

Στοχαστικές Διαδικασίες και Μέθοδοι Υπολογιστικής Φυσικής

Σπυρίδων Μπαξεβανάκης



ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΙΟΝΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ



Αυτή η πτυχιακή έχει άδεια χρήσης Creative Commons CC0 1.0 Universal (CC0 1.0) Public Domain Dedication.

Επιβλέπων

Αυλωνίτης Μάρκος, Αναπληρωτής Καθηγητής,
Ιόνιο Πανεπιστήμιο

Τριμελής Επιτροπή

Αυλωνίτης Μάρκος, Αναπληρωτής Καθηγητής,
Ιόνιο Πανεπιστήμιο

Βλάμος Παναγιώτης, Καθηγητής – Πρόεδρος του Τμήματος Πληροφορικής,
Ιόνιο Πανεπιστήμιο

Έξαρχος Θεμιστοκλής, Επίκουρος Καθηγητής,
Ιόνιο Πανεπιστήμιο

Αφιερωμένο στην οικογένειά μου που με στήριξε κατά την διάρκεια των σπουδών μου.

Περιεχόμενα

Πρόλογος vii

Ευχαριστίες ix

1	Εισαγωγή στη Julia	1
1.1	Το Πρόβλημα των Δύο Γλωσσών	1
1.2	Τελεστές, Μεταβλητές & Συναρτήσεις	3
1.2.1	Συνάρτηση Cumsum	5
1.3	Δομές Δεδομένων Διαφόρων Διαστάσεων	6
1.3.1	Indexing - Ευρετηρίαση	8
1.3.2	Χρήση του Splatting	9
1.4	Επαναλήψεις και Έλεγχος Ροής	9
1.5	Broadcasting - Vectorized Operations	11
1.5.1	Σχολιάζοντας Κώδικα	12
1.6	Εγκαθιστώντας και Χρησιμοποιώντας Πακέτα	12
1.6.1	Το Πακέτο Plots	13
2	Στοχαστικές Διαδικασίες και Μοντελοποίηση	15
2.1	Τυχαίος Περίπατος	15

2.1.1	Διαδικασία Λευκού θορύβου Gauss	18
2.1.2	Τυχαίος Περίπατος σε Πλέγμα	18
2.2	Διαδικασία Wiener	19
2.2.1	Αλγοριθμική Κατασκευή	21
2.2.2	Γενικευμένη Κίνηση Brown	26
2.2.3	Γεωμετρική Κίνηση Brown	26
2.3	Κατανομή & Διαδικασία Poisson	29
2.3.1	Αλγοριθμική Κατασκευή	30
2.4	Διαδικασία Ornstein-Uhlenbeck	33
2.5	Το Μοντέλο Olami-Feder-Christensen για Σεισμούς	34
3	Μέθοδοι Υπολογιστικής Φυσικής	39
3.1	Πεπερασμένες Διαφορές σε Μερικές Διαφορικές Εξισώσεις	39
3.2	Αριθμητική Επίλυση της Εξίσωσης Διάχυσης Θερμότητας	45
3.3	Οριακές και Αρχικές Συνθήκες	49
	Βιβλιογραφία	53

Πρόλογος

Σκοπός του συγγραφέα με το παρόν κείμενο είναι η δημιουργία ενός εγχειριδίου (textbook) Στοχαστικών Διαδικασιών και Μεθόδων Υπολογιστικής Φυσικής στο οποίο γίνεται ρητή σύνδεση θεωρητικών και πρακτικών οπτικών. Για αυτόν τον σκοπό ο συγγραφέας κάνει χρήση της υπολογιστικής γλώσσας Julia (έκδοση 1.4), συμπεριλαμβάνοντας κώδικα σε κάθε τομέα του εγχειριδίου. Το παρόν κείμενο είναι μία εισαγωγή στον κόσμο της μοντελοποίησης, της προσομοίωσης αλλά και του αριθμητικού υπολογισμού. Βασικές γνώσεις πιθανοτήτων και προγραμματισμού (όχι απαραίτητα στην γλώσσα Julia) είναι απαραίτητες ώστε να μεγιστοποιήσει κανείς τη χρησιμότητα του εγχειριδίου. Επίσης, συνίσταται ο αναγνώστης να πειραματίζεται με τον κώδικα που δίνεται, καθώς ακολουθεί το κείμενο. Η γλώσσα Julia επιλέχθηκε λόγω της ευκολίας ανάγνωσης, της ομοιότητας με τον ψευδοκώδικα αλλά και της αυξανόμενης υιοθέτησής της στα επιστημονικά πεδία που πραγματεύεται το κείμενο. Οι αλγόριθμοι, εικόνες που εμφανίζονται στο παρόν εγχειρίδιο μπορούν να βρεθούν στο αποθετήριο <https://github.com/spirosbax/textbook-companion> μαζί με επιπλέον βοηθητικό περιεχόμενο.

ΣΠΥΡΙΔΩΝ ΜΠΑΞΕΒΑΝΑΚΗΣ

Ιόνιο Πανεπιστήμιο, Κέρκυρα.

5 Μαρτίου, 2020

Ευχαριστίες

Ο συγγραφέας θα ήθελε να ευχαριστήσει όλους αυτούς που συνέβαλαν στην συγγραφή αυτής της πτυχιακής εργασίας παρέχοντας επικοδομητικά σχόλια και κριτική:

Ιωάννης Κερασίωτης, Δημήτρης Κοτινάς, Στέλιος Γαβράς και Στέλλα Κοκολάκη.

Ιδιαίτερα ευχαριστώ τον καθηγητή Μάρκο Αυλωνίτη για την πολύτιμη καθοδήγησή του.

1 Εισαγωγή στη Julia

Στην ιστορία της επιστήμης των υπολογιστών έχουν δημιουργηθεί δεκάδες γλώσσες προγραμματισμού που απαρτίζονται από μια μεγάλη γκάμα χαρακτηριστικών.

Σε γενικές γραμμές χωρίζουμε τις γλώσσες προγραμματισμού σε δύο κατηγορίες: γλώσσες χαμηλού επιπέδου (low level) και γλώσσες υψηλού επιπέδου (high level). Συνήθως οι γλώσσες χαμηλού επιπέδου έχουν πολύ καλύτερη απόδοση σε σχέση με τις γλώσσες υψηλού επιπέδου, αυτό γίνεται γιατί μία γλώσσα χαμηλού επιπέδου είναι πιο κοντά στο υλικό του υπολογιστή και αυτό της επιτρέπει να υποστηρίξει εγγενώς λειτουργίες όπως για παράδειγμα την διαχείριση μνήμης, κάτι που έχει ως αποτέλεσμα στην βέλτιστη διαχείριση των υπολογιστικών πόρων. Εν αντιθέση, μία γλώσσα υψηλού επιπέδου ασχολείται κυρίως με το τί υπολογίζει ο υπολογιστής παρά με το πώς υλοποιείται ο υπολογισμός, έτσι ανταλλάσει απόδοση με ευκολία χρήσης. Για παράδειγμα γλώσσες όπως οι C, C++, Go, Assembly και Java είναι χαμηλού επιπέδου ενώ οι Python, R, Matlab και Wolfram είναι υψηλού επιπέδου.

Η γλώσσα Julia πρωτοεμφανίστηκε το 2012 και συνδυάζει τα καλύτερα σημεία και των δύο κόσμων. Είναι εύκολη στην χρήση ενώ παράλληλα παρέχει απόδοση που θυμίζει την γλώσσα χαμηλού επιπέδου C. Στον εικόνα 1.1 εμφανίζεται μία σύγκριση της απόδοσης διαφόρων γλωσσών συμπελαμβανομένης και της Julia.

1.1 Το Πρόβλημα των Δύο Γλωσσών

Ένα από τα συνηθισμένα προβλήματα που αντιμετωπίζουν οι επιστήμονες των υπολογιστών είναι το πρόβλημα των δύο γλωσσών ή αλλιώς The Two Language Problem. Για να το καταλάβουμε, ας φανταστούμε έναν επιστήμονα που θέλει να δημιουργήσει ένα υπολογιστικό μοντέλο προσομοίωσης μίας διαδικασίας.

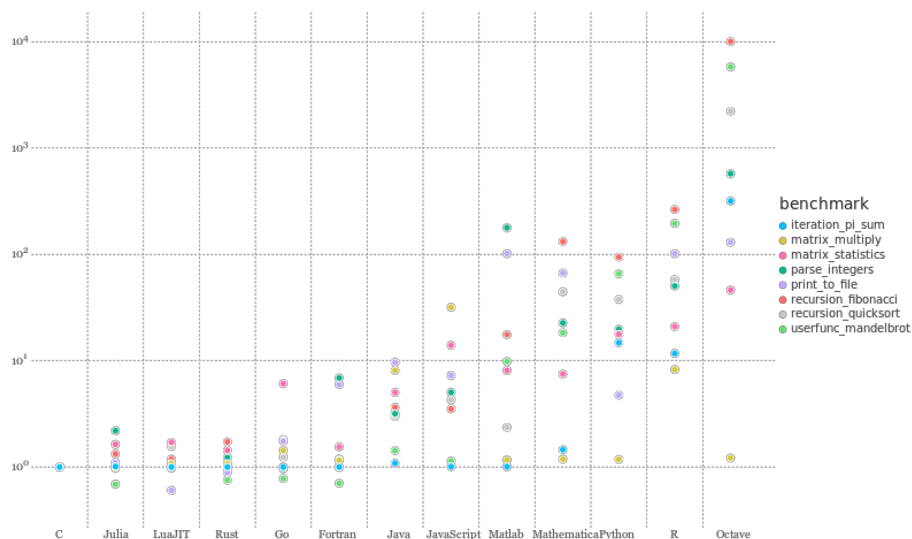


Figure 1.1. Σύγκριση ταχύτητας της Julia σε σχέση με άλλες γλώσσες προγραμματισμού.

Εάν χρησιμοποιήσει μία γλώσσα υψηλού επιπέδου (παραδείγματος χάριν Python):

- θα είναι εύκολο και γρήγορο να προγραμματίσει το μοντέλο,
- όμως το πρόγραμμα θα περιορίζεται από την απόδοση της γλώσσας
- και θα είναι πολύ χρονοβόρο να χρησιμοποιηθεί πολλαπλές φορές ώστε να εξάγει συμπεράσματα.

Από την άλλη, εάν χρησιμοποιήσει μία γλώσσα χαμηλού επιπέδου (παραδείγματος χάριν C):

- η ανάπτυξη του προγράμματος θα πιο είναι δύσκολη και χρονοβόρα,
- το πρόγραμμα θα απαιτεί πολύ λιγότερο χρόνο για να εκτελεστεί.

Είναι προφανές και οι δύο λύσεις έχουν πλεονεκτήματα και μειονεκτήματα. Η προσέγγιση που ακολουθείται συνήθως είναι να γίνεται χρήση δύο γλωσσών προγραμματισμού. Μία γλώσσα υψηλού επιπέδου για την υλοποίηση ενός προτύπου και μία γλώσσα χαμηλού επιπέδου για την υλοποίηση του τελικού υπολογιστικού μοντέλου. Αυτό ονομάζουμε ως το πρόβλημα των δύο γλωσσών, το οποίο η Julia επιλύει έχοντας ταυτόχρονα ευκολία χρήσης και απόδοση παρόμοια με γλώσσες όπως η C.

1.2 Τελεστές, Μεταβλητές & Συναρτήσεις

Η Julia υποστηρίζει όλους του συνηθισμένους τελεστές πρόσθεση, αφαίρεση, πολλαπλασιασμός, διαίρεση, ύψωση σε δύναμη με τον εξής τρόπο:

```
julia> 1 + 1
2
julia> 1 - 1
0
julia> 2 * 5
10
julia> 1 / 5
0.2
julia> 2 ^ 3
8
```

Μπορούμε να δημιουργήσουμε μία μεταβλητή βάζοντας το όνομα της μεταβλητής στα αριστερά μετά ένα ίσον και στο δεξί μέρος βάζουμε την τιμή της μεταβλητής.

```
julia> a = 3
3
```

Επιπλέον μπορούμε να δηλώσουμε πολλές μεταβλητές με την ίδια τιμή ως εξής:

```
julia> a = b = 5
5
julia> a
5
julia> b
5
```

Για να ορίσουμε μία συνάρτηση χρησιμοποιούμε την δεσμευμένη λέξη `function` ακολουθούμενη από το όνομα της συνάρτησης μαζί με τα ορίσματά της και τέλος χρησιμοποιούμε την δεσμευμένη λέξη `end` για να ορίσουμε το τέλος της:

```
julia> function myplus(x, y)
    x+y
end
myplus (generic function with 1 method)
```

Η παραπάνω συνάρτηση δέχεται δύο ορίσματα `x,y`, τα προσθέτει και επιστρέφει το αποτέλεσμα. Στην Julia η τελευταία τιμή που υπολογίζεται επιστρέφεται αυτόματα από την συνάρτηση, αντ' αυτού μπορούμε να χρησιμοποιήσουμε την δεσμευμένη λέξη `return`. Επιπλέον, υπάρχει η δυνατότητα προσδιορισμού των τύπων των μεταβλητών ως εξής:

```
julia> function myplus(x::Int, y::Int)
    x+y
end
myplus (generic function with 2 methods)
```

Αυτό αυξάνει την ταχύτητα εκτέλεσης του προγράμματος καθώς η Julia δεν χρειάζεται να ξοδέψει υπολογιστική ισχύ για να βρει αυτόματα τον τύπο της μεταβλητής. Σε αυτό το σημείο είναι σημαντικό να πούμε ότι όλες οι μεταβλητές στην Julia έχουν τύπο είτε το δηλώσουμε είτε όχι. Κάποιοι διαθέσιμοι τύποι μεταβλητών περιλαμβάνουν:

- `Int8`, `Int16`, `Int32`, `Int64`, `Int128`
- `UInt8`, `UInt16`, `UInt32`, `UInt64`, `UInt128`
- `Float16`, `Float32`, `Float64`

Εκτός από τον παραπάνω, υπάρχουν επιπλέον δύο τρόποι να ορίσει κανείς συναρτήσεις στην Julia. Όταν κάποιος θέλει να ορίσει μία συνάρτηση που αποτελείται από μία γραμμή τότε μπορεί να το κάνει ως εξής:

```
julia> square(x, y) = (x + y) ^ 2
square (generic function with 1 method)
```

Όπως και την προηγούμενη συνάρτηση η τελευταία τιμή επιστρέφεται αυτόματα. Ο τελευταίος τρόπος ορισμού μίας συνάρτησης είναι η δημιουργία μίας ανώνυμης συνάρτησης (anonymous function) με τον παρακάτω τρόπο:

```
julia> anonym = (x, y) -> (x + y) ^ 2
#1 (generic function with 1 method)
```

Η παραπάνω συνάρτηση δεν έχει όνομα αλλά μπορούμε να την χρησιμοποιήσουμε καθώς είναι αποθηκευμένη στην μεταβλητή `anonym`. Τις συναρτήσεις που ορίσαμε μπορούμε να τις καλέσουμε ως εξής:

```
julia> myplus(5, 4)
9
julia> square(2, 3)
25
julia> anonym(2, 3)
25
```

1.2.1 Συνάρτηση Cumsum

Στους αλγορίθμους του παρόντος κειμένου γίνεται συχνή χρήση της συνάρτησης `cumsum` ή αλλιώς του σωρευτικού αθροίσματος. Ας δούμε ένα παράδειγμα χρήσης και μετά θα εξηγήσουμε πως δουλεύει η συγκεκριμένη συνάρτηση.

```
julia> a = [1, 2, 3, 4, 5]
5-element Array{Int64,1}:
 1
 2
 3
 4
 5
julia> cumsum(a)
5-element Array{Int64,1}:
 1
 3
 6
10
15
```

Η συνάρτηση `cumsum` που δέχεται σαν είσοδο την λίστα `a` δρα ως εξής:

- προσθέτει το 1 στο τρέχον άθροισμα $1 + 0 = 1$
- προσθέτει το 2 στο τρέχον άθροισμα $2 + 1 = 3$
- προσθέτει το 3 στο τρέχον άθροισμα $3 + 3 = 6$
- προσθέτει το 4 στο τρέχον άθροισμα $4 + 6 = 10$

Η διαδικασία συνεχίζεται με τον ίδιο τρόπο για όλα τα στοιχεία της λίστας εισόδου.

1.3 Δομές Δεδομένων Διαφόρων Διαστάσεων

Τα arrays, οι λίστες και οι πίνακες είναι ορολογία που μπερδεύει τους νεοεισερχόμενους στην Julia. Οπότε είναι σημαντικό να ξεκαθαρίσουμε την ορολογία από την αρχή. Τα arrays μπορεί να είναι πολυδιάστατα ενώ ένα μονοδιάστατο array είναι το αντίστοιχο της λίστας και ένα δισδιάστατο array είναι ένας πίνακας.

Μία λίστα φτιάχνεται ως εξής:

```
julia> a = [1, 2, 3]
3-element Array{Int64,1}:
 1
 2
 3
julia> push!(a, 4)
4-element Array{Int64,1}:
 1
 2
 3
 4
```

στον παραπάνω κώδικα φτιάχνουμε μία λίστα (μονοδιάστατο array) με όνομα **a** και με την συνάρτηση **push!** προσθέτουμε τον αριθμό 4 στο τέλος της. Το θαυμαστικό (!) στο τέλος του ονόματος της συνάρτησης σημαίνει ότι η συγκεκριμένη συνάρτηση τροποποιεί τα ορίσματα της.

