ False,formatVal='')],
                [libF.DataString(np.min(di),head=r'k_{{min}}',spac
                ↪ e=False,formatVal='')],
```

```
52                  [libF.DataString(np.max(di),head=r'k_{{max}}',spac
                    ↪  e=False,formatVal='')],
53                  [libF.DataString(kAvr,head=r'\langle
                    ↪  k\rangle',space=False)]
54              ],(0.8,0.4),
55              (0,0.15)
56          )
57
58
59          kis = np.argsort(di) # Vector for the sorted degrees
60          kit = [['City','k']] # Table for the sorted degrees
61          for i in range(10):
62              li = kis[-i-1]
63              kit.append([li2Name[li],di[li]])
64          libF.TextBlock(fig,kit,(1.175,0.5),(0.2,0.4))
65
66
67          # Histogram plot
68          def EstimateBinNumber(v):
69              # The option «'fd'» stands for «Freedman-Diaconis» and
70              # uses Numpy to calculate the optimal edges for the data
71              # edges = np.histogram_bin_edges(v,bins='fd')
72              edges = np.histogram_bin_edges(v,bins='doane')
73              return len(edges) - 1
74          libF.CreateHistogramPlot(di,EstimateBinNumber(di),figData.
            ↪  fig)
75
76          #region
77              # Scatter plot
78              # scP = plt.scatter(k,Pk,label="Empirical",s=10)
79
80              # mplcursors.cursor(scP,hover=False).connect(
81              #     "add",lambda sel: sel.annotation.set_text(
82              #         f"k={k[sel.index]}, P(k)={Pk[sel.index]:.3f}"
83              #     )
84              # )
85
86
87              # # Manual fitting (ML)
88
89              # # Estimators
90              # m = (1/N)*np.sum(np.log(d)) # Estimated mean
91              # s2 = (1/N)*np.sum((np.log(d)-m)**2)
92              # s = np.sqrt(s2) # Estimated standard deviation
93
94              # # Evaluation
95              # def lognormalPDF(x,m,s):
96              #     y = np.zeros_like(x)
97              #     v = x>0
98              #     C = 1/(x[v]*s*np.sqrt(2*np.pi))
99              #     y[v] = C*np.exp(-(np.log(x[v])-m)**2/(2*s**2))
100             #     return y
101
102             # # Plotting
103             # scF = plt.plot(
104             #     x,lognormalPDF(x,m,s),
105             #     label="Manual lognormal fit (ML)", # Maximum
                    ↪  likelyhood
106             #     color="red",
107             #     linewidth=1
108             # )
109         #endregion
110
111
112         # SciPy fitting (ML)
113         libF.CreateLognormalFitPlot(di.reshape(1,-1),figData.fig)
114
```

```
115
116             # Style
117             # libF.CentreFig()
118             libF.SetFigStyle(
119                 r"$k$",r"$P(k)$",
120                 # [0,300],[0,0.03],
121                 data=figData.fig
122             )
123             figData.SaveFig('DegreeDistribution')
124
125         def WeightDistributionFig(self):
126             W = self.W
127             wi = self.wi
128             Nc = self.Nc
129
130             li2Name = self.li2Name
131             name2li = self.name2li
132
133             figData = self.figData
134             fig = figData.SetFigs()
135
136
137             # Histogram plot
138             binw, binPw = libF.CreateHistogramPlot(wi,19,figData.fig,x
                ↪  Scale='log')
139
140
141             # SciPy regression
142             # binw = (hgPlot[1][1:]+hgPlot[1][:-1])/2
143             # binPw = hgPlot[0]
144             # sc = plt.scatter(binw,binPw,c='r',s=50)
145
146             # Fit in log-log space
147             slope = libF.CreateLogRegressionPlot(binw,binPw,figData.fig)
148
149             kAvr = np.mean(wi)
150             libF.TextBlock(
151                 fig,[
152                     [libF.DataString(np.min(wi),head=r'w_{min}',space=
                        ↪  False,formatVal='')],
153                     [libF.DataString(np.max(wi),head=r'w_{max}',space=
                        ↪  False,formatVal='')],
154                     [libF.DataString(kAvr,head=r'\langle
                        ↪  w\rangle',space=False,formatVal='.3f')],
155                     [libF.DataString(slope,head=r'\alpha',space=False,
                        ↪  formatVal='.3f')]
156                 ],(0.75,0.5),
157                 (0,0.1)
158             )
159
160
161             iWu,jWu = np.triu_indices_from(W,k=1)
162             values = W[iWu,jWu]
163             ls = np.argsort(values)[-5:][::-1] # Vector or sorted links
164
165             Wt = [['Connection [extracted]', 'w']]
166             for l in ls:
167                 lir = iWu[l]; lic = jWu[l]
168                 Wt.append([f'{li2Name[lir]}-{li2Name[lic]}',W[lir,lic]])
169             libF.TextBlock(fig,Wt,(1.3,0.65),(0.3,0.2))
170
171
172             Wt = [['Connection [reference]','w']]
173             for link in [
174                 ['CAGLIARI','SASSARI'],
175                 ['SASSARI','OLBIA'],
176                 ['CAGLIARI','ASSEMINI'],
```

```
177                 ['PORTO TORRES','SASSARI'],
178                 ['CAGLIARI','CAPOTERRA']
179             ]:
180                 lir = name2li[link[0]]
181                 lic = name2li[link[1]]
182                 Wt.append([f'{li2Name[lir]}-{li2Name[lic]}',W[lir,lic]])
183             libF.TextBlock(fig,Wt,(1.3,0.35),(0.3,0.2))
184
185
186             # Style
187             # libF.CentreFig()
188             libF.SetFigStyle(
189                 r"$w$",r"$P(w)$",
190                 xScale="log",yScale="log",
191                 data=figData.fig
192             )
193             figData.SaveFig('WeightDistribution')
194
195         def StrengthDistributionFig(self):
196             li2Name = self.li2Name
197             si = self.si
198
199             figData = self.figData
200             fig = figData.SetFigs()
201
202
203             # Histogram plot
204             bins, binPs = libF.CreateHistogramPlot(si,20,figData.fig,x
                ↪ Scale='log')
205
206
207             # SciPy regression
208             # bins = (hgPlot[1][1:]+hgPlot[1][:-1])/2
209             # binPs = hgPlot[0]
210             # sc = plt.scatter(binw,binPw,c='r',s=50)
211
212             # Fit in log-log space
213             v = binPs>0; v[:6] = 0
214             slope = libF.CreateLogRegressionPlot(bins[v],binPs[v],figD
                ↪ ata.fig)
215
216
217             kAvr = np.mean(si)
218             libF.TextBlock(
219                 fig,[
220                     [libF.DataString(np.min(si),head=r's_{min}',space=
                        ↪ False,formatVal='')],
221                     [libF.DataString(np.max(si),head=r's_{max}',space=
                        ↪ False,formatVal='')],
222                     [libF.DataString(kAvr,head=r'\langle
                        ↪ s\rangle',space=False,formatVal='.3f')],
223                     [libF.DataString(slope,head=r'\alpha',space=False,
                        ↪ formatVal='.3f')]
224                 ],(0.75,0.5),
225                 (0,0.1)
226             )
227
228
229             sis = np.argsort(si) # Vector for the sorted strengths
230             sit = [['City','s']] # Table for the sorted strengths
231             for i in range(10):
232                 li = sis[-i-1]
233                 sit.append([li2Name[li],si[li]])
234             libF.TextBlock(fig,sit,(1.175,0.5),(0.2,0.4))
235
236
```

```
        # Style
        # libF.CentreFig()
        libF.SetFigStyle(
            r"$s$",r"$P(s)$",
            xScale="log",yScale="log",
            data=figData.fig
        )
        figData.SaveFig('StrengthDistribution')

    def BetweennessCentralityFig(self):
        A = self.A
        di = self.di

        figData = self.figData
        fig = figData.SetFigs()

        G = nx.from_numpy_array(A)
        bc = nx.betweenness_centrality(G,normalized=False)
        bcd = np.array([bc[i] for i in range(len(di))])

        aAvr = np.mean(list(bc.values()))
        libF.TextBlock(
            fig,[
                [libF.DataString(np.min(list(bc.values())),head=r'
                ↪ g_{min}',space=False,formatVal='.3f')],
                [libF.DataString(np.max(list(bc.values())),head=r'
                ↪ g_{max}',space=False,formatVal='.0f')],
                [libF.DataString(aAvr,head=r'\langle
                ↪ g\rangle',space=False,formatVal='.3f')]
            ],(0.75,0.25),
            (0,0.075)
        )

        libF.CreateScatterPlot(
            di,bcd,
            self.figData.fig,
            label=''
        )


        # Style
        # libF.CentreFig()
        libF.SetFigStyle(
            r"$k$",r"$g(i)$",
            # [0.5e1,0.5e3],[1,0.5e5],
            xScale='log',yScale='log',
            data=self.figData.fig
        )
        self.figData.SaveFig('BetweennessCentrality')

    def StrengthVsDegreeFig(self):
        si = self.si
        di = self.di
        dk = self.dk
        Nk = self.Nk

        figData = self.figData
        fig = figData.SetFigs()


