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

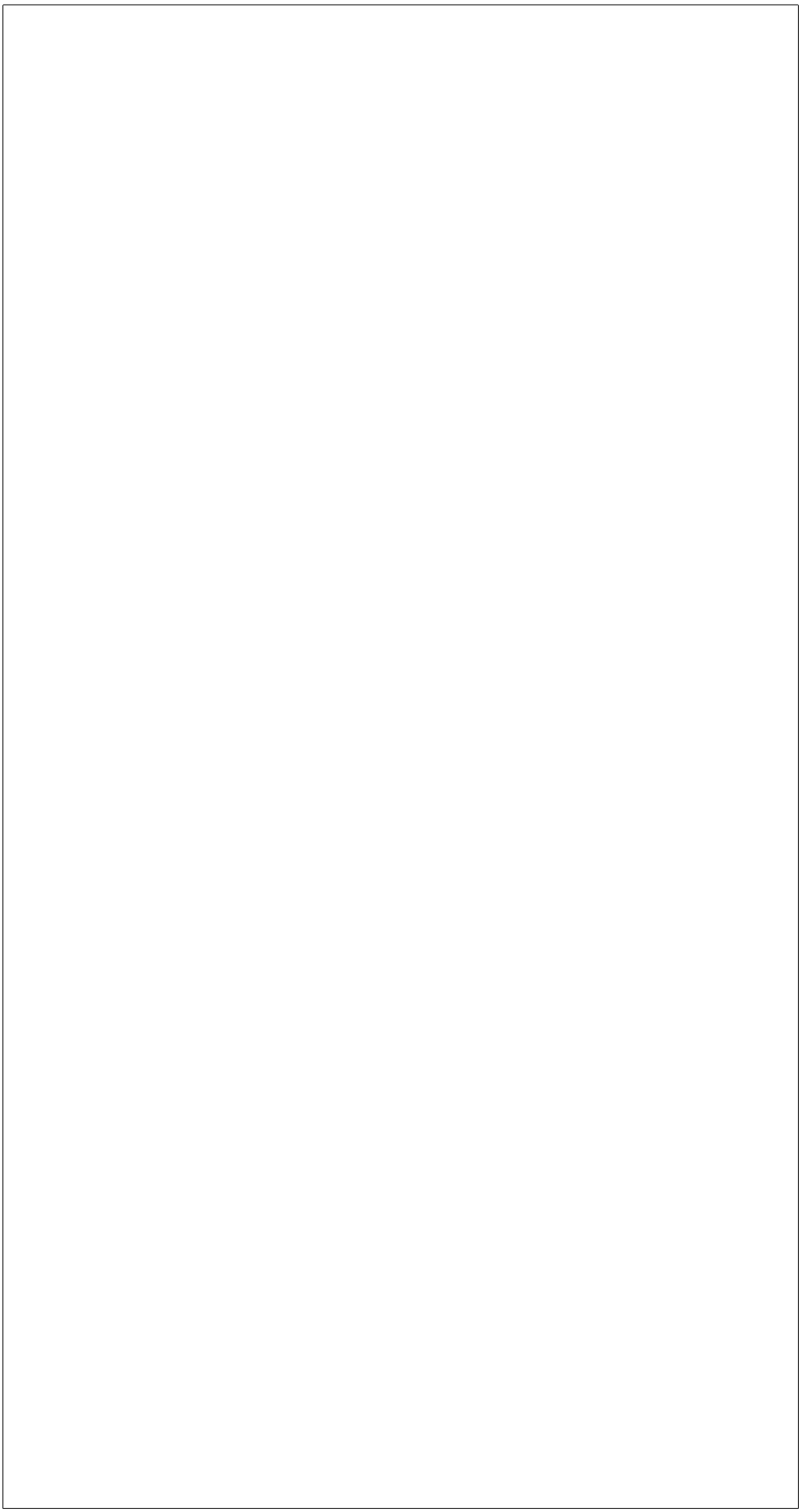
firma del relatore

.....

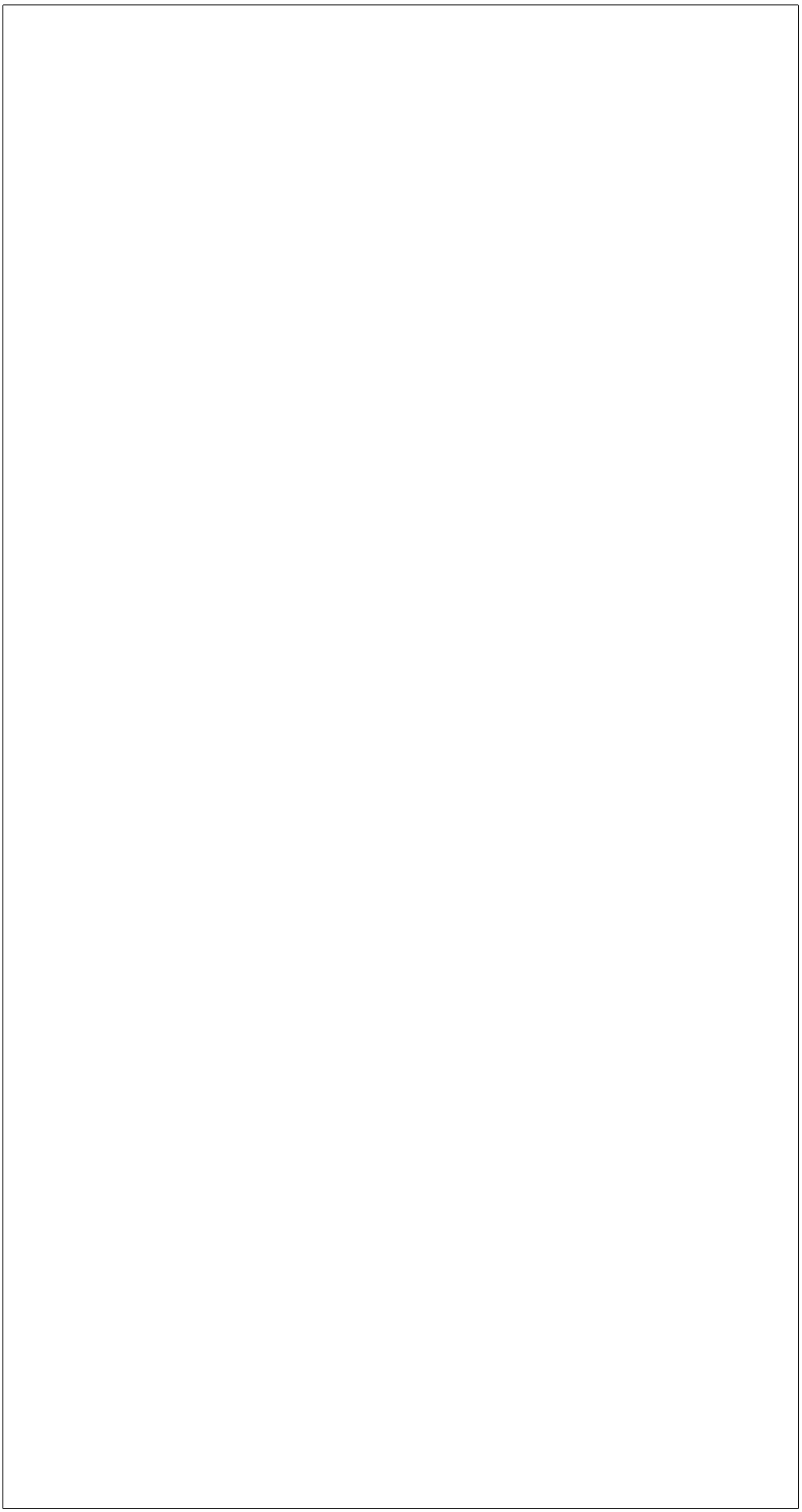
firma del candidato

.....

Anno Accademico 2025-2026



Ai miei genitori,
Elisabetta e *Marco*

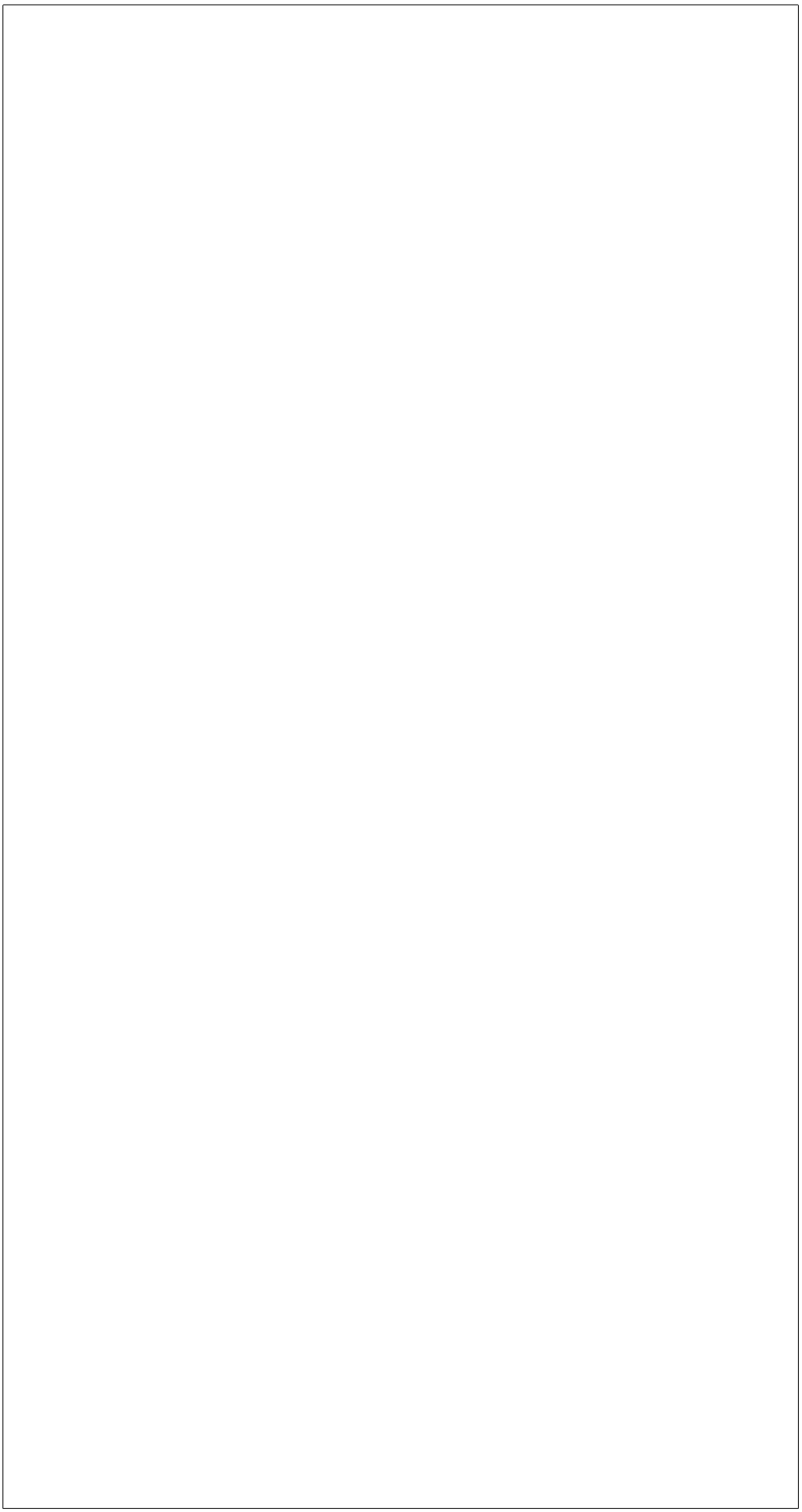


SOMMARIO

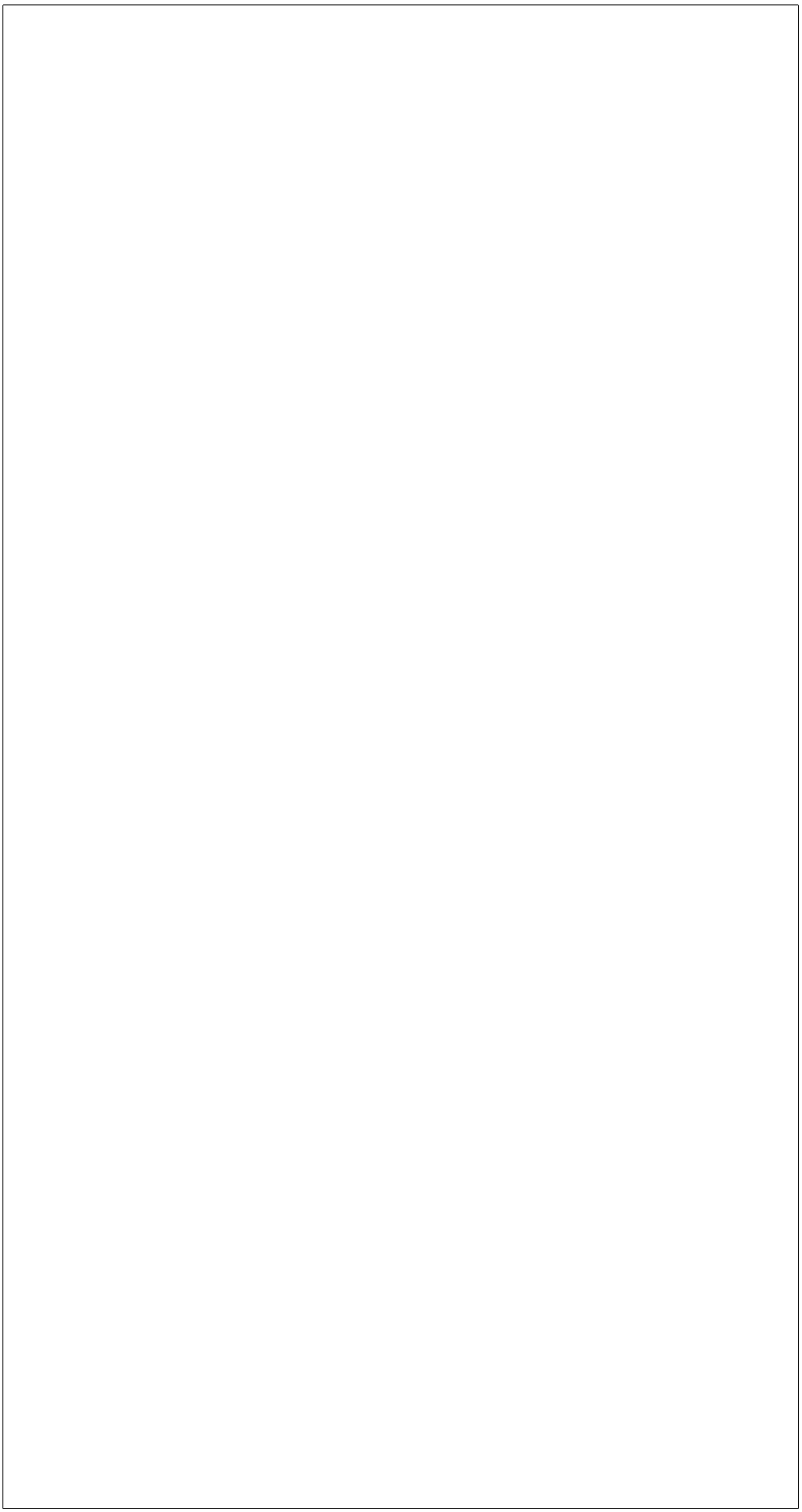
La corrente tesi studia la distribuzione della popolazione mediante la teoria cinetica dei sistemi multiagente su reti spaziali, interpretando le città come agenti/vertici e rappresentando le connessioni interurbane come lati. Si propongono varie regole d'emigrazione per regolare le interazioni tra agenti, 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 19-esimo secolo notò una caratteristica poi formalizzata successivamente da Zipf[1] [19] quasi cinquantanni 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 10 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 [19, 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 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

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

per una certa costante $c \in \mathbb{R}_+$ e con $R(s)$ uguale alla funzione di ripartizione complementare di S: essa rappresenta matematicamente 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).

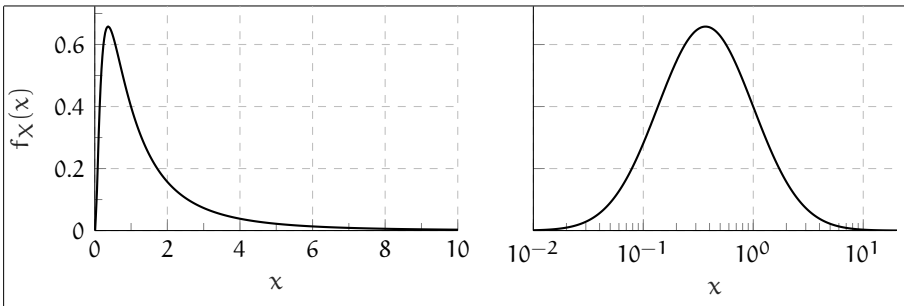
Piú recentemente [8, 9] si è scoperto che al di fuori della coda la $f_S(s)$ è ben fittata dalla distribuzione lognormale con densità di probabilità

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

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

1 In realtà, come Zipf stesso ammette [19, 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.



(a) Scala lineare. (b) Scala semilogaritmica.

Figura 1.1: Funzione Lognormale$(0,1)$.

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

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

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 si potrà poi meditare sulle implicazioni della legge d'interazione riguardo al fenomeno dell'emigrazione, che è alla base delle interazioni tra città.

Si sottolinea che qui le città verranno invece considerate come agenti, caratterizzati dalla loro popolazione, che interagiscono attraverso la struttura sottostante di un grafo [spaziale]; tale descrizione, salvo il grafo, 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 le teorie cinetiche, la distribuzione della città e i grafi, nonostante la rappresentazione di queste su un grafo sia del tutto naturale; piú nel dettaglio gli articoli in letteratura

- ◇ 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].

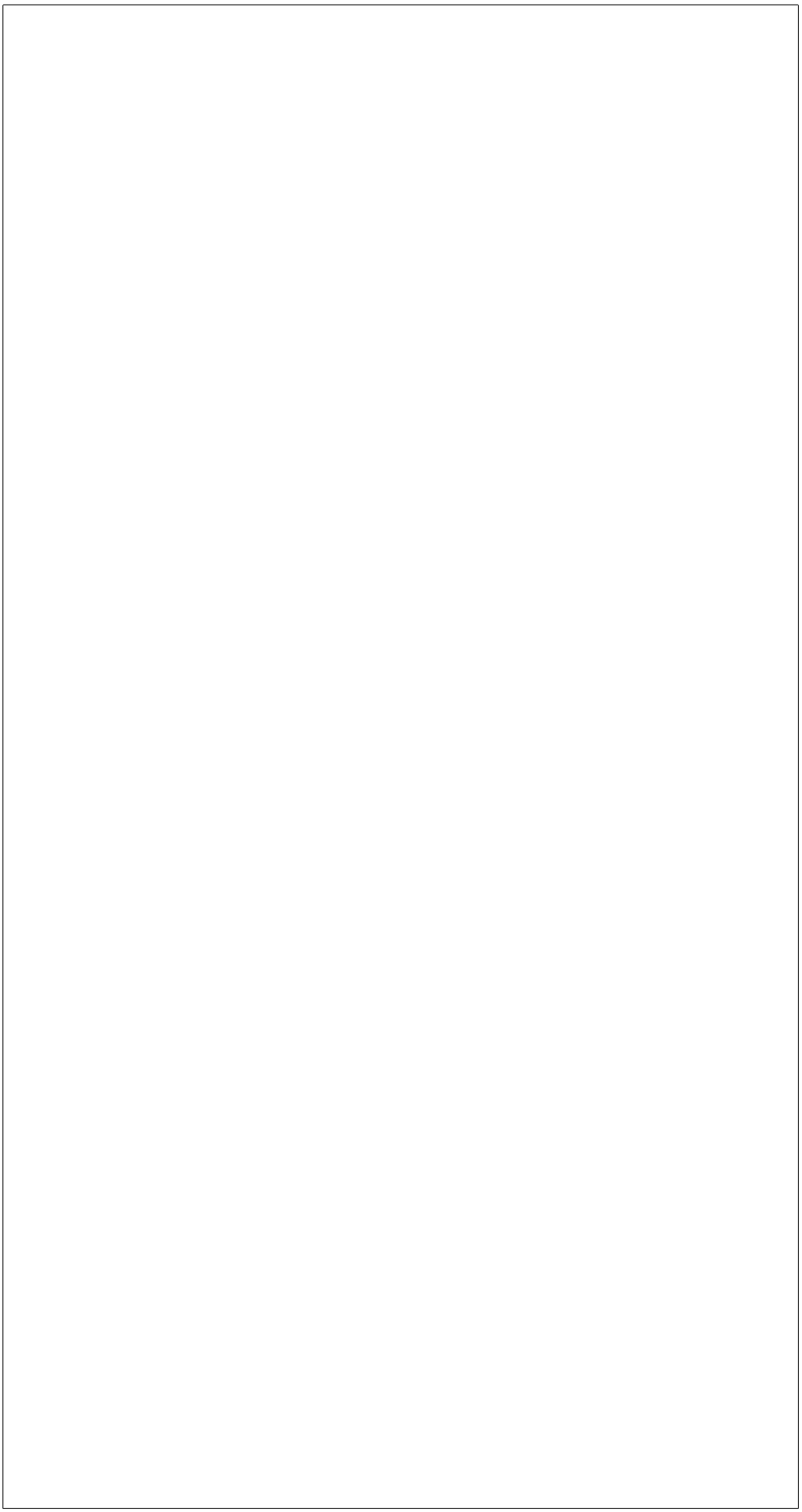
Questa lacuna è probabilmente attribuibile al fatto che gli sviluppi della teoria dei grafi applicata alla TCSMA sono stati piuttosto recenti e prin-

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*.

cipalmente concentrati sulla prospettiva nodo-luogo anziché nodo-agente. Dunque l'aspetto piú innovativo di questo lavoro è mostrare che la prospettiva nodo-agente è altrettanto valida quanto quella piú comune e recente del nodo-luogo; per il resto questa tesi dev'essere considerata come un'esplorazione formale applicativa con pochi approfondimenti analitici.

Lo scritto sarà 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 si illustreranno i principali risultati concludendo con delle note finali e potenziali sviluppi futuri.

Per il lettore interessato è anche presente un'appendice ove il codice di Python dell'implementazione è spiegato a grandi linee.



2 NOTE DI TEORIA DEI GRAFI

In questo capitolo sono prima introdotti alcuni oggetti di base della teoria dei grafi, 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 formato dalla coppia $\mathcal{G} = (\mathcal{I}, \mathcal{E})$, ove $\mathcal{I}$ è l'insieme degl'indici dei nodi mentre $\mathcal{I}$ è l'insieme dei lati, 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 dato $(i,j) \in \mathcal{E}$ allora $(j,i) \in \mathcal{E}$ e la direzione è trascurabile; altrimenti è detto diretto.

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

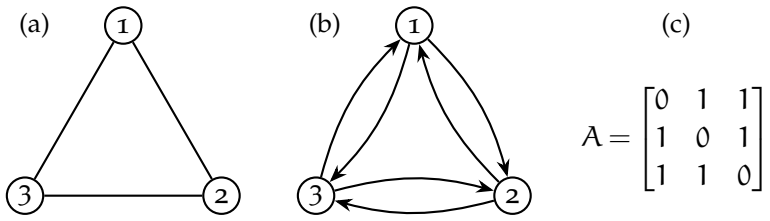


Figura 2.1: Esempio di un grafo indiretto, della sua equivalente forma diretta [simmetrica] e della loro [identica] matrice d'adiacenza.

Definizione 2.3 (Matrice d'adiacenza)**.** Sia $N \equiv |\mathcal{I}| \in \mathbb{N}_+$ la cardinalità dell'insieme degl'indici, 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 $\mathbf{A} \in \mathbb{R}^{N \times N}$:

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

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 $N \in \mathbb{N}_+$, allora s'indica $\mathbf{M}^\top$ 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 $N \in \mathbb{N}_+$, allora

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

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 queste 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 possibili tra le città

Difatti 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 la rete di trasporto di tram, di bus e della 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 nella Cap. 1, è chiaro che per predire la distribuzione della popolazione tra città valgono le seguenti osservazioni:

1. ogni rappresentazione intraurbana va scartata perché sono troppo fini, 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 del 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 qui la risposta è immediata dopo due osservazioni:

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'opposta: le potenziali interazioni sono simmetriche.

Dunque il grafo in questione è diretto ma con struttura indiretta (Def. 2.2), ovvero la sua matrice d'adiacenza $\mathbf{A}$ è simmetrica.

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

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*; ovviamente si tratta di una forte semplificazione dato che nella realtà la topologia delle connessioni interurbane si è 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 servirà tenerlo a mente quando si definiranno alcune regole d'interazione nel Cap. 4. Perdipiú, contrariamente a quanto si possa pensare di primo acchito, tale grafo non è a invarianza di scala [2] a causa del tipo di grafo [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 proprio dalla natura spaziale del grafo.

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 comuni considerati sono tutt'i quell'italiani (8100) nel 1991, quindi è necessario estrapolare da essi le matrici di pendolarismo delle singole regioni e parallelamente quella complessiva dell'Italia. Si possono definire due tipi di matrici:

1. quella d'adiacenza unitaria in cui un lato esiste se almeno un pendolare (cioè il «[n]umero di persone» nella Tab. 2.1) si muove tra i due comuni e
2. quella d'adiacenza pesata che è analoga alla precedente ma con peso $q_{i,j}$ uguale alla somma totale dei pendolari tra due comuni.

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

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

Tabella 2.1: Formattazione di una sola riga nei dati ISTAT[1]; le linee barrate corrispondono a dati trascurati.

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», «246».

In entrambi i casi i dati relativi [a quelli che sono potenzialmente refusi] sono stati trascurati nella corrente tesi; il primo errore è già di per sé molto raro, mentre il secondo è eccessivamente ambiguo da risolvere come si può notare dalla 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 tutt'i comuni collo stesso codice comunale, quello che minimizza la distanza geografica (ossia euleriana) siccome le coordinate dei punti rappresentativi di ogni comune sono disponibili nell'ISTAT [13]. In ogni caso il numero cosí esiguo d'eccezioni rendono una tale "correzione" del tutto innecessaria, oltre che irrilevante come il Cap. 4 mostrerà.

