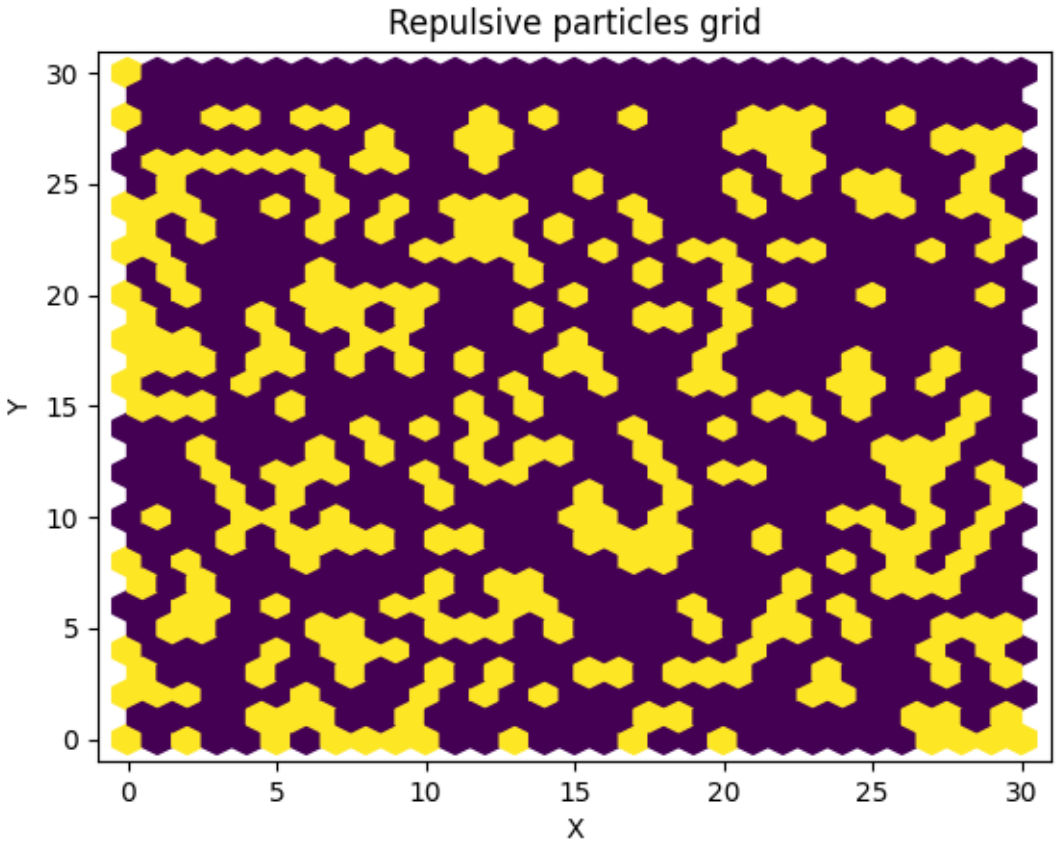
Simulazione Particelle

Dighero Tommaso

Sommario



Il programma proposto si pone l'obiettivo di simulare un settore di spazio bidimensionale nel quale vengono poste un numero N di particelle, con lo scopo di osservare come la loro distribuzione varia e si assesta nel tempo.

1 Il Problema Presentato

Il problema proposto richiede di simulare delle particelle con un comportamento repulsivo all'interno di una griglia esagonale, con condizione di continuità al bordo.

2 Teoria necessaria

Per simulare correttamente il sistema richiesto è possibile schematizzare il nostro sistema come un insieme di N particelle disposte in una griglia esagonale, questo significa che in base alla posizione di ciascuna particella sono possibili S configurazioni o stati totali nel sistema. L'obiettivo ora è quello di

utilizzare il metodo di estrazione delle catene di markov per far mutare la configurazione fino a potenzialmente raggiungere uno stato di equilibrio, che minimizza l'energia totale del sistema. Utilizzare le catene di markov ci permette di garantire innanzitutto che durante l'estrazione di un nuovo stato i in cui far passare il sistema, la probabilità di estrarlo $P(i_k/i_{k-1})$ dipende solo dal suo stato precedente (ovvero lo stato attuale del sistema), senza dover tenere conto di tutti gli step passati. Vogliamo inoltre che il nostro sistema evolva naturalmente verso uno stato di equilibrio, ovvero:

$$\lim_{m\to\infty} P^{(0)}\Pi^m = P^{eq}$$

dove Π è la matrice che codifica il passaggio da qualunque stato in qualunque altro stato, $P^{(0)}$ è lo stato di partenza e m è il numero di iterazioni (o passaggi di stato). Π deve rispettare due condizioni, una di Ergodicità, rappresentata dalla relazione:

$$(\Pi^m)_{ij} > 0 \text{ e } (\Pi^n)_{ji} > 0$$

Che esprime come dati 2 qualunque stati i e j è possibile passare da uno all'altro in un numero finito di step (m e n), garantendo l'esplorazione totale del sistema e il potenziale raggiungimento di qualunque stato a partire da uno stato randomico iniziale. L'altra condizione necessaria è di aperiodicità, espressa dalla relazione:

$$\Pi_i = 1 \text{ e quindi } \Pi_{ij} = 0 \text{ per } i \neq j$$

Ovvero assicurarsi che durante le iterazioni non ci si intrappoli in uno stato che permette il passaggio solamente da se stesso a se stesso. Entrambe le condizioni verranno garantite in pratica semplicemente programmando correttamente le iterazioni di ogni step. Queste condizioni fino a ora non garantiscono ancora che lo stato raggiunto a iterazioni infinite sia quello di equilibrio, per fare ciò dobbiamo imporre che:

$$P^{eq}\Pi = P^{eq}$$

Nonostante ciò la matrice Π non è ancora univoca, per renderla tale dobbiamo ancora imporre la condizione di bilancio dettagliato:

$$P_i^{eq}\Pi - ij = P_j^{eq}\Pi_{ji}$$

Ovvero imporre che le probabilità di essere nello stato i ed estrarre la mossa i-¿j sono uguali.

Tutte queste considerazioni in un contesto di meccanica statistica si traducono nell'imporre che:

$$P_i^{eq} = \alpha e^{-\frac{E_i}{K_b T}}$$

Dove α è una costante di normalizzazione e E_i è l'energia della configurazione i, che è calcolata come $K\sum_{<i,j>} n_i, n'_j$ ovvero una costante k moltiplicata per il numero di coppie di primi vicini.

3 Il Modello

Durante le simulazioni utilizzeremo una griglia esagonale di taglia LxL, ipotizzata a contatto con una sorgente termica che lo mantiene a temperatura costante. Nel sistema viene introdotto un numero N di particelle che non varia durante la simulazione, ogni particella possiede una specifica energia colcolata come il numero (intero) di primi vicini occupati da altre particelle, che in una griglia esagonale è $0 <= E <= 6$

4 Implementazione

Arrivando all'implementazione vera e propria dovremo fare uso dell'algoritmo di metropolis, che indica come scegliere una possibile transizione di stato e se accettare o meno lo spostamento. Useremo uno struct HexGrid che farà la maggior parte del lavoro, ma le parti più importanti sono:

```rust
pub fn initialize(&mut self) {
    for _ in 0..((self.size * self.size) as f32 * self.fillrate) as u16 {
        let mut c: bool = true;
        while c {
            let x: u16 = self.rng.gen_range(0..self.size);
            let y: u16 = self.rng.gen_range(0..self.size - 1);
            let hex: Hex = self.get_cell(x as i16, y as i16);
            if hex.value == false{
                self.set_cell(x as i16, y as i16, value: true);
                c = false;
            }
        }
    }
}
```

La funzione initialize si occupa di riempire la griglia esagonale di un numero di particelle fissato, scegliendo ogni volta una cella casuale e riempendola se è vuota, fino a che il numero desiderato di particelle non è stato raggiunto, questo rappresenterà il nostro stato di partenza