        # Scatter
        sk = np.zeros_like(dk,dtype=float)
        for i,ki in enumerate(dk):
            v = np.nonzero(di == ki)[0]
            sk[i] = np.sum(si[v])
            sk[i] /= Nk[i]

```

```
301             libF.CreateScatterPlot(dk,sk,figData.fig)
302 
303             # Fit in log-log space
304             slope = libF.CreateLogRegressionPlot(dk,sk,figData.fig)
305 
306             sAvr = np.mean(si)
307             libF.TextBlock(
308                 fig,[
309                     [libF.DataString(np.min(si),head=r's_{min}',space=
                        ↪ False,formatVal='')],
310                     [libF.DataString(np.max(si),head=r's_{max}',space=
                        ↪ False,formatVal='')],
311                     [libF.DataString(sAvr,head=r'\langle
                        ↪ s\rangle',space=False,formatVal='.3f')],
312                     [libF.DataString(slope,head=r'\alpha',space=False,
                        ↪ formatVal='.3f')]
313                 ],(0.75,0.25),
314                 (0,0.1)
315             )
316 
317 
318             # Style
319             # libF.CentreFig()
320             libF.SetFigStyle(
321                 r"$k$",r"$s(k)$",
322                 xScale="log",yScale="log",
323                 data=figData.fig
324             )
325             figData.SaveFig('StrengthVsDegree')
326 
327         def AClusteringCoefficientFig(self):
328             Nc = self.Nc
329             A = self.A
330             di = self.di
331             dk = self.dk
332             Nk = self.Nk
333 
334             figData = self.figData
335             fig = figData.SetFigs()
336 
337             G = nx.from_numpy_array(A)
338             C = nx.clustering(G)
339             Cd = np.array([C[i] for i in range(Nc)])
340 
341 
342             Ck = np.zeros_like(dk,dtype=float)
343             for i,ki in enumerate(dk):
344                 v = di == ki
345                 Ck[i] = np.sum(Cd[v])/Nk[i]
346             self.Ck = Ck
347 
348             #region
349                 # # Manual counting
350                 # Cd = np.zeros_like(d,dtype=float)
351                 # for i,ki in enumerate(d):
352                 #     if ki>1: # Consider only nodes with more than 2
                        ↪ neighbours
353                 #         vi = A[i,:]
354                 #         indeces = np.nonzero(vi)[0]
355 
356                 #         Ei = 0
357                 #         for n in indeces:
358                 #             for m in indeces:
359                 #                 if A[n,m] == 1:
360                 #                     Ei += 1
361                 #         Ei /= 2 # Divided by 2 since each edge is
                        ↪ counted twice
```

```
362
363             #         Cd[i] = 2*Ei/(ki*(ki-1))
364             #     else:
365             #         Cd[i] = 0
366
367             # Ck = np.zeros_like(k,dtype=float)
368             # for i,ki in enumerate(k):
369             #     v = (d == ki)
370             #     Nk = np.sum(v)
371
372             #     Ck[i] = np.sum(Cd[v])/Nk
373         #endregion
374
375         # CAvr = nx.average_clustering(G)
376         CAvr = Ck.mean()
377         libF.TextBlock(
378             fig,[
379                 [libF.DataString(np.min(Cd),head=r'C_{min}',space=
                    ↪ False,formatVal='.3f')],
380                 [libF.DataString(np.max(Cd),head=r'C_{max}',space=
                    ↪ False,formatVal='.3f')],
381                 [libF.DataString(CAvr,head=r'\langle
                    ↪ C\rangle',space=False,formatVal='.3f')]
382             ],(0.75,0.75),
383             (0,0.075)
384         )
385
386         libF.CreateScatterPlot(dk,Ck,figData.fig,label='')
387
388         # Style
389         # libF.CentreFig()
390         libF.SetFigStyle(
391             r"$k$",r"$C(k)$",
392             # [0,300],[0,0.8],
393             data=figData.fig
394         )
395         figData.SaveFig('AClusteringCoefficient')
396
397     def WClusteringCoefficientFig(self):
398         A = self.A
399         W = self.W
400
401         si = self.si
402         di = self.di
403         dk = self.dk
404         Nk = self.Nk
405         Ck = self.Ck
406
407
408         figData = self.figData
409         fig, ax = figData.SetFigs(2)
410
411
412         # Manual counting
413         Cd = np.zeros_like(di,dtype=float)
414         for i, ki in enumerate(di):
415             if ki>1: # Consider only nodes with more than 2
                    ↪ neighbours
416                 vi = A[i,:]
417                 indices = np.nonzero(vi)[0]
418
419                 Ei = 0
420                 for n in indices:
421                     for m in indices:
422                         Ei += (W[i,n]+W[i,m])*A[n,m]
423                 Ei /= 2 # Divided by 2 since each edge is counted
                        ↪ twice
```

```
424
425                 Cd[i] = Ei/(si[i]*(ki-1))
426             else:
427                 Cd[i] = 0
428
429
430         Ckw = np.zeros_like(dk,dtype=float)
431         for i, ki in enumerate(dk):
432             v = di == ki
433             Ckw[i] = np.sum(Cd[v])/Nk[i]
434
435         libF.CreateScatterPlot(dk,Ckw,figData.fig1,label='',ax=ax[
            ↪  0])
436
437         # Style
438         libF.SetFigStyle(
439             r"$k$",r"$C^w(k)$",
440             xScale="log",
441             ax=ax[0],
442             data=figData.fig1
443         )
444
445
446         CkwRel= (Ckw-Ck)/Ck
447         libF.CreateScatterPlot(dk,CkwRel,figData.fig2,label='',ax=
            ↪  ax[1])
448         self.CkwRel = CkwRel
449
450         # Style
451         libF.SetFigStyle(
452             r"$k$",r"$C^w_{rel}(k)$",
453             xScale="log",yScale="log",
454             ax=ax[1],
455             data=figData.fig2
456         )
457
458
459         # libF.CentreFig()
460         figData.SaveFig('WClusteringCoefficient')
461
462     def AAssortativityFig(self):
463         A = self.A
464         dk = self.dk
465
466         figData = self.figData
467         fig = figData.SetFigs()
468
469         G = nx.from_numpy_array(A)
470         ad = nx.average_neighbor_degree(G)
471         ak = nx.average_degree_connectivity(G)
472
473         aAvr = np.mean(list(ad.values()))
474         libF.TextBlock(
475             fig,[
476                 [libF.DataString(np.min(list(ad.values())),head=r'
                    ↪  k_{nn}^{min}',space=False,formatVal='.3f')],
477                 [libF.DataString(np.max(list(ad.values())),head=r'
                    ↪  k_{nn}^{max}',space=False,formatVal='.3f')],
478                 [libF.DataString(aAvr,head=r'\langle k_{nn}\rangle
                    ↪  ',space=False,formatVal='.3f')]
479             ],(0.75,0.75),
480             (0,0.075)
481         )
482
483         knn = np.array([ak[ki] for ki in dk])
484         libF.CreateScatterPlot(dk,knn,figData.fig,label='')
485         self.knn = knn
```

```
486
487
488             # Style
489             # libF.CentreFig()
490             libF.SetFigStyle(
491                 r"$k$",r"$k_{nn}(k)$",
492                 # [0,300],[40,95],
493                 data=figData.fig
494             )
495             figData.SaveFig('AAssortativity')
496
497         def WAssortativityFig(self):
498             W = self.W
499             dk = self.dk
500             knn = self.knn
501
502             figData = self.figData
503             fig, ax = figData.SetFigs(2)
504
505             Gw = nx.from_numpy_array(W)
506             # adw = nx.average_neighbor_degree(Gw,weight="weight")
507             akw = nx.average_degree_connectivity(Gw,weight="weight")
508
509
510             knnw = np.array([akw[ki] for ki in dk])
511             libF.CreateScatterPlot(dk,knnw,figData.fig1,label='',ax=ax
                ↪ [0])
512
513             # Style
514             libF.SetFigStyle(
515                 r"$k$",r"$k_{nn}^w(k)$",
516                 xScale="log",yScale="log",
517                 ax=ax[0],
518                 data=figData.fig1
519             )
520
521
522             knnwRel = (knnw-knn)/knn
523             libF.CreateScatterPlot(dk,knnwRel,figData.fig2,label='',ax
                ↪ =ax[1])
524
525             # Style
526             libF.SetFigStyle(
527                 r"$k$",r"$k_{nn,rel}^w(k)$",
528                 xScale="log",yScale="log",
529                 ax=ax[1],
530                 data=figData.fig2
531             )
532
533
534             # libF.CentreFig()
535             figData.SaveFig('WAssortativity')
536
537         def ShowFig(self):
538             from matplotlib.pyplot import show
539             show()
```

Il sesto e ultimo modulo è `libKTMAS.py` che racchiude tutte le funzioni legate alla TCSMA e all'elaborazione dei risultati cosí ottenuti. È notevole menzionare che il ciclo principale, descritto dalla funzione `MonteCarloAlgorithm()`, è avviato parallelamente con al piú tre lavoratori contemporanei; tale limite discende fondamentalmente dal fatto che la macchina su cui sono state svolte le simulazioni dispone di soli quattro nuclei di calcolo.

Listato A.7: Modulo `libKTMAS.py`.