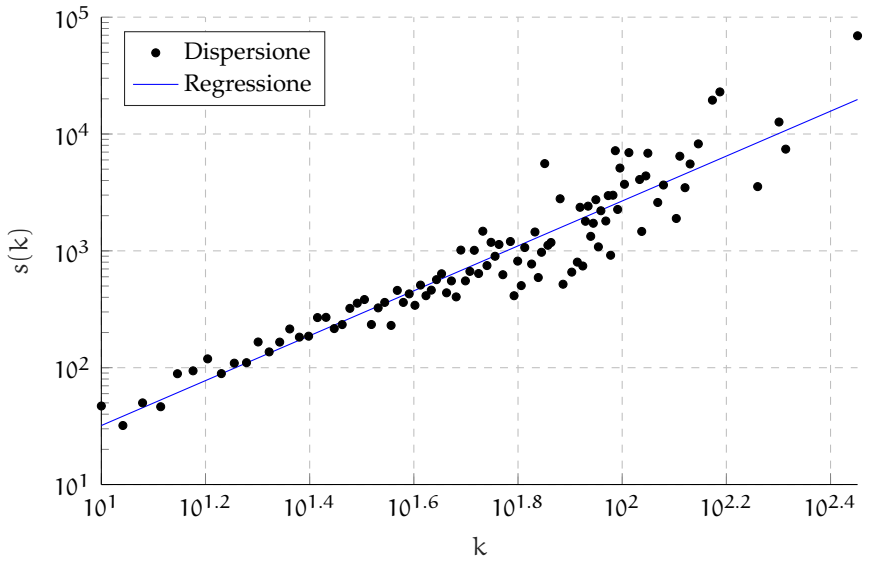


Figura 2.2: Forza contro grado per la Sardegna.

Infine nelle Figg. 2.2 e 2.3 sono stati anche riprodotti i risultati di [6] colle seguenti osservazioni:

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. 2.3c 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 rapppresentato nella Fig. 2.3g ha chiaramente una tendenza decrescente, anche se molto lenta, ragion 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: ciò si può giustificare molto probabilmente come refuso da parte di [6] poiché i collegamenti «Cagliari-Sassari» e «Sassari-Olbia» sono molto distanti geograficamente[2], per cui è ragionevole che i flussi siano bassi.

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

(a) Pesi maggiori correnti.

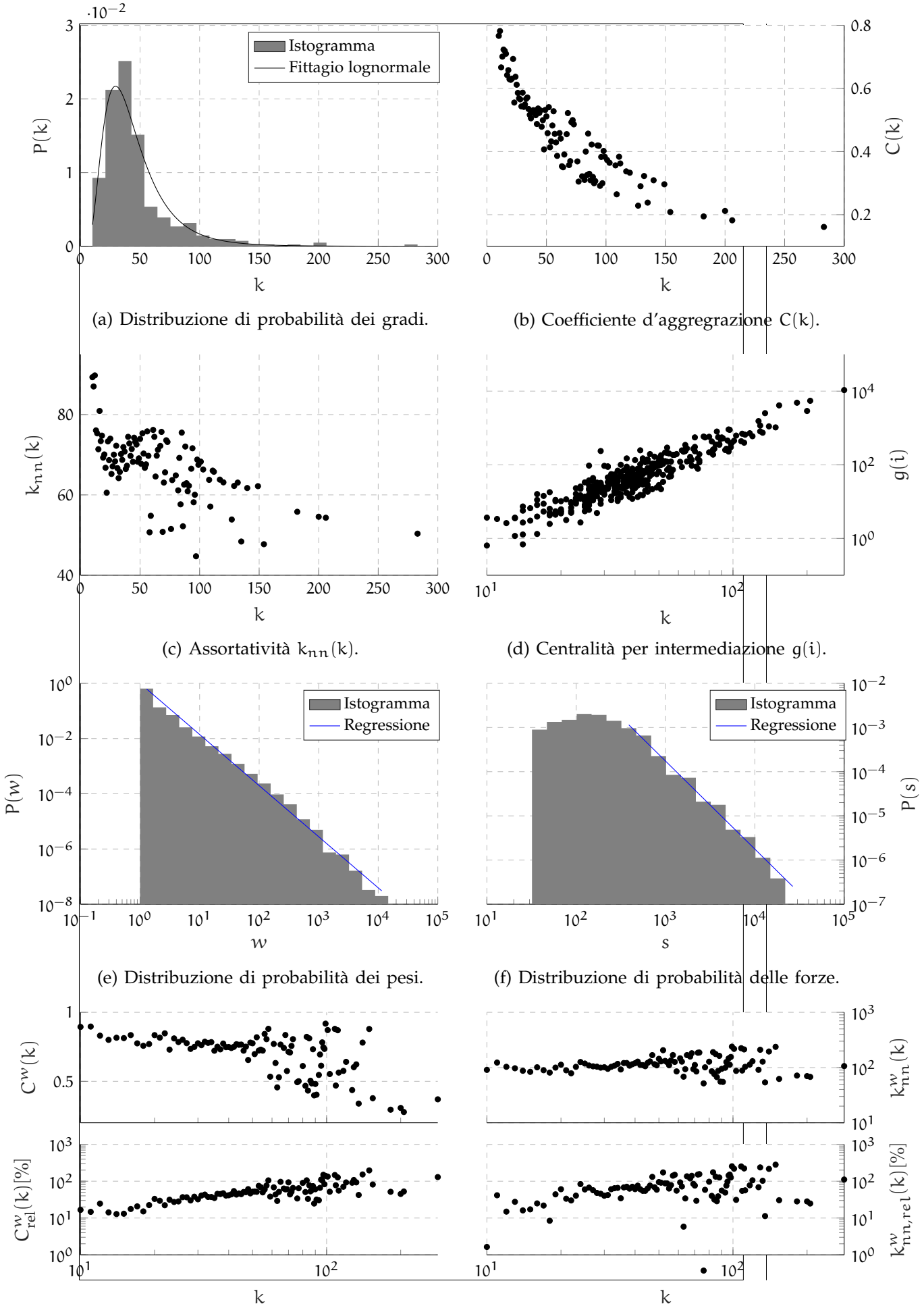
Connessione	Peso
Cagliari-Sassari	13953
Sassari-Olbia	7246
Cagliari-Assemini	4226
Porto Torres-Sassari	3993
Cagliari-Capoterra	3731

(b) Pesi maggiori di [6]

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

2 Il primo equivale a percorrere l'intera lunghezza dell'isola ogni giorno.

Queste discrepanze sono perché sono lievi e, per quanto concerne questo elaborato, irrilevanti. Inoltre, l'unico risultato di rilievo è la correlazione positiva tra la forza e il grado nella Fig. 2.2.



(a) Distribuzione di probabilità dei gradi.

(b) Coefficiente d'aggregrazione $C(k)$.

(c) Assortatività $k_{nn}(k)$.

(d) Centralità per intermediazione $g(i)$.

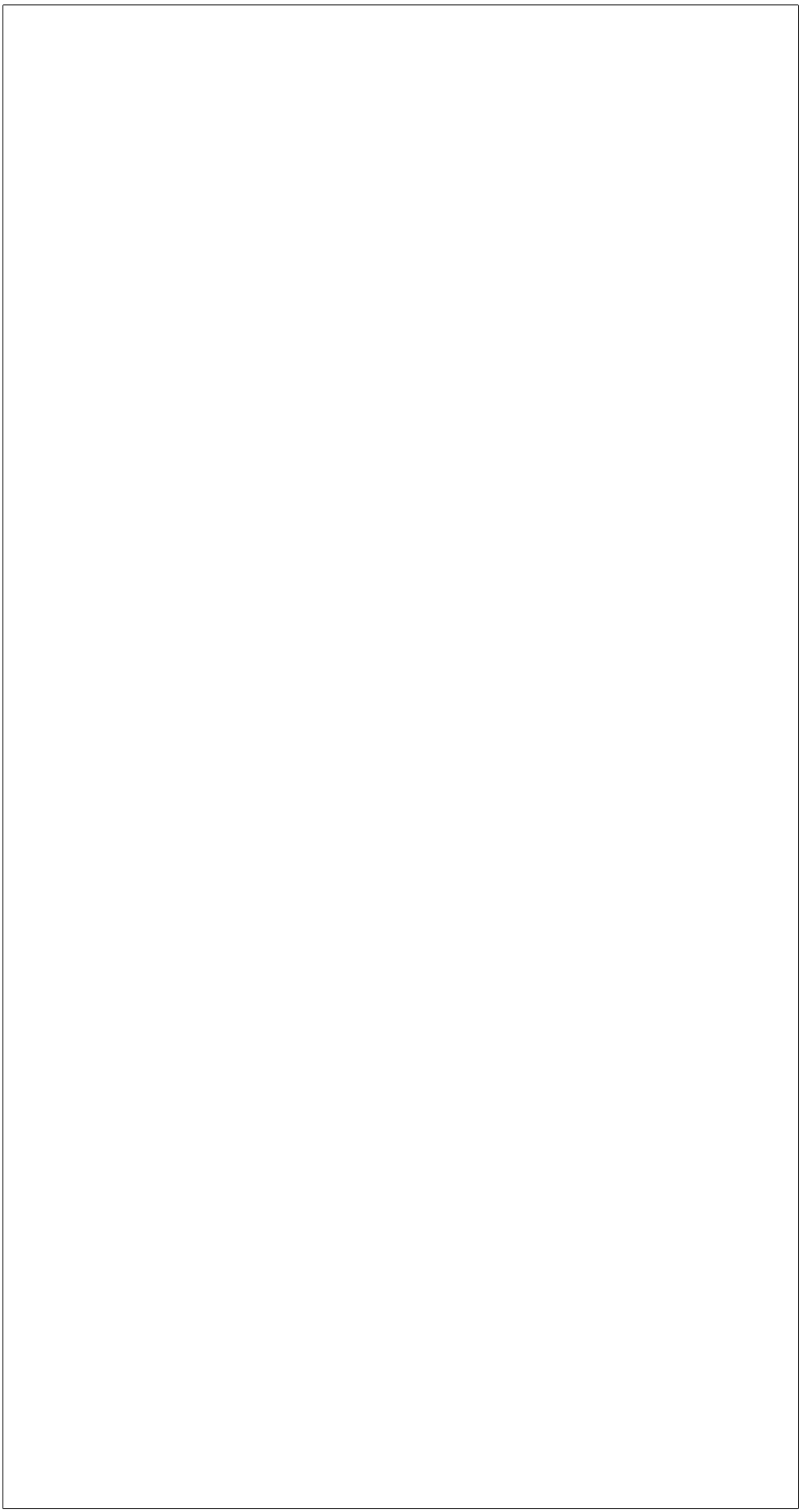
(e) Distribuzione di probabilità dei pesi.

(f) Distribuzione di probabilità delle forze.

(g) Coefficiente [relativo] d'aggregazione pesato $C^w(k)$.

(h) Assortatività [relativa] pesata $k^w_{nn}(k)$.

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



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; successivamente si descrivono le equazioni cinetiche nel caso corrente; infine 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 senza spiegare un intero corso; in ogni caso si rimanda a [7] per gl'interessati.

Definizione 3.1 (Semiretta intera/reale positiva)**.** 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; quest'ultimo definisce a sua volta uno spazio di probabilità $(\Omega, \mathcal{F}, \mathbb{P})$ composto da una σ-algebra $\mathcal{F}$ 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 $\mathbb{P}_X$ di 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 le quali $f_X(x)\mathrm{d}\nu$ va inteso come $f_X(\mathrm{d}\nu)$.

Definizione 3.4 (Densità congiunta di probabilità di $\mathbf{X}$)**.** 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 $\mathbb{P}_{\mathbf{X}}$ di $\mathbf{X}$:

$$\begin{aligned} \mathbb{P}_{\mathbf{X}}(A) = \mathbb{P}(\mathbf{X} \in A) &= \int_A f_{\mathbf{X}}(\mathbf{x}) d\mathbf{x} \\ &= \int_A f_{\mathbf{X}}(\mathbf{x}) dx_1 dx_2 \cdots dx_n \quad \forall A \subseteq \mathbb{R}^n \text{ misurabile.} \end{aligned}$$

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:

$$f_{X_i}(x_i) = \int_{\mathbb{R}^{n-1}} f_{\mathbf{X}}(\mathbf{x}) dx_1 dx_2 \cdots dx_{i-1} dx_{i+1} \cdots dx_n \quad \forall i \in \{1, 2, \cdots, n\}; \tag{3.1}$$

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.

Inoltre, qualora il contesto non dia ambiguità, si sottintendono le variabili o i vettori aleatori cui fanno riferimento le relative densità: dunque f è la densità di X, $\mathbf{X}$ e X_i, a seconda del caso; in alternativa si possono usare anche forme piú leggere, come f_i per f_{X_i}, se necessario.

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 allora 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

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

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

$$\mathbb{E}[\varphi(\mathbf{X})] \equiv \int_{\mathbb{R}^n} \varphi(\mathbf{x}) f_{\mathbf{X}}(\mathbf{x}) d\mathbf{x} = \int_{\mathbb{R}^n} \varphi(\mathbf{x}) f_{\mathbf{X}}(\mathbf{x}) dx_1 dx_2 \cdots dx_n.$$

Definizione 3.7 (Valore atteso condizionato)**.** Sia $X \in \mathbb{R}$ una variabile aleatoria assolutamente continua a media finita $\mathbb{E}[X] \infty$ e sia $\mathcal{G} \subseteq \mathcal{F}$ una sotto-σ-algebra, allora l'attesa di X condizionata a $\mathcal{G}$ è una variabile aleatoria Y 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; in effetti 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 caso misti con X scalare e Y vettoriale.

Osservazione 3.3. Presa una variabile aleatoria $Y \in \mathbb{R}$, nella precedente definizione $\mathcal{G}$ può essere anche la σ-algebra definita da Y:

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

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

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

Proposizione 3.1. *Dati*

1. $n, h, k \in \mathbb{N}_+$ *tali che* $n = h + k$;
2. *un vettore aleatorio* $\mathbf{Z} = (\mathbf{X}, \mathbf{Y}) \in \mathbb{R}^n$ *assolutamente continuo costituito da* $\mathbf{X} \in \mathbb{R}^h$ *e* $\mathbf{Y} \in \mathbb{R}^k$ *con densità congiunta* $f_{\mathbf{Z}}(\mathbf{x}, \mathbf{y})$ *e marginali* $f_{\mathbf{X}}$ *e* $f_{\mathbf{Y}}$;
3. *una funzione* $\varphi \colon \mathbb{R}^n \to \mathbb{R}$ *misurabile e tale che*

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

allora si può definire $h \colon \mathbb{R}^k \to \mathbb{R}$ *tramite la seguente equazione*

$$\int_{\mathbb{R}^h} h(\mathbf{y}) f_{\mathbf{Z}}(\mathbf{x}, \mathbf{y}) d\mathbf{x} = \int_{\mathbb{R}^h} \varphi(\mathbf{x}, \mathbf{y}) f_{\mathbf{Z}}(\mathbf{x}, \mathbf{y}) d\mathbf{x} \quad \forall \mathbf{y} \in \mathbb{R}^k, \tag{3.2}$$

che soddisfà $h(\mathbf{Y}) \in \mathbb{E}[\varphi(\mathbf{X}, \mathbf{Y})|\mathbf{Y}]$.

Dimostrazione. Per la AC1 è sufficiente esplicitare la forma di $\sigma(h(\mathbf{Y}))$:

$$\sigma(h(\mathbf{Y})) \equiv \sigma(\mathbf{Y}^{-1}(h^{-1}(B)) \mid B \in \mathcal{B}(\mathbb{R})) \subseteq \sigma(\mathbf{Y}^{-1}(B) \mid B \in \mathcal{B}(\mathbb{R}^h)),$$

a parole si considera la controimmagine tramite $\mathbf{Y}$ dei borelliani filtrati da h, ragion per cui la σ-algebra non potrà che essere contenuta in quella non filtrata.

Per la AC2 si verifica la condizione $\forall A \in \sigma(\mathbf{Y})$ avvalendosi della caratterizzazione di $h(\mathbf{Y})$ tramite (3.2):

$$\begin{aligned}
\int_A h(\mathbf{Y}) d\mathbb{P} &= \int_B h(\mathbf{y}) f_{\mathbf{Y}}(\mathbf{y}) d\mathbf{y} \overset{(3.1)}{=} \int_B h(\mathbf{y}) \int_{\mathbb{R}^h} f_{\mathbf{Z}}(\mathbf{x}, \mathbf{y}) d\mathbf{x} d\mathbf{y} \\
&= \int_B \int_{\mathbb{R}^h} h(\mathbf{y}) f_{\mathbf{Z}}(\mathbf{x}, \mathbf{y}) d\mathbf{x} d\mathbf{y} \overset{(3.2)}{=} \int_B \int_{\mathbb{R}^h} \varphi(\mathbf{x}, \mathbf{y}) f_{\mathbf{Z}}(\mathbf{x}, \mathbf{y}) d\mathbf{x} d\mathbf{y} \\
&= \int_{\mathbb{R}^n} \chi_B(\mathbf{y}) \varphi(\mathbf{x}, \mathbf{y}) f_{\mathbf{Z}}(\mathbf{x}, \mathbf{y}) d\mathbf{x} d\mathbf{y} = \int_\Omega \chi_B(\mathbf{Y}) \varphi(\mathbf{X}, \mathbf{Y}) d\mathbb{P} \\
&= \int_\Omega \chi_C(\mathbf{Z}) \varphi(\mathbf{X}, \mathbf{Y}) d\mathbb{P} = \int_\Omega \chi_A \varphi(\mathbf{X}, \mathbf{Y}) d\mathbb{P} = \int_A \varphi(\mathbf{X}, \mathbf{Y}) d\mathbb{P},
\end{aligned}$$

ove χ_A, χ_B e χ_C sono funzioni indicatrici dei rispettivi insiemi, e in piú vale $B \equiv \mathbf{Y}(A) \subseteq \mathbb{R}^k$ e $C \equiv \mathbf{Z}(\mathbb{R}^h \times A) \subseteq \mathbb{R}^n$ dimodoché

$$A = \mathbf{Y}^{-1}(B) = \mathbf{Z}^{-1}(C).$$

□

Osservazione 3.4. La precedente dimostrazione svela un'interpretazione piú pratica dell'attesa condizionata rispetto alla sua definizione alquanto teorica.

Si consideri la (3.2), allora $h(y)$ si può cosí riformulare:

$$h(\mathbf{y}) = \int_{\mathbb{R}^h} \varphi(\mathbf{x},\mathbf{y}) f_{\mathbf{X}|\mathbf{Y}=\mathbf{y}}(\mathbf{x}) d\mathbf{x} = \mathbb{E}[\varphi(\mathbf{X},\mathbf{Y})|\mathbf{Y}=\mathbf{y}] \quad \forall \mathbf{y} \in \mathbb{R}^k, \tag{3.3}$$

in cui

$$f_{\mathbf{X}|\mathbf{Y}=\mathbf{y}} \equiv \begin{cases} f_{\mathbf{Z}}(\mathbf{x},\mathbf{y})/f_{\mathbf{Y}}(\mathbf{y}) & \text{se } f_{\mathbf{Y}}(\mathbf{y}) \neq 0 \\ 0 & \text{se } f_{\mathbf{Y}}(\mathbf{y}) = 0 \end{cases}$$

è la densità di $\mathbf{X}$ condizionata dall'evento $\mathbf{Y}=\mathbf{y}$.

La (3.3), in pratica, si calcola svolgendo la media di $\varphi(\mathbf{X},\mathbf{Y})$ rispetto a $\mathbf{X}$ dopo aver "fissato" l'evento $\mathbf{Y}=\mathbf{y}$, vale a dire considerando la $\mathbf{Y}$ come fosse un parametro uguale alla realizzazione $\mathbf{y}$.

Successivamente, se s'impone $A = \Omega$ (sempre lecito essendo $\mathcal{G}$ una σ-algebra) allora dalla AC2 vale

$$\mathbb{E}[\varphi(\mathbf{X},\mathbf{Y})] = \mathbb{E}[\mathbb{E}[\varphi(\mathbf{X},\mathbf{Y})|\mathbf{Y}]], \tag{3.4}$$

il che significa che l'attesa di $\varphi(\mathbf{X},\mathbf{Y})$ si calcola mediando rispetto a $\mathbf{Y}$ la variabile aleatoria $\mathbb{E}[\varphi(\mathbf{X},\mathbf{Y})|\mathbf{Y}]$, ricavata dalla (3.3) ma considerando $\mathbf{y}$ arbitrario.

Definizione 3.8 (Processo stocastico)**.** Data una dimensione $n \in \mathbb{N}_+$, un processo stocastico è un insieme di vettori aleatori $\mathbf{X}_t$ 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

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

Comunemente s'interpreta l'indice t come il tempo ponendo $I \equiv \mathbb{R}_+$, e cosí sarà fatto in questo scritto.

3.2 DESCRIZIONE CINETICA DI [TIPO] BOLTZMANN

3.2.1 Equazione di Boltzmann omogenea

Descrizione classica

Innanzitutto è necessario caratterizzare 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 interazioni binarie possono aver luogo o, 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, binariamente 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[1].

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

Proposizione 3.2 (Regole d'interazione)**.** *Esistono due funzioni*

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

lineari che soddisfanno (CI, CE) e tali che

$$\begin{aligned} \mathbf{v}' &= \psi(\mathbf{v}, \mathbf{v}_*) \equiv \mathbf{v} + [(\mathbf{v}_* - \mathbf{v}) \cdot \mathbf{n}]\mathbf{n}, \\ \mathbf{v}'_* &= \psi_*(\mathbf{v}, \mathbf{v}_*) \equiv \mathbf{v}_* + [(\mathbf{v} - \mathbf{v}_*) \cdot \mathbf{n}]\mathbf{n}, \end{aligned} \tag{3.5}$$

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

Dimostrazione. Assumendo l'*ansatz*

$$\mathbf{v}' = \mathbf{v} - \gamma\mathbf{n} \quad \text{e} \quad \mathbf{v}'_* = \mathbf{v}_* + \gamma\mathbf{n}, \tag{3.6}$$

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

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

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

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

per cui γ è in realtà una funzione delle velocità precollisionali; la (3.6) diventa allora

$$\mathbf{v}' = \mathbf{v} + [(\mathbf{v}_* - \mathbf{v}) \cdot \mathbf{n}]\mathbf{n} \quad \text{e} \quad \mathbf{v}'_* = \mathbf{v}_* + [(\mathbf{v} - \mathbf{v}_*) \cdot \mathbf{n}]\mathbf{n}. \tag{3.7}$$

□

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

Si noti come la precedente dimostrazione lasci arbitrario $n \in \mathbb{S}^2$, seppure da un punto di vista fisico sia ragionevole porlo parallelo al vettore passante per i centri delle particelle collidenti.

Si osservi come le regole d'interazione nella 3.5 sono bilineari e simmetriche secondo la seguente definzione:

Definizione 3.9. Se le regole d'interazione del tipo (3.5) verificano

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

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 $\{\mathbf{V}_t^{*\prime}\}_{t\in\mathbb{R}_+}$.

Osservazione 3.5. Le variabili aleatorie $\mathbf{V}_t$ e $\mathbf{V}_t^*$ non sono indicizzate (per es. $\mathbf{V}_t^i$) proprio per l'indistinguibilità delle particelle assunta dalla G3, da cui si deduce 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 (3.5) diventano

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

in cui anche n risulta aleatoriamente distribuito; usualmente s'ipotizza $n \sim \mathcal{U}(\mathbb{S}^2)$, vale a dire che non vi sono direzioni preferenziali essendo queste uniformemente distribuite sulla sfera unitaria.

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

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

che è una variabile aleatoria di Bernoulli, indipendente 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 le quali possono influenzare il tasso d'interazione tra le molecole[2];
2. un passo temporale $\Delta t > 0$ per discretizzare il tempo.

Osservazione 3.6. 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.

2 Si noti come B dipenda da $(\mathbf{V}_* - \mathbf{V})\cdot n$, termine ripreso dalla Prop. 3.2, cosa che permette d'interpretare n come la direzione lungo la quale le interazioni sono piú frequenti.

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:

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

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 $\mathbf{V}^{*\prime}_t$) modificando il loro stato al tempo successivo.

