

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ & ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ



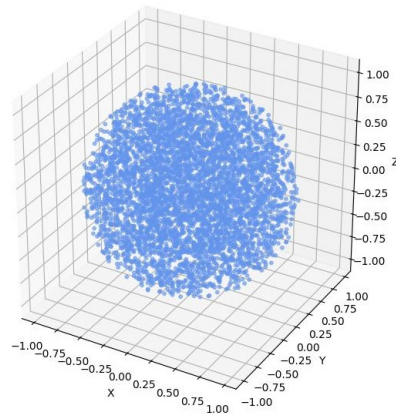
ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
Εθνικόν και Καποδιστριακόν
Πανεπιστήμιον Αθηνών
— ΙΔΡΥΘΕΝ ΤΟ 1837 —



ΥΠΟΛΟΓΙΣΤΙΚΗ ΓΕΩΜΕΤΡΙΑ

SAMPLING IN POLYTOPES

ΧΡΙΣΤΟΦΟΡΟΣ ΖΑΧΑΡΗΣ



ΑΚΑΔΗΜΑΪΚΟ ΈΤΟΣ 2025-2026

Περιεχόμενα

Περίληψη	11
1 Εισαγωγή	13
1.1 Δειγματοληψία (Sampling)	13
2 Βασικές Μέθοδοι Δειγματοληψίας	15
2.1 Accept - Reject Sampling	15
2.2 Εφαρμογή της μεθόδου στη μονοδιάστατη περίπτωση	22
2.3 Εφαρμογή της μεθόδου σε πολύγωνα	27
2.4 Γεωμετρικοί τυχαίοι περίπατοι (MCMC μέθοδοι)	31
2.4.1 Markov Chains	31
2.4.2 Εφαρμογή μιας αλυσίδας Markov.	33
2.4.3 Mixing Time	39
2.4.4 Ball Walk	40
2.4.5 Εφαρμογή της μεθόδου ball walk	44
2.4.6 Hit and Run	61
2.4.7 Εφαρμογή της Hit and Run	62

2.4.8	Dikin Walk	69
2.4.9	Εφαρμογή της Dikin Walk	72
2.4.10	Billiard Walk	80
2.4.11	Εφαρμογή της Billiard Walk	82
2.4.12	Χαμιλτονιανή Μηχανική	87
2.4.13	Σωματίδιο σε κουτί 1 διάστασης	93
2.4.14	Reflective Hamiltonian Monte Carlo method	96
2.4.15	Εφαρμογή της Hamiltonian μεθόδου	99
2.5	Μετρικές σύγκλισης	113
2.5.1	PSRF	113
2.5.2	ESS	115
2.5.3	Εφαρμογή των PSRF και ESS	116
2.5.4	Global Radial Scaling Diagnostic - KS test	120
2.6	Polytope Rounding	125
2.6.1	Εφαρμογή του Polytope Rounding	128
2.7	Null Space Technique	129
2.7.1	The Problem: Zero Volume	129
2.7.2	The Solution: Parameterization	130
2.7.3	Transforming the Constraints	130
2.7.4	Numerical Conditioning	131

3.1	Volesti Library	133
3.2	Installing Volesti	134
3.2.1	Failed attempts	134
3.2.2	Successful attempt	135
3.3	Running the tutorial example	137
3.4	Running a sampling example with billiard walk	140
3.5	Sampling example analysis	142
3.6	Billiard walk in Volesti	150
3.7	Vanilla billiard walk vs Volesti	158
3.8	Billiard Walk Variations	160
3.9	Creating our own example for sampling	161
3.9.1	Setting up the necessary files	161
3.9.2	Starting to work on our example using Ball Walk	162
3.9.3	Applying the convergence metrics	169
3.9.4	Generating samples dynamically based on ESS	170
3.9.5	Applying for all random walk methods and adding time	173
3.9.6	Expanding for multiple dimensions	177
3.9.7	Optimizing the code	178
3.10	Running our example	182
3.10.1	Time trial for d-hypercube and constant ess threshold	183
3.10.2	Time trial for 500-hypercube and varied ess threshold	184

3.10.3	Walk length vs total points for constant ess threshold	186
3.10.4	Time trial for barrier functions walks for hypercube	188
3.10.5	Time trial for Simplex and 3k ess threshold	189
3.10.6	Time trial for Birkhoff and 3k ess threshold	190
4	PolytopeWalk	193
4.1	PolytopeWalk Library	193
4.2	Installing PolytopeWalk	194
4.3	Running polytope walk tutorial example	195
4.4	Creating our own polytope walk example	199
4.4.1	Setting up the diagnostic functions	199
4.4.2	Setting up the polytope generators	201
4.4.3	Basic set-up of our example	201
4.4.4	Run for multiple dimensions and up to 3k ESS	203
4.5	Running our example	207
4.5.1	Walks vs r parameter	207
4.5.2	Choosing r parameter	209
4.5.3	Time trial for hypercube	209
5	PolytopeSampler	211
5.1	Installing PolytopeSampler	211
5.2	Running demo tutorial	212
5.2.1	Understanding the result - the K-S statistic	214

5.2.2	Methodology and Implementation	215
5.3	Creating our own demo	217
5.4	Running our demo	220
5.4.1	Time trial for d-hypercube using 3k ess threshold	220
5.4.2	Time Trial for d-Simplex and 3k ESS threshold	222
5.4.3	Cube vs Simplex	222
6	Dingo	225
7	PvP	227
7.1	Barrier Walks	227
7.2	Ball Walk, Hit and Run	228
7.3	Free for All	228
7.3.1	Cube for 3k ESS	228
7.3.2	Simplex for 3k ESS	229
7.3.3	Cube for 800 ESS	230
7.3.4	Simplex for 800 ESS	231
7.3.5	KS test for 500 Hypercube	232
7.3.6	Birkhoff for 800 ESS	234
7.3.7	E.Coli polytope for 200 ESS	235
7.3.8	Mixing Time and Time efficiency for Simplex	236
7.3.9	Mixing Time and Time efficiency Birkhoff	237
7.3.10	Mixing Time and Time efficiency for Biological Polytopes	238

7.3.11	Mixing Time and Time efficiency for Rounded Birkhoff	239
7.3.12	Comparing rounded vs original Birkhoff	239
7.3.13	Empirical CDF για την CDHR	240
7.3.14	Dingo vs Volesti Sampling time	241
7.3.15	NetLib Polytopes	241
7.3.16	Sparse Billiard vs Normal Billiards	244
7.3.17	Breaking the KS test	246
7.3.18	Vanilla Birkhoff benchmark	251
7.4	Biological Polytopes Preprocessing	253

Περίληψη

SOME INTRO

Κεφάλαιο 1

Εισαγωγή

1.1 Δειγματοληψία (Sampling)

Η δειγματοληψία (sampling) αποτελεί μια βασική και θεμελιώδη τεχνική της υπολογιστικής γεωμετρίας και των αλγορίθμων της. Σε γενικές γραμμές αφορά την επιλογή ενός αντιπροσωπευτικού υποσυνόλου δεδομένων από ένα πολύ μεγαλύτερο σύνολο δεδομένων. Αυτό επιτρέπει στους αλγόριθμους να προσεγγίζουν αποτελεσματικά τις λύσεις χωρίς να επεξεργάζονται το πλήρες σύνολο εισόδου, κάτι που είναι ιδιαίτερα επωφελές όταν πρόκειται για μεγάλα ή υψηλής διάστασης δεδομένα. Στο πλαίσιο της υπολογιστικής γεωμετρίας, η δειγματοληψία δεν είναι μόνο ένα εργαλείο για τη μείωση του υπολογιστικού κόστους, αλλά και μια στρατηγική για τη διατήρηση της ακρίβειας και της αξιοπιστίας των προσεγγίσεων. Διαδραματίζει ζωτικό ρόλο στη βελτιστοποίηση της απόδοσης των αλγορίθμων, στη διατήρηση της μνήμης και στη διασφάλιση της επεκτασιμότητας σε πρακτικές εφαρμογές.

Για πολλά γεωμετρικά προβλήματα — όπως αναζητήσεις πλησιέστερου γείτονα, ερωτήματα εύρους, προσεγγίσεις σχήματος και εύρεση όγκου — η δειγματοληψία βοηθά στην επίτευξη ισορροπίας μεταξύ ακρίβειας και ταχύτητας. Αυτή η συμβιβαστική λύση είναι συχνά απαραίτητη σε πραγματικές εφαρμογές όπου ο ακριβής υπολογισμός μπορεί να είναι πολύ δαπανηρός ή ανέφικτος.

Γενικά, η υπολογιστική γεωμετρία που χρησιμοποιεί τις τεχνικές της δειγματοληψίας, αποτελεί ένα εξειδικευμένο κλάδο της επιστήμης των υπολογιστών που επικεντρώνεται στο σχεδιασμό, την ανάλυση και την εφαρμογή αλγορίθμων για την επίλυση γεωμετρικών προβλημάτων που αφορούν σημεία, γραμμές, πολύγωνα και αντικείμενα υψηλότερης διάστασης. Αποτελεί τη θεωρητική βάση πολλών συστη-

μάτων χωρικής υπολογιστικής και έχει ένα ευρύ φάσμα εφαρμογών στον πραγματικό κόσμο σε διάφορους κλάδους.

Η χρήση της δειγματοληψίας στους γεωμετρικούς υπολογισμούς έχει εξελιχθεί σημαντικά τις τελευταίες δεκαετίες. Οι πρώτοι αλγόριθμοι βασίζονταν συχνά στην τυχαία δειγματοληψία, όπου τα σημεία δεδομένων επιλέγονταν ομοιόμορφα και τυχαία από ένα σύνολο δεδομένων για την εκτίμηση ιδιοτήτων ή την κατασκευή γεωμετρικών δομών, όπως κυρτά περιβλήματα και διαγράμματα Voronoi. Αν και απλή και αποτελεσματική σε πολλές περιπτώσεις, η τυχαία δειγματοληψία μερικές φορές παρουσίαζε προβλήματα άνισης κάλυψης ή παράλειψης κρίσιμων χαρακτηριστικών των δεδομένων.

Για να αντιμετωπιστούν αυτοί οι περιορισμοί εισήχθησαν πιο προηγμένες τεχνικές, όπως η στρωματοποιημένη δειγματοληψία, η δειγματοληψία σημασίας και η δειγματοληψία χαμηλής απόκλισης (ημι-τυχαία). Αυτές οι μέθοδοι στοχεύουν στην παροχή καλύτερης κάλυψης, στη μείωση της διακύμανσης και στη βελτίωση της ακρίβειας των εκτιμήσεων — ιδιαίτερα σε χώρους υψηλής διάστασης ή όταν συγκεκριμένες περιοχές των δεδομένων είναι πιο σημαντικές από άλλες.

Τα τελευταία χρόνια, οι μέθοδοι δειγματοληψίας έχουν επίσης προσαρμοστεί ώστε να ταιριάζουν σε συγκεκριμένα αλγοριθμικά παραδείγματα, όπως coresets, ε-nets και υπογραμμικοί αλγόριθμοι, διευρύνοντας τα όρια του τι μπορεί να υπολογιστεί αποτελεσματικά στη γεωμετρία. Η ανάπτυξη αυτών των τεχνικών έχει συμβάλει καθοριστικά στο να καταστεί η σύγχρονη υπολογιστική γεωμετρία πρακτική και επεκτάσιμη.

Κεφάλαιο 2

Βασικές Μέθοδοι Δειγματοληψίας

2.1 Accept - Reject Sampling

Η μέθοδος accept-reject ή αλλιώς rejection sampling αποτελεί μια από τις πιο βασικές μεθόδους δειγματοληψίας σε κάποια κατανομή. Έστω πως θέλουμε να πάρουμε σημεία εντός κάποιας τυχαίας και δύσκολης κατανομής. Η βασική λογική της μεθόδου έγκειται στην δειγματοληψία πάνω σε μια πιο εύκολη κατανομή που προσεγγίζει την αρχική και στην συνέχεια την αποδοχή ή απόρριψη των δειγμάτων αυτών βάση μιας πιθανότητας.

Θα δούμε την γενική λογική της μεθόδου σε μια διάσταση ώστε να κατανοήσουμε καλύτερα την λογική της. Θεωρούμε μια συνάρτηση πυκνότητας πιθανότητας κάποιας συνεχούς τυχαίας μεταβλητής. Η τυχαία αυτή μεταβλητή δεν έχει προς το παρόν σημασία τι εκφράζει, μπορεί να αφορά για παράδειγμα τις πιθανές τιμές των βαθμών των μαθητών ενός τμήματος. Ας ονομάσουμε τους βαθμούς των μαθητών s και την συνάρτηση πυκνότητας πιθανότητας ως $p(s)$. Η συνάρτηση πυκνότητας πιθανότητας δίνεται ως ο λόγος μιας μη αρνητικής συνάρτησης που δίνει πόσο πιθανός είναι κάθε βαθμός διαιρεμένη με ένα παράγοντα κανονικοποίησης:

$$p(s) = \frac{f(s)}{NC} \quad (2.1)$$

όπου ο παράγοντας κανονικοποίησης NC δίνεται ως:

$$NC = \int f(s) ds \quad (2.2)$$

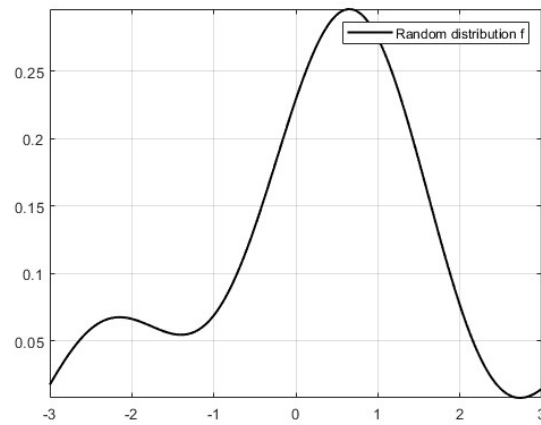
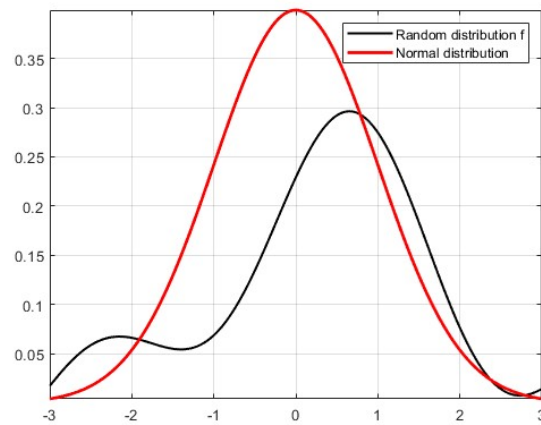
με το ολοκλήρωμα να κινείται στο σύνολο του πεδίου ορισμού της συνάρτησης. Στην ουσία ο παράγοντας κανονικοποίησης είναι απαραίτητος ώστε να μετατρέψουμε την συνάρτηση σε μια πυκνότητα πιθανότητας, το ολοκλήρωμα της οποίας στο πεδίο ορισμού της να είναι ίσο με την μονάδα και έτσι να εκφράζει πιθανότητα. Αυτό σημαίνει πως η πιθανότητα να βρίσκεται για παράδειγμα κάποιος βαθμός από το $-\infty$ στο $+\infty$ είναι ίσος με την μονάδα και άρα σίγουρα ένας τυχαίος βαθμός βρίσκεται σε αυτό το διάστημα.

Σε πολλές περιπτώσεις γνωρίζουμε την συνάρτησης πιθανότητας $f(s)$ στον αριθμητή αλλά είναι εξαιρετικά δύσκολη η εύρεση του παράγοντα κανονικοποίησης στον παρονομαστή. Ο υπολογισμός ενός ολοκληρώματος από το $-\infty$ στο $+\infty$ δεν είναι πάντα εύκολη υπόθεση και υπολογίζεται αναλυτικά μόνο σε λίγες συγκεκριμένες περιπτώσεις. Υποθέτουμε πως το ίδιο σενάριο ισχύει και στην περίπτωση μας.

Όταν η συνάρτηση πυκνότητας πιθανότητας της κατανομής είναι γνωστή, τότε η δειγματοληψία γίνεται εύκολη. Σκεφτείται για παράδειγμα την γκαουσιανή ή κανονική κατανομή. Στην περίπτωση μας όμως, η συνάρτηση αυτή είναι άγνωστη. Πως λοιπόν θα γίνει η δειγματοληψία; Η ιδέα είναι η δειγματοληψία να πραγματοποιηθεί σε μια γνωστή κατανομή που προσεγγίζει την κατανομή υπό διερεύνηση. Έστω η κατανομή αυτή ονομάζεται $g(s)$. Βασικές προϋποθέσεις της κατανομής αυτής είναι δυο: η δειγματοληψία πάνω της να είναι εύκολη και επιπλέον το πεδίο ορισμού της να εκτείνεται στον ίδιο χώρο με την αρχική κατανομή. Η ομοιότητα με την αρχική κατανομή δεν αποτελεί απαραίτητη προϋπόθεση, αλλά η εγκύτητα αυτή θα βοηθήσει υπολογιστικά το μοντέλο.

Καλύτερη κατανόηση θα επιτευχθεί με ένα παράδειγμα. Θεωρούμε μια τυχαία κατανομή για την οποία γνωρίζουμε την συνάρτηση πιθανότητας, αλλά ο υπολογισμός του παράγοντα κανονικοποίησης για την εύρεση της πυκνότητας πιθανότητας περιλαμβάνει ένα πολύπλοκο ολοκλήρωμα από το $-\infty$ στο $+\infty$. Δείτε τα παρακάτω σχήματα όπου αρχικά έχουμε την τυχαία κατανομή, η οποία μάλλον θυμίζει την κανονική κατανομή την οποία και σχεδιάζουμε στην συνέχεια.

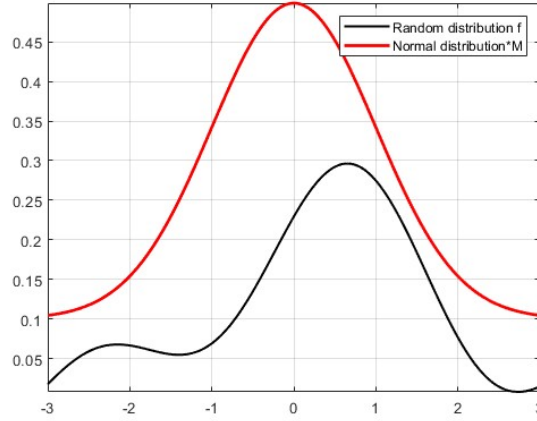
Το επόμενο βήμα είναι να προσαρμόσουμε την κανονική κατανομή $g(s)$ έτσι ώστε να βρίσκεται πάντα υψηλότερα στο γράφημα από ότι η τυχαία κατανομή. Προφανώς αν η δειγματοληψία στην τωρινή κατάσταση, υπάρχουν σημεία τα οποία θα βρίσκονται μεν εντός της κανονικής κατανομής, αλλά εκτός της κατανομής f . Για τον σκοπό αυτό θα πολλαπλασιάσουμε την $g(s)$ με κάποιον μεγάλο ενδεχομένως

Σχήμα 2.1: Μια τυχαία κατανομή f .Σχήμα 2.2: Μια τυχαία κατανομή f και η κανονική κατανομή.

αριθμό M ώστε να την φέρουμε ψηλότερα από την κατανομή f . Μπορείτε να δείτε την προσαρμογή αυτή στο παρακάτω σχήμα.

Μπορεί τώρα να ξεκινήσει η δειγματοληψία. Αρχικά, δειγματοληπτούμε από την εύκολη κατανομή $g(s)$. Το σημείο αυτό μπορεί να πέσει οπουδήποτε με μεγαλύτερη πιθανότητα να βρίσκεται σε περιοχές μεγαλύτερης πυκνότητας πιθανότητας. Στην συνέχεια, αποδεχόμαστε ή απορρίπτουμε το δείγμα αυτό βάση μιας πιθανότητας που δίνεται από την σχέση:

$$Prob \left[\frac{f(s)}{M \cdot g(s)} \right] \quad (2.3)$$



Σχήμα 2.3: Μια τυχαία κατανομή f και η σχεδόν κανονική κατανομή $Mg(s)$.

όπου s το δείγμα που πήραμε από την $g(s)$. Αποτελεί ο παραπάνω λόγος πιθανότητα; Η απάντηση είναι πως ναι, αν θυμηθείτε πως το M επιλέχθηκε ακριβώς έτσι ώστε η κατανομή $M \cdot g(s)$ να βρίσκεται συνεχώς πάνω από την κατανομή $f(s)$. Κατά αυτόν τον τρόπο το κλάσμα αυτό παίρνει τιμές από 0 έως 1 και άρα πράγματι εκφράζει πιθανότητα.

Μετά από εκτεταμένη δειγματοληψία θα καταλήφουμε με δυο σύνολα δειγμάτων, αυτά που έχουμε αποδεχθεί βάσει της παραπάνω πιθανότητας και αυτά που έχουμε απορρίψει. Το μαγικό βήμα της μεθόδου είναι πως το σύνολο των δειγμάτων που έχουμε αποδεχθεί, βρίσκονται εξ'ολοκλήρου εντός της κατανομής f και έτσι είναι σαν να έχουμε δειγματολήψει από την κατανομή f . Μια μαθηματική απόδειξη αποδεικνύει τον παραπάνω ισχυρισμό. Ας δούμε λοιπόν με τι ισούται η πυκνότητα πιθανότητας $D(s|A)$ ενός δείγματος, δεδομένου ότι το έχουμε αποδεχθεί. Από τον νόμο του Bayes:

$$D(s|A) = \frac{P(A|s) D(s)}{P(A)} \quad (2.4)$$

με $P(A|s)$ η πιθανότητα να έχουμε αποδεχθεί το δείγμα δεδομένης της τιμής του, $D(s)$ η πυκνότητα της τιμής s και $P(A)$ η πιθανότητα αποδοχής ενός δείγματος. Όμως, η πιθανότητα να έχουμε αποδεχθεί το δείγμα δεδομένης της τιμής του είναι ακριβώς η σχέση (2.3) μέσω της οποίας κατασκευάσαμε το σύνολο A . Παράλληλα, η πυκνότητα του s είναι η $g(s)$ από την οποία έχει προέλθει. Έτσι:

$$D(s|A) = \frac{\frac{f(s)}{Mg(s)} \cdot g(s)}{P(A)} \quad (2.5)$$

Η πιθανότητα τώρα του να αποδεχθούμε ένα δείγμα μπορεί να υπολογιστεί εάν σαρώσουμε το σύνολο των τιμών και της πιθανότητας να αποδεχθούμε κάθε μια από αυτές:

$$P(A) = \int_{-\infty}^{+\infty} g(s) \cdot \frac{f(s)}{Mg(s)} ds = \frac{1}{M} \int_{-\infty}^{+\infty} f(s) ds = \frac{NC}{M} \quad (2.6)$$

Το τελευταίο βήμα προέκυψε από την σχέση (2.2). Αν συνδυάσουμε τώρα τα παραπάνω καταλήγουμε στο εξής συμπέρασμα:

$$D(s|A) = \frac{f(s)/M}{NC/M} = \frac{f(s)}{NC} = p(s) \quad (2.7)$$

που σύμφωνα με την (2.1) είναι ακριβώς η άγνωστη πυκνότητα πιθανότητας της ζητούμενης κατανομής. Αυτό σημαίνει πως τα δείγματα που έχουμε αποδεχθεί, ακολουθούν την πυκνότητα πιθανότητας που ψάχνουμε.

Αξίζει να κατανοήσουμε κάπως πιο διαισθητικά το γιατί δουλεύει αυτή η μέθοδος. Δείτε τον λόγο $f(s)/g(s)$ που εμφανίζεται στην πιθανότητα αποδοχής ενός δείγματος. Όσο η πιθανότητα $f(s)$ ενός δείγματος μεγαλώνει, τόσο πιο πιθανό είναι το δείγμα αυτό να βρίσκεται στην άγνωστη κατανομή $p(s)$ καθώς τα δυο μεγέθη είναι ανάλογα. Συνεπώς, το ίδιο το δείγμα έχει μεγάλη πιθανότητα να βρίσκεται εντός της ζητούμενης κατανομής και άρα έχει νόημα να το αποδεχθούμε. Ομοίως, αν ο παρονομαστής μειωθεί, αυτό σημαίνει πως το δείγμα είναι σπάνιο στην κατανομή $g(s)$, δεν εμφανίζεται συχνά, υπάρχει όμως καλή πιθανότητα να βρίσκεται στην ζητούμενη κατανομή. Τότε καλό θα ήταν να γίνει αποδεκτό, αφού πιθανώς δεν θα το ξαναδούμε. Συνεπώς, όσο μεγαλύτερος ο παραπάνω λόγος, τόσο πιο σημαντικό και με καλές προοπτικές είναι το υπό εξέταση δείγμα και άρα αποδεχόμαστε. Ο παράγοντας M υπάρχει εκεί ώστε να κανονικοποιήσει τον λόγο αυτό μεταξύ 0 και 1 και έτσι να το δούμε ως πιθανότητα. Συνπερασματικά λοιπόν, έχουμε καταφέρει να πάρουμε δείγματα τα οποία ακολουθούν μια συγκεκριμένη περίπλοκη πυκνότητα πιθανότητας, δειγματοληπτώντας πάνω σε μια εύκολη και γνωστή κατανομή.

Μια από τις βασικές δυσκολίες της μεθόδου αποδοχής-απόρριψης είναι η επιλογή της κατάλληλης κατανομής δειγματοληψίας. Ας για παράδειγμα ο παράγοντας

Μ γίνεται πολύ μεγάλος ώστε η κατανομή που έχουμε επιλέξει να καλύψει την άγνωστη κατανομή, τότε η πιθανότητα με την οποία θα αποδεχόμαστε δείγματα (σχέση 2.6) θα γίνει ελάχιστη με συνέπεια το υπολογιστικό κόστος της μεθόδου να αυξηθεί.

Αν και εδώ παρουσιάσαμε την μέθοδο σε μία διάσταση, η λογική της εφαρμόζεται πανομοιότυπα και σε δύο ή περισσότερες διαστάσεις, για παράδειγμα σε περιοχές εντός πολυγώνων τα οποία μας ενδιαφέρουν ιδιαίτερα. Στην περίπτωση αυτή, η κατανομή στόχος είναι η ομοιόμορφη κατανομή εντός ενός πολυγώνου, το οποίο ορίζεται από ένα σύνολο ανισοτήτων της μορφής:

$$P = \{(x, y) \in R^2 | a_i x + b_i y \leq c_i, i = 1, 2, \dots, k\}$$

Η συνάρτηση πυκνότητας πιθανότητας της ομοιόμορφης κατανομής μέσα στο πολύγωνο P δίνεται από την σχέση:

$$p(x, y) = \begin{cases} \frac{1}{A_P}, & \text{αν } (x, y) \in P \\ 0, & \text{διαφορετικά} \end{cases}$$

όπου A_P είναι το εμβαδόν του πολυγώνου και η ποσότητα $1/A_P$ αφορά την πυκνότητα πιθανότητας σε κάθε σημείο εντός του πολυγώνου και η ολοκλήρωση πάνω σε όλο το πολύγωνο δίνει αποτέλεσμα 1.