```
# Library to apply the Kinetic Theory for Multi-Agent Systems

from dataclasses import dataclass

from concurrent.futures import ProcessPoolExecutor
import multiprocessing as mp
from multiprocessing import shared_memory
import time
# from tqdm import tqdm

import numpy as np
from scipy import stats
from numba import njit

from matplotlib.pyplot import get_cmap
from matplotlib.colors import LogNorm

from libData import WriteSimulationData
from libParameters import CopyWorkerShMTemplate
workersShM = CopyWorkerShMTemplate()
import libFigures as libF


### Main classes ###

class KineticSimulation():
    def __init__(self,clsPrm,clsReg):
        self.il = int(clsPrm.interactionRule)

        self.l = float(clsPrm.attractivity)
        self.a = float(clsPrm.convincibility)
        self.s = clsPrm.fluctuations*float(clsPrm.deviation)
        self.z = clsPrm.zetaFraction*float(clsPrm.zetaValue)

        self.Ni = int(clsPrm.iterations)
        Nc = int(clsReg.Nc); self.Nc = Nc
        self.li2Name = clsReg.li2Name

        self.progressBar = clsPrm.progressBar
        dt = float(clsPrm.timestep); self.dt = dt
        Nt = int(clsPrm.timesteps); self.Nt = Nt

        Ns = int(clsPrm.snapshots); self.Ns = Ns
        ns = np.array(
            [i*Nt/Ns for i in range(Ns+1)],
            dtype=np.int64
        ); self.ns = ns

        Nw = clsPrm.smoothingFactor; self.Nw = Nw
        ks = int(Ns/Nw); self.ks = ks # Kernel size
        self.times = ns[::ks]*dt

        self.R = int(clsPrm.region)
        self.P = int(clsPrm.population)
        self.p0 = float(self.P/self.Nc)
        self.realSizeDistr = clsReg.sizeDistr
        self.logRealSizeDistr = np.log10(clsReg.sizeDistr)

        # Exact unitary adjacency matrix
        M = clsReg.A
        self.dk = np.sum(M,axis=1,dtype=np.int32)
        self.Mdt = M*dt

        # Approximated adjacency matrix
```

```
65          Mn = np.sum(M)
66          self.wOdt = np.sum(M[:,:],axis=1)*dt/Mn
67          self.wI = np.sum(M[:,:],axis=0)
68          # M[:,:,1]  = wO@wI/Mn
69          # In this case wO=wI but it's better to define them
            ↪ rigorously
70
71          self.ew = clsPrm.edgeWeights
72          if clsPrm.edgeWeights:
73              self.di = np.sum(clsReg.W/np.max(clsReg.W),axis=1,dtyp
                ↪ e=np.float64)
74          else:
75              self.di = np.sum(clsReg.A,axis=1,dtype=np.int32)
76          self.idi = np.array(1.0/self.di,dtype=np.float64) # Inverse
            ↪ degrees
77          # self.D = di@invdi.T
78
79          self.typ = np.array([0,1],dtype=np.int64)
80          self.lblEn = ('Ext.','Apx.','Real')
81          self.lblIt = (
82              ('esatto','appr.','reale'),
83              ('esatta','appr.','reale'),
84              ('Esatto','Appr.','Reale'),
85          )
86          self.clr = ('#0072B2','#D55E00','#000000')
87          # self.clr = ('blue','red')
88
89          if clsPrm.parametricStudy:
90              self.figData = None
91              self.Nv = clsPrm.numberPrmStudy
92              self.studiedPrmString = None
93          else:
94              self.figData = libF.FigData(clsPrm,'KS')
95              self.Nv = None
96              self.fw = 8 # Figure width
97              self.fh = 6 # Figure height
98          self.sid = None
99
100     # Simulation
101     def MonteCarloSimulation(self):
102         Ni = self.Ni
103         Nc = self.Nc
104         p0 = self.p0
105
106         gui = self.progressBar
107         Nt = self.Nt
108         Ns = self.Ns
109
110         sid = self.sid
111         Nv = self.Nv
112
113         l = self.l
114         a = self.a
115         s = self.s
116         z = self.z
117         il = self.il
118         typ = self.typ.size
119
120         process = range(Ni)
121
122         shmPrm = {}
123         shmHandles = {}
124
125         for key in workersShM['parameters'].keys():
126             spec, shm = BuildShM(getattr(self,key))
127             shmPrm[key] = spec
128             shmHandles[key] = shm
```

```
129
130             ctx = mp.get_context("spawn")
131
132             if gui:
133                 for key,dtype in workersShM['gui'].items():
134                     spec, shm = BuildShM(Ni=Ni,dtype=dtype)
135                     shmPrm[key] = spec
136                     shmHandles[key] = shm
137
138                 # Import of ProgressBarsGUI locally [and not at the top
                    ↪ of the module] to avoid freezing the GUI when in a
                    ↪ parametric study (Tk must live only in the parent
                    ↪ thread but spawn under Windows imports it in each
                    ↪ worker [if written at the top of the module] making
                    ↪ it prone to errors)
139                 from libGUIs import ProgressBarsGUI
140                 bar = ProgressBarsGUI(Ni,Nt,sid,Nv,shmPrm,LoadShM)
141
142                 try:
143                     with ProcessPoolExecutor(
144                         max_workers=3,
145                         mp_context=ctx,
146                         initializer=InitializeMCSWorker,
147                         initargs=(shmPrm,True)
148                     ) as executor:
149                         futures = {}
150                         for p in range(Ni):
151                             futures[p] = executor.submit(
152                                 MonteCarloAlgorithm,
153                                 p,Nc,Ns,p0,typ,
154                                 l,a,s,z,il,gui
155                             )
156                         bar.mainloop()
157
158                 finally:
159                     data = [None]*Ni
160                     for p, fut in futures.items():
161                         data[p] = fut.result()
162                     self.data = data
163
164                     for shm in shmHandles.values():
165                         shm.close(); shm.unlink()
166
167             else:
168                 try:
169                     with ProcessPoolExecutor(
170                         max_workers=3,
171                         mp_context=ctx,
172                         initializer=InitializeMCSWorker,
173                         initargs=(shmPrm,)
174                     ) as executor:
175                         data = list(
176                             executor.map(
177                                 MonteCarloAlgorithm,
178                                 process,[Nc]*Ni,[Ns]*Ni,
179                                 [p0]*Ni,[typ]*Ni,[l]*Ni,
180                                 [a]*Ni,[s]*Ni,[z]*Ni,
181                                 [il]*Ni,[gui]*Ni
182                             )
183                         ); self.data = data
184                 finally:
185                     for shm in shmHandles.values():
186                         shm.close(); shm.unlink()
187
188             self.EvaluateSimulationData()
189             WriteSimulationData(
190                 self.vrtState,
```

```
            self.snapshots,
            self.siVrtState,
            self.typ,
            self.lblEn,
            self.li2Name,
            Ni,
            self.sid
        )

    def EvaluateSimulationData(self):
        Ni = self.Ni
        data = self.data
        Nw = self.Nw
        ks = self.ks

        def Convolve(v):
            Nrw, _, Nty = v.shape # Number of rows, times and types

            filter = np.array([float(1/ks)]*ks)
            sv = np.zeros((Nrw,Nw+1,Nty),dtype=float) # Smoothed
            ↪  vector

            for r in range(Nrw):
                for t in range(Nty):
                    sv[r,0,t] = v[r,0,t].copy()
                    conv = np.convolve(
                        v[r,1:,t],
                        filter,
                        'valid'
                    )
                    sv[r,1:,t] = conv[::ks].copy()