Για να φτιάξουμε έναν πίνακα (δισδιάστατο array) χρησιμοποιούμε παρόμοια σύνταξη με μία μικρή διαφορά:

```
julia> b = [1 2 3]
1×3 Array{Int64,2}:
 1  2  3
```

στον παραπάνω κώδικα οι αριθμοί 1,2 και 3 δεν χωρίζονται με κόμμα. Αυτή η σύνταξη κατασκευάζει έναν πίνακα με μία μόνο γραμμή. Στο συγκεκριμένο παράδειγμα κατασκευάζουμε έναν πίνακα με μία γραμμή και τρεις στήλες. Ενώ εάν θέλουμε να προσθέσουμε και άλλες γραμμές τότε χρησιμοποιούμε το ελληνικό ερωτηματικό (;):

```
julia> c = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4
```

σε αυτό το παράδειγμα η μεταβλητή `c` περιέχει ένα δισδιάστατο array (πίνακας) με δύο γραμμές και δύο στήλες.

Για να μετατρέψουμε έναν πίνακα σε μία λίστα μπορούμε να χρησιμοποιήσουμε την εντολή `vec`:

```
julia> b = [1 2 3]
1×3 Array{Int64,2}:
 1  2  3
julia> vec(b)
3-element Array{Int64,1}:
 1
 2
 3
```

αντίστοιχα μπορούμε να μετατρέψουμε μία λίστα σε έναν πίνακα με την εντολή `hcat`:

```
julia> a = [1, 2, 3]
3-element Array{Int64,1}:
 1
 2
 3
julia> hcat(a)
3×1 Array{Int64,2}:
 1
 2
 3
julia> hcat(a) '
1×3 LinearAlgebra.Adjoint{Int64,Array{Int64,2}}:
 1  2  3
```

στον παραπάνω κώδικα η εντολή `hcat` μετατρέπει την λίστα `a` σε έναν πίνακα 3×1 και στην επόμενη γραμμή χρησιμοποιώντας τον τελεστή `'` μεταθέτουμε (transpose) τον πίνακα από 3×1 σε 1×3 .

Συνδυάζοντας όλα τα παραπάνω βλέπουμε και άλλους τρόπους να τροποποιήσουμε πίνακες:

```
julia> c = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4
julia> vcat(c, [5 6])
3×2 Array{Int64,2}:
 1  2
 3  4
 5  6
julia> [c; 7 8]
3×2 Array{Int64,2}:
 1  2
 3  4
 7  8
```

1.3.1 Indexing - Ευρετηρίαση

Η ευρετηρίαση (indexing) στην Julia γίνεται χρησιμοποιώντας την σύνταξη "[γραμμή, στήλη]". Επιπλέον μπορούμε να χρησιμοποιήσουμε τον τελεστή ":" για να δείξουμε ότι θέλουμε όλες τις γραμμές ή τις στήλες. Σε δομές με περισσότερες ή λιγότερες διαστάσεις (λ.χ. σε λίστες) η ευρετηρίαση γίνεται προσθέτοντας ή αφαιρώντας στοιχεία από τις αγκύλες.

```
julia> c = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4
julia> c[1,2] # πρώτη γραμμή, δεύτερη στήλη
2
julia> c[:,1] # όλες οι γραμμές, πρώτη στήλη
2-element Array{Int64,1}:
 1
 3
julia> a = [1,2,3]
3-element Array{Int64,1}:
 1
 2
 3
julia> a[1]
1
julia> a[2]
2
```

1.3.2 Χρήση του Splatting

Ένας συχνά χρησιμοποιούμενος τελεστής στην Julia είναι ο τελεστής "...” (splatting). Αυτός μας επιτρέπει να εξαγάγουμε τα περιεχόμενα μία δομής και να τα περάσουμε σε μία συνάρτηση ως ξεχωριστά ορίσματα. Σε άλλη περίπτωση, θα έπρεπε να δώσουμε ολόκληρη την δομή σε ένα όρισμα στην συνάρτηση. Ας δούμε ένα παράδειγμα:

```
julia> splat_example(a,b,c) = a * b * c
splat_example (generic function with 1 method)
julia> args = [1,2,5]
3-element Array{Int64,1}:
 1
 2
 5
julia> splat_example(args...)
10
julia> args...
Error: syntax: "...” expression outside call
```

στον παραπάνω κώδικα πρώτα ορίζουμε μία συνάρτηση `splat_example`, έπειτα δημιουργούμε μία λίστα με όνομα `args`. Χρησιμοποιούμε τον τελεστή `splatting` όταν δίνουμε τα ορίσματα στην συνάρτηση `splat_example`. Αυτό έχει ως αποτέλεσμα τα περιεχόμενα `1,2,5` της `args` να δωθούν ως ξεχωριστά ορίσματα στην συνάρτηση `splat_example` και να ανατεθούν στις μεταβλητές `a,b` και `c` αντίστοιχα. Τέλος, παρατηρήστε ότι ο τελεστής `splatting` δεν δουλεύει εκτός της περιοχής ορισμάτων των κλήσεων συναρτήσεων.

1.4 Επαναλήψεις και Έλεγχος Ροής

Οι περισσότερες ανάγκες επανάληψης στην Julia καλύπτονται από την δομή `for`. Η σύνταξη της οποίας φαίνεται παρακάτω:

```
julia> for a in 1:3
println(a)
end
1
2
3
```

Πρώτα χρησιμοποιούμε την δεσμευμένη λέξη **for** ακολουθούμενη από μία μεταβλητή στην οποία θα γίνεται η ανάθεση των τιμών της επανάληψης. Έπειτα, στα δεξιά, βάζουμε μία λίστα ή ένα αντικείμενο **range**. Το αντικείμενο **range** περιγράφει ένα σύνολο τιμών, στο συγκεκριμένο παράδειγμα το `range 1:3` περιγράφει τις τιμές 1,2 και 3. Η **for** θα αναθέσει κάθε τιμή του **range** στην μεταβλητή **a**. Όπως και όταν ορίζουμε μία συνάρτηση, το **for** τελειώνει με την δεσμευμένη λέξη **end**. Ανάμεσα από το **for** και το **end** τοποθετείται ο κώδικας που θα επαναληφθεί.

Το παρακάτω παράδειγμα περιέχει επιπλέον τρόπους χρήσης του **for**:

```
julia> # μπορούμε να χρησιμοποιήσουμε in ή =
# το αποτέλεσμα είναι το ίδιο
for a = 5:7
    println(a)
end
5
6
7
julia> # χρησιμοποιούμε λίστα αντί για αντικείμενο
# range
b = [1,2,3]
3-element Array{Int64,1}:
 1
 2
 3
julia> for value in b
    println(value)
end
1
2
3
```

Εκτός από την δομή **for** μπορούμε να επαναλάβουμε κομμάτια κώδικα και με ένα `list comprehension`. Ο συγκεκριμένος τρόπος μας επιτρέπει να συλλέξουμε τα αποτελέσματα μίας επανάληψης και χρησιμοποιείται ως εξής:

```
julia> square = x -> x*x
Error: invalid redefinition of constant square
julia> a = [square(i) for i in 1:10]
Error: MethodError: no method matching square(::Int64)
Closest candidates are:
  square(::Any, !Matched::Any) at none:1
```

Στην αρχή ορίζουμε μία συνάρτηση `square` η οποία δέχεται ένα όρισμα και επιστρέφει το τετράγωνό του. Στη συνέχεια χρησιμοποιούμε ένα `list comprehension` για να τετραγωνίσουμε τους αριθμούς από το 1 μέχρι και το 10 και να αποθηκεύσουμε τα αποτελέσματα μία λίστα `a`.

1.5 Broadcasting - Vectorized Operations

Ο τελεστής `."` στην Julia ονομάζεται Broadcasting και μας επιτρέπει να κάνουμε, σε τεχνικούς όρους, Vectorized Operations. Δηλαδή αντί να χρησιμοποιήσουμε πολλές `for` επαναλήψεις χρησιμοποιούμε τον τελεστή `."`.

```
julia> a = [1, 2, 3]
3-element Array{Int64,1}:
 1
 2
 3
julia> b = [1, 2, 3]
3-element Array{Int64,1}:
 1
 2
 3
julia> c = a .* b
3-element Array{Int64,1}:
 1
 4
 9
```

Στον παραπάνω κώδικα φτιάχνουμε δύο λίστες `a`, `b` και έπειτα κάνουμε broadcast τον τελεστή πολλαπλασιασμού σε αυτές. Αυτό υπολογίζει τα: $1 * 1$, $2 * 2$, $3 * 3$ και τα βάζει στην λίστα `c`. Θα μπορούσαμε να πραγματοποιήσουμε τον αντίστοιχο υπολογισμό με χρήση της `for` ως εξής:

```
julia> a = [1, 2, 3]
3-element Array{Int64,1}:
 1
 2
 3
julia> b = [1, 2, 3]
3-element Array{Int64,1}:
 1
 2
 3
```

```

julia> # λίστα μηδενικών μεγέθους 3 τύπου Int64
c = zeros{Int64, 3}
3-element Array{Int64,1}:
 0
 0
 0
julia> # ή eachindex(b), η συνάρτηση eachindex επιστρέφει
# ένα αντικείμενο range από το 1 μέχρι το μέγεθος
# της δομής
for i in eachindex(a)
    c[i] = a[i] * b[i]
end
julia> c
3-element Array{Int64,1}:
 1
 4
 9

```

1.5.1 Σχολιάζοντας Κώδικα

Στην Julia ένα σχόλιο μίας γραμμής ξεκινάει με τον χαρακτήρα `"#"` (δίεση), ενώ ένα σχόλιο πολλαπλών γραμμών ξεκινάει με `"#="` και τελειώνει με `"=#"`. Για παράδειγμα:

```

julia> # αυτό είναι ένα σχόλιο μίας γραμμής
#= αυτό είναι ένα σχόλιο
πολλαπλών γραμμών =#

```

1.6 Εγκαθιστώντας και Χρησιμοποιώντας Πακέτα

Για να χρησιμοποιήσει κανείς ένα πακέτο σε έναν κώδικα **Julia** πρέπει να χρησιμοποιήσει την δεσμευμένη λέξη **using**. Για παράδειγμα:

```

julia> using Distributions
julia> rand(Normal(0,1), 10)
10-element Array{Float64,1}:
-0.1397444750783605
-0.6744608568720255
 0.6386594237287526
 0.637761130731613
-0.8505893347128612
-0.33057598839616725
-0.3074289964848192

```

```
1.6556465530897078
-0.5272694456180088
1.1600782686821025
```

Στον παραπάνω κώδικα εισαγάγουμε το πακέτο `Distributions` το οποίο υλοποιεί διάφορες κατανομές. Έπειτα χρησιμοποιούμε την κατανομή `Normal` που εισαγάγαμε από το πακέτο για να παράξουμε 10 τυχαίους αριθμούς κανονικής κατανομής.

Συχνά κανείς θέλει να χρησιμοποιήσει πακέτα που δεν είναι προεγκατεστημένα και πρέπει να εγκατασταθούν. Για να γίνει αυτό χρησιμοποιούμε το πακέτο `Pkg` ως εξής:

```
julia> # using Pkg
# Pkg.add("DataFrames")
```

στον παραπάνω κώδικα εισαγάγουμε το πακέτο `Pkg` και μετά χρησιμοποιούμε την συνάρτηση του `Pkg.add` για να κάνουμε εγκατάσταση το πακέτο `DataFrames` το οποίο προσφέρει συναρτήσεις για τη διαχείριση δεδομένων. Αφού ολοκληρωθεί η εγκατάσταση τότε μπορούμε να εισαγάγουμε το πακέτο με την δεσμευμένη λέξη `using`.

1.6.1 Το Πακέτο Plots

Το πακέτο `Plots` προσφέρει πολλές δυνατότητες οπτικοποίησης δεδομένων. Αφού εγκατασταθεί με τον τρόπο που συζητήσαμε, χρησιμοποιείται ως εξής:

```
julia> # using Plots
# a = rand(100)
# plot(a)
```

Το πακέτο `Plots` υλοποιεί πάρα πολλούς τρόπους απεικόνισης και υποστηρίζει μεγάλη γκάμα διαφορετικών δεδομένων. Για αυτό τον λόγο δεν θα καλυφθεί περαιτέρω. Τα τεχνικά έγγραφα (documentation) του `Plots` είναι διαθέσιμα στο σύνδεσμο: <https://docs.juliaplots.org/latest/>.

2 Στοχαστικές Διαδικασίες και Μοντελοποίηση

2.1 Τυχαίος Περίπατος

Ο τυχαίος περίπατος ή αλλιώς Random Walk είναι, αν όχι η βασικότερη τότε σίγουρα, μία από τις πιο βασικές έννοιες του ευρύτερου πεδίου των Στοχαστικών Διαδικασιών. Μπορεί να χρησιμοποιηθεί από την μοντελοποίηση της κίνησης σωματιδίων¹ έως την μοντελοποίηση χρηματιστηριακών μετοχών². Το συναρπαστικό όμως με αυτή την έννοια δεν είναι μόνο η πληθώρα εφαρμογών που απολαμβάνει αλλά κυρίως η απλότητά της, η οποία της επιτρέπει να παίρνει διάφορες μορφές ανάλογα την εφαρμογή.

Μιλώντας πιο ουσιαστικά ας περιγράψουμε τον αλγόριθμο παραγωγής ενός τυχαίου περιπάτου σε μία διάσταση. Έστω ένας περιπατητής με αρχική θέση $x = 0$ ο οποίος σε κάθε χρονική στιγμή $t = 1, 2, 3 \dots$ αλλάζει θέση στην $x + \epsilon$, όπου ϵ τυχαία μεταβλητή κανονικής κατανομής. Ο αλγόριθμος 2.1 παράγει μονοδιάστατο τυχαίο περίπατο αφού η τυχαία μεταβολή επηρεάζει μόνο μία κατεύθυνση, την κάθετη. Μπορούμε όμως, να επεκτείνουμε τον απλό τυχαίο περίπατο που δημιουργήσαμε σε περισσότερες διαστάσεις. Αυτό σημαίνει ότι η τυχαία μεταβολή θα επηρεάζει παραπάνω από μία κατευθύνσεις. Η υλοποίηση του κώδικα για δύο διαστάσεις σχετικά με τον αλγόριθμο 2.1 είναι αρκετά απλή και δίνεται στον αλγόριθμο 2.2 ο οποίος επιστρέφει έναν πίνακα όπου κάθε στήλη ή αλλιώς κάθε κατεύθυνση κίνησης είναι ένας μονοδιάστατος τυχαίος περίπατος. Με τον ίδιο τρόπο μπορούμε να κατασκευάσουμε έναν αλγόριθμο παραγωγής τυχαίου περιπάτου N διαστάσεων. Αυτός δίνεται στον αλγόριθμο 2.3.

¹ A. W. Visser, "Using Random Walk Models to Simulate the Vertical Distribution of Particles in a Turbulent Water Column," Marine Ecology Progress Series, vol. 158, pp. 275–281, 1997.

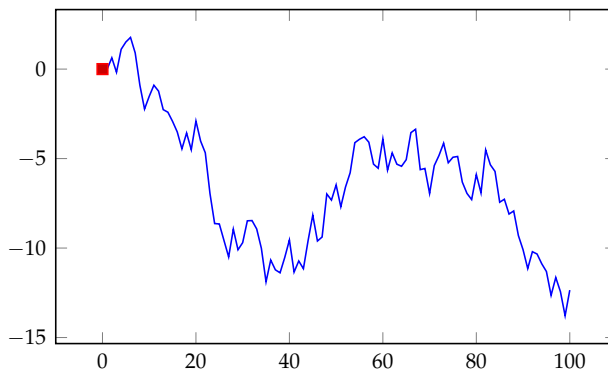
² O. Vasicek, "An Equilibrium Characterization of the Term Structure," Journal of Financial Economics, vol. 5, no. 2, pp. 177–188, 1977. [Online]. Available: <https://EconPapers.repec.org/RePEc:eee:jfinec:v:5:y:1977:i:2:p:177-188>.

```

# πακέτο που υλοποιεί διάφορες κατανομές
using Distributions
function random_walk1d(timesteps)
    previous_position::Float64 = 0
    positions::Array{Float64,1} = []
    #= κάθε τυχαίος περίπατος πρέπει να
    ξεκινάει από το 0 =#
    push!(positions, previous_position)

    for t in 2:timesteps # για κάθε χρονική στιγμή
        # τυχαία μεταβολή κανονικής κατανομής
        new_pos = previous_position + rand(Normal(0,1))
        # πρόσθεσε την νέα θέση στο τέλος της λίστας θέσεων
        push!(positions, new_pos)
        # ενημέρωσε την προηγούμενη θέση
        previous_position = new_pos
    end
    positions # επέστρεψε την λίστα θέσεων
end

```



Algorithm 2.1. Συνάρτηση που δέχεται ως είσοδο αριθμό βημάτων και επιστρέφει μονοδιάστατο τυχαίο περίπατο αντίστοιχου μήκους.