Αξίζει να σημειωθεί μια σημαντική διαφορά μεταξύ της γενικής περίπτωσης μιας τυχαίας κατανομής μίας διάστασης και της περίπτωσης ομοιόμορφης δειγματοληψίας εντός ενός πολυγώνου. Στην γενική περίπτωση, η πυκνότητα πιθανότητας $p(s)$ δεν είναι γνωστή πλήρως, καθώς η κανονικοποιημένη σταθερά NC είναι δύσκολο να υπολογιστεί. Αντίθετα, στην περίπτωση ομοιόμορφης κατανομής μέσα σε ένα πολύγωνο P , η πυκνότητα πιθανότητας είναι πλήρως γνωστή και δίνεται από την παραπάνω σχέση. Επομένως, η μέθοδος αποδοχής–απόρριψης στην διδιάστατη περίπτωση ενός πολυγώνου, λειτουργεί απλώς ως γεωμετρικό φίλτρο. Δειγματοληπτούμε από μια γνωστή και εύκολη κατανομή (π.χ. ορθογώνιο που περικλείει το πολύγωνο) και αποδεχόμαστε μόνο τα σημεία που ικανοποιούν τις ανισότητες του πολυγώνου, εξασφαλίζοντας έτσι την ομοιόμορφη κατανομή εντός του P .

Έτσι λοιπόν, επιλέγουμε μια εύκολη κατανομή $g(x, y)$ από την οποία μπορούμε να δειγματοληπτήσουμε άμεσα, και η οποία καλύπτει πλήρως το πολύγωνο P . Συνήθως ως g χρησιμοποιείται η ομοιόμορφη κατανομή μέσα σε ένα ορθογώνιο (bounding box) που περιέχει το P . Δειγματοληπτούμε ένα σημείο (x, y) από

την $g(x, y)$. Τέλος, λέγουμε αν το σημείο αυτό ικανοποιεί όλες τις ανισότητες που ορίζουν το πολύγωνο P . Αν ναι, το αποδεχόμαστε, αλλιώς το απορρίπτουμε.

Στην περίπτωση της ομοιόμορφης δειγματοληψίας σε πολύγωνο, η αποδοχή μπορεί να περιγραφεί με την πιθανότητα:

$$Prob_{accept}(x, y) = \begin{cases} 1, & \text{αν } (x, y) \in P \\ 0, & \text{διαφορετικά} \end{cases}$$

Η μέθοδος είναι ιδιαίτερα απλή γιατί η $f(x, y)$ εδώ είναι σταθερή μέσα στο πολύγωνο και μηδενική έξω από αυτό, οπότε δεν χρειάζεται να υπολογίσουμε συντελεστή M . όπως στην γενική περίπτωση. Η αποδοτικότητα της μεθόδου εξαρτάται από τον λόγο:

$$\eta = \frac{A_P}{A_B} \quad (2.8)$$

όπου A_B είναι το εμβαδόν του bounding box. Όσο μεγαλύτερη είναι η τιμή της η , τόσο λιγότερα σημεία απορρίπτονται και τόσο πιο γρήγορη είναι η δειγματοληψία. Η παραπάνω σχέση υπολογίζεται εύκολα ως η πιθανότητα ένα σημείο της κατανομής g να πέσει εντός του P :

$$\eta = \mathbb{P}_{(x,y) \sim g}[(x, y) \in P] = \int_{\mathbb{R}^2} g(x, y) \mathbf{1}_P(x, y) dx dy = \int_P \frac{1}{A_B} dx dy = \frac{A_P}{A_B}.$$

Το βασικό πρόβλημα της παραπάνω αποδοτικότητας της μεθόδου είναι πως δυστυχώς, όσο οι διαστάσεις αυξάνουν, ο λόγος των εμβαδών τείνει στο μηδέν. Στην πράξη, καθώς αυξάνει το d , ο όγκος του bounding box που δεν ανήκει στην περιοχή-στόχο μεγαλώνει εκθετικά με αποτέλεσμα η η να τείνει πολύ γρήγορα στο μηδέν. Αυτό σημαίνει ότι για να παραχθεί επαρκής αριθμός αποδεκτών δειγμάτων απαιτείται τεράστιος αριθμός δοκιμών, γεγονός που καθιστά τη μέθοδο εξαιρετικά αναποτελεσματική σε υψηλές διαστάσεις.

Αν και στην προηγούμενη παράγραφο υποθέσαμε ότι η περιοχή δειγματοληψίας είναι ένα πολύγωνο ορισμένο από γραμμικές ανισότητες, η μέθοδος μπορεί να επεκταθεί αυτούσια και σε πιο πολύπλοχα σχήματα, ακόμη και όταν τα όρια είναι καμπύλες ή παρουσιάζουν τυχαία μορφή. Το μόνο που απαιτείται για την εφαρμογή της μεθόδου αποδοχής-απόρριψης είναι η ύπαρξη ενός μηχανισμού που, δοθέντος

ενός σημείου (x, y) , να μπορεί να ελέγξει αν αυτό βρίσκεται εντός ή εκτός της περιοχής.

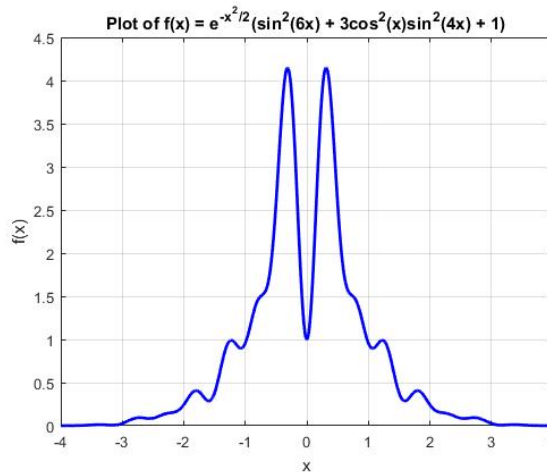
2.2 Εφαρμογή της μεθόδου στη μονοδιάστατη περίπτωση

Η εφαρμογή των θεωρητικών εννοιών βοηθά πάντα στην καλύτερη κατανόησή τους. Για τον σκοπό αυτό ξεκινούμε από την μονοδιάστατη περίπτωση, κατά την οποία θα πάρουμε μια τυχαία κατανομή στόχη από την οποία θα προσπαθήσουμε να εκτελέσουμε δειγματοληψία με την μέθοδο accept-reject sampling ακολουθώντας την μορφή της.

Ξεκινούμε με μια τυχαία συνάρτηση πιθανότητας δικής μας επιλογής:

$$f(x) = e^{-x^2/2} \cdot (\sin^2(6x) + 3 \cdot \cos^2(x) \cdot \sin^2(4x) + 1) \quad (2.9)$$

Στο σχήμα μπορείτε να δείτε την γραφική παράσταση της συνάρτησης.



Σχήμα 2.4: $f(x) = e^{-x^2/2} \cdot (\sin^2(6x) + 3 \cdot \cos^2(x) \cdot \sin^2(4x) + 1)$.

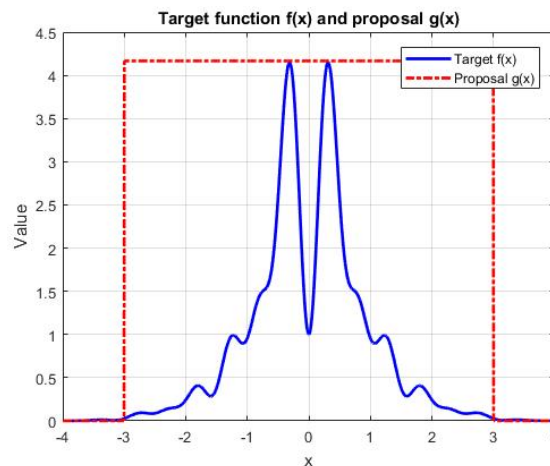
Η εύρεση του παράγοντα κανονικοποίησης NC είναι δύσκολη υπόθεση. Αυτό σημαίνει πως δεν γνωρίζουμε την ακριβή συνάρτηση πυκνότητας πιθανότητας μέσω της οποίας υπάρχει πιθανότητα για άμεση δειγματοληψία, όταν βέβαια η μορφή της είναι απλή. Για τον σκοπό αυτό, η δειγματοληψία θα γίνει με την μέθοδο accept-reject.

Αρχικά πρέπει να γίνει κατανοητό τι ακριβώς θέλουμε να πετύχουμε. Η δειγματοληψία έγκειται στην τυχαία επιλογή τιμών εντός κάποιου διαστήματος, εδώ για παράδειγμα από το -3 έως το 3. Ωστόσο, θέλουμε οι τυχαίες τιμές που θα επιλέξουμε να είναι τελικά κατανεμημένες με συγκεκριμένο τρόπο μέσα στο διάστημα αυτό. Στο παράδειγμά μας, μια τυχαία τιμή έχει μεγαλύτερη πιθανότητα να είναι η 0.5 από ότι η 3. Έτσι, μετά την επιλογή πολλών τυχαίων δειγμάτων, περισσότερα από αυτά πρέπει να βρίσκονται κοντά στο 0.5 από ότι στο 3. Πως λοιπόν, επιλέγοντας τυχαίους αριθμούς από το -3 έως το 3 θα πετύχουμε την κατανομή αυτή; Στην πιο απλή περίπτωση όπου κάθε αριθμός έχει την ίδια πιθανότητα εμφάνισης, η κατανομή προφανώς θα είναι ομοιόμορφη. Έτσι, αυτό δεν είναι αρκετό.

Εφαρμόζοντας την μέθοδο accept-reject, επιλέγουμε μια εύκολη κατανομή από την οποία γνωρίζουμε πως να λάβουμε δείγματα, γνωρίζοντας ακριβώς την συνάρτηση πυκνότητας πιθανότητας. Η πιο απλή περίπτωση είναι η ομοιόμορφη κατανομή. Ωστόσο, όπως αναφέραμε παραπάνω, η ομοιόμορφη κατανομή από μόνη της δεν είναι αρκετή. Για τον λόγο αυτό, σύμφωνα με τα όσα είδαμε στην θεωρία, η αποδοχή ή απόρριψη των τιμών της ομοιόμορφης κατανομής επαφίεται σε μια συγκεκριμένη πιθανότητα η οποία δίνεται ως ο λόγος των δυο κατανομών:

$$\Pr ob \left[\frac{f(s)}{M \cdot g(s)} \right]$$

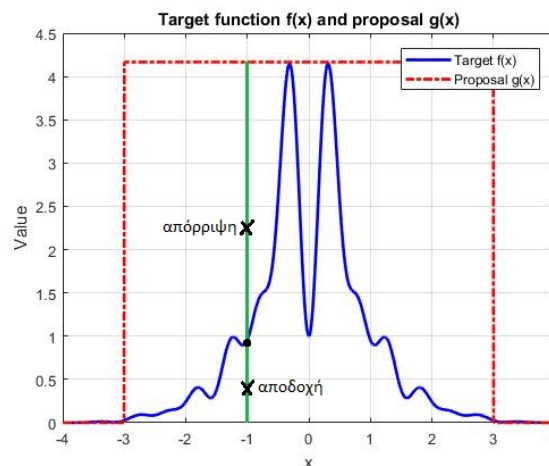
Στο επόμενο γράφημα έχουμε σχεδιάσει τις δυο κατανομές, αφού πρώτα έχουμε κλιμακώσει κατάλληλα την ομοιόμορφη κατανομή πολλαπλασιάζοντας με 25 (δηλαδή $M = 25$).



Σχήμα 2.5: Η τυχαία κατανομή $f(x)$ και η ομοιόμορφη κατανομή την οποία θεωρούμε ως την εύκολη κατανομή δειγματοληψίας.

Είμαστε έτοιμοι για την δειγματοληψία. Επιλέγουμε μια τυχαία τιμή από την ομοιόμορφη κατανομή μεταξύ -3 και 3 . Ας υποθέσουμε πως μας έτυχε η τιμή -1 . Η τιμή αυτή περιλαμβάνεται στο πεδίο ορισμού και της κατανομής στόχου. Η ερώτηση όμως είναι αν την αποδεχόμαστε ή όχι. Η απόφαση προκύπτει από την εφαρμογή της παραπάνω πιθανότητας με την μέθοδο Monte Carlo. Γενικά, όταν η απόφασή μας για μια δράση επαφίεται στην επιλογή ενός τυχαίου αριθμού, η μέθοδος ονομάζεται Monte Carlo. Χρησιμοποιούμε την ομοιόμορφη κατανομή στο $(0,1)$ και επιλέγουμε έναν τυχαίο αριθμό, έστω u . Αν $u \leq f(x)/Mg(x)$ το δείγμα που επελέγει παραπάνω γίνεται αποδεκτό, αλλιώς απορρίπτεται. θυμηθείτε πως ο λόγος αυτός έχει την ιδιότητα να συμπεριφέρεται ως πιθανότητα και άρα να παίρνει τιμές από 0 ως 1 . Για αυτό και μπορεί να χρησιμοποιηθεί κατά αυτόν τον τρόπο.

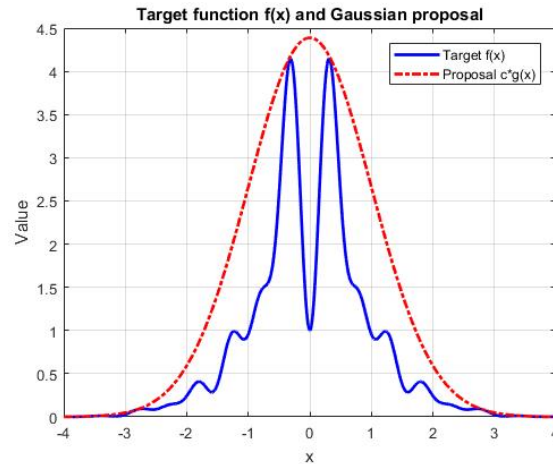
Γίνεται κατανοητό πως σε κάθε βήμα επιλέγονται δυο τυχαίοι αριθμοί. Ο ένας είναι το δείγμα από την προτεινόμενη κατανομή και ο δεύτερος ένας αριθμός από μια ομοιόμορφη κατανομή στο $(0,1)$ που μας βοηθά στην απόφαση. Το επόμενο σχήμα δίνει μια οπτική εικόνα για την μέθοδο Monte Carlo που χρησιμοποιήσαμε. Βλέπουμε τις δυο κατανομές και το δείγμα που επελέγει, μαζί με την κάθετη στο δείγμα.



Σχήμα 2.6: Η πράσινη κατακόρυφος έχει σχεδιαστεί στο δείγμα -1 . Μπορούμε να σκεφτούμε την μέθοδο Monte Carlo σαν να επιλέγουμε έναν αριθμό ομοιόμορφα πάνω σε αυτήν την κατακόρυφο. Αποδεχόμαστε όταν ο τυχαίος αριθμός βρίσκεται πιο χαμηλά από την καμπύλη πιθανότητας της κατανομής, ενώ απορρίπτουμε αλλιώς.

Ένα πρόβλημα με την επιλογή της ομοιόμορφης κατανομής ως προτεινόμενης κατανομής δειγματοληψίας είναι πως η περιοχή απόρριψης είναι αρκετά μεγαλύτερη από την περιοχή αποδοχής όπως δείχνει και το σχήμα. Κατά αυτόν τον τρόπο απαιτούνται επιπλέον δείγματα και αυξάνει η υπολογιστική πολυπλοκότητα. Μια

καλύτερη περίπτωση αποτελεί η γκαουσιανή κατανομή. Δείτε το επόμενο σχήμα στο οποίο διαπιστώνεται η καλύτερη προσαρμογή της κανονικής κατανομής στην κατανομή στόχο.



Σχήμα 2.7: Η μορφή της κατανομής στόχου μας θυμίζει μια κανονική κατανομή. Για τον σκοπό αυτό και για την βελτιστοποίηση της πολυπλοκότητας του προβλήματος, η χρήση της κανονικής κατανομής ως κατανομής πρότασης αποτελεί μια πιο σίγουρη επιλογή, η οποία μειώνει στο ελάχιστο την περιοχή απόρριψης. Ο παράγοντας κανονικοποίησης της κανονικής κατανομής με $\mu=0$ και $\sigma=1$ ώστε να υπερκαλύψει την συνάρτηση $f(x)$ είναι $M = 11$

Ας γράψουμε ένα μικρό κομμάτι κώδικα που εφαρμόζει την μέθοδο accept-reject sampling για έκαστη εκ των προτεινόμενων κατανομών:

```

1 import numpy as np
2 import scipy.stats as stats
3 import matplotlib.pyplot as plt
4 import time
5
6 # Our Target function
7 def target_fn(x):
8     return np.exp(-(x**2)/2) * (
9         np.sin(6*x)**2 + 3*np.cos(x)**2*np.sin(4*x)**2 + 1
10    )
11
12 # Both proposals, uniform and gaussian will be used to compare
13 # the timings.
14 # use_gauss_proposal boolean will determine what proposal is used.
15 # M is the scaling constant and N the number of samples
16 def sampler(use_gauss_proposal, M, N=1_000_000):
17     # Proposal distributions
18     prop_norm = stats.norm(loc=0, scale=1)           # Gaussian
19     proposal

```

```

18 prop_uni = stats.uniform(loc=-3, scale=6)    # Uniform on
      [-3,3]
19
20 if use_gauss_proposal:
21
22     # Draw N samples from the standard normal distribution
      N(0,1).
23     # This is the Gaussian proposal distribution g(x).
24     X = np.random.randn(N)    # samples from g(x)
25
26     # Evaluate the proposal density g(x) at each sampled
      point X.
27     # For the Gaussian case, this gives g(x) = (1/sqrt(2)) *
      exp(-x^2 / 2).
28     # These values are needed in the acceptance test u <=
      f(x)/(M*g(x)).
29     # g_vals is thus an array containing these values
30     g_vals = prop_norm.pdf(X)
31 else:
32     # Draw from Uniform
33     X = np.random.uniform(-3, 3, size=N)
34     g_vals = prop_uni.pdf(X)
35
36     # Draw uniform u ~ U(0,1)
37     u = np.random.rand(N)
38
39     # General acceptance condition: u <= f(x)/(M g(x))
40     accepted = X[u <= (target_fn(X) / (M * g_vals))]
41     return accepted
42
43 # Run both proposals and count the time
44 start = time.perf_counter()
45 samples_uniform = sampler(False, M=25)
46 time_uniform = time.perf_counter() - start
47 print(f"Uniform proposal time: {time_uniform:.3f} seconds,
      accepted rate: {len(samples_uniform)/1_000_000:.3f}")
48
49 start = time.perf_counter()
50 samples_gaussian = sampler(True, M=11)
51 time_gaussian = time.perf_counter() - start
52 print(f"Gaussian proposal time: {time_gaussian:.3f} seconds,
      accepted rate: {len(samples_gaussian)/1_000_000:.3f}")

```

Listing 2.1: Run the accept reject sampling using the example $f(x)$ used and both uniform and gaussian proposal.

Η έξοδος του παραπάνω κώδικα είναι η εξής:

Uniform proposal time: 0.125 seconds, accepted rate: 0.235

Gaussian proposal time: 0.156 seconds, accepted rate: 0.536

Εκ πρώτης όψεως η κανονική κατανομή έτρεξε πιο γρήγορα για τα 1000000 δείγματα. Προφανώς η κανονική κατανομή δίνεται από μια πιο απλή σχέση. Ωστόσο, η ταχύτητα εκτέλεσης δεν είναι το κύριο μέγεθος ενδιαφέροντος. Για την κανονική κατανομή, έχουμε αποδεχθεί μόλις το 24% των δειγμάτων, ενώ αντιθέτως με την κανονική κατανομή το αντίστοιχο ποσοστό είναι 54%. Συνεπώς, ο ρυθμός αποδοχής ανά χρονική στιγμή για την κανονική κατανομή υπερσχύει του ρυθμού αποδοχής της ομοιόμορφης κατανομής, μια χρήσιμη ιδιότητα όταν ο χρόνος που διαθέτουμε είναι περιορισμένος. Επιπλέον, φτάνουμε σε κάποιον επιθυμητό αριθμό αποδεκτών δειγμάτων πολύ πιο γρήγορα, κάτι που συνήθως αποτελεί τον αντικειμενικό σκοπό μας. Αν μεταβάλουμε τον κώδικα ώστε να περιμένουμε 1000000 αποδεκτά τώρα δείγματα, δείτε το αποτέλεσμα:

```
Total accepted samples for uniform: 1000000
Uniform proposal time: 0.727 seconds
Total accepted samples for gaussian: 1000000
Gaussian proposal time: 0.382 seconds
```

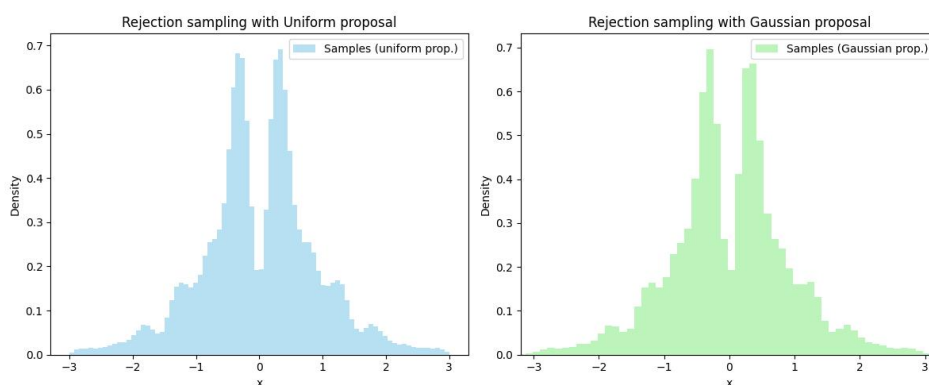
Συνεπώς, η χρήση της γκαουσιανής κατανομής παρουσιάζει εμφανή πλεονεκτήματα στην πολυπλοκότητά της. Γενικά όσο καλύτερη η πρόταση της κατανομής προσεγγίζει το στόχο, τόσο το καλύτερο για την πολυπλοκότητα. Επιδιώκουμε η περιοχή απόρριψης να έχει οτ δυνατόν ελάχιστο μέγεθος.

Ωραία όλα αυτά, αλλά ακολουθούν τα δείγματα τον στόχο μας; Για τον σκοπό αυτό σχεδιάζουμε το ιστόγραμμα των δειγμάτων με το matplotlib και παρουσιάζουμε το αποτέλεσμα στο σχήμα 2.8. Όπως γίνεται κατανοητό, τα αποδεκτά δείγματα ακολουθούν την κατανομή στόχο. Αν και απλή, η μέθοδος accept-reject αποτελεί ένα πανίσχυρο εργαλείο για την δειγματοληψία από δύσκολες κατανομές.

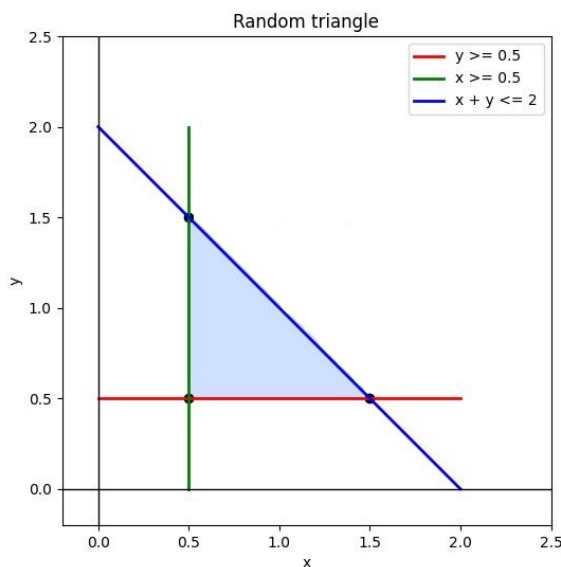
2.3 Εφαρμογή της μεθόδου σε πολύγωνα

Θα χρησιμοποιήσουμε τώρα την μέθοδο accept-reject sampling για την δειγματοληψία σε κάποιο τυχαίο πολύγωνο. Ξεκινώντας απλά, ας θεωρήσουμε το πιο απλό πολύγωνο, ένα τρίγωνο. Δείτε το σχήμα 2.8.

Η στρατηγική είναι η ακόλουθη. Αρχικά, θεωρούμε πως το πολύγωνο μας δίνεται ως ένα σύνολο ανισοτήτων, που αποτελεί μια από τις πιο συνηθισμένες μεθόδους για την αναπαράσταση πολυγώνων. Αφού λοιπόν, γνωρίζουμε τις ανισότητες



Σχήμα 2.8: Το ιστόγραμμα των αποδεκτών δειγμάτων από το σύνολο του 1.000.000. Αριστερά τα δείγματα με πρόταση την ομοιόμορφη κατανομή και δεξιά με πρόταση την γκαουσιανή. Γίνεται φανερό πως τα δείγματα ακολουθούν την κατανομή στόχο που θεωρήσαμε αρχικά. Το παραπάνω σχήμα δείχνει κατά αυτόν τον τρόπο την δύναμη αλλά και την απλότητα της μεθόδου accept-reject sampling.



Σχήμα 2.9: Θεωρούμε το παραπάνω τυχαίο τρίγωνο, το οποίο δίνεται ως το σύνολο των ανισοτήτων που φαίνονται στο σχήμα. Ο στόχος μας είναι η ομοιόμορφη δειγματοληψία εντός του τριγώνου.

του τριγώνου, τότε αρκεί για κάθε δείγμα να ελέγξουμε αν αυτό τις ικανοποιεί. Στην περίπτωση αυτή αποδεχόμαστε αλλιώς απορρίπτουμε. Ως σημείο εκκίνησης των δειγμάτων χρησιμοποιούμε και εδώ μια γνωστή και εύκολη κατανομή όπως στην

μονοδιάστατη περίπτωση. Συνήθως το bounding box του πολυγώνου, δηλαδή ένα τετράγωνο, κύβο κτλ που περιλαμβάνει εξ'ολοκλήρου το υπό εξέταση πολύγωνο. Στη συνέχεια, θεωρούμε ομοιόμορφα δείγματα εντός του bounding box και αποδεχόμαστε αυτά μόνο που ανήκουν στο τρίγωνο με πιθανότητα 1, ενώ απορρίπτουμε αυτά εκτός του τριγώνου.

