

# Travail pratique #2

Lucas Gonzalez  
GONL29019907

<b>Présentation générale</b>	<b>2</b>
Technologies	2
Utilisation	2
Structure du projet	2
<b>Partie 1 - Création d'un générateur</b>	<b>3</b>
Présentation	3
Constructeur	3
Random Int	3
Random Float	4
Random Float Range	4
Random Int Range	4
<b>Partie 2 - Teste du générateur</b>	<b>4</b>
Condition du test	4
Résultat graphique	5
Pour 1 000 itération	5
Pour 100 000 itération	6
Test du Khi2	7
Result Khi2	7
<b>Partie 3 - Utilisation du générateur</b>	<b>8</b>
Mise en oeuvre de la marche aléatoire	8
random walk	8
non reversing walk	8
self avoiding walk	9
Validation graphique	10
random walk	11
non reversing walk	11
self avoiding walk	12
Etude end to en mean Square	12
Procédure	12
<b>Annexe</b>	<b>14</b>
Interface	14
P1 / P2 / P3	14
Graphe P3.4	14

# Présentation générale

## Technologies

Le projet utilise les technologies suivantes :

- Javascript : logique du code / algorithmie
- Html, Css : interfaces / styles
- P5js : librairie graphique simplifie l'utilisation de forme dans les canvas
- Chart.js : pour produire des graphes
- Webpack : pour faire un bundle du code (exécutable unique) (couple à Babel)

## Utilisation

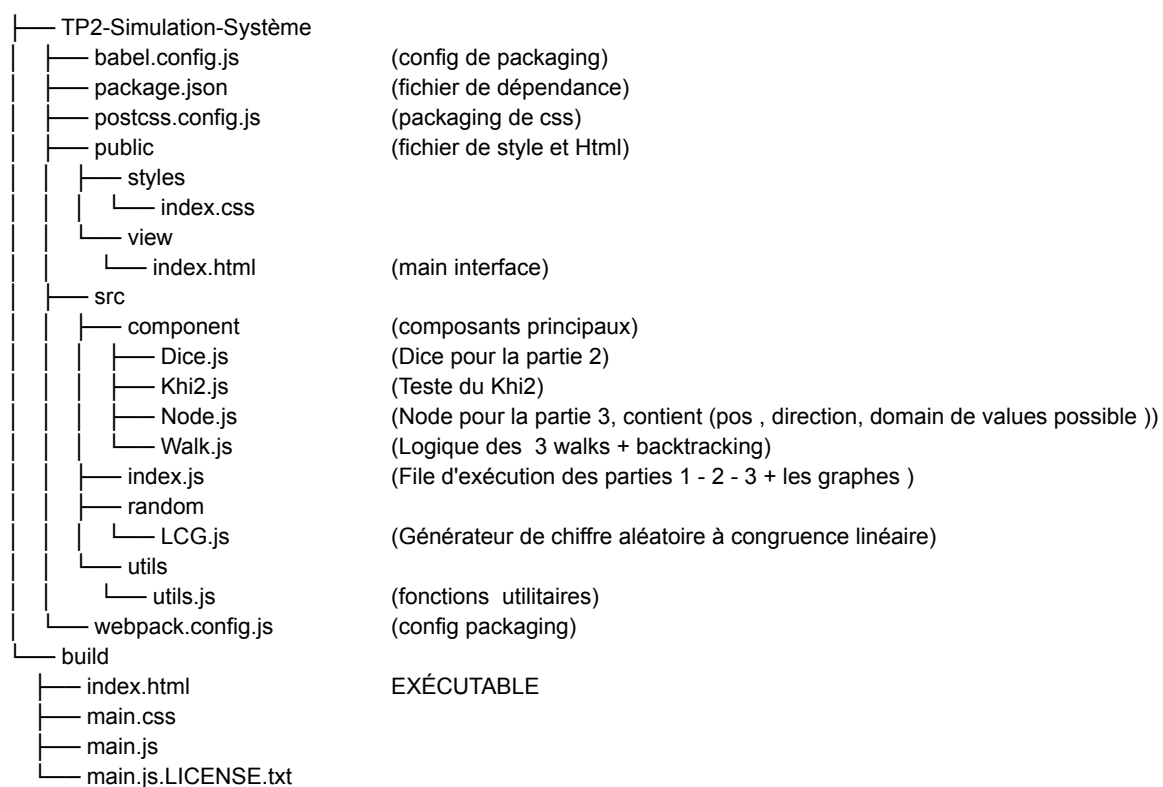
**Important : Internet est requis pour faire fonctionner l'interface (CDN bootstrap, JQuery etc...)**