            # This functions applies the mean every ks unit of the
            ↪  Ns snapshots
            return sv

        self.vrtState = np.array([data[p][0] for p in range(Ni)])
        self.logVrtState = np.log10(np.array([data[p][0] for p in
        ↪  range(Ni)]))
        self.snapshots = np.array([data[p][1] for p in range(Ni)])
        self.avrState = Convolve(np.mean(self.snapshots,axis=1))

        self.logNodeSimAvrVrtState =
        ↪  np.mean(self.logVrtState,axis=1)
        self.logNodeSimMinVrtState = np.min(self.logVrtState,axis=1)
        self.logNodeSimMaxVrtState = np.max(self.logVrtState,axis=1)
        self.logNodeSimSumVrtState = np.sum(self.logVrtState,axis=1)

        self.logNodeRealAvrVrtState = np.mean(self.logRealSizeDistr)
        self.logNodeRealMinVrtState = np.min(self.logRealSizeDistr)
        self.logNodeRealMaxVrtState = np.max(self.logRealSizeDistr)
        self.logNodeRealSumVrtState = np.sum(self.logRealSizeDistr)

        if Ni>1:
            self.ta = stats.t.ppf(0.975,df=Ni-1)

            self.itAvrVrtState = np.mean(self.vrtState,axis=0)
            self.itAvrSnapshots = np.mean(self.snapshots,axis=0)
        else:
            self.ta = None

            self.itAvrVrtState = self.vrtState[0,:,:]
            self.itAvrSnapshots = self.snapshots[0,:,:,:]

        self.itAvrConvSnapshots = Convolve(self.itAvrSnapshots)
        self.siVrtState = np.argsort(self.vrtState,axis=1)
        self.siAvrState = np.argsort(self.itAvrVrtState,axis=0)
```

```
254
255         # Figures
256         def SizeDistrFittingsFig(
257             self,
258             ax=None,
259             idx=1,
260             figData=None,
261             saveFig=False
262         ):
263             Ni = self.Ni
264             ta = self.ta
265
266             csSv = self.vrtState
267             csRv = self.realSizeDistr.reshape(1,-1)
268
269             logCsSv = self.logVrtState
270             logCsRv = self.logRealSizeDistr.reshape(1,-1)
271
272             logCsSAvr = self.logNodeSimAvrVrtState
273             logCsSMin = self.logNodeSimMinVrtState
274             logCsSMax = self.logNodeSimMaxVrtState
275             logCsSSum = self.logNodeSimSumVrtState
276
277             logCsRAvr = self.logNodeRealAvrVrtState
278             logCsRMin = self.logNodeRealMinVrtState
279             logCsRMax = self.logNodeRealMaxVrtState
280             logCsRSum = self.logNodeRealSumVrtState
281
282             typ = self.typ
283             # lbl = self.lblEn
284             lbl = self.lblIt
285             clr = self.clr
286
287             Nf = 9 # Numbers of figures
288             if figData is None:
289                 figData = self.figData
290                 fig, ax =
                    ↪  figData.SetFigs(1,Nf,size=(self.fw*Nf,self.fh))
291                 saveFig = True
292
293             sMaxEA = csSv.max(); sMinEA = csSv.min()
294
295             sMaxER = max(csSv[:,:,0].max(),csRv.max())
296             sMinER = min(csSv[:,:,0].min(),csRv.min())
297
298             sMaxAR = max(csSv[:,:,1].max(),csRv.max())
299             sMinAR = min(csSv[:,:,1].min(),csRv.min())
300
301             def EstimateBinNumber(v):
302                 vf = v.ravel()
303                 vl = np.log10(vf[vf>0])
304
305                 # edges = np.histogram_bin_edges(vl,bins='fd')
306                 edges = np.histogram_bin_edges(vl,bins='doane')
307                 # edges = np.histogram_bin_edges(vl,bins='scott')
308                 # edges = np.histogram_bin_edges(vl,bins='stone')
309
310                 return len(edges)-1
311
312             nBins = np.zeros((3,),dtype=np.int32)
313             for i,v in enumerate([csSv[:,:,0],csSv[:,:,1],csRv]):
314                 for j in range(v.shape[0]):
315                     nBin = EstimateBinNumber(v[j,:])
316                     if nBin > nBins[i]: nBins[i] = nBin
317
318             p = (.5,1.075); dp = (1,.05)
319             axis = 4; xScale='lin';
```

```
320         string = 'h' if xScale == 'lin' else 's'
321
322         for t in typ: # t[ype]
323             ### Lognormal fit ###
324
325             # Exact-Approximated plot
326             libF.CreateHistogramPlot(
327                 csSv[:,:,t] if xScale == 'log' else logCsSv[:,:,t],
328                 np.max(nBins[:2]),
329                 getattr(figData,f'fig{idx}'),
330                 limits=(
331                     (sMinEA,sMaxEA)
332                 ) if xScale == 'log' else (
333                     (np.log10(sMinEA),np.log10(sMaxEA))
334                 ),
335                 xScale=xScale,
336                 Ni=Ni,
337                 ta=ta,
338                 # label=f'{lbl[t]} histogram',
339                 label=f'Istogramma {lbl[0][t]}',
340                 color=clr[t],
341                 alpha=(0.35,0.45) if Ni>1 else 0.35,
342                 idx=t+1,
343                 ax=ax[0]
344             )
345             xiS,m1S,s1S,m2S,s2S = libF.CreateLognormalFitPlot(
346                 csSv[:,:,t] if xScale == 'log' else logCsSv[:,:,t],
347                 getattr(figData,f'fig{idx}'),
348                 limits=(
349                     (sMinEA,sMaxEA)
350                 ) if xScale == 'log' else (
351                     (np.log10(sMinEA),np.log10(sMaxEA))
352                 ),
353                 xScale=xScale,
354                 Ni=Ni,
355                 ta=ta,
356                 # label=f'{lbl[t]} lognormal fit (ML)',
357                 label=fr"Adatt. BLN {lbl[0][t]}",# ($\mathit{{ML}}$)
358                 # (
359                 #     fr'{lbl[t]} mean value $\langle k\rangle$',
360                 #     f'{lbl[t]} lognormal fit (ML)'
361                 # ),
362                 color=clr[t],#(clr[t],clr[t]),
363                 alpha=(1,0.15) if Ni>1 else 1,
364                 bimodal=True,
365                 idx=t+1,
366                 ax=ax[0]
367             )
368
369             # Exact-Real and Approximated-Real plots
370             libF.CreateHistogramPlot(
371                 csSv[:,:,t] if xScale == 'log' else logCsSv[:,:,t],
372                 max(nBins[t],nBins[2]),
373                 getattr(figData,f'fig{idx+t+1}'),
374                 limits=(
375                     (sMinER,sMaxER) if t == 0 else (sMinAR,sMaxAR)
376                 ) if xScale == 'log' else (
377                     (np.log10(sMinER),np.log10(sMaxER)) if t == 0
378                     else (np.log10(sMinAR),np.log10(sMaxAR))
379                 ),
380                 xScale=xScale,
381                 Ni=Ni,
382                 ta=ta,
383                 # label=f'{lbl[t]} histogram',
384                 label=f'Istogramma {lbl[0][t]}',
385                 color=clr[t],
386                 alpha=(0.35,0.45) if Ni>1 else 0.35,
```

```
                    idx=1,
                    ax=ax[0+t+1]
                )
                libF.CreateLognormalFitPlot(
                    csSv[:,:,t] if xScale == 'log' else logCsSv[:,:,t],
                    getattr(figData,f'fig{idx+t+1}'),
                    limits=(
                        (sMinER,sMaxER) if t == 0 else (sMinAR,sMaxAR)
                    ) if xScale == 'log' else (
                        (np.log10(sMinER),np.log10(sMaxER)) if t == 0
                        else (np.log10(sMinAR),np.log10(sMaxAR))
                    ),
                    xScale=xScale,
                    Ni=Ni,
                    ta=ta,
                    # label=f'{lbl[t]} lognormal fit (ML)',
                    label=fr"Adatt. BLN {lbl[0][t]}",# ($\mathit{{ML}}$)
                    color=clr[t],#(clr[t],clr[t]),
                    alpha=(1,0.15) if Ni>1 else 1,
                    bimodal=True,
                    idx=1,
                    ax=ax[0+t+1]
                )

                libF.CreateHistogramPlot(
                    csRv if xScale == 'log' else logCsRv,
                    max(nBins[t],nBins[2]),
                    getattr(figData,f'fig{idx+t+1}'),
                    limits=(
                        (sMinER,sMaxER) if t == 0 else (sMinAR,sMaxAR)
                    ) if xScale == 'log' else (
                        (np.log10(sMinER),np.log10(sMaxER)) if t == 0
                        else (np.log10(sMinAR),np.log10(sMaxAR))
                    ),
                    xScale=xScale,
                    # label=f'{lbl[2]} histogram',
                    label=f'Istogramma {lbl[0][2]}',
                    color=clr[2],
                    alpha=0.35,
                    idx=2,
                    ax=ax[0+t+1]
                )
                xiR,m1R,s1R,m2R,s2R = libF.CreateLognormalFitPlot(
                    csRv if xScale == 'log' else logCsRv,
                    getattr(figData,f'fig{idx+t+1}'),
                    limits=(
                        (sMinER,sMaxER) if t == 0 else (sMinAR,sMaxAR)
                    ) if xScale == 'log' else (
                        (np.log10(sMinER),np.log10(sMaxER)) if t == 0
                        else (np.log10(sMinAR),np.log10(sMaxAR))
                    ),
                    xScale=xScale,
                    # label=f'{lbl[2]} lognormal fit (ML)',
                    label=fr"Adatt. BLN {lbl[0][2]}",# ($\mathit{{ML}}$)
                    color=clr[2],
                    alpha=1,
                    bimodal=True,
                    idx=2,
                    ax=ax[0+t+1]
                )