Ο κώδικας είναι σχετικά απλός:

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 # Triangle check function. Return true if all inequalities are
   satisfied
5 def is_inside_shifted_triangle(x, y):
6     x = np.array(x)
7     y = np.array(y)
8     return (-x <= -0.5) & (-y <= -0.5) & (x + y <= 2)
9
10 N_candidates = 1_000_000
11
12 # Generate candidate points in bounding box using uniform
   distribution
13 # Using arrays is recommended when working with Numpy instead of
   using a for loop.
14 x_candidate = np.random.uniform(0.5, 1.5, N_candidates)
15 y_candidate = np.random.uniform(0.5, 1.5, N_candidates)
16
17 # Check which points are inside the triangle.
18 # Numpy makes things quite simple, only 2 lines needed
19 mask = is_inside_shifted_triangle(x_candidate, y_candidate)
20 samples = np.column_stack((x_candidate[mask], y_candidate[mask]))
21
22 #Efficiency
23 efficiency = len(samples) / N_candidates
24 print(f"Number of accepted samples: {len(samples)}")
25 print(f"Efficiency: {efficiency:.3f}")

```

Listing 2.2: Run the accept reject sampling over the triangle using as proposed distribution the bounding box that includes the triangle.

Η έξοδος του κώδικα είναι η παρακάτω:

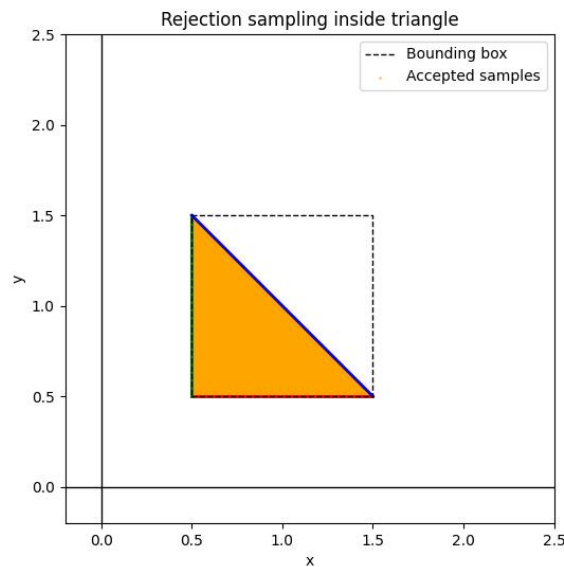
```

Number of accepted samples: 499955
Efficiency: 0.500

```

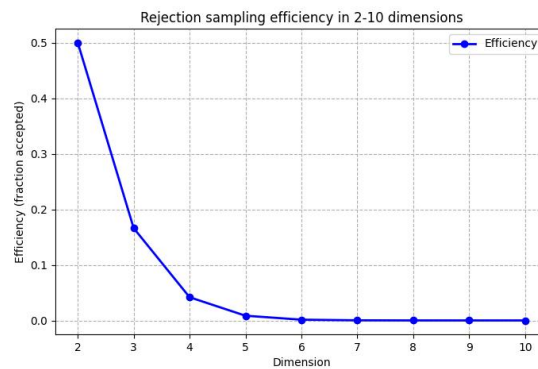
Όπως θα φανεί και στο σχήμα, τα μισά από τα σημεία έχουν γίνει αποδεκτά. Η αποδοτικότητα είναι αυτή που αναμένουμε από την θεωρία, ο λόγος των δυο εμβα-

δών. Το bounding box έχει διπλάσιο εμβαδό από το τρίγωνο και έτσι η θεωρητική τιμή της αποδοτικότητας είναι πράγματι 0.5. Δείτε που βρίσκονται τα αποδεκτά σημεία σε σχέση με το τρίγωνο στο επόμενο σχήμα 2.10 το οποίο έχουμε σχεδιάσει με τα δεδομένα εξόδου του κώδικα.



Σχήμα 2.10: Στο σχήμα βλέπουμε το τρίγωνο και το bounding box που έχουμε θεωρήσει. Με χρώμα φαίνονται τα αποδεκτά σημεία της μεθόδου accept-reject τα οποία βρίσκονται εντός του τριγώνου, ομοιόμορφα καταναμημένα.

Ας δούμε τώρα την αναμενόμενη αποτυχία της μεθόδου με το bounding box σε ανώτερες διαστάσεις. Σχεδιάζουμε το εξής πείραμα. Σε κάθε διάσταση θεωρούμε το πιο απλό σχήμα που υπάρχει, το simplex. Στις τρεις διαστάσεις αυτό είναι ένα τετράεδρο, δηλαδή ένα σχήμα με πλευρές τρίγωνο. Επιπλέον, θεωρούμε το πιο απλό bounding box που περιλαμβάνει το simplex, έναν υπερκύβο. Τρέχουμε την μέθοδο ομοίως με τις δυο διαστάσεις. Δειγματοληπτούμε ομοιόμορφα εντός του υπερκύβου και αποδεχόμαστε μόνο τα δείγματα εντός του simplex. Φτιάχνουμε ένα γράφημα που να δείχνει την αποδοτικότητα συναρτήσει της διάστασης. Δείτε το αποτέλεσμα το σχήμα 2.11.



Σχήμα 2.11: Η αποδοτικότητα της μεθόδου accept-reject σε σχέση με τις διαστάσεις της γεωμετρίας. Ως σχήμα στόχο έχουμε θεωρήσει ένα simplex γύρω από την αρχή των αξόνων και ως κατανομή πρόταση έναν υπερκύβο που περιλαμβάνει το σχήμα στόχο. Δυστυχώς σε πολλές διαστάσεις, ο όγκος του υπερκύβου γίνεται κατά πολύ μεγαλύτερος του simplex με συνέπεια τα περισσότερα δείγματα να βρίσκονται εκτός του simplex.

2.4 Γεωμετρικοί τυχαίοι περίπατοι (MCMC μέθοδοι)

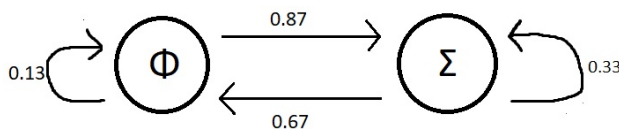
2.4.1 Markov Chains

Οι αλυσίδες Markov ως ιδέα προέρχονται από τις αρχές του 20ου ως απάντηση στο ερώτημα για το αν οι νόμοι των πιθανοτήτων ισχύουν για εξαρτημένα γεγονότα. Θεωρήστε το πείραμα της ρίψης ενός δίκαιου ζαριού. Κάθε ρίψη αποτελεί ένα ανεξάρτητο πείραμα σε σχέση με προηγούμενες ρίψεις. Εκτελέοντας διαδοχικές ρίψεις και καταγράφοντας το αποτέλεσμα, παρατηρούμε πως η αναλογία κορώνας και γραμμάτων συγκλίνει στο 0.5, δηλαδή στις μισές ρίψεις έχουμε κορώνα και στις υπόλοιπες γράμματα. Στο ίδιο αποτέλεσμα καταλήγουμε εφαρμόζοντας τις ιδιότητες των πιθανοτήτων όπου αφού υπάρχουν δυο ισοπίθανα ενδεχόμενα, περιμένουμε αναλογία 50 - 50 για κάθε ένα. Αυτό είναι ο νόμος των μεγάλων αριθμών.

Ισχύει όμως ο νόμος των μεγάλων αριθμών σε περιπτώσεις όπου τα πειράματα δεν είναι ανεξάρτητα μεταξύ τους; Την απάντηση έδωσε ο Markov ο οποίος απέδειξε τον ισχυρισμό αυτό. Στην συνέχεια, ως παράδειγμα μελέτησε τα γράμματα ενός ποιήματος του Pushkin. Αφού κατέγραψε τις 20000 πρώτες λέξεις και αφαίρεσε τα κενά και τα σημεία στίξης, μέτρησε πως 43% των γραμμάτων είναι φωνήεντα ενώ το υπόλοιπο 57% σύμφωνα. Η πιθανότητα όμως κάποιο γράμμα μιας λέξης να είναι σύμφωνο ή φωνήεν εξαρτάται ισχυρά από ποιο γράμμα προηγείται στη λέξη. Έτσι, επικεντρώθηκε στην μελέτη των 4 δυνατών συνδυασμών, φωνήεν-

φωνήεν, φωνήεω-σύμφωνο, σύμφωνο-φωνήεν και σύμφωνο-σύμφωνο. Στην περίπτωση ανεξάρτητων γραμμάτων, η πιθανότητα δυο συνεχόμενων φωνηέντων θα ήταν $0.43 \times 0.43 = 0.18$, ωστόσο μετρώντας την εμφάνιση όλων των ζευγαριών, βρήκε πως μόλις το 6% ήταν ζευγάρια φωνήεν-φωνήεν. Το γεγονός αυτό αποδεικνύει την εξάρτηση του είδους κάθε γράμματος από το προηγούμενό του. Ίδια ήταν τα αποτελέσματα και για τα υπόλοιπα ζεύγη.

Για να ελέγξει τον νόμο των μεγάλων αριθμών, ο Markov έστησε μια μηχανή προβλέψεων ως εξής. Θεωρούμε τις δυο καταστάσεις, φωνήεν και σύμφωνο ως δυο κύκλους από τους οποίους φεύγουν βέλη που οδηγούν στις υπόλοιπες καταστάσεις. Στην περίπτωση της κατάστασης φωνήεν, υπάρχει πιθανότητα $0.06/0.43 = 0.13$ να οδηγηθούμε πάλι σε φωνήεν, ενώ υπάρχει πιθανότητα $1.00 - 0.13 = 0.87$ να οδηγηθούμε σε σύμφωνο, αφού όλες οι περιπτώσεις πρέπει να αθροίζονται στη μονάδα. Δείτε το σχήμα 2.12 που αποτυπώνει τα παραπάνω.



Σχήμα 2.12: Η μηχανή προβλέψεων που κατασκεύασε ο Markov. Ξεκινώντας από κάθε κατάσταση, σύμφωνο ή φωνήεν, υπάρχουν δυο ενδεχόμενα, είτε να παραμείνουμε στην ίδια κατάσταση είτε να περάσουμε στην επόμενη. Κάθε μετάβαση εκφράζει έναν εκ των 4 πιθανών συνδυασμών, φωνήεν-φωνήεν, φωνήεν-σύμφωνο, σύμφωνο-φωνήεν και σύμφωνο-σύμφωνο. Μπορούμε τώρα να εκτελέσουμε το πείραμα πολλές φορές ξεκινώντας από μια τυχαία κατάσταση και καταγράφοντας την αλληλουχία γραμμάτων που προκύπτει. Μετρώντας τελικά το σύνολο κάθε είδους γραμμάτων, βρίσκουμε πως προκύπτουν 43% φωνήεντα και 57% σύμφωνα, ακριβώς όπως τα ποσοστά που ο Markov μέτρησε με το χέρι.

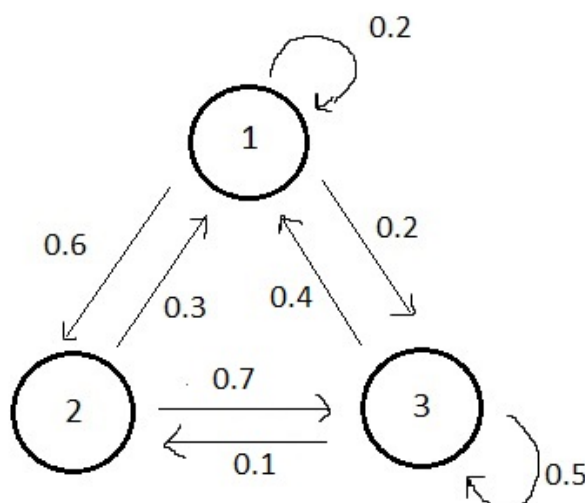
Μπροούμε τώρα με την ίδια λογική που ρίχνουμε το νόμισμα, να τρέξουμε το συγκεκριμένο πείραμα πολλές φορές. Ξεκινώντας από μια τυχαία κατάσταση, εκτελούμε πολλά τυχαία βήματα, καταγράφοντας κάθε φορά το γράμμα που προκύπτει. Εδώ είναι που μπαίνει στην εξίσωση και η μέθοδος Monte Carlo, αφού η απόφασή μας για το σε ποιά κατάσταση οδηγούμαστε σε κάθε βήμα προκύπτει από έναν τυχαίο δείγμα της ομοιόμορφης κατανομής στο διάστημα $[0,1]$. Κάθε βήμα αυτής της διαδικασίας αποτελεί ένα εξαρτόμενο πείραμα σε σχέση με το προηγούμενο βήμα, αφού οι πιθανότητες μετάβασης αλλάζουν ανάλογα με την κατάσταση στην οποία βρισκόμαστε. Μετά το πέρας της διαδικασίας, μετράμε το αριθμό των γραμμάτων που αντιστοιχούν σε κάθε είδος. Προκύπτει πως το 43% των γραμμάτων είναι φωνήεντα, ενώ τα υπόλοιπα σύμφωνα, επιβεβαιώνοντας τα εμπειρικά δεδομένα.

Βασική ιδιότητα των αλυσίδων Markov είναι πως κάθε επόμενο βήμα εξαρ-

τάται από την κατάσταση του τωρινού και μόνο βήματος. Η ιστορία για το πως φτάσαμε ως εδώ δεν έχει καμία σημασία. Ο καιρός αύριο για παράδειγμα εξαρτάται από τον καιρό σήμερα, ενώ στον μικρόκοσμο, η συμπεριφορά ενός σωματιδίου εξαρτάται από την διάταξη των υπολοίπων την τωρινή στιγμή. Αντιθέτως, ένα πείραμα όπου για παράδειγμα επιλέγουμε αριθμούς χωρίς επανάθεση από το 1 ως το 100 δεν αποτελεί αλυσίδα Markov γιατί η ιστορία, δηλαδή το πόσοι αριθμοί έχουμε αφαιρεθεί, επηρεάζει την πιθανότητα εμφάνισης κάθε εναπομείναντα αριθμού.

2.4.2 Εφαρμογή μιας αλυσίδας Markov.

Ας στήσουμε μια δική μας αλυσίδα Markov ώστε να μελετήσουμε πιο αναλυτικά το πως εφαρμόζονται οι παραπάνω ιδέες. Στο σχήμα 2.13, μπορείτε να δείτε τις παραμέτρους του παραδειγματός μας. Πρόκειται για ένα σύστημα με 3 καταστάσεις και συγκεκριμένες πιθανότητες μετάβασης μεταξύ των καταστάσεων, αυτές που φαίνονται στο σχήμα. Πρόκειται για μια από τις πιο απλές αλυσίδες που μπορεί να θεωρήσει κανείς αφού χαρακτηρίζεται από ωραίες ιδιότητες όπως το ότι κάθε μέλος της αλυσίδας είναι προσβάσιμο από κάθε άλλο. Η αλυσίδα δεν μπορεί να υποδιαιρεθεί. Σε κάθε περίπτωση οι ιδιότητες των αλυσίδων Markov βρίσκονται ακτός πλαισίων της εργασίας.



Σχήμα 2.13: Μια τυχαία αλυσίδα Markov με 3 καταστάσεις και συγκεκριμένες πιθανότητες μετάβασης.

Υπολογίζουμε αρχικά την θεωρητική πιθανότητα εμφάνισης κάθε κατάστασης μακροπρόθεσμα σε κάποια στιγμή. Για τον σκοπό αυτό καλό είναι να εισάγουμε τον απαραίτητο συμβολισμό. Μπορούμε να δούμε το σχήμα 2.13 ως ένα κατευθυνόμενο

γράφημα με τα βέλη να δείχνουν τις διαθέσιμες κατευθύνσεις. Ένα τέτοιο γράφημα μπορεί να παρασταθεί σε μορφή πίνακα ως εξής:

$$A = \begin{bmatrix} 0.2 & 0.6 & 0.2 \\ 0.3 & 0.0 & 0.7 \\ 0.4 & 0.1 & 0.5 \end{bmatrix}$$

όπου A_{ij} η πιθανότητα μετάβασης από την κατάσταση i στην κατάσταση j . Γενικά, μια αλυσίδα Markov είναι μια ακολουθία καταστάσεων $X_0, X_1, X_2, \dots$ με πιθανότητες μετάβασης:

$$P_{ij} = \text{Prob}(X_{t+1} = j | X_t = i)$$

δηλαδή η πιθανότητα να βρεθούμε στην κατάσταση j στο βήμα $t + 1$ εξαρτάται μόνο από την κατάσταση i στο βήμα t .

Θεωρήστε τώρα την ποσότητα $P_{ij}(n)$ ως την πιθανότητα να φτάσουμε από την κατάσταση i στην κατάσταση j σε ακριβώς n βήματα. Στην απλή περίπτωση για παράδειγμα όπου $i = 1, j = 2, n = 1$ υπάρχει μόνο μια διαδρομή που μας οδηγεί από την 1 στην 2 με πιθανότητα που σύμφωνα με την πρώτη γραμμή και δεύτερη στήλη του πίνακα είναι: $P_{12}(1) = 0.6$. Γενικά λοιπόν έχουμε πως όταν το $n = 1$ ισχύει πως:

$$P_{ij}(1) = A_{ij}$$

Τι συμβαίνει στην περίπτωση όπου $n = 2$; Τώρα υπάρχουν περισσότεροι του ενός τρόποι να καταλήξουμε από την κατάσταση 1 στην κατάσταση 2 σε δυο βήματα. Είτε στο πρώτο βήμα να πάμε στην 3 και από εκεί στην 2 είτε να παραμείνουμε στο πρώτο βήμα στην 1 και στο δεύτερο βήμα να βρεθούμε στην 2. Σε κάθε μια εκ των περιπτώσεων, η πιθανότητα να συμβεί δίνεται από το γινόμενο των πιθανοτήτων κάθε μετάβασης, δηλαδή στην διαδρομή $1 \rightarrow 3 \rightarrow 2$ είναι $A_{13} \times A_{32} = 0.02$, ενώ για την διαδρομή $1 \rightarrow 1 \rightarrow 2$ είναι $A_{11} \times A_{12} = 0.12$. Συνολικά λοιπόν, η πιθανότητα να βρεθούμε από την 1 στην 2 σε δυο βήματα είναι $0.02 + 0.12 = 0.14$.

Δείτε όμως την γενική εικόνα του πως έχει προκύψει το αποτέλεσμα αυτό. Ας γράψουμε αναλυτικά την έκφραση:

$$P_{12}(2) = A_{11} \times A_{12} + A_{12} \times A_{22} + A_{13} \times A_{32}$$

Οι παραπάνω συνδυασμοί αποτελούν τις μοναδικές περιπτώσεις να βρεθούμε από την κατάσταση 1 στην κατάσταση 2 μετά από δυο βήματα. Τον δεύτερο όρο δεν τον υπολογίσαμε παραπάνω καθώς δεν υπάρχει πιθανότητα να παραμείνουμε στην κατάσταση 2 αν φτάσουμε σε αυτήν. Μπορούμε να γράψουμε την παραπάνω σχέση με διανυσματική μορφή για απλότητα:

$$P_{12}(2) = \begin{bmatrix} A_{11} & A_{12} & A_{13} \end{bmatrix} \times \begin{bmatrix} A_{12} \\ A_{22} \\ A_{32} \end{bmatrix}$$

όπου τώρα φαίνεται εύκολα πως πρόκειται για τον πολλαπλασιασμό της πρώτης γραμμής με την δεύτερη στήλη του πίνακα των μεταβάσεων. Μπορείτε να επιβεβαιωθείτε το αποτέλεσμα αντικαθιστώντας τις τιμές. Με την ίδια λογική παρατηρούμε πως για παράδειγμα η πιθανότητα $P_{13}(2)$ δίνεται ως:

$$P_{13}(2) = \begin{bmatrix} A_{11} & A_{12} & A_{13} \end{bmatrix} \times \begin{bmatrix} A_{13} \\ A_{23} \\ A_{33} \end{bmatrix}$$

Συνεπώς, ο πρώτος δείκτης μας δείχνει την γραμμή και ο δεύτερος την στήλη τις οποίες πρέπει να πολλαπλασιάσουμε ώστε να βρούμε το αποτέλεσμα.

Ο υπολογισμός των παραπάνω πιθανοτήτων θυμίζει τα στοιχεία του πολλαπλασιασμού δυο πινάκων. Στη συγκεκριμένη περίπτωση του πίνακα μεταβάσεων με τον εαυτό του. Ας δούμε ποιο είναι το αποτέλεσμα:

$$A^2 = \begin{pmatrix} 0.2 & 0.6 & 0.2 \\ 0.3 & 0.0 & 0.7 \\ 0.4 & 0.1 & 0.5 \end{pmatrix} \times \begin{pmatrix} 0.2 & 0.6 & 0.2 \\ 0.3 & 0.0 & 0.7 \\ 0.4 & 0.1 & 0.5 \end{pmatrix} = \begin{pmatrix} 0.30 & 0.14 & 0.56 \\ 0.34 & 0.25 & 0.41 \\ 0.31 & 0.29 & 0.40 \end{pmatrix}$$

Δείτε την θέση 1,2 του πίνακα. Πρόκειται βέβαια για την πιθανότητα $P_{12}(2)$ που βρήκαμε παραπάνω η οποία προέκυψε ως ο πολλαπλασιασμός της πρώτης γραμμής με την δεύτερη στήλη, αποτέλεσμα που αναπαράγεται κατά τον υπολογισμό του πίνακα A^2 . Γενικά λοιπόν ισχύει η εξής ωραία ιδιότητα:

$$P_{ij}(2) = A_{ij}^2$$

Αρχίζει πλέον να γίνεται εμφανής η γενίκευση στην περίπτωση n βημάτων. Η πιθανότητα να καταλήξουμε στην κατάσταση j ξεκινώντας από την i έπειτα από n βήματα, δίνεται από την γενική σχέση:

$$P_{ij}(n) = A_{ij}^n \quad (2.10)$$

Όταν $n \rightarrow \infty$, παρατηρούμε πως κάθε στήλη του πίνακα A_{ij}^n καταλήγει στην ίδια σταθερή τιμή. Δείτε το αποτέλεσμα για το παράδειγμά μας:

$$A^{10} = \begin{pmatrix} 0.3139 & 0.2336 & 0.4526 \\ 0.3139 & 0.2336 & 0.4526 \\ 0.3139 & 0.2336 & 0.4526 \end{pmatrix}$$

Το γεγονός αυτό υποδεικνύει πως για πολλά βήματα, δεν έχει σημασία το σημείο εκκίνησης της αλυσίδας. Πρέπει βέβαια να ικανοποιούνται κάποιες ωραίες ιδιότητες της αλυσίδας, όπως να είναι αδιαίρετη και απεριοδική, όπως συμβαίνει εδώ, ωστόσο η μελέτη αυτή βρίσκεται εκτός πλαισίων της εργασίας.

Οι παραπάνω τιμές αντιπροσωπεύουν λοιπόν, την στατική κατανομή της αλυσίδας και τον νόμο των μεγάλων αριθμών. Όπως στην περίπτωση του ζαριού, κάθε πλευρά περιμένουμε να εμφανίζεται συγκεκριμένο αριθμό περιπτώσεων, έτσι και εδώ τα παραπάνω ποσοστά μας δίνουν το πόσο συχνά εμφανίζεται κάθε κατάσταση μετά από πολλά βήματα. Μπορούμε να απομονώσουμε μια από τις γραμμές του πίνακα και να θεωρήσουμε το διάνυσμα ως την κατάσταση ισορροπίας:

$$\pi = (0.3139 \quad 0.2336 \quad 0.4526)$$

Τι ισχύει λοιπόν στην κατάσταση ισορροπίας; Ο πολλαπλασιασμός με των πίνακα των μεταβάσεων αφήνει το διάνυσμα σταθερό, δηλαδή ισχύει πως:

$$\pi A = \pi$$

Η παραπάνω εξίσωση αποτελεί μια εξίσωση ιδιοτιμών, την οποία μπορούμε να λύσουμε και να βρούμε τις ιδιοτιμές του πίνακα. Μια αρχική κατάσταση της αλυσίδας μπορεί να είναι η $\pi = (1 \quad 0 \quad 0)$ κατά την οποία βρισκόμαστε στην κατάσταση 1. Καθώς εξελίσσουμε την κατάσταση αυτή μέσω του πίνακα μεταβάσεων, θα καταλήξουμε εν τέλει στην σταθερή κατάσταση ισορροπίας που μας δίνει τις πιθανότητες

εμφάνισης κάθε κατάστασης. Τελικά λοιπόν, αρκεί να λύσουμε το σύστημα εξισώσεων της εξίσωσης ιδιοτιμών. Στο παράδειγμά μας:

$$\pi = (\pi_1, \pi_2, \pi_3), \quad A = \begin{bmatrix} 0.2 & 0.6 & 0.2 \\ 0.3 & 0.0 & 0.7 \\ 0.4 & 0.1 & 0.5 \end{bmatrix}$$

$$\pi A = \pi \iff (\pi_1, \pi_2, \pi_3) \begin{bmatrix} 0.2 & 0.6 & 0.2 \\ 0.3 & 0.0 & 0.7 \\ 0.4 & 0.1 & 0.5 \end{bmatrix} = (\pi_1, \pi_2, \pi_3).$$

Αναπτύσσουμε τις εξισώσεις:

$$\begin{cases} \pi_1 = 0.2\pi_1 + 0.3\pi_2 + 0.4\pi_3, \\ \pi_2 = 0.6\pi_1 + 0.0\pi_2 + 0.1\pi_3, \\ \pi_3 = 0.2\pi_1 + 0.7\pi_2 + 0.5\pi_3, \end{cases}$$

και εισάγουμε και την συνθήκη κανονικοποίησης:

$$\pi_1 + \pi_2 + \pi_3 = 1.$$

Η συνθήκη κανονικοποίησης είναι απαραίτητη καθώς κάθε πολλαπλασιαστικός παράγοντας της λύσης της εξίσωσης ιδιοτιμών παραμένει λύση της εξίσωσης αφού απλοποιείται και στα δυο μέρη. Κατά αυτόν τον τρόπο, εξασφαλίζουμε και την ερμηνεία του αποτελέσματος ως πιθανότητα, αφού το άθροισμα των πιθανοτήτων να βρεθούμε σε κάποια τυχαία κατάσταση πρέπει να αθροίζει στην μονάδα. Η λύση του παραπάνω συστήματος, είναι ακριβώς η λύση που βρήκαμε πολλαπλασιάζοντας τον πίνακα μεταβάσεων με τον εαυτό του πολλές φορές. Πρόκειται βέβαια για τον αλγεβρικό υπολογισμό της στατικής κατανομής.

Θα ελέγξουμε τα παραπάνω με έναν απλό κώδικα, ο οποίος πρώτον θα λύνει την εξίσωση ιδιοτιμών υπολογίζοντας την αναμενόμενη πιθανότητα εμφάνισης κάθε κατάστασης και έπειτα θα τρέξει την αλυσίδα για πληθώρα βημάτων ώστε να επιβεβαιώσει τα αποτελέσματα. Δείτε τον κώδικα παρακάτω.