Figure 2.1. Απεικόνιση της συνάρτησης `random_walk1d` (μονοδιάστατος τυχαίος περίπατος) σε δύο διαστάσεις για 100 χρονικές στιγμές-βήματα. Σε κάθε χρονική στιγμή ο περιπατητής αλλάζει τη θέση του στον άξονα y ανάλογα με μία τυχαία μεταβλητή. Το κόκκινο τετράγωνο υποδεικνύει την αρχική θέση του περιπατητή.

```

function random_walk2d(timesteps)
    # φτιάξε έναν timesteps x 2 πίνακα
    [random_walk1d(timesteps) random_walk1d(timesteps)]
end

```

Algorithm 2.2. Συνάρτηση που δέχεται ως είσοδο αριθμό βημάτων και επιστρέφει δισδιάστατο τυχαίο περίπατο αντίστοιχου μήκους χρησιμοποιώντας τον Algorithm 2.1.

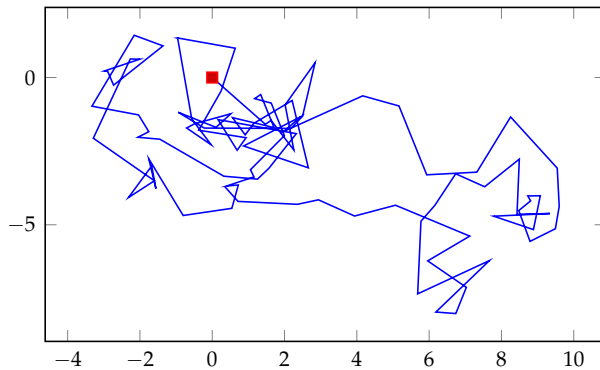


Figure 2.2. Απεικόνιση της συνάρτησης `random_walk2d` (δισδιάστατος τυχαίος περίπατος) σε δύο διαστάσεις για 100 χρονικές στιγμές-βήματα. Σε κάθε χρονική στιγμή ο περιπατητής μεταβάλλει τη θέση του και στους δύο άξονες ανάλογα με μία τυχαία μεταβλητή. Το κόκκινο τετράγωνο υποδεικνύει την αρχική θέση του περιπατητή.

```
function random_walkNd(timesteps,N)
    #=
    η συνάρτηση reduce εφαρμόζει επαναληπτικά την συνάρτηση hcat
    στην λίστα. Αυτό έχει ως αποτέλεσμα να μετατραπεί η
    μονοδιάστατη λίστα λιστών σε δισδιάστατο πίνακα όπου
    κάθε στήλη είναι μονοδιάστατος τυχαίος περίπατος.
    =#
    reduce(hcat,[random_walk1d(timesteps) for i in 1:N])
end
```

Algorithm 2.3. Συνάρτηση που δέχεται ως είσοδο αριθμό βημάτων και διαστάσεων και επιστρέφει αντίστοιχο τυχαίο περίπατο.

2.1.1 Διαδικασία Λευκού θορύβου Gauss

Όταν η τυχαία μεταβλητή ϵ που επηρεάζει την κίνηση του τυχαίου περιπατητή προέρχεται από την Gaussian/κανονική κατανομή τότε ο τυχαίος περίπατος μπορεί να θεωρηθεί το σωρευτικό άθροισμα (cumulative sum) μίας διαδικασίας λευκού θορύβου της ίδιας κατανομής. Η διαδικασία λευκού θορύβου κανονικής κατανομής είναι ουσιαστικά μία ακολουθία τυχαίων αριθμών της κατανομής αυτής. Έστω τώρα ότι από μία τέτοια ακολουθία θέλουμε να κατασκευάσουμε ένα τυχαίο περίπατο. Θέτουμε την αρχική θέση $pos = x, y = 0, 0$ και σε κάθε βήμα το x αυξάνεται κατά ένα, ενώ το y σε κάθε βήμα είναι το άθροισμα των x πρώτων τυχαίων μεταβλητών της ακολουθίας λευκού θορύβου.

```
using Distributions
function white_noise(timesteps)
    rand(Normal(), timesteps)
end
```

Algorithm 2.4. Συνάρτηση που δέχεται ως είσοδο αριθμό βημάτων και επιστρέφει λευκό θόρυβο κανονικής κατανομής αντίστοιχου μήκους.

Η υλοποίηση μίας συνάρτησης παραγωγής διαδικασίας λευκού θορύβου κανονικής κατανομής δίνεται στον αλγόριθμο 2.4.

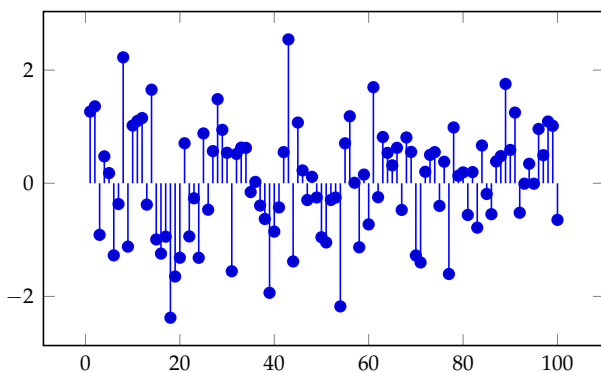


Figure 2.3. Απεικόνιση της συνάρτησης `white_noise` για 100 βήματα.

2.1.2 Τυχαίος Περίπατος σε Πλέγμα

Λίγα μοντέλα απολαμβάνουν τόσο μεγάλη πληθώρα εφαρμογών όσο ο τυχαίος περίπατος σε πλέγμα. Στη συγκεκριμένη παραλλαγή του τυχαίου περιπάτου,

η διαδικασία λαμβάνει χώρα πάνω σε ένα πλέγμα σημείων. Όταν η πιθανότητα μετάβασης του περιπατητή από μία θέση προς μία γειτονική είναι ίδια για όλα τα γειτονικά σημεία, τότε λέμε ότι ο τυχαίος περίπατος είναι συμμετρικός. Στην αντίθετη περίπτωση λέμε ότι είναι ασύμμετρος. Η κύρια διαφορά με τον δισδιάστατο τυχαίο περιπατητή που είδαμε προηγουμένως είναι ότι ο χώρος στον οποίο μπορεί να κινηθεί ο περιπατητής είναι διακριτός και όχι συνεχής. Δηλαδή υπάρχει περιορισμένος αριθμός σημείων στον χώρο. Βλέπουμε μία απλή υλοποίηση ενός τέτοιου αλγορίθμου στον αλγόριθμο 2.5, ενώ μία πιο περίπλοκη, αλλά ταυτόχρονα πιο όμορφη, υλοποίηση υπάρχει στον αλγόριθμο 2.6. Και οι δύο υλοποιήσεις παράγουν όμοια τυχαία διαδικασία η οποία φαίνεται στην εικόνα 2.5 για 10,000 βήματα. Η έκδοση του περιπάτου σε πλέγμα που επιλέξαμε να παρουσιάσουμε σε αυτή τη παράγραφο επιτρέπει σε έναν περιπατητή που βρίσκεται σε ένα σημείο να μετακινηθεί μόνο στα σημεία που φαίνονται στην εικόνα 2.4.

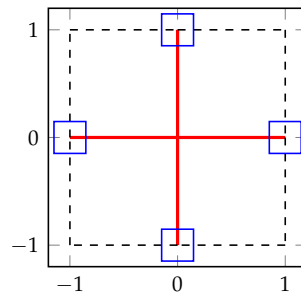


Figure 2.4. Απεικόνιση των πιθανών κατευθύνσεων που μπορεί να πάρει ο περιπατητής.

Δηλαδή σε κάθε χρονική στιγμή ο περιπατητής επιτρέπεται να μετακινηθεί μόνο πάνω, κάτω, δεξιά ή αριστερά. Άλλες παραλλαγές του τυχαίου περιπάτου σε πλέγμα περιλαμβάνουν, να επιτρέπεται η κίνηση διαγώνια, να επιτρέπεται να μένει στο ίδιο σημείο, να είναι δυνατή η αυξομείωση του μεγέθους του χώρου ή/και των αποστάσεων μεταξύ των σημείων, και άλλα.

2.2 Διαδικασία Wiener

Η διαδικασία Wiener είναι μία στοχαστική διαδικασία συνεχούς χρόνου που πήρε το όνομα της από τον μαθηματικό Norbert Wiener. Ο Norbert Wiener και ο Albert Einstein ήταν οι πρώτοι που θεμελίωσαν μαθηματικά την κίνηση

```

function lattice_random_walk(timesteps)
    x = [0] # λίστα των x, ξεκινάει από το 0
    y = [0] # λίστα των y, ξεκινάει από το 0
    for i in 1:timesteps
        # τυχαία ισοπίθανη επιλογή μεταξύ 0 και 1,
        if rand([0,1]) == 0
            # μετακινούμε τον περιπατητή μόνο
            # στον άξονα των x με μοναδιαίο βήμα
            push!(x, x[end] + rand([-1,1]))
            push!(y, y[end])
        else
            # μετακινούμε μόνο στον y
            push!(x, x[end])
            push!(y, y[end] + rand([-1,1]))
        end
    end
    # ενώνουμε τις δύο λίστες σε έναν
    # δισδιάστατο πίνακα με δύο στήλες.
    hcat(x,y)
end

```

Algorithm 2.5. Αλγόριθμος που δέχεται ως είσοδο αριθμό βημάτων και παράγει δισδιάστατο τυχαίο περίπατο σε πλέγμα αντίστοιχου μήκους.

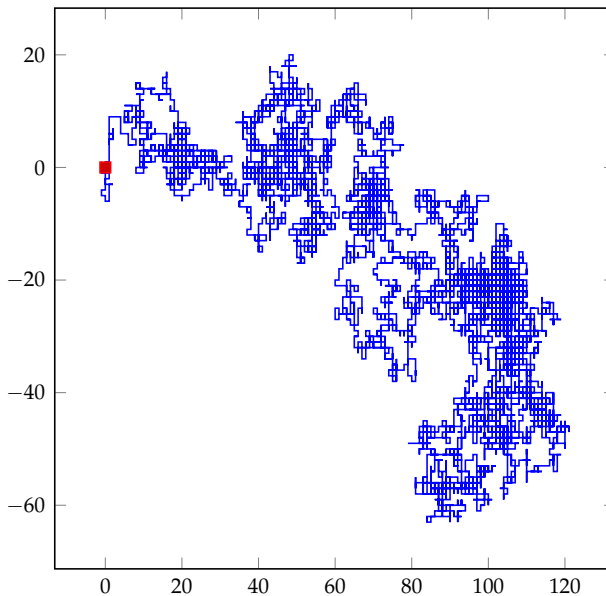


Figure 2.5. Απεικόνιση του δισδιάστατου τυχαίου περιπάτου που παράγεται από τον αλγόριθμο `lattice_random_walk` μήκους 10000 βημάτων. Σε κάθε χρονική στιγμή ο περιπατητής μεταβάλλει τη θέση του σε έναν από τους δύο άξονες κατά +1 ή -1. Ο άξονας μετακίνησης κάθε χρονικής στιγμής διαλέγεται τυχαία. Το κόκκινο τετράγωνο υποδεικνύει την αρχική θέση του περιπατητή.

```

function lattice_random_walk_advanced(ts)
    #= επιλέγουμε ts φορές μία από
    τις ισοπίθανες κατευθύνσεις =#
    r = rand([[0 1],[0 -1],[1 0],[-1 0]],ts)
    #= πρώτα μετατρέπουμε την λίστα λιστών σε
    δισδιάστατο πίνακα δύο στηλών και μετά
    υπολογίζουμε το σωρευτικό άθροισμα του
    πίνακα ανά γραμμή =#
    cumsum(reduce(vcat,r),dims=1)
end

```

Algorithm 2.6. Καλύτερη υλοποίηση του Algorithm 2.5.

Brown την οποία είχε παρατηρήσει το 1827 ο βοτανολόγος Robert Brown. Ο Brown τοποθέτησε έναν κόκκο γύρης σε νερό και παρατήρησε καθώς αυτός βυθιζονταν, περίμενε ότι ο κόκκος θα ακολουθήσει μία ήρεμη κάθετη πορεία χωρίς οριζόντιες μεταβολές αφού η μόνη δύναμη που τον επηρέαζε ήταν η βαρύτητα. Προς έκπληξη του Brown όμως, ο κόκκος δεχόταν συνεχείς, φαινομενικά τυχαίες, μικρές οριζόντιες μεταβολές. Αυτή του είδους η κίνηση ενός μικροσκοπικού σωματιδίου σε υγρό ή αέριο ονομάστηκε κίνηση Brown και η στοχαστική διαδικασία που τη μοντελοποίησε διαδικασία Wiener^{3,4}.

³ T.E. of Encyclopaedia Britannica, “Brownian Motion,” Encyclopaedia Britannica, 2017.

⁴ D.O. Siegmund, “Probability Theory,” Encyclopaedia Britannica, 2017.

2.2.1 Αλγοριθμική Κατασκευή

Η κατασκευή μία διαδικασίας Wiener είναι αλγοριθμικά απλή όμως ενσωματώνει θεμελιώδη θεωρήματα της στατιστικής. Η μέθοδος που θα εξετάσουμε ονομάζεται “κλιμακούμενος τυχαίος περίπατος” (scaled random walk) ή αλλιώς, το όριο του τυχαίου περιπάτου (limit of the random walk). Η κεντρική ιδέα είναι ότι εάν η χρονική περίοδος μείνει σταθερή (π.χ. $t_0 = 0$ και $t_\tau = 1$) και μέσα σε αυτή παραγάγουμε τυχαίους περιπάτους με όλο και μεγαλύτερο αριθμό βημάτων τότε καθώς το μέγεθος αυτών τείνει στο άπειρο ο διακριτός τυχαίος περίπατος θα μετατραπεί σε συνεχή κίνηση Brown στο χρονικό διάστημα $[t_0, t_\tau]$ ⁵. Για αρχή, θα χρειαστούμε έναν τυχαίο περίπατο κανονικής κατανομής Y_i μήκους n , με $Y_0 = 0$. Έπειτα αντί για 1 βήμα ανά δευτερόλεπτο (timestep) που ίσχυε στον τυχαίο περίπατο εδώ θα έχουμε k βήματα όπου $k \in \mathbb{Z}^+$, δηλαδή θετικός ακέραιος. Άρα στη χρονική περίοδο $[0, 1]$ με $k = 10$ θα έχουμε χρονικό βήμα $t = \frac{1}{k} = 0.1$ έτσι ώστε να έχουμε k βήματα, $t \cdot k = 0.1 \cdot 10 = 1$.

⁵ K. Taipale, Mathematical Preparation for Finance, Ch. 7. 2017, pp. 125–128.

Μπορούμε λοιπόν να ορίσουμε τη διαδικασία Wiener για το χρονικό διάστημα $[0,1]$ ως:

$$W_t^n = \sum_{i=1}^{k \cdot t} Y_i \quad (2.1)$$

Ο αλγόριθμος 2.7 υλοποιεί αυτό τον ορισμό. Προσέξτε ότι $W^n(t) = W_t^n$ στο πεδίο των στοχαστικών διαδικασιών.

```
# πακέτο που υλοποιεί διάφορες κατανομές
using Distributions
function wiener_process1d_no_scaling(timesteps)
    N = timesteps;
    #= πρώτα δημιουργούμε πίνακα που περιέχει
    τις χρονικές στιγμές των τιμών του τυχαίου
    περιπάτου. Ξεκινώντας από το 0 συνεχίζουμε
    με βήμα 1/N μέχρι να έχουμε πίνακα μεγέθους
    N =#
    X = collect(range(0, step = 1/N, length = N));
    #= Έπειτα δημιουργούμε έναν τυχαίο περίπατο ως εξής:
    Τραβάμε N-1 τυχαίους αριθμούς κανονικής κατανομής
    μέσου 0 και απόκλισης 1, υπολογίζουμε το σωρευτικό
    άθροισμα και προσθέτουμε ένα μηδενικό στην πρώτη
    θέση του πίνακα. Με αυτόν τον τρόπο κατασκευάσαμε
    τυχαίο περίπατο μήκους N. Αντί για την παρακάτω
    γραμμή θα μπορούσαμε να είχαμε καλέσει τον
    αλγόριθμο random_walk1d(N) =#
    Y = prepend!(cumsum(rand(Normal(0, 1), N-1)), 0);
    # επιστρέφουμε τα X, Y ως πλειάδα (tuple) λιστών
    (X, Y)
end
```

Algorithm 2.7. Αλγόριθμος που κατασκευάζει λανθασμένα μία διαδικασία Wiener.

Παρόλου που μέχρι στιγμής δεν έχουμε κάνει κάποιο λογικό σφάλμα στην μεθοδολογία της κατασκευής μας, η εξίσωση 2.5 και ο αλγόριθμος που την υλοποιεί παραβιάζουν ήδη μία από τις βασικές ιδιότητες της διαδικασίας Wiener. Αυτή είναι η $Var(W_t) = t$, δηλαδή ότι η διακύμανση της διαδικασίας πρέπει να αυξάνεται γραμμικά στον χρόνο. Εάν σκεφτεί κανείς ότι ο αλγόριθμός μας κάνει k βήματα το δευτερόλεπτο αντί για ένα βήμα ανά δευτερόλεπτο που ισχύει στον τυχαίο περίπατο που είδαμε νωρίτερα, τότε γίνεται λίγο

πιο ξεκάθαρη η αιτία αυτής της παραβίασης. Ας δούμε τώρα πως μπορούμε μαθηματικά να διορθώσουμε το λάθος.