Derivazione modello

L'idea successiva, al fine di ricavare un modello dell'evoluzione di 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 una volta applicato $\varphi(\cdot)$ alle $[\mathrm{AR}]_{\mathbf{V}}$,

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

e mediando $\mathbb{E}[\cdot]$ si arriva a

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

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

$$\begin{aligned} \mathbb{E}[\varphi(\mathbf{V}_{t+\Delta t})] &= \mathbb{E}[\varphi((1-\Theta)\mathbf{V}_t + \Theta \mathbf{V}'_t)], \\ &= \mathbb{E}[\mathbb{E}[\varphi((1-\Theta)\mathbf{V}_t + \Theta \mathbf{V}'_t) \mid \mathbf{V}_t, \mathbf{V}^*_t, n]], \\ &= \mathbb{E}[\varphi(\mathbf{V}_t)(1 - B((\mathbf{V}^*_t - \mathbf{V}_t)\cdot n)\Delta t)] \\ &\quad + \mathbb{E}[\varphi(\mathbf{V}'_t) B((\mathbf{V}^*_t - \mathbf{V}_t)\cdot n)]\Delta t, \end{aligned} \tag{3.8}$$

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

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

e passando formalmente al tempo continuo col limite $\Delta t \to 0^+$[3] si ha

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

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

$$\begin{aligned} \frac{d}{dt}\int_{\mathbb{R}^3} \varphi(\mathbf{v}) f(\mathbf{v},t)\,d\mathbf{v} &= \int_{\mathbb{R}^6} \langle B(\mathbf{v},\mathbf{v}_*,n)(\varphi(\mathbf{v}') - \varphi(\mathbf{v}))\rangle f_{\mathbf{V}}(\mathbf{v},\mathbf{v}_*,t)\,d\mathbf{v}d\mathbf{v}_*, \\ \frac{d}{dt}\int_{\mathbb{R}^3} \varphi(\mathbf{v}^*) f(\mathbf{v}_*,t)\,d\mathbf{v}_* &= \int_{\mathbb{R}^6} \langle B(\mathbf{v},\mathbf{v}_*,n)(\varphi(\mathbf{v}'_*) - \varphi(\mathbf{v}_*))\rangle f_{\mathbf{V}}(\mathbf{v},\mathbf{v}_*,t)\,d\mathbf{v}d\mathbf{v}_*, \end{aligned} \tag{3.9}$$

[3] Ciò permette di soddisfare la condizione descritta nel'3.6 per qualunque valore di $B((\mathbf{V}^*_t - \mathbf{V}_t)\cdot n)$.

in cui

$$\langle\cdot\rangle \equiv \frac{1}{4\pi}\int_{S^2} \cdot \, d\mathfrak{n}$$

indica la media rispetto a $\mathfrak{n}$, $B(\mathbf{v},\mathbf{v}_*,\mathfrak{n})$ è una forma breve per $B((\mathbf{v}-\mathbf{v}_*)\cdot\mathfrak{n})$ e si è posto $\mathbf{V}\equiv(\mathbf{V}_t,\mathbf{V}_t^*)$ la cui densità congiunta è $f_{\mathbf{V}}(\mathbf{v},\mathbf{v}_*,t)$. È infine necessario sommare le due equazioni in (3.9). S'inizi dal primo membro:

$$\frac{d}{dt}\int_{\mathbb{R}^3}\varphi(\mathbf{v})f(\mathbf{v},t)d\mathbf{v}+\frac{d}{dt}\int_{\mathbb{R}^3}\varphi(\mathbf{v}^*)f(\mathbf{v}_*,t)d\mathbf{v}_* = 2\frac{d}{dt}\int_{\mathbb{R}^3}\varphi(\mathbf{v})f(\mathbf{v},t)d\mathbf{v}, \quad (3.10)$$

infatti dall'Oss. 3.5 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:

$$B((\mathbf{v}-\mathbf{v}_*)\cdot\mathfrak{n}) = B((\mathbf{v}_*-\mathbf{v})\cdot\mathfrak{n}) \quad \forall \mathbf{v},\mathbf{v}_* \in \mathbb{R}^3; \quad (3.11)$$

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

Ipotesi 3.3 (Caos molecolare)**.** Le particelle interagenti secondo le $[AR]_V$ sono campionante indipendentemente. Tal'ipotesi è piú facile da giustificare matematicamente che fisicamente perché semplifica di molto 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.

Osservazione 3.7. Dal'Ip. 3.3, unitamente alla proprietà 3.1, si deduce 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 $\varphi(v_*)$ nel secondo membro della seconda equazione della (3.9) può essere cosí riformulato:

$$\int_{\mathbb{R}^6}\langle(\varphi(\mathbf{v}))B(\mathbf{v},\mathbf{v}_*,\mathfrak{n})\rangle f(\mathbf{v},t)f(\mathbf{v}_*,t)d\mathbf{v}d\mathbf{v}_*, \quad (3.12)$$

seguendo la medesima logica della (3.10). Applicando i due risultati illustrati nelle (3.10, 3.12) si perviene finalmente all'equazione di Bolztmann omogenea asimmetrica in forma debole:

$$\frac{d}{dt}\int_{\mathbb{R}^3}\varphi f d\mathbf{v} = \frac{1}{4\pi}\int_{\mathbb{R}^6}\int_{S^2} B(\mathbf{v},\mathbf{v}_*,\mathfrak{n})\left(\frac{\varphi'+\varphi_*'}{2}-\varphi\right)ff_* d\mathfrak{n}d\mathbf{v}d\mathbf{v}_*, \quad (3.13)$$

ove, per brevità di notazione, si sono sottintese le dipendenze per tutte le funzioni, espresse, salvo per il nucleo di collisione, da apici $'$ e asterischi $*$ (per una spiegazione piú dettagliata si legga poco dopo nel § 3.2.3).

Tuttavia, qualora le regole d'interazione siano simmetriche secondo la Def. 3.9, come in questo caso data la Prop. 3.2, sempre con un cambio di variabili e sfruttando la parità del nucleo di collisione vale

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

da cui segue l'equazione di Bolztmann omogenea simmetrica in forma debole[4]:

$$\frac{d}{dt}\int_{\mathbb{R}^3}\varphi f d\mathbf{v} = \frac{1}{4\pi}\int_{\mathbb{R}^6}\int_{S^2} B(\mathbf{v},\mathbf{v}_*,n)\left(\varphi'-\varphi\right) f f_* dn d\mathbf{v} d\mathbf{v}_*. \qquad [EtB]_V$$

Si può anche ricavare la forma forte della $[EtB]_V$ considerando le regole d'interazione inverse della $[RI]_V$ [16, § 2.5, p. 15], ma il conto esula dagli scopi di questo elaborato; cionnonostante si riporta come riferimento per il paragrafo a venire:

$$\frac{\partial f}{\partial t} = \frac{1}{4\pi}\int_{\mathbb{R}^3}\int_{S^2} B(\mathbf{v},\mathbf{v}_*,n)\left(f'f'_*-ff_*\right) dn d\mathbf{v}_*. \qquad (3.14)$$

3.2.2 Equazione di Boltzmann disomogenea

L'equazione originariamente ricavata da Boltzmann non è la (3.14), bensí è quella disomogenea la cui unica differenza è la presenza di un termine avvettivo $\mathbf{v}\cdot\nabla f$ a primo membro:

$$\frac{\partial f}{\partial t} + \mathbf{v}\cdot\nabla f = \frac{1}{4\pi}\int_{\mathbb{R}^3}\int_{S^2} B(\mathbf{v},\mathbf{v}_*,n)\left(f'f'_*-ff_*\right) dn d\mathbf{v}_*, \qquad (3.15)$$

dove l'operatore $\nabla \equiv (\partial_{x_1},\partial_{x_2},\partial_{x_3})$ è il gradiente spaziale e la distribuzione 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 a tempo dato.

Il primo membro della (3.15) non è altro che la derivata materiale della distribuzione f: 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.

Proposizione 3.3. *La soluzione dell'equazione del trasporto lineare*

$$\frac{\partial f}{\partial t} + \mathbf{v}\cdot\nabla f = 0 \qquad (3.16)$$

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 (3.16) la velocità $\mathbf{v}$ nell'operatore avvettivo non dipende né dallo spazio né dal tempo, quindi si possono prendere le curve $\mathbf{x}(t)$ tali che

$$\frac{d\mathbf{x}}{dt} = \mathbf{v} \quad\Longrightarrow\quad \mathbf{x}(t) = \mathbf{x}_0 + \mathbf{v}t \qquad (3.17)$$

dette, appunto, curve caratteristiche della (3.16); valutando allora la distribuzione $f(\mathbf{x},\mathbf{v},t)$ lungo di esse si può definire $\hat{f}(t) = f(\mathbf{x}(t),\mathbf{v},t)$ che derivata equivale a

$$\frac{d\hat{f}}{dt} = \frac{\partial f}{\partial t} + \nabla f\cdot\frac{d\mathbf{x}}{dt} = \frac{\partial f}{\partial t} + \mathbf{v}\cdot\nabla f = 0 \quad\Longrightarrow\quad \hat{f}(t) = \text{costante} \quad \forall t \geqslant 0$$

4 Si dice debole siccome essa vale per una una funzione *test* φ arbitraria.

da cui si deduce che f è costante lungo le caratteristiche; ma dalla (3.17) si può scrivere, trascurando la dipendenza dal tempo, $\mathbf{x}_0 = \mathbf{x} - \mathbf{v}t$, dunque la soluzione in un generico punto $(\mathbf{x},\mathbf{v},t)$ sarà data da

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

considerando nella prima uguaglianza la caratteristica che al tempo t passa per $\mathbf{x}$. □

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 $\mathbf{v}$, movimento che rappresenta la variazione medi statistica delle posizioni molecolari.

D'altra parte il secondo membro della (3.15) rappresenta la variazione media statistica dalle velocità molecolari a causa delle collisioni (viene perciò anche chiamato operatore collisionale).

Pertanto la (3.15) 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 tramite la (3.15) era di raccordare due mondi: il mondo 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 fisiche anche in condizioni di equilibrio termodinamico (agitazione termica).

Ciò è stato possibile grazie all'introduzione dei momenti statistici, ossia di quantità macroscopiche calcolabili a partire dalla distribuzione f soddisfacente la (3.15); i principali sono

$$\begin{aligned}
\rho(\mathbf{x},t) &\equiv \int_{\mathbb{R}^3} f(\mathbf{x},\mathbf{v},t)\,d\mathbf{v} && \text{densità,}\\
\mathbf{u}(\mathbf{x},t) &\equiv \frac{1}{\rho(\mathbf{x},t)}\int_{\mathbb{R}^3} \mathbf{v} f(\mathbf{x},\mathbf{v},t)\,d\mathbf{v} && \text{velocità massica,}\\
E(\mathbf{x},t) &\equiv \frac{1}{\rho(\mathbf{x},t)}\|\mathbf{v}\|^2\int_{\mathbb{R}^3} f(\mathbf{x},\mathbf{v},t)\,d\mathbf{v} && \text{energia totale,}\\
e(\mathbf{x},t) &\equiv \frac{1}{\rho(\mathbf{x},t)}\int_{\mathbb{R}^3} \|\mathbf{v}-\mathbf{u}(\mathbf{x},t)\|^2 f(\mathbf{x},\mathbf{v},t)\,d\mathbf{v} && \text{energia interna,}\\
\theta(\mathbf{x},t) &\equiv \frac{1}{3}e(\mathbf{x},t) && \text{temperatura.}
\end{aligned} \tag{3.18}$$

Tali quantità sono definite da f seppure questa sia in genere infattibile da ricavare dalla (3.15), 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 (3.18). La convergenza di questi a un valore stazionario è il nesso tra i due mondi attraverso la 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 $[\text{EtB}]_V$ e
2. sono applicati a contesti[5] molto distanti da quello originale delle velocità di particelle di un gas.

Si ripropongono in questo paragrafo 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[6] che descrive i suoi $n \in \mathbb{N}_+$ stati microscopici al tempo t; allora le Ipp. 3.1 e 3.3 si possono cosí riformulare:

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

- B1 la distribuzione statistica dei microstati è uniforme nello spazio, e quindi non dipende da esso;
- B2 gli agenti interagiscono solo binariamente, ovvero le interazioni a coppia sono le piú frequenti;
- B3 gli agenti sono indistinguibili cosicché ognuno è rappresentativo del loro insieme;
- B4 due agenti interagenti 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

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

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. 3.11), 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.

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

$$\mu(\mathbf{X}_t, \mathbf{X}^*_t, \mathbf{w}, t) \colon \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^k \times [0, +\infty) \to \mathbb{R}_+, \qquad [\text{NI}]_\chi$$

per controllare quant'è "probabile" che i due agenti interagiscano, e coefficienti $\mathbf{w} \in \mathbf{R}^k$, con $k \in \mathbb{N}$, potenzialmente stocastici.

5 Esempi importati sono quello sociofisico, come i vari modelli sulle opinioni, ed econofisico, come quello qui considerato delle città

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

Osservazione 3.8. Se non si sa o non si vuole toccare quell'aspetto modellistico si può semplicemente porre unitario:

$$\mu \equiv 1, \quad \forall (\mathbf{x},\mathbf{x}_*,\mathbf{w},t) \in \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^k \times [0,+\infty)$$

sottintendendo in tal modo un'uniforme "probabilità" d'interazione.

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

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

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

Osservazione 3.9. l'acronimo $[\mathrm{AR}]_\chi$ sta per «Azione-Reazione» poiché si suppone che ogn'interazione (azione), ossia $\Theta = 1$, necessariamente modifica entrambi gli stati degli agenti coinvolti (reazione); in alcuni contesti [17] questa regola può essere alterata dimodoché solo l'agente interagente sia modificato portando alle interazioni «Azione-Azione».

Infine, riapplicando un ragionamento analogo a quello svolto nel § 3.2.1 colla quantità osservabile $\varphi\colon \mathbb{R}^n \to \mathbb{R}$, si perviene all'equazione di tipo Boltzmann omogenea generale

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

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.

Osservazione 3.10 ($[\mathrm{EtB}]_\chi$ simmetrica)**.** Qualora $[\mathrm{NI}]_\chi$ sia pari (Def. 3.10) e le regole d'interazione $[\mathrm{RI}]_\chi$ siano simmetriche (Def. 3.11), la $[\mathrm{EtB}]_\chi$ si può riformulare come

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

mediante un cambio di variabili atto a invertire $\mathbf{x}$ e $\mathbf{x}_*$ solo per la differenza $\varphi'_* - \varphi_*$.

Definizione 3.10 ($[\mathrm{NI}]_\chi$ pari)**.** Se il tasso d'interazione in $[\mathrm{NI}]_\chi$ 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 si dice pari.

Definizione 3.11 ($[\mathrm{RI}]_\chi$ simmetriche)**.** Se le regole d'interazione del tipo $[\mathrm{RI}]_\nu$ verificano

$$\begin{aligned} \psi(\mathbf{x},\mathbf{x}^*,\mathbf{y}) &= \psi_*(\mathbf{x}^*,\mathbf{x},\mathbf{y}) \\ \psi_*(\mathbf{x},\mathbf{x}^*,\mathbf{y}_*) &= \psi(\mathbf{x}^*,\mathbf{x},\mathbf{y}_*) \end{aligned} \quad \forall \mathbf{x},\mathbf{x}_* \in I \subseteq \mathbb{R}^n \text{ e } \forall \mathbf{y},\mathbf{y}_* \in \mathbb{R}^h,$$

allora si dicono simmetriche. In altre parole, la simmetria implica

$$\mathbf{X}'_t = \psi_*(\mathbf{X}^*_t,\mathbf{X}_t,\mathbf{y}) \quad \text{e} \quad \mathbf{X}^{*\prime}_t = \psi(\mathbf{X}^*_t,\mathbf{X}_t,\mathbf{y}_*).$$

Notazione

Nei previ paragrafi si è fatto spesso uso, specie per le varie equazioni [di tipo] Boltzmann (3.13–3.15), $[\mathrm{EtB}]_V$ e $[\mathrm{EtB}]_X$, di una notazione abbreviata per tutti gli oggetti salvo il tasso d'interazione μ. In questo breve paragrafo si definiscono in maniera piú esplicita queste abbreviazioni facendo riferimento al contesto generale sviluppato poco fa.

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 le variabili postinterazionali:

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

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

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

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

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

Analisi dimensionale

Finora tutte le equazioni passate fanno riferimento a un tempo adimensionale, sebbene l'introduzione di un tempo sia una modifica alquanto indolore e che lasci trasparire un'interessante interpretazione del passo temporale Δt.

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

Osservazione 3.11. 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

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

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

$$\frac{d}{d\tilde{t}}\int_{\mathbb{R}^n}\varphi\tilde{f}d\mathbf{x}=\frac{1}{\tau}\int_{\mathbb{R}^{2n}}\left\langle\mu(\mathbf{x},\mathbf{x}_*,w,\tilde{t}/\tau)\frac{\varphi'+\varphi'_*-\varphi-\varphi_*}{2}\right\rangle\tilde{f}\tilde{f}_*d\mathbf{x}d\mathbf{x}_*. \tag{3.20}$$

Per capire come varia il tasso d'interazione μ è sufficinete dimensionalizzare la $[Ber]_X$:

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

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 (3.20) nell'equazione di tipo Boltzmann dimensionale

$$\frac{d}{d\tilde{t}}\int_{\mathbb{R}^n}\varphi\tilde{f}d\mathbf{x}=\int_{\mathbb{R}^{2n}}\left\langle\tilde{\mu}(\mathbf{x},\mathbf{x}_*,w,\tilde{t})\frac{\varphi'+\varphi'_*-\varphi-\varphi_*}{2}\right\rangle\tilde{f}\tilde{f}_*d\mathbf{x}d\mathbf{x}_*. \tag{3.21}$$

Osservazione 3.12 (frequenza d'interazione)**.** Dall'identità $\Delta\tilde{t}=\tau\Delta t$ si può interpretare Δt come la frazione della dimensione τ dopo la quale può avvenire un'interazione, da cui si deduce che $\Delta\tilde{t}$ è l'intervallo di tempo tra due interazioni.

Di conseguenza si può vedendo l'$[AR]_X$ come un fenomeno periodico di periodo $\Delta\tilde{t}$ e frequenza

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

in cui, come prima $1/\tau$ è la dimensione della frequenza adimensionale f.

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 3.13 (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;
- ◇ similmente 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 (3.21) 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 scale 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 analizza 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 connessi; è anche ovvio che l'interazione modifichi, in qualche modo da definire, la loro popolazione.

In questo paragrafo si applicano i concetti della teoria sviluppata nei §§ 2.1 e 3.2.3 e, in particolare, la notazione usata e definita. S'inizia mostrando una derivazione esatta la quale, cioè, considera la topologia esatta; quindi si approfondisce un'approssimazione nella quale si "perde" la topologia indotta dalla rete; infine, si considerano i collegamenti tra la derivazione esatta e qualla 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[7], 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 3.14 (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 rispetto a un intervallo continuo. Tal'è la ragione nel scegliere S_t reale: dopo aver fatto i conti normalmente si può poi approssimare per eccesso o difetto ricavando la taglia intera effettiva; 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

$$f(i,s,t) \colon \mathcal{I} \times \mathbb{R}_+ \times [0,+\infty) \to \mathbb{R}_+,$$

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

$$f(i,s,t) \equiv \frac{1}{N} \sum_{j \in \mathcal{I}} f_j(s,t) \otimes \delta(i-j), \tag{3.22}$$

7 Perdipiú la B2 è valida *a un tempo* t, ma nell'unità di tempo *dimensionale* $\tilde{t}$ le interazioni si possono considerate come non binarie, come già discusso alla fine del § 3.2.3.

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

$$f_i = f_i(s,t) : \mathbb{R}_+ \times [0,+\infty) \to \mathbb{R}_+$$

è la densità di S_t dell'agente i-esimo.

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:

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

mentre le densità f_i preservano la naturale distinguibilità degli agenti indotta dalla topologia sottostante.

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

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

ove $A : \mathcal{I}\times\mathcal{I} \to \mathbb{R}$ è una funzione che a ogni coppia d'indici (I,I_*) associa la relativa entrata nella 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); in tal modo l'interazione non ha possibilità d'avvenire se i due agenti non sono connessi.

Osservazione 3.15. Rispetto al caso generale $[\mathrm{Ber}]_\chi$ il tasso d'interazione ha forma

$$\mu(i,i_*) \equiv A(i,i_*), \tag{3.23}$$

esso, cioè, è per definizione costante sia rispetto agli stati S_t e S^*_t che rispetto al tempo, oltre a non avere coefficienti[8] $\mathbf{w}$.

Osservazione 3.16. 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 regole d'interazione nella $[\mathrm{RI}]_\chi$ diventano

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

Osservazione 3.17. Confrontato a $[\mathrm{RI}]_\chi$, solo la funzione della città interagente ψ presenta un coefficiente stocastico $\gamma \equiv \gamma \in \mathbb{R}$, mentre quella relativa alla città ricevente ψ_* non dipende da potenziali coefficienti[9] $\mathbf{y}_*$.

8 Questa situazione si può simbolicamente rappresentare ponendo $k = 0$.

9 Come nell'Oss. 3.15, tale condizione si può simbolicamente rappresentare ponendo $h_* = 0$.

Unendo le $[\mathrm{Ber}]_S^{\mathbf{A}}$ e $[\mathrm{RI}]_S$, le $[\mathrm{AR}]_{\mathbf{X}}$ si scrivono

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

coerentemente col fatto che in un cotesto urbano, qualora una città-nodo interagisca con un'altra, entrambe debbano variare il loro stato (Oss. 3.9).

Osservazione 3.18. Paragonato a $[\mathrm{AR}]_{\mathbf{X}}$ ci si potrebbe chiedere perché non si considera l'intero vettore aleatorio $\mathbf{X}_t$, come pure nella $[\mathrm{RI}]_S$; la ragione è che I è una componente statica che non varia nel tempo.

In ogni caso, 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 soddisfa la staticità di I e I_* e completa, assieme a $[\mathrm{AR}]_S$, la formulazione $[\mathrm{AR}]_{\mathbf{X}}$ piú generale.

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

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_{\mathcal{I}}\int_{\mathbb{R}_+}\Phi f\,\mathrm{d}v\mathrm{d}i = \int_{\mathcal{I}^2}\int_{\mathbb{R}_+^2} A(i,i_*)\frac{\langle \Phi' + \Phi_*' - \Phi - \Phi_*\rangle}{2} f f_*\,\mathrm{d}s\mathrm{d}s_*\mathrm{d}i\mathrm{d}i_*, \quad [\mathrm{EtB}]_S^{\mathbf{A}}$$

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 $[\mathrm{RI}]_S$.

Osservazione 3.19. Anche se la derivazione della $[\mathrm{EtB}]_S^{\mathbf{A}}$ è già stata spiegata nel dettaglio nel § 3.2.1, rimane interessante riportare la forma del passaggio piú difficile, ossia quella della media condizionata (3.8) (qui mostrato solo per $S_{t+\Delta t}$):

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

Si noti come il condizionamento sia solo rispetto a I a I_*, proprio poiché Θ in $[\mathrm{Ber}]_S^{\mathbf{A}}$, tramite il tasso d'interazione μ nella (3.23), dipende solo da essi.

Analisi delle regole d'interazione

Si approfondiscono in questo paragrafo le regole d'interazione $[\mathrm{RI}]_S$.

Innanzitutto, il perché nella $[\mathrm{AR}]_S$ si considerano interazioni «Azione-Reazione» (Oss. 3.9) discende da un vincolo fisico: se una città interagisce con un'altra, scambiando popolazione, necessariamente anche l'altra varia il proprio stato.

Dopodiché, passando alle realizzazioni di tutte le variabili aleatorie considerate, vale a dire scrivendole in minuscolo e omettendo la dipendenza dal tempo, le regole d'interazione $[\mathrm{RI}]_S$ sono cosí definite[10]

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

ove

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

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

Ipotesi 3.5. Per avere un postato s fisicamente sensato, ossia positivo, quando $\gamma = 0$, si assume che il tasso d'emigrazione E sia superiormente limitato dalla costante $\lambda \equiv \sup \mathrm{E}$, tale che $\lambda \in (0,1)$, e inferiormente limitato dall'origine:

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

D'altra parte il postato s'_* [per il momento] non si considera poiché è per definizione sempre positivo qualunque sia il tasso d'immigrazione $\mathrm{I} \in [0,+\infty)$.

Se invece γ è non nullo, dev'essere anch'esso limitato:

$$s' = s(1-\mathrm{E}+\gamma) > 0 \implies \gamma > \mathrm{E}-1, \quad \forall \mathrm{E} \in [0,\lambda], \tag{3.24}$$

La scelta di $>$ anziché $\geqslant$ nella (3.24) è ben fondata: di fatto si sta supponendo che le fluttuazioni non possano spopolare una città[11]; tale condizione discende dal rapporto s_*/s contenuto nelle regole d'emigrazione proposte nel § 4.3.

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

Ipotesi 3.6. γ deve soddisfare le seguenti caratteristiche:

- F1 può assumere sia valori positivi che negativi, ma non minori del vincolo imposto da (3.24);
- F2 la media è scelta arbitrariamente posta allo 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 allora analizzare alcune distribuzioni continue:

1. la distribuzione normale $\mathcal{N}(\mu,\gamma)$ non soddisfà la F1 poiché può assumere valori reali arbitrari con probabilità non nulla;

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

11 Ciò non significa che il modello non possa prevedere un tale fenomeno, poiché la taglia può arbitrariamente avvicinarsi a zero ma mai esserne uguale, raggiungendolo solo *a posteriori* dopo l'approssimazione dai numeri reali a quelli interi (Oss. 3.14).

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 soddisfà 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

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

ove α e β sono i parametri rispettivamente di forma e di scala, mentre

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

è la funzione gamma

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

Imponendo la F1 si ottiene

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

in cui σ^2 indica la varianza, da cui

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

La F2 si può semplicemente soddisfare traslando i valori campionanti della $\hat{\gamma}$ 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'altra parte per la F1 bisogna salvaguardarsi dai casi degeneri della distribuzione gamma: essa infatti se $\alpha = 1$ diventa $\mathrm{Exp}(\beta)$, mentre se $\alpha < 1$ diverge all'origine; per avere quindi probabilità nulla di campionare da $\hat{\gamma}$ l'origine [e quindi -$\langle\hat{\gamma}\rangle$ dopo la traslazione] è necessario porre

$$\alpha > 1 \implies \frac{(1-E)^2}{\sigma^2} \geqslant \frac{(1-\lambda)^2}{\sigma^2} > 1 \implies \sigma < |1-\lambda| = 1-\lambda. \tag{3.25}$$

La (3.25) implica quindi che non è possibile avere una varianza arbitraria per poter soddisfare la F1, ma che questa è limitata superiormente dal massimo tasso d'emigrazione λ, interpretabile come attrattività: piú grande è λ piú piccola è la varianza, 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 dunque la distribuzione gamma tende a un'esponenziale; questo ha l'effetto d'introdurre una pericolosa asimmetria nella densità, come si può notare nella la Fig. 3.1: una volta centrata rispetto alla media la moda (il valore piú frequente) ha valore negativo portando, in

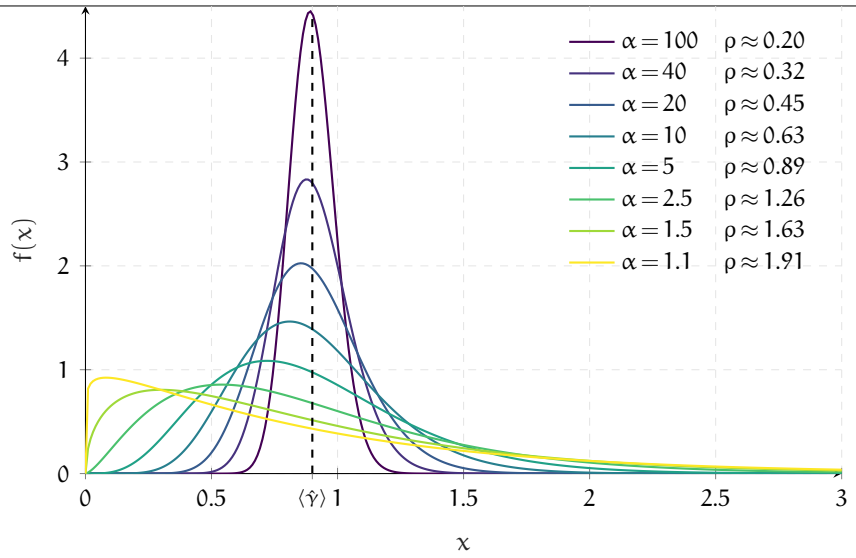


Figura 3.1: Transizione della densità gamma verso l'asimmetria con $\lambda = 0.1$ ed $E = \lambda$, da cui $\langle\hat{\gamma}\rangle = 0.9$ e $\beta = \langle\hat{\gamma}\rangle/\alpha$.

un orizzonte temporale finito, le perturbazioni negative a essere quelle piú frequenti.