```
1 import numpy as np
2
```

```

3 # Our example transition array
4 A = np.array([
5     [0.2, 0.6, 0.2],
6     [0.3, 0.0, 0.7],
7     [0.4, 0.1, 0.5]
8 ])
9
10 # First, lets find the theoretical stationary distribution
11 # We solve the eigenvalue equation A = with 1+2+3 = 1
12 w, v = np.linalg.eig(A.T)
13 stat = v[:, np.isclose(w, 1)]
14 stat = stat[:, 0].real
15 stationary = stat / stat.sum()
16
17 print("Theoretical stationary distribution:")
18 print(stationary)
19
20 # Lets simulate the Markov chain to check the theory
21 n_steps = 100_000
22 counts = np.zeros(3, dtype=int)
23
24 # start from state 0. It doesnt matter what state we choose
25 # 0,1,2 represent the example states 1,2,3
26 current = 0
27 for _ in range(n_steps):
28     counts[current] += 1
29     current = np.random.choice([0, 1, 2], p=A[current])
30
31 empirical = counts / counts.sum()
32
33 print("\nEmpirical frequencies after 100000 steps:")
34 print(empirical)

```

Listing 2.3: This piece of code runs the Markov chain of our example both theoretical and empirical.

Το παραπάνω κομμάτι κώδικα δείχνει μερικές από τις εντυπωσιακές δυνατότητες του numpy. Αρχικά, στην γραμμή 12 βλέπουμε πως υπάρχει εντολή για την αυτόματη λύση ενός συστήματος ιδιοτιμών. Χρειάζεται ωστόσο να λύσουμε για τον πίνακα A^T λόγω του ότι η είσοδος θεωρείται σε μορφή στήλης. Λύνουμε δηλαδή την εξίσωση $A^T \pi^T = \pi^T$. Η μεταβλητή w είναι ένα διάνυσμα με τις ιδιοτιμές, ενώ η μεταβλητή v ένας πίνακας 3 επί 3 με κάθε στήλη του να αντιστοιχεί στο αντίστοιχο ιδιοδιάνυσμα κάθε ιδιοτιμής. Έπειτα, αρκεί να κρατήσουμε την ιδιοτιμή ίση με την μονάδα η οποία αντιστοιχεί στην στατική κατάσταση της αλυσίδας Markov.

Δείτε τώρα την εντολή στην γραμμή 29. Εδώ επιλέγουμε τυχαία την επόμενη κατάσταση. Ο πιο απλός τρόπος θα ήταν να επιλέξουμε έναν τυχαίο αριθμό ομοιόμορφης κατανομής από το 0 έως το 1 και με μια εντολή if else if να διακρίνουμε

περιπτώσεις. Το numpry έχει προβλέψει αυτή την ανάγκη μας και μας δίνει έτοιμη την λύση. Η πρώτη παράμετρος δίνει τους αριθμούς από τους οποίους θα γίνει η επιλογή, ενώ η δεύτερη παράμετρος δίνει την βαρύτητα με την οποία επιλέγεται κάθε αριθμός. Current για εμάς είναι η τωρινή κατάσταση, η μοναδική που μας έχει σημασία στην συνέχεια της αλυσίδας. Έτσι, ανάλογα με την τωρινή κατάσταση 0,1 ή 2 επιλέγονται ως βαρύτητες η αντίστοιχη γραμμή του πίνακα μεταβάσεων.

Το αποτέλεσμα είναι το παρακάτω:

Theoretical stationary distribution:
[0.31386 0.23357 0.45255]

Empirical frequencies after 100000 steps:
[0.31487 0.23303 0.45210]

Κατά αυτόν τον τρόπο, η αλυσίδα ακολουθεί τον νόμο των μεγάλων αριθμών παρότι η πιθανότητα κάθε επόμενης κατάστασης είναι δεσμευμένη από το αποτέλεσμα του προηγούμενου βήματος.

2.4.3 Mixing Time

Ο χρόνος ανάμειξης (mixing time) μιας αλυσίδας Markov είναι ο χρόνος μέχρι η αλυσίδα Markov να πλησιάσει την κατανομή της σταθερής κατάστασής της. Είδαμε πως μια μη περιοδική και μη αναγώγιμη αλυσίδα έχει μια μοναδική κατανομή π και ανεξάρτητα από την αρχική κατάσταση, η κατανομή της αλυσίδας στο χρόνο t συγκλίνει στο π καθώς το t τείνει στο άπειρο. Έτσι, μια αλυσίδα Markov δεν έχει μνήμη, ή αλλιώς η αρχική κατάσταση δεν επηρεάζει στο ελάχιστο την κατάσταση ισορροπίας. Το mixing time μετρά λοιπόν το πόσο γρήγορα η αλυσίδα ξεχνά την αρχική κατάσταση με συνέπεια το κάθε δείγμα να θεωρείται σχεδόν ανεξάρτητο.

Σε σχέση με τα πολύτοπα και την δειγματοληψία σε αυτά που θα δούμε στην συνέχεια, μπορούμε να ορίσουμε το mixing time ως εξής: Έστω πολύτοπο $P \subset \mathbb{R}^d$ και αλυσίδα $X_0, X_1, \dots$ με στατική κατανομή π έστω ομοιόμορφη εντός του πολύτοπου. Ο χρόνος ανάμειξης $t_{\min}(\epsilon)$ είναι ο μικρότερος αριθμός από βήματα t τέτοιος ώστε η κατανομή X_t ξεκινώντας από οποιοδήποτε σημείο $X_0 \in P$ να βρίσκεται σε απόσταση μικρότερη από ϵ από την κατανομή ισορροπίας π ως προς την ολική μεταβολή.

$$t_{\text{mix}}(\varepsilon) := \inf \left\{ t \geq 0 \mid \sup_{x_0 \in P} \|\mathbb{P}[X_t \in \cdot \mid X_0 = x_0] - \pi\|_{\text{TV}} \leq \varepsilon \right\}.$$

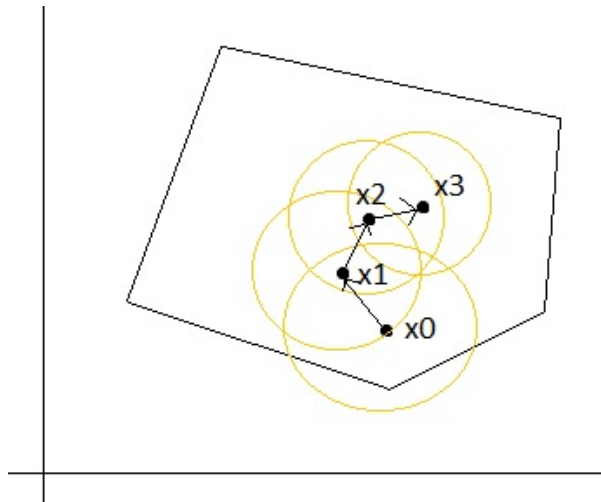
Θυμηθείτε το παράδειγμα της προηγούμενης ενότητας. Σε κάθε βήμα, η κατανομή των πιθανοτήτων εμφάνισης κάθε κατάστασης προσσεγγίζει όλο και περισσότερο την σταθερή κατανομή. Από κάποιο σημείο και έπειτα λοιπόν, η διαφορά αυτή γίνεται συνεχώς μικρότερη έτσι ώστε όσο μικρό αριθμό ε και να επιλέξουμε, να ισχύει η παραπάνω συνθήκη. Από το σημείο αυτό και πέρα, θεωρούμε πως το σύστημα έχει ξεχάσει την αρχική κατάσταση και τα μετέπειτα δείγματα είναι ανεξάρτητα μεταξύ τους.

Όταν για παράδειγμα θέλουμε να επιτύχουμε μια ομοιόμορφη κατανομή εντός ενός πολυτόπου, τα αρχικά δείγματα ενός τυχαίου περιπάτου έχουν άμεση σχέση με το σημείο εκκίνησης. Αντιθέτως, έπειτα από κάποιο αριθμό βημάτων, τα δείγματα που παράγονται θα αντιστοιχούν στην κατανομή στόχο. Συνεπώς, τα βήματα του mixing time αποτελούν τον ελάχιστο αριθμό βημάτων ώστε να πετύχουμε την ζητούμενη κατανομή εντός του πολυτόπου. Αν η δειγματοληψία περιλαμβάνει λιγότερα βήματα, αυτά κατά μεγάλη πιθανότητα θα είναι συσχετιζόμενα μεταξύ τους και δεν θα ανταποκρίνονται στην κατανομή στόχο.

2.4.4 Ball Walk

Η αποτυχία του accept - reject sampling για την δειγματοληψία σε πολλές διαστάσεις μας οδηγεί αναπόφευκτα στην αναζήτηση διαφορετικών μεθόδων δειγματοληψίας, ιδανικά ανεξάρτητες του αριθμού των διαστάσεων. Η πρώτη από αυτές τις μεθόδους είναι η ονομαζόμενη ball walk το όνομα της οποίας θα γίνει κατανοητό στη συνέχεια.

Θεωρούμε ένα πολύγωνο τυχαίων διαστάσεων $P \subset \mathbb{R}^d$ και μια ακτίνα $r > 0$. Αρχικά, επιλέγουμε ένα αρχικό σημείο εντός του πολυγώνου $x_0 \in P$. Στη συνέχεια, θεωρούμε την υπερσφαίρα με κέντρο το σημείο x_0 και ακτίνα r . Ονομάζουμε την υπερσφαίρα αυτή "μπάλα" από όπου προέρχεται και η ονομασία της μεθόδου. Δειγματοληπτούμε λοιπόν, από την μπάλα $B(x_0, r)$ ομοιόμορφα. Αν το δείγμα βρίσκεται εντός του πολυγώνου, δηλαδή $u \in P$, τότε το αποδεχόμαστε και θέτουμε $x_{t+1} = x_t$. Το δείγμα γίνεται τώρα το καινούργιο κέντρο από το οποίο σχεδιάζουμε την υπερσφαίρα με ακτίνα r . Αν $u \notin P$, τότε το απορρίπτουμε και συνεχίζουμε την δειγματοληψία εντός της ίδιας υπερσφαίρας έως ότου βρούμε σημείο εντός του πολυγώνου. Η παρακάτω εικόνα δείχνει τα βήματα της μεθόδου.



Σχήμα 2.14: Η μέθοδος τυχαίου περιπάτου ball walk. Ξεκινούμε από ένα σημείο x_0 εντός του πολυγώνου και βρίσκουμε την σφαίρα με ακτίνα r και κέντρο το x_0 . Λαμβάνουμε ένα δείγμα από την σφαίρα το οποίο ελέγχουμε εάν βρίσκεται εντός του πολυγώνου. Συνεχίζουμε με τον ίδιο τρόπο. Η τελική διάταξη των δειγμάτων εντός του πολυγώνου ακολουθεί την ομοιόμορφη κατανομή, δηλαδή ο λόγος δειγμάτων/όγκο είναι σταθερός σε όλο το πολύγωνο.

Γίνεται φανερή τώρα, το γιατί η μέθοδος χαρακτηρίζεται ως MCMC. Η δειγματοληψία προσομοιώνει μια αλυσίδα Markov κατά την οποία κάθε νέο πείραμα (η εύρεση ενός δείγματος) είναι εξαρτώμενο μόνο από το προηγούμενό του, δηλαδή την θέση του κέντρου της μπάλας από την οποία δειγματοληπτούμε στο παρόν βήμα. Η αλυσίδα, βάση της κατασκευής της έχει τις ιδιότητες που είδαμε στο παράδειγμα της προηγούμενης ενότητας. Είναι μη αναγώγιμη, δηλαδή κάθε κατάσταση είναι προσβάσιμη από όλες τις υπόλοιπες έπειτα από ικανό αριθμό βημάτων και απεριοδική, δηλαδή δεν υπάρχουν βρόγχοι στους οποίους η διαδικασία εγκλωβίζεται. Τελικά λοιπόν, η αρχική κατάσταση, ή αλλιώς το αρχικό σημείο που επιλέγουμε ώστε να αρχίσει η δειγματοληψία δεν έχει σημασία αφού η αλυσίδα δεν έχει μνήμη. Τα δείγματα τελικά θα κατανεμηθούν σε όλο τον όγκο του πολυγώνου. Επιπλέον, η στατική κατάσταση της αλυσίδας είναι η ομοιόμορφη κατανομή των δειγμάτων εντός του πολυγώνου και έτσι δεν είναι απαραίτητη η γνώση της ακριβούς κατανομής του πολυγώνου εκ των προτέρων.

Σημαντικό ρόλο παίζει εδώ η επιλογή της ακτίνας r . Μικρή ακτίνα και ο περιπάτος γίνεται υπερβολικά αργός με βαρύ υπολογιστικό κόστος. Από την άλλη, μεγάλη ακτίνα δίνει μεγάλη πιθανότητα τα δείγματα να βρίσκονται τελικά εκτός της μπάλας και έτσι η διαδικασία να κολλάει και να κοστίζει υπολογιστικά. Ιδανικά επιλέγουμε την ακτίνα ανάλογα με την γεωμετρία του πολυγώνου. Καλή επιλογή αποτελεί η αναλογία $r \sim 1/\sqrt{d}$.

Ας επιβεβαιώσουμε την σημαντική ιδιότητα που ισχυρίζεται πως η τελική κατανομή των δειγμάτων είναι ομοιόμορφη. Σε αντίθεση με το παράδειγμα της προηγούμενης ενότητας, κάθε κατάσταση τώρα οδηγεί σε άπειρες εν γένει καταστάσεις, όσα και τα σημεία εντός της μπάλας. Έτσι, η πιθανότητα μετάβασης σε κάθε μια από αυτές τις άπειρες καταστάσεις είναι μηδενική. Για τον λόγο αυτό, αντί του πίνακα μεταβάσεων χρησιμοποιούμε τώρα τον πυρήνα Markov ο οποίος δίνει την πιθανότητα μετάβασης ανά μονάδα όγκου της μπάλας $K(x, dy)$. Η πιθανότητα μετάβασης τώρα, σε μια κατάσταση εντός ενός συνόλου $B \subset K$ θα δίνεται από την σχέση:

$$\Pr ob(X_{t+1} \in B | X_t = x) = \int_B K(x, dy)$$

Μπορούμε να διαβάσουμε το K ως την πιθανότητα να πάμε από την κατάσταση x σε ένα μικρό χώρο γύρω από την κατάσταση y . Όπως στην περίπτωση της πεπερασμένης αλυσίδας το άθροισμα κάθε γραμμής του πίνακα είναι ίσο με την μονάδα, έτσι και εδώ ισχύει πως:

$$\int_{y \in K} K(x, dy) = 1$$

και η εξίσωση ιδιοτιμών που πρέπει να λύσουμε για τον υπολογισμό της στατικής κατανομής είναι η:

$$\pi K = \pi$$

Στην συγκεκριμένη περίπτωση, αντί να λύσουμε την δύσκολη κατά τα άλλα εξίσωση ιδιοτιμών θα χρησιμοποιήσουμε μια πιο εύκολη μέθοδο. Ισχύει το επόμενο θεώρημα (να κάνω reference to paper): Έστω K ο πίνακας μεταβάσεων μιας αλυσίδας Markov X_n και υποθέστε πως υπάρχει κατανομή π τέτοια ώστε να ισχύει $\pi_i K_{ij} = \pi_j K_{ji}$ (θυμηθείτε πως K_{ij} η πιθανότητα μετάβασης από την κατάσταση i στην κατάσταση j). Τότε η π είναι η στατική κατανομή της αλυσίδας και η X_n είναι αντιστρέψιμη σε σχέση με το π . Η απόδειξη του παραπάνω είναι απλή:

$$\sum_i \pi_i K_{ij} = \sum_i \pi_j K_{ji} = \pi_j \sum_i K_{ji} = \pi_j$$

και συνεπώς $\pi = \pi K$, άρα η π είναι στατική κατανομή αφού ικανοποιεί την εξίσωση ιδιοτιμών.

Χρησιμοποιώντας την ιδιότητα αυτή, αρκεί να μαντέψουμε την στατική κατανομή και να δοκιμάσουμε αν επιβεβαιώνει το θεώρημα. Έτσι λοιπόν, έστω π η κατάσταση ομοιόμορφης κατανομής:

$$\pi(dy) = \frac{1}{\lambda(K)} dy, \quad y \in K,$$

όπου $\lambda(K)$ είναι ο όγκος του K .

Για $x, y \in K$ με $x \neq y$, έχουμε πως:

$$\pi(dx) K(x, dy) = \frac{1}{\lambda(K)} dx \cdot \frac{1}{\text{Vol}(B(0, r))} 1_{\{\|x-y\| \leq r\}} dy,$$

και παρομοίως:

$$\pi(dy) K(y, dx) = \frac{1}{\lambda(K)} dy \cdot \frac{1}{\text{Vol}(B(0, r))} 1_{\{\|y-x\| \leq r\}} dx.$$

Όμως $1_{\{\|x-y\| \leq r\}} = 1_{\{\|y-x\| \leq r\}}$, και έπεται πως

$$\pi(dx) K(x, dy) = \pi(dy) K(y, dx).$$

Επειδή η προτεινόμενη κατανομή είναι συμμετρική στα x και y , ισχύει το θεώρημα και άρα $\pi K = \pi$. Συνεπώς, η ομοιόμορφη κατανομή αποτελεί στατική κατανομή της αλυσίδας του τυχαίου περιπάτου.

Μια καλή ερώτηση που προκύπτει από την παραπάνω ανάλυση αφορά το ποιά είναι η αρχική κατάσταση της αλυσίδας. Μπορεί να είναι οποιοδήποτε σημείο εντός του πολυτόπου, λόγω της ιδιότητας της αλυσίδας να μην κρατά μνήμη, αλλά εμείς δεν ξέρουμε κανένα τέτοιο σημείο. Συνεπώς, η εύρεση ενός αρχικού σημείου από όπου θα ξεκινήσει η δειγματοληψία χρειάζεται ξεχωριστή ανάλυση η οποία θα φανεί στο παράδειγμα της επόμενης ενότητας. Η γενική μέθοδος για την εύρεση ενός κατάλληλου σημείου είναι ο γραμμικός προγραμματισμός.

Όσον αφορά το mixing time της μεθόδου, σύμφωνα με την βιβλιογραφία είναι της τάξης του:

$$O\left(d^2 R^2 / r^2 \cdot \log\left(\frac{1}{\varepsilon}\right)\right)$$

όπου R και r είναι οι ακτίνες της περιγεγραμμένης και της εγγεγραμμένης μπάλας στο πολύτοπο.

2.4.5 Εφαρμογή της μεθόδου ball walk

Στα προηγούμενα είδαμε την θεωρία και την λογική πίσω από την μέθοδο ball walk. Πρόκειται για μια γεωμετρική μέθοδο τυχαίου περιπάτου της κατηγορίας MCMC την οποία και θα δοκιμάσουμε ευθύς αμέσως με ένα χαρακτηριστικό παράδειγμα, πρώτα στις απολύτως κατανοητές δυο διαστάσεις, αλλά έπειτα και σε πολλές διαστάσεις. Αρχικά είναι απαραίτητο να μελετήσουμε κάποια ειδικά θέματα, τα οποία αποδεικνύονται ζωτικής σημασίας στην εφαρμογή της μεθόδου. Το πρώτο από αυτά αφορά την ομοιόμορφη δειγματοληψία πάνω σε μια υπερσφαίρα. Έπειτα από την σχεδίαση της υπερσφαίρας με κέντρο το προηγούμενο έγκυρο δείγμα, πρέπει να βρούμε ένα τυχαίο σημείο εντός αυτής.

Ας ξεκινήσουμε ορίζοντας τα αντικείμενα αυτά. Χρησιμοποιούμε τον όρο υπερσφαίρα για να αναφερθούμε στη γενική εκδοχή της σφαίρας σε έναν αυθαίρετο αριθμό διαστάσεων. Έτσι, η δισδιάστατη υπερσφαίρα και η τρισδιάστατη υπερσφαίρα αντιστοιχούν, αντίστοιχα, σε αυτό που συνήθως ονομάζουμε κύκλο και σφαίρα. Μπορούν επίσης να οριστούν υπερσφαίρες με πολλές περισσότερες διαστάσεις, ακόμη και αν η ικανότητά μας να τις απεικονίσουμε είναι περιορισμένη από την καθημερινή μας εμπειρία του τρισδιάστατου χώρου (προσπαθήστε για μια στιγμή να φανταστείτε την όψη μιας σφαίρας σε τέσσερις διαστάσεις). Ωστόσο, για να γλιτώσουμε από το να διαβάζουμε πολλές μακρές λέξεις, θα αναφερόμαστε στις υπερσφαίρες απλά ως σφαίρες. Επιπλέον, για λόγους απλότητας, θα εξετάσουμε μόνο την περίπτωση της μονάδας σφαίρας, η οποία έχει ακτίνα $r = 1$.

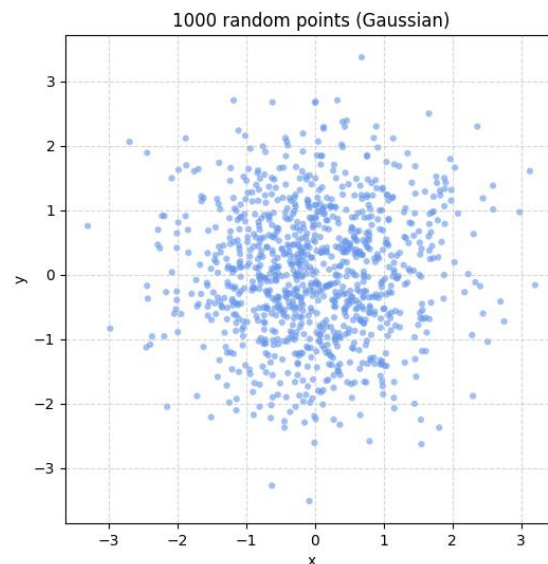
Γνωρίζουμε ήδη να δειγματοληπτούμε σε μια σφαίρα μέσω της accept-reject μεθόδου όπου αρκεί να βρούμε έναν κύβο που την περιέχει. Ωστόσο, η μέθοδος καταρρέει σε πολλές διαστάσεις. Για τον λόγο αυτό χρειαζόμαστε μια διαφορετική προσέγγιση επί του θέματος. Θα χρησιμοποιήσουμε την κανονική (γκαουσιανή) κατανομή. Η κατανομή αυτή έχει την ενδιαφέρουσα ιδιότητα πως ο συνδυασμός d ανεξαρτήτων τέτοιων κατανομών δημιουργεί μια ισοτροπική κατανομή σε d διαστάσεις. Ισοτροπική σημαίνει ομοιόμορφη σε όλες τις κατευθύνσεις. Συνεπώς, σε όποια κατεύθυνση από το κέντρο της σφαίρας προς τα έξω κοιτάζουμε, θα βρούμε πάνω κάτω τον ίδιο αριθμό δειγμάτων.

Ο κώδικας για να δείξουμε τον παραπάνω ισχυρισμό είναι απλός. Ξεκινώντας από τις δυο διαστάσεις, θεωρούμε ένα κύκλο με κέντρο το 0 και ακτίνα 1. Παίρνουμε τώρα, τυχαία διανύσματα, κάθε συντεταγμένη των οποίων προέρχεται από ένα τυχαίο αριθμό της κανονικής κατανομής $x, y \sim N(0, 1)$. Τοποθετούμε τα σημεία αυτά στο γράφημα ώστε να μελετήσουμε την κατανομή τους.