Θέλουμε $Var(W_t) = t$ και ισχύει $Var(W_t) = k \cdot t$. Οπότε:

$$Var(W_t) = k \cdot t \Rightarrow \frac{Var(W_t)}{k} = t \quad (2.2)$$

Από τις ιδιότητες της διακύμανσης (Variance) ισχύει ότι $\alpha^2 Var(X) = Var(\alpha X)$ ⁶, οπότε:

$$\alpha^2 = \frac{1}{k} \Rightarrow \alpha = \frac{1}{\sqrt{k}} \quad (2.3)$$

Χρησιμοποιώντας την εξίσωση 2.3 στην εξίσωση 2.2 έχουμε:

$$Var\left(\frac{W_t}{\sqrt{k}}\right) = t \quad (2.4)$$

Άρα σύμφωνα με τα παραπάνω, για να διορθώσουμε την λανθασμένη υλοποίηση μας πρέπει να διαιρέσουμε κάθε τιμή του τυχαίου περιπάτου μήκους k με $\sqrt{k}$. Ο νέος και διορθωμένος ορισμός της διαδικασίας Wiener είναι ο:

$$W_t^n = \frac{\sum_{i=1}^{k \cdot t} Y_i}{\sqrt{k}} \quad (2.5)$$

Ο αλγόριθμος 2.8 και η εικόνα 2.7 παρουσιάζουν την σωστή υλοποίηση της μεθόδου κατασκευής μίας διαδικασίας Wiener. Είναι πολύ σημαντικό να παρατηρήσει κανείς ότι στην λανθασμένη υλοποίηση η διακύμανση των διαδικασιών είναι πολύ μεγαλύτερη από ότι στην σωστή. Για την ακρίβεια είναι k φορές μεγαλύτερη (εξίσωση 2.2). Επιπλέον αυτό φαίνεται και από την σύγκριση του εύρους των τιμών του άξονα των y στις εικόνες 2.7 και 2.6.

Το θεώρημα του Donsker απέδειξε ότι :

$$\lim_{n \rightarrow +\infty} W_t^n = W_t \quad (2.6)$$

δηλαδή ότι καθώς το n τείνει στο άπειρο και το t στο μηδέν τότε ο αλγόριθμος παράγει πραγματική συνεχή διαδικασία Wiener⁷. Το θεώρημα του Donsker αποτελεί συναρτησιακή επέκταση του Κεντρικού Οριακού Θεωρήματος⁸.

⁶ D. O. Siegmund, “Probability Theory,” Encyclopædia Britannica, 2017.

⁷ S. Lalley, “Mathematical Finance 345 Lecture 5: Brownian Motion,” 2001.

⁸ M. D. Donsker, An Invariance Principle for Certain Probability Limit Theorems. 1951.

```

# πακέτο που υλοποιεί διάφορες κατανομές
using Distributions
function wiener_process1d(timesteps)
    N = timesteps;
    #= πρώτα δημιουργούμε πίνακα που περιέχει
    τις χρονικές στιγμές των τιμών του τυχαίου
    περιπάτου. Ξεκινώντας από το 0 συνεχίζουμε
    με βήμα 1/N μέχρι να έχουμε πίνακα μεγέθους
    N =#
    X = collect(range(0, step = 1/N, length = N));
    #= Έπειτα δημιουργούμε έναν τυχαίο περίπατο ως εξής:
    Τραβάμε N-1 τυχαίους αριθμούς κανονικής κατανομής
    μέσου 0 και απόκλισης 1, υπολογίζουμε το σωρευτικό
    άθροισμα και προσθέτουμε ένα μηδενικό στην πρώτη
    θέση του πίνακα. Με αυτόν τον τρόπο κατασκευάσαμε
    τυχαίο περίπατο μήκους N. Αντί για την επόμενη
    γραμμή θα μπορούσαμε να είχαμε καλέσει τον
    αλγόριθμο random_walk1d(N) =#
    Y = prepend!(cumsum(rand(Normal(0, 1), N-1)), 0);
    #= κλιμάκωση του τυχαίου περιπάτου έτσι ώστε
    να διατηρηθεί η γραμμική αύξηση της διακύμανσης
    στον χρόνο =#
    Y = Y / sqrt(N);
    # επιστρέφουμε τα X, Y ως πλειάδα (tuple)
    (X, Y)
end

```

Algorithm 2.8. Αλγόριθμος που κατασκευάζει σωτά μία διαδικασία Wiener.

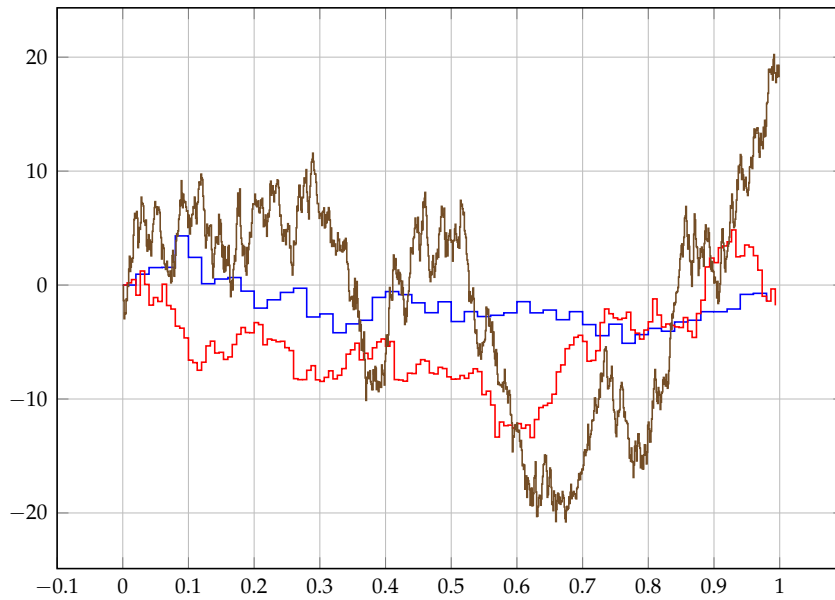


Figure 2.6. Λανθασμένη διαδικασία Wiener με 50 (μπλε), 150 (κόκκινο) και 1000 (καφέ) βήματα.

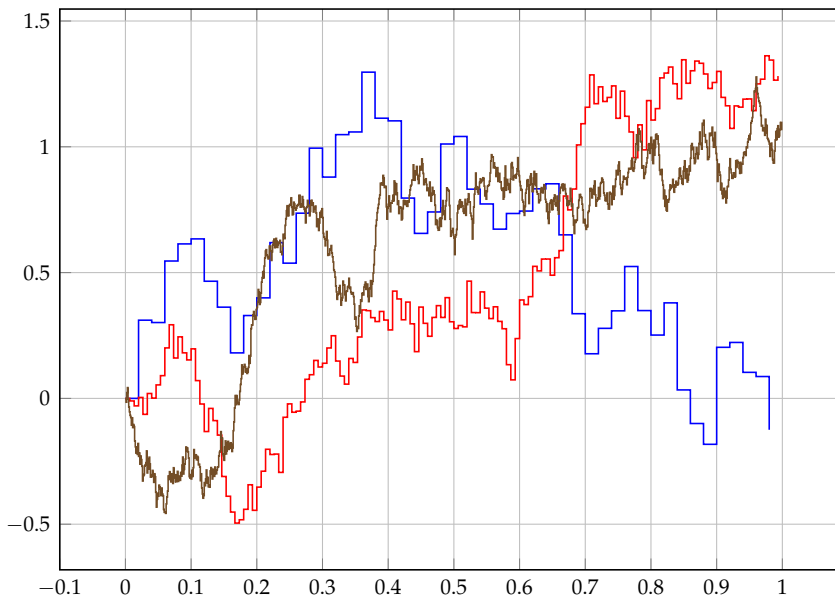


Figure 2.7. Διαδικασία Wiener με 50 (μπλε), 150 (κόκκινο) και 1000 (καφέ) βήματα.

2.2.2 Γενικευμένη Κίνηση Brown

Η γενικευμένη κίνηση Brown ή αλλιώς κίνηση Brown με drift επεκτείνει την κανονική κίνηση Brown (διαδικασία Wiener) ως εξής:

$$B_t = x_0 + \mu \cdot t + \sigma \cdot W_t \quad (2.7)$$

όπου x_0 είναι η αρχική τιμή, μ ο μέσος όρος ή αλλιώς ο όρος drift, σ η κανονική απόκλιση της διαδικασίας. Η εξίσωση 2.7 αποτελεί αναλυτική λύση της παρακάτω διαφορικής εξίσωσης:

$$dB_t = \mu \cdot dt + \sigma \cdot dW_t \quad (2.8)$$

Σύμφωνα με τις παραπάνω εξισώσεις, η γενικευμένη κίνηση Brown θέτει αρχική τιμή x_0 , προσθέτει το μέσο μ και πολλαπλασιάζει την κανονική απόκλιση σ σε μία υπάρχουσα κανονική διαδικασία Wiener n βημάτων. Στην εικόνα 2.8 εμφανίζονται τα αποτελέσματα του αλγορίθμου 2.9 που υλοποιεί την γενικευμένη κίνηση Brown. Παρατηρήστε ότι εδώ για τη διακύμανση της διαδικασίας ισχύει: $\text{Var}(B_t) = \sigma^2 \cdot t$ σε αντίθεση με τη διακύμανση της κανονικής διαδικασίας Wiener όπου $\text{Var}(W_t) = t$ αφού $\sigma = 1$. Η διακύμανση της διαδικασίας λοιπόν, εξαρτάται από τη διακύμανση και το πλήθος των βημάτων της.

2.2.3 Γεωμετρική Κίνηση Brown

Η στοχαστική διαδικασία που ορίζεται από την διαφορική εξίσωση:

$$dS_t = \mu \cdot S_t \cdot dt + \sigma \cdot S_t \cdot dW_t \quad (2.9)$$

ονομάζεται γεωμετρική κίνηση Brown ή αλλιώς εκθετική κίνηση Brown και έχει αναλυτική λύση ⁹:

$$S_t = S_0 \cdot e^{\mu \cdot t + \sigma \cdot W_t} \quad (2.10)$$

Όπου S_0 είναι η αρχική τιμή της διαδικασίας. Παρατηρήστε ότι ο εκθέτης του φυσικού αριθμού e στην παραπάνω εξίσωση είναι η γενικευμένη κίνηση Brown όπως περιγράφηκε στην εξίσωση 2.7 με $x_0 = 0$. Η γεωμετρική κίνηση Brown είναι μία από τις πιο δημοφιλείς διαδικασίες για την μοντελοποίηση τιμών μετοχών. Δεν μπορεί να πάρει αρνητικές τιμές και για αυτό προτιμάται από την γενικευμένη κίνηση Brown ¹⁰. Κύριο χαρακτηριστικό της είναι ότι

⁹ C. Pacatis, “Brownian Motion and Geometric Brownian Motion: Graphical Representations,” 2010.

¹⁰ S. M. Ross, Introduction to Probability Models, Ch. 10.3. Academic press, 2014.

```

function extended_brownian_motion1d(x0, μ, σ, timesteps)
    N = timesteps;
    #= χρησιμοποιούμε την κανονική διαδικασία Wiener
    που δημιουργήσαμε σε προηγούμενο αλγόριθμο.
    Επιστρέφει το X που είναι πίνακας με τους χρόνους
    των τιμών, και το Y που είναι πίνακας με τις
    τιμές της διαδικασίας =#
    (X, Y) = wiener_process1d(N);
    #= Μετασχηματίζουμε τη διαδικασία Wiener
    σύμφωνα με την εξίσωση που αναφέραμε.
    Η συνάρτηση enumerate παίρνει μία λίστα X και
    επιστρέφει μία πλειάδα λιστών όπου η πρώτη λίστα
    είναι οι δείκτες της X και η δεύτερη οι τιμές της =#
    B = [x0 + μ * t + σ * Y[i] for (i, t) in enumerate(X)];
    #= επιστρέφουμε τις τιμές όπως και στον αλγόριθμο
    της διαδικασίας Wiener =#
    (X, B)
end

```

Algorithm 2.9. Αλγόριθμος που κατασκευάζει μία γενικευμένη κίνηση Brown.

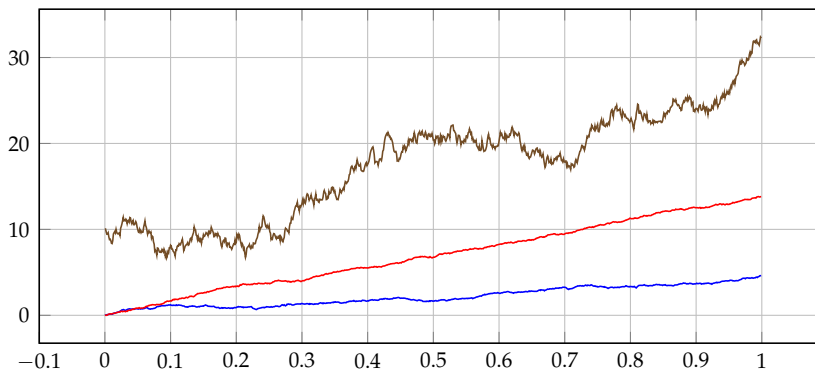


Figure 2.8. Γενικευμένη Κίνηση Brown με $x_0 = 0, \mu = 5, \sigma = 1$ (μπλε), $x_0 = 0, \mu = 15, \sigma = 1$ (κόκκινο) και $x_0 = 10, \mu = 15, \sigma = 10$ (καφέ) βήματα. Όλες οι διαδικασίες έχουν 1000 βήματα.

ο μέσος όρος μ , ή αλλιώς ο όρος drift, και η κανονική απόκλιση σ έχουν πολλαπλασιαστικό αντίκτυπο στην εξέλιξη τη διαδικασίας. Αυτό φαίνεται πιο εύκολα εάν αναδιατάξουμε τους όρους της 2.9 ως εξής:

$$dS_t = S_t \cdot (\mu \cdot dt + \sigma \cdot dW_t) \quad (2.11)$$

Η υλοποίηση εξίσωσης 2.10 παρουσιάζεται στον αλγόριθμο 2.10 και τα αποτελέσματα στην εικόνα 2.9.

```
function geometric_brownian_motion1d(S0, μ, σ, timesteps)
    N = timesteps;
    #= χρησιμοποιούμε την γενικευμένη κίνηση brown
    που δημιουργήσαμε σε προηγούμενο αλγόριθμο.
    Επιστρέφει το X που είναι πίνακας με τους χρόνους
    των τιμών, και το Y που είναι πίνακας με τις
    τιμές της διαδικασίας=#
    (X, Y) = extended_brownian_motion1d(0, μ, σ, N);
    #= μετασχηματίζουμε την γενικευμένη κίνηση Brown
    σύμφωνα με την εξίσωση της γεωμετρικής κίνησης=#
    S = S0 * exp.(Y);
    #= επιστρέφουμε τις τιμές όπως και στον αλγόριθμο
    της γενικευμένης κίνησης brown=#
    (X, S)
end
```

Algorithm 2.10. Αλγόριθμος που κατασκευάζει γεωμετρική κίνηση Brown.

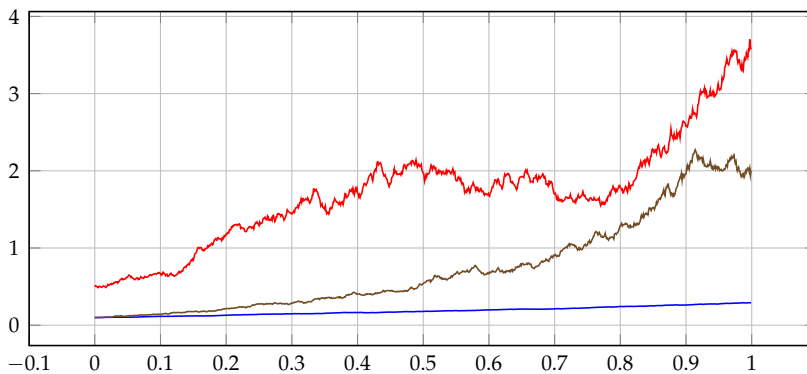


Figure 2.9. Γεωμετρική κίνηση Brown με $S_0 = 0.1, \mu = 1, \sigma = 0.1$ (μπλε), $S_0 = 0.5, \mu = 1, \sigma = 0.5$ (κόκκινο) και $S_0 = 0.1, \mu = 3, \sigma = 0.5$ (καφέ) βήματα. Όλες οι διαδικασίες έχουν 1000 βήματα.