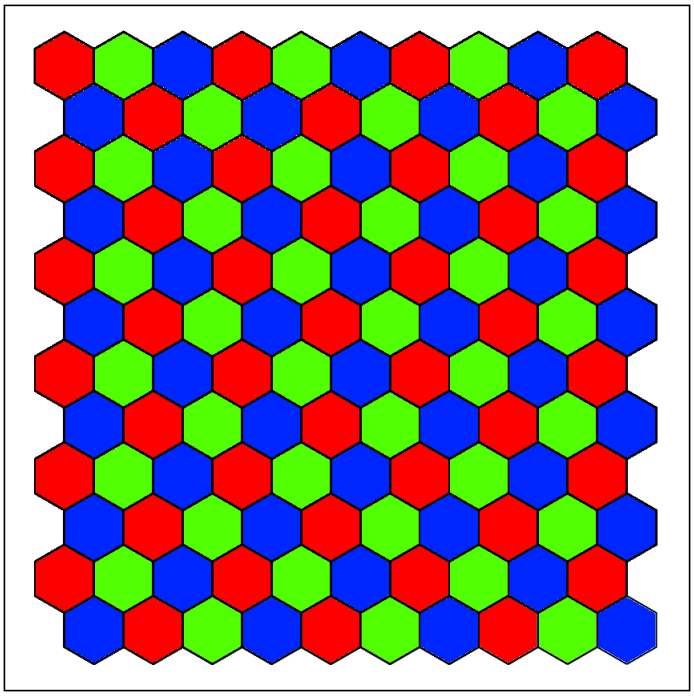
```rust
pub fn advance_timestep_repulsive(&mut self) {
    let rx: u16 = self.rng.gen_range(0..self.size);
    let ry: u16 = self.rng.gen_range(0..self.size);
    let hex: Hex = self.get_cell(x: rx as i16, y: ry as i16);
    if hex.value == true {
        let start_energy: u8 = self.get_energy(hex);
        let ne: Vec<Hex> = self.get_neighbours(x: rx as i16, y: ry as i16);
        let i: usize = self.rng.gen_range(0..ne.len());
        let dest: Hex = ne[i];
        if dest.value == false {
            let end_energy: f64 = (self.get_energy(hex: dest) as i16 - 1) as f64;

            if start_energy as f64 > end_energy {
                self.set_cell(x: rx as i16, y: ry as i16, value: false);
                self.set_cell(dest.x, dest.y, value: true);
            }

            else if start_energy as f64 <= end_energy {
                let delta: f64 = (start_energy as f64 - end_energy) as f64;
                let check: bool = self.accept_change(delta_energy: delta);
                if check == true {
                    self.set_cell(x: rx as i16, y: ry as i16, value: false);
                    self.set_cell(dest.x, dest.y, value: true);
                }
            }
        }
    }
} fn advance_timestep_repulsive
```

Questa funzione invece si occupa di far avanzare il sistema nel tempo, a ogni step (o estrazione della nostra catena) scegliamo a caso una particella del nostro sistema, ne troviamo i primi vicini e scegliamo a caso una direzione di possibile spostamento. Calcoliamo ora l'energia iniziale e finale dello spostamento e muoviamo la particella secondo l'algoritmo di metropolis. Abbiamo infine la funzione che calcola il parametro d'ordine del sistema

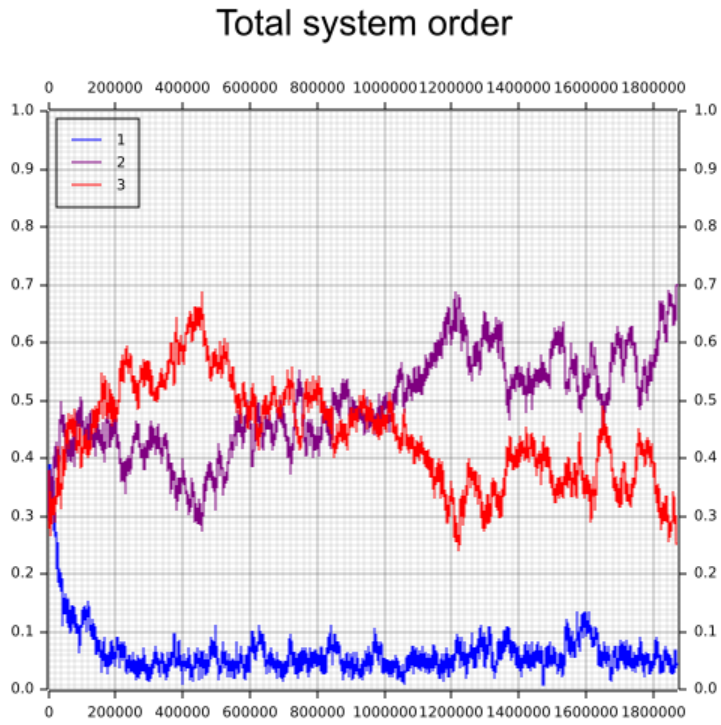
```
for i: i32 in 0..3 {
    for x: u16 in 0..self.size {
        for y: u16 in 0..self.size {
            let hex: Hex = self.get_cell(x as i16, y as i16);
            if y % 2 == 0{
                if i == 0 {
                    if x % 3 == 0 && hex.value == true{
                        count_a += 1.0;
                    }
                }
                else if i == 1 {
                    if (x as i16 - 1) % 3 == 0 && hex.value == true{
                        count_b += 1.0;
                    }
                }
                else if i == 2 {
                    if (x as i16 - 2) % 3 == 0 && hex.value == true{
                        count_c +=1.0;
                    }
                }
            }
            else if (y - 1) % 2 == 0 {
                if i == 0 {
                    if (x as i16 - 1) % 3 == 0 && hex.value == true{
                        count_a += 1.0;
                    }
                }
                else if i == 1 {
                    if (x as i16 - 2) % 3 == 0 && hex.value == true{
                        count_b += 1.0;
                    }
                }
                else if i == 2 {
                    if x % 3 == 0 && hex.value == true{
                        count_c += 1.0;
                    }
                }
```