```
1 n = 1000
2 x = np.random.normal(size=n)
3 y = np.random.normal(size=n)
4
5 # Plot
6 plt.figure(figsize=(6,6))
7 plt.scatter(x, y, c="cornflowerblue", s=20, alpha=0.6,
8             edgecolors="none")
9 plt.xlabel("x")
10 plt.ylabel("y")
11 plt.title("1000 random points (Gaussian)")
12 plt.grid(True, linestyle="--", alpha=0.5)
13 plt.show()
```

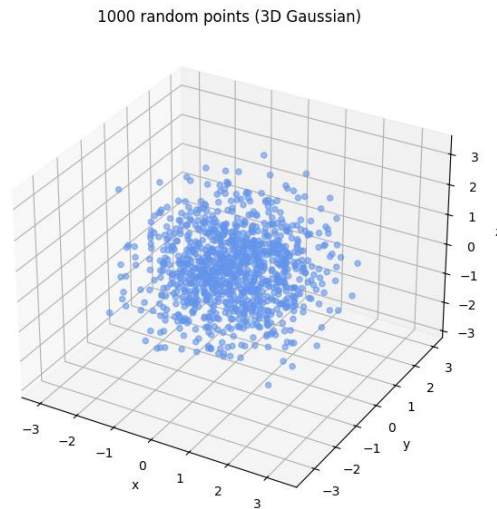
Listing 2.4: We sample 2D vectors using the Gaussian Distribution.

Το αποτέλεσμα φαίνεται στην επόμενη εικόνα.



Σχήμα 2.15: Στο γράφημα βλέπουμε τοποθετημένα 1000 δείγματα. Η x και η y συντεταγμένη κάθε δείγματος έχουν προέλθει τυχαία από την κανονική κατανομή με μέσο το 0 και απόκλιση 1.

Στο σχήμα παρατηρούμε την υποψία ενός κύκλου αλλά με τα περισσότερα σημεία εστιάζονται γύρω από το κέντρο. Σε κάθε περίπτωση όμως, η ισοτροπία της κατανομής είναι δεδομένη. Επιπλέον, για την κανονική κατανομή ισχύει πως $f(x) \propto \exp(-\frac{1}{2}\|x\|^2)$ με συνέπεια η πιθανότητα δειγμάτων στα όρια του κύκλου να είναι μικρή καθώς μεγαλώνει η απόσταση από το κέντρο. Στη συνέχεια δείχνουμε και το αποτέλεσμα για τις 3 διαστάσεις.



Σχήμα 2.16: Στο γράφημα βλέπουμε τοποθετημένα 1000 δείγματα. Η x , η y και η z συντεταγμένη κάθε δείγματος έχουν προέλθει τυχαία από την κανονική κατανομή με μέσο το 0 και απόκλιση 1.

Αν και η κατανομές αυτές είναι ισότροπες, δεν είναι ομοιόμορφες όπως απαιτεί η μέθοδος ball walk. Πρέπει να μετατρέψουμε αυτό το σύννεφο από σημεία σε μια τέλεια σφαίρα. Ο τελικός στόχος είναι να καταλήξουμε σε μια σταθερή τιμή για τα σημεία/όγκο. Πρέπει να προσέξουμε ιδιαίτερα καθώς όσο απομακρυνόμαστε από το κέντρο, ο όγκος της σφαίρας μεγαλώνει εκθετικά και άρα χρειάζονται περισσότερα σημεία προς το άκρο της σφαίρας ώστε να πετύχουμε την ομοιόμορφη κατανομή.

Αρχικά, κανονικοποιούμε κάθε διάνυσμα ώστε το μέτρο του να είναι μεταξύ 0 και 1. Αρκεί να διαιρέσει κανείς με την νόρμα του διανύσματος $u = z/\|z\|$. Στη συνέχεια, παρατηρούμε πως ο όγκος της σφαίρας αυξάνει ανάλογα με το r^d (θυμηθείτε τις σχέσεις που δίνουν την επιφάνεια ενός κύκλου και τον όγκο μιας σφαίρας). Τότε, αν θέλουμε ομοιόμορφη κατανομή πρέπει:

$$P(\|x\| \leq r) = \frac{V(r)}{V(R)} = \left(\frac{r}{R}\right)^d$$

Αν παραγωγίσουμε ως προς r :

$$f(r) = \frac{d}{R^d} r^{d-1}, 0 \leq r \leq 1$$

Αυτή είναι η ακτινική πυκνότητα πιθανότητας η οποία περιγράφει και την ομοιόμορφη κατανομή στην σφαίρα. Για τον σκοπό αυτό, κανονικοποιούμε την νόρμα κάθε δείγματος με τον αριθμό $\rho = R \cdot U^{1/d}$ με U ένα τυχαίο δείγμα της ομοιόμορφης κατανομής από το 0 στο 1, $U \sim \text{uniform}[0, 1]$. Ο μετασχηματισμός λειτουργεί καθώς τα δείγματα θα κατανέμονται τώρα ως:

$$P(\rho \leq r) = P\left(U^{1/d} \leq \frac{r}{R}\right) = P\left(U \leq \left(\frac{r}{R}\right)^d\right) = \left(\frac{r}{R}\right)^d$$

με πυκνότητα πιθανότητα που δίνεται από την παραγωγή ως προς r και άρα ικανοποιεί την ομοιόμορφη κατανομή.

Ο κώδικας είναι σχετικά απλός καθώς το numpy περιέχει έτοιμες συναρτήσεις που λύνουν τα περισσότερα προβλήματά μας:

```

1 def sample_ball(d, r=1.0, n=1):
2     """
3     This function samples n points uniformly from a d-dimensional
4     ball of radius r.
5
6     Args:
7         d : int
8             Dimension of the ball
9         r : float
10            Radius of the ball
11         n : int
12            Number of points to sample
13
14     Returns:
15         numpy array of shape (n, d)
16     """
17     #Sample Gaussian vector for direction
18     z = np.random.normal(size=(n, d))
19     norms = np.linalg.norm(z, axis=1, keepdims=True)
20     u = z / norms # unit vectors

```

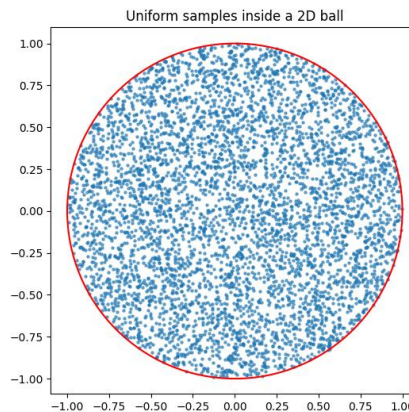
```

20     #Scale samples using  $R * U^{(1/d)}$ 
21     u_radius = np.random.uniform(0, 1, size=(n, 1))
22     rho = r * u_radius**(1/d)
23
24     return u * rho
25
26 # --- Lets run the 2D case ---
27 n_points = 5000
28 samples = sample_ball(d=2, r=1.0, n=n_points)
29
30 plt.figure(figsize=(6,6))
31 plt.scatter(samples[:,0], samples[:,1], s=5, alpha=0.6)
32 circle = plt.Circle((0,0), 1, color='r', fill=False,
33                     linewidth=1.5)
34 plt.gca().add_artist(circle)
35 plt.gca().set_aspect('equal', 'box')
36 plt.title("Uniform samples inside a 2D ball")
37 plt.show()

```

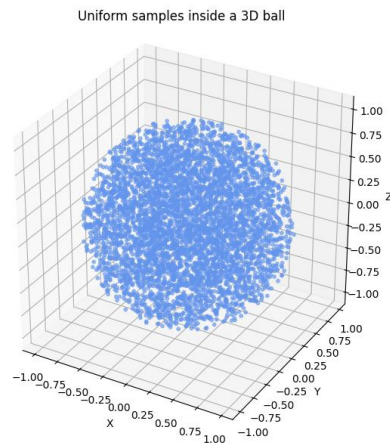
Listing 2.5: This code samples n points uniformly from a d -dimensional ball of radius r and displays the result.

Δείτε τις γραμμές 16, 17 και 21. Πρόκειται για έτοιμες εντολές του numpy που βρίσκουν τυχαίες τιμές εσ κατανομές ή υπολογίζουν αυτόματα τις νόρμες διανυσμάτων ανεξαρτήτως διαστάσεων. Το αποτέλεσμα που δίνει ο παραπάνω κώδικας φαίνεται στις επόμενες εικόνες, πρώτα για την 2D περίπτωση και έπειτα για την 3D.



Σχήμα 2.17: Στο γράφημα βλέπουμε τοποθετημένα 1000 δείγματα. Η x , η y και η z συντεταγμένη κάθε δείγματος έχουν προέλθει τυχαία από την κανονική κατανομή με μέσο το 0 και απόκλιση 1.

Φαίνεται λοιπόν πως έχουμε την σωστή κατανομή εντός της μπάλας. Συνεπώς,



Σχήμα 2.18: Στο γράφημα βλέπουμε τοποθετημένα 1000 δείγματα. Η x , y και z συντεταγμένη κάθε δείγματος έχουν προέλθει τυχαία από την κανονική κατανομή με μέσο το 0 και απόκλιση 1.

η διαδικασία με την οποία θα παίρνουμε δείγματα από τις μπάλες κατά την διάρκεια του τυχαίου περιπάτου είναι τώρα κατανοητή. Ένα σημαντικό πλεονέκτημα της παραπάνω μεθόδου με την χρήση της κανονικής κατανομής είναι πως δεν γίνεται έλεγχος για το αν το προκύπτων δείγμα ανήκει στην μπάλα. Όλα τα δείγματα που προκύπτουν βρίσκονται σίγουρα εντός αυτής, αρκεί να επιλέξουμε σωστά τις παραμέτρους της μπάλας κατά την εκτέλεση του κώδικα.

Όλα τα παραπάνω βέβαια, αφορούν μια μπάλα στην αρχή των αξόνων και με ακτίνα $r = 1$. Ωστόσο, κατά την διαδικασία της δειγματοληψίας, το κέντρο της μπάλας θα μεταβάλλεται κάθε φορά ανάλογα με το τρέχον δείγμα. Η αντιμετώπιση αυτής της δυσκολίας είναι αρκετά εύκολη. Μπορούμε να συνεχίσουμε να δειγματοληπτούμε από μια ιδανική μπάλα στην αρχή των αξόνων και απλώς να προσθέσουμε στις συντεταγμένες του δείγματος, το νέο κέντρο $(x_{\text{τελικό}}, y_{\text{τελικό}}) = (x_{\text{αρχικό}}, y_{\text{αρχικό}}) + c_{\text{κέντρον}}$. Οι εντολές που κωδικοποιούν την συμπεριφορά αυτή είναι οι παρακάτω:

```

1 #Consider a new random center
2 center = np.array([2.0, 3.0])
3 #and a smaller ball radius
4 radius = 0.5
5 #Then, sample points inside a ball with given center and radius
6 samples = sample_ball(d=2, r=radius, n=n_points) + center

```

Listing 2.6: In order to sample from a ball with a random center, we just translate the coordinates of the sample to the new center.

Η ομοιόμορφη κατανομή των δειγμάτων δεν αλλοιώνεται από τέτοιες χωρικές μετατοπίσεις, με συνέπεια η πρόσθεση των συντεταγμένων του νέου κέντρου στα δείγματα, απλά να τα μετατοπίσει στο επιθυμητό σημείο στο χώρο.

Το επόμενο πρόβλημα αφορά τώρα, την εύρεση ενός αρχικού δείγματος εντός του πολυτόπου από το οποίο να ξεκινήσει ο τυχαίος περίπατος. Στην περίπτωση που το πολύτοπο δίνεται ως ένα σύνολο ανισοτήτων, η καλύτερη μέθοδος είναι ο γραμμικός προγραμματισμός. Πρόκειται για μια μέθοδο βελτιστοποίησης ενός μοντέλου, τα αντικείμενα του οποίου εκράζονται στη μορφή ανισοτήτων. Τέτοιου είδους προβλήματα βελτιστοποίησης μπορεί να αφορούν οτιδήποτε, όπως την μεγιστοποίηση του κέρδους ή την ελαχιστοποίηση του κόστους μιας εταιρίας ή την εύρεση ενός σημείου στο εσωτερικό ενός πολυτόπου.

Ο γεωμετρικός τύπος των σημείων που ικανοποιούν τις ανισότητες του προβλήματος είναι ένα κυρτό πολύτοπο, αφού κάθε ανισότητα που οριοθετεί το πρόβλημα αντιπροσωπεύεται με μια ευθεία στον διδιάστατο χώρο. Το πρόβλημα της βελτιστοποίησης αφορά την εύρεση μιας μέγιστης ή μιας ελάχιστης τιμής. Οι τιμές αυτές αναγκαστικά βρίσκονται στα άκρα του γεωμετρικού τύπου, δηλαδή πάνω στις κορυφές του πολυτόπου.

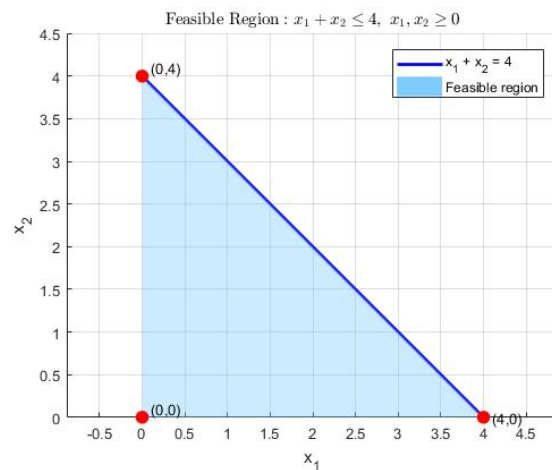
Ας δούμε ένα απλό παράδειγμα. Θεωρήστε πως θέλουμε να μεγιστοποιήσουμε την ποσότητα $3x_1 + 2x_2$ με τους εξής περιορισμούς: $x_1 + x_2 \leq 4$, $x_1 \geq 0$, $x_2 \geq 0$. Στο παράδειγμα αυτό υπάρχουν τρεις ανισότητες που πρέπει να ικανοποιούνται. Κάθε ανισότητα δείχνει ένα ημιεπίπεδο στο διδιάστατο χώρο. Η τομή όλων των ημιεπιπέδων αυτών, δίνει τον γεωμετρικό τύπο των σημείων που τις ικανοποιούν. Στο επόμενο σχήμα έχουμε σχεδιάσει τις ευθείες που αντιστοιχούν σε κάθε ανισότητα και έχουμε χρωματίσει την τομή των ημιεπιπέδων που ορίζουν.

Η μεγιστοποίηση της ζητούμενης ποσότητας γίνεται σε κάποια τιμοδοσία που αντιστοιχεί σε κορυφή του πολυτόπου. Στην συγκεκριμένη περίπτωση έχουμε τις τρεις εξής κορυφές:

Vertex (x_1, x_2)	$3x_1 + 2x_2$
$(0, 0)$	$3(0) + 2(0) = 0$
$(4, 0)$	$3(4) + 2(0) = 12$
$(0, 4)$	$3(0) + 2(4) = 8$

Συνεπώς, η βέλτιστη λύση για το πρόβλημα αυτό είναι η $x_1 = 4$ και $x_2 = 0$.

Στην περίπτωση ενός πολυτόπου ο σκοπός μας δεν είναι ακριβώς η βελτιστοποίηση κάποιας ποσότητας, παρά η εύρεση ενός έγκυρου σημείου για το σύνολο



Σχήμα 2.19: Στο γράφημα βλέπουμε το χώρο των σημείων που ικανοποιούν ταυτόχρονα όλες τις ανισότητες.

των ανισοτήτων που διαθέτουμε. Το σημείο αυτό μπορεί να χρησιμοποιηθεί για την εκκίνηση του τυχαίου περιπάτου.

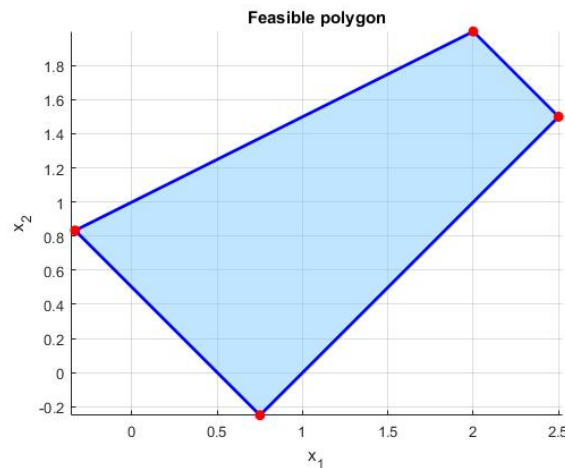
Για καλή μας τύχη, η python προσφέρει έτοιμα πακέτα για την επίλυση ενός προβλήματος γραμμικού προγραμματισμού. Για τον λόγο αυτό ας θεωρήσουμε κάτι λίγο πιο σύνθετο. Θεωρούμε το εξής πολύτοπο:

$$\begin{cases} x_1 + x_2 \leq 4, \\ x_1 - x_2 \leq 1, \\ -x_1 + 2x_2 \leq 2, \\ x_1 + x_2 \geq 0.5. \end{cases}$$

Οι ανισότητες πρέπει να δίνονται στην μορφή $Ax \leq B$ για αυτό και την τελευταία ανισότητα την βλέπουμε ως $-x_1 - x_2 \leq -0.5$. Μπορούμε να δούμε το γεωμετρικό τόπο των ανισοτήτων αυτών στο σχήμα 2.20.

Για το πολύγωνο αυτό, θα χρησιμοποιήσουμε γραμμικό προγραμματισμό ώστε να βρούμε ένα βέλτιστο σημείο που ικανοποιεί όλες τις ανισότητες. Όπως αναφέραμε και προηγουμένως, ο χαρακτηρισμός βέλτιστο δεν έχει εδώ ιδιαίτερη σημασία. Μας αρκεί ένα σημείο εντός του πολυγώνου, αφού οποιοδήποτε και αν είναι αυτό η κατανομή μετά από αρκετά βήματα καταλήγει πάντα ομοιόμορφη.

Δείτε τώρα πόσο εύκολο είναι το πρόβλημα του γραμμικού προγραμματισμού μέσω του πακέτου Scipy της python. Αρχικά, ορίζουμε το σύνολο των ανισοτήτων



Σχήμα 2.20: Ο γεωμετρικός τόπος των σημείων που ικανοποιούν ταυτόχρονα τις ανισότητες του παραδείγματος.

σε μορφή πίνακα:

```

1 import numpy as np
2 from scipy.optimize import linprog
3
4 # Objective function (we just want a feasible point, so we set to
  zero)
5 c = np.array([0, 0])
6
7 # Inequalities in the form A*x <= b as required
8 A = np.array([
9     [1, 1],      # x1 + x2 <= 4
10    [1, -1],     # x1 - x2 <= 1
11    [-1, 2],     # -x1 + 2x2 <= 2
12    [-1, -1]     # -x1 - x2 <= -0.5 (x1 + x2 >= 0.5)
13 ])
14 b = np.array([4, 1, 2, -0.5])
15
16 # No explicit bounds needed. The inequalities will define them.
17 bounds = [(None, None), (None, None)]

```

Listing 2.7: Here, we define the inequalities as a numpy matrix.

Η βασική λειτουργία του κώδικα πραγματοποιείται με την παρακάτω εντολή:

```

1 # Solve linear program
2 res = linprog(c, A_ub=A, b_ub=b, bounds=bounds, method='highs')

```

Listing 2.8: This line solves the linear programming.

Η συνάρτηση `linprog` δέχεται τα δεδομένα εισόδου, επιλύει το πρόβλημα με τη μέθοδο HiGHS και επιστρέφει το αποτέλεσμα στο αντικείμενο `res` (ή την αποτυχία της λύσης). Η μέθοδος HiGHS είναι μια αρκετά εκλεπτυσμένη μέθοδος, ικανή για την επίλυση των περισσότερων προβλημάτων σε σύντομο χρόνο. Συνίσταται στις περισσότερες περιπτώσεις. Όσον αφορά το αντικείμενο `res`, αυτό εκτός του αποτελέσματος περιλαμβάνει και πολλές επιπλέον πληροφορίες που μπορεί να μας φανούν χρήσιμες. Το μόνο που απομένει τώρα είναι να τυπώσουμε το αποτέλεσμα:

```

1 if res.success:
2     feasible_point = res.x
3     print("Feasible starting point found:", feasible_point)
4 else:
5     print("No feasible point exists.")

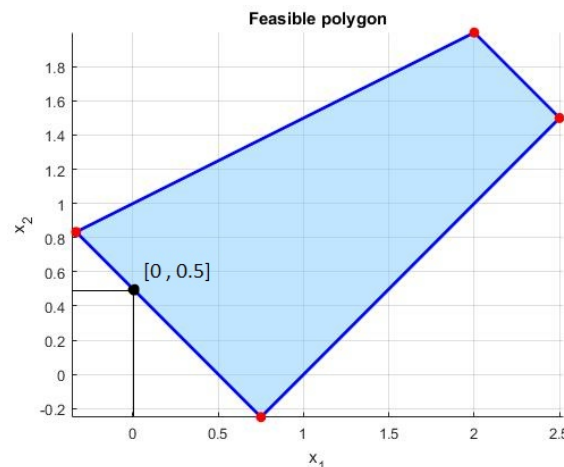
```

Listing 2.9: To print the result, we first check if there is one.

Το αποτέλεσμα είναι το παρακάτω:

Feasible starting point found: [0. 0.5]

Το σχήμα 2.21 δείχνει που βρίσκεται το σημείο που έχει επιστραφεί. Δεν πρόκειται για κάποια κορυφή του πολυτόπου αλλά αυτό δεν έχει σημασία. Το σημαντικό είναι πως ικανοποιεί όλες τις ανισότητες. Αυτό λοιπόν θα είναι και το σημείο αφετηρίας του περιπάτου.



Σχήμα 2.21: Το σημείο που επιστρέφει ο γραμμικός προγραμματισμός.

Αφού λύσαμε και το πρόβλημα της εύρεσης του σημείου εκκίνησης, μένει τώρα να συνδυάσουμε όλα τα παραπάνω ώστε να τρέξουμε την μέθοδο `ball walk`.

Ανακεφαλαιώνουμε:

1. Αρχικά ορίζουμε το πολύτοπό μας ως ένα σύνολο ανισοτήτων.
2. Στη συνέχεια, λύνουμε το πρόβλημα γραμμικού προγραμματισμού ώστε να βρούμε ένα σημείο που ανήκει στο πολύτοπο.
3. Με αφετηρία το σημείο αυτό, ξεκινούμε τον τυχαίο περίπατο.
4. Σχεδιάζουμε μια μπάλα με κέντρο το αρχικό σημείο και κάποια ακτίνα κατάλληλη για το πολύτοπο. Παίρνουμε ένα δείγμα με ομοιόμορφη κατανομή εντός της μπάλας και ελέγχουμε αν ικανοποιεί τις ανισότητες. Αν όχι παίρνουμε καινούργιο δείγμα, αν ναι το αποθηκεύουμε και σχεδιάζουμε μια νέα μπάλα με αυτό ως κέντρο και ίδια ακτίνα. Συνεχίζουμε κατά αυτόν τον τον τρόπο για όσα δείγματα επιλέξουμε.

Η υλοποίηση του παραπάνω αλγορίθμου συνδυάζει όλα τα τμήματα κώδικα που έχουμε δει έως τώρα. Αρχίζουμε ορίζοντας το παράδειγμά μας:

```

1 # -----
2 # 1. Polytope definition
3 # -----
4 A = np.array([
5     [1, 1],
6     [1, -1],
7     [-1, 2],
8     [-1, -1]
9 ])
10 b = np.array([4, 1, 2, -0.5])

```

Listing 2.10: The first step is to define our polytope.

Πρόκειται για το ίδιο παράδειγμα που χρησιμοποιήσαμε προηγουμένως για την επίλυση του γραμμικού προγραμματισμού. Η επιλογή βέβαια του πολυτόπου δεν επηρεάζει τον κώδικα στη συνέχεια, αρκεί αυτό να ορίζεται καλώς και να είναι πεπερασμένο. Επιπλέον, εδώ δουλεύουμε σε 2 διαστάσεις. Θα δούμε την γενίκευση σε περισσότερες διαστάσεις στην συνέχεια.

Συνεχίζουμε με γραμμικό προγραμματισμό, για την εύρεση ενός αρχικού σημείου που ανήκει στο πολύτοπο και από το οποίο θα ξεκινήσει ο τυχαίος περίπατος.

```

1 # -----
2 # 2. Find feasible starting point
3 # -----
4 c = np.array([0, 0])

```

```

5 res = linprog(c, A_ub=A, b_ub=b, bounds=[(None, None), (None,
    None)], method='highs')
6 if not res.success:
7     raise ValueError("No feasible starting point found!")
8 current = res.x

```

Listing 2.11: Use linear programming to find a starting point.

Ορίζουμε τις παραμέτρους του περιπάτου, δηλαδή τον αριθμό των βημάτων που επιθυμούμε, καθώς και την ακτίνα κάθε μπάλας ανάλογα με το σχήμα που δειγματοληπτούμε. Η ακτίνα δεν πρέπει να είναι ούτε πολύ μεγάλη, γιατί θα χάνουμε πολλά δείγματα τα οποία θα απορρίπτονται, αλλά ούτε και πολύ μικρή γιατί θα εξαντλήσουμε τους υπολογιστικούς πόρους χωρίς αντίκρισμα. Υπάρχει προσεγγιστικές τεχνικές ώστε να επιλέξει κανείς μια καλή ακτίνα, αλλά προς το παρόν θα την θέσουμε εμπειρικά.

```

1 # -----
2 # 3. Ball walk parameters
3 # -----
4 n_steps = 40000
5 radius = 0.2
6 samples = [current.copy()] # store only actual positions

```

Listing 2.12: Set random walk parameters.

Προχωρούμε τώρα στην καρδιά της μεθόδου, τον τυχαίο περίπατο. Χρησιμοποιούμε την συνάρτηση `sample-ball` που παρουσιάσαμε στο Listing 2.5.

```

1 # -----
2 # 4. Ball walk loop
3 # -----
4 n_accepted = 0
5
6 for i in range(1, n_steps):
7     step = sample_ball(d=2, r=radius)[0] # sample in current ball
8     proposal = current + step # find new coordinates
9
10    # check whether new point satisfy the inequalities
11    if np.all(A @ proposal <= b + 1e-12):
12        current = proposal # accept
13        n_accepted += 1 # count how many got accepted
14
15    samples.append(current.copy()) # add sample to list
16
17 samples = np.array(samples)

```

Listing 2.13: Execute the random walk.

Θα μετρήσουμε επιπλέον το πόσα σημεία γίνονται αποδεκτά, δηλαδή τον ρυθμό αποδοχής. Για τον σκοπό αυτό ορίζουμε μια μεταβλητή που να κρατά αυτό τον αριθμό. Στη συνέχεια ξεκινά ο περίπατος. Τρέχουμε μια επανάληψη σε όλο το εύρος των βημάτων που έχουμε επιλέξει. Ξεκινώντας από το αρχικό σημείο που βρήκαμε με γραμμικό προγραμματισμό, χρησιμοποιούμε την συνάρτηση `sample-ball` η οποία επιστρέφει ένα δείγμα ομοιόμορφης κατανομής πάνω σε μια σφαίρα ακτίνας r και με κέντρο το τρέχον δείγμα. Ελέγχουμε αν αυτό ικανοποιεί όλες τις ανισότητες και πράττουμε ανάλογα, με μια μικρή ανοχή σε ακραίες περιπτώσεις όπου το σημείο πέφτει κοντά στα άκρα του πολυτόπου. Δείτε εδώ την δύναμη του NumPy στην γραμμή 11. Με μια μόνο εντολή, δίνουμε την οδηγία να ελεγχθεί αν το παρόν σημείο ικανοποιεί όλες τις ανισότητες.

Κλείνουμε εκτυπώνοντας τον ρυθμό αποδοχής δειγμάτων. Σε δυο διαστάσεις μπορούμε να επιτύχουμε μεγάλες τιμές αποδοχής, αλλά αυτή πέφτει αρκετά όσο προχωρούμε σε περισσότερες διαστάσεις όπως θα γίνει φανερό στη συνέχεια.