                ### Power law vs bimodal lognormal fit log-log ###

                if t: libF.CreateParetoFitPlot(
                    csRv,
                    getattr(figData,f'fig{idx+3}'),
```

```
                yScale='log',
                # label=(
                #     f'{lbl[2]} empirical CCDF',
                #     f'{lbl[2]} Pareto fit (ML)',
                #     f'{lbl[2]} bimodal lognormal fit (ML)'
                # ),
                label=(
                    f'FRC empirica {lbl[1][2]}',
                    fr"Adatt. Pareto {lbl[0][2]}",# ($\mathit{{ML}}$)
                    fr"Adatt. BLN {lbl[0][2]}"# ($\mathit{{ML}}$)
                ),
                color=(clr[2],clr[2]),
                alpha=(0.6,1),
                bimodal=True,
                idx=1,
                ax=ax[3]
            )
            libF.CreateParetoFitPlot(
                csSv[:,:,t],
                getattr(figData,f'fig{idx+3+t+1}'),
                upperbound=sMaxEA,
                yScale='log',
                Ni=Ni,
                ta=ta,
                # label=(
                #     f'{lbl[t]} empirical CCDF',
                #     f'{lbl[t]} Pareto fit (ML)',
                #     f'{lbl[t]} bimodal lognormal fit (ML)'
                # ),
                label=(
                    f'FRC empirica {lbl[1][t]}',
                    fr"Adatt. Pareto {lbl[0][t]}",# ($\mathit{{ML}}$)
                    fr"Adatt. BLN {lbl[0][t]}"# ($\mathit{{ML}}$)
                ),
                color=(clr[t],clr[t]),
                alpha=((0.6,0.3),(1,0.15)) if Ni>1 else (0.6,1),
                bimodal=True,
                idx=1,
                ax=ax[3+t+1]
            )


            ### Power law fit log-log ###

            # Exact-Approximated plot
            libF.CreateParetoFitPlot(
                csSv[:,:,t],
                getattr(figData,f'fig{idx+6}'),
                upperbound=sMaxEA,
                yScale='log',
                Ni=Ni,
                ta=ta,
                # label=(
                #     f'{lbl[t]} empirical CCDF',
                #     f'{lbl[t]} Pareto fit (ML)'
                # ),
                label=(
                    f'FRC empirica {lbl[1][t]}',
                    fr"Adatt. Pareto {lbl[0][t]}"# ($\mathit{{ML}}$)
                ),
                color=(clr[t],clr[t]),
                alpha=((0.6,0.3),(1,0.15)) if Ni>1 else (0.6,1),
                idx=t+1,
                ax=ax[6]
            )
```

```
519
520             # Exact-Real and Approximated-Real plots
521             blS = libF.CreateParetoFitPlot(
522                 csSv[:,:,t],
523                 getattr(figData,f'fig{idx+6+t+1}'),
524                 upperbound=sMaxER if t == 0 else sMaxAR,
525                 yScale='log',
526                 Ni=Ni,
527                 ta=ta,
528                 # label=(
529                 #     f'{lbl[t]} empirical CCDF',
530                 #     f'{lbl[t]} Pareto fit (ML)'
531                 # ),
532                 label=(
533                     f'FRC empirica {lbl[1][t]}',
534                     fr"Adatt. Pareto {lbl[0][t]}"# ($\mathit{{ML}}$)
535                 ),
536                 color=(clr[t],clr[t]),
537                 alpha=((0.6,0.3),(1,0.15)) if Ni>1 else (0.6,1),
538                 idx=1,
539                 ax=ax[6+t+1]
540             )
541             blR = libF.CreateParetoFitPlot(
542                 csRv,
543                 getattr(figData,f'fig{idx+6+t+1}'),
544                 upperbound=sMaxER if t == 0 else sMaxAR,
545                 yScale='log',
546                 # label=(
547                 #     f'{lbl[2]} empirical CCDF',
548                 #     f'{lbl[2]} Pareto fit (ML)'
549                 # ),
550                 label=(
551                     f'FRC empirica {lbl[1][2]}',
552                     fr"Adatt. Pareto {lbl[0][2]}"# ($\mathit{{ML}}$)
553                 ),
554                 color=(clr[2],clr[2]),
555                 alpha=(0.6,1),
556                 idx=2,
557                 ax=ax[6+t+1]
558             )
559
560
561             libF.Text(
562                 ax[axis],(p[0],p[1]-t*dp[1]),
563                 # fr'{lbl[t]}:$\quad$'+
564                 fr"{lbl[2][t]}:$\quad$"+
565                 libF.DataString(xiS,Ni,ta,r'\xi')+
566                 libF.DataString(m1S,Ni,ta,r'\mu_1')+
567                 libF.DataString(s1S,Ni,ta,r'\sigma_1')+
568                 libF.DataString(m2S,Ni,ta,r'\mu_2')+
569                 libF.DataString(s2S,Ni,ta,r'\sigma_2')+
570                 libF.DataString(logCsSMin[:,t],Ni,ta,fr'{string}_{ ↵
                    ↪  {min}}')+
571                 libF.DataString(logCsSMax[:,t],Ni,ta,fr'{string}_{ ↵
                    ↪  {max}}')+
572                 libF.DataString(logCsSAvr[:,t],Ni,ta,fr'\langle ↵
                    ↪  {string}\rangle')+
573                 libF.DataString(
574                     logCsSSum[:,t],Ni,ta,fr'{string}_{{\Sigma}}',
575                     formatVal='.2e',formatErr='.2e'
576                 )+
577                 libF.DataString(blS,Ni,ta,r'\beta',space=False),
578                 ha='center',
579                 color=clr[t]
580             )
581
582         libF.Text(
```

```
            ax[axis],(p[0],p[1]+dp[1]),
            # fr'{lbl[t]}:$\quad$'j
            fr"{lbl[2][2]}:$\quad$"+
            libF.DataString(xiR,head=r'\xi')+
            libF.DataString(m1R,head=r'\mu_1')+
            libF.DataString(s1R,head=r'\sigma_1')+
            libF.DataString(m2R,head=r'\mu_2')+
            libF.DataString(s2R,head=r'\sigma_2')+
            libF.DataString(logCsRMin,head=fr'{string}_{{min}}')+
            libF.DataString(logCsRMax,head=fr'{string}_{{max}}')+
            libF.DataString(logCsRAvr,head=fr'\langle
            ↪  {string}\rangle')+
            libF.DataString(
                logCsRSum,head=fr'{string}_{{\Sigma}}',
                formatVal='.2e',formatErr='.2e'
            )+
            libF.DataString(blR,head=r'\beta',space=False),
            ha='center',
            color=clr[2]
        )

        if idx == 1:
            libF.TextBlock(
                ax[1],[[
                    fr'$Nc={self.Nc}$',
                    fr'$R={self.R}$',
                    # libF.DataString(blR,head=r'\beta',space=Fals
                    ↪  e),
                    # libF.DataString(xiR,head=r'\xi',space=False)
                ]],
                p=(.5,1.01),
                dp=(dp[0]/6,dp[1])
            )

            # p = (.5,p[1])
            # dp = (.8,dp[1])
            libF.TextBlock(
                ax[-2],[[
                    fr'$\lambda={self.l}$',
                    fr'$\sigma={self.s}$',
                    fr'$ζ={self.z}$',
                    fr'$Ni={Ni}$',
                    fr'$Ns={self.Ns}$',
                ],[
                    fr'$\alpha={self.a}$',
                    fr'$il={self.il+1}$',
                    fr'$d_i={'w_i' if self.ew else 'k_i'}$',
                    fr'$\Delta t={self.dt}$',
                    fr'$Nw={self.Nw}$',
                ]],
                p=(.5,1.05),
                dp=(dp[0]/1.4,dp[1])
            )

        for f in range(Nf):
            libF.SetFigStyle(
                # r'$s$',r'$\bar f_N(s)$',
                r'$r$',r'$\bar f_N(r)$' if f<3 else r'$FRC(r)$',
                yNotation='sci' if f<3 else 'plain',
                xScale=(
                    'lin' if f>2 else 'log'
                ) if xScale == 'log' else ('lin'),
                yScale='log' if f>2 else 'lin',
                ax=ax[f],
                data=getattr(figData,f'fig{idx+f}')
            )

```

```
        # CentreFig()
        if saveFig: figData.SaveFig('SizeDistributionFittings')
        else: libF.Text(ax[-1],(1.1,.5),self.studiedPrmString)

    def AverageSizeFig(
        self,
        ax=None,
        idx=1,
        figData=None,
        saveFig=False
    ):
        Ni = self.Ni
        ta = self.ta

        times = self.times
        csSa = self.avrState
        csRa = np.ones_like(times)*self.realSizeDistr.mean()