4.1 Risultato e conclusioni

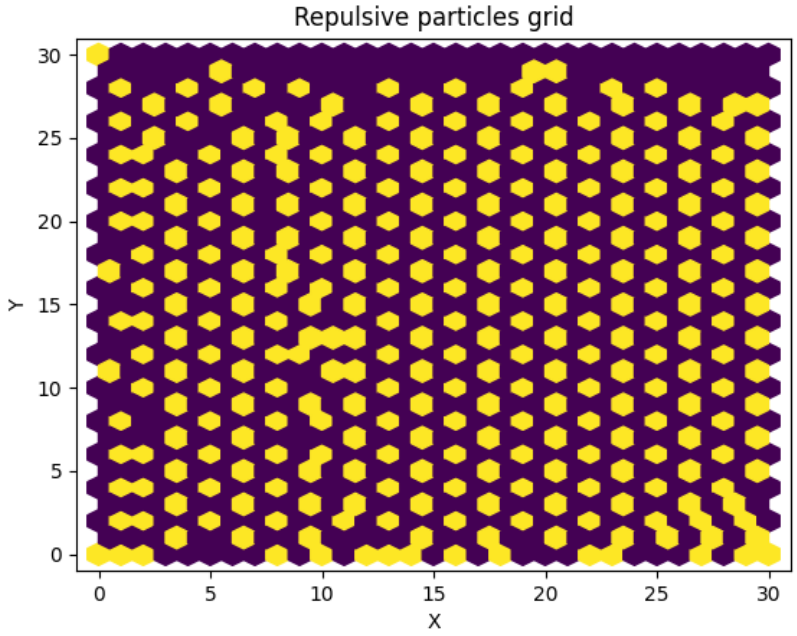
La prima importante discussione da fare riguarda il parametro d'ordine, a differenza dal caso visto in classe che presentava 2 stati di equilibrio in cui il sistema poteva finire a tempo infinito, con una griglia esagonale ne sono stati individuati 3, disposti come segue:



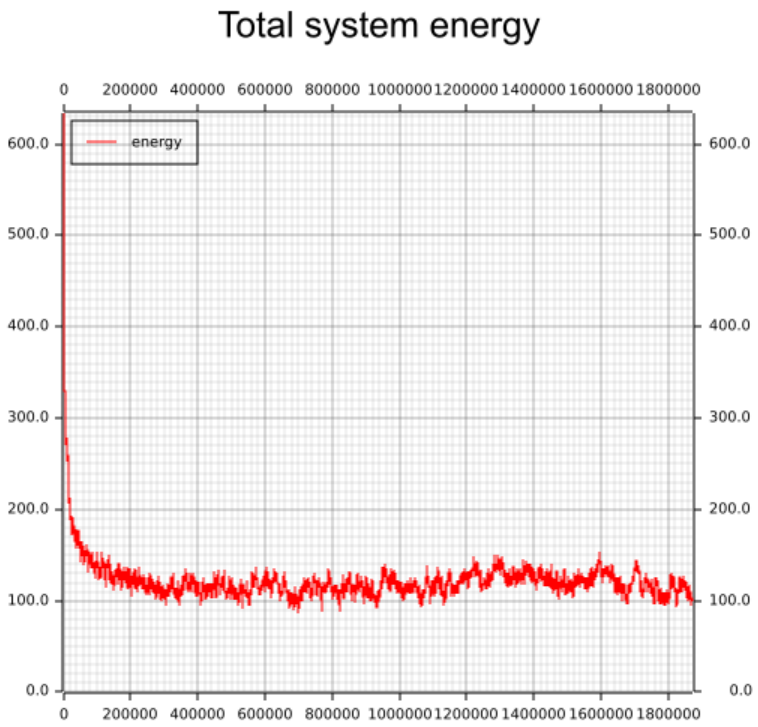
per rappresentare l'adesione del sistema in uno di questi 3 stati di equilibrio viene usato un grafico triplice, dove ogni timeseries rappresenta quanto il sistema si avvicina a uno stato di equilibrio in una scala che va da 0 a 1