Exécutable (**recommandé**) : **/build/index.html** (double click)

Dev : npm run install (racine du dossier TP2 -Simulation-Système)

Code source : **./TP2-Simulation-Système/\***

## Structure du projet



# Partie 1 - Création d'un générateur

## Présentation

Le code du générateur se trouve ici : **/random/LCG.js** (Linear congruential generator)

Les valeurs par défaut sont choisies selon les références suivantes :

- ref : [https://en.wikipedia.org/wiki/Linear\\_congruential\\_generator](https://en.wikipedia.org/wiki/Linear_congruential_generator)
- ref : [https://rosettacode.org/wiki/Linear\\_congruential\\_generator](https://rosettacode.org/wiki/Linear_congruential_generator)

Le générateur est capable de générer :

- un nombre aléatoire entre 0 et  $2^{32}$
- un float entre 0 et 1
- un float entre (min et max)
- un int entre (min et max)

## Constructeur

```
/**
 * From ref BSD formula : Seed : timeInt mod : 2**32, mult : 214013, inc : 2531011
 * @param {Object} param0
 */
constructor({ seed = timeInt(), mod = 2 ** 32, mult = 214013, inc = 2531011 }) {
  this.seed = seed;
  this.mod = mod;
  this.mult = mult;
  this.inc = inc;
}
```

## Random Int

Utilise la formule suivante :  $X(n+1) = (mult * Xn + inc) \% mod$

Génère un nombre entre 0 et  $2^{32}$

```
/**
 * Genere un nombre entre [0 et 2**32]
 * @returns {Number}
 */
randomInt() {
  this.seed = (this.seed * this.mult + this.inc) % this.mod;
  return this.seed;
}
```

## Random Float

```
/**
 * Return un chiffre random entre 0 et 1
 * @return {Number}
 */
randomFloat() {
    return this.randomInt() / this.mod;
}
```

## Random Float Range

Utilise la fonction randomFloat ci-dessus

```
/**
 * Return float entre min et max
 * ref : https://stackoverflow.com/questions/1064901/random-number-between-2-double-numbers
 * @param {Number} min
 * @param {Number} max
 * @returns {Number}
 */
randomFloatRange(min, max) {
    return min + this.randomFloat() * (max - min);
}
```

## Random Int Range

Utilise la fonction randomFloat ci-dessus et arrondie à l'entier avec Math.floor()

```
/**
 * Return float entre min et max
 * @param {Number} min
 * @param {Number} max
 * @returns {Number}
 */
randomIntRange(min, max) {
    return Math.floor(this.randomFloatRange(min, max));
}
```

# Partie 2 - Teste du générateur

## Condition du test

Les probabilités (2-3) et (11-12) ne sont pas regroupées. On compte donc  $k = 11$  catégories.

On utilise la class Dice pour générer un roll de dés.

La class contient sa propre instance de générateur aléatoire

On utilise la fonction roll pour générer un nombre entre 1 et le nombre de face (défaut 6)

```

class Dice {
  /**
   * @param {Number} sides
   */
  constructor(sides = 6) {
    /** @type {Number}*/ this.sides = sides;
    /** @type {LCG}*/ this.random = new LCG(getLCGparams());
    /** @type {Array<Number>}*/ this.prob_comb_side6 =
      [1/36, 2/36, 3/36, 4/36, 5/36, 6/36, 5/36, 4/36, 3/36, 2/36, 1/36]
  }

  /**
   * Genere un nombre aleatoire entre 1 et nb sides
   * @returns {Number} */
  roll() {
    return this.random.randomIntRange(1, this.sides + 1);
  }
}

```

On effectue ITER lancers

```

// Generate roll combinaison
for (let i = 0; i < ITER; i++) {
  random_values_dice.push(dice.roll() + dice.roll());
}

```

On calcule :