```

1 # -----
2 # Acceptance rate
3 # -----
4 acceptance_rate = n_accepted / n_steps
5 print("Total steps:", n_steps)
6 print("Accepted moves:", n_accepted)
7 print("Acceptance rate:", acceptance_rate)
```

Listing 2.14: Print the results.

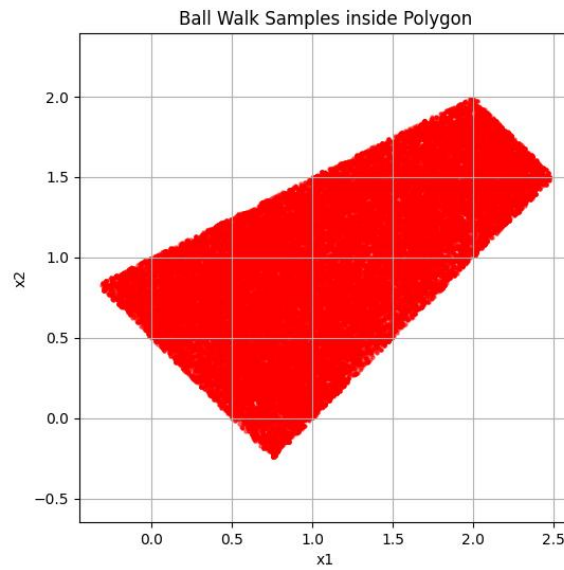
Η έξοδος του κώδικα είναι η εξής:

```

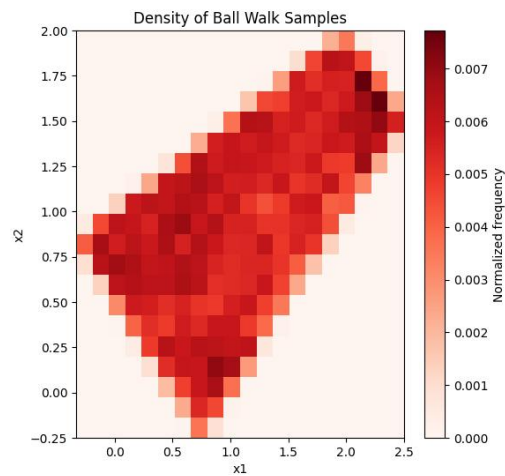
Total steps: 40000
Accepted moves: 35590
Acceptance rate: 0.88975
```

Αποοχή σχεδόν 90% είναι εξαιρετική. Ωστόσο, ποια είναι η κατανομή όλων αυτών των σημείων. Ας σχεδιάσουμε το γράφημα τοποθετώντας τα σημεία αυτό στο επίπεδο. Δείτε το σχήμα 2.22. Το σχήμα που προκύπτει θυμίζει έντονα το πολύτοπο του παραδείγματος. Το γεγονός αυτό δεν είναι τυχαίο βέβαια, καθώς βλέπουμε καθαρά την ομοιόμορφη κατανομή που ακολουθούν τα δείγματά μας εντός του πολυτόπου του παραδείγματος. Ως ένας επιπλέον έλεγχος της ομοιομορφίας, ας σχεδιάσουμε ένα θερμικό χάρτη, όπου χωρίζουμε το πολύτοπο σε μικρά τμήματα και βρίσκουμε τον λόγο δειγμάτων ανά επιφάνεια.

Μπορούμε να δοκιμάσουμε αρκετά ενδιαφέροντα πειράματα. Ας δοκιμάσουμε αρχικά να πειράζουμε την ακτίνα r της μπάλας, θέτοντας ακραίες τιμές σε αυτήν.



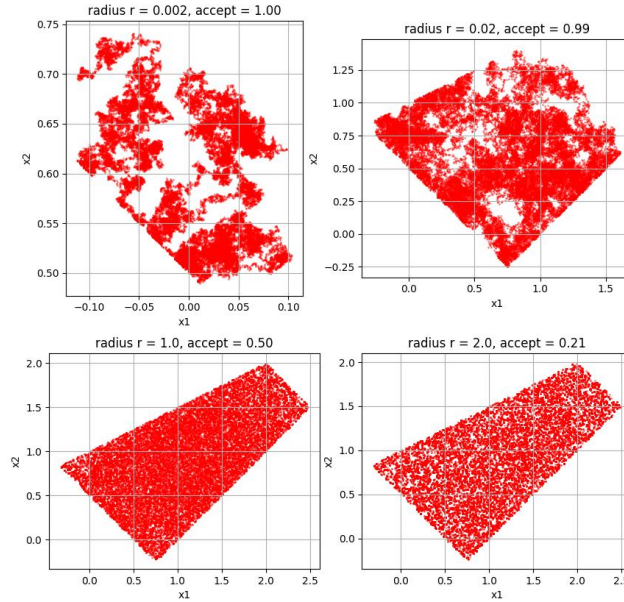
Σχήμα 2.22: Το διάγραμμα διασποράς των αποδεκτών σημείων της δειγματοληψίας μας με την μέθοδο ball walk.



Σχήμα 2.23: Ο θερμικός χάρτης των αποδεκτών δειγμάτων. Η θερμότητα είναι παρόμοια σε όλες τις περιοχές, με απειροελάχιστες διαφορές οι οποίες μειώνονται ακόμη περισσότερο με την αύξηση των βημάτων. Η κατανομή είναι σίγουρα ομοιόμορφη σε όλο το εμβαδόν του πολυτόπου.

Δοκιμάζουμε τις εξής ενδιαφέρουσες τιμές: $r = 0.002, r = 0.02, r = 1, r = 2$. Πρόκειται για τιμές τάξεις μεγέθους διαφορετικές από την ωραία τιμή που χρησι-

μπουήσαμε έως τώρα. Εφαρμόζοντας διαδοχικά τις τιμές αυτές και τρέχοντας την μέθοδο, σχεδιάζουμε το διάγραμμα διασποράς ώστε να οπτικοποιήσουμε την κατανομή των σημείων και τα παρουσιάζουμε ταυτόχρονα ώστε να γίνει η σύγκριση. Ταυτόχρονα, δίνουμε και τον ρυθμό αποδοχής. Δείτε το σχήμα 2.24.



Σχήμα 2.24: Στο παραπάνω γράφημα βλέπουμε την κατανομή των δειγμάτων για διαφορετικές ακτίνες της μπάλας, καθώς και τον ρυθμό αποδοχής για κάθε περίπτωση.

Συμβαίνουν τα εξής ενδιαφέροντα τα οποία αναμένουμε από την θεωρία. Για μικρές ακτίνες, ο περίπατος δεν έχει προλάβει να καλύψει το μεγαλύτερο μέρος του πολυτόπου. Στο γράφημα για την μικρή ακτίνα, μπορείτε μάλιστα να δείτε και τα τυχαία μονοπάτια που έχουν προκύψει καθώς και το πως αυτά διαδίδονται. Ο ρυθμός αποδοχής είναι βέβαια κοντά στο 100% αλλά αυτό δεν έχει σημασία. Έχουμε χρησιμοποιήσει την υπολογιστική μας ισχύ για να καλύψουμε μόλις μια γωνία του πολυτόπου. Αντιθέτως, στις μεγάλες ακτίνες, παρά τα πολλά βήματα, η πυκνότητα των σημείων ανά επιφάνεια κρατιέται σε χαμηλά επίπεδα. Οι μπάλες με τέτοιες ακτίνες υπερκαλύπτουν το πολύτοπο με συνέπεια τα περισσότερα δείγματα να βρίσκονται εκτός. Η πτώση του ρυθμού αποδοχής είναι δραματική.

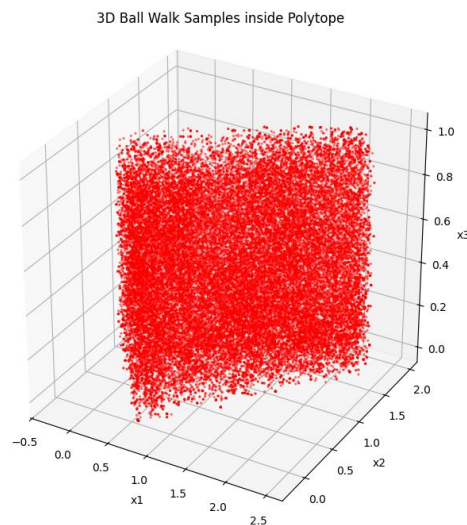
Για την συνέχεια, μπορούμε να ελέγξουμε την τρισδιάστατη περίπτωση. Μπορούμε να κρατήσουμε την ίδια τοπολογία και να προσθέσουμε απλά μια επιπλέον μεταβλητή ώστε να μας επεκτείνει στην 3η διάσταση. Για τον σκοπό αυτό, προσθέτουμε την ανισότητα $0 \leq x_3 \leq 1$, την οποία βέβαια πρέπει να δούμε ως δυο ανισότητες, τις $x_3 \leq 1$, $-x_3 \leq 0$. Συνεπώς, το πολύτοπό μας ορίζεται τώρα ως εξής:

```
1 A = np.array([
2     [1, 1, 0],
3     [1, -1, 0],
4     [-1, 2, 0],
5     [-1, -1, 0],
6     [0, 0, 1],
7     [0, 0, -1]
8 ])
9 b = np.array([4, 1, 2, -0.5, 1, 0])
```

Listing 2.15: We expand our example to the 3D case by adding an extra variable.

Δεν ξεχνάμε επίσης να αλλάξουμε την παράμετρο της συνάρτησης `sample-ball` σε 3, τιμή που δηλώνει τις νέες διαστάσεις. Η εφαρμογή της μεθόδου δίνει το σχήμα 2.25 και ρυθμό αποδοχής μικρότερο από την 2D περίπτωση.

Total steps: 40000
Accepted moves: 33435
Acceptance rate: 0.835875



Σχήμα 2.25: Η εφαρμογή της μεθόδου ball walk για τρεις διαστάσεις.

Το σχήμα του παραδείγματος έχει σχηματιστεί ξεκάθαρα, ενώ φαίνεται και η επέκταση στην 3η διάσταση δηλαδή από πάνω προς τα κάτω. Το σημαντικό είναι

πως η κατανομή παραμένει ομοιόμορφη και άρα η συνθήκη αυτή παραμένει σταθερή σε επιπλέον διαστάσεις, χωρίς να χρειάζεται οπτικός έλεγχος, κάτι που είναι βέβαια αδύνατο.

Συνεχίζουμε με 4, 6, 8, 10 διαστάσεις. Για κάθε επιπλέον διάσταση, προσθέτουμε μια ακόμη μεταβλητή όπως στην 3D περίπτωση, δηλαδή $0 \leq x_i \leq 1$. Ορίζουμε εκ νέου τις κατάλληλες ανισότητες, αλλάζουμε την παράμετρο του γραμμικού προγραμματισμού, την παράμετρο της συνάρτησης sample-ball, την παράμετρο bounds εντός του linprog και τρέχουμε τον κώδικα. Τα αποτελέσματα φαίνονται στην συνέχεια.

```
Dimensions: 4
Total steps: 40000
Accepted moves: 31466
Acceptance rate: 0.78665
```

```
Dimensions: 6
Total steps: 40000
Accepted moves: 29235
Acceptance rate: 0.730875
```

```
Dimensions: 8
Total steps: 40000
Accepted moves: 26850
Acceptance rate: 0.67125
```

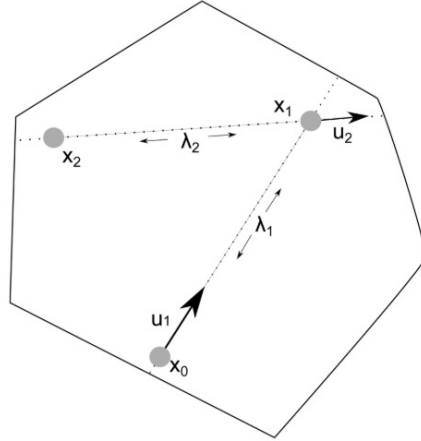
```
Dimensions: 10
Total steps: 40000
Accepted moves: 24570
Acceptance rate: 0.61425
```

Γίνεται εμφανής η μείωση του ρυθμού αποδοχής καθώς αυξάνονται οι διαστάσεις. Αυτό συμβαίνει γιατί σε περισσότερες διαστάσεις, η επιφάνεια του πολυτόπου σε σχέση με τον όγκο αυξάνει. Συνεπώς, βρισκόμαστε για περισσότερα βήματα κοντά στα εξωτερικά υπερεπίπεδα που ορίζουν το πολύτοπο και άρα περισσότερα βήματα πέφτουν εκτός του πολυτόπου. Μια καλή στρατηγική για να μειώσουμε τον αντίκτυπο των πολλών διαστάσεων είναι η αναπροσαρμογή της ακτίνας της μπάλας ανάλογα με τον αριθμό των διαστάσεων $r \sim 1/\sqrt{d}$. Είδαμε στο προηγούμενο πείραμα πως η μείωση της ακτίνας αυξάνει τον ρυθμό αποδοχής αλλά αργεί να καλύψει το πολύτοπο. Η παραπάνω αναλογία λοιπόν, θεωρείται η χρυσή τομή για την τιμή της ακτίνας.

2.4.6 Hit and Run

Η μέθοδος ball walk δεν είναι η μόνη διαθέσιμη μέθοδος για δειγματοληψία εντός ενός πολυτόπου. Μάλιστα, σε πολλές διαστάσεις η απόδοσή της μειώνεται όπως είχαμε την ευκαρία να διαπιστώσουμε στην προηγούμενη ενότητα. Μια μέθοδος που έρχεται να λύσει αυτό το πρόβλημα είναι η μέθοδος hit and run. Πρόκειται φυσικά για μια MCMC μέθοδο, όπου κάθε βήμα εξαρτάται από το προηγούμενο του και μόνο και έτσι σχηματίζει μια αλυσίδα Markov. Επιστρέφει δείγματα εντός ενός δεδομένου πολυτόπου σε ομοιόμορφη κατανομή εντός του όγκου του.

Η μέθοδος λειτουργεί σε πολλά σημεία παρόμοια με την ball walk: Έστω πολύτοπο $P = \{x \in R^d | Ax \leq b\}$ και ένα αρχικό του σημείο x_0 . θεωρούμε μια τυχαία κατεύθυνση (με δειγματοληψία από μια μοναδιαία σφαίρα) $u \in S^{d-1}$ και σχεδιάζουμε μια ευθεία που περνά από το παρόν σημείο $l_k(t) = x_k + tu, t \in R$. Βρίσκουμε τα σημεία τομής της ευθείας με τα σύνορα του πολυτόπου $l_k(t) \cap P$. Από το ευθύγραμμο τμήμα που ορίζουν τα σημεία αυτά, λαμβάνουμε ένα τυχαίο δείγμα με ομοιόμορφη κατανομή στο διάστημα $[l_{\min}, l_{\max}]$. Το δείγμα αυτό είναι τώρα το νέο μας σημείο $x_{k+1} = x_k + tu$, από το οποίο συνεχίζεται η διαδικασία με τον ίδιο τρόπο. Το επόμενο σχήμα δίνει μια οπτική της μεθόδου.



Σχήμα 2.26: Η μέθοδος hit-and-run. Τα νέα σημεία προκύπτουν από την ομοιόμορφη δειγματοληψία πάνω σε ένα τυχαίο ευθύγραμμο τμήμα που περνά από το τωρινό σημείο.

Στο τέλος της διαδικασίας, η κατανομή προκύπτει ομοιόμορφη εντός του πολυτόπου. Η απόδειξη είναι παρόμοια με την περίπτωση της ball walk και δεν μας απασχολεί εδώ. Σε αντίθεση με την ball walk όπου κάθε βήμα είναι κοτά στο προηγούμενο και ο αλγόριθμος εγκλωβίζεται για μικρές ακτίνες, η hit-and-run μπορεί να πραγματοποιεί τεράστια άλματα σε κάθε βήμα, από το ένα άκρο στο άλλο, με συνέπεια ακόμη και για λίγα βήματα η απόδοση να είναι ικανοποιητική και επιπλέον,

να καλύπτουμε περισσότερο από τον όγκο του πολυτόπου.

2.4.7 Εφαρμογή της Hit and Run

Περνάμε ευθύς αμέσως στην εφαρμογή της μεθόδου. Εκμεταλλευόμαστε στο έπακρο τον κώδικα για την μέθοδο ball walk κατά την οποία έχουμε αναπτύξει πολλές από τις τεχνικές που χρειαζόμαστε ξανά, όπως την δειγματοληψία πάνω σε υπερσφαίρα και τον γραμμικό προγραμματισμό για την εύρεση ενός αρχικού σημείου. Θα χρησιμοποιήσουμε τον ίδιο ακριβώς κώδικα για τις εργασίες αυτές.

Μια συνθήκη με την οποία δεν έχουμε ασχοληθεί είναι η εύρεση των σημείων τομής μιας ευθείας με τα υπερεπίπεδα που αποτελούν τα όρια του πολυτόπου. Ας μελετήσουμε τις περιπτώσεις αυτές με ένα παράδειγμα σε 2 διαστάσεις. Θεωρήστε το αρχικό σημείο $(0, 0.5)$ που δίνει ο γραμμικός προγραμματισμός για το πολύτοπο που χρησιμοποιούμε ως παράδειγμα. Βρίσκουμε τώρα μια τυχαία κατεύθυνση στο χώρο. Δειγματοληπτούμε στον μοναδιαίο κύκλο και έστω επιστρέφει το σημείο $(0.5, 0.5)$. Το σημείο αυτό, αν το δούμε ως διάνυσμα έχει ένα μέτρο και μια διεύθυνση. Επειδή εμάς μας ενδιαφέρει μόνο η διεύθυνση, κανονικοποιούμε το διάνυσμα αυτό:

$$u = \frac{z}{\|z\|} = \frac{(0.5, 0.5)}{\sqrt{0.5^2 + 0.5^2}} = (0.707, 0.707)$$

Μπορούμε τώρα να σχηματίσουμε την παραμετρική εξίσωση της ευθείας που περνά από το αρχικό σημείο ως εξής:

$$x(t) = x_0 + tu = \begin{bmatrix} 0 \\ 0.5 \end{bmatrix} + t \begin{bmatrix} 0.707 \\ 0.707 \end{bmatrix} = \begin{bmatrix} 0.707t \\ 0.5 + 0.707t \end{bmatrix}.$$

το οποίο αναλυτικά γίνεται:

$$\begin{cases} x_1(t) = 0 + 0.707t = 0.707t \\ x_2(t) = 0.5 + 0.707t \end{cases}$$

Εφαρμόζουμε τις παραπάνω σχέσεις στο σύνολο των ανισοτήτων που περιγράφουν το πολύτοπο:

$$\begin{cases} x_1 + x_2 \leq 4 & \Rightarrow 0.707t + (0.5 + 0.707t) \leq 4 \Rightarrow t \leq 2.474 \\ x_1 - x_2 \leq 1 & \Rightarrow 0.707t - (0.5 + 0.707t) \leq 1 \Rightarrow \text{ισχύει πάντα} \\ -x_1 + 2x_2 \leq 2 & \Rightarrow -0.707t + 2(0.5 + 0.707t) \leq 2 \Rightarrow t \leq 1.414 \\ -x_1 - x_2 \leq -0.5 & \Rightarrow -0.707t - (0.5 + 0.707t) \leq -0.5 \Rightarrow t \geq 0 \end{cases}$$

$$\Rightarrow t \in [0, 1.414]$$

Δειγματοληπτούμε ομοιόμορφα στο διάστημα 0 έως 1.414. Έστω τυχαίνει $t = 0.9$. Αντικαθιστούμε και έχουμε:

$$x_{\text{new}} = x_0 + t u = (0, 0.5) + 0.9 \cdot (0.707, 0.707) = (0.636, 1.136)$$

Ο τρόπος με τον οποίο έχουμε καταλήξει στο νέο σημείο, μας εγγυάται πως βρίσκεται εντός του πολυτόπου. Αυτό πορκύπτει από το γεγονός πως έχουμε διαλέξει συγκεκριμένα τα όρια της παραμέτρου t έτσι ώστε να ικανοποιούνται όλες οι ανισότητες που ορίζουν το πολύτοπο. Συνεπώς, αντίθετα με την μέθοδο ball walk, όλα τα δείγματα της hit-and-run γίνονται αποδεκτά. Η μόνη ασάφεια που υπάρχει είναι στο πόσα δείγματα χρειαζόμαστε ώστε να καλύψουμε ικανοποιητικά το πολύτοπο.

Κατά την διάρκεια του αλγορίθμου τα πράγματα είναι λίγο διαφορετικά καθώς δεν γνωρίζουμε εκ των προτέρων τις ακριβές τιμές που προκύπτουν για τα δείγματα. Ας δούμε την γενική περίπτωση. Έστω μια εκ των ανισοτήτων:

$$a^\top x \leq b,$$

και μια ευθεία από το σημείο x_0 στην κατεύθυνση u :

$$x(t) = x_0 + t u, \quad t \in \mathbb{R}.$$

Αντικαθιστούμε την ευθεία στην ανισότητα:

$$a^\top (x_0 + t u) \leq b \quad \Rightarrow \quad (a^\top u) t \leq b - a^\top x_0.$$

Πρέπει τώρα να λύσουμε ως προς t . Το πρόβλημα εδώ είναι πως δεν ξέρουμε την φορά της ανισότητας. Αυτή εξαρτάται από το πρόσημο του συντελεστή μπροστά από το t . Για τον λόγο αυτό, όταν κωδικοποιούμε τον αλγόριθμο αυτό, πρέπει πρώτα να υπολογίσουμε τον συντελεστή αυτό και να διακρίνουμε περιπτώσεις ανάλογα με το πρόσημό του. Υπάρχουν 3 περιπτώσεις:

Θετικός συντελεστής — $a^\top u > 0$:

$$t \leq \frac{b - a^\top x_0}{a^\top u}, \quad \text{επιτρεπόμενα όρια: } t \in (-\infty, (b - a^\top x_0)/(a^\top u)].$$

Αρνητικός συντελεστής — $a^\top u < 0$:

$$t \geq \frac{b - a^\top x_0}{a^\top u}, \quad \text{επιτρεπόμενα όρια: } t \in [(b - a^\top x_0)/(a^\top u), +\infty).$$

Μηδενικός συντελεστής — $a^\top u = 0$:

$$0 \leq b - a^\top x_0 \quad (\text{κανένας περιορισμός στο } t), \quad 0 > b - a^\top x_0 \quad (\text{δεν υπάρχουν σημεία τομής}).$$

Φροντίζουμε λοιπόν, για κάθε ανισότητα να αναβαθμίζουμε σωστά τα όρια του t . Για τον σκοπό αυτό πρέπει να υπολογιστεί το εσωτερικό γινόμενο $a^\top u$ πριν από την όποια απόφασή μας.

Ανακεφαλαιώνουμε τα βήματα του αλγορίθμου:

1. Αρχικά ορίζουμε το πολύτοπό μας ως ένα σύνολο ανισοτήτων.
2. Στη συνέχεια, λύνουμε το πρόβλημα γραμμικού προγραμματισμού ώστε να βρούμε ένα σημείο που ανήκει στο πολύτοπο. Με αφετηρία το σημείο αυτό, ξεκινούμε τον τυχαίο περίπατο με την μέθοδο hit-and-run.
3. Επιλέγουμε μια τυχαία κατεύθυνση στο χώρο. Αυτό γίνεται με την ομοιόμορφη δειγματοληψία σε μια μοναδιαία σφαίρα. Το διάνυσμα κέντρο-δείγμα θα δώσει την ζητούμενη τυχαία κατεύθυνση.
4. Σχεδιάζουμε την ευθεία που περνά από το τωρινό σημείο και βρίσκουμε τα σημεία τομής με το πολύτοπο.
5. Δειγματοληπτούμε με την ομοιόμορφη κατανομή σε όλο το μήκος του ευθύγραμμου τμήματος που βρίσκεται εντός του πολυτόπου. Το νέο αυτό σημείο μας οδηγεί στο επόμενο βήμα.

Τα βήματα 1, 2 και 3 λύνονται με τον κώδικα που χρησιμοποιήσαμε στην μέθοδο ball walk. Δείτε τα Listing 2.5, Listing 2.10 και Listing 2.11. Ας δούμε τώρα την ουσία της μεθόδου, τμηματικά:

```

1 # -----
2 # Hit-and-Run loop
3 # -----
4 n_steps = 40000 # number of steps
5 samples = [current.copy()] # current is the starting point
6
7 for i in range(1, n_steps): # loop n_steps times
8     # Random direction on unit circle
9     direction = sample_ball(d=2, r=1.0, n=1)[0] # See listing 2.5
10    for sample_ball
11    direction /= np.linalg.norm(direction) # Normalise the
12    vector, keep only the direction

```

Listing 2.16: We begin the hit-and-run loop by defining the number of steps and the starting point that was calculated with linear programming. Loop over step count and start each loop by finding a random direction.

Ξεκινάμε με το αρχικό σημείο, αποτέλεσμα του γραμμικού προγραμματισμού και τον αριθμό των βημάτων. Πρώτη δουλειά σε κάθε βήμα (δηλαδή σε κάθε επανάληψη) η εύρεση μιας τυχαίας διεύθυνσης. Χρησιμοποιούμε την γνωστή συνάρτηση sample-ball από το Listing 2.5 η οποία επιστρέφει ένα τυχαίο ομοιόμορφο δείγμα σε μια σφαίρα συγκεκριμένων διαστάσεων και ακτίνας, εδώ μοναδιαίας. Κανονικοποιούμε το δείγμα αυτό και έτσι προκύπτει η τυχαία διεύθυνση. Συνεχίζουμε:

```

1 # Find intersection of line {current + t*direction} with Ax
2 # <= b
3 # For each constraint: a_i · (current + t u) <= b_i
4 # => (a_i · u) t <= b_i - a_i · current
5
6 t_min, t_max = -np.inf, np.inf # start the bounds to infinity
7
8 for a_i, b_i in zip(A, b): #zip is a nice function that
9    pairs each row of A with its row in b.
10    #Thus, this loop iterates over each
11    inequality.
12
13    a_dot_u = np.dot(a_i, direction) # Find the product a^T*u.
14    rhs = b_i - np.dot(a_i, current) # Find the right hand
15    side (rhs)
16
17 # As explained in the text, we consider cases.
18 if abs(a_dot_u) < 1e-12:
19     # This is the zero case, direction is almost parallel
20     to this constraint
21     if rhs < 0:

```

```

17         # Entire line is infeasible
18         t_min, t_max = 0, -1 # empty interval
19         break
20     else:
21         continue # no bound from this constraint
22
23     bound = rhs / a_dot_u # The result of the inequality
24
25     if a_dot_u > 0:
26         # t <= bound
27         t_max = min(t_max, bound) # Update max bound
28     else:
29         # t >= bound
30         t_min = max(t_min, bound) # Update min bounds

```

Listing 2.17: This is the part of the code that holds the main logic of the hit-and-run method. We need to find the bounds of parameter t that will constrain the line inside the polytope. We begin with the infinite line. For each inequality, calculate the coefficients and based on their sign update the bounds accordingly.

Τα σχόλια εξηγούν σχολαστικά κάθε εντολή. Συνοπτικά, αρχίζουμε με την ευθεία να κινείται ως το άπειρο. Υπολογίζουμε τους συντελεστές που εξηγήσαμε αναλυτικά στο παράδειγμα, δηλαδή τις ποσότητες που προκύπτουν από την αντικατάσταση της ευθείας σε κάθε μια εκ των ανισοτήτων. Προσέξτε πως υπάρχει μια δεύτερη επανάληψη που τρέχει σε όλες τις ανισότητες. Για κάθε ανισότητα, ανάλογα με το πρόσημο των συντελεστών αναβαθμίζουμε το κατάλληλο όριο για την παράμετρο t έως ότου τελικά καταλήξουν στα σωστά όρια που περιορίζουν την ευθεία εντός του πολυτόπου. Η συνέχεια τώρα είναι πολύ απλή:

```

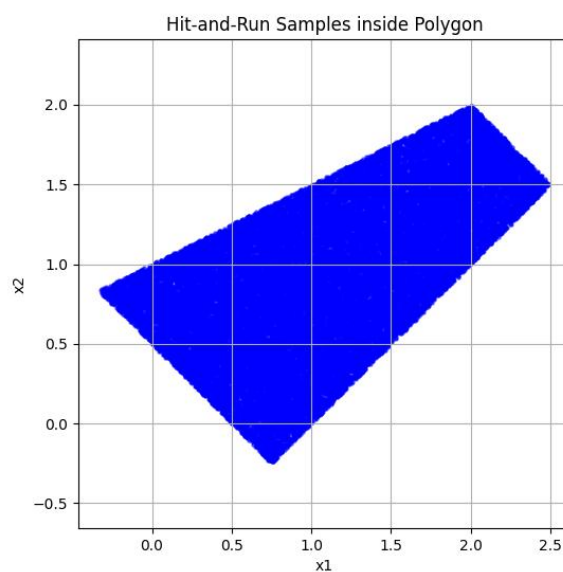
1     # Sample t uniformly in [t_min, t_max]
2     t = np.random.uniform(t_min, t_max)
3
4     # Move to new point
5     current = current + t * direction
6     samples.append(current.copy())
7
8 samples = np.array(samples)
9
10 # -----
11 # Plotting
12 # -----
13 plt.figure(figsize=(6,6))
14 plt.scatter(samples[:,0], samples[:,1], s=5, alpha=0.5,
15             color='blue')
16 plt.xlabel('x1')
17 plt.ylabel('x2')
18 plt.title('Hit-and-Run Samples inside Polygon')
19 plt.grid(True)
20 plt.axis('equal')

```

```
plt.show()
```

Listing 2.18: Sample uniformly inside the calculated bounds. This is the new step of the random walk.

Δειγματοληπτούμε ομοιόμορφα εντός των υπολογισμένων ορίων. Το `numpy` εκτελεί αυτόματα την εργασία αυτή. Το τυχαίο σημείο που προκύπτει αποτελεί τον επόμενο κρόκο στην αλυσίδα Markov του τυχαίου περιπάτου της μεθόδου hit-and-run. Οι υπόλοιπες εντολές σχεδιάζουν το γράφημα των δειγμάτων στον 2D χώρο. Το εν λόγω γράφημα είναι το παρακάτω.



Σχήμα 2.27: Το γράφημα διασποράς των δειγμάτων της μεθόδου hit-and-run. Δείτε την τέλεια ομοιόμορφη κάλυψη του πολυτόπου του παραδείγματος.

Η μέθοδος hit-and-run έχει λοιπόν κάποια πλεονεκτήματα σε σχέση με την ball walk. Παρατηρήστε πως δεν έχουμε ασχοληθεί καθόλου με την επιλογή ακτίνας ή με ρυθμούς αποδοχής. Όλα τα δείγματα που επιστρέφει η μέθοδος ανήκουν εκ κατασκευής εντός του πολυτόπου. Κατά συνέπεια, η μέθοδος θεωρείται πιο αποδοτική από την ball walk, ειδικά σε περιπτώσεις πολλών διαστάσεων. Δεν έχουμε αναφερθεί βέβαια ακόμα στο mixing time των μεθόδων. Αυτό θα γίνει συγκεντρωτικά σε επόμενη ενότητα.

Σε πολλές διαστάσεις ο κώδικας είναι σχεδόν ίδιος. Ο τρόπος με τον οποίο έχουμε ορίσει τις διάφορες επαναλήψεις και πράξεις είναι ανεξάρτητος από το πόσες διαστάσεις και πόσες ανισότητες υπάρχουν. Έστω 6 διαστάσεις. Προσθέτουμε, 4

επιπλέον μεταβλητές οι οποίες κινούνται από το 0 ως το 1. Επεκτείνουμε έτσι το σχήμα του παραδείγματος σε επιπλέον διαστάσεις. Το πολύτοπο τώρα, ορίζεται ως εξής (θυμηθείτε πως το $0 \leq x_i \leq 1$ πρέπει να το δούμε ως $x_i \leq 1$ και $-x_i \leq 0$ και έτσι κάθε διάσταση προσθέτει δυο ανισότητες):

```

1 A = np.array([
2     [1, 1, 0, 0, 0, 0],
3     [1, -1, 0, 0, 0, 0],
4     [-1, 2, 0, 0, 0, 0],
5     [-1, -1, 0, 0, 0, 0],
6     [0, 0, 1, 0, 0, 0],
7     [0, 0, -1, 0, 0, 0],
8     [0, 0, 0, 1, 0, 0],
9     [0, 0, 0, -1, 0, 0],
10    [0, 0, 0, 0, 1, 0],
11    [0, 0, 0, 0, -1, 0],
12    [0, 0, 0, 0, 0, 1],
13    [0, 0, 0, 0, 0, -1]
14 ])
15 b = np.array([4, 1, 2, -0.5, 1, 0, 1, 0, 1, 0, 1, 0])

```

Listing 2.19: This is a definition of a 6D polytope based on our 2D example.

Οι υπόλοιπες αλλαγές εντοπίζονται στην αλλαγή της παραμέτρου της sample-ball σε 6 και την αλλαγή των παραμέτρων του γραμμικού προγραμματισμού ώστε να ανταποκρίνονται σε 6 διαστάσεις:

```

1 c = np.array([0, 0, 0, 0, 0, 0])
2 res = linprog(c, A_ub=A, b_ub=b, bounds=[(None, None)]*6,
3               method='highs')

```

Listing 2.20: Changes to linear programming to accomodate 6D.

Όλα τα υπόλοιπα παραμένουν ως έχουν. Τρέχοντας την μέθοδο για 6 διαστάσεις, μπορούμε να τυπώσουμε κάποια από τα δείγματα:

Last 5 samples:

```

[1.91233743 1.66151881 0.49191862 0.6369366 0.5939465 0.89776445]
[2.26715258 1.55695573 0.4193729 0.38261959 0.28192524 0.67343816]
[2.05618278 1.89110734 0.10869821 0.20156858 0.1803433 0.66443523]
[1.93479388 1.94434458 0.15205338 0.38542244 0.27939236 0.76030397]
[2.04975209 1.93664461 0.22759441 0.2796369 0.51125325 0.53955223]

```

2.4.8 Dikin Walk

Μια εναλλακτική MCMC μέθοδος αποτελεί η Dikin Walk. Η μέθοδος αυτή χρησιμοποιείται για την δειγματοληψία σε κυρτά πολύγωνα, πετυχαίνοντας την ομοιόμορφη κατανομή στο σύνολο του όγκου τους. Η βασική διαφορά με τις ball walk και hit-and-run είναι πως η μέθοδος εκμεταλλεύεται την τοπική γεωμετρία γύρω από το παρόν βήμα, δειγματοληπώντας σε ένα ελλειψοειδές το οποίο υπολογίζεται σε σχέση με το που βρισκόμαστε στο πολύτοπο την τωρινή στιγμή. Αυτό έχει ως αποτέλεσμα να πετυχαίνει καλύτερο mixing time σε σχέση με τις προηγούμενες μεθόδους.

Έστω το πολύτοπο $P = \{x \in R^d : a_i^T x \leq b_i, i = 1, \dots, m\}$ που δίνεται ως ένα σύνολο ανισοτήτων. Κάθε ανισότητα μπορούμε να την δούμε ως εξής:

$$a_i^T x \leq b_i \Rightarrow b_i - a_i^T x \geq 0$$

Ας ονομάσουμε $s_i(x) = b_i - a_i^T x$, με συνέπεια τα σημεία εντός του πολυτόπου να ικανοποιούν τις σχέσεις $s_i(x) \geq 0$. Η λογική είναι τώρα να ορίσουμε μια ποινή για σημεία που βρίσκονται κοντά στα όρια του πολυτόπου. Αυτό μπορεί να γίνει με την συνάρτηση του λογαρίθμου, η οποία σκάει όταν πλησιάζουμε το μηδέν. Θεωρούμε λοιπόν τις ποσότητες:

$$-\log(s_i(x))$$

Το πρόσημο είναι αρνητικό ώστε να απειρίζεται θετικά. Όσο το x , δηλαδή το σημείο, πλησιάζει τα σύνορα του πολυτόπου, μια τουλάχιστον από τις ποσότητες $s_i(x)$ τείνει στο μηδέν, αυτή που αντιστοιχεί στο σύνορο που προσεγγίζει το σημείο μας. Συνεπώς, ο λογάριθμος τείνει στο άπειρο. Αντιθέτως, μακριά από τα σύνορα η ποινή αυτή είναι μικρή. Ορίζουμε λοιπόν την συνάρτηση φράγματος ως το άθροισμα όλων των λογαρίθμων για όλες τις ποσότητες $s_i(x)$:

$$\varphi(x) = \sum_{i=1}^m -\log(s_i(x))$$

Η συνάρτηση αυτή μας αποκαλύπτει το αν βρισκόμαστε κοντά σε κάποιο σύνορο, αλλά δεν μας λέει τίποτα για την κατεύθυνση που είναι επικίνδυνη. Πρόκειται για ένα βαθμωτό μέγεθος. Για τον σκοπό αυτό, παίρνουμε τον Εσσιανό πίνακα (Hessian) της παραπάνω συνάρτησης.

Θυμίζουμε τον ορισμό της Εσσιανής. Για μια βαθμωτή συνάρτηση $\varphi : R^d \rightarrow R$ έχουμε την κλίση της συνάρτησης:

$$\nabla \phi(x) = \begin{bmatrix} \frac{\partial \phi}{\partial x_1} \\ \vdots \\ \frac{\partial \phi}{\partial x_d} \end{bmatrix}.$$

και την Εσσιανή που στην ουσία είναι η δεύτερη παράγωγος:

$$\nabla^2 \phi(x) = \begin{bmatrix} \frac{\partial^2 \phi}{\partial x_1^2} & \cdots & \frac{\partial^2 \phi}{\partial x_1 \partial x_d} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 \phi}{\partial x_d \partial x_1} & \cdots & \frac{\partial^2 \phi}{\partial x_d^2} \end{bmatrix}.$$

Το ίχνος του Εσσιανού πίνακα δίνει την γνωστή λαπλασιανή της συνάρτησης:

$$\Delta \phi(x) = \text{Tr}(\nabla^2 \phi(x)) = \sum_{j=1}^d \frac{\partial^2 \phi}{\partial x_j^2}.$$

Ο Εσσιανός πίνακας δείχνει την μεταβολή της κυρτότητας της συνάρτησης στις διάφορες διευθύνσεις. Έτσι, κοντά στο σύνορο μιας εκ των ανισοτήτων η κυρτότητα εκρύνγεται μόνο σε κατευθύνσεις που βλέπουν προς την επιφάνεια που ορίζει το σύνορο. Συνεπώς, η δειγματοληψία δεν πρέπει να προχωρήσει πολύ προς τα εκεί γιατί κινδυνεύουμε να βγούμε εκτός του πολυτόπου.

Χρησιμοποιούμε τώρα, την Εσσιανή για να ορίσουμε το σχήμα από το οποίο δειγματοληπτούμε σε κάθε βήμα. Πρόκειται για το ελλειψοειδές Dikin, το οποίο ορίζεται ως:

$$E(x) = \left\{ y \in R^d : (y - x)^T \nabla^2 \varphi(x) (y - x) \leq r^2 \right\}$$

Μπορούμε να σκεφτούμε πως η Εσσιανή ορίζει μια μετρική στον χώρο του πολυτόπου. Μια μετρική ορίζει τον τρόπο με τον οποίο μετράμε τις αποστάσεις στην

γεωμετρία μας. Η μετρική της Εσσιανής μας λέει πως συμπεριφέρεται η κυρτότητα εντός του πολυτόπου. Κοντά στα σύνορα είναι έντονη, ενώ στο εσωτερικό μικρότερη. Η ποσότητα $(y - x)$ είναι απλώς η απόσταση ενός υποψηφίου σημείου του ελλειψοειδούς από το κέντρο, που είναι το παρών δείγμα. Το σύνολο των όρων αυτών ορίζουν μια νόρμα:

$$(y - x)^T \nabla^2 \varphi(x) (y - x) = \|y - x\|_x^2$$

Ο δείκτης x μας λέει πως έχουμε την τοπική νόρμα στο σημείο x . Η παραπάνω σχέση είναι όμοια με την γνωστή σχέση μας για την εξίσωση ενός κύκλου. Η νόρμα είναι τότε:

$$\|y - x\|_2^2 = (y - x)^T (y - x) = \sum_{i=1}^d (y_i - x_i)^2$$

ενώ η εξίσωση της σφαίρας είναι:

$$\|y - x\|_2^2 \leq r^2$$

Κατά αυτόν τον τρόπο, η σχέση:

$$\|y - x\|_x^2 \leq r^2$$

ορίζει ένα ελλειψοειδές, ανάλογα με την τοπική κυρτότητα στο δείγμα. Το ελλειψοειδές θα είναι συμπιεσμένο κοντά στα σύνορα και εκτεταμένο κοντά στο κέντρο του πολυτόπου. Έτσι, σε κάθε νέο βήμα ορίζουμε το ελλειψοειδές και λαμβάνουμε ένα δείγμα εντός του. Το δείγμα αυτό αποτελεί το νέο κέντρο για το επόμενο ελλειψοειδές, με τον ίδιο τρόπο που στην ball walk κάθε νέο δείγμα αποτελεί το κέντρο της αντίστοιχης μπάλας.

Μάλιστα, δεν χρειάζεται να πάρουμε δείγμα απ'ευθείας εντός της κατανομής του ελλειψοειδούς, αλλά μπορούμε να πάρουμε δείγμα από μια μοναδιαία μπάλα το οποίο στη συνέχεια να μετατρέψουμε στην κατανομή του ελλειψοειδούς. Θα δούμε την μετατροπή αυτή στην επόμενη ενότητα. Πρέπει εδώ να σημειωθεί πως αν και το νέο δείγμα έχει μικρή πιθανότητα να βρεθεί εκτός του πολυτόπου, εντούτοις η πιθανότητα αυτή είναι υπαρκτή. Για τον σκοπό αυτό συχνά πραγματοποιούμε ένα βήμα αποδοχής Metropolis–Hastings.

2.4.9 Εφαρμογή της Dikin Walk

Θα προσπαθήσουμε εδώ να εφαρμόσουμε την Dikin Walk βήμα βήμα ώστε να λάβουμε δείγματα εντός του γνωστού πολυτόπου που χρησιμοποιούμε έως τώρα στις δυο διαστάσεις. Πολλά από τα στοιχεία του κώδικα είναι τα ίδια που έχουμε δει έως τώρα, όπως ο γραμμικός προγραμματισμός και η συνάρτηση `sample-ball`, οπότε δεν θα επαναλάβουμε τα στοιχεία αυτά.

Πριν προχωρήσουμε, ας θυμηθούμε τα βήματα της μεθόδου:

1. Ορίζουμε το πολύτοπο ως ένα σύνολο ανισοτήτων.
2. Χρησιμοποιούμε γραμμικό προγραμματισμό ώστε να βρούμε ένα αρχικό σημείο εντός του πολυτόπου.
3. Υπολογίζουμε την συνάρτηση φράγματος $\varphi(x)$ που αντιστοιχεί στο πολύτοπό μας.
4. Υπολογίζουμε τον Εσσιανό πίνακα της συνάρτησης φράγματος. Δεν χρειάζεται να παραγωγίσουμε την συνάρτηση φράγματος στον κώδικα. Αν εκτελέσουμε τους θεωρητικούς υπολογισμούς, η Εσσιανή δίνεται από την σχέση:

$$H(x) = \sum_{i=1}^m \frac{a_i a_i^T}{(b_i - a_i^T x)^2}$$

δηλαδή συναρτήσει μόνο των σχέσεων ορισμού του πολυτόπου.

5. Υπολογίζουμε το $H(x)^{-1/2}$, το οποίο μας δίνει τον μετασχηματισμό από την κατανομή της μοναδιαίας μπάλας στην κατανομή του ελλειψοειδούς: $y = x + H(x)^{-1/2}u$ με u ομοιόμορφη κατανομή εντός μιας μπάλας ακτίνας r .
6. Λαμβάνουμε ένα δείγμα στο ελλειψοειδές και ελέγχουμε αν ανήκει εντός του πολυτόπου. Αν ναι, το αποδεχόμαστε και συνεχίζουμε την μέθοδο με κέντρο το νέο αυτό δείγμα.

Κατά αυτόν τον τρόπο, αφού ορίσουμε το γνωστό πολύτοπο των δυο διαστάσεων που μπορείτε να δείτε στο Listing 2.10, βρίσκουμε ένα αρχικό σημείο εντός του πολυτόπου μέσω του γραμμικού προγραμματισμού Listing 2.11. Προκύπτει η ανάγκη για μια μικρή αλλαγή στον γραμμικό προγραμματισμό. Επειδή το σημείο που επιστρέφει βρίσκεται στο σύνορο του πολυτόπου, κάτι το οποίο αποτελεί ακραία περίπτωση για την μέθοδο Dikin walk, τροποποιούμε τον κώδικα ώστε το

σημείο να βρίσκεται όσο το δυνατόν εσωτερικά του πολυτόπου. Αυτό γίνεται εισάγοντας μια παράμετρο δ για κάθε ανισότητα την οποία συνέχεια ζητούμε να μεγιστοποιήσουμε. Οι λεπτομέρειες δεν μας ενδιαφέρουν εδώ, αλλά το σημαντικό είναι πως το αρχικό σημείο θα αλλάξει από το $[0, 0.5]$ που βλέπουμε στο σχήμα 2.21 στο σημείο $[0.75, 0.833]$ το οποίο θα δούμε στην συνέχεια.

Ας δούμε τον χειροκίνητο υπολογισμό της συνάρτησης φράγματος και του ελλειψοειδούς για το αρχικό σημείο, κάτι το οποίο θα μας βοηθήσει στην συνέχεια στον κώδικα. Αρχίζουμε με την συνάρτηση φράγματος:

$$\phi(x) = - \sum_{i=1}^4 \log(b_i - a_i^\top x)$$

Υπολογίζουμε τις ποσότητες $s_i(x)$ για κάθε ανισότητα:

$$\begin{aligned} s_1 &= 4 - (1 \cdot 0.75 + 1 \cdot 0.8333) = 2.4167 \\ s_2 &= 1 - (1 \cdot 0.75 + (-1) \cdot 0.8333) = 1.0833 \\ s_3 &= 2 - (-1 \cdot 0.75 + 2 \cdot 0.8333) = 1.0833 \\ s_4 &= -0.5 - (-1 \cdot 0.75 + -1 \cdot 0.8333) = 1.0833 \end{aligned}$$

Η τιμή λοιπόν της συνάρτησης φράγματος στο αρχικό σημείο x_0 είναι:

$$\phi(x_0) = -(\log(2.4167) + 3 \cdot \log(1.0833)) \approx -1.122$$

Προχωρούμε στην Εσσιανή, από τον τύπο:

$$H(x_0) = \sum_{i=1}^4 \frac{a_i a_i^\top}{s_i^2}$$

Εκτελούμε αναλυτικά τους υπολογισμούς:

$$1. \quad a_1 = [1, 1], \quad s_1 = 2.4167$$

$$\frac{a_1 a_1^\top}{s_1^2} = \frac{\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}}{2.4167^2} = \frac{\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}}{5.840} \approx \begin{bmatrix} 0.1712 & 0.1712 \\ 0.1712 & 0.1712 \end{bmatrix}$$

$$2. \ a_2 = [1, -1], \quad s_2 = 1.0833$$

$$\frac{a_2 a_2^\top}{s_2^2} = \frac{\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}}{1.1736} \approx \begin{bmatrix} 0.852 & -0.852 \\ -0.852 & 0.852 \end{bmatrix}$$

$$3. \ a_3 = [-1, 2], \quad s_3 = 1.0833$$

$$a_3 a_3^\top = \begin{bmatrix} 1 & -2 \\ -2 & 4 \end{bmatrix} \Rightarrow H_3 = \frac{1}{1.1736} \begin{bmatrix} 1 & -2 \\ -2 & 4 \end{bmatrix} \approx \begin{bmatrix} 0.852 & -1.704 \\ -1.704 & 3.408 \end{bmatrix}$$

$$4. \ a_4 = [-1, -1], \quad s_4 = 1.0833$$

$$a_4 a_4^\top = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \Rightarrow H_4 \approx \begin{bmatrix} 0.852 & 0.852 \\ 0.852 & 0.852 \end{bmatrix}$$

Συνεπώς, η τιμή της Εσσιανής είναι το άθροισμα των παραπάνω όρων το οποίο είναι το:

$$\begin{bmatrix} 2.727 & -1.533 \\ -1.533 & 5.283 \end{bmatrix}$$

Μπορούμε τώρα να βρούμε την αναλυτική εξίσωση του ελλειψοειδούς Dikin γύρω από το αρχικό σημείο σύμφωνα με την σχέση:

$$\{y \in \mathbb{R}^2 : (y - x_0)^\top H(x_0)(y - x_0) \leq 1\}$$

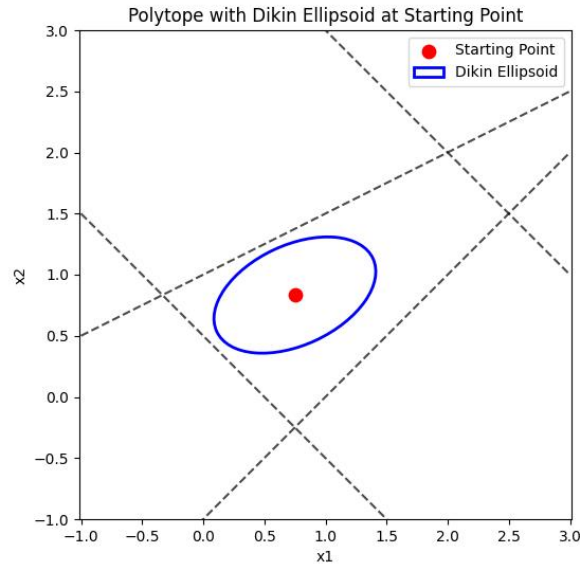
Αντικαθιστούμε τους παραπάνω όρους:

$$\begin{bmatrix} y_1 - 0.75 \\ y_2 - 0.8333 \end{bmatrix}^\top \begin{bmatrix} 2.727 & -1.533 \\ -1.533 & 5.283 \end{bmatrix} \begin{bmatrix} y_1 - 0.75 \\ y_2 - 0.8333 \end{bmatrix} \leq 1$$

και το αποτέλεσμα στην τετραγωνική μορφή είναι:

$$2.727(y_1 - 0.75)^2 - 3.066(y_1 - 0.75)(y_2 - 0.8333) + 5.283(y_2 - 0.8333)^2 \leq 1$$

Μπορούμε να σχεδιάσουμε το παραπάνω ελλειψοειδές ώστε να έχουμε μια εικόνα για την μορφή του. Δείτε το επόμενο σχήμα 2.28. Στο ίδιο γράφημα βλέπουμε το πολύτοπο καθώς και την θέση του αρχικού σημείου.



Σχήμα 2.28: Στην εικόνα βλέπουμε τα όρια του πολυτόπου του παραδείγματος, το αρχικό σημείο το οποίο προέκυψε από τον γραμμικό προγραμματισμό, καθώς και το ελλειψοειδές Dikin που αντιστοιχεί στο δεδομένο σημείο.

Για την συνέχεια, πρέπει να πάρουμε ένα ομοιόμορφο δείγμα εντός του ελλειψοειδούς το οποίο θα είναι και το καινούργιο κέντρο για το επόμενο ελλειψοειδές. Όπως αναφέραμε παραπάνω, μπορούμε να εκτελέσουμε την δειγματοληψία σε μια μπάλα με την συνάρτηση `sample-ball` του Listing 2.5 και στην συνέχεια να το μετατρέψουμε σε σημείο πάνω στο ελλειψοειδές. Ο σωστός μετασχηματισμός είναι ο:

$$y = x + H(x)^{-1/2}u$$

Για να επιβεβαιώσουμε τον παραπάνω ισχυρισμό, χρησιμοποιούμε το ελλειψοειδές που μόλις έχουμε υπολογίσει αναλυτικά για το αρχικό σημείο του πολυτόπου. Το πείραμά μας θα λειτουργήσει ως εξής: θα λάβουμε 1000 σημεία σε ομοιόμορφη κατανομή πάνω σε μια μοναδιαία σφαίρα μέσω της συνάρτησης `sample-ball` και θα τα μετατρέψουμε σε σημείο στο ελλειψοειδές. Ιδανικά, αναμένουμε η κατανομή να παραμείνει ομοιόμορφη και εντός του ελλειψοειδούς. Μερικές από τις γραμμές κώδικα που εκτελούν την παραπάνω διαδικασία είναι οι εξής:

```

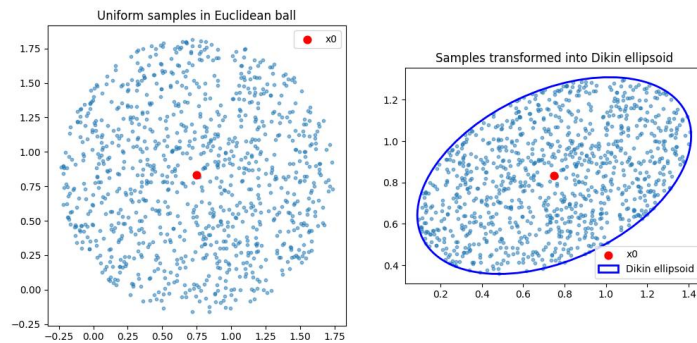
1 n_samples = 1000
2 v = sample_ball(d=2, r=1.0, n=n_samples)
3 points_ball = x0 + v

```

```
4 points_elli = x0 + (Hx_inv_sqrt @ v.T).T
```

Listing 2.21: Part of the code to run the transformation experiment to verify that sampling uniformly on a ball can give us a uniform distribution on the ellipsoid as well.

Σχεδιάζουμε το γράφημα διασποράς για τα παραπάνω σημεία καθώς και τα όρια του ελλειψοειδούς. Δείτε το επόμενο σχήμα 2.29 για το αποτέλεσμα.



Σχήμα 2.29: Αριστερά η κατανομή των 1000 σημείων στην σφαίρα και δεξιά η κατανομή των μετασχηματισμένων σημείων στο ελλειψοειδές. Η κατανομή έχει παραμείνει ομοιόμορφη στον όγκο του ελλειψοειδούς.

Μπορούμε τώρα, να ξεκινήσουμε με τα βασικά σημεία του κώδικα που υπολογίζουν την συνάρτηση φράγματος και την Εσσιανή. Για την συνάρτηση φράγματος, πολύ απλά είναι:

```
1 def phi(x, A=A, b=b):
2     s = b - A @ x
3     if np.any(s <= 0):
4         return np.inf
5     return -np.sum(np.log(s))
```

Listing 2.22: Here we define the border function $\varphi(x)$. The @ symbol is just the matrix multiplication operator introduced in later python versions.

ενώ για την Εσσιανή έχουμε το παρακάτω:

```
1 def hessian_phi(x, A=A, b=b):
2     #Remember that H = sum_i (a_i a_i^T) / (b_i - a_i^T x)^2
3     s = b - A @ x
4     # Case that x not in polytope
5     if np.any(s <= 0):
6         return np.eye(len(x)) * 1e30
```

```

7   H = np.zeros((d, d))
8   # Also rememebr that zip pairs every row of A with every row
    of s
9   for a_i, s_i in zip(A, s):
10      # Convert ai from row to column so that matrix
        multiplication works fine.
11      ai = a_i.reshape(-1, 1)
12      H += (ai @ ai.T) / (s_i**2)
13  return H

```

Listing 2.23: This function calculates the Hessian at the given point.

Στην περίπτωση που το x είναι εκτός του πολυτόπου επιστρέφουμε μια τεράστια τιμή ως ποινή. Η περίπτωση αυτή βέβαια είναι εξαιρετικά σπάνια καθώς το ελλειψοειδές βρίσκεται εξ'ολοκλήρου εντός του πολυτόπου. Στη συνέχεια, απλά υπολογίζουμε την Εσσιανή βάση της σχέσης που είδαμε παραπάνω. Οι παράμετροι της συνάρτησης είναι οι πίνακες A και B , δηλαδή το πολύτοπο και το τρέχον σημείο.

Στην συνέχεια, χρειαζόμαστε και την τετραγωνική ρίζα του αντιστρόφου της Εσσιανής ώστε να μετατρέψουμε την κατανομή στην μπάλα στην κατανομή στο ελλειψοειδές. Η εύρεση του αντιστρόφου υποβοηθάται από το γεγονός πως η Εσσιανή είναι συμμετρική. Πράγματι, για τους όρους που εμφανίζονται στην Εσσιανη ισχύει πως:

$$(a_i a_i^T)^T = a_i a_i^T$$

Αυτό σημαίνει πως ο πίνακας αυτός είναι συμμετρικός και έτσι εξασφαλίζουμε πως 1) έχει πραγματικές ιδιοτιμές και 2) μπορεί να διαγωνοποιηθεί ως εξής:

$$H = V \Lambda V^T$$

όπου V ένας ορθογώνιος πίνακας των ιδιοδιανυσμάτων του H και Λ ένας διαγώνιος πίνακας με τις ιδιοτιμές του H . Τότε, θα έχουμε πως:

$$H^{-1/2} = V \Lambda^{-1/2} V^T$$

και άρα είναι εύκολο να βρούμε τον αντίστροφο πίνακα.

Με βάση τα παραπάνω, μια συνάρτηση που υπολογίζει το ζητούμενο είναι η παρακάτω:

```

1 def inv_sqrt_matrix(H, eps=1e-12):
2     # Numpy can find eigenvalues/eigenvectors easily.
3     vals, vecs = np.linalg.eigh(H)
4     # This is a check to ensure values are not too small.
5     vals_reg = np.maximum(vals, eps)
6     # Now find the necessary matrix.
7     inv_sqrt_vals = 1.0 / np.sqrt(vals_reg)
8     H_inv_sqrt = (vecs * inv_sqrt_vals) @ vecs.T
9     return H_inv_sqrt, vals_reg

```

Listing 2.24: This function calculates the inverse square root of H.

Μένει τώρα, να τρέξουμε τον τυχαίο περίπατο. Ορίζουμε αρχικά τις παραμέτρους του περιπάτου. Θέτουμε 40000 βήματα, περιθώριο λάθους το $1e-12$ και ακτίνα της σφαίρας 0.2 όπως και στις προηγούμενες περιπτώσεις. Ας δούμε την μέθοδο:

```

1 for i in range(1, n_steps):
2     # Compute H and its inverse sqrt using the functions we
3     # defined above
4     Hx = hessian_phi(current)
5     Hx_inv_sqrt, eigvals_x = inv_sqrt_matrix(Hx, eps=1e-12)
6
7     # Sample in Euclidean ball and transform into ellipsoid
8     v = sample_ball(d=d, r=r, n=1)[0]
9     proposal = current + Hx_inv_sqrt @ v
10
11     n_proposed += 1
12
13     # ACCEPT if inside polytope, otherwise REJECT
14     if np.all(A @ proposal <= b + tol):
15         # accept case
16         move_norm = np.linalg.norm(proposal - current)
17         accepted_move_norms.append(move_norm)
18         current = proposal
19         n_accepted += 1
20     else:
21         # reject case
22         n_outside += 1
23
24     samples.append(current.copy())
25
26 samples = np.array(samples)

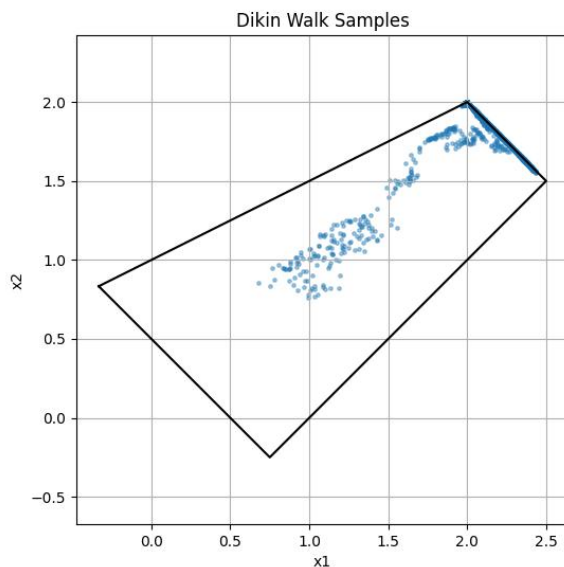
```

Listing 2.25: Run the Dikin walk with $n\text{-steps} = 40000$, $r = 0.2$ and $\text{tol} = 1e-12$.

Υπολογίζουμε αρχικά την Εσσιανή και την αντίστροφή της για κάθε δείγμα, χρησιμοποιώντας τις συναρτήσεις που ορίσαμε παραπάνω. Δειγματοληπτούμε σε

μια σφαίρα ομοιόμορφα και μετατρέπουμε το δείγμα στο ελλειψοειδές. Ελέγχουμε αν το δείγμα βρίσκεται εντός ή εκτός του πολυτόπου. Στην περίπτωση που βρίσκεται εντός το αποδεχόμαστε, το αποθηκεύουμε και το χρησιμοποιούμε ως το νέο δείγμα.

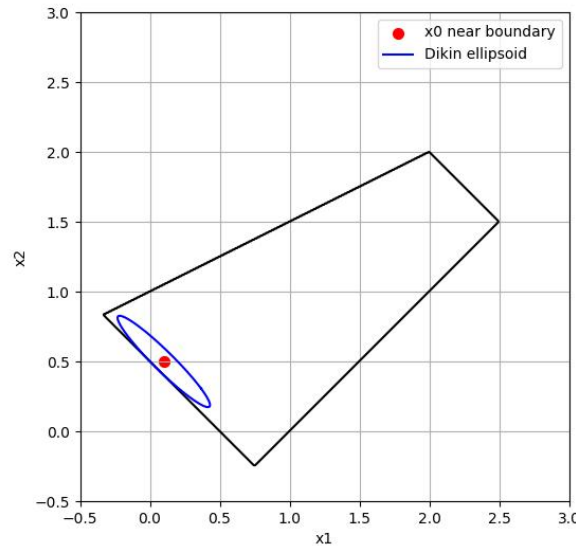
Αφού τρέξουμε τον παραπάνω κώδικα, σχεδιάζουμε το γράφημα διασποράς των σημείων που έχουν προκύψει καθώς και τα όρια του πολυτόπου του παραδείγματος. Το αποτέλεσμα φαίνεται στο παρακάτω γράφημα:



Σχήμα 2.30: Το διάγραμμα διασποράς των δειγμάτων με την μέθοδο Dikin Walk. Προφανώς η δειγματοληψία έχει αποτύχει. Παρατηρούμε πως ενώ η μέθοδος ξεκινά από το αρχικό σημείο πολλά υποσχόμενη, εντούτοις καταρρέει όταν προσεγγίσει κάποιο σύνορο του πολυτόπου.

Η μέθοδος έχει αποτύχει να δώσει μια ομοιόμορφη κατανομή εντός του πολυτόπου. Φαίνεται πως εγκλωβίζεται όταν πλησιάσει κάποιο από τα σύνορα του πολυτόπου. Κοντά στα σύνορα, η ποσότητα $s_i(x)$ που αντιστοιχεί στο σύνορο τείνει στο 0. Συνεπώς, η Εσσιανή, όπου η ποσότητα αυτή εμφανίζεται στον παρονομαστή της, γίνεται υπερβολικά μεγάλη. Το αποτέλεσμα είναι το ελλειψοειδές να επιμηκύνεται στην διεύθυνση του συνόρου, αλλά να συμπιέζεται στην κάθετη διεύθυνση. Δείτε το επόμενο σχήμα 2.31 όπου δείχνουμε το υπολογισμένο ελλειψοειδές για ένα σημείο κοντά στο σύνορο.

Μια σκέψη για την διόρθωση του προβλήματος είναι η εξής: Βάση του σχήματος 2.31, η ομοιόμορφη κατανομή πάνω στο ελλειψοειδές είναι άδικη, υπό την έννοια πως το νέο δείγμα έχει πολύ μεγάλη πιθανότητα να βρεθεί παράλληλα και



Σχήμα 2.31: Όταν το σημείο πλησιάζει το σύνορο, το ελλειψοειδές επιμηκώνεται στην διεύθυνση του συνόρου και συμπιέζεται στην κάθετη διεύθυνση.

κοντά στο σύνορο σε σχέση με την πιθανότητα να βρεθεί πιο μακριά. Για τον σκοπό αυτό, τροποποιούμε την επιλογή του δείγματος πάνω στο ελλειψοειδές κατά τέτοιον τρόπο ώστε να υπάρχει μεγαλύτερη πιθανότητα να βρούμε το σημείο στην κάθετη στο σύνορο διεύθυνση από ότι στην παράλληλη στο σύνορο. Ένας τρόπος να γίνει αυτό είναι να προσθέσουμε κάποια παράμετρο β στην μετατροπή της κατανομής από την μπάλα στο ελλειψοειδές. Αυτή η σκέψη ωστόσο, μάλλον μας οδηγεί κοντά στην μέθοδο ball walk, αφού θα μετατρέψουμε την κατανομή δειγματοληψίας στην μορφή μιας μπάλας.

Μια δεύτερη σκέψη είναι να στρίψουμε το ελλειψοειδές κατά 90 μοίρες. Αυτό θα βοηθήσει να ξεκολλήσουμε από το σύνορο, αλλά θα δώσει πολλά σημεία εκτός του πολυτόπου και επιπλέον προκύπτει το ερώτημα το κατά πόσον ακολουθούμε πλέον την μέθοδο Dikin. Σε κάθε περίπτωση, αφήνουμε το πρόβλημα αυτό ανοικτό.

2.4.10 Billiard Walk

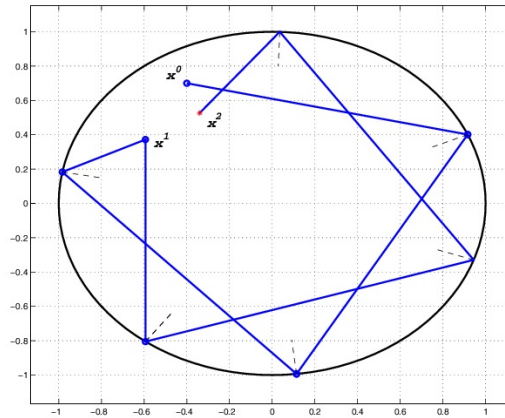
Η επόμενη μέθοδος τυχαίου περιπάτου βασίζεται στον νόμο της ανάκλασης που γνωρίζουμε από την φυσική. Συγκεκριμένα, όταν το φως προσπίπτει σε ένα καθρέπτη η γωνία πρόσπτωσης είναι ίδια με την γωνία ανάκλασης. Το ίδιο συμβαίνει και με τις μπάλες του μπιλιάρδου όταν προσκρούουν στα όρια της επιφάνειας.

Αν χτυπήσουμε μια μπάλα με δύναμη, η πορεία που ακολουθεί καθώς χτυπάει στα όρια ικανοποιεί τον νόμο της ανάκλασης, εξ' ου και το όνομα της μεθόδου. Ξεκινώντας λοιπόν, από ένα αρχικό σημείο και προς μια τυχαία κατεύθυνση στέλνουμε ένα σωματίδιο να κινηθεί εντός του πολύτοπου. Ανά τακτά χρονικά διαστήματα, καταγράφουμε την θέση του σωματιδίου. Όταν το σωματίδιο χτυπήσει κάποιο άκρο, τότε ακολουθεί τον νόμο της ανάκλασης και συνεχίζει την πορεία του.

Πιο συγκεκριμένα, έστω πολύτοπο $P = \{x \in R^d : Ax \leq b\}$ με $A \in R^{m \times d}$, $b \in R^m$. Βρίσκουμε το αρχικό σημείο με γραμμικό προγραμματισμό και λαμβάνουμε μια τυχαία κατεύθυνση με ομοιόμορφη δειγματοληψία πάνω σε μια σφαίρα $u \in S^{d-1}$. Θεωρούμε επιπλέον μήκος L το οποίο θα είναι το μήκος δειγματοληψίας. Κάθε φορά δηλαδή που ταξιδεύουμε αυτό το μήκος, κρατάμε το δείγμα. Ξεκινούμε την κίνηση στην κατεύθυνση $l(\tau) = x_0 + \tau u$, $\tau \geq 0$, δηλαδή πάνω στην ευθεία που ορίζει η κατεύθυνση. Αν δεν πετύχουμε κάποιο σύνορο μετά από απόσταση L , τότε θέτουμε $x_1 = x_0 + Lu$. Σε διαφορετική περίπτωση, έστω πως πετυχαίνουμε το σύνορο $a_i^T x = b_i$. Η εφαρμογή του νόμου της ανάκλασης δίνεται ως:

$$u' = u - 2 \frac{\langle u, a_i \rangle}{\|a_i\|^2} a_i$$

Αυτή είναι και η νέα κατεύθυνση κατά την οποία θα κινηθεί το σωματίδιο. Συνεχίζει την πορεία του έως ότου συμπληρωθεί το μήκος L . Αποθηκεύουμε το δείγμα και ξεκινούμε από την αρχή την διαδικασία με νέο μήκος L και νέα κατεύθυνση. Η διαδικασία τρέχει έως ένα συγκεκριμένο αριθμό βημάτων. Η παρακάτω εικόνα δείχνει μια ενδεικτική εκτέλεση της μεθόδου.



Σχήμα 2.32: Μια ενδεικτική εφαρμογή της μεθόδου Billiard Walk.

2.4.11 Εφαρμογή της Billiard Walk

Θα επιχειρήσουμε τώρα να εφαρμόσουμε την μέθοδο Billiard walk, αρχικά στο 2D παράδειγμα μας και στη συνέχεια σε παραπάνω διαστάσεις. Πολλά από τα στοιχεία που χρειαζόμαστε τα έχουμε δει στην μέθοδο hit-and-run, όπως την επιλογή τυχαίας κατεύθυνσης και την εξίσωση της ευθείας. Αρχικά ας δούμε αναλυτικά τα βήματα του αλγορίθμου:

1. Ορίζουμε το πολύτοπο ως ένα σύνολο ανισοτήτων.
2. Χρησιμοποιούμε γραμμικό προγραμματισμό ώστε να βρούμε ένα αρχικό σημείο εντός του πολυτόπου.
3. Λαμβάνουμε μια τυχαία κατεύθυνση στο χώρο μέσω της δειγματοληψίας από μια μπάλα, ίσων διαστάσεων.
4. Θεωρούμε ένα τυχαίο μήκος βήματος L μέσω της δειγματοληψίας από μια κατανομή, πχ ομοιόμορφη ή εκθετική.
5. Ξεκινώντας από το αρχικό σημείο και την αρχική κατεύθυνση, κινούμαστε μήκος L . Αν χτυπήσουμε σε κάποιο σύνορο, βρίσκουμε την νέα κατεύθυνση μέσω του νόμου της ανάκλασης. Σημειώνουμε το νέο σημείο και επαναλαμβάνουμε για τον κατάλληλο αριθμό βημάτων.

Ας γράψουμε μια συνάρτηση αυτή την φορά, ώστε να είναι πιο απλός ο πειραματισμός στην συνέχεια. Ορίζουμε την παρακάτω συνάρτηση:

```

1 def billiard_walk(A, b, start, n_samples=1000, L_scale=1.0):
2     """
3     Billiard walk inside polytope {x : A x <= b}
4
5     A: (m, d) inequality matrix
6     b: (m,) vector b
7     start: starting point (found by linear programming)
8     n_samples: number of samples to generate
9     L_scale: mean length for exponential distribution
10
11     Returns: array of shape (n_samples, d)
12     """
13     d = A.shape[1]
14     samples = []
15     current = np.array(start, dtype=float)
```

Listing 2.26: Define the billiard walk function. Each parameter is explained in the comments.

Η συνάρτηση δέχεται ως παραμέτρους το πολύτοπο, το αρχικό σημείο, τον αριθμό των βημάτων και μια παράμετρο βημάτων η οποία είναι χρήσιμη ώστε να μεταβάλλουμε το μήκος σε μικρότερο ή μεγαλύτερο μέγεθος. Με d είναι οι διαστάσεις του πολυτόπου οι οποίες είναι όσες και οι ανεξάρτητες μεταβλητές του A .

Στη συνέχεια ψάχνουμε να βρούμε μια τυχαία κατεύθυνση. Χρησιμοποιούμε τον κώδικα από το Listing 2.16 που είδαμε στην μέθοδο hit-and-run. Για τον λόγο αυτό δεν επαναλαμβάνουμε τον κώδικα εδώ. Ακολουθεί η δειγματοληψία ενός μήκους βήματος, δηλαδή το πόσο θα ταξιδέψουμε στο παρών βήμα. Συνήθως το μήκος επιλέγεται μέσω της εκθετικής κατανομής. Κάλλιστα όμως μπορεί να χρησιμοποιηθεί και η ομοιόμορφη κατανομή από 0 έως κάποιο L_{max} .

```

1      # Step 4. Sample travel length L
2      L = np.random.exponential(scale=L_scale)
3      remaining = L

```

Listing 2.27: Sample travel length L, how much we will move this step.

Χρησιμοποιούμε μια επιπλέον μεταβλητή remaining με την οποία θα μετρούμε πόσο μήκος απομένει έπειτα από την ανάκλαση σε κάποιο σύνορο.

Περνάμε τώρα στην ουσία της μεθόδου billiard walk. Ένας τρόπος να εφαρμόσουμε την μέθοδο είναι να βρούμε πόσο απέχει το σύνορο στην τυχαία κατεύθυνση u , ώστε να ξέρουμε αν θα υπάρξει ανάκλαση ή όχι ανάλογα με το αν η απόσταση αυτή είναι μεγαλύτερη ή όχι του L . Πρέπει λοιπόν να μάθουμε το σημείο τομής της ημιευθείας που ορίζει η κατεύθυνση u και το αρχικό σημείο με το αντίστοιχο σύνορο σε αυτήν την κατεύθυνση. Για τον λόγο αυτό πρέπει να λύσουμε την εξίσωση:

$$a_i^T (x_0 + tu) = b_i$$

για κάθε i σύνορο. Αν λύσουμε ως προς t :

$$t = \frac{b_i - a_i^T x_0}{a_i^T u}$$

Όπως είδαμε στην μέθοδο Dikin walk, ο αριθμητής εδώ είναι πάντα θετικός για σημεία που ανήκουν εντός του πολυτόπου $s_i = b_i - a_i^T x_0 \geq 0$ και άρα τα πάντα θα εξαρτηθούν από το πρόσημο στον παρονομαστή, δηλαδή το πρόσημο του εσωτερικού γινομένου $a_i^T u$. Υπάρχουν οι εξής περιπτώσεις:

Θετικό γινόμενο — $a^T u > 0$: Στην περίπτωση αυτή η πορεία δείχνει προς το

υπερεπίπεδο i που σχηματίζει το σύνορο του πολυτόπου. Τότε, ισχύει επίσης πως $t > 0$. Αυτή είναι μια υποψήφια προς τα εμπρός τομή σε απόσταση t κατά μήκος της ακτίνας.

Αρνητικό γινόμενο — $a^\top u < 0$: Η πορεία προς την κατεύθυνση u είναι αντίθετη σε σχέση με το υπερεπίπεδο u με συνέπεια να μην υπάρχει περίπτωση να πετύχουμε το σύνορο αυτό.

Μηδενικό γινόμενο — $a^\top u = 0$: Κατεύθυνση u και υπερεπίπεδο είναι παράλληλα και άρα δεν υπάρχει περίπτωση να υπάρξει κρούση και ανάκλαση.

Όπως πάντα, χρησιμοποιούμε εκτελούμε τον έλεγχο σε σχέση όχι με το 0, αλλά με κάποια μικρή ανοχή που θα μας λύσει το πρόβλημα των οριακών περιπτώσεων και της στρογγυλοποίησης του υπολογιστή. Πρέπει λοιπόν, κατά την διάρκεια του κώδικα να υπολογίσουμε αυτό το εσωτερικό γινόμενο και να ελέγξουμε το προσημό του. Ας δούμε πως θα γίνει αυτό.

```

1  # Step 5. Follow trajectory with reflections
2  while remaining > 1e-12: # while we still have length to
   travel
3      t_hit = np.inf # Set initial t_hit to infinity. No wall
   hit yet
4      hit_index = -1 # This variable holds the index of the
   border we hit
5
6      # Find nearest boundary intersection in current direction
7      for i, (a_i, b_i) in enumerate(zip(A, b)): # Check all
   borders
8          a_dot_u = np.dot(a_i, direction) # inner product  $a^\top u$ .
9          if a_dot_u <= 1e-12:
10             continue # moving away or parallel to this
   constraint, move on to the next
11         numerator = b_i - np.dot(a_i, current)
12         t = numerator / a_dot_u
13         if 1e-12 < t < t_hit: # valid forward intersection
14             t_hit = t
15             hit_index = i

```

Listing 2.28: We try to find the bordered that we might crash into. For this reason we check every border and find the inner product discussed above.

Αρχικά, η εξωτερική επανάληψη ελέγχει το εναπομείναν μήκος. Όσο είναι μεγαλύτερο του μηδενός, πρέπει να ελέγξουμε για τυχόν ανακλάσεις. Στην συνέχεια ορίζουμε δυο μεταβλητές. Η t_hit θα αποθηκεύει το υποψήφιο σημείο προς ανάκλαση, ενώ η hit_index το υποψήφιο σύνορο. Αρχικά τους δίνουμε κάποιες περιεργές τιμές. Ξεκινά τώρα, ο έλεγχος για κάθε σύνορο. Αφού η κατεύθυνση είναι

τυχαία, δεν γνωρίζουμε εκ των προτέρων σε ποιο σύνορο θα χτυπήσουμε και για αυτό θα πρέπει να τα ελέγξουμε όλα. Για κάθε σύνορο, υπολογίζουμε το εσωτερικό γινόμενο $a_i^T \cdot u$ και ελέγχουμε αν είναι μεγαλύτερο του μηδενός (πιο συγκεκριμένα μεγαλύτερο από ένα ελάχιστο όριο ανοχής). Αν ναι, τότε το σύνορο βρίσκεται στην πορεία μας και έτσι υπάρχει πιθανότητα να το χτυπήσουμε. Από όλα τα θετικά t κρατάμε το μικρότερο. Τα υπόλοιπα θετικά t αντιστοιχούν και αυτά σε σύνορα που βρίσκουμε στην πορεία μας, αλλά η τομή γίνεται εκτός των ορίων του πολυτόπου.

Συνεχίζουμε τώρα, με τον έλεγχο για το αν τελικά θα χτυπήσουμε το σύνορο και θα γίνει η ανάκλαση:

```

1      # Distance to next wall
2      if t_hit > remaining or hit_index == -1:
3          # We can travel full remaining length without hitting
4              boundary
5              current = current + remaining * direction
6              remaining = 0.0
7      else:
8          # Travel to the wall
9          current = current + t_hit * direction
10         remaining -= t_hit
11         # Reflect direction:  $u' = u - 2(u \cdot n / ||n||^2) n$ 
12         a_i = A[hit_index]
13         n = a_i / np.linalg.norm(a_i) # normalized normal
14         direction = direction - 2 * np.dot(direction, n) * n
15
16     samples.append(current.copy())

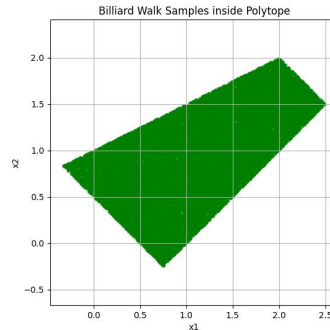
```

Listing 2.29: Finally, check if there is sufficient distance left to hit the border. If not, just update the current point, if yes find the reflected distance and move the remaining distance.

Ξεκινούμε με τον έλεγχο για το αν το υποψήφιο σημείο βρίσκεται σε απόσταση μεγαλύτερη του L ή δεν υπάρχει κανένα υποψήφιο σημείο (αυτό θεωρητικά είναι αδύνατον, μπορεί να συμβεί μόνο σε οριακές περιπτώσεις ή αν έχει λάθος ο κώδικας). Αν το σημείο τομής βρίσκεται σε μεγάλη απόσταση, τότε στην ουσία το μόνο που έχουμε να κάνουμε είναι να αναβαθμίσουμε το τρέχον σημείο. Θυμηθείτε πως το διάνυσμα της κατεύθυνσης είναι μοναδιαίο και άρα ο πολλαπλασιασμός με αυτό δεν αλλάζει το μέτρο. Σε διαφορετική περίπτωση θα χτυπήσουμε στο σύνορο. Αφαιρούμε την απόσταση που έχουμε διανύσει από το remaining και υπολογίζουμε την κατεύθυνση της ανάκλασης από τον τύπο που είδαμε στην θεωρία. Τελικά, αφού η τιμή της remaining είναι μεγαλύτερη του μηδέν, ξανά μπαίνουμε στην επανάληψη while και συνεχίζουμε την διαδικασία.

Μπορούμε τώρα να τρέξουμε την μέθοδο για το παράδειγμά μας και για 40000 βήματα ως συνήθως. Αρκεί να καλέσουμε την συνάρτηση με παραμέτρους τα A και

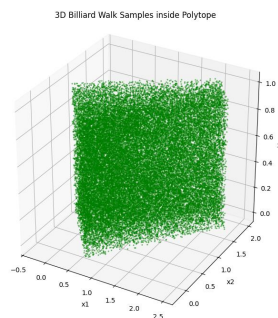
b του παραδείγματος και $