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 dimodoché l'indice di asimmetria

$$\rho \equiv \frac{2}{\sqrt{\alpha}}$$

sia sufficientemente piccolo; dalla Fig. 3.1 si può perciò imporre $\rho \leqslant 0.2$, cioè

$$\alpha \geqslant 100 \implies \frac{(1-E)^2}{\sigma^2} \geqslant 100 \implies \sigma \leqslant \frac{1-\lambda}{10}, \tag{3.26}$$

dove, come nella (3.25), si è imposto nell'ultima implicazione il caso peggiore $E = \lambda$. Dunque il vero limite superiore, per preservare la simmetria, è di un ordine di grandezza inferiore a quello precedentemente stimato.

Osservazione 3.20. 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 oppure si può troncare una gaussiana affinché soddisfi il vincolo di positività (3.24).

Non manca che caratterizzare il tasso d'immigrazione I:

Ipotesi 3.7. Si postula che l'interazione 14 conservi [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 3.21. Le 14 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, si discutono quindi con vari studi parametrici per poi infine interpretarle analizzando cosa queste dicono sul fenomeno della migrazione.

3.3.2 Equazione di tipo Boltzmann approssimata

L'approssimazione si fonda sulla seguente matrice:

Definizione 3.12 (Matrice dei gradi **B**)**.** Sia $\mathbf{B} \in \mathbb{R}^{N\times N}$ una matrice di rango uno

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

definite tramite il prodotto diadico dei vettori

$$\begin{aligned} \mathbf{k}^+ &\equiv (k_1^+, k_2^+, \dots, k_N^+)^\top = \mathbf{A}\mathbf{1} \in \mathbb{R}^N, \\ \mathbf{k}^- &\equiv (k_1^-, k_2^-, \dots, k_N^-)^\top = \mathbf{A}^\top \mathbf{1} \in \mathbb{R}^N, \end{aligned} \tag{3.27}$$

rispettivamente dei gradi uscenti ed entranti, e la costante[12]

$$D_N \equiv \sum_{i\in\mathcal{I}} k_i^+ = \mathbf{1}^\top \mathbf{k}^+ = \sum_{i\in\mathcal{I}} k_i^- = \mathbf{1}^\top \mathbf{k}^- = \mathbf{1}^\top \mathbf{A}\mathbf{1} = \|\mathbf{A}\|_1. \tag{3.28}$$

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

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

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. 3.12, la $[\mathrm{Ber}]_S^{\mathbf{A}}$ diventa

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

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

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

dove g e g_* sono densità definite come la (3.22) ma 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 (3.27)? La risposta non è immediata ma discende in essenza su come la $[\mathrm{EtB}]_S^{\mathbf{B}}$ può essere riformulata. A tal scopo si necessita di un osservazione:

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

Osservazione 3.22 (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 (3.22) 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. 3.22 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: usando, infatti, l'Ip. 3.8 nella $[\text{EtB}]^{\mathbf{B}}_{\mathbf{S}}$ porta a

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

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

$$\frac{d}{dt}\int_{\mathbb{R}_+} \varphi \mathbf{g} ds = \frac{1}{N}\int_{\mathbb{R}_+^2} \frac{\langle \varphi' + \varphi_*' - \varphi - \varphi_*\rangle}{2} \mathbf{g} \odot \mathbf{B}\mathbf{g}_* ds ds_*, \tag{3.29}$$

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 (3.22) rispetto alla popolazione s:

$$\bar{g}_N(s,t) \equiv \int_{\mathcal{I}} g(i,s,t)di = \frac{1}{N}\sum_{i\in\mathcal{I}} g_i = \frac{1}{N}\mathbf{1}^\top \mathbf{g}, \tag{3.30}$$

ove $\mathbf{1} \equiv (1,1,\ldots,1) \in \mathbb{R}^N$, che non è che la media tra le densità di S_t tra tutt'i vertici[13]; premoltiplicando la (3.29) per $\frac{1}{N}\mathbf{1}^\top$ e usando la (3.30) si ha

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

Osservazione 3.23. 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. 3.22, cioè che la [E$\bar{g}$] si ricava perdendo le informazioni legati agl'indici negli osservabili.

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

La [Eḡ] non è ancora un'equazione chiusa per $\bar{g}$ 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

$$\frac{d}{dt}\int_{\mathbb{R}_+} \varphi \bar{g}_N ds = \frac{1}{N^2}\int_{\mathbb{R}_+^2} \frac{\langle \varphi' + \varphi'_* - \varphi - \varphi_* \rangle}{2D_N} (\mathbf{k}^\top \mathbf{g})(\mathbf{k}^\top \mathbf{g}_*) ds ds_*, \tag{3.31}$$

che richiede d'introdurre una nuova densità di probabilità:

Definizione 3.13 (Densità dei gradi **k**)**.** Sia

$$d_N : \mathbb{R}_+ \times \{0,1,\dots,N\} \times \mathbb{R}_+ \to \mathbb{R}_+,$$

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

$$\mathcal{I}_k \equiv \{i \in \mathcal{I} \mid k_i = k\} \subseteq \mathcal{I} \tag{3.32}$$

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

$$\bar{g}_N(s,t) = \frac{1}{N}\sum_{i \in \mathcal{I}} g_i(s,t) = \sum_{k=0}^{N} \frac{1}{N} \sum_{i \in \mathcal{I}_k} g_i(s,t) = \sum_{k=0}^{N} d_N(s,k,t), \tag{3.33}$$

$$\mathbf{k}^\top \mathbf{g}(s,t) = \sum_{i \in \mathcal{I}} k_i g_i(s,t) = \sum_{k=0}^{N} k \sum_{i \in \mathcal{I}_k} g_i(s,t) = N \sum_{k=0}^{N} k d_N(s,k,t).$$

Un analogo discorso vale per le f_i in luogo delle g_i.

E dalla Def. 3.13 segue la sua normalizzazione:

Definizione 3.14 (Densità dei gradi **k** normalizzati)**.** Siano

$$\hat{k}_i \equiv \frac{k_i}{N} \in \hat{\mathcal{K}}, \quad \forall i \in \mathcal{I},$$

i gradi normalizzati di un generico nodo rappresentativo di $\mathcal{G}$, in cui

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

è l'insieme discreto dei gradi normalizzati; allora la densità dei gradi normalizzati

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

è cosí definita

$$\hat{d}_N(s,\hat{k},t) \equiv N d_N(s, N\hat{k}, t). \tag{3.34}$$

Osservazione 3.24. Visto che due elementi consecutivi dell'insieme $\hat{\mathcal{K}}$ sono separati da un passo costante e uguale a $1/N$, si può definire $\Delta\hat{k} \equiv 1/N$ che da (3.34) 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 (basta definire $\Delta k \equiv 1$) essendone un mero riscalamento.

Osservazione 3.25. L'Oss. 3.24 dà la possibilità anche d'interpretare la (3.33) come un uguaglianza tra densità marginali: $\bar{g}_N = \bar{d}_N = \bar{\hat{d}}_N$.

Con tutte queste definizioni e osservazioni, la (3.31) si può riformulare in termini di $\hat{d}_N$

$$\frac{d}{dt}\sum_{k=0}^{N}\int_{\mathbb{R}_+}\varphi d_N ds = N^2\sum_{k,k_*=0}^{N}\int_{\mathbb{R}_+^2}\frac{kk_*}{D_N}\frac{\langle\varphi'+\varphi'_*-\varphi-\varphi_*\rangle}{2}d_N d^*_N ds ds_*,$$

dove k_* è il grado associato a g^*_N, e poi in termini di $\hat{d}_N$

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

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

$$\bar{D}_N \equiv \frac{D_N}{N^2} = \sum_{\hat{k}\in\hat{\mathcal{K}}}\int_{\mathbb{R}_+}\hat{k}\hat{d}_N ds\Delta\hat{k}, \tag{3.35}$$

da cui

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

Valutando il limite $N\to\infty$ alla (3.35) e ai membri Ⓘ e Ⓘⓘ

$$(3.35)\xrightarrow{N\to\infty}\bar{D}\equiv\int_0^1\int_{\mathbb{R}_+}\hat{k}\hat{d}\,ds d\hat{k},\qquad \text{Ⓘ}\xrightarrow{N\to\infty}\frac{d}{dt}\int_0^1\int_{\mathbb{R}_+}\varphi\hat{d}\,ds d\hat{k}$$
$$\text{Ⓘⓘ}\xrightarrow{N\to\infty}\int_0^1\int_0^1\int_{\mathbb{R}_+^2}\frac{\hat{k}\hat{k}_*}{\bar{D}}\frac{\langle\varphi'+\varphi'_*-\varphi-\varphi_*\rangle}{2}\hat{d}\hat{d}^*\,ds ds_* d\hat{k}d\hat{k}_*,$$

si perviene infine a un'equazione di tipo Boltzmann:

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

Si è dunque dimostrato il seguente teorema:

Teorema 3.1 (di rilassamento della topologia)**.** *La* $[EtB]^{\mathbf{B}}_S$ *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* $\mu(\hat{k},\hat{k}_*)\equiv(\hat{k}\hat{k}_*)/\bar{D}$.

Il significato del Teo. 3.1 è profondo: approssimare $[EtB]^{\mathbf{A}}_S$ con $[EtB]^{\mathbf{B}}_S$ 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. Nella Fig. 3.2 è presente uno schema riassuntivo del corrente paragrafo.

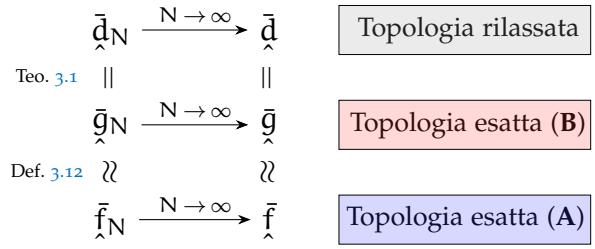


Figura 3.2: Schema riassuntivo del § 3.3.2.

Osservazione 3.26. 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 $[\mathrm{EtB}]^k_S$.

Osservazione 3.27. La Fig. 3.2 contiene al suo interno $\hat{f}$ che ha un'analoga definizione alla Def. 3.14, ma normalizzando gl'indici: siano

$$\hat{\imath} \equiv \frac{i}{N} \in \hat{\mathcal{I}}, \quad \forall i \in \mathcal{I},$$

gl'indici normalizzati di un generico nodo rappresentativo di $\mathcal{G}$, in cui

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

è l'insieme discreto degl'indici normalizzati. Allora la (3.22) normalizzata, per analogia colla (3.34), si può definire

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

ma per la condizione di normalità $\mathbb{P}([0,1] \times \mathbb{R}_+) = 1$, $\forall t \geqslant 0$, deve valere[14]

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

anch'essa analoga alla (3.34); difatti con questa la $\hat{f}$ si scrive

$$\hat{f}(\hat{\imath},s,t) = \frac{1}{N} \sum_{\hat{\jmath} \in \hat{\mathcal{I}}} f_{\hat{\jmath}}(s,t) \otimes \delta(N(\hat{\imath} - \hat{\jmath})) = \frac{1}{N} \sum_{\hat{\jmath} \in \hat{\mathcal{I}}} f_{\hat{\jmath}}(s,t) \otimes \delta(\hat{\imath} - \hat{\jmath}).$$

Con tali definizioni la densità marginale $\bar{f}$ si può riformulare mediante il cambio di variabili $i = N\hat{\imath}$:

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

e, definendo il passo $\Delta\hat{\imath} \equiv 1/N$ tra due indici normalizzati consecutivi, si arriva al limite $N \to \infty$ a

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

14 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.

mentre la $\hat{f}$ diventa

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

Similmente vale per $\bar{g}$.

Osservazione 3.28 (Sulla bontà di $\mathbf{A} \approx \mathbf{B}$)**.** Su un aspetto è 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 3.29. 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 di $[RI]_\chi$ lineari e simmetriche.

3.3.3 Altri nessi distinguibile-indistinguibile

Nel precedente paragrafo si è esplorato come si può "perdere" la struttura sottostante indotta dal grafo passando a una densità dipendente dal grado dei nodi, recuperando in tal modo un'equazione classica di tipo Boltzmann. Risulta perciò stimolante esplorare se sussitano certe ipotesi che permettano di trasformare la $[E\bar{g}]$ in una forma della $[EtB]_\chi$, approfondendo cosí i legami tra la teoria retale sviluppata nei due previ due paragrafi e la teoria di tipo Botlzmann.

Ipotesi semplificative

A questo scopo valgono le seguenti:

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

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

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

S2 Gli agenti sono indistinguibili rispetto al grafo:

$$f_i(s,t) = f_j(s,t) = f(s,t) \quad \forall i,j \in \mathcal{I},$$

S3 Le regole d'interazione $[RI]_S$ sono simmetriche (Def. 3.11).

Analisi della S1

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

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

ma ricordando (3.30) si arriva a

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

la quale è analoga alla classica equazione di tipo Boltzmann $[\mathrm{EtB}]_\chi$ con microstato $\mathbf{X}_t \equiv S_t$ scalare, regole d'interazione $[\mathrm{RI}]_S$ e nucleo d'interazione unitario[15] $\mu \equiv 1$ nella $[\mathrm{Ber}]_\chi$. Ciò significa che con S1 gli agenti, nonostante siano distinti per il grafo, si possono vedere come indistinguibili purché si consideri la distribuzione media (3.30).

Analisi della S2

Sotto tal'ipotesi, che equivale coll'assumere

$$\mathbf{f} = \mathbf{1}f \quad \text{e} \quad \mathbf{f}_* = \mathbf{1}f_*,$$

la [Eḡ] diventa

$$\begin{aligned}\frac{d}{dt}\int_{\mathbb{R}_+}\varphi\bar{f}ds &= \frac{1}{N^2}\int_{\mathbb{R}_+^2}\frac{\langle\varphi'+\varphi'_*-\varphi-\varphi_*\rangle}{2}f\mathbf{1}^\top\mathbf{A}\mathbf{1}f_*dsds_* \\ &= \frac{\mathbf{1}^\top\mathbf{A}\mathbf{1}}{N^2}\int_{\mathbb{R}_+^2}\frac{\langle\varphi'+\varphi'_*-\varphi-\varphi_*\rangle}{2}ff_*dsds_*\end{aligned}$$

e rimembrando (3.28) si ottiene

$$\frac{d}{dt}\int_{\mathbb{R}_+}\varphi fds = \int_{\mathbb{R}_+^2}\frac{D_N}{N^2}\frac{\langle\varphi'-\varphi+\varphi'_*-\varphi_*\rangle}{2}ff^*dsds_*.$$

In questo contesto il rapporto $D_N/N^2 \in [0,1]$ (grado medio normalizzato della rete) rappresenta quanto la rete è topologicamente simile a una completamente connessa[16].

Per il resto l'equazione è analoga alla classica equazione di tipo Boltzmann $[\mathrm{EtB}]_\chi$ con microstato $\mathbf{X}_t \equiv S_t$ scalare, regole d'interazione $[\mathrm{RI}]_S$ e nucleo d'interazione nella $[\mathrm{Ber}]_\chi$ costante $\mu \equiv D_N/N^2 = \bar{D}_N$.

Dunque l'indistinguibilità degli agenti ha una notevole conseguenza sulla [Eḡ]: riassume 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.

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

16 Difatti D_N è interpretabile come il numero di lati presenti in un grafo diretto e che ha come limite superiore prorio N^2, ossia il numero totale di coppie [e quindi lati] dati N nodi.

Analisi della S1, S2 e S3

Visto che vale la S1 si può partire dalla (3.36) nella quale la densità media (3.30) diventa per S2

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

ossia la $\bar{f}$ coincide con quella di tutti gli agenti[17], essendo questi, appunto, indistinguibili.

In tal modo la (3.36) diventa

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

che unita all'S3 porta all'equivalenza

$$\int_{\mathbb{R}_+^2} \langle \varphi'_* - \varphi_* \rangle f f_* ds ds_* = \int_{\mathbb{R}_+^2} \langle \varphi' - \varphi \rangle f f^* ds ds_*,$$

dalla quale si ha

$$\frac{d}{dt}\int_{\mathbb{R}_+} \varphi f ds = \int_{\mathbb{R}_+^2} \langle \varphi' - \varphi \rangle f f^* ds ds_*.$$

Quest'equazione, come per gli altri casi, è analoga alla classica equazione di tipo Boltzmann simmetrica (3.19) con microstato $\mathbf{X}_t \equiv S_t$ scalare, regole d'interazione $[RI]_S$ e nucleo d'interazione nella $[Ber]_\chi$ unitario $\mu \equiv 1$.

17 Si noti anche che la perdita della dipendenza della distribuzione media dal numero di nodi N è coerente col limite $N \to \infty$, sempre ben approssimabile nei casi classici con un numero di agenti $N \gg 1$ [come lo studio di un gas].

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 per analizzare i risultati per poi motivare rapidamente il perché le fluttuazioni possono [anzi devono] essere trascurate. Per le simulazioni si propongono varie leggi d'emigrazione che studiate tramite dei studi parametrici, cercando successivamente d'interpretare cosa queste dicono sul fenomeno della migrazione in sé. Infine si mostra brevemente una simulazione per l'Italia, confermando che i risultati migliorano col numero d'agenti.

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 $[EtB]_X$ 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 $[EtB]_X$, specie nel caso di $[RI]_X$ 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 (3.15), dipende dalla densità medesima.

Per tali ragioni si procede in maniera piú semplice: conoscendo $[AR]_X$, 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.

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

Osservazione 4.2. Piú in generale, nella linea 1, $\mathcal{S}_0$ si può campionare da una densità $\bar{l}_0$ iniziale; tuttavia non conoscendone alcuna per la distribuzione della popolazione, si è deciso di definire $\mathcal{S}^0$ come un vettore uniforme rispetto a una popolazione totale P iniziale.

1 Vale a dire la parte integrale della forma forte dell'equazione di tipo Boltzmann, in genere scritta a secondo membro.

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

Dati: $N \in \mathbb{N}_+$, $\Delta t \leqslant 1$, σ, $T > 0$, P, $\mathbf{A}$ e $\mathbf{B}$;

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

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 $[EtB]_S^{\mathbf{A}}$ mediante Alg. 1, e quindi l'algoritmo $[AR]_{\mathbf{X}}$, si sta introducendo nei risultati un rumore di natura stocastica: piú simulazioni daranno risultati diversi per cui la singola non ha 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 $(S_1, S_2, \dots, S_R)$ di $\mathbf{S}$ sono indipendenti e identicamente distribuite dalla densità q, 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 l, 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 stima 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.

Il significato dell'intervallo di confidenza in essenza è l'errore statistico commesso: esso valuta quanto è probabile che la stima 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.

Osservazione 4.3. La stima intervallare (4.2) appena ricavata vale tanto per il vettore aleatorio $\mathbf{S}$ che per le sue realizzazioni $\mathbf{s}$, le quali sono, appunto, il risultato delle R simulazioni.

4.2.2 Struttura dei dati

I dati presentano 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 il seguente significato

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

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

2 Vale a dire μ *non* è una variabile aleatoria ma l'esatto parametro della media di l.

- $N_t \equiv \lfloor T/\Delta t \rfloor$ 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 la riduzione sono campinate 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, da cui

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

e ugualmente si assume per 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 $\underset{\sim}{\mathbf{s}} \in \mathbb{R}_+^{N_s \times 2 \times N \times R}$ delle catture:

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

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

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

vale a dire si mediano ogni Δf elementi delle N_s catture. Tale mollificazione è necessaria per rendere $\mathbf{s}$ meno rumoroso rispetto a $\underset{\sim}{\mathbf{s}}$ 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:

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

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

Osservazione 4.4. I parametri temporali N_t, N_s ed N_f non contano il tempo iniziale perché fa riferimento alla distribuzione iniziale $\mathbf{s}_0 \in \mathbb{R}_+^N$.

Ipotesi 4.3. In questa trattazione si considerano $\Delta t = 0.01$, $N_s = 1000$, $N_f = 50$ mentre N_t viene scelto per essere poco superiore al tempo di convergenza della simulazione, 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, 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 tale che $\mathbf{s}^T \equiv \underset{\sim}{\mathbf{s}}^{N_s}$.

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 gl'istogrammi. Siano N_c il numero di intervalli della griglia, ossia il numero di classi, allora definita Istogramma: $\mathbb{R}^N \to \mathbb{R}^{N_c}$ come la funzione che restituisce i valori [normalizzati] delle classi dell'istogramma, il tensore $\mathbf{h} \in \mathbb{R}_+^{2\times N_c \times R}$ si scrive

$$\mathbf{h}_{\cdot,r}^{a} \equiv \text{Istogramma}(\mathbf{s}_{\cdot,r}^{T,a}), \ \ \forall a \text{ e } \forall r,$$

Valutati gl'intervalli di confidenza

$$IC_R^{0.05}(\mathbf{h}_c^a) = \bar{h}_c^a \pm t_{R-1}^{0.025}\sqrt{\frac{\nu_c^a}{R}}, \ \ \forall a \text{ e } \forall c,$$

si possono rappresentare con $\bar{\mathbf{h}}^a$ l'istogramma medio della simulazione esatta e approssimata, e con $\pm t_{R-1}^{0.025}\sqrt{\nu_c^a/R}$ l'errore stocastico sulle stime dei valori medi delle classi.

Lognormale bimodale

Per quanto riguarda i fittaggi lognormali bimodali la logica è simile agl'istogrammi: sia

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

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

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

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 tutt'i fittaggi in un unico vettore. Allora gl'intervalli di confidenza hanno forma

$$IC_R^{0.05}(\mathcal{L}(x;\mathbf{s}^{T,a})) = \bar{\mathcal{L}}^a(x) \pm t_{R-1}^{0.025}\sqrt{\frac{\nu^a(x)}{R}}, \ \ \forall x \in [s_{min}, s_{max}] \text{ e } \forall a,$$

in cui, come prima, $\bar{\mathcal{L}}^a(x)$ è la funzione media mentre $\pm t_{R-1}^{0.025}\sqrt{\nu^a(x)/R}$ la sua incertezza stocastica.

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

Si nota, in conclusione, che i tre grafici sono in scala semilogaritmica per far emergere la distribuzione normale dalla lognormale bimodale.

Funzione di ripartizione empirica

La distribuzione di Pareto è fittata a partire dalla funzione di ripartizione complementare (FRC) empirica che quindi va prima analizzata, ma solo nella coda della distribuzione della popolazione.

Il dominio in questo caso viene definito mediante 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 di $\mathbf{s}^{T,a}_{\cdot,r}$ tale che

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

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

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

$$\mathcal{F}(x;\mathbf{s}^{a}_{\cdot,r}) \equiv 1 - \frac{1}{N}\sum_{i=1}^{N} \chi_{[0,x]}\left(s^{T,a}_{i,r}\right), \quad \forall a \text{ e } \forall r, \tag{4.3}$$

dove $\chi_{[0,x]} : \mathbb{R} \to \{0,1\}$ 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 (4.3) 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

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

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

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

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

Osservazione 4.7. In scala logaritmica la (1.2) si scrive

$$\log(R) = \log(c) - \beta \log(s),$$

che equivale a una retta traslata di $\log(c)$; indicando con R' il rapporto R/c segue

$$\log(R') \equiv \log(R) - \log(c) = -\beta \log(s),$$

per la quale il coefficiente di Pareto β è invariato.

Da quest'osservazione si può pertanto scegliere $\mathcal{F}(x;\mathbf{s}^{T,a}_{\cdot,r})$ in scala logaritmica per una maggiore leggibilità dei valori i quali altrimenti, con $\mathcal{F}(x;\mathbf{p}^{a}_{\cdot,r})$,

sarebbero bassi. 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

$$\mathcal{F}(x;\mathbf{p}^a_{\cdot,r}) + \frac{1}{2N_{1/4}}, \quad \forall a \text{ e } \forall r, \tag{4.4}$$

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 tale modifica è valida *solo graficamente* e non per il calcolo dell'indice di Pareto.

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

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

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

$$IC^{0.05}_R(\mathbf{p}^a_i) = \bar{p}^a_i \pm t^{0.025}_{R-1}\sqrt{\frac{\nu^a_i}{R}}, \quad \forall a \text{ e } \forall i.$$

Osservazione 4.8. 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.

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

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

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

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

la funzione vettoriale tale che

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

Allora gl'intervalli di confidenza hanno forma

$$IC^{0.05}_R\left(\frac{N}{N_{1/4}}\mathcal{P}(x;\mathbf{p}^a)\right) = \bar{\mathcal{P}}^a(x) \pm t^{0.025}_{R-1}\sqrt{\frac{\nu^a(x)}{R}}, \quad \forall x \in [s_{3/4}, s_{max}] \text{ e } \forall a,$$

ove $N/N_{1/4}$ è il fattore di riscalamento descritto nel § 4.2.3 necessario per confrontare l'adattamento colla FRC empirica.

Osservazione 4.9. È fittando i dati $\mathbf{p}$ alla distribuzione di Pareto che si ricava il relativo indice β, non dalla FRC empirica: ecco perché, essendo puramente grafiche, le modifiche nella (4.4) sono tollerabili.

Evoluzione della taglia media

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

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

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

$$IC_R^{0.05}(\bar{\mathbf{s}}^{n,a}) = \bar{\bar{s}}^{n,a} \pm t_{R-1}^{0.025} \sqrt{\frac{v^{n,a}}{R}}, \quad \forall n \text{ e } \forall a.$$

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

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

a istanti temporali contingui; lo stesso vale per gli estremi degl'intervalli di confidenza $\pm t_{R-1}^{0.025} \sqrt{v^{n,a}/R}$, che quindi diventano un fascio di confidenza.

Taglie medie vs gradi

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

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

allora 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.