2.3 Κατανομή & Διαδικασία Poisson

Πολλές φορές θέλουμε να μοντελοποιήσουμε ή και να προσομοιώσουμε διαδικασίες τυχαίων γεγονότων. Η συνήθης διαδικασία που χρησιμοποιείται για αυτό τον σκοπό είναι η διαδικασία Poisson που παράγεται από την ομώνυμη κατανομή. Πριν προχωρήσουμε όμως σε αυτή είναι καλό να αναπτύξουμε εν συντομία τον πρόγονό της, την Διωνυμική κατανομή (binomial distribution). Η Διωνυμική κατανομή εκφράζει την πιθανότητα να συμβούν k γεγονότα σε ένα συγκεκριμένο πλήθος δοκιμών, κάθε γεγονός έχει πιθανότητα να συμβεί ίση με p και το συνολικό πλήθος των δοκιμών είναι ίσο με n . Η κατανομή εκφράζεται από την εξίσωση:

$$P(X = k) = \binom{n}{k} * \left(\frac{\lambda}{n}\right)^k * \left(1 - \frac{\lambda}{n}\right)^{n-k} \quad (2.12)$$

όπου λ είναι η αναμενόμενη ή αλλιώς μέση τιμή της κατανομής και ορίζεται από τον τύπο $E(X) = \lambda = n * p$, από τον οποίο συνεπάγεται ότι $p = \frac{\lambda}{n}$. Το λ αναφέρεται συχνά και ως ο ρυθμός άφιξης ή ρυθμός της διαδικασίας. Για παράδειγμα, ας υποθέσουμε ότι θέλουμε να μοντελοποιήσουμε τον αριθμό των αυτοκινήτων που περνούν από έναν δρόμο μέσα σε μία ώρα. Θέτουμε $n = 60$ αφού η ώρα έχει 60 λεπτά και έχουμε:

$$P(X = k) = \binom{60}{k} * \left(\frac{\lambda}{60}\right)^k * \left(1 - \frac{\lambda}{60}\right)^{60-k} \quad (2.13)$$

Ας δούμε λίγο αναλυτικότερα τις παραπάνω εξισώσεις. Για τη διευκόλυνση μας, θα λέμε ότι έχουμε επιτυχία εάν σε μία δοκιμή υπάρξει γεγονός αλλιώς θα λέμε ότι έχουμε αποτυχία. Ο όρος $\binom{n}{k}$ εκφράζει όλους τους δυνατούς τρόπους με τους οποίους μπορούμε να έχουμε k επιτυχίες από n δοκιμές. Ο δεύτερος όρος $\left(\frac{\lambda}{n}\right)^k$ εκφράζει την πιθανότητα να έχουμε k επιτυχίες. Τέλος, ο τρίτος όρος $\left(1 - \frac{\lambda}{n}\right)^{n-k}$ εκφράζει την πιθανότητα να έχουμε $n - k$ αποτυχίες από n δοκιμές.

Στο παράδειγμά μας έχουμε επιλέξει να χρησιμοποιήσουμε $n = 60$ αλλά στην πραγματικότητα μία ώρα δεν έχει μόνο 60 δοκιμές. Θα μπορούσε για παράδειγμα να περάσει πάνω από ένα αυτοκίνητο σε ένα λεπτό, κάτι που το μοντέλο μας δεν επιτρέπει. Το επόμενο λογικό βήμα που μπορούμε να κάνουμε λοιπόν είναι να αυξήσουμε τον αριθμό των δοκιμών, δηλαδή αντί για 60 λεπτά μπορούμε να χρησιμοποιήσουμε $n = 3600$ δευτερόλεπτα. Η εξίσωση

της διωνυμικής κατανομής τότε θα εξελίσσονταν σε:

$$P(X = k) = \binom{3600}{k} * \left(\frac{\lambda}{3600}\right)^k * \left(1 - \frac{\lambda}{3600}\right)^{3600-k} \quad (2.14)$$

Αυτή η έκδοση του μοντέλου είναι μία καλύτερη προσέγγιση της πραγματικότητας αλλά κάποιος θα μπορούσε να πει ότι το πρόβλημα δεν λύθηκε καθώς είναι δυνατό δύο αυτοκίνητα να περάσουν μέσα σε ένα δευτερόλεπτο. Καθώς το μέγεθος της εκάστοτε χρονικής περιόδου γίνεται απειροελάχιστο και ο αριθμός των δοκιμών n τείνει στο άπειρο, φτάνουμε στην κατανομή Poisson:

$$P(X = k) = \lim_{n \rightarrow +\infty} \left(\binom{n}{k} * \left(\frac{\lambda}{n}\right)^k * \left(1 - \frac{\lambda}{n}\right)^{n-k} \right) = \frac{\lambda^k}{k!} * e^{-\lambda} \quad (2.15)$$

Η διαδικασία Poisson που βασίζεται στην εξίσωση 2.15 είναι μία διαδικασία διακριτού χρόνου στην οποία βασικές υποθέσεις είναι η ανεξαρτησία μεταξύ των γεγονότων και το ότι το λ παραμένει σταθερό κατά την διάρκεια της. Στο παράδειγμα με τα αυτοκίνητα που αναφέραμε πριν αυτά σημαίνουν ότι πρώτον, η πιθανότητα p να περάσει ένα αυτοκίνητο τη χρονική στιγμή t δεν εξαρτάται από τις προηγούμενες αφίξεις αυτοκινήτων, και δεύτερον, ότι ο ρυθμός άφιξης αυτοκινήτων παραμένει σταθερός.

2.3.1 Αλγοριθμική Κατασκευή

Ένας από τους πιο συνήθεις τρόπους κατασκευής μία διαδικασίας Poisson σε ένα διακριτό χρονικό διάστημα $[0, T]$ είναι χρησιμοποιώντας τους χρόνους μεταξύ των γεγονότων. Έστω $T > 0$ το όριο του χρονικού διαστήματος, $\lambda > 0$ ο ρυθμός της διαδικασίας και S_n η χρονική στιγμή του n -ιστού γεγονότος της διαδικασίας. Θα χρησιμοποιήσουμε το γεγονός ότι οι χρόνοι μεταξύ συνεχόμενων γεγονότων είναι ανεξάρτητες τυχαίες μεταβλητές που ακολουθούν εκθετική κατανομή. Έτσι και σύμφωνα με τα παραπάνω μπορούμε να βρούμε τη χρονική στιγμή του επόμενου γεγονότος S_{n+1} προσθέτοντας μία εκθετική τυχαία μεταβλητή στο S_n , δηλαδή $S_{n+1} = S_n + \text{Exp}(\lambda)$.

Οι αλγόριθμοι 2.11 και 2.12 προσομοιώνουν μία διαδικασία Poisson και τα αποτελέσματα απεικονίζονται στην εικόνα 2.10. Ο αλγόριθμος 2.12 χρησιμοποιεί μία τεχνική που ονομάζεται "Inverse Transform Sampling" για να μετατρέψει μία τυχαία μεταβλητή ομοιόμορφης κατανομής σε μία τυχαία μεταβλητή εκθετικής κατανομής.

```

# πακέτο που υλοποιεί διάφορες κατανομές
using Distributions
function poisson_point_process( $\lambda$ ,  $\tau_0$ ,  $\tau$ )
    #= φτιάχνουμε πίνακα με floats που θα αποθηκεύει τους χρόνους
    των γεγονότων =#
    times::Array{Float64, 1} = [0.]
    while true # συνεχόμενη επανάληψη
        #= υπολογισμός διαφοράς χρόνων με χρήση της συνάρτησης
        rand και την εκθετική κατανομή, χρησιμοποιούμε  $1/\lambda$  σαν
        παράμετρο της κατανομής καθώς σε αυτή το  $\lambda$  λειτουργεί
        αντίθετα από ότι με την κατανομή Poisson. Δηλαδή στην
        κατανομή Poisson  $\lambda = \lambda$ , ενώ σε αυτή την υλοποίηση της
        εκθετικής κατανομής  $\lambda = 1 / \lambda$  =#
        interevent = rand(Exponential( $1/\lambda$ ))
        # εφόσον δεν έχουμε ξεπεράσει το χρονικό όριο
        if times[end] + interevent <  $\tau$ 
            #= πρόσθεσε στο προηγούμενο χρόνο την διαφορά και
            αποθήκευσε το αποτέλεσμα στο τέλος του πίνακα =#
            push!(times, times[end] + interevent)
        else # σταμάτα όταν ξεπεράσουμε το χρονικό όριο
            break
        end
    end
    times # επιστρέφουμε τον πίνακα με τους χρόνους
end

```

Algorithm 2.11. Αλγόριθμος που παράγει διαδικασία Poisson χρησιμοποιώντας την εκθετική κατανομή.

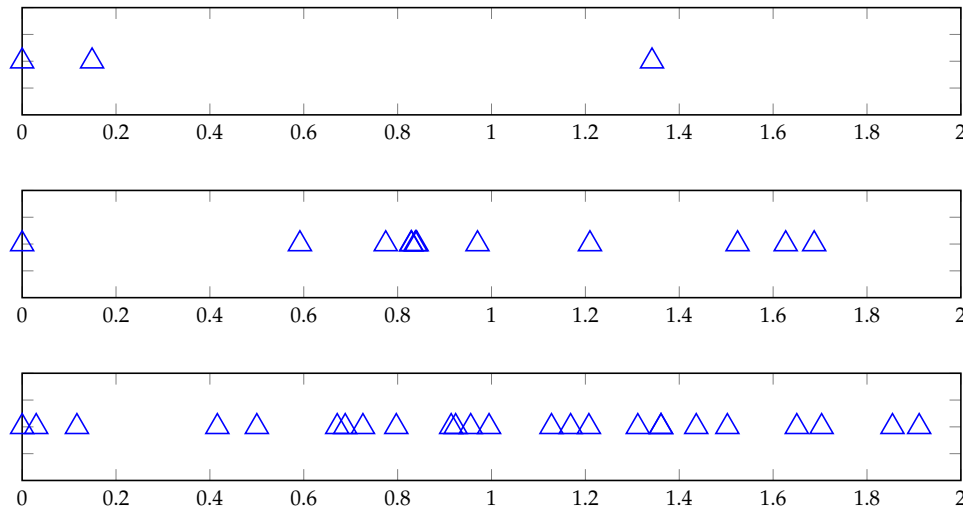


Figure 2.10. Τρεις διαδικασίες Poisson, με $\lambda = 2$, $\lambda = 6$, $\lambda = 10$. Τα τρίγωνα αναπαριστούν γεγονότα. Παρατηρήστε ότι όσο αυξάνεται ο ρυθμός της διαδικασίας τόσο περισσότερα γεγονότα συμβαίνουν στο ίδιο χρονικό διάστημα.

```
function poisson_point_process_transform( $\lambda$ ,  $\tau_0$ ,  $\tau$ )
    #= φτιάχνουμε πίνακα με floats που θα αποθηκεύει τους χρόνους
    των γεγονότων =#
    times::Array{Float64, 1} = [0.]
    while true # συνεχόμενη επανάληψη
        #= υπολογισμός διαφοράς χρόνων με χρήση της συνάρτησης
        rand και του inverse transform sampling =#
        interevent = (-log(rand())) /  $\lambda$ 
        # εφόσον δεν έχουμε ξεπεράσει το χρονικό όριο
        if times[end] + interevent <  $\tau$ 
            #= πρόσθεσε στο προηγούμενο χρόνο την διαφορά και
            αποθήκευσε το αποτέλεσμα στο τέλος του πίνακα =#
            push!(times, times[end] + interevent)
        else # σταμάτα όταν ξεπεράσουμε το χρονικό όριο
            break
        end
    end
    times # επιστρέφουμε τον πίνακα με τους χρόνους
end
```

Algorithm 2.12. Αλγόριθμος που παράγει διαδικασία Poisson χρησιμοποιώντας inverse transform sampling.

2.4 Διαδικασία Ornstein-Uhlenbeck

Η διαδικασία Ornstein-Uhlenbeck είναι μία στοχαστική διαδικασία που ικανοποιεί την παρακάτω στοχαστική εξίσωση:

$$dY_t = \theta \cdot (\mu - Y_t) \cdot dt + \sigma \cdot dB_t \quad (2.16)$$

Η διαδικασία Ornstein-Uhlenbeck προτάθηκε από τους Ornstein και Uhlenbeck το 1930 ¹¹ ως μία διαδικασία μοντελοποίησης της ταχύτητας ενός σωματιδίου που ακολουθεί κίνηση Brown. Σήμερα χρησιμοποιείται στα χρηματοοικονομικά μαθηματικά ως το μοντέλο Vasicek ¹² για την μοντελοποίηση των επιτοκίων, επιπλέον στην φυσική αναφέρεται συχνά ως εξίσωση Langevin και μοντελοποιεί θορυβώδεις διαδικασίες χαλάρωσης (noisy relaxation processes). Το κύριο χαρακτηριστικό της διαδικασίας Ornstein-Uhlenbeck είναι ότι μακροπρόθεσμα τείνει να επιστρέφει σε μία κατάσταση ισορροπίας (equilibrium) μ με ταχύτητα θ . Διαδικασίες που παρουσιάζουν τέτοια συμπεριφορά ονομάζονται mean-reverting. Οι παράγοντες που συνδράμουν στην παραπάνω εξίσωση (2.17) είναι οι εξής:

- Y_t : έτσι συμβολίζουμε τη διαδικασία Ornstein-Uhlenbeck
- θ : παράγοντας ταχύτητας επιστροφής στην κατάσταση ισορροπίας,
- μ : είναι η κατάσταση ισορροπίας ή αλλιώς ο μέσος όρος, η διαδικασία τείνει μακροπρόθεσμα να κινείται γύρω από αυτή την τιμή,
- σ : ο όρος θορύβου ή αλλιώς η κανονική απόκλιση, όσο μεγαλύτερη είναι η τιμή αυτού του παράγοντα τόσο μεγαλύτερο αντίκτυπο έχουν οι τυχαίες διακυμάνσεις στη διαδικασία,
- dB_t : αυτός ο παράγοντας εκφράζει ανεξάρτητες και όμοια κατανεμημένες αυξήσεις κίνησης Brown. Πιο συγκεκριμένη αναφέρεται στην κίνηση Brown όπως την ορίσαμε στην εξίσωση 2.8 και στον αλγόριθμο 2.9. Το πρόθεμα d εκφράζει διαφορές της κίνησης Brown, οι οποίες όπως είδαμε στο αντίστοιχο κεφάλαιο είναι ομοιόμορφα κατανεμημένες, δηλαδή $W_{t_{n+1}} - W_{t_n} = \Delta W_n \sim N(0, \Delta t) = \sqrt{\Delta t} N(0, 1)$

Σύμφωνα με τα παραπάνω λοιπόν, η τελική εξίσωση την οποία θα υλοποιήσουμε με αλγόριθμο είναι η εξής:

$$dY_t = \theta \cdot (\mu - Y_t) \cdot dt + \sigma \cdot \sqrt{dt} \cdot N(0, 1) \quad (2.17)$$

¹¹ G. E. Uhlenbeck and L. S. Ornstein, "On the Theory of the Brownian Motion," Phys. Rev., vol. 36, pp. 823–841, 5 1930. [Online]. Available: <https://link.aps.org/doi/10.1103/PhysRev.36.823>.

¹² O. Vasicek, "An Equilibrium Characterization of the Term Structure," Journal of Financial Economics, vol. 5, no. 2, pp. 177–188, 1977. [Online]. Available: <https://EconPapers.repec.org/RePEc:eee:jfinec:v:5:y:1977:i:2:p:177-188>.

Ο αλγόριθμος 2.13 υλοποιεί την εξίσωση 2.17 και επιπλέον προσθέτει τη δυνατότητα ρύθμισης της αρχικής θέσης x_0 της διαδικασίας.

```
function ornstein_uhlenbeck_process(total_time, dt, x0, μ, σ, θ)
    #= υπολογίζουμε πόσα συνολικά βήματα θα κάνουμε και
    χρησιμοποιούμε την συνάρτηση floor για να αποφύγουμε δεκαδικό
    αριθμό βημάτων =#
    total_steps = floor(Int64, total_time / dt)
    # δημιουργούμε πίνακα με τους χρόνους των βημάτων
    X = range(0, step = dt, length = total_steps) |> collect
    #= δημιουργούμε άδειο πίνακα για να αποθηκεύσουμε τις τιμές
    της διαδικασίας στους χρόνους X =#
    Y = Array{Float64, 1}(undef, total_steps)
    Y[1] = x0 # θέτουμε την αρχική τιμή της διαδικασίας
    for i = 2:total_steps # για κάθε ένα από τα υπόλοιπα βήματα
        #= υπολογίζουμε το επόμενο βήμα σύμφωνα με την εξίσωση,
        το προσθέτουμε στο προηγούμενο, αποθηκεύουμε στην
        κατάλληλη i-οστή θέση του πίνακα τιμών Y =#
        new_step = θ * (μ - Y[i-1]) * dt + sqrt(dt) * σ *
        rand(Normal(0, 1))
        Y[i] = Y[i-1] + new_step
    end
    (X, Y) # επιστρέφουμε πλειάδα (tuple) των χρόνων και των τιμών
end
```

Algorithm 2.13. Αλγόριθμος που παράγει διαδικασία Ornstein-Uhlenbeck.

2.5 Το Μοντέλο Olami-Feder-Christensen για Σεισμούς

Το μοντέλο Olami-Feder-Christensen ¹³ ή όπως αναφέρεται συχνότερα OFC είναι ένα μοντέλο κυτταρικών αυτομάτων που αναπαράγει τις βασικές συμπεριφορές των σεισμών όπως αυτές περιγράφονται από τον νόμο Gutenberg-Richter ¹⁴. Το OFC απλοποιεί το προγενέστερο μοντέλο Burridge-Knopoff ¹⁵ και αποτελείται από ένα πλέγμα κελιών τα οποία αλληλεπιδρούν μεταξύ τους διαχέοντας ενέργεια. Αναλυτικότερα κάθε κελί αρχικοποιείται με μηδενική ενέργεια η οποία αυξάνεται σε κάθε χρονική στιγμή κατά μία σταθερά. Αυτό γίνεται έως ότου η ενέργεια του κελιού γίνει ίση ή μεγαλύτερη από μία σταθερή οριακή τιμή που συνήθως είναι διαφορετική για κάθε κελί. Όταν συμβεί αυτό, η ενέργεια διαχέεται στους οριζόντιους και κάθετους γείτονες του κελιού με έναν συντελεστή διάχυσης α . Η παραπάνω περιγραφή εμφανίζεται γραφικά

¹³ Z. Olami, H. J. S. Feder, and K. Christensen, “Self-Organized Criticality in a Continuous, Nonconservative Cellular Automaton Modeling Earthquakes,” *Physical Review Letters*, vol. 68, no. 8, p. 1244, 1992.