        # lbl = self.lblEn
        lbl = self.lblIt
        clr = self.clr

        if figData is None:
            figData = self.figData
            fig = figData.SetFigs()
            saveFig = True

        for t in self.typ:
            libF.CreateFunctionPlot(
                times,
                csSa[:,:,t],
                figData.fig if saveFig
                else getattr(figData,f'fig{idx}'),
                Ni=Ni,
                ta=ta,
                # label=rf"{lbl[t]} city size average $\langle
                ↪  s\rangle$",
                label=fr'Taglia media {lbl[1][t]}',
                color=clr[t],
                alpha=(1,0.15) if Ni>1 else 1,
                idx=t+1,
                ax=ax
            )

        libF.CreateFunctionPlot(
            times,
            csRa,
            figData.fig if saveFig
            else getattr(figData,f'fig{idx}'),
            linestyle='--',
            linewidth=0.75,
            # label=rf"{lbl[2]} city size average $\langle
            ↪  s\rangle$",
            label=rf'Taglia media {lbl[1][2]}',
            color=clr[2],
            alpha=1,
            idx=3,
            ax=ax
        )

        # CentreFig()
        libF.SetFigStyle(
            r"$t$",r"$\langle s\rangle$",
            data=figData.fig if saveFig
            else getattr(figData,f'fig{idx}'),
            ax=ax
        )
```

```
713
714             if saveFig: figData.SaveFig('AverageSize')
715             else: libF.Text(ax,(1.1,.5),self.studiedPrmString)
716
717         def SizeVsDegreeFig(
718             self,
719             ax=None,
720             idx=1,
721             figData=None,
722             saveFig=False
723         ):
724             ki = self.dk
725
726             csSv = self.itAvrVrtState
727             csRv = self.realSizeDistr
728
729             typ = self.typ
730             # lbl = self.lblEn
731             lbl = self.lblIt
732             clr = self.clr
733
734             si = self.siAvrState
735             li2Name = self.li2Name
736
737             if figData is None:
738                 figData = self.figData
739                 Nf = 3 # Numbers of figures
740                 fig, ax =
                    ↪  figData.SetFigs(1,Nf,size=(self.fw*Nf,self.fh))
741                 saveFig = True
742
743             for t in typ:
744                 libF.CreateScatterPlot(
745                     ki,csSv[:,t],
746                     getattr(figData,f'fig{idx}'),
747                     # label=lbl[t],
748                     label=f'Dispersione {lbl[1][t]}',
749                     color=clr[t],
750                     alpha=0.7,
751                     idx=t+1,
752                     ax=ax[0]
753                 )
754
755                 libF.CreateScatterPlot(
756                     ki,csSv[:,t],
757                     getattr(figData,f'fig{t+1+idx}'),
758                     # label=lbl[t],
759                     label=f'Dispersione {lbl[1][t]}',
760                     color=clr[t],
761                     alpha=0.6,
762                     idx=1,
763                     ax=ax[t+1]
764                 )
765
766                 libF.CreateScatterPlot(
767                     ki,csRv,
768                     getattr(figData,f'fig{t+1+idx}'),
769                     # label=lbl[2],
770                     label=f'Dispersione {lbl[1][2]}',
771                     color=clr[2],
772                     alpha=0.6,
773                     idx=2,
774                     ax=ax[t+1]
775                 )
776
777             for f in range(3):
778                 libF.SetFigStyle(
```

```
                r'$k$',r'$s(k)$',
                yScale='log',xScale='log',
                data=getattr(figData,f'fig{f+idx}'),
                ax=ax[f]
            )

        for t in typ:
            libF.TextBlock(
                ax[0],[[
                    libF.DataString(
                        csSv[si[i,t],t],
                        head=li2Name[si[i,t]],
                        formatVal='.2e',
                        space=False
                    )
                ] for i in range(-1,-6,-1)],
                p=(.8,.35-t*.225),
                dp=(0,.15),
                ha='center',
                color=clr[t]
            )

        # CentreFig()
        if saveFig: figData.SaveFig('SizeVsDegree')
        else: libF.Text(ax[-1],(1.1,.5),self.studiedPrmString)

    def SizeDistrEvolutionFig(
        self,
        ax=None,
        idx=1,
        figData=None,
        saveFig=False
    ):
        snapshots = self.itAvrSnapshots

        ns = self.ns
        Ns = self.Ns

        dt = self.dt
        typ = self.typ

        if figData is None:
            figData = self.figData
            Nf = 2 # Numbers of figures
            fig, ax =
            ↪  figData.SetFigs(1,Nf,size=(self.fw*Nf,self.fh))
            saveFig = True
        ax[0].set_title('Exact'); ax[1].set_title('Approximated')

        samples = 6
        clrmap = get_cmap('inferno') # magma
        colours = [clrmap(i/(samples-1)) for i in range(samples)]

        sMax = np.max(snapshots[:,-1,:])
        sMin = np.min(snapshots[:,-1,:])

        for t in typ: # t[ype]
            for j,s in
            ↪  enumerate(np.linspace(0,Ns,samples,dtype=int)):
                libF.CreateHistogramPlot(
                    snapshots[:,s,t],21,
                    getattr(figData,f'fig{t+idx}'),
                    limits=(sMin,sMax),
                    xScale='log',
                    label=f"t = {int(ns[s]*dt)}",
                    color=colours[j][:-1],
                    alpha=0.4,
```

```
                    idx=j+idx,
                    ax=ax[t],
                    norm=False
                ) # Histogram plot

            libF.SetFigStyle(
                r"$cs$",r"$P(cs)$",
                yNotation="sci", # ,xNotation="sci"
                xScale='log',
                yScale='log',
                ax=ax[t],
                data=getattr(figData,f'fig{t+idx}')
            ) # Style

        # CentreFig()
        if saveFig: figData.SaveFig('SizeDistributionEvolution')
        else: libF.Text(ax[-1],(1.1,.5),self.studiedPrmString)

    def SizeEvolutionsFig(
        self,
        ax=None,
        idx=1,
        figData=None,
        saveFig=False
    ):
        # Nc = self.Nc
        typ = self.typ

        times = self.times
        snapshots = self.itAvrConvSnapshots

        ki = self.dk
        sf = self.Nw

        if figData is None:
            figData = self.figData
            Nf = 2 # Numbers of figures
            fig, ax =
            ↪  figData.SetFigs(1,Nf,size=(self.fw*Nf,self.fh))
            saveFig = True
        ax[0].set_title('Exact'); ax[1].set_title('Approximated')

        dk, _ = np.unique(ki,return_counts=True); Nk = dk.size
        snapshotsk = np.zeros((Nk,sf+1,2),dtype=np.float64)
        for i,k in enumerate(dk):
            bk = ki == k
            snapshotsk[i,:,:] = snapshots[bk,:,:].mean(axis=0)

        clrmap = get_cmap('inferno') # magma
        norm = LogNorm(vmin=dk.min(),vmax=dk.max())
        colours = clrmap(norm(dk[::-1]))
        # colours = [clrmap(i/(Nc-1)) for i in range(Nc)]

        labels = ['']*Nk
        classes = np.linspace(0,Nk-1,4,dtype=np.int64)
        for i in classes:
            # labels[i] = f'Class k={dk[i]}'
            labels[i] = f'Classe k={dk[i]}'

        for t in typ: # t[ype]
            # norm = LogNorm(
            #     vmin=snapshotsk[:,-1,t].min(),
            #     vmax=snapshotsk[:,-1,t].max()
            # )
            # colours = clrmap(norm(snapshotsk[:,-1,t]))

            for k in np.linspace(Nk-1,0,Nk,dtype=np.int64):
```

```
                libF.CreateFunctionPlot(
                    times,snapshotsk[k,:,t],
                    getattr(figData,f'fig{t+idx}'),
                    color=colours[k,:],
                    alpha=0.8,
                    linewidth=1,
                    label=labels[k],
                    idx=k+idx,
                    ax=ax[t]
                ) # Histogram plot

            libF.SetFigStyle(
                r"$t$",r"$s$",
                # yNotation="sci", # ,xNotation="sci"
                yScale='log',
                data=getattr(figData,f'fig{t+idx}'),
                ax=ax[t]
            ) # Style

        # CentreFig()
        if saveFig: figData.SaveFig('SizeEvolutions')
        else: libF.Text(ax[-1],(1.1,.5),self.studiedPrmString)

    def ShowFig(self):
        from matplotlib.pyplot import show
        show()

class ParametricStudy():
    def __init__(self,clsPrm,clsReg):
        sp = clsPrm.studiedParameter

        sV = clsPrm.startValuePrmStudy; self.sV = sV
        eV = clsPrm.endValuePrmStudy; self.eV = eV
        Nv = clsPrm.numberPrmStudy; self.Nv = Nv

        self.KS = [None]*Nv
        frmt = '.3f'
        for s,val in enumerate(np.linspace(sV,eV,Nv)):
            match sp:
                case 0:
                    clsPrm.attractivity = val
                    string = fr'$\lambda={val:{frmt}}$'
                case 1:
                    clsPrm.convincibility = val
                    string = fr'$\alpha={val:{frmt}}$'
                case 2:
                    clsPrm.zetaFraction = val
                    string = fr'$\zeta={val:{frmt}}$'

            self.KS[s] = KineticSimulation(clsPrm,clsReg)
            self.KS[s].sid = s+1
            self.KS[s].studiedPrmString = string

        self.fw = 8
        self.fh = 6
        self.figData = libF.FigData(clsPrm,'KS')