- les **valeur attendues** pour un nombre de lancé donné
- la **distribution** des résultats que l'on a obtenue :
  - ex : (1 : [1,1,1 ..., 1], 2 : [2,2,...2],..., 12 : [12,12,12,...,12])
- les **valeurs observées** avec notre dice + générateur

```

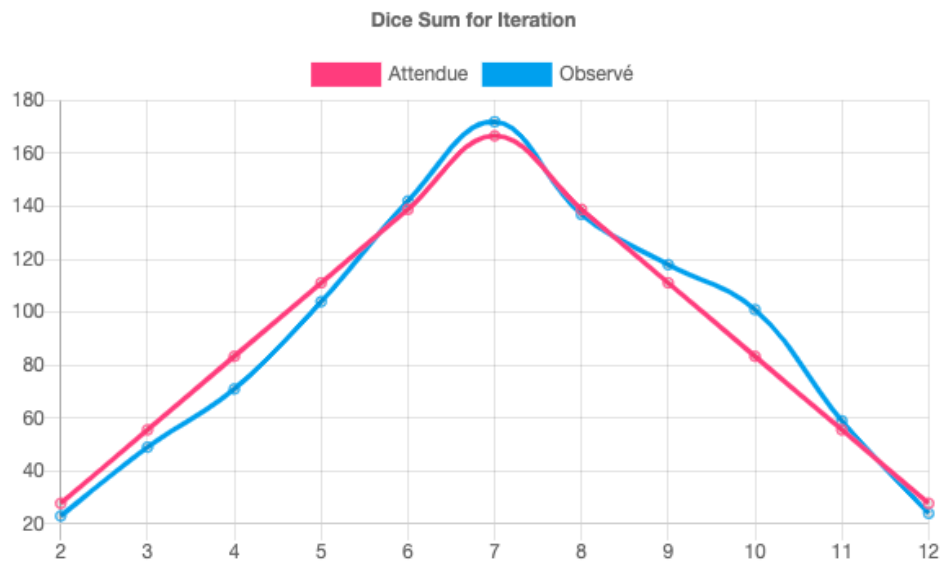
// Variables
const attendue = dice.prob_comb_side6.map((prob) => prob * ITER);
const distribution = distribute(random_values_dice);
const observed = Object.keys(distribution).map((key, index) => distribution[key].length);

```

## Résultat graphique

Pour 1 000 itération

## Partie 2

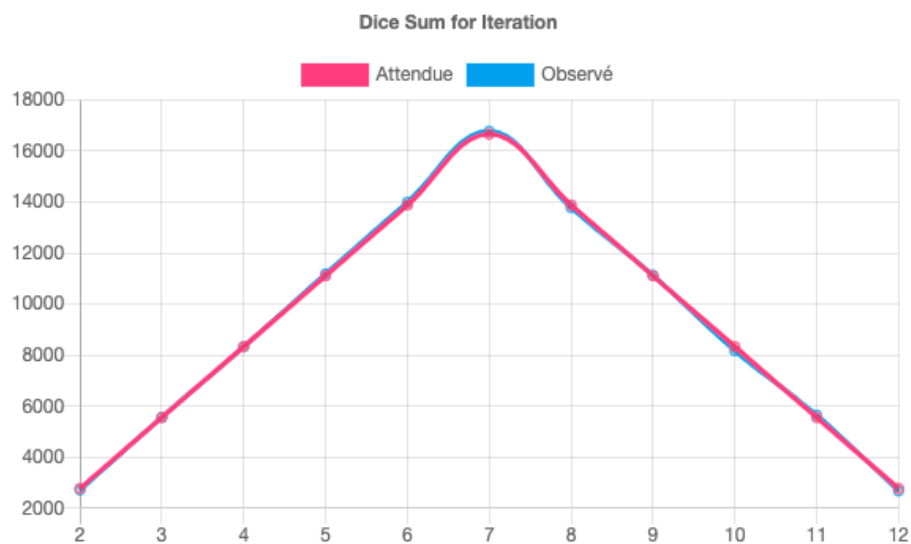


Seed: 312810000 Mod: 4294967296 Mult: 214013 Inc: 2531011

Iteration :

Pour 100 000 itération

## Partie 2



Seed: 1185802624 Mod: 4294967296 Mult: 214013 Inc: 2531011

Iteration :