Osservazione 4.10. In questo caso non si disegnano gl'intervalli di confidenza erché il grafico è già molto denso e aggiungere ulteriori barre verticali/orizzontali lo renderebbe difficilmente leggibile.

Evoluzioni delle taglie medie delle classi dei gradi

Sia $\mathcal{K} \subset \{1,2,\ldots,N\}$ l'insieme dei gradi distinti di $\mathbf{k}$, 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 $\mathcal{I}_k$ viene dalla (3.32) e 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 tra tutte le simulazioni.

Si conclude notando che non si disegnano gl'intervalli di confidenza per la medesima ragione chiarita nell'Oss. 4.10.

4.2.4 Sulle fluttuazioni γ

Nelle [RE] 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 (3.26) le uniche perturbazioni simmetriche sono per loro natura piccole, essendo concentrate in un intorno dell'origine.

In effetti la Fig. 4.1 conferma che γ introduce solo una lieve incertezza sui risultati, simulati coi parametri nella Tab. 4.1 e colla regola d'interazione

$$E(s,s_*,i,i_*) \equiv (1-\zeta)\lambda \frac{\frac{(s_*/s)(w_{i_*}/w_i)}{\alpha}}{1+\frac{(s_*/s)(w_{i_*}/w_i)}{\alpha}} + \zeta\lambda \frac{\frac{(s/s_*)(w_i/w_{i_*})}{\alpha}}{1+\frac{(s/s_*)(w_i/w_{i_*})}{\alpha}}, \tag{4.5}$$

che è spiegata implicitamente nel corso del § 4.3 ed è brevemente discussa piú nel dettaglio nel § 4.4.1.

	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

Tabella 4.1: Parametri della Fig. 4.1.

Dunque γ introduce solo del lieve rumore che, sebbene abbia il nobile scopo di rendere piú realistico il modello, non aiuta a interpretare i risultati rispetto a quelli reali.

4.3 REGOLE D'EMIGRAZIONE

In questo paragrafo si specializza la regola d'emigrazione E definita nelle [RE] per completare le $[RI]_S$; tutte le forme qui presenti condividono comunque tre aspetti: sono simmetriche, non lineari e associano i microstati (i,s) e (i_*,s_*) rispettivamente alla 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 fitta 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 adattati dell'indice di Pareto e della lognormale bimodale.

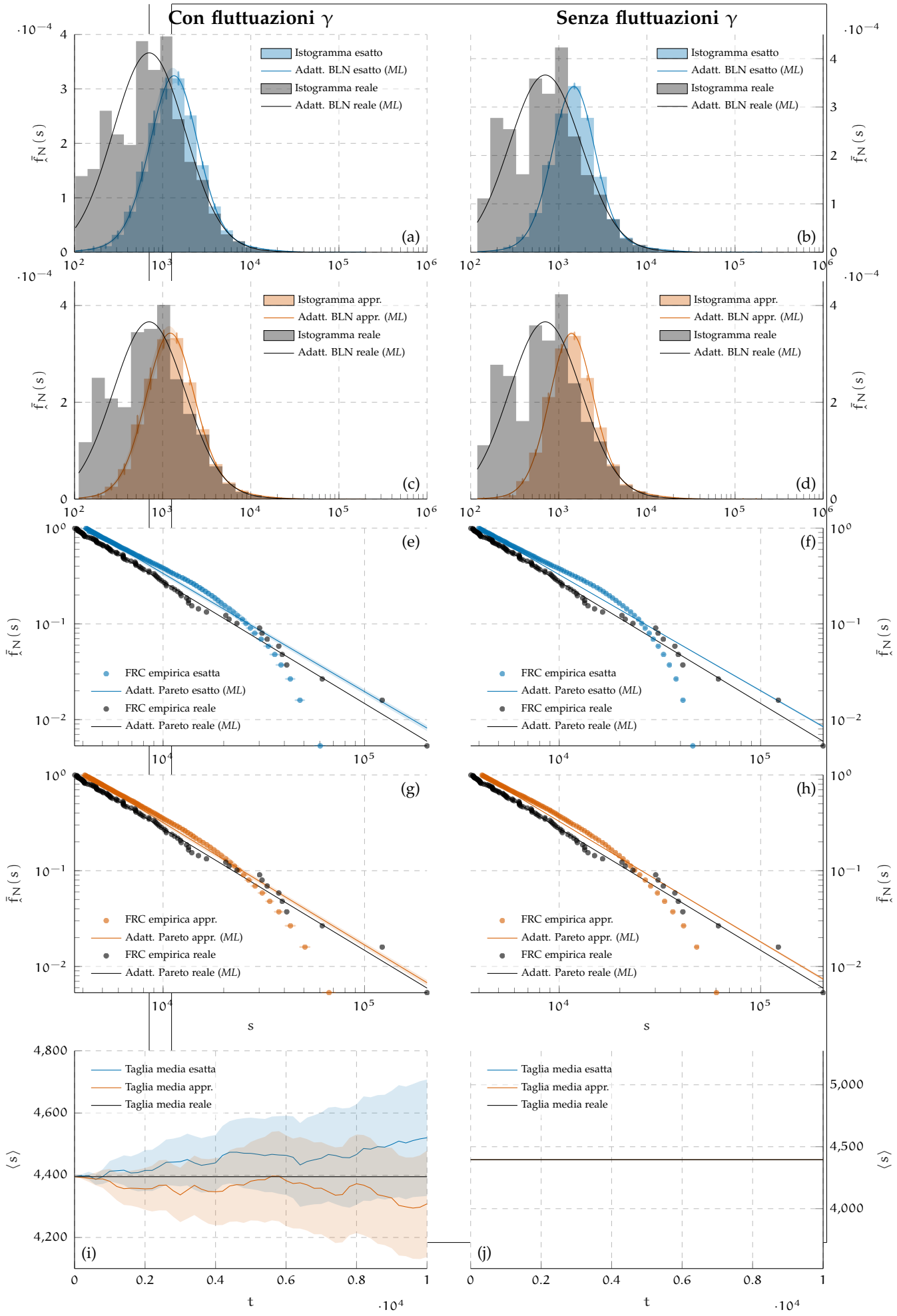


Figura 4.1: Confronto di una simulazione con e senza fluttuazioni; la regola d'emigrazione è la (4.5) mentre i parametri sono illustrati nella Tab. 4.1.

N	P	β	ξ
375	$\approx 1.65 \times 10^6$	1.27	0.93

Tabella 4.2: Dati e parametri della Sardegna.

Cionnonostante, nei §§ 4.4.2 e 4.4.3 si studiano comunque la Valle d'Aosta e l'Italia essendo casi estremi rispetto al numero di nodi.

4.3.1 Regola taglia

Una prima possibilità consiste nella

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

in cui $\lambda \in (0,1)$ e $\alpha \in \mathbb{R}^+$; in essenza è la [10, (2.2), § 2, p. 223] modificata mediante la [10, (4.5), § 4, p. 228], ossia è una funzione di Hill di ordine α, in cui v'è un tasso di emigrazione maggiore verso città con popolazione relativa, data dal rapporto s_*/s, maggiore.

I risultati nella {§}, ottenuti tramite i parametri nella Tab. 4.3, non sono purtroppo corretti dato che non solo mostrano dei centri abitati spopolarsi, ma che questi sono piú per la maggior parte i le città piú popolate.

λ	α	N_t
1	1	1

Tabella 4.3: Dati e parametri della $[\mathrm{RE}]_T$.

4.3.2 Regola taglia-gradi

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

λ	α	N_t
1	1	1

Tabella 4.4: Dati e parametri della $[\mathrm{RE}]_{TD}$.

4.3.3 Regola frazionata

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

4.3.4 Regola taglia-forza

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

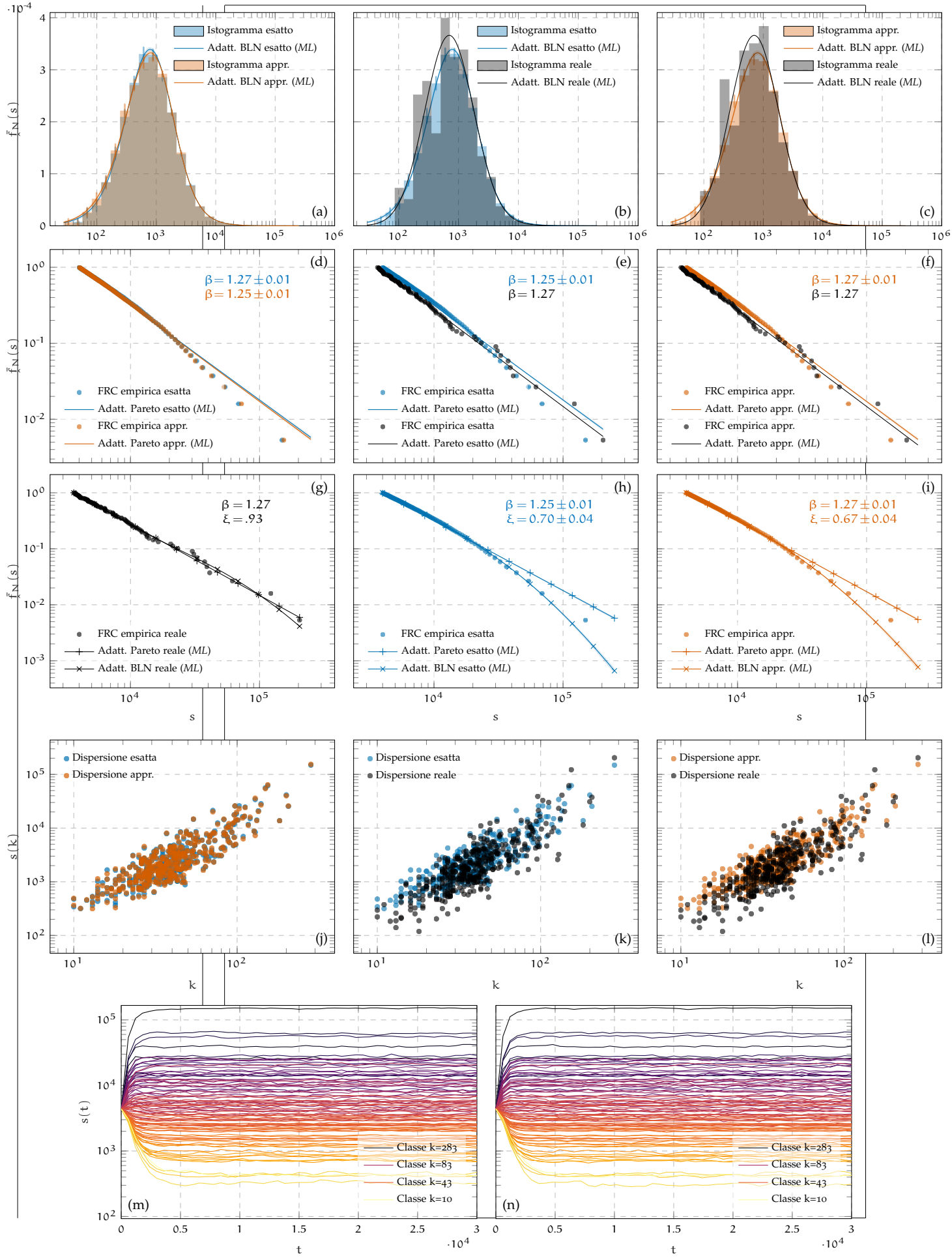


Figura 4.2: Studio della configurazione di riferimento della $[\mathrm{RE}]_{\mathrm{TF}}$ con parametri dalla Tab. 4.6; per la spiegazione dei grafici si veda il § 4.2.3.

λ	α	ζ	N_t
1	1	0.1	1

Tabella 4.5: Dati e parametri della $[RE]^f_{TD}$.

λ	α	N_t
1	1	1

Tabella 4.6: Dati e parametri della $[RE]_{TF}$.

4.3.5 Interpretazioni

In poche parole la ($[RE]_T$) descrive la tendenza degl'individui di aggregarsi per i piú svariati motivi: lavoro, sicurezza, famiglia, eccetera.

I due principali parametri del modello possono quindi essere cosí interpretati:

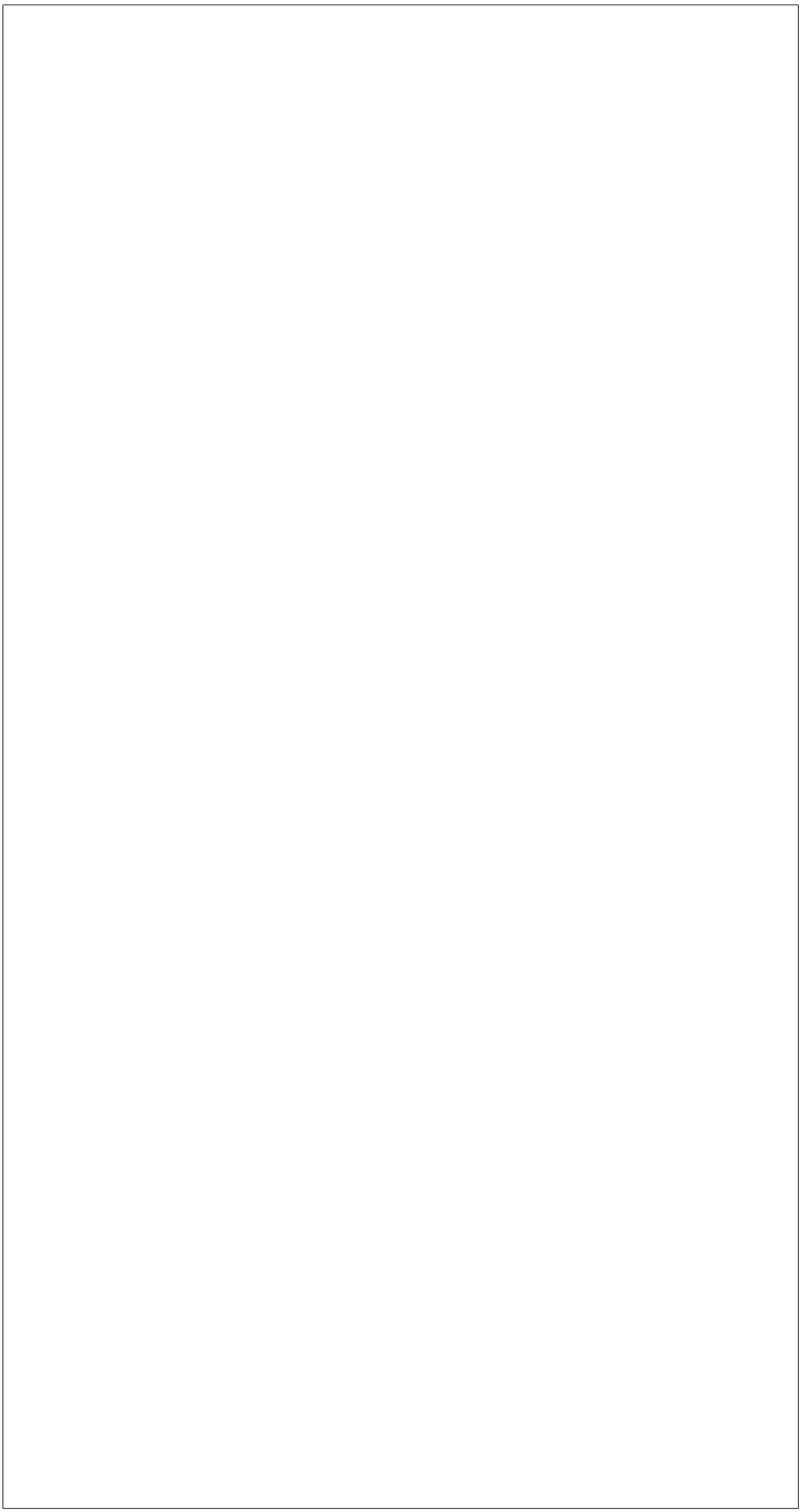
- $\lambda \in (0,1)$ rappresenta l'attrattività dei poli, ossia la frazione che le città piú popolose riescono al massimo ad attrarre in un'interazione;
- $\alpha \in \mathbb{R}^+$ indica la rapidità d'emigrazione e influenza quanto rapidamente il rapporto s_*/s raggiunge la massima attrattività λ.

4.4 ALTRI CASI NOTEVOLI

4.4.1 Varianti delle regole d'interazione

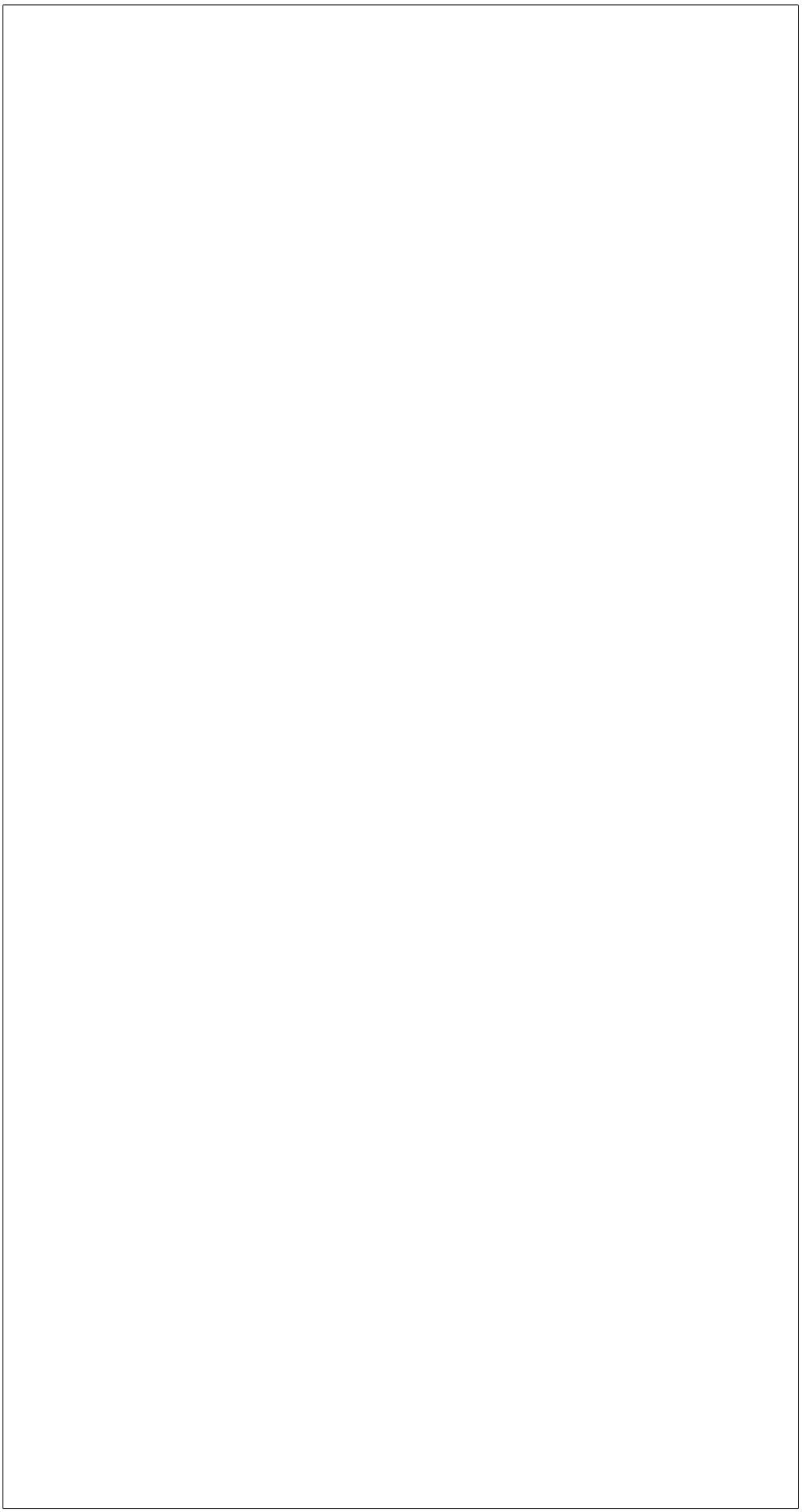
4.4.2 Il caso dell'Italia

4.4.3 Il caso della Valle d'Aosta



5 CONCLUSIONI

1. Maggiori sviluppi teorici, specialmente per ricavare un'equazione di Fokker-Planck per la distribuzione stazionaria.
2. Maggiori sviluppi pratici per abbandonare l'ipotesi semplificativa della rete statica.
3. Applicare questa teoria anche a reti europee o internazionali.



APPENDICE

A CODICE

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:
            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()
```

```
58
59          # clsKS.ShowFig()
60
61  if __name__ == "__main__":
62      freeze_support()
63      main()
```

Listato A.2: Codice «libParameters.py».

```
1   # Library to more easily define/change parameters
2
3   from dataclasses import dataclass
4   from typing import Any
5
6   from copy import deepcopy
7   import numpy as np
8
9   from pathlib import Path
10
11
12  ### Module attributes ###
13
14  mainFolder     = Path(__file__).resolve().parent
15  projectFolder  = mainFolder.parent
16  dataFolder     = projectFolder/'Dati'
17
18  matrixZipPath  = dataFolder/'MatriciPendolarismo1991.zip'
19  sizeZipPath    = dataFolder/'CensimentoRegioni1991.zip'
20  coordZipPath   = dataFolder/'LimitiRegioni1991.zip'
21  shpFilePath    = f"zip://{coordZipPath}!Limiti1991_g/Com1991_g/Com
    ↪ 1991_g_WGS84.shp"
22
23  regDataZipPath = dataFolder/'DatiRegioni1991.zip'
24  simDataZipFile = dataFolder/'DatiSimulazione.zip'
25
26  parameters = {
27      "population": {
28          "text": "S",
29          "val": 1
30      },
31      "attractivity": {
32          "text": "λ",
33          "val": 1.0
34      },
35      "convincibility": {
36          "text": "α",
37          "val": 1.0
38      },
39      "deviation": {
40          "text": "σ",
41          "val": 1.0
42      },
43      "region": {
44          "text": "Region selected",
45          "val": "region"
46      },
47      "zetaValue": {
48          "text": "ζ",
49          "val": 1.0
50      },
51      "timestep": {
52          "text": "Δt",
53          "val": 1.0
54      },
55      "timesteps": {
56          "text": "Nt",
57          "val": 1
58      },
```

```
        "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
        },
        "interactingRule": {
            "text": "Interacting law",
            "val": "law"
        },
        "PdfPopUp": {
            "text": "Open PDF",
            "val": False
        },
        "LaTeXConversion": {
            "text": "LaTeX 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": .001, # .18
                "convincibility": 1, # .44
                "deviation": 0.05,
                "region": 19,
                "zetaValue": 0.1,
                "timestep": 0.01,
                "timesteps": int(2e8),
                "iterations": 1,
                "progressBar": False,
                "extraction": False,
                "analysis": False,
                "edgeWeights": False,
                "fluctuations": False,
                "zetaFraction": False,
                "interactingRule": 0,
                "PdfPopUp": False,
                "LaTeXConversion": False,
                "snapshots": 1000,
                "smoothingFactor": 50,
                "studiedParameter": 1,
                "startValuePrmStudy": 4e0,
                "endValuePrmStudy": 1e1,
                "numberPrmStudy": 10,
                "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,
                "interactingRule": 1,
                "PdfPopUp": False,
                "LaTeXConversion": 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,
                "interactingRule": 1,
                "PdfPopUp": False,
                "LaTeXConversion": 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,
                "interactingRule": 2,
                "PdfPopUp": False,
                "LaTeXConversion": 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,
                "interactingRule": 3,
                "PdfPopUp": False,
                "LaTeXConversion": 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
        val: Any = None
        var: Any = None
        lbl: Any = None
        wid: Any = None
        frame: Any = None
        list: Any = None
        cbid: Any = None

    class Parameters():
        def __init__(self,**kwargs):
            for text,value in kwargs.items():
                setattr(self,text,value)

    class ComboBoxList():
        def __init__(self,attribute,list):
            attribute.list = list
            self.list = list
            self.code = {
                r:i for i,r in enumerate(list)
            }

    def CopyWorkerShMTemplate(): return deepcopy(workersShMTemplate)
```

Listato A.3: Codice «libData.py».