Per quanto riguarda lo stato del sistema raggiunto nella stessa simulazione otteniamo:



Possiamo infine avere un'altra conferma che il mio sistema si è mosso verso uno stato di equilibrio, e di conseguenza a energia minore, osservando appunto l'andamento dell'energia totale del sistema in funzione del tempo:



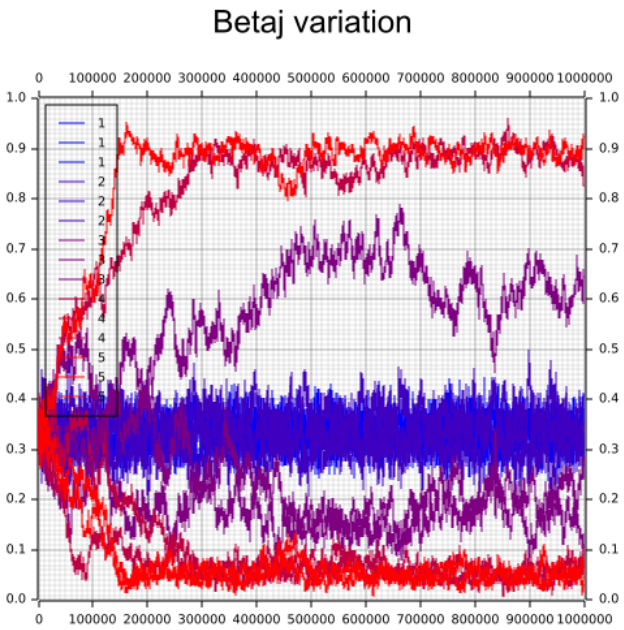
Passiamo ora a osservare alcuni risultati sperimentali della nostra simulazione

4.2 βJ Critico

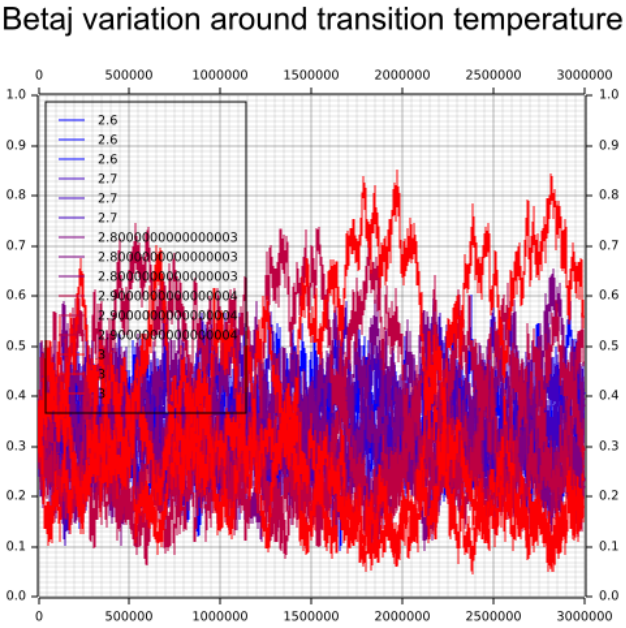
In base alla geometria del problema visto in classe avevamo stabilito che il valore βj critico è stato individuato sperimentalmente, con un valore pari a:

$$\beta J = \frac{J}{k_b T} \approx 1.7627$$

Dove J è la differenza di energia dallo stato iniziale a quello finale. nel nostro caso tuttavia questo valore non garantisce il raggiungimento di uno stato di equilibrio, sono state fatte quindi numerose prove con valori diversi per provare a stimarlo.