¹⁴ B. Gutenberg and C. F. Richter, “Earthquake Magnitude, Intensity, Energy, and Acceleration: (Second Paper),” *Bulletin of the Seismological Society of America*, vol. 46, no. 2, pp. 105–145, 1956.

¹⁵ R. Burridge and L. Knopoff, “Model and Theoretical Seismicity,” *Bulletin of the Seismological Society of America*, vol. 57, no. 3, pp. 341–371, 1967.

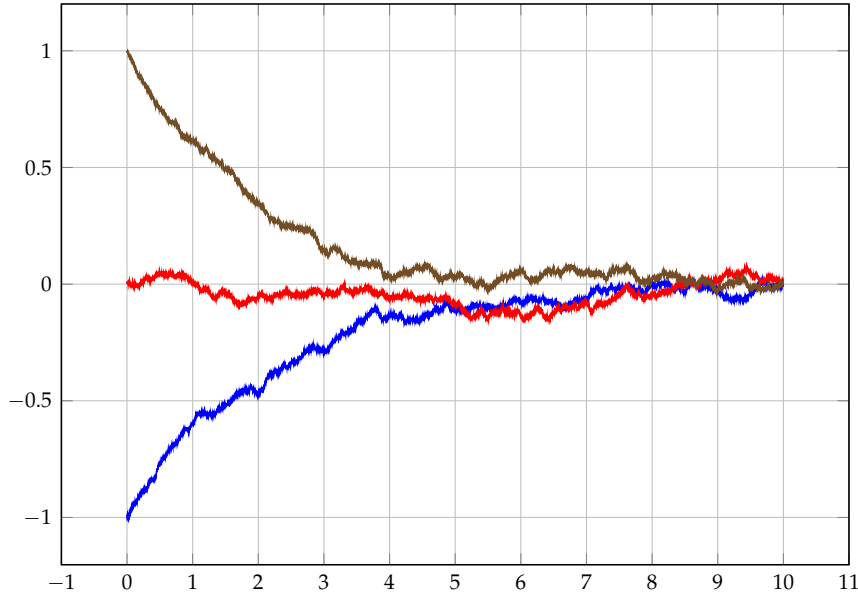


Figure 2.11. Τρεις προσομοιώσεις της διαδικασίας Ornstein-Uhlenbeck με $total_time = 10$, $dt = 0.001$, $\mu = 0$, $\sigma = 0.05$, $\theta = 0.5$ και $x_0 = -1$ (μπλε), $x_0 = 0$ (κόκκινο), $x_0 = 1$ (καφέ). Και τα τρία μονοπάτια συγκλίνουν στο μέσο της διαδικασίας.

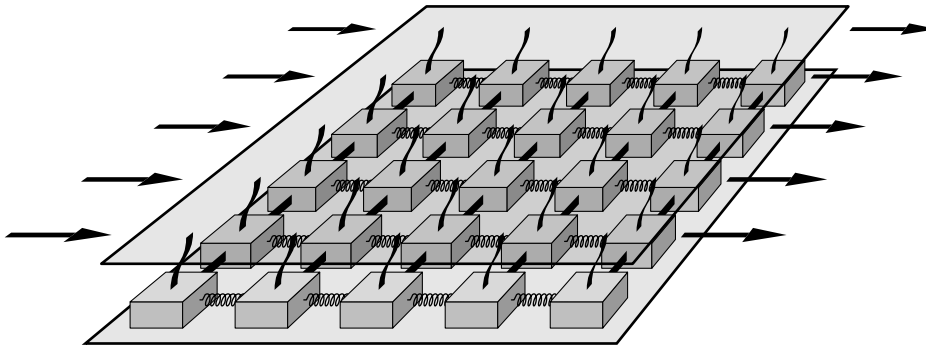


Figure 2.12. Γραφική απεικόνιση του μοντέλου OFC.

στην εικόνα 2.13 (εικόνα από ¹⁶). Στο κάτω μέρος εμφανίζεται ένα πλέγμα με κελιά τα οποία είναι συνδεδεμένα μεταξύ τους. Κάθε κελί συνδέεται με τον πάνω, κάτω, αριστερό και δεξιό γείτονα του. Αυτή η σύνδεση επιτρέπει τη διάχυση ενέργειας όπως περιγράφηκε παραπάνω. Επίσης όλα τα κελιά συνδέονται με μία κινούμενη πάνω πλάκα. Καθώς κινείται η πάνω πλάκα

¹⁶ S. Hergarten, "Landslides, Sand-piles, and Self-Organized Criticality," 2003.

αυξάνει την ένταση μεταξύ της και των κελιών τα οποία αντιστέκονται στην κίνηση καθώς είναι πάγια στην κάτω πλάκα. Αυτή η αύξηση της έντασης είναι η αιτία της αύξησης της ενέργειας του κάθε κελιού. Τέλος, κάποια στιγμή η κίνηση πλάκας υπερσχύει την αντίσταση του κελιού, κάτι το οποίο γίνεται όταν ένα κελί υπερβεί την οριακή τιμή ενέργειας του.

Το σύστημα που περιγράψαμε έχει απλούς κανόνες και παρόλα αυτά έχει τη δυνατότητα να παράγει πολύπλοκες συμπεριφορές, ιδιότητα που εμφανίζεται συχνά στα μοντέλα κυτταρικών αυτομάτων. Επίσης, το μοντέλο OFC χρησιμοποιείται εκτενώς για την προσομοίωση φυσικών φαινομένων και ιδιαίτερα στο πεδίο της σεισμολογίας για την προσομοίωση σεισμών^{17, 18}.

Το μέγεθος των σεισμών (κλίμακα Richter) ακολουθεί κατανομή Power Law, δηλαδή υπάρχουν λίγοι σεισμοί με μεγάλο μέγεθος και πολλοί σεισμοί με πολύ μικρό μέγεθος. Ο αλγόριθμος 2.14 υλοποιεί το μοντέλο OFC και στην εικόνα 2.13 παρουσιάζουμε τα αποτελέσματα μίας προσομοίωσής του σε σχέση με το μέγεθος των σεισμών που δημιουργήθηκαν. Είναι σημαντικό να σημειωθεί ότι ο κώδικας που παρουσιάζεται εδώ δεν περιέχει σχολιασμό για λόγους συντομίας, ο πλήρης κώδικας μπορεί να βρεθεί στο αποθετήριο του εγχειριδίου <https://github.com/spirosbax/textbook-companion>.

¹⁷ M. Avlonitis and G. Papadopoulos, “Foreshocks and B Value: Bridging Macroscopic Observations to Source Mechanical Considerations,” *Pure and Applied Geophysics*, vol. 171, no. 10, pp. 2569–2580, 2014.

¹⁸ S.-J. W. Bin-Quan Li, “Self-Organized Criticality in an Anisotropic Earthquake Model,” *Communications in Theoretical Physics*, vol. 69, no. 03, 280, p. 280, 2018. [Online]. Available: https://ctp.itp.ac.cn/EN/abstract/article_17223.shtml.

```

using Distributions, ImageFiltering
function ofc(N, timesteps, force_increment,  $\alpha$ )
    grid = zeros(N, N)
    critical_values = rand(Uniform(1,5), N, N)
    excited_cells = Array{Int64, 1}(undef, timesteps)
    for timestep in 1:timesteps
        grid .+= force_increment
        check_excited = critical_values .- grid
        replace!(x -> x <= 0 ? 1 : 0, check_excited)
        no_of_excited = sum(check_excited)
        while sum(check_excited) > 0
            grid += imfilter(grid .* check_excited,
                            centered([0  $\alpha$  0;
                                      $\alpha$  -1  $\alpha$ ;
                                     0  $\alpha$  0])),
                            Fill(zero(eltype(grid)), (1, 1)));
            check_excited = critical_values .- grid
            replace!(x -> x <= 0 ? 1 : 0, check_excited)
            no_of_excited += sum(check_excited)
        end
        excited_cells[timestep] = no_of_excited
    end
    excited_cells
end

```

Algorithm 2.14. Αλγόριθμος που υλοποιεί το μοντέλο Olami-Feder-Christensen. Δέχεται σαν είσοδο το μέγεθος της πλευράς του πλέγματος, τον αριθμό των χρονικών βημάτων *timesteps*, την σταθερά αύξησης ενέργειας *force_increment* και την σταθερά διάχυσης ενέργειας α . Επιστρέφει σαν έξοδο πίνακα που περιέχει τον αριθμό των κελιών που ξεπέρασαν την οριακή τους ενέργεια (έγιναν excited) την κάθε χρονική στιγμή. Αξίζει να σημειωθεί ότι για να απλοποιήσουμε τον αλγόριθμο χρησιμοποιούμε μόνο τετράγωνα πλέγματα.

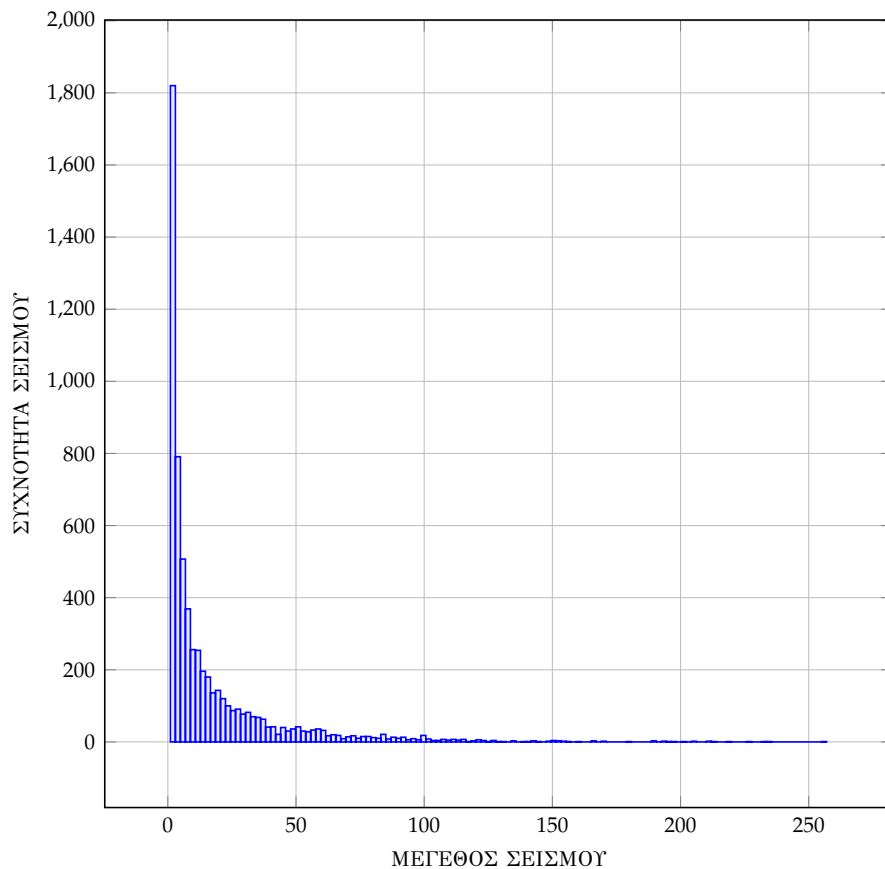


Figure 2.13. Αποτελέσματα μίας προσομοίωσης του μοντέλου Olami-Feder-Christensen με παραμέτρους $\mu = 20$, $timesteps = 10000$, $force_{increment} = 0.005$, $\alpha = 0.24$. Στον άξονα εμφανίζεται το μέγεθος του σεισμού μετρημένο σύμφωνα με τον αριθμό των κελιών που έγιναν excited. Στον άξονα εμφανίζεται η συχνότητα του σεισμού με μέγεθος. Βλέπουμε ότι το ιστόγραμμα ακολουθεί κατανομή Power Law και συμφωνεί με το πραγματικό φαινόμενο του σεισμού.

3 Μέθοδοι Υπολογιστικής Φυσικής

3.1 Πεπερασμένες Διαφορές σε Μερικές Διαφορικές Εξισώσεις

Στην φύση πολλά φαινόμενα εκφράζονται με μερικές διαφορικές εξισώσεις πολυμεταβλητών συναρτήσεων, δηλαδή συναρτήσεων που περιέχουν περισσότερες από μία μεταβλητές. Για παράδειγμα μία συνάρτηση που περιγράφει την εξέλιξη ενός σημείου σε τρισδιάστο χώρο μπορεί να εκφραστεί ως εξής: $f(x, y, z, t)$. Οι μερικές διαφορικές εξισώσεις προκύπτουν όταν θέλουμε να δούμε το πώς εξελίσσεται η συνάρτηση f ως προς μία ή περισσότερες διαστάσεις της. Δηλαδή αντί να παραγωγίσουμε την f ως προς όλες τις εισόδους της, παραγωγίζουμε ως προς κάποιες από αυτές. Με άλλα λόγια θα μπορούσαμε να πούμε ότι: Μία συνήθης διαφορική εξίσωση περιέχει παραγώγους μόνο ως προς μία μεταβλητή ενώ μία μερική διαφορική εξίσωση περιέχει παραγώγους ως προς πολλές ανεξάρτητες μεταβλητές.

Για να μοντελοποιήσει κανείς φυσικά φαινόμενα πρέπει να μελετήσει τις διαφορικές εξισώσεις που τα εκφράζουν. Κάποιες αξιοσημείωτες και συχνά χρησιμοποιούμενες μερικές διαφορικές εξισώσεις φυσικών φαινομένων είναι:

- Η εξίσωση της διάδοσης κυμάτων

$$\frac{\partial^2 u}{\partial t^2} = c^2 \nabla^2 u$$

όπου c η ταχύτητα διάδοσης του κύματος,

- Η εξίσωση της διάχυσης ή της θερμότητας

$$\frac{\partial u}{\partial t} = \alpha \nabla^2 u \quad (3.1)$$

όπου α ο συντελεστής θερμικής διάχυσης του υλικού,

- Η εξίσωση του Laplace

$$\nabla^2 u = 0$$

όπου

$$\nabla = \left(\frac{\partial}{\partial x_1} + \frac{\partial}{\partial x_2} + \cdots + \frac{\partial}{\partial x_n} \right)$$

και

$$\nabla^2 = \left(\frac{\partial^2}{\partial x_1^2} + \frac{\partial^2}{\partial x_2^2} + \cdots + \frac{\partial^2}{\partial x_n^2} \right)$$

Το σύμβολο ∇ ονομάζεται "nabla" και χρησιμοποιείται συχνά στις διαφορικές εξισώσεις. Επιπλέον άλλοι συμβολισμοί που συνηθίζονται περιλαμβάνουν, για παράδειγμα η εξίσωση της θερμότητας:

$$\frac{\partial u}{\partial t} = \alpha \nabla^2 u$$

μπορεί να γραφεί ως:

- Τελεστής Laplace αντί για nabla

$$\frac{\partial u}{\partial t} = \alpha \Delta u$$

όπου $\Delta = \nabla \cdot \nabla$,

- Απλός τελεστής παραγώγου

$$\frac{du}{dt} = \alpha \nabla^2 u$$

- Σύνθετος τελεστής μερικής παραγώγου

$$\partial_t u = \alpha \nabla^2 u$$

- Παράγωγος ως προς τον χρόνο με τελεία

$$\dot{u} = \alpha \nabla^2 u$$

όπου $\dot{u} = \frac{du}{dt}$.

Όπως είδαμε και στο προηγούμενο κεφάλαιο, ένας υπολογιστής δεν μπορεί να υπολογίσει σε συνεχείς χώρους. Για αυτό τον λόγο έχουν αναπτυχθεί αριθμητικές τεχνικές διακριτοποίησης όπως η μέθοδος Πεπερασμένων Στοιχείων και η μέθοδος Πεπερασμένων Διαφορών. Την τελευταία θα μελετήσουμε σε αυτό το υποκεφάλαιο.

Πριν προχωρήσουμε όμως στην μέθοδο των Πεπερασμένων Διαφορών όμως ας θυμηθούμε πρώτα έναν από τους ορισμούς της παραγώγου:

$$\frac{df}{dx} = \lim_{dx \rightarrow 0} \frac{f(x+dx) - f(x)}{dx} \quad (3.2)$$

Αυτός ο ορισμός μας λέει ότι εάν θέλουμε να πάρουμε την παράγωγο μίας συνάρτησης σε ένα σημείο x τότε παίρνουμε την διαφορά των τιμών της συνάρτησης f στα σημεία $x+dx$ και x και διαιρούμε διά dx . Καθώς το dx , δηλαδή η διαφορά στον άξονα των x , γίνεται απειροελάχιστο τότε θα έχουμε την πραγματική τιμή της παραγώγου. Για να το κατανοήσουμε αυτό είναι σημαντικό να θυμόμαστε ότι η παράγωγος εκφράζει ρυθμό αλλαγής.