```
# Library to handle [especially writing and reading] data

import zipfile as zf
from zipfile import ZipFile as ZF

from io import TextIOWrapper as tiow
from io import BytesIO as bio
from io import StringIO as sio

import pandas as pd
import csv,json
import geopandas as gpd

import numpy as np

import libParameters as libP


### Main functions ###

def ExtractRegionData():
    dictMun,dictReg = ReadMunRegCodes()
```

```

    BuildAdjacencyMatrices(dictMun,dictReg)
    BuildSizeDistributions(dictReg)

    WriteRegionData(dictReg)

def LoadRegionData(code):
    el = {
        'A':'.txt',
        'W':'.txt',
        'li2Name':'.json',
        'li2Coord':'.txt',
        'name2li':'.json',
        'sizeDistr':'.txt',
        'Nc':'.json'
    }
    parameters = {}

    with ZF(libP.regDataZipPath) as z:
        for data,ext in el.items():
            path = f'{code:02d}/{data}{ext}'
            with z.open(path,'r') as f:
                match data:
                    case 'A' | 'W' | 'li2Coord' | 'sizeDistr':
                        parameters[data] = np.loadtxt(
                            tiow(f,encoding='utf-8'),
                            delimiter=",",dtype=int
                        )

                    case 'li2Name':
                        parameters[data] = {
                            int(k): v for k, v in
                            ↪ json.load(f).items()
                        } # 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

                    case 'Nc' | 'name2li':
                        parameters[data] = json.load(f)

    return libP.Parameters(**parameters)


### Auxiliary functions ###

def xls2csv(
    xlsFile,
    zipFile=None
):
    if zipFile is None:
        b = xlsFile
    else:
        with ZF(zipFile) as z:
            b = z.read(xlsFile)  # Extract raw bytes from
            ↪ «elencom91.xls»

    streamXls = bio(b)  # Create a binary stream in RAM from the
    ↪ bytes «b»
    df = pd.read_excel( # Read the Excel content
        streamXls,      # from «streamXLS»
        dtype=str,      # treating columns as strings
        engine="xlrd",  # using «xlrd»
        sheet_name=0    # for only the first worksheet
    ) # «df» stands for «DataFrame» from «pandas.DataFrame»
    streamCsv = sio()   # Allocate a text buffer in RAM that
    ↪ behaves like a writable text file (UTF-8 encoding)
```

```
        df.to_csv(          # Serialize «df» as «.csv»
            streamCsv,      # into «streamCSV» and
            index=False     # without the DataFrame's row numbers
        )
        streamCsv.seek(0)   # Reset the text buffer cursor to the start
        ↪  so it can be read from the beginning

        return streamCsv

    def ReadMunRegCodes(): # Read Municipality-Region Codes
        # Conversion of «fileName» from the old format «.xls» to a more
        ↪  manageable «.csv»
        matrixCSV = xls2csv('elencom91.xls',libP.matrixZipPath)

        # 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
        dictMun = {} # Dictionary to link municipality codes with
        ↪  region codes
        dictReg = {
            i+1:{
                'li2Name':{},  # Dictionary to link local indices with
                ↪  the municipality name
                'li2Coord':{}, # Dictionary to link local indices with
                ↪  the municipality representative point
                'name2li':{},  # Dictionary to link local indices with
                ↪  the municipality name
                'code2li':{},  # Dictionary to link municipality codes
                ↪  with local indices
                'Nc':0  # Number of cities in a region
            } for i in range(21)
        } # The index 21 is arbitrarily associated to Italy viewed as
        ↪  the 21th region, hence its local index is actually the
        ↪  global one

        gdf = gpd.read_file(libP.shpFilePath).set_index('PRO_COM_T').g
        ↪  eometry.representative_point()
        reader = csv.reader(matrixCSV)
        next(reader)  # Skip header line containing metadata labels
        for row in reader:
            try: # If the code is not empty
                codeReg = int(row[0]) # Region code
                codeMun = row[3]      # Municipality code
                nameMun = row[4]      # Municipality name

                dictMun[codeMun] = codeReg
                # In reality «codeMun» it's more like «(Province
                ↪  code)+(Municipality code)»

                UpdateDictionary(
                    dictReg,
                    codeReg,
                    nameMun,
                    codeMun,
                    gdf
                )

                UpdateDictionary(
                    dictReg,
                    21,
                    nameMun,
                    codeMun,
                    gdf
                )

            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,
```

```
                        oMun,dMun
                    )
                except Exception:
                    continue

            # else:
            #     if oMun != dMun:
            #         prova = f'{oMun[:3]}{dMun[:3]}'
            #         try:
            #             dictMun[prova]
            #         except KeyError:
            #             print(line)

    # There are in total two majors typos in the data:
    #    1. There is a municipality of code «'022008'» which isn't listed in «elencom91.xls», raising always a KeyError
    #    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:
    #          '215   ','229   ','241   ','216   ','203   ','224   ','236   ','246   '
    #       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
    #       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

    # Therefore the if statement excludes municipalities whose codes are only partially written due to, I presume, typos from ISTAT
    # while, the try statement catches the few cases where the code does not match any municipality (only one: «022008»)

def UpdateMatrices(
    dictReg,commuters,
    oReg,dReg,
    oMun,dMun
):
    oI = dictReg[oReg]['code2li'][oMun] # Local/Global origin index
    dI = dictReg[dReg]['code2li'][dMun] # Local/Global destination index
    # The index is considered global iff «oReg=dReg=21»

    dictReg[oReg]['A'][oI,dI] = 1
    dictReg[dReg]['A'][dI,oI] = 1
    dictReg[oReg]['W'][oI,dI] += commuters
    dictReg[dReg]['W'][dI,oI] += commuters
    # The sum in «matricesReg[oReg]['W'][oI,dI] += commuters» is necessary as there are repeating origin-destination links in the dataset

def BuildSizeDistributions(
    dictReg
):
    sizeDistribution = {}

    with ZF(libP.sizeZipPath) as z:
        Nc = dictReg[21]['Nc']
        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",
```

```
                                delimiter=","
                            )

                        case '.json':
                            json.dump(
                                dictReg[r+1][data],
                                buf,
                                indent=3 if data != 'Nc' else 0
                            )

                    z.writestr(path,buf.getvalue())

    def WriteSimulationData(
        vrtState,
        snapshots,
        siVrtState,
        typ,
        lbl,
        li2Name,
        Ni,
        sid
    ):
        lbl = [t.replace('.','') for t in lbl]

        mode = 'a' if sid is not None and sid != 1 else 'w'
        with ZF(
            libP.simDataZipFile,mode,
            compression=zf.ZIP_DEFLATED,
            compresslevel=9
        ) as z:
            for r in range(Ni):
                dicName2SortedPop = {
                    t:{
                        li2Name[i]:vrtState[r,i,t] for i in
                        ↪  siVrtState[r,::-1,t]
                    } for t in typ
                }

                folder = (
                    f'{'' if sid is None else f's{sid}/'}'
                    f'{0 if r+1<10 else ""}{r+1}'
                )
                for t in typ:
                    buf = bio()
                    path = f'{folder}/{lbl[t]}CitySizesFinal.npy'
                    np.save(buf,vrtState[r,:,t])
                    z.writestr(path,buf.getvalue())

                    buf = sio()
                    path = f'{folder}/{lbl[t]}CitySizesSorted.json'
                    json.dump(dicName2SortedPop[t],buf,indent=3)
                    z.writestr(path,buf.getvalue())

                    buf = bio()
                    path = f'{folder}/{lbl[t]}Snapshots.npy'
                    np.save(buf,snapshots[r,:,:,t])
                    z.writestr(path,buf.getvalue())

    def SetParameters(cls):
        parameters = libP.parameters
        for name,kwargs in parameters.items():
            setattr(cls,name,libP.Parameter(**kwargs))

    def LoadCaseStudies(cls):
        data = libP.caseStudies

        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                 ('interactingRule','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
```

Listato A.4: Codice «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.LaTeXConversion = clsPrm.LaTeXConversion
31 
32          for ext in ('.pdf','.mat','.tex'):
33              if ext=='.pdf' or self.LaTeXConversion:
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=0.2
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.LaTeXConversion:
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)
```

```
            # Define the folder path and create all the missing ones if necessary
            return folderPath

        def FigPath(self,figName,ext):
            return (getattr(self,f'{ext[1:]}FolderPath')/figName).with_suffix(ext)

        def cmdTeX(self):
            m = self.matFilePath.as_posix()
            t = self.texFilePath.as_posix()

            cmd = (
                r'matlab -batch "mat2tex('
                fr"'{m}','fr"'{t}'"r')"'
            )
            return cmd

    # Primary/Basic plots
    def CreateFunctionPlot(
        x,y,
        figData,
        Ni=1,
        ta=None,
        yErr=True,
        label='',
        linestyle='-',
        linewidth=1,
        marker='None',
        hatch='None',
        color='black',
        alpha=1,
        idx='',
        ax=None
    ):
        if ax is None: ax = plt.gca()

        if Ni == 1:
            ax.plot(
                x,y if y.ndim == 1 else y.ravel(),
                label=label,
                color=color,
                linestyle=linestyle,
                linewidth=linewidth,
                marker=marker,
                markevery=x.size//10-1,
                mfc=color,
                alpha=alpha
            )
            figData['plots'][f'functionPlot{idx}'] = {
                't':'function','x':x,'y':y,
                'l':label,'c':color,'m':marker
            }

        else:
            if yErr:
                avrFunc, err95Func = EvaluateConfidenceInterval(y,ta,Ni)

                # To avoid [a possible] lower negative confidence interval, the error has to be clipped
                lowerEstimate = np.maximum(avrFunc-err95Func,0)
                yerr95 = np.array([
                    avrFunc-lowerEstimate,
                    err95Func
                ]); xerr95 = None; y = avrFunc

                if (y-yerr95[0]).min()<0: raise("Negative values")
```

```
            if (y+yerr95[1]).min()<0: raise("Negative values")

        else:
            avrFunc, err95Func = EvaluateConfidenceInterval(x,ta,Ni)

            # To avoid [a possible] lower negative confidence
            ↪  interval, the error has to be clipped
            lowerEstimate = np.maximum(avrFunc-err95Func,0)
            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]})"
        #     )
        # )

    def CreateHistogramPlot(
        x,nBins,
        figData,
        limits=None,
        xScale='lin',
        Ni=1,
        ta=None,
        label='Histogram',
        color='#808080',
        norm=True,
        alpha=1,
        idx='',
        ax=None
    ):
        def HistogramPlot(
            binEdges,
            binMidPoints,
            binAverages,
            alpha=alpha
        ):
            hgPlot = ax.hist(
                binMidPoints,
                bins=binEdges, # 'auto'
                weights=binAverages,
                # density=norm,
                color=color,
                edgecolor="none", # "black"
                label=label,
                alpha=alpha
            )

            figData['plots'][f'histogramPlot{idx}'] = {
                't':'histogram','l':label,'c':color,'a':alpha,
                'x':binMidPoints,'b':binEdges,'w':binAverages
            }

            # hgPlot[0] = heights,
            # hgPlot[1] = bin edges,
            # hgPlot[2] = patches (Rectangle objects)

            # mplcursors.cursor(hgPlot[2],hover=False).connect(
            #     "add",lambda sel: sel.annotation.set_text(
            #         f"{hgPlot[0][sel.index]:.3f}"
            #     )
            # )

        if ax is None: ax = plt.gca()

        if limits is None:
            xMin = np.min(x); xMax=np.max(x)
        else:
            xMin = limits[0]; xMax=limits[1]

        if xScale == 'lin':
            binPoints = np.linspace(
                xMin,
                xMax,
```

```
                num=2*nBins+1
            )
        else:
            binPoints = np.logspace(
                np.log10(xMin),
                np.log10(xMax),
                num=2*nBins+1
            )

        binEdges = binPoints[0::2]
        binMidPoints = binPoints[1::2]
        # binMidPoints = (binEdges[:-1]+binEdges[1:])/2 # It only works in the linear case

        if Ni == 1:
            binAverages, _ = np.histogram(
                x if x.ndim == 1 else x.ravel(),
                binEdges,density=norm
            )
            HistogramPlot(binEdges,binMidPoints,binAverages,alpha)

            return binMidPoints, binAverages
        else:
            hgData = [None]*Ni
            for r in range(Ni):
                hgData[r] = np.histogram(x[r,:],binEdges,density=norm)

            binAverages, hgErr95 = EvaluateConfidenceInterval(
                [hgData[r][0] for r in range(Ni)],ta,Ni
            )

            HistogramPlot(binEdges,binMidPoints,binAverages,alpha[0])

            # To avoid [a possible] lower negative confidence interval in the lower tail, the error has to be clipped
            lowerBinEstimate = np.maximum(binAverages-hgErr95,0)
            binErr95 = np.array([
                binAverages-lowerBinEstimate,
                hgErr95
            ])

            CreateConfidenceIntervalPlot(
                binMidPoints,
                binAverages,
                figData,
                typ='errorbar',
                yerr95=binErr95,
                color=color,
                ax=ax,
                idx=idx,
                alpha=alpha[1]
            )

            return binMidPoints, binAverages

    # Secondary/Compound plots
    def CreateLogRegressionPlot(
        x,y,
        figData,
        l='Regression',
        color='blue',
        idx='',
        ax=None
    ):
        if ax is None: ax=plt.gca()

        xp = x[y>0]; yp = y[y>0]
```

```
        logx  = np.log10(xp); logy = np.log10(yp)

        slope, intercept, _, _, _ = stats.linregress(logx,logy)
        regression = 10**(intercept+slope*logx)

        fPlot = ax.plot(
            xp,regression,
            label=l,
            color=color,
            linewidth=1
        )

        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

        for r in range(Ni):
            if bimodal:
                xi,scale1,shape1,scale2,shape2 =
                ↪  FitLognormalData(v[r,:],bimodal)
                yFData[r] = (
```

```
                    xi*stats.lognorm.pdf(xF,s=shape1,scale=np.exp(scal
                    ↪  e1)) +
                    (1-xi)*stats.lognorm.pdf(xF,s=shape2,scale=np.exp(
                    ↪  scale2))
                )
                xiData[r] = xi
            else:
                shape,loc,scale = FitLognormalData(v[r,:],bimodal)
                yFData[r] =
                ↪  stats.lognorm.pdf(xF,shape,loc=loc,scale=scale)
                xiData[r] = 1

        if Ni>1: yF = yFData; xi = xiData
        else: yF = yFData[0]; xi = xiData[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

    def CreateParetoFitPlot(
        v,
        figData,
        upperbound=None,
        yScale='lin',
        Ni=1,
        ta=None,
        label=(
            'Scatter',
            'Pareto fit (ML)' # Minimum likelihood
        ),
        color=(
            'black',
            'black'
        ),
        alpha=1,
        bimodal=False,
        idx='',
        ax=None
    ):
        if ax is None: ax=plt.gca()

        if upperbound is None:
            xF = np.logspace(
                np.log10(np.quantile(v,0.75)),
                np.log10(v.max()),
                100
            )
        else:
            xF = np.logspace(
                np.log10(np.quantile(v,0.75)),
                np.log10(upperbound),
                100
            )

```

```
        vTails  = [None]*Ni
        b       = [None]*Ni
        ccdfFit = [None]*Ni

        for r in range(Ni):
            # Select the last quarter of city sizes
            vQuarter = np.quantile(v[r,:],.75)
            vTail = v[r,v[r,:] >= vQuarter]
            vTails[r] = np.sort(vTail) # Ascending values

            # Fitted CCDF from a Pareto function
            b[r], loc, scale =
            ↪  stats.pareto.fit(vTail,floc=0,fscale=vQuarter) #
            ↪  b≈alpha
            ccdfFit[r] = stats.pareto.sf(xF,b[r],loc=loc,scale=scale) #
            ↪  Survival function

        # Empirical CCDF on the tail
        n = vTails[0].size
        ccdfEmp = (
            n-np.arange(1,n+1,dtype=float)
            +(0.5 if yScale == 'log' else 0)
        )/n # P(X⩾x)
        yS = ccdfEmp

        """
            The survival function is the Complementary Cumulative
            ↪  Distribution Function (CCDF)

            Formally it should be

                    1-np.arange(1,n+1,dtype=float)/Nc-(Nc-n)/Nc=
                    (n-np.arange(1,n+1,dtype=float))/Nc=
            (*)     [(n-np.arange(1,n+1,dtype=float))/n][n/Nc]

            hence the correct variables are:

                xS = vTails
                yS = (n-np.arange(1,n+1,dtype=float))/n*(n/Nc)
                yF = [ccdfFit[r]*(n/Nc) for r in range(Ni)]

            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.

            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
        """

        if Ni>1:
            xS = vTails
            yF = ccdfFit
        else:
            xS = vTails[0]
            yF = ccdfFit[0]
            b = b[0]

        CreateScatterPlot(
            xS,yS,
            figData,
            Ni=Ni,
            ta=ta,
```

```
            size=10,
            yErr=False,
            label=label[0],
            color=color[0],
            alpha=alpha[0],
            idx=idx,
            ax=ax
        )

        CreateFunctionPlot(
            xF,yF,
            figData,
            Ni=Ni,
            ta=ta,
            label=label[1],
            color=color[1],
            alpha=alpha[1],
            marker='+' if bimodal else 'None',
            hatch='|' if bimodal else 'None',
            idx=idx,
            ax=ax
        )

        if bimodal:
            xFl = np.log(xF)
            yFData = [None]*Ni
            Nc = v.shape[1]

            for r in range(Ni):
                xi,loc1,shape1,loc2,shape2 = FitLognormalData(v[r,:] if
                ↪  Ni>1 else v,bimodal)

                ccdfFit1 = stats.norm.sf(xFl,loc=loc1,scale=shape1)
                ccdfFit2 = stats.norm.sf(xFl,loc=loc2,scale=shape2)

                yFData[r] = xi*ccdfFit1+(1-xi)*ccdfFit2
                # yFData[r] /= yFData[r][0] # Normalization
                yFData[r] /= n/Nc # Rescaling
                # It's necessary to effectively compare it with the
                ↪  Pareto fitting

            yF = yFData if Ni>1 else yFData[0]

            CreateFunctionPlot(
                xF,yF,
                figData,
                Ni=Ni,
                ta=ta,
                label=label[2],
                color=color[1],
                alpha=alpha[1],
                marker='x',
                hatch='/',
                idx=idx+1,
                ax=ax
            )

        return b


    ### Auxiliary functions ###

    def SetTextStyle():
        plt.rcParams["mathtext.fontset"] = "stix"
        plt.rcParams["font.family"] = "serif"
        plt.rcParams["font.serif"] = ["Times New Roman"]
        plt.rcParams["font.size"] = 13
```

```
683 SetTextStyle()
684
685 def SetFigStyle(
686     xLabel=None,
687     yLabel=None,
688     xDom=None,
689     yDom=None,
690     xScale='lin',
691     yScale='lin',
692     xNotation='plain',
693     yNotation='plain',
694     ax=None,data=None
695 ):
696     if ax is None: ax = plt.gca()
697
698     if xLabel: ax.set_xlabel(xLabel)
699     if yLabel: ax.set_ylabel(yLabel)
700
701     if xDom: ax.set_xlim(xDom)
702     if yDom: ax.set_ylim(yDom)
703
704     ax.set_xscale('linear' if xScale == 'lin' else xScale)
705     ax.set_yscale('linear' if yScale == 'lin' else yScale)
706
707     if xScale == 'lin':
708         ax.ticklabel_format(style=xNotation,axis='x',scilimits=(0,
            ↪ 0))
709     if yScale == 'lin':
710         ax.ticklabel_format(style=yNotation,axis='y',scilimits=(0,
            ↪ 0))
711
712     ax.grid(True,linestyle=":",linewidth=1)
713     ax.set_axisbelow(True)
714
715     _, labels = ax.get_legend_handles_labels()
716     if any(labels):
717         ax.legend()
718         data['style']['legend'] = True
719     else:
720         data['style']['legend'] = False
721     # Create a legend iff there are labels connected to graphs
722
723     data['style']['scale'] = {'x':xScale,'y':yScale}
724     data['style']['labels'] = {'x':xLabel,'y':yLabel}
725
726 def CentreFig():
727     fig = plt.gcf()
728     manager = plt.get_current_fig_manager()
729     manager.window.update_idletasks()
730
731     # Get screen size
732     screenW = manager.window.winfo_screenwidth()
733     screenH = manager.window.winfo_screenheight()
734
735     # Save figure size in pixels
736     figW, figH = fig.get_size_inches()*fig.dpi
737
738     # Centre coordinates
739     x = int((screenW-figW)/2)
740     y = int((screenH-figH)/2)
741
742     # Move the figure window
743     manager.window.geometry(f"+{x}+{y}")
744
745 def EvaluateConfidenceInterval(
746     data,
747     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 FitLognormalData(v,bimodal):
        if not bimodal:
            shape,loc,scale = stats.lognorm.fit(v[v>0],floc=0)
            # The average is «μ=np.log(scale)» while the standard
            ↪  deviation is «σ=shape»
            return shape,loc,scale
        else:
            vp = v[v>0]
            vl = np.log(vp)

            # 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 sum
                mixture = xi*pdf1+(1-xi)*pdf2

                # Prevent log(0) errors
                epsilon = 1e-10
                return -np.sum(np.log(mixture + epsilon))

            # Initial heuristic guesses
            m0 = np.mean(vl)
            s0 = np.std(vl)
            prm0 = [
                0.8, # The first gaussian is initially the most
                ↪  important one
                m0,s0,
                m0+0.5,s0
```

```
            ] # Same standard deviation but translated average

            bounds = ( # xi must be in [0,1]
                (0,1),       # xi
                (None,None), # m1
                (1e-5,None), # s1
                (None,None), # m2
                (1e-5, None) # s2
            ) # mi does not have a bound while si>0

            # Optimization
            minimization =
            ↪  minimize(NegativeLogLikelihood,prm0,bounds=bounds)

            # Extract results
            xi,m1,s1,m2,s2 = minimization.x

            return xi,m1,s1,m2,s2

    def DataString(
        data,
        Ni=1,
        ta=None,
        head='',
        formatVal='.2f',
        formatErr='.2f',
        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
```

Listato A.5: Codice «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}}',space=False,formatVal='')],
                    [libF.DataString(np.max(di),head=r'k_{{max}}',space=False,formatVal='')],
                    [libF.DataString(kAvr,head=r'\langle k\rangle',space=False)]
                ],(0.8,0.4),
                (0,0.15)
            )


            kis = np.argsort(di) # Vector for the sorted degrees
            kit = [['City','k']] # Table for the sorted degrees
            for i in range(10):
                li = kis[-i-1]
                kit.append([li2Name[li],di[li]])
            libF.TextBlock(fig,kit,(1.175,0.5),(0.2,0.4))


            # Histogram plot
            def EstimateBinNumber(v):
                # The option «'fd'» stands for «Freedman-Diaconis» and
                # uses Numpy to calculate the optimal edges for the data
                edges = np.histogram_bin_edges(v,bins='fd')
                return len(edges) - 1
            libF.CreateHistogramPlot(di,EstimateBinNumber(di),figData.fig)

            #region
                # Scatter plot
                # scP = plt.scatter(k,Pk,label="Empirical",s=10)

                # mplcursors.cursor(scP,hover=False).connect(
                #     "add",lambda sel: sel.annotation.set_text(
                #         f"k={k[sel.index]}, P(k)={Pk[sel.index]:.3f}"
                #     )
                # )


                # # Manual fitting (ML)

                # # Estimators
                # m = (1/N)*np.sum(np.log(d)) # Estimated mean
                # s2 = (1/N)*np.sum((np.log(d)-m)**2)
                # s = np.sqrt(s2) # Estimated standard deviation

                # # Evaluation
                # def lognormalPDF(x,m,s):
                #     y = np.zeros_like(x)
                #     v = x>0
                #     C = 1/(x[v]*s*np.sqrt(2*np.pi))
                #     y[v] = C*np.exp(-(np.log(x[v])-m)**2/(2*s**2))
                #     return y

                # # Plotting
                # scF = plt.plot(
                #     x,lognormalPDF(x,m,s),
                #     label="Manual lognormal fit (ML)", # Maximum likelyhood
                #     color="red",
                #     linewidth=1
                # )
            #endregion
```

```
            # SciPy fitting (ML)
            libF.CreateLognormalFitPlot(di.reshape(1,-1),figData.fig)


            # Style
            # libF.CentreFig()
            libF.SetFigStyle(
                r"$k$",r"$P(k)$",
                # [0,300],[0,0.03],
                data=figData.fig
            )
            figData.SaveFig('DegreeDistribution')

        def WeightDistributionFig(self):
            W = self.W
            wi = self.wi
            Nc = self.Nc

            li2Name = self.li2Name
            name2li = self.name2li

            figData = self.figData
            fig = figData.SetFigs()


            # Histogram plot
            binw, binPw = libF.CreateHistogramPlot(wi,19,figData.fig,xScale='log')


            # SciPy regression
            # binw = (hgPlot[1][1:]+hgPlot[1][:-1])/2
            # binPw = hgPlot[0]
            # sc = plt.scatter(binw,binPw,c='r',s=50)

            # Fit in log-log space
            slope = libF.CreateLogRegressionPlot(binw,binPw,figData.fig)

            kAvr = np.mean(wi)
            libF.TextBlock(
                fig,[
                    [libF.DataString(np.min(wi),head=r'w_{min}',space=False,formatVal='')],
                    [libF.DataString(np.max(wi),head=r'w_{max}',space=False,formatVal='')],
                    [libF.DataString(kAvr,head=r'\langle w\rangle',space=False,formatVal='.3f')],
                    [libF.DataString(slope,head=r'\alpha',space=False,formatVal='.3f')]
                ],(0.75,0.5),
                (0,0.1)
            )


            iWu,jWu = np.triu_indices_from(W,k=1)
            values = W[iWu,jWu]
            ls = np.argsort(values)[-5:][::-1] # Vector or sorted links

            Wt = [['Connection [extracted]', 'w']]
            for l in ls:
                lir = iWu[l]; lic = jWu[l]
                Wt.append([f'{li2Name[lir]}-{li2Name[lic]}',W[lir,lic]])
            libF.TextBlock(fig,Wt,(1.3,0.65),(0.3,0.2))


            Wt = [['Connection [reference]','w']]
            for link in [
```

```
                ['CAGLIARI','SASSARI'],
                ['SASSARI','OLBIA'],
                ['CAGLIARI','ASSEMINI'],
                ['PORTO TORRES','SASSARI'],
                ['CAGLIARI','CAPOTERRA']
            ]:
                lir = name2li[link[0]]
                lic = name2li[link[1]]
                Wt.append([f'{li2Name[lir]}-{li2Name[lic]}',W[lir,lic]])
            libF.TextBlock(fig,Wt,(1.3,0.35),(0.3,0.2))


            # Style
            # libF.CentreFig()
            libF.SetFigStyle(
                r"$w$",r"$P(w)$",
                xScale="log",yScale="log",
                data=figData.fig
            )
            figData.SaveFig('WeightDistribution')

        def StrengthDistributionFig(self):
            li2Name = self.li2Name
            si = self.si

            figData = self.figData
            fig = figData.SetFigs()


            # Histogram plot
            bins, binPs = libF.CreateHistogramPlot(si,20,figData.fig,x
            ↪  Scale='log')


            # SciPy regression
            # bins = (hgPlot[1][1:]+hgPlot[1][:-1])/2
            # binPs = hgPlot[0]
            # sc = plt.scatter(binw,binPw,c='r',s=50)

            # Fit in log-log space
            v = binPs>0; v[:6] = 0
            slope = libF.CreateLogRegressionPlot(bins[v],binPs[v],figD
            ↪  ata.fig)


            kAvr = np.mean(si)
            libF.TextBlock(
                fig,[
                    [libF.DataString(np.min(si),head=r's_{min}',space=
                    ↪  False,formatVal='')],
                    [libF.DataString(np.max(si),head=r's_{max}',space=
                    ↪  False,formatVal='')],
                    [libF.DataString(kAvr,head=r'\langle
                    ↪  s\rangle',space=False,formatVal='.3f')],
                    [libF.DataString(slope,head=r'\alpha',space=False,
                    ↪  formatVal='.3f')]
                ],(0.75,0.5),
                (0,0.1)
            )


            sis = np.argsort(si) # Vector for the sorted strengths
            sit = [['City','s']] # Table for the sorted strengths
            for i in range(10):
                li = sis[-i-1]
                sit.append([li2Name[li],si[li]])
```

```
            libF.TextBlock(fig,sit,(1.175,0.5),(0.2,0.4))