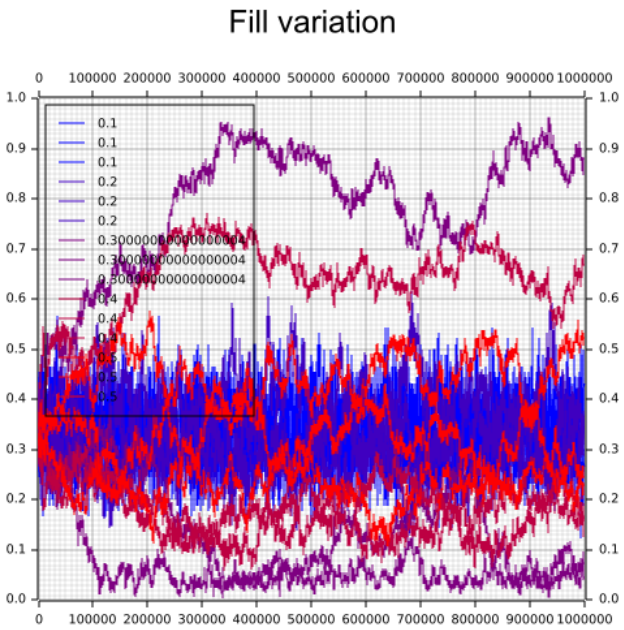
Si nota come valori inferiori a 3 non si discostano significativamente da 0.3 per ciascun parametro d'ordine, segnalando che il sistema non riesce a stabilizzarsi in una configurazione di equilibrio. Possiamo effettuare un altro test per avvicinarci meglio alla stima del valore:



possiamo concludere che il valore critico sia approssimativamente $2.8 < \beta j < 3$

4.3 Percentuale di riempimento

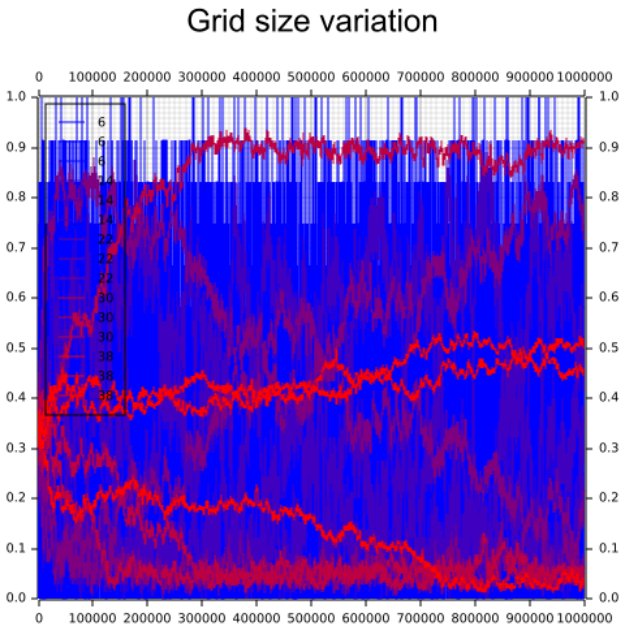
altre osservazioni che possiamo fare sono per esempio notare come la percentuale di riempimento influisca sul raggiungimento di uno stato di equilibrio:



Come precedentemente supposto la percentuale di riempimento più favorevole per il raggiungimento dell'equilibrio risulta essere proprio $\frac{1}{3}$, congruente con la divisione dello spazio negli stati di equilibrio finali possibili, mentre valori significativamente diversi da $\frac{1}{3}$ non riescono a raggiungere uno stato di ordine globale.

4.4 Dimensioni della griglia

Un'ultima osservazione che possiamo fare è come le dimensioni della griglia influiscano sul raggiungimento dell'equilibrio, qui di seguito i risultati:



L'unica conclusione traibile da questo esperimento è che, nonostante tutti i sistemi tendano a raggiungere uno stato d'ordine, griglie troppo piccole si verificano essere molto instabili nel mantenimento di un solo stato di equilibrio, mentre dimensioni molto grandi impiegano molto più tempo per raggiungere l'equilibrio.