Με την ίδια λογική του ορισμού 3.2 λειτουργούν και οι ορισμοί:

$$\frac{df}{dx} = \lim_{dx \rightarrow 0} \frac{f(x+dx) - f(x-dx)}{2dx} \quad (3.3)$$

αλλά και

$$\frac{df}{dx} = \lim_{dx \rightarrow 0} \frac{f(x) - f(x-dx)}{dx} \quad (3.4)$$

Ανακεφαλαιώνοντας έχουμε:

- Ο ορισμός 3.2 αφορά διαφορά μεταξύ $f(x+dx)$ και $f(x)$
- Ο ορισμός 3.3 αφορά διαφορά μεταξύ $f(x+dx)$ και $f(x-dx)$
- Ο ορισμός 3.4 αφορά διαφορά μεταξύ $f(x)$ και $f(x-dx)$

Οι παραπάνω ορισμοί αφορούν συνεχείς συναρτήσεις και συνεχείς χώρους. Το πρώτο βήμα λοιπόν στην μέθοδο των πεπερασμένων διαφορών είναι η διακριτοποίηση. Έτσι θα ορίσουμε την παράγωγο πεπερασμένων διαφορών προς τα εμπρός ως:

$$\frac{df}{dx} \approx \frac{f(x+dx) - f(x)}{dx} \quad (3.5)$$

την κεντρική παράγωγο πεπερασμένων διαφορών ως:

$$\frac{df}{dx} \approx \frac{f(x+dx) - f(x-dx)}{2dx} \quad (3.6)$$

και την παράγωγο πεπερασμένων διαφορών προς τα πίσω ως:

$$\frac{df}{dx} \approx \frac{f(x) - f(x-dx)}{dx} \quad (3.7)$$

Οι ορισμοί 3.5, 3.5, 3.5 είναι προσεγγίσεις των ορισμών 3.2, 3.3, 3.4 αφού έχουμε αφαιρέσει το όριο. Συμπεραίνουμε λοιπόν, ότι στους προσεγγιστικούς υπολογισμούς μας όσο μικρότερο είναι το βήμα dx τόσο καλύτερη η προσέγγιση στην πραγματική τιμή της παραγώγου.

Το επόμενο βήμα προς την μέθοδο των πεπερασμένων διαφορών είναι να ορίσουμε μαθηματικά πόσο καλές είναι αυτές οι προσεγγιστικές παράγωγοι. Για να το κάνουμε αυτό θα χρησιμοποιήσουμε σειρές Taylor. Η σειρά Taylor μίας συνάρτησης $f(x)$ ως προς ένα σημείο x_0 είναι:

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \frac{f'''(x_0)}{3!}(x - x_0)^3 + \dots \quad (3.8)$$

θέτουμε $dx = x - x_0 \Rightarrow x = x_0 + dx$ και αντικαθιστούμε στην 3.8:

$$f(x_0 + dx) \approx f(x_0) + f'(x_0)dx + \frac{f''(x_0)}{2!}dx^2 + \frac{f'''(x_0)}{3!}dx^3 + \dots \quad (3.9)$$

τέλος αντικαθιστούμε το x_0 με x και έχουμε:

$$\begin{aligned} f(x+dx) &= f(x) + \partial_x f dx + \frac{1}{2!} \partial_x^2 f dx^2 + \frac{1}{3!} \partial_x^3 f dx^3 + \dots \\ &= \sum_{n=0}^{\infty} \frac{1}{n!} \partial_x^n f dx^n \end{aligned} \quad (3.10)$$

όπου $\partial_x = \frac{d}{dx}$.

Έπειτα κατασκευάζουμε το δεξί μέρος της 3.5 στο αριστερό μέρος της 3.10:

$$\begin{aligned}
 f(x+dx) &= f(x) + f'dx + \frac{1}{2!}f''dx^2 + \frac{1}{3!}f'''dx^3 + \dots \Rightarrow \\
 \frac{f(x+dx) - f(x)}{dx} &= f' + \frac{1}{2!}f''dx + \frac{1}{3!}f'''dx^2 + \dots \Rightarrow \\
 \frac{f(x+dx) - f(x)}{dx} &= f' + O(dx) \Rightarrow \\
 \frac{f(x+dx) - f(x)}{dx} &\approx f'
 \end{aligned} \tag{3.11}$$

Διαπιστώνουμε λοιπόν ότι η πεπερασμένη διαφορά προς τα εμπρός είναι ακριβής μέχρι και όρους πρώτης τάξεως. Το γεγονός αυτό θα γίνει πιο ουσιώδες αφού κάνουμε την ανάλυση σφάλματος και για τις εξισώσεις 3.7, 3.6. Χάριν συντομίας δεν θα αναπτυχθεί η σειρά Taylor της πεπερασμένης διαφοράς προς τα πίσω καθώς έχει σφάλμα ίδιας τάξεως με την διαφορά προς τα εμπρός που μόλις είδαμε. Αντ' αυτού θα επικεντρωθούμε στην πιο ενδιαφέρον κεντρική πεπερασμένη διαφορά. Γνωρίζουμε από την 3.11 ότι:

$$f(x+dx) = f(x) + \partial_x f dx + \frac{1}{2!}\partial_x^2 f dx^2 + \frac{1}{3!}\partial_x^3 f dx^3 + \dots \tag{3.12}$$

και

$$f(x-dx) = f(x) - \partial_x f dx + \frac{1}{2!}\partial_x^2 f dx^2 - \frac{1}{3!}\partial_x^3 f dx^3 + \dots \tag{3.13}$$

αφαιρούμε την 3.13 από την 3.12 και διαιρούμε διά $2dx$ ώστε να φτιάξουμε το δεξί μέρος της 3.6:

$$\begin{aligned}
 f(x+dx) - f(x-dx) &= f(x) - f(x) + \partial_x f dx + \partial_x f dx + \frac{1}{2!}\partial_x^2 f dx^2 \\
 &\quad - \frac{1}{2!}\partial_x^2 f dx^2 + \frac{1}{3!}\partial_x^3 f dx^3 + \frac{1}{3!}\partial_x^3 f dx^3 + \dots \\
 \Rightarrow \frac{f(x+dx) - f(x-dx)}{2dx} &= \partial_x f + \frac{1}{3!}\partial_x^3 f dx^2 + \frac{1}{5!}\partial_x^5 f dx^4 + \dots \\
 \Rightarrow \frac{f(x+dx) - f(x-dx)}{2dx} &= \partial_x f + O(dx^2)
 \end{aligned}$$

```

# πακέτο που προσφέρει βοηθητικές στατιστικές συναρτήσεις
using StatsBase
function sine_finite_diff(L, nx, λ)
    #= πρώτα υπολογίζουμε το dx, υπάρχουν nx σημεία στον χώρο
    άρα υπάρχουν nx - 1 διαστήματα dx, διαιρούμε το L διά
    τον αριθμό τους για να βρούμε το μήκος τους =#
    dx = L / (nx - 1)
    X = collect(range(0, L, length = nx))
    #= το λ είναι ένας ακέραιος αριθμός που εκφράζει το μήκος
    του κύματος σε μονάδες dx, δηλαδή εάν έχουμε σαν είσοδο
    λ = 25, τότε πολλαπλασιάζουμε επί dx ώστε το μήκος του
    κύματος να είναι 25dx =#
    λ *= dx
    #= το k περιέχει το 2π το οποίο θα μετατρέψει τους
    αριθμούς του X σε ακτίνια και το 1 / λ το οποίο αναπαριστά
    την συχνότητα του κύματος. θυμηθείτε ότι
    συχνότητα = 1 / περίοδο =#
    k = 2π / λ
    #= εφαρμόζουμε την συνάρτηση του ημιτόνου ώστε να
    δημιουργήσουμε διακριτό ημιτονοειδές κύμα =#
    F = sin.(k * X)
    #= υπολογίζουμε την αναλυτική παράγωγο, θυμηθείτε ότι
    sin(kx)' = kcos(kx) =#
    ad = k * cos.(k * X)
    #= δημιουργούμε άδείο πίνακα μηδενικών μήκους nx στον
    οποίο θα βάλουμε τα αποτελέσματα =#
    nd = zeros(nx)
    #= για κάθε τιμή του κύματος εφαρμόζουμε την
    Κεντρική Πεπερασμένη Διαφορά=#
    for i in 2:nx-1
        nd[i] = (F[i+1] - F[i-1]) / 2dx
    end
    #= υπολογίζουμε το Root Mean Squared Error ώστε να
    αξιολογήσουμε το σφάλμα της προσέγγισης σε σχέση με
    την πραγματική παράγωγο =#
    rmse = ad[2:end-1] - nd[2:end-1] .|> (x->x^2) |> mean |> sqrt
    #= επιστρέφουμε πλειάδα τεσσάρων στοιχείων που περιέχει
    τα στοιχεία του άξονα X, τον πίνακα με την αναλυτική παράγωγο,
    τον πίνακα με την αριθμητική παράγωγο και το rmse =#
    (X, ad, nd, rmse)
end

```

Algorithm 3.1. Αλγόριθμος που δέχεται σαν είσοδο το συνολικό μήκος του κύματος L , τον συνολικό αριθμό των σημείων στον χώρο nx και το λ που αναπαριστά το μήκος κύματος. Έπειτα υπολογίζει την αναλυτική και την αριθμητική πρόοδο, συγκρίνει τα αποτελέσματα και τα επιστρέφει σε πλειάδα.

από τις παραπάνω εξισώσεις βλέπουμε ότι η κεντρική πεπερασμένη διαφορά είναι ακριβής μέχρι και όρους δεύτερης τάξεως. Αυτό την καθιστά πιο ακριβής από την διαφορά προς τα εμπρός και προς τα πίσω και για αυτό χρησιμοποιείται πιο συχνά σε εφαρμογές.

Σε αυτό το σημείο είναι καλό να χρησιμοποιήσουμε όλα όσα μάθαμε για την μέθοδο Πεπερασμένων Διαφορών σε ένα παράδειγμα πάνω σε μία ημιτονοειδή συνάρτηση. Ο αλγόριθμος 3.1 εφαρμόζει την Κεντρική Πεπερασμένη Διαφορά (εξίσωση 3.6) στην συνάρτηση $\sin(kx)$ και συγκρίνει την προσεγγιστική αριθμητική παράγωγο που προκύπτει με την ακριβή αναλυτική παράγωγο $k\cos(kx)$. Όπου $k = \frac{2\pi}{\lambda}$. Στην εικόνα 3.1 παρουσιάζονται τα αποτελέσματα και στην εικόνα 3.2 γίνεται ανάλυση του σφάλματος της αριθμητικής παραγώγου σε σχέση με τον αριθμό σημείων ανά μήκος κύματος.

3.2 Αριθμητική Επίλυση της Εξίσωσης Διάχυσης Θερμότητας

Σε αυτό το υποκεφάλαιο θα χρησιμοποιήσουμε την μέθοδο των Πεπερασμένων Διαφορών για να επιλύσουμε αριθμητικά ένα πρόβλημα διάχυσης θερμότητας σε μία διάσταση. Το πρώτο βήμα είναι να προσαρμόσουμε την εξίσωση διάχυσης θερμότητας 3.1 σε μία διάσταση. Το σύμβολο ∇ εκφράζει μερική παράγωγο ως προς κάθε διάσταση οπότε αφού επιλέγουμε να χρησιμοποιήσουμε μόνο μία διάσταση η 3.1 εξελίσσεται ως εξής:

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2} \quad (3.14)$$

Όπου u είναι η συνάρτηση $u(x, t)$ που μας δίνει την θερμότητα σε ένα σημείο x κατά την χρονική στιγμή t . Την παραπάνω εξίσωση καλούμαστε να εκφράσουμε αποκλειστικά με τις εξισώσεις πεπερασμένων διαφορών που έχουμε δει 3.5, 3.6, 3.7 αλλά και μία εξίσωση πεπερασμένων διαφορών που θα αναπτύξουμε στην συνέχεια για την δεύτερη παράγωγο του u ως προς x που υπάρχει στο δεξί μέρος της εξίσωσης 3.14. Για την δεύτερη παράγωγο εφαρμόζουμε τον ίδιο τρόπο σκέψης της πρώτης παραγώγου όπου χρησιμοποιήσαμε την διαφορά μεταξύ δύο σημείων της f και διαιρέσαμε διά dx . Έτσι έχουμε:

$$f'' \approx \frac{\frac{f(x+dx)-f(x)}{dx} - \frac{f(x)-f(x-dx)}{dx}}{dx} \quad (3.15)$$

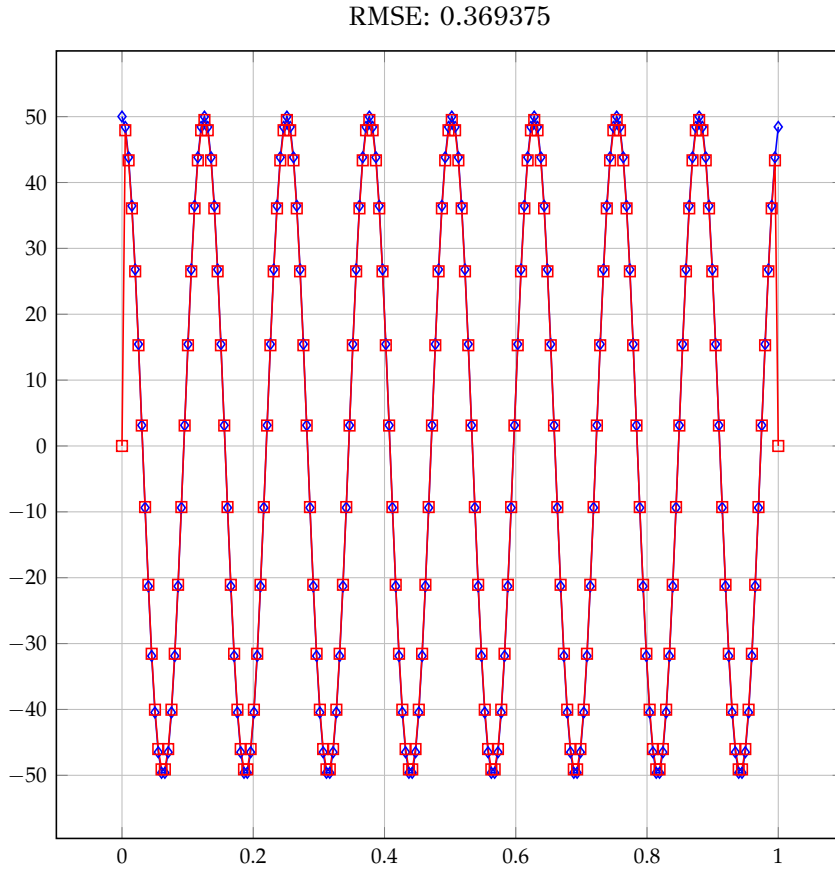


Figure 3.1. Σύγκριση της αναλυτικής (μπλε) και της αριθμητικής (κόκκινο) παραγωγού.

με λίγες αλγεβρικές πράξεις καταλήγουμε στην εξίσωση της δεύτερης παραγωγού με πεπερασμένες διαφορές:

$$\frac{d^2 f}{dx^2} \approx \frac{f(x+dx) - 2f(x) + f(x-dx)}{dx^2} \quad (3.16)$$

Για να διευκολύνουμε την διακριτοποίηση της εξίσωσης 3.14 θα υιοθετήσουμε την εξής σημειογραφία (notation):

Συνεχής		Διακριτό
$u(x, t)$	$\rightarrow$	u_j^n
$u(x_j, t_n + dt)$	$\rightarrow$	u_j^{n+1}

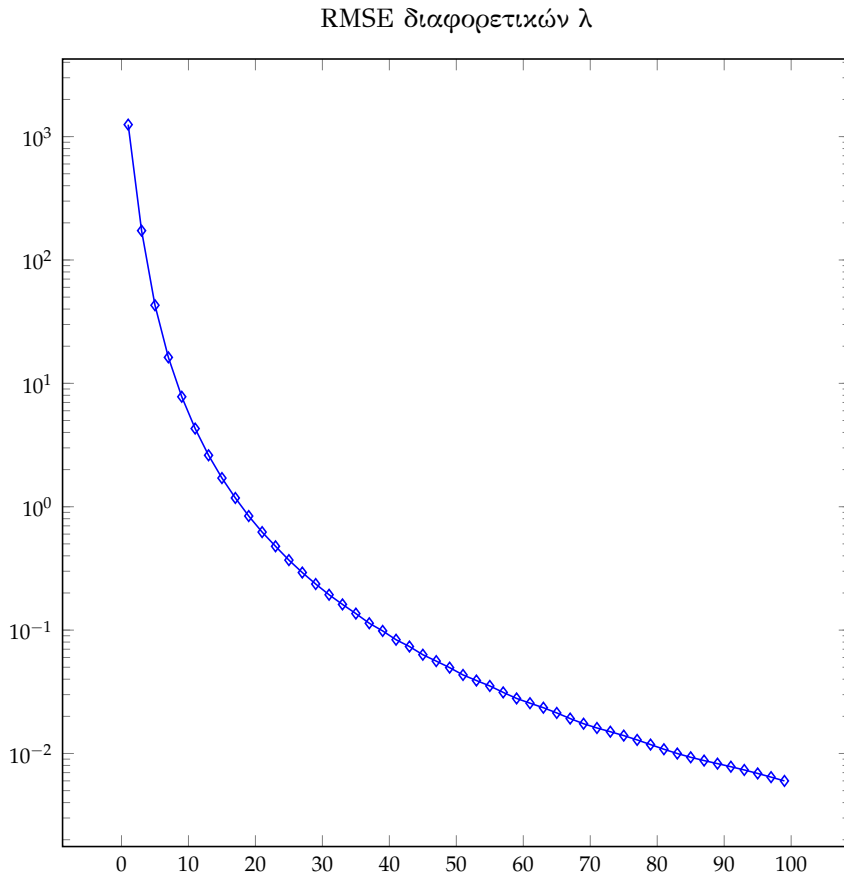


Figure 3.2. Σύγκριση του Root Mean Squared Error για διαφορετικές τιμές του λ από το 1 μέχρι το 100 με βήμα 2. Ο άξονας y είναι σε λογαριθμική κλίμακα. Φαίνεται ότι όσο περισσότερα αυξάνονται τα σημεία ανά μήκος κύματος τόσο μικρότερο γίνεται το σφάλμα.