            # 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]
```

```
297             sk[i] = np.sum(si[v])
298             sk[i] /= Nk[i]
299 
300         libF.CreateScatterPlot(dk,sk,figData.fig)
301 
302         # Fit in log-log space
303         slope = libF.CreateLogRegressionPlot(dk,sk,figData.fig)
304 
305         sAvr = np.mean(si)
306         libF.TextBlock(
307             fig,[
308                 [libF.DataString(np.min(si),head=r's_{min}',space=
                    ↪ False,formatVal='')],
309                 [libF.DataString(np.max(si),head=r's_{max}',space=
                    ↪ False,formatVal='')],
310                 [libF.DataString(sAvr,head=r'\langle
                    ↪ s\rangle',space=False,formatVal='.3f')],
311                 [libF.DataString(slope,head=r'\alpha',space=False,
                    ↪ formatVal='.3f')]
312             ],(0.75,0.25),
313             (0,0.1)
314         )
315 
316 
317         # Style
318         # libF.CentreFig()
319         libF.SetFigStyle(
320             r"$k$",r"$s(k)$",
321             xScale="log",yScale="log",
322             data=figData.fig
323         )
324         figData.SaveFig('StrengthVsDegree')
325 
326     def AClusteringCoefficientFig(self):
327         Nc = self.Nc
328         A = self.A
329         di = self.di
330         dk = self.dk
331         Nk = self.Nk
332 
333         figData = self.figData
334         fig = figData.SetFigs()
335 
336         G = nx.from_numpy_array(A)
337         C = nx.clustering(G)
338         Cd = np.array([C[i] for i in range(Nc)])
339 
340 
341         Ck = np.zeros_like(dk,dtype=float)
342         for i,ki in enumerate(dk):
343             v = di == ki
344             Ck[i] = np.sum(Cd[v])/Nk[i]
345         self.Ck = Ck
346 
347         #region
348             # # Manual counting
349             # Cd = np.zeros_like(d,dtype=float)
350             # for i,ki in enumerate(d):
351             #     if ki>1: # Consider only nodes with more than 2
                    ↪ neighbours
352             #         vi = A[i,:]
353             #         indeces = np.nonzero(vi)[0]
354 
355             #         Ei = 0
356             #         for n in indeces:
357             #             for m in indeces:
358             #                 if A[n,m] == 1:
```

```
            #                     Ei += 1
            #         Ei /= 2 # Divided by 2 since each edge is
            ↪  counted twice

            #         Cd[i] = 2*Ei/(ki*(ki-1))
            #     else:
            #         Cd[i] = 0

            # Ck = np.zeros_like(k,dtype=float)
            # for i,ki in enumerate(k):
            #     v = (d == ki)
            #     Nk = np.sum(v)

            #     Ck[i] = np.sum(Cd[v])/Nk
        #endregion

        # CAvr = nx.average_clustering(G)
        CAvr = Ck.mean()
        libF.TextBlock(
            fig,[
                [libF.DataString(np.min(Cd),head=r'C_{min}',space=
                ↪  False,formatVal='.3f')],
                [libF.DataString(np.max(Cd),head=r'C_{max}',space=
                ↪  False,formatVal='.3f')],
                [libF.DataString(CAvr,head=r'\langle
                ↪  C\rangle',space=False,formatVal='.3f')]
            ],(0.75,0.75),
            (0,0.075)
        )

        libF.CreateScatterPlot(dk,Ck,figData.fig,label='')

        # Style
        # libF.CentreFig()
        libF.SetFigStyle(
            r"$k$",r"$C(k)$",
            # [0,300],[0,0.8],
            data=figData.fig
        )
        figData.SaveFig('AClusteringCoefficient')

    def WClusteringCoefficientFig(self):
        A = self.A
        W = self.W

        si = self.si
        di = self.di
        dk = self.dk
        Nk = self.Nk
        Ck = self.Ck


        figData = self.figData
        fig, ax = figData.SetFigs(2)


        # Manual counting
        Cd = np.zeros_like(di,dtype=float)
        for i, ki in enumerate(di):
            if ki>1: # Consider only nodes with more than 2
            ↪  neighbours
                vi = A[i,:]
                indices = np.nonzero(vi)[0]

                Ei = 0
                for n in indices:
                    for m in indices:
```

```
                            Ei += (W[i,n]+W[i,m])*A[n,m]
                    Ei /= 2 # Divided by 2 since each edge is counted
                    ↪  twice

                    Cd[i] = Ei/(si[i]*(ki-1))
                else:
                    Cd[i] = 0


            Ckw = np.zeros_like(dk,dtype=float)
            for i, ki in enumerate(dk):
                v = di == ki
                Ckw[i] = np.sum(Cd[v])/Nk[i]

            libF.CreateScatterPlot(dk,Ckw,figData.fig1,label='',ax=ax[
            ↪  0])

            # Style
            libF.SetFigStyle(
                r"$k$",r"$C^w(k)$",
                xScale="log",
                ax=ax[0],
                data=figData.fig1
            )


            CkwRel= (Ckw-Ck)/Ck
            libF.CreateScatterPlot(dk,CkwRel,figData.fig2,label='',ax=
            ↪  ax[1])
            self.CkwRel = CkwRel

            # Style
            libF.SetFigStyle(
                r"$k$",r"$C^w_{\text{rel}}(k)$",
                xScale="log",yScale="log",
                ax=ax[1],
                data=figData.fig2
            )


            # libF.CentreFig()
            figData.SaveFig('WClusteringCoefficient')

        def AAssortativityFig(self):
            A = self.A
            dk = self.dk

            figData = self.figData
            fig = figData.SetFigs()

            G = nx.from_numpy_array(A)
            ad = nx.average_neighbor_degree(G)
            ak = nx.average_degree_connectivity(G)

            aAvr = np.mean(list(ad.values()))
            libF.TextBlock(
                fig,[
                    [libF.DataString(np.min(list(ad.values())),head=r'
                    ↪  k_{nn}^{min}',space=False,formatVal='.3f')],
                    [libF.DataString(np.max(list(ad.values())),head=r'
                    ↪  k_{nn}^{max}',space=False,formatVal='.3f')],
                    [libF.DataString(aAvr,head=r'\langle k_{nn}\rangle
                    ↪  ',space=False,formatVal='.3f')]
                ],(0.75,0.75),
                (0,0.075)
            )

```

```
            knn = np.array([ak[ki] for ki in dk])
            libF.CreateScatterPlot(dk,knn,figData.fig,label='')
            self.knn = knn


            # Style
            # libF.CentreFig()
            libF.SetFigStyle(
                r"$k$",r"$k_{nn}(k)$",
                # [0,300],[40,95],
                data=figData.fig
            )
            figData.SaveFig('AAssortativity')

        def WAssortativityFig(self):
            W = self.W
            dk = self.dk
            knn = self.knn

            figData = self.figData
            fig, ax = figData.SetFigs(2)

            Gw = nx.from_numpy_array(W)
            # adw = nx.average_neighbor_degree(Gw,weight="weight")
            akw = nx.average_degree_connectivity(Gw,weight="weight")


            knnw = np.array([akw[ki] for ki in dk])
            libF.CreateScatterPlot(dk,knnw,figData.fig1,label='',ax=ax
            ↪  [0])

            # Style
            libF.SetFigStyle(
                r"$k$",r"$k_{nn}^w(k)$",
                xScale="log",yScale="log",
                ax=ax[0],
                data=figData.fig1
            )


            knnwRel = (knnw-knn)/knn
            libF.CreateScatterPlot(dk,knnwRel,figData.fig2,label='',ax
            ↪  =ax[1])

            # Style
            libF.SetFigStyle(
                r"$k$",r"$k_{nn,rel}^w(k)$",
                xScale="log",yScale="log",
                ax=ax[1],
                data=figData.fig2
            )


            # libF.CentreFig()
            figData.SaveFig('WAssortativity')

        def ShowFig(self):
            from matplotlib.pyplot import show
            show()
```

Listato A.6: Codice «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.interactingRule)

        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

        if clsPrm.edgeWeights:
            self.di = np.sum(clsReg.W/np.max(clsReg.W),axis=1,dtyp
            ↪  e=np.float64)
        else:
            self.di = np.sum(clsReg.A,axis=1,dtype=np.int32)
        self.idi = np.array(1.0/self.di,dtype=np.float64) # Inverse
        ↪  degrees
        # self.D = di@invdi.T

        # Exact unitary adjacency matrix
        M = clsReg.A
        self.dk = np.sum(M,axis=1,dtype=np.int32)
        self.Mdt = M*dt

        # Approximated adjacency matrix
        Mn = np.sum(M)
```

```
            self.wOdt = np.sum(M[:,:],axis=1)*dt/Mn
            self.wI = np.sum(M[:,:],axis=0)
            # M[:,:,1]  = wO@wI/Mn
            # In this case wO=wI but it's better to define them
            ↪  rigorously

            self.typ = np.array([0,1],dtype=np.int64)
            self.lblEn = ('Ext.','Apx.','Real')
            self.lblIt = (
                ('esatto','appr.','reale'),
                ('esatta','appr.','reale'),
                ('Esatto','Appr.','Reale'),
            )
            self.clr = ('#0072B2','#D55E00','#000000')
            # self.clr = ('blue','red')

            if clsPrm.parametricStudy:
                self.figData = None
                self.Nv = clsPrm.numberPrmStudy
                self.studiedPrmString = None
            else:
                self.figData = libF.FigData(clsPrm,'KS')
                self.Nv = None
                self.fw = 8 # Figure width
                self.fh = 6 # Figure height
            self.sid = None

        # Simulation
        def MonteCarloSimulation(self):
            Ni = self.Ni
            Nc = self.Nc
            p0 = self.p0

            gui = self.progressBar
            Nt = self.Nt
            Ns = self.Ns

            sid = self.sid
            Nv = self.Nv

            l = self.l
            a = self.a
            s = self.s
            z = self.z
            il = self.il
            typ = self.typ.size

            process = range(Ni)

            shmPrm = {}
            shmHandles = {}

            for key in workersShM['parameters'].keys():
                spec, shm = BuildShM(getattr(self,key))
                shmPrm[key] = spec
                shmHandles[key] = shm

            ctx = mp.get_context("spawn")

            if gui:
                for key,dtype in workersShM['gui'].items():
                    spec, shm = BuildShM(Ni=Ni,dtype=dtype)
                    shmPrm[key] = spec
                    shmHandles[key] = shm

```

```
                # 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)
                from libGUIs import ProgressBarsGUI
                bar = ProgressBarsGUI(Ni,Nt,sid,Nv,shmPrm,LoadShM)

                try:
                    with ProcessPoolExecutor(
                        max_workers=3,
                        mp_context=ctx,
                        initializer=InitializeMCSWorker,
                        initargs=(shmPrm,True)
                    ) as executor:
                        futures = {}
                        for p in range(Ni):
                            futures[p] = executor.submit(
                                MonteCarloAlgorithm,
                                p,Nc,Ns,p0,typ,
                                l,a,s,z,il,gui
                            )
                        bar.mainloop()

                finally:
                    data = [None]*Ni
                    for p, fut in futures.items():
                        data[p] = fut.result()
                    self.data = data

                    for shm in shmHandles.values():
                        shm.close(); shm.unlink()

            else:
                try:
                    with ProcessPoolExecutor(
                        max_workers=3,
                        mp_context=ctx,
                        initializer=InitializeMCSWorker,
                        initargs=(shmPrm,)
                    ) as executor:
                        data = list(
                            executor.map(
                                MonteCarloAlgorithm,
                                process,[Nc]*Ni,[Ns]*Ni,
                                [p0]*Ni,[typ]*Ni,[l]*Ni,
                                [a]*Ni,[s]*Ni,[z]*Ni,
                                [il]*Ni,[gui]*Ni
                            )
                        ); self.data = data
                finally:
                    for shm in shmHandles.values():
                        shm.close(); shm.unlink()

            self.EvaluateSimulationData()
            WriteSimulationData(
                self.vrtState,
                self.snapshots,
                self.siVrtState,
                self.typ,
                self.lblEn,
                self.li2Name,
                Ni,
                self.sid
            )

```

```
198         def EvaluateSimulationData(self):
199             Ni = self.Ni
200             data = self.data
201             Nw = self.Nw
202             ks = self.ks
203 
204             def Convolve(v):
205                 Nrw, _, Nty = v.shape # Number of rows, times and types
206 
207                 filter = np.array([float(1/ks)]*ks)
208                 sv = np.zeros((Nrw,Nw+1,Nty),dtype=float) # Smoothed
                    ↪  vector
209 
210                 for r in range(Nrw):
211                     for t in range(Nty):
212                         sv[r,0,t] = v[r,0,t].copy()
213                         conv = np.convolve(
214                             v[r,1:,t],
215                             filter,
216                             'valid'
217                         )
218                         sv[r,1:,t] = conv[::ks].copy()
219 
220                 # This functions applies the mean every ks unit of the
                    ↪  Ns snapshots
221                 return sv
222 
223             self.vrtState = np.array([data[p][0] for p in range(Ni)])
224             self.snapshots = np.array([data[p][1] for p in range(Ni)])
225             self.avrState = Convolve(np.mean(self.snapshots,axis=1))
226 
227             self.nodeAvrVrtState = np.mean(self.vrtState,axis=1)
228             self.nodeMinVrtState = np.min(self.vrtState,axis=1)
229             self.nodeMaxVrtState = np.max(self.vrtState,axis=1)
230             self.nodeSumVrtState = np.sum(self.vrtState,axis=1)
231 
232             if Ni>1:
233                 self.ta = stats.t.ppf(0.975,df=Ni-1)
234 
235                 self.itAvrVrtState = np.mean(self.vrtState,axis=0)
236                 self.itAvrSnapshots = np.mean(self.snapshots,axis=0)
237             else:
238                 self.ta = None
239 
240                 self.itAvrVrtState = self.vrtState[0,:,:]
241                 self.itAvrSnapshots = self.snapshots[0,:,:,:]
242 
243             self.itAvrConvSnapshots = Convolve(self.itAvrSnapshots)
244             self.siVrtState = np.argsort(self.vrtState,axis=1)
245             self.siAvrState = np.argsort(self.itAvrVrtState,axis=0)
246 
247         # Figures
248         def SizeDistrFittingsFig(
249             self,
250             ax=None,
251             idx=1,
252             figData=None,
253             saveFig=False
254         ):
255             Ni = self.Ni
256             ta = self.ta
257 
258             csSv = self.vrtState
259             csRv = self.realSizeDistr.reshape(1,-1)
260 
261             csAvr = self.nodeAvrVrtState
262             csMin = self.nodeMinVrtState
```

```
            csMax = self.nodeMaxVrtState
            csSum = self.nodeSumVrtState

            typ = self.typ
            # lbl = self.lblEn
            lbl = self.lblIt
            clr = self.clr

            Nf = 13 # Numbers of figures
            if figData is None:
                figData = self.figData
                fig, ax =
                ↪  figData.SetFigs(1,Nf,size=(self.fw*Nf,self.fh))
                saveFig = True

            sMaxEA = csSv.max(); sMaxR = csRv.max();
            sMinEA = csSv.min(); sMinR = csRv.min();

            sMaxER = max(csSv[:,:,0].max(),csRv.max())
            sMinER = min(csSv[:,:,0].min(),csRv.min())

            sMaxAR = max(csSv[:,:,1].max(),csRv.max())
            sMinAR = min(csSv[:,:,1].min(),csRv.min())

            nBins = np.zeros((3,),dtype=np.int32)
            for i,v in enumerate([csSv[:,:,0],csSv[:,:,1],csRv]):
                def EstimateBinNumber(v):
                    vf = v.ravel()
                    vp = vf[vf > 0]
                    vl = np.log10(vp)

                    # The option «'fd'» stands for «Freedman-Diaconis»
                    ↪  and
                    # uses Numpy to calculate the optimal edges for the
                    ↪  data
                    edges = np.histogram_bin_edges(vl,bins='fd')
                    return len(edges)-1

                for j in range(v.shape[0]):
                    nBin = EstimateBinNumber(v[j,:])
                    if nBin > nBins[i]: nBins[i] = nBin

            p = (.5,1.07); dp = (.6,.05)

            for t in typ: # t[ype]
                ### Lognormal fit ###

                # Exact-Approximated plot
                libF.CreateHistogramPlot(
                    csSv[:,:,t],
                    np.max(nBins[:2]),
                    getattr(figData,f'fig{idx}'),
                    limits=(sMinEA,sMaxEA),
                    xScale='log',
                    Ni=Ni,
                    ta=ta,
                    # label=f'{lbl[t]} histogram',
                    label=f'Istogramma {lbl[0][t]}',
                    color=clr[t],
                    alpha=(0.35,0.45) if Ni>1 else 0.35,
                    idx=t+1,
                    ax=ax[0]
                )
                xiS = libF.CreateLognormalFitPlot(
                    csSv[:,:,t],
                    getattr(figData,f'fig{idx}'),
                    limits=(sMinEA,sMaxEA),
```

```
                    xScale='log',
                    Ni=Ni,
                    ta=ta,
                    # label=f'{lbl[t]} lognormal fit (ML)',
                    label=fr"Adatt. BLN {lbl[0][t]} ($\mathit{{ML}}$)",
                    # (
                    #     fr'{lbl[t]} mean value $\langle k\rangle$',
                    #     f'{lbl[t]} lognormal fit (ML)'
                    # ),
                    color=clr[t],#(clr[t],clr[t]),
                    alpha=(1,0.15) if Ni>1 else 1,
                    bimodal=True,
                    idx=t+1,
                    ax=ax[0]
                )

                # Exact-Real and Approximated-Real plots
                libF.CreateHistogramPlot(
                    csSv[:,:,t],
                    max(nBins[t],nBins[2]),
                    getattr(figData,f'fig{idx+t+1}'),
                    limits=(sMinER,sMaxER) if t == 0 else
                    ↪  (sMinAR,sMaxAR),
                    xScale='log',
                    Ni=Ni,
                    ta=ta,
                    # label=f'{lbl[t]} histogram',
                    label=f'Istogramma {lbl[0][t]}',
                    color=clr[t],
                    alpha=(0.35,0.45) if Ni>1 else 0.35,
                    idx=1,
                    ax=ax[0+t+1]
                )
                libF.CreateLognormalFitPlot(
                    csSv[:,:,t],
                    getattr(figData,f'fig{idx+t+1}'),
                    limits=(sMinER,sMaxER) if t == 0 else
                    ↪  (sMinAR,sMaxAR),
                    xScale='log',
                    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,
                    max(nBins[t],nBins[2]),
                    getattr(figData,f'fig{idx+t+1}'),
                    limits=(sMinER,sMaxER) if t == 0 else
                    ↪  (sMinAR,sMaxAR),
                    xScale='log',
                    # 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 = libF.CreateLognormalFitPlot(
                    csRv,
                    getattr(figData,f'fig{idx+t+1}'),
```

```
                    limits=(sMinER,sMaxER) if t == 0 else
                    ↪  (sMinAR,sMaxAR),
                    xScale='log',
                    # 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]
                )

                # Real plot
                if t:
                    libF.CreateHistogramPlot(
                        csRv,
                        nBins[2],
                        getattr(figData,f'fig{idx+3}'),
                        limits=(sMinR,sMaxR),
                        xScale='log',
                        # label=f'{lbl[2]} histogram',
                        label=f'Istogramma {lbl[0][2]}',
                        color=clr[2],
                        alpha=0.35,
                        idx=2,
                        ax=ax[3]
                    )
                    libF.CreateLognormalFitPlot(
                        csRv,
                        getattr(figData,f'fig{idx+3}'),
                        limits=(sMinR,sMaxR),
                        xScale='log',
                        # 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[3]
                    )


                ### Power law vs bimodal lognormal fit log-log ###

                if t: libF.CreateParetoFitPlot(
                    csRv,
                    getattr(figData,f'fig{idx+4}'),
                    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[4]
                )
                libF.CreateParetoFitPlot(
                    csSv[:,:,t],
```

```
                    getattr(figData,f'fig{idx+4+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[4+t+1]
                )


                ### Power law fit log-log ###

                # Exact-Approximated plot
                libF.CreateParetoFitPlot(
                    csSv[:,:,t],
                    getattr(figData,f'fig{idx+7}'),
                    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[7]
                )

                # Exact-Real and Approximated-Real plots
                libF.CreateParetoFitPlot(
                    csSv[:,:,t],
                    getattr(figData,f'fig{idx+7+t+1}'),
                    upperbound=sMaxER if t == 0 else sMaxAR,
                    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=1,
                    ax=ax[7+t+1]
```

```
            )
            libF.CreateParetoFitPlot(
                csRv,
                getattr(figData,f'fig{idx+7+t+1}'),
                upperbound=sMaxER if t == 0 else sMaxAR,
                yScale='log',
                # label=(
                #     f'{lbl[2]} empirical CCDF',
                #     f'{lbl[2]} Pareto fit (ML)'
                # ),
                label=(
                    f'FRC empirica {lbl[1][t]}',
                    fr"Adatt. Pareto {lbl[0][t]} ($\mathit{{ML}}$)"
                ),
                color=(clr[2],clr[2]),
                alpha=(0.6,1),
                idx=2,
                ax=ax[7+t+1]
            )


            ### Power law fit log-lin ###

            # Exact-Approximated plot
            blS = libF.CreateParetoFitPlot(
                csSv[:,:,t],
                getattr(figData,f'fig{idx+10}'),
                upperbound=sMaxEA,
                yScale='lin',
                Ni=Ni,
                ta=ta,
                # label=(
                #     f'{lbl[t]} empirical CCDF',
                #     fr'{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[10]
            )

            # Exact-Real and Approximated-Real plots
            libF.CreateParetoFitPlot(
                csSv[:,:,t],
                getattr(figData,f'fig{idx+10+t+1}'),
                upperbound=sMaxER if t == 0 else sMaxAR,
                yScale='log',
                Ni=Ni,
                ta=ta,
                # label=(
                #     f'{lbl[t]} empirical CCDF',
                #     fr'{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=1,
                ax=ax[10+t+1]
            )
            blR = libF.CreateParetoFitPlot(
```

```
                    csRv,
                    getattr(figData,f'fig{idx+10+t+1}'),
                    upperbound=sMaxER if t == 0 else sMaxAR,
                    yScale='log',
                    # label=(
                    #     f'{lbl[2]} empirical CCDF',
                    #     fr'{lbl[2]} Pareto fit (ML)'
                    # ),
                    label=(
                        f'FRC empirica {lbl[1][2]}',
                        fr"Adatt. Pareto {lbl[0][2]} ($\mathit{{ML}}$)",
                    ),
                    color=(clr[2],clr[2]),
                    alpha=(0.6,1),
                    idx=2,
                    ax=ax[10+t+1]
                )


                libF.Text(
                    ax[7],(p[0],p[1]+(1/2-t)*dp[1]),
                    # fr'{lbl[t]:$\quad$'+
                    fr"{lbl[2][t]:$\quad$"+
                    libF.DataString(xiS,Ni,ta,r'\xi')+
                    libF.DataString(csMin[:,t],Ni,ta,r's_{{min}}')+
                    libF.DataString(csMax[:,t],Ni,ta,r's_{{max}}')+
                    libF.DataString(csAvr[:,t],Ni,ta,r'\langle
                    ↪  s\rangle')+
                    libF.DataString(
                        csSum[:,t],Ni,ta,r's_{{\Sigma}}',
                        formatVal='.2e',formatErr='.2e'
                    )+
                    libF.DataString(blS,Ni,ta,r'\beta',space=False),
                    ha='center',
                    color=clr[t]
                )

            if idx == 1:
                libF.TextBlock(
                    ax[1],[[
                        fr'$Nc={self.Nc}$',
                        fr'$R={self.R}$',
                        libF.DataString(blR,head=r'\beta',space=False),
                        libF.DataString(xiR,head=r'\xi',space=False)
                    ]],
                    p=(.5,p[1]-dp[1]/2),
                    dp=(dp[0]/1.5,dp[1])
                )

                offset = 0 if self.z>0 else 1
                p = (.5,p[1])
                dp = (1-.3*offset,dp[1])
                libF.TextBlock(
                    ax[-1],[[ '',
                        fr'$\lambda={self.l}$',
                        fr'$\sigma={self.s}$',
                        fr'$Ni={Ni}$',
                        fr'$Ns={self.Ns}$',
                    ],[
                        fr'$ζ={self.z}$',
                        '','','',''
                    ],[
                        '',
                        fr'$\alpha={self.a}$',
                        fr'$il={self.il+1}$',
                        fr'$\Delta t={self.dt}$',
                        fr'$Nw={self.Nw}$',
```

```
                ]],
                p=p,dp=dp,
                offset=offset
            )

        for f in range(Nf):
            libF.SetFigStyle(
                r'$s$',r'$\bar f_N(s)$',
                yNotation='sci' if f<3 else 'plain',
                xScale='log',
                yScale='log' if f<10 and f>3 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
            )

            if saveFig: figData.SaveFig('AverageSize')
            else: libF.Text(ax,(1.1,.5),self.studiedPrmString)

        def SizeVsDegreeFig(
            self,
            ax=None,
            idx=1,
            figData=None,
            saveFig=False
        ):
            ki = self.dk

            csSv = self.itAvrVrtState
            csRv = self.realSizeDistr

            typ = self.typ
            # lbl = self.lblEn
            lbl = self.lblIt
            clr = self.clr

            si = self.siAvrState
            li2Name = self.li2Name

            if figData is None:
                figData = self.figData
                Nf = 3 # Numbers of figures
                fig, ax =
                ↪  figData.SetFigs(1,Nf,size=(self.fw*Nf,self.fh))
                saveFig = True

            for t in typ:
                libF.CreateScatterPlot(
                    ki,csSv[:,t],
                    getattr(figData,f'fig{idx}'),
                    # label=lbl[t],
                    label=f'Dispersione {lbl[1][t]}',
                    color=clr[t],
                    alpha=0.7,
                    idx=t+1,
                    ax=ax[0]
                )

                libF.CreateScatterPlot(
                    ki,csSv[:,t],
                    getattr(figData,f'fig{t+1+idx}'),
                    # label=lbl[t],
                    label=f'Dispersione {lbl[1][t]}',
                    color=clr[t],
                    alpha=0.6,
                    idx=1,
                    ax=ax[t+1]
                )
```

```
788
789             libF.CreateScatterPlot(
790                 ki,csRv,
791                 getattr(figData,f'fig{t+1+idx}'),
792                 # label=lbl[2],
793                 label=f'Dispersione {lbl[1][2]}',
794                 color=clr[2],
795                 alpha=0.6,
796                 idx=2,
797                 ax=ax[t+1]
798             )
799
800         for f in range(3):
801             libF.SetFigStyle(
802                 r'$k$',r'$s(k)$',
803                 yScale='log',xScale='log',
804                 data=getattr(figData,f'fig{f+idx}'),
805                 ax=ax[f]
806             )
807
808         for t in typ:
809             libF.TextBlock(
810                 ax[0],[[
811                     libF.DataString(
812                         csSv[si[i,t],t],
813                         head=li2Name[si[i,t]],
814                         formatVal='.2e',
815                         space=False
816                     )
817                 ] for i in range(-1,-6,-1)],
818                 p=(.8,.35-t*.225),
819                 dp=(0,.15),
820                 ha='center',
821                 color=clr[t]
822             )
823
824         # CentreFig()
825         if saveFig: figData.SaveFig('SizeVsDegree')
826         else: libF.Text(ax[-1],(1.1,.5),self.studiedPrmString)
827
828     def SizeDistrEvolutionFig(
829         self,
830         ax=None,
831         idx=1,
832         figData=None,
833         saveFig=False
834     ):
835         snapshots = self.itAvrSnapshots
836
837         ns = self.ns
838         Ns = self.Ns
839
840         dt = self.dt
841         typ = self.typ
842
843         if figData is None:
844             figData = self.figData
845             Nf = 2 # Numbers of figures
846             fig, ax =
                ↪  figData.SetFigs(1,Nf,size=(self.fw*Nf,self.fh))
847             saveFig = True
848         ax[0].set_title('Exact'); ax[1].set_title('Approximated')
849
850         samples = 6
851         clrmap = get_cmap('inferno') # magma
852         colours = [clrmap(i/(samples-1)) for i in range(samples)]
853
```

```
            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 sim in self.KS:
                sim.MonteCarloSimulation()

        # Figures
        def SizeDistrFittingsFig(self):
            figData = self.figData
            fw = self.fw
            fh = self.fh

            nCol = 13 # 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.SizeDistrFittingsFig(
                    ax=ax[s,:],
                    idx=nCol*s+1,
                    figData=figData
                )

            figData.SaveFig('SizeDistributionFittings')

        def AverageSizeFig(self):
            figData = self.figData
            fw = self.fw
            fh = self.fh

            nCol = 1 # 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.AverageSizeFig(
                    ax=ax[s],
                    idx=nCol*s+1,
                    figData=figData
                )

            figData.SaveFig('AverageSize')

        def SizeVsDegreeFig(self):
            figData = self.figData
            fw = self.fw
            fh = self.fh

            nCol = 3 # 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.SizeVsDegreeFig(
                    ax=ax[s,:],
                    idx=nCol*s+1,
                    figData=figData
                )

            figData.SaveFig('SizeVsDegree')

        def SizeDistrEvolutionFig(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.SizeDistrEvolutionFig(
                    ax=ax[s,:],
                    idx=nCol*s+1,
                    figData=figData
                )

            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
```

```
1120
1121        spec = ShMSpec(
1122            name=shm.name,
1123            shape=a.shape,
1124            dtype=a.dtype.str
1125        )
1126
1127        return spec, shm
1128
1129    def LoadShM(spec):
1130        shm = shared_memory.SharedMemory(name=spec.name)
1131        arr = np.ndarray(
1132            spec.shape,
1133            dtype=np.dtype(spec.dtype),
1134            buffer=shm.buf
1135        )
1136        return shm, arr
1137
1138    def InitializeMCSWorker(
1139        shmPrm,
1140        gui=False
1141    ):
1142        for key in workersShM['parameters'].keys():
1143            shm, arr = LoadShM(shmPrm[key])
1144            workersShM['handles'].append(shm)
1145            workersShM['parameters'][key] = arr
1146
1147        if gui:
1148            for key in workersShM['gui'].keys():
1149                shm, arr = LoadShM(shmPrm[key])
1150                workersShM['handles'].append(shm)
1151                workersShM['gui'][key] = arr
1152
1153    def MonteCarloAlgorithm(
1154        p,Nc,Ns,p0,typ,
1155        l,a,s,z,il,gui
1156    ):
1157        Mdt = workersShM['parameters']["Mdt"]
1158        wOdt = workersShM['parameters']["wOdt"]
1159        wI = workersShM['parameters']["wI"]
1160        di = workersShM['parameters']["di"]
1161        idi = workersShM['parameters']["idi"]
1162        ns = workersShM['parameters']["ns"]
1163
1164        # 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
1165
1166        # Uniform initial state for all vertices
1167        vrtState = np.full((Nc,typ),p0,dtype=np.float64)
1168        snapshots = np.full((Nc,Ns+1,typ),p0,dtype=np.float64)
1169
1170        P = np.arange(Nc,dtype=np.int32)
1171
1172        nk = ns[1]
1173        hNc = Nc//2 # Nc half
1174        nsid = 1
1175
1176        if gui: # If ProgressGUI is required
1177            progress = workersShM['gui']['progress']
1178            elapsed = workersShM['gui']['elapsed']
1179            done = workersShM['gui']['done']
1180
1181            EvolveState(
1182                vrtState,P,
1183                Nc,hNc,nk,
1184                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)
```

```
1252            # pi = P[:hNc]; pr = P[hNc:]
1253
1254            for i in range(hNc):
1255                ii = P[i]; ir = P[i+hNc]
1256
1257                # t = 0
1258                p = Mdt[ii,ir]
1259                if p > 0:
1260                    theta = np.random.random() < p
1261
1262                    if theta == 1:
1263                        si = cs[ii,0]; sr = cs[ir,0]
1264
1265                        e = NonLinearEmigration(si,idi[ii],sr,di[ir],i
                            ↪  l,l,a,z)
1266                        ga = StochasticFluctuations(s,e) if s>0 else 0
1267
1268                        cs[ii,0] = si*(1-e+ga)
1269                        cs[ir,0] = sr+si*e
1270
1271                # t = 1
1272                p = wOdt[ii]*wI[ir]
1273                # 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()»
1274                # 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
1275                # In other words this trick does not work with the
                    ↪  adjacency matrix «Mdt[ii,ir]» which cannot be
                    ↪  computed from simpler elements
1276                theta = np.random.random() < p
1277
1278                if theta == 1:
1279                    si = cs[ii,1]; sr = cs[ir,1]
1280
1281                    e = NonLinearEmigration(si,idi[ii],sr,di[ir],il,l,
                        ↪  a,z)
1282                    ga = StochasticFluctuations(s,e) if s>0 else 0
1283
1284                    cs[ii,1] = si*(1-e+ga)
1285                    cs[ir,1] = sr+si*e
1286
1287                # 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
1288                # However its approximations Ap does not have zero
                    ↪  components by definition (it's a complete network),
                    ↪  hence that check becomes useless
1289
1290                # p = 1 if p>1 else (0 if p<0 else p)
1291                # theta = np.random.binomial(1,p)
1292
1293    @njit(cache=True)
1294    def FYDInPlaceShuffle(v,n):
1295        for i in range(n-1,0,-1):
1296            j = np.random.randint(0,i+1)
1297            tmp = v[i]
1298            v[i] = v[j]
1299            v[j] = tmp
1300
1301            # In randint i+1 is necessary to include the i-th index
1302
1303        # 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
```

```
1370
1371            # rsk = (sr/si)*(dr*idi)  # Relative population*degree ratio
1372            # irsk = (si/sr)*(di*idr) # Inverse relative
                ↪  population*degree ratio
1373            # efl                     # Actual emigration rate for the
                ↪  lumping fraction
1374            # efs                     # Actual emigration rate for the
                ↪  separation fraction
1375
1376        # 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
1377
1378    @njit(cache=True)
1379    def StochasticFluctuations(sigma,E):
1380        alpha = ((1-E)**2)/(sigma**2)
1381        beta  = (sigma**2)/(1-E)
1382
1383        # if alpha<=1:
1384        #     raise ValueError("α must be >1 to have a non-degenerate
            ↪  gamma distribution, and thus always admissible
            ↪  fluctuations")
1385
1386        ga = np.random.gamma(alpha,beta) # Initial sampling
1387
1388        return ga+E-1 # Final left translation
```