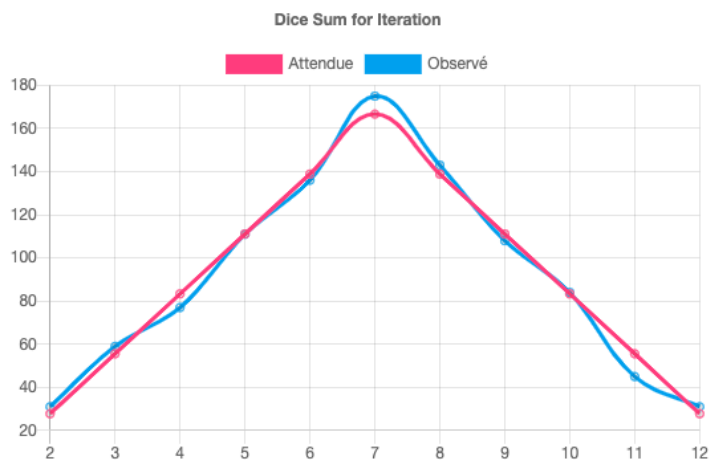
## Test du Khi2

Le test Khi2 return la sum  $((obs - att) ** 2 / att)$

```
class Khi2 {  
    /**  
     * Effectue le teste du Khi2 sur les valeurs observed et attendues  
     * @param {Array<Number>} observed  
     * @param {Array<Number>} attendue  
     * @returns {Number}*/  
    execute(observed, attendue) {  
        let result = 0;  
        // zip prend deux arr et en forme une ex :  
        // [a1 , a2, a3] et [b1, b2, b3] --> [ [a1, b1] , [a2, b2], [a3, b3] ]  
        let arr = zip(observed, attendue);  
        for (var [obs, att] of arr) result += (obs - att) ** 2 / att;  
        return result;  
    }  
}
```

## Result Khi2

### Partie 2



Seed: 1056550680 Mod: 4294967296 Mult: 214013 Inc: 2531011

Iteration :

Khi2 Result :  
4.139

Après plusieurs itération (testable de l'exécutable) mes valeurs du khi2 varient entre : 2 et 20  
(Raison : la seed est aléatoire pour chaque simulation)

Selon la table des valeurs critiques suivante avec un degré liberté de  $k-1 \rightarrow 11-1 = 10$   
Le taux de confiance est satisfaisant.

	$\alpha = 0,01$	$\alpha = 0,05$	$\alpha = 0,10$
ddl			
1	6,6349	3,8415	2,7055
2	9,2104	5,9915	4,6052
3	11,3449	7,8147	6,2514
4	13,2767	9,4877	7,7794
5	15,0863	11,0705	9,2363
6	16,8119	12,5916	10,6446
7	18,4753	14,0671	12,0170
8	20,0902	15,5073	13,3616
9	21,6660	16,9190	14,6837
10	23,2093	18,3070	15,9872
11	24,7250	19,6752	17,2750
12	26,2170	21,0261	18,5493
13	27,6882	22,3620	19,8119

## Partie 3 - Utilisation du générateur

### Mise en oeuvre de la marche aléatoire

#### random walk

On génère un chiffre entre 0 et le nombre de direction

On récupère la position [X,Y] dans le canvas en fonction de la direction

On push le noeud dans le chemin courant

```
randomWalk() {
  let possibleDir = [-2, -1, 1, 2];

  for (let i = 0; i < this.n; i++) {
    let dir = possibleDir[this.random.randomIntRange(0, possibleDir.length)];
    let pos = this.getPosition(dir);
    let node = new Node(dir, pos);
    this.path.push(node);
    this.pos = node.getIntPos();
  }
  return true;
}
```

#### non reversing walk

Identique au random walk, seulement,

tant que la direction est la même que la précédente, on génère une nouvelle direction.



```

nonReversingWalk() {
  let last = Number.MIN_SAFE_INTEGER;
  let possibleDir = [-2, -1, 1, 2];

  for (let i = 0; i < this.n; i++) {
    // Logic
    let dir = possibleDir[this.random.randomIntRange(0, possibleDir.length)];
    while (dir == last * -1) {
      dir = possibleDir[this.random.randomIntRange(0, possibleDir.length)];
    }
    last = dir;
    // Node
    let pos = this.getPosition(dir);
    let node = new Node(dir, pos);
    this.path.push(node);
    this.pos = node.getIntPos();
  }
  return true;
}

```

## self avoiding walk

J'ai testé plusieurs algorithmes pour implémenter le self avoiding walk mais les temps d'exécutions obtenues étaient bien trop long pour simplement gérer des walks de  $n = 20$  steps.

J'ai donc choisi d'utiliser la méthode du **backtracking** pour résoudre ce problème. Je peux maintenant générer des walks de  $n = 900 - 1000$  steps en quelques secondes.