όπου u_j^n η θερμοκρασία στο σημείο $x = j$ την χρονική στιγμή $t = n$. Εφαρμόζοντας τις εξισώσεις πεπερασμένων διαφορών 3.5, 3.6, 3.7, 3.14 καθώς και την νέα σημειογραφία στην μονοδιάστατη εξίσωση διάχυσης θερμότητας 3.14, έχουμε:

$$\begin{aligned} \frac{u(x, t + dt) - u(x, t)}{dt} &= \alpha \frac{u(x + dx, t) - 2u(x, t) + u(x - dx, t)}{dx^2} \\ \Rightarrow \frac{u_j^{n+1} - u_j^n}{dt} &= \alpha \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{dx^2} \end{aligned}$$

έπειτα λύνουμε ως προς u_j^{n+1} :

$$\begin{aligned} \frac{u_j^{n+1} - u_j^n}{dt} &= \alpha \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{dx^2} \\ \Rightarrow u_j^{n+1} &= \frac{\alpha dt}{dx^2} (u_{j+1}^n - 2u_j^n + u_{j-1}^n) + u_j^n \end{aligned} \quad (3.17)$$

Σύμφωνα με την εξίσωση 3.17 μπορούμε να βρούμε την θερμοκρασία σε ένα σημείο $x = j$ την χρονική στιγμή $n + 1$ χρησιμοποιώντας την θερμοκρασία στα σημεία $j - 1, j, j + 1$ την χρονική στιγμή n . Το γεγονός αυτό αναπαρίσταται

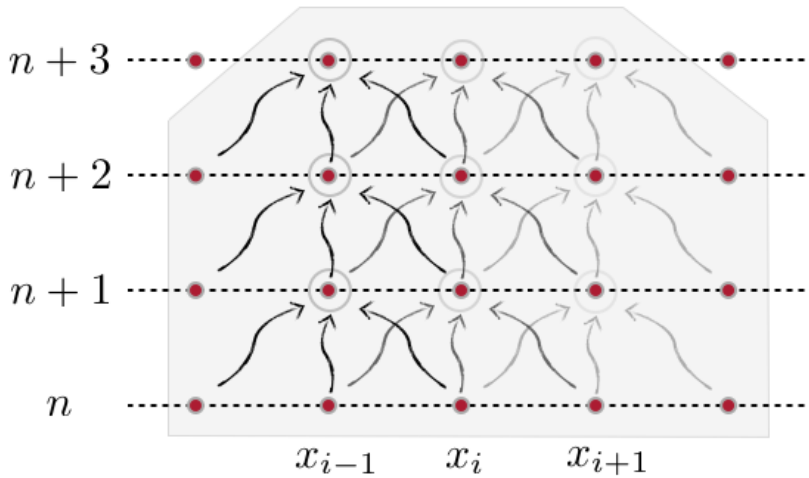


Figure 3.3. Γραφική απεικόνιση της εξίσωσης διάχυσης θερμότητας με πεπερασμένες διαφορές για τρεις χρονικές στιγμές και τρία σημεία στον χώρο. Η θερμοκρασία ένα σημείο x_i την χρονική στιγμή $n + 1$ επηρεάζεται από την θερμοκρασία της προηγούμενης χρονική στιγμής n στα σημεία x_{i-1}, x_i, x_{i+1} . (Copyright (c) Barbagroup).

γραφικά στην εικόνα 3.3.

Με την βοήθεια των εργαλείων που αναπτύξαμε, θα δημιουργήσουμε αλγόριθμο που θα μας δείξει το πώς διαχέεται η θερμοκρασία σε μία ράβδο γραφίτη, το αριστερό άκρο της οποίας θα έχει σταθερή θερμοκρασία 100°C και το δεξί της άκρο 0°C . Ο αλγόριθμος 3.2 υλοποιεί αυτό το σενάριο και τα αποτελέσματα για διάφορες χρονικές στιγμές φαίνονται στην εικόνα 3.4. Τέλος, ένα βίντεο της προσομοίωσης υπάρχει στο αποθετήριο του βιβλίου στον

εξής σύνδεσμο <https://github.com/spirosbax/textbook-companion/blob/master/rod.gif>.

Η μέθοδος διακριτοποίησης που έχουμε συζητήσει σε αυτό το υποκεφάλαιο και υλοποιείται στον αλγόριθμο 3.2 έχει έναν περιορισμό σταθερότητας. Συγκεκριμένα η σταθερότητα της αριθμητικής επίλυσης απαιτεί:

$$0 < \alpha \frac{\Delta t}{\Delta x^2} \leq \frac{1}{2} \quad (3.18)$$

η ποσότητα αυτή στον αλγόριθμο 3.2 δίνεται από την παράμετρο σ . Τέλος το Δt (dt) υπολογίζεται ως εξής:

$$\begin{aligned} \sigma &= \alpha \frac{\Delta t}{\Delta x^2} \\ \Rightarrow \Delta t &= \sigma \frac{\Delta x^2}{\alpha} \end{aligned}$$

3.3 Οριακές και Αρχικές Συνθήκες

Αρχική συνθήκη ονομάζουμε τις αρχικές τιμές του συστήματος πριν ξεκινήσει ο υπολογισμός των τιμών της χρονικής στιγμής $n + 1$. Στον αλγόριθμο 3.2 οι αρχικές συνθήκες είναι ένας μηδενικός πίνακας εκτός από την πρώτη και την τελευταία θέση που αναπαριστούν την θερμοκρασία στο αριστερό και το δεξί άκρο της ράβδου αντίστοιχα. Εκτός από τις αρχικές συνθήκες, οι οριακές συνθήκες (boundary conditions) εκφράζουν το πώς αλληλεπιδρά το σύστημα με τα όρια του χώρου. Οι δύο πιο συχνά χρησιμοποιούμενοι τύποι οριακών συνθηκών είναι οι οριακές συνθήκες Dirichlet και οι οριακές συνθήκες Neumann. Οι οριακές συνθήκες Dirichlet αφορούν την διατήρηση ενός ή περισσότερων ορίων του χώρου σε σταθερές συγκεκριμένες τιμές, σε αντίθεση με την μεταβλητή αριθμητική επίλυση που λαμβάνει χώρα στον υπόλοιπο χώρο. Συνθήκη Dirichlet χρησιμοποιούμε και στα δύο όρια του αλγορίθμου 3.2 αφού η θερμοκρασία παραμένει σταθερή στα δύο άκρα. Για παράδειγμα στην εικόνα 3.4 το αριστερό και το δεξί άκρο παραμένουν στους 100°C και 0°C αντίστοιχα. Ανάλογα, οι οριακές συνθήκες Neumann αφορούν την διατήρηση της τιμής της πρώτης παραγώγου ενός ή περισσότερων ορίων του χώρου σε σταθερές. Αυτό εκφράζεται μαθηματικά ως εξής:

$$\left. \frac{\partial T}{\partial x} \right|_{x=1} = q(t)$$

```

function heat_equation_rod(L, nx, nt, σ, α, bcs)
    #= πρώτα υπολογίζουμε το dx, υπάρχουν nx σημεία στον χώρο=#
    #= υπολογίζουμε το dx όπως και στο παράδειγμα με την
    ημιτονοειδή συνάρτηση=#
    dx = L / (nx - 1)
    # παράγουμε πίνακα με τα σημεία του άξονα X
    X = collect(range(0, L, length = nx))
    #= ο τρόπος υπολογισμού του dt περιγράφεται αναλυτικά στο
    κείμενο=#
    dt = σ * (dx ^ 2) / α
    #= οριακές συνθήκες για την ράβδο, θερμοκρασία στο αριστερό
    και το δεξί μέρος=#
    (bc_left, bc_right) = bcs
    #= πίνακας μηδενικών που θα αποθηκευτούν τα αποτελέσματα,
    θα έχουμε nt χρονικές στιγμές - γραμμές και nx διακριτά σημεία
    πάνω στην ράβδο=#
    u = zeros(nt, nx)
    # θέτουμε τις οριακές συνθήκες για κάθε χρονική στιγμή
    u[:, 1] .= bc_left
    u[:, end] .= bc_right
    #= για κάθε χρονική στιγμή εκτός από την πρώτη (καθώς η πρώτη
    περιέχει τις αρχικές συνθήκες του προβλήματος) και για κάθε
    σημείο της ράβδου εκτός από τις άκρες (καθώς οι άκρες
    περιέχουν τις οριακές συνθήκες) εφάρμοσε την εξίσωση
    διάχυσης θερμότητας για το σημείο j την επόμενη χρονική στιγμή
    n+1=#
    for n = 1:nt-1
        for j = 2:nx-1
            u[n+1, j] = α * dt / dx^2 * (u[n, j+1] - 2u[n, j] +
            u[n, j-1]) + u[n, j]
        end
    end
    # επιστρέφουμε τον πίνακα με τα αποτελέσματα
    u
end

```

Algorithm 3.2. Αλγόριθμος που υλοποιεί την μονοδιάστατη εξίσωση της θερμότητας σε μία ράβδο που τα δύο άκρα της διατηρούνται σε σταθερές θερμοκρασίες. Η συνάρτηση δέχεται ως είσοδο το μήκος της ράβδου L , τον συνολικό αριθμό σημείων του χώρου n_x , των συνολικό αριθμό χρονικών στιγμών n_t , το σ που αναπαριστά το μήκος του χρονικού βήματος dt , το α που είναι η σταθερά διάχυσης θερμότητας του υλικού και μία πλειάδα bcs που περιέχει τις οριακές συνθήκες στο αριστερό και το δεξιό άκρο της ράβδου αντίστοιχα.

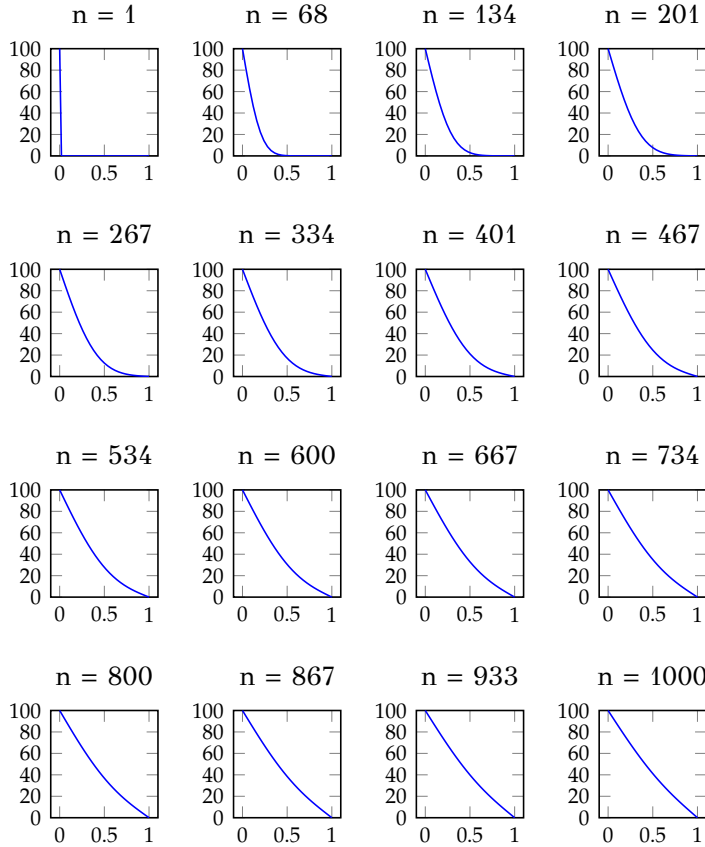


Figure 3.4. Εξέλιξη της διάχυσης της θερμότητας σε μία ράβδο γραφίτη μήκους ενός μέτρου για 1000 χρονικές στιγμές. Οι παράμετροι που χρησιμοποιήθηκαν ήταν: $L = 1$, $n_x = 51$, $n_t = 1000$, $\sigma = 0.5$, $\alpha = 1.22e-3$, $bcs = (100, 0)$. Η σταθερή λύση της συγκεκριμένης διαφορικής εξίσωσης είναι μία ευθεία γραμμή με αρχικό και τελικό σημείο την αριστερή και δεξιά θερμοκρασία της ράβδου αντίστοιχα. Αφού η λύση φτάσει σε αυτή την σταθερή κατάσταση, το σχήμα της δεν θα αλλάξει όσες χρονικές στιγμές και να περάσουν. Στην εικόνα φαίνεται ότι η καμπύλη της λύσης πλησιάζει αυτή την ευθεία γραμμή.

και εφαρμόζεται στην μέθοδο πεπερασμένων διαφορών ως εξής:

$$\left. \frac{\partial T}{\partial x} \right|_{x=1} \approx \frac{T_N^n - T_{N-1}^n}{\Delta x} = q(t)$$

όπου $q(t)$ μία συνάρτηση με μόνη παράμετρο τον χρόνο. Στον αλγόριθμο 3.2 δεν χρησιμοποιήσουμε τέτοιου τύπου οριακές συνθήκες. Τέλος είναι σημαντικό να αναφέρουμε ότι συχνά χρησιμοποιούνται και οι δύο τύποι οριακών συνθηκών στο ίδιο πρόβλημα.

Βιβλιογραφία

1. M. Avlonitis and G. Papadopoulos, “Foreshocks and B Value: Bridging Macroscopic Observations to Source Mechanical Considerations,” *Pure and Applied Geophysics*, vol. 171, no. 10, pp. 2569–2580, 2014 (cit. on p. 36).
2. S.-J. W. Bin-Quan Li, “Self-Organized Criticality in an Anisotropic Earthquake Model,” *Communications in Theoretical Physics*, vol. 69, no. 03, 280, p. 280, 2018. [Online]. Available: https://ctp.itp.ac.cn/EN/abstract/article_17223.shtml (cit. on p. 36).
3. R. Burridge and L. Knopoff, “Model and Theoretical Seismicity,” *Bulletin of the Seismological Society of America*, vol. 57, no. 3, pp. 341–371, 1967 (cit. on p. 34).
4. M. D. Donsker, An Invariance Principle for Certain Probability Limit Theorems. 1951 (cit. on p. 23).
5. B. Gutenberg and C. F. Richter, “Earthquake Magnitude, Intensity, Energy, and Acceleration: (Second Paper),” *Bulletin of the Seismological Society of America*, vol. 46, no. 2, pp. 105–145, 1956 (cit. on p. 34).
6. S. Hergarten, “Landslides, Sandpiles, and Self-Organized Criticality,” 2003 (cit. on p. 35).
7. S. Lalley, “Mathematical Finance 345 Lecture 5: Brownian Motion,” 2001 (cit. on p. 23).
8. T. E. of Encyclopaedia Britannica, “Brownian Motion,” *Encyclopædia Britannica*, 2017 (cit. on p. 21).
9. Z. Olami, H. J. S. Feder, and K. Christensen, “Self-Organized Criticality in a Continuous, Nonconservative Cellular Automaton Modeling Earthquakes,” *Physical Review Letters*, vol. 68, no. 8, p. 1244, 1992 (cit. on p. 34).
10. C. Pacatis, “Brownian Motion and Geometric Brownian Motion: Graphical Representations,” 2010 (cit. on p. 26).
11. S. M. Ross, *Introduction to Probability Models*, Ch. 10.3. Academic press, 2014 (cit. on p. 26).

12. D. O. Siegmund, “Probability Theory,” *Encyclopædia Britannica*, 2017 (cit. on pp. 21, 23).
13. K. Taipale, *Mathematical Preparation for Finance*, Ch. 7. 2017, pp. 125–128 (cit. on p. 21).
14. G. E. Uhlenbeck and L. S. Ornstein, “On the Theory of the Brownian Motion,” *Phys. Rev.*, vol. 36, pp. 823–841, 5 1930. [Online]. Available: <https://link.aps.org/doi/10.1103/PhysRev.36.823> (cit. on p. 33).
15. O. Vasicek, “An Equilibrium Characterization of the Term Structure,” *Journal of Financial Economics*, vol. 5, no. 2, pp. 177–188, 1977. [Online]. Available: <https://EconPapers.repec.org/RePEc:eee:jfinec:v:5:y:1977:i:2:p:177-188> (cit. on pp. 15, 33).
16. A. W. Visser, “Using Random Walk Models to Simulate the Vertical Distribution of Particles in a Turbulent Water Column,” *Marine Ecology Progress Series*, vol. 158, pp. 275–281, 1997 (cit. on p. 15).