    # Simulation
    def MonteCarloSimulation(self):
        for s,KS in enumerate(self.KS):
            print(f'Starting simulation {s+1}')
            KS.MonteCarloSimulation()

    # Figures
    def SizeDistrFittingsFig(self):
        figData = self.figData
        fw = self.fw
```

```
977             fh = self.fh
978
979             nCol = 9 # Figures for each row
980             nRow = self.Nv
981             fig, ax = figData.SetFigs(nRow,nCol,size=(fw*nCol,fh*nRow))
982
983             for s,KS in enumerate(self.KS):
984                 KS.SizeDistrFittingsFig(
985                     ax=ax[s,:],
986                     idx=nCol*s+1,
987                     figData=figData
988                 )
989
990             figData.SaveFig('SizeDistributionFittings')
991
992         def AverageSizeFig(self):
993             figData = self.figData
994             fw = self.fw
995             fh = self.fh
996
997             nCol = 1 # Figures for each row
998             nRow = self.Nv
999             fig, ax = figData.SetFigs(nRow,nCol,size=(fw*nCol,fh*nRow))
1000
1001            for s,KS in enumerate(self.KS):
1002                KS.AverageSizeFig(
1003                    ax=ax[s],
1004                    idx=nCol*s+1,
1005                    figData=figData
1006                )
1007
1008            figData.SaveFig('AverageSize')
1009
1010        def SizeVsDegreeFig(self):
1011            figData = self.figData
1012            fw = self.fw
1013            fh = self.fh
1014
1015            nCol = 3 # Figures for each row
1016            nRow = self.Nv
1017            fig, ax = figData.SetFigs(nRow,nCol,size=(fw*nCol,fh*nRow))
1018
1019            for s,KS in enumerate(self.KS):
1020                KS.SizeVsDegreeFig(
1021                    ax=ax[s,:],
1022                    idx=nCol*s+1,
1023                    figData=figData
1024                )
1025
1026            figData.SaveFig('SizeVsDegree')
1027
1028        def SizeDistrEvolutionFig(self):
1029            figData = self.figData
1030            fw = self.fw
1031            fh = self.fh
1032
1033            nCol = 2 # Figures for each row
1034            nRow = self.Nv
1035            fig, ax = figData.SetFigs(nRow,nCol,size=(fw*nCol,fh*nRow))
1036
1037            for s,KS in enumerate(self.KS):
1038                KS.SizeDistrEvolutionFig(
1039                    ax=ax[s,:],
1040                    idx=nCol*s+1,
1041                    figData=figData
1042                )
1043
```

```
            figData.SaveFig('SizeDistributionEvolution')

        def SizeEvolutionsFig(self):
            figData = self.figData
            fw = self.fw
            fh = self.fh

            nCol = 2 # Figures for each row
            nRow = self.Nv
            fig, ax = figData.SetFigs(nRow,nCol,size=(fw*nCol,fh*nRow))

            for s,KS in enumerate(self.KS):
                KS.SizeEvolutionsFig(
                    ax=ax[s,:],
                    idx=nCol*s+1,
                    figData=figData
                )

            figData.SaveFig('SizeEvolutions')

        def ShowFig(self):
            from matplotlib.pyplot import show
            show()


    ### Auxiliary functions ###

    @dataclass(frozen=True)
    class ShMSpec:
        name: str
        shape: tuple[int,...]
        dtype: str

    def BuildShM(
        array=None,
        Ni=None,
        dtype=None
    ):
        if array is not None:
            a = np.ascontiguousarray(array)
        else:
            a = np.zeros(Ni,dtype=dtype)

        shm = shared_memory.SharedMemory(
            create=True,
            size=a.nbytes
        )

        view = np.ndarray(
            a.shape,
            dtype=a.dtype,
            buffer=shm.buf
        )
        view[:] = a

        spec = ShMSpec(
            name=shm.name,
            shape=a.shape,
            dtype=a.dtype.str
        )

        return spec, shm

    def LoadShM(spec):
        shm = shared_memory.SharedMemory(name=spec.name)
        arr = np.ndarray(
            spec.shape,
```

```
            dtype=np.dtype(spec.dtype),
            buffer=shm.buf
        )
        return shm, arr

    def InitializeMCSWorker(
        shmPrm,
        gui=False
    ):
        for key in workersShM['parameters'].keys():
            shm, arr = LoadShM(shmPrm[key])
            workersShM['handles'].append(shm)
            workersShM['parameters'][key] = arr

        if gui:
            for key in workersShM['gui'].keys():
                shm, arr = LoadShM(shmPrm[key])
                workersShM['handles'].append(shm)
                workersShM['gui'][key] = arr

    def MonteCarloAlgorithm(
        p,Nc,Ns,p0,typ,
        l,a,s,z,il,gui
    ):
        Mdt = workersShM['parameters']["Mdt"]
        wOdt = workersShM['parameters']["wOdt"]
        wI = workersShM['parameters']["wI"]
        di = workersShM['parameters']["di"]
        idi = workersShM['parameters']["idi"]
        ns = workersShM['parameters']["ns"]

        # rng = np.random.default_rng() # Even though it's recommended
        ↪  by Numpy it is not efficiently implemented in Numba, hence
        ↪  it halves the iterations per second if used

        # Uniform initial state for all vertices
        vrtState = np.full((Nc,typ),p0,dtype=np.float64)
        snapshots = np.full((Nc,Ns+1,typ),p0,dtype=np.float64)

        P = np.arange(Nc,dtype=np.int32)

        nk = ns[1]
        hNc = Nc//2 # Nc half
        nsid = 1

        print(f'Starting iteration {p+1}')

        if gui: # If ProgressGUI is required
            progress = workersShM['gui']['progress']
            elapsed = workersShM['gui']['elapsed']
            done = workersShM['gui']['done']

            EvolveState(
                vrtState,P,
                Nc,hNc,nk,
                Mdt,wOdt,wI,
                di,idi,il,
                l,a,s,z
            ) # Warm-up iteration to avoid polluting the initial time t0
            snapshots[:,nsid,0] = vrtState[:,0]
            snapshots[:,nsid,1] = vrtState[:,1]
            nsid += 1

            t0 = time.perf_counter()
            for st in range(Ns-1): # step
                EvolveState(
                    vrtState,P,
```

```
                    Nc,hNc,nk,
                    Mdt,wOdt,wI,
                    di,idi,il,
                    l,a,s,z
                )
                snapshots[:,nsid,0] = vrtState[:,0]
                snapshots[:,nsid,1] = vrtState[:,1]
                nsid += 1

                progress[p] = (st+2)*nk # ns[nsid]
                elapsed[p] = time.perf_counter()-t0

            progress[p] = Ns*nk # ns[-1]
            elapsed[p] = time.perf_counter()-t0
            done[p] = True

            return vrtState, snapshots

        else:
            EvolveState(
                vrtState,P,
                Nc,hNc,nk,
                Mdt,wOdt,wI,
                di,idi,il,
                l,a,s,z
            ) # Warm-up iteration to avoid polluting the initial time t0
            snapshots[:,nsid,0] = vrtState[:,0]
            snapshots[:,nsid,1] = vrtState[:,1]
            nsid += 1

            t0 = time.perf_counter()
            for _ in range(Ns-1):
                EvolveState(
                    vrtState,P,
                    Nc,hNc,nk,
                    Mdt,wOdt,wI,
                    di,idi,il,
                    l,a,s,z
                )
                snapshots[:,nsid,0] = vrtState[:,0]
                snapshots[:,nsid,1] = vrtState[:,1]
                nsid += 1

            return vrtState, snapshots

    @njit(cache=True)
    def EvolveState(
        cs,P,Nc,
        hNc,nk,Mdt,
        wOdt,wI,
        di,idi,il,
        l,a,s,z
    ):
        for _ in range(nk):
            FYDInPlaceShuffle(P,Nc)
            # P = np.random.permutation(Nc)
            # pi = P[:hNc]; pr = P[hNc:]

            for i in range(hNc):
                ii = P[i]; ir = P[i+hNc]