Simple algorithme récursif qui renvoi vrai ou faux si un chemin de  $n$ -step à été trouvé.

On crée un nodes pour  $n$  steps

On vérifie si une solution a été trouvé sinon

On prend aléatoirement une direction dans les valeurs possibles.

Si la position n'a jamais été inscrite dans le chemin, on assigne le noeud au chemin

On appelle récursivement la fonction pour qu'elle corrige dans les assignments dans la stack.

```

selfAvoidingWalk() {
  if (this.isComplete()) return true;

  let node = this.selectUnsignedVariable();

  for (var value of node.domain) {
    let random = this.random.randomIntRange(0, node.domain.length);

    let position = this.getPosition(node.domain[random]);

    if (this.isValid(position)) {
      let save = new Array(...node.domain);

      node.dir = node.domain[random];
      node.setPos(position);
      node.domain.splice(random, 1);
      this.assigne(node);
      this.pos = node.getIntPos();

      let succes = false;

      succes = this.selfAvoidingWalk();

      if (succes) {
        return true;
      } else {
        node.domain = save;
        this.unnasigne(node);
        this.pos = this.path[this.path.length - 1].getIntPos();
      }
    }
  }
}
}

```

## Validation graphique

Toutes ces simulation peuvent être effectués dans le build de l'app :

n steps :

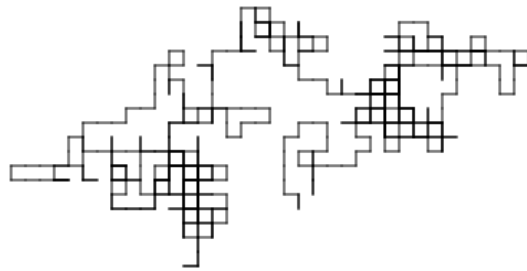
☒ randomWalk ☐ nonReversingWalk ☐ selfAvoidingWalk

Run

Les simulations qui suivent sont effectuées avec n-steps = 600

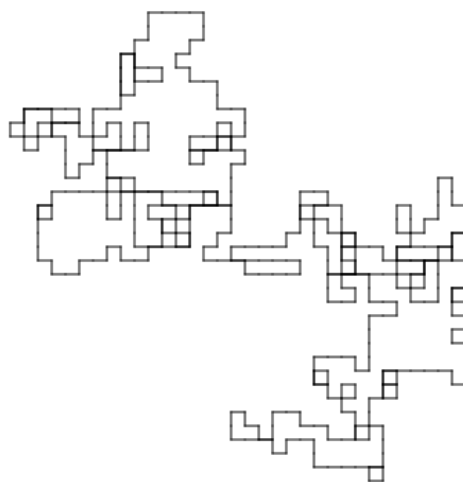
random walk

### Partie 3



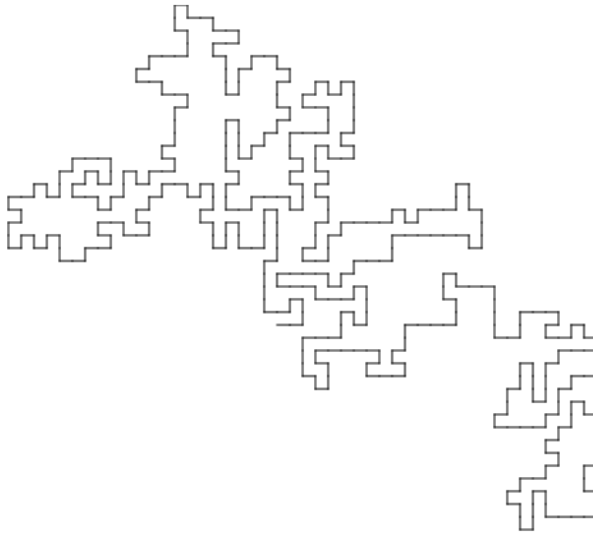
non reversing walk

### Partie 3



self avoiding walk

### Partie 3



## Etude end to en mean Square

Peux prendre un peu de temps selon les paramètres :

n Step:

50

k Iteration:

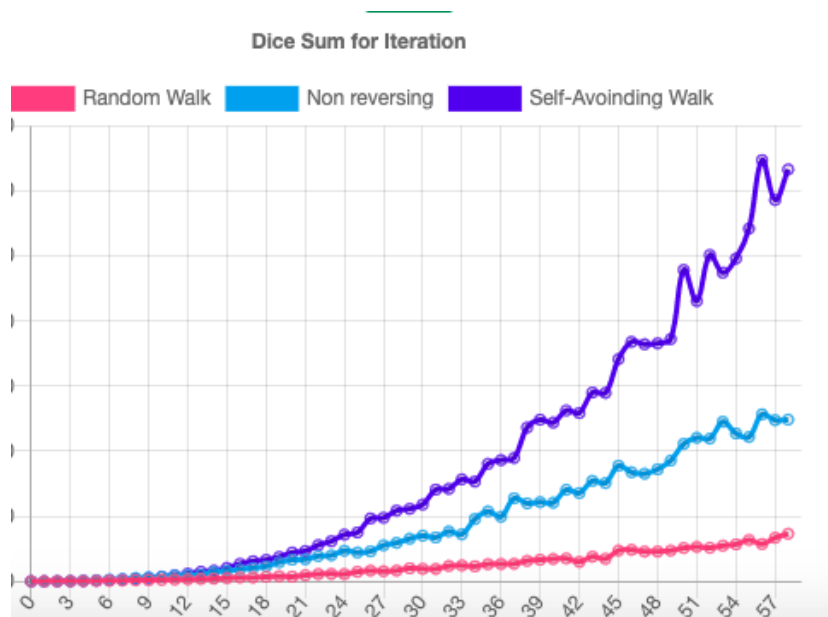
400

### Procédure

Chaque walk est générée k fois pour 0 à n steps (ici : k = 500, n = 0 à 60)

Pour chaque itération 0 à n on fait le mean square du end to end

Code disponible index.js → fonction Partie\_3\_4()



Le end to end est calculé en utilisant la distance euclidienne en (posStart) et (posEnd)

```
getEndToEndDistance() {
  var start = this.path[0].getIntPos();
  var end = this.path[this.path.length - 1].getIntPos();
  var euclidean = getEuclideanDistance(start, end);
  return euclidean;
}
```

```
function getEuclideanDistance(pos1, pos2) {
  var a = pos1[0] - pos2[0];
  var b = pos1[1] - pos2[1];
  var distance = Math.sqrt(a * a + b * b);
  return distance ** 2;
}
```

Le mean Square est calculé de la façon qui suit :

Pour chaque valeur on la met au carré, on fait la somme et on divise par le nombre de valeurs.

```
function MeanSquare(arr) {
  return arr.map((val) => val ** 2).reduce((acum, val) => acum + val) / arr.length;
}
```

Les valeurs sont mises au carré pour une meilleure représentation sur le graph.

# Annexe

## Interface

Note : LCG est configurable, Seed = str(Time()) || int, Mod = str(2\*\*32) || int

### P1 / P2 / P3

LCG

Seed:

Time()

Mod:

2\*\*32

Mult:

214013

Inc:

2531011

Seed: Mod: Mult: Inc:

Iteration : 1000

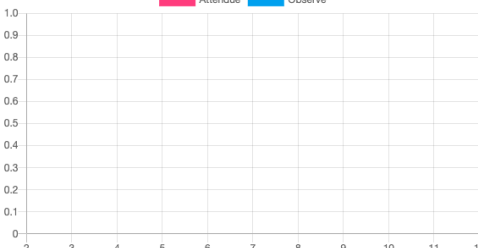
Khi2 Result :

Run

Partie 2

Dice Sum for Iteration

Attendue Observé



Partie 3

n steps : 200

☒ randomWalk ☐ nonReversingWalk ☐ selfAvoidingWalk

Run

Dice Sum for Iteration

Random Walk Non reversing Self-Avoiding Walk



## Graphe P3.4

2531011

Seed: Mod: Mult: Inc:

Iteration : 1000

Khi2 Result :

Run

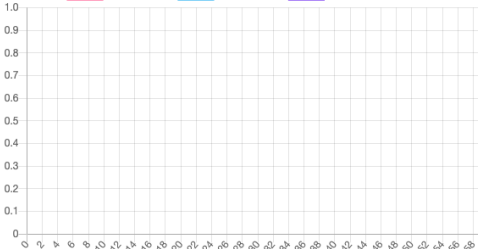
n steps : 200

☒ randomWalk ☐ nonReversingWalk ☐ selfAvoidingWalk

Run

Dice Sum for Iteration

Random Walk Non reversing Self-Avoiding Walk



n Step: 60

k Iteration: 400

Run