Listato A.7: Codice «mat2tex.mat».

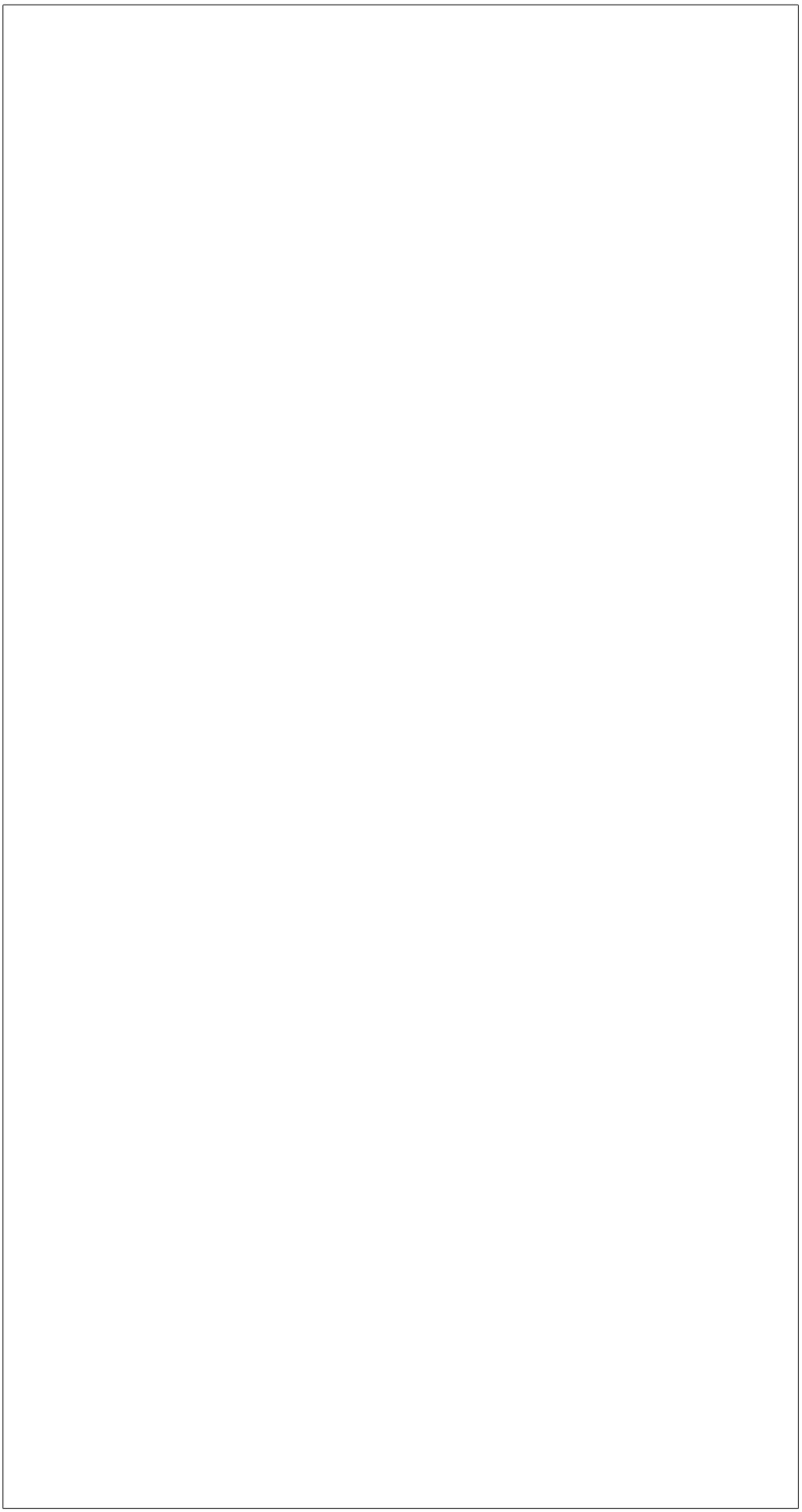
```
1   function mat2tex(matpath,texpath)
2       s = load(matpath);
3       f = string(fieldnames(s.plots));
4
5       figure('Visible','off');
6       hold on
7
8       ax = gca;
9
10      grid on
11      ax.GridLineStyle = '--';
12      ax.YMinorGrid = 'off';
13      ax.XMinorGrid = 'off';
14
15      xlabel( ...
16          s.style.labels.x,...
17          'interpreter','latex' ...
18      );
19      ylabel( ...
20          s.style.labels.y,...
21          'interpreter','latex' ...
22      );
23
24      ax.XScale = s.style.scale.x;
25      ax.YScale = s.style.scale.y;
26
27      for i = 1:numel(f)
28          field = f(i);
29
30          switch s.plots.(field).t
31              case 'scatter'
32                  h = scatter( ...
33                      double(s.plots.(field).x), ...
34                      double(s.plots.(field).y) ...
35                  );
36
37                  % h.SizeData = 16;
38                  h.SizeData = double(s.plots.(field).s)+10;
39                  h.DisplayName = s.plots.(field).l;
```

```
                    h.MarkerFaceColor = s.plots.(field).c;
                    h.MarkerFaceAlpha = double(s.plots.(field).a);
                    h.MarkerEdgeColor = "none";

                case 'function'
                    h = plot( ...
                        double(s.plots.(field).x), ...
                        double(s.plots.(field).y) ...
                    );

                    h.Color = s.plots.(field).c;

                    if(s.plots.(field).m ~= "None")
                        h.Marker = s.plots.(field).m;

                        passo = 9;
                        num_punti = numel(s.plots.(field).x);
                        h.MarkerIndices = 1:passo:num_punti;
                    end

                    if(s.plots.(field).l == "")
                        h.HandleVisibility = "off";
                    else
                        h.DisplayName = s.plots.(field).l;
                    end

                case 'histogram'
                    h = histogram( ...
                        'BinEdges',double(s.plots.(field).b), ...
                        'BinCounts',double(s.plots.(field).w) ...
                    );

                    h.DisplayName = s.plots.(field).l;
                    h.FaceColor = s.plots.(field).c;
                    h.FaceAlpha = double(s.plots.(field).a);
                    h.EdgeColor = 'None';

                    w = double(s.plots.(field).w);
                    wMin = min(w(w>0));
                    if(ax.YLim(1)>=wMin && s.style.scale.y == "log")
                        ax.YLim(1) = (10^(-0.3))*wMin;
                    end % 10^(-0.3) in logarithmic scale is a downward
                    ↪  translation of the equivalent -0.3 in linear
                    ↪  scale: log10[(10^(-0.3))*wMin]=-0.3+log10(wMin)

                case 'errorbar'
                    if(s.plots.(field).e == "y")
                        h = errorbar( ...
                            double(s.plots.(field).x), ...
                            double(s.plots.(field).y), ...
                            double(s.plots.(field).ye(1,:)), ...
                            double(s.plots.(field).ye(2,:)) ...
                        );
                    else
                        h = errorbar( ...
                            double(s.plots.(field).x), ...
                            double(s.plots.(field).y), ...
                            double(s.plots.(field).xe(1,:)), ...
                            double(s.plots.(field).xe(2,:)), ...
                            'horizontal' ...
                        );
                    end

                    h.LineStyle = "none";
                    h.HandleVisibility = "off";
                    h.Color = s.plots.(field).c;
                    h.Marker = 'none';
```

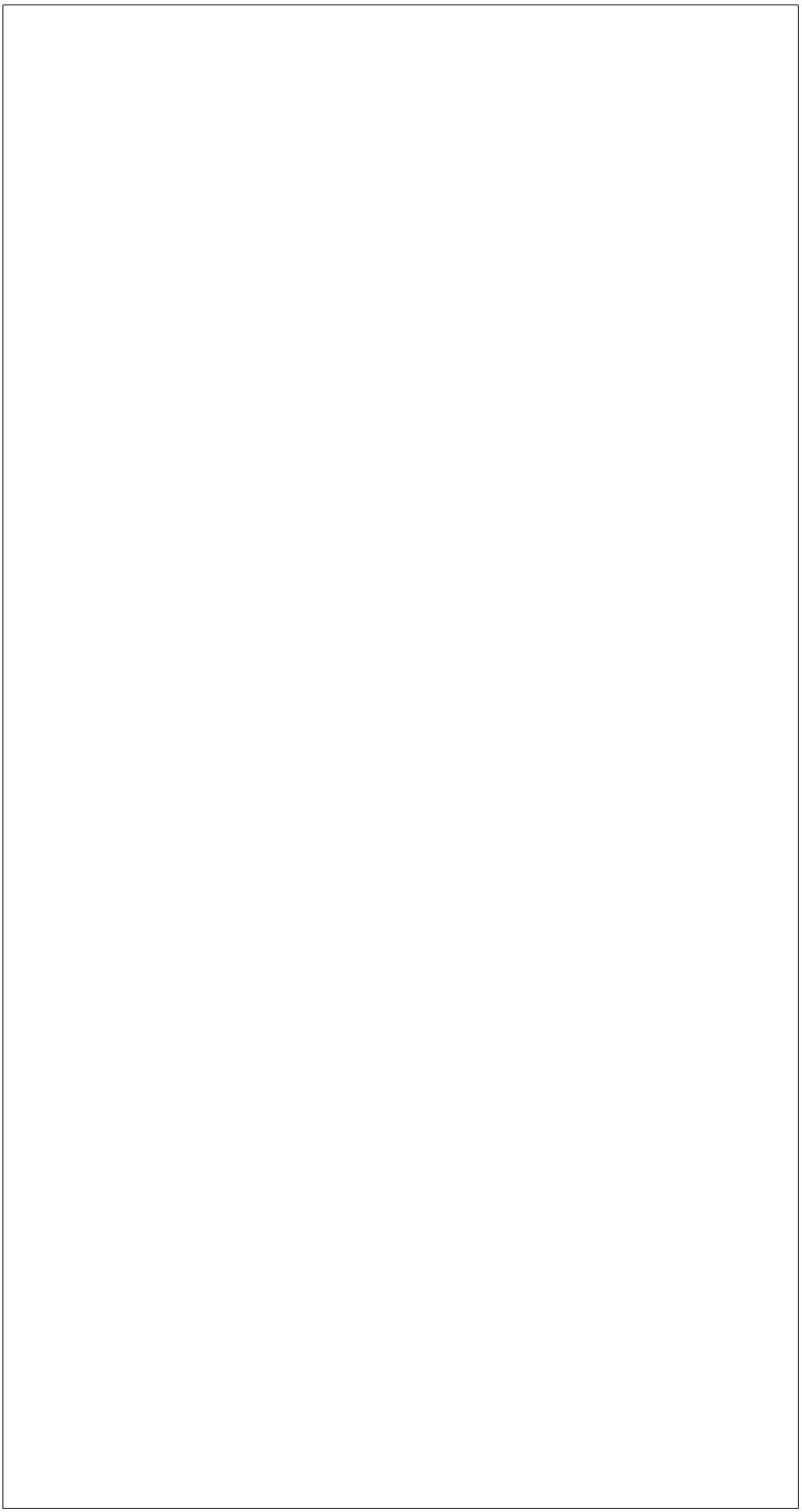
```
                    h.CapSize = 0;

                    h.UserData = sprintf('every error bar/.append style
                    ↪  ={opacity=%.2f}',double(s.plots.(field).a));

                case 'functionfill'
                    if(s.plots.(field).e == "y")
                        h = fill( ...
                            [double(s.plots.(field).x),flip(double(s.p
                            ↪  lots.(field).x))], ...
                            [double(s.plots.(field).y)+double(s.plots.
                            ↪  (field).ye(2,:)), ...
                            flip(double(s.plots.(field).y)-double(s.pl
                            ↪  ots.(field).ye(1,:)))], ...
                            sscanf(s.plots.(field).c(2:end),'%2x%2x%2x
                            ↪  ',[1 3])/255 ...
                        );
                    else
                        h = fill( ...
                            [double(s.plots.(field).x)+double(s.plots.
                            ↪  (field).xe(2,:)), ...
                            flip(double(s.plots.(field).x)-double(s.pl
                            ↪  ots.(field).xe(1,:)))], ...
                            [double(s.plots.(field).y),flip(double(s.p
                            ↪  lots.(field).y))], ...
                            sscanf(s.plots.(field).c(2:end),'%2x%2x%2x
                            ↪  ',[1 3])/255 ...
                        );
                    end

                    h.FaceAlpha = double(s.plots.(field).a);
                    h.HandleVisibility = "off";
                    h.EdgeColor = "none";

                    if(s.plots.(field).h ~= "None")
                        switch s.plots.(field).h
                            case "|"
                                h.UserData = 'postaction={pattern={Lin
                                ↪  es[angle=90,distance={5pt},line
                                ↪  width={0.5pt}]}, pattern
                                ↪  color=mycolor1!60}';
                            case "/"
                                h.UserData = 'postaction={pattern={Lin
                                ↪  es[angle=45,distance={5pt},line
                                ↪  width={0.5pt}]}, pattern
                                ↪  color=mycolor1!60}';
                        end
                    end
            end
        end

        if(s.style.legend == 1)
            legend( ...
                'Location','northeast',...
                'interpreter','latex' ...
            );
        end

        try
            cleanfigure('handle',gcf,'targetResolution', 600);
        catch
            warning('Cleanfigure skipped');
        end

```

```
154         ishistogram = isfield(s.plots,"histogramPlot") ||
            ↪  isfield(s.plots,"histogramPlot1"); % If there is at least a
            ↪  histogram plot
155 
156         if(ishistogram && s.style.scale.y == "log")
157             matlab2tikz( ...
158                 texpath, ...
159                 'showinfo',false, ...
160                 'standalone',false, ...
161                 'floatFormat','%.5g', ...
162                 'parseStrings', false, ...
163                 'extraAxisOptions', {'log origin=infty'} ...
164             ); % This correction is necessary to avoid having upside
                ↪  down bins in the «.tex» file when the y-scale is
                ↪  logarithmic
165         else
166             matlab2tikz( ...
167                 texpath, ...
168                 'showinfo',false, ...
169                 'standalone',false, ...
170                 'floatFormat','%.5g', ...
171                 'parseStrings', false ...
172             );
173         end
174     end
```

ELENCO DELLE FIGURE

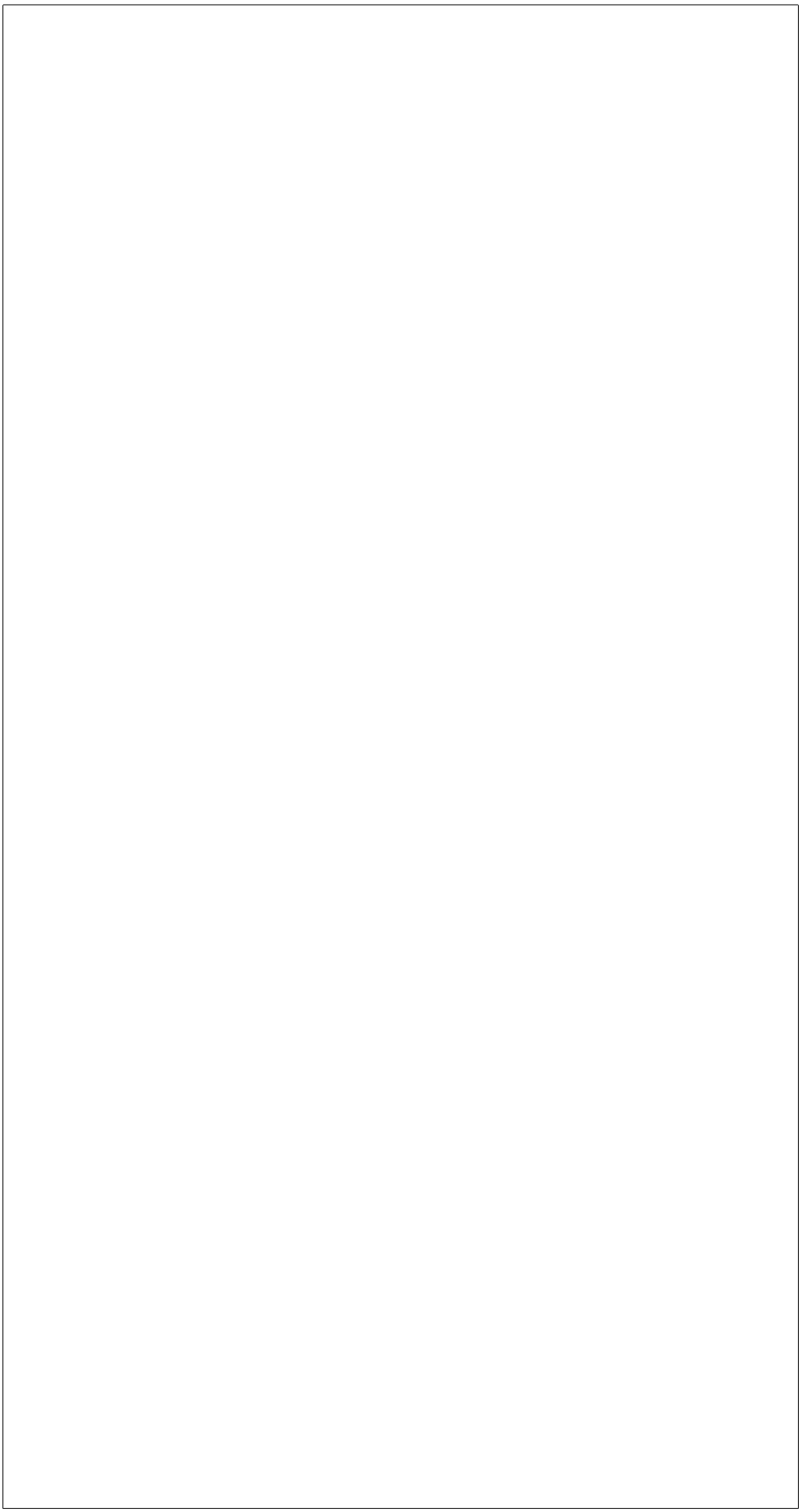


ELENCO DELLE TABELLE



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] George Kingsley Zipf. *Human behavior and the principle of least effort: An introduction to human ecology*. Ravenio books, 2016.