                # t = 0
                p = Mdt[ii,ir]
                if p > 0:
                    theta = np.random.random() < p

                    if theta == 1:
```

```
                    si = cs[ii,0]; sr = cs[ir,0]

                    e = NonLinearEmigration(si,idi[ii],sr,di[ir],i⌋
                    ↪  l,l,a,z)
                    ga = StochasticFluctuations(s,e) if s>0 else 0

                    cs[ii,0] = si*(1-e+ga)
                    cs[ir,0] = sr+si*e

            # t = 1
            p = wOdt[ii]*wI[ir]
            # Apparently it's more efficient to access two values
            ↪  from two separate vectors and to multiply them,
            ↪  than it is to access the same pre-computed value
            ↪  from a matrix; the same holds for the product
            ↪  «di[ir]*idi[ii]» inside «NonLinearEmigration()»
            # However, in order for this property to be valid the
            ↪  matrix has to have an underlying more simple
            ↪  structure which in both cases is rank 1
            # In other words this trick does not work with the
            ↪  adjacency matrix «Mdt[ii,ir]» which cannot be
            ↪  computed from simpler elements
            theta = np.random.random() < p

            if theta == 1:
                si = cs[ii,1]; sr = cs[ir,1]

                e = NonLinearEmigration(si,idi[ii],sr,di[ir],il,l,⌋
                ↪  a,z)
                ga = StochasticFluctuations(s,e) if s>0 else 0

                cs[ii,1] = si*(1-e+ga)
                cs[ir,1] = sr+si*e

            # In the exact case it's reasonalbe to check whether p
            ↪  is actually positive before evaluating the
            ↪  Bernoulli distribution with «np.random.random()<p»
            ↪  since A can has zero components
            # However its approximations Ap does not have zero
            ↪  components by definition (it's a complete network),
            ↪  hence that check becomes useless

            # p = 1 if p>1 else (0 if p<0 else p)
            # theta = np.random.binomial(1,p)

@njit(cache=True)
def FYDInPlaceShuffle(v,n):
    for i in range(n-1,0,-1):
        j = np.random.randint(0,i+1)
        tmp = v[i]
        v[i] = v[j]
        v[j] = tmp

        # In randint i+1 is necessary to include the i-th index

    # Fisher-Yates-Durstenfeld [in-place] shuffle
    # https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle#T⌋
    ↪  he_modern_algorithm

@njit(cache=True)
def NonLinearEmigration(
        si,idii, # Interacting city size
        sr,dir,  # Receiving city size
        il,l,a,z
    ):
    if si <= 0:
```

```
            return 0

        if il == 0:
            rs = sr/si
            efl = l*(rs**a)/(1+rs**a)

            if z == 0:
                return efl
            else:
                if sr <= 0:
                    return 0
                else:
                    irs = 1/rs
                    efs = l*(irs**a)/(1+irs**a)
                    return (1-z)*efl+z*efs

        elif il == 1:
            rsk = (sr/si)*dir*idii
            efl = l*(rsk/a)/(1+rsk/a)

            if z == 0:
                return efl
            else:
                if sr <= 0:
                    return 0
                else:
                    irsk = 1/rsk
                    efs = l*(irsk/a)/(1+irsk/a)
                    return (1-z)*efl+z*efs

        elif il == 2:
            rsk = (sr/si)*dir*idii
            efl = l*(rsk**a)/(1+rsk**a)

            if z == 0:
                return efl
            else:
                if sr <= 0:
                    return 0
                else:
                    irsk = 1/rsk
                    efs = l*(irsk**a)/(1+irsk**a)
                    return (1-z)*efl+z*efs

        else:
            rsk = (sr/si)*dir*idii
            efl = l*(rsk/(1+rsk))**a

            if z == 0:
                return efl
            else:
                if sr <= 0:
                    return 0
                else:
                    irsk = 1/rsk
                    efs = l*(irsk/(1+irsk))**a
                    return (1-z)*efl+z*efs

            # rsk = (sr/si)*(dr*idi)  # Relative population*degree ratio
            # irsk = (si/sr)*(di*idr) # Inverse relative
            ↪  population*degree ratio
            # efl                     # Actual emigration rate for the
            ↪  lumping fraction
            # efs                     # Actual emigration rate for the
            ↪  separation fraction
```

```
1356        # Numba does not officially support the match structure, hence,
            ↪  even if it appears to work, it's better to use an
            ↪  «if/elif/else» one for the sake of performance and stability
1357
1358    @njit(cache=True)
1359    def StochasticFluctuations(sigma,E):
1360        alpha = ((1-E)**2)/(sigma**2)
1361        beta  = (sigma**2)/(1-E)
1362
1363        # if alpha<=1:
1364        #     raise ValueError("α must be >1 to have a non-degenerate
            ↪  gamma distribution, and thus always admissible
            ↪  fluctuations")
1365
1366        ga = np.random.gamma(alpha,beta) # Initial sampling
1367
1368        return ga+E-1 # Final left translation
```

ELENCO DELLE FIGURE

ELENCO DELLE TABELLE

ELENCO DEI LISTATI

ELENCO DEGLI ALGORITMI

BIBLIOGRAFIA

[1] Felix Auerbach. «Das Gesetz der Bevölkerungskonzentration [The law of population concentration]». In: *Petermanns Geographische Mitteilungen* 59 (1913), pp. 74–76.

[2] Albert-László Barabási, Réka Albert & Hawoong Jeong. «Mean-field theory for scale-free random networks». In: *Physica A: Statistical Mechanics and its Applications* 272.1-2 (1999), pp. 173–187.

[3] Marc Barthélemy. «Spatial networks». In: *Physics Reports* 499.1 (2011), pp. 1–101. ISSN: 0370-1573. DOI: https://doi.org/10.1016/j.physrep.2010.11.002. URL: https://www.sciencedirect.com/science/article/pii/S037015731000308X.

[4] Marcus Berliant & Axel H Watanabe. «A scale-free transportation network explains the city-size distribution». In: *Quantitative Economics* 9.3 (2018), pp. 1419–1451.

[5] Anna D Broido & Aaron Clauset. «Scale-free networks are rare». In: *Nature communications* 10.1 (2019), p. 1017.

[6] Andrea De Montis et al. «The structure of interurban traffic: a weighted network analysis». In: *Environment and Planning B: Planning and Design* 34.5 (2007), pp. 905–924.

[7] Rick Durrett. *Probability: theory and examples*. Vol. 49. Cambridge university press, 2019.

[8] Jan Eeckhout. «Gibrat's law for (all) cities». In: *American Economic Review* 94.5 (2004), pp. 1429–1451.

[9] Stefano Gualandi & Giuseppe Toscani. «Human behavior and lognormal distribution. A kinetic description». In: *Mathematical Models and Methods in Applied Sciences* 29.04 (2019), pp. 717–753.

[10] Stefano Gualandi & Giuseppe Toscani. «Size distribution of cities: A kinetic explanation». In: *Physica A: Statistical Mechanics and its Applications* 524 (2019), pp. 221–234.

[11] International Standards Organisation. *ISO 80000-3 Quantities and units—Part 3: Space and time*.

[12] ISTAT. *Basi territoriali e variabili censuarie*. Riferimento specifico al censimento della popolazione e delle abitazioni del 1991. 2024. URL: https://www.istat.it/notizia/basi-territoriali-e-variabili-censuarie/ (visitato il giorno 12/02/2026).

[13] ISTAT. *Confini delle unità amministrative a fini statistici al 1° gennaio 2018. Dati storici (1991)*. Riferimento specifico ai dati del 1991. 2025. URL: https://www.istat.it/notizia/confini-delle-unita-amministrative-a-fini-statistici-al-1-gennaio-2018-2/ (visitato il giorno 12/02/2026).

[14] ISTAT. *Matrici di contiguità, distanza e pendolarismo. Dati storici (1991)*. Riferimento specifico ai dati della matrice di pendolarismo del 1991. 2025. URL: https://www.istat.it/notizia/matrici-di-contiguita-distanza-e-pendolarismo/ (visitato il giorno 12/01/2026).

[15] Nadia Loy & Andrea Tosin. «A viral load-based model for epidemic spread on spatial networks». In: *arXiv preprint arXiv:2104.12107* (2021). URL: https://arxiv.org/abs/2104.12107.

[16] Nadia Loy & Andrea Tosin. «Essentials of the kinetic theory of multi-agent systems». In: *arXiv preprint arXiv:2503.11554* (2025). URL: https://arxiv.org/abs/2503.11554.

[17] Marco Nurisso, Matteo Raviola & Andrea Tosin. «Network-based kinetic models: Emergence of a statistical description of the graph topology». In: *European Journal of Applied Mathematics* (2024), pp. 1–22. DOI: 10.1017/S0956792524000020.

[18] Lorenzo Pareschi & Giuseppe Toscani. *Interacting multiagent systems: kinetic equations and Monte Carlo methods*. OUP Oxford, 2013.

[19] Valerio Taralli. *PoliTo_Tesi-Magistrale_332689*. https://github.com/Svistaio/PoliTo_Tesi-Magistrale_332689/tree/master. Repositorio GitHub del codice usato nell'elaborato corrente. 2026.

[20] George Kingsley Zipf. *Human behavior and the principle of least effort: An introduction to human ecology*. Ravenio books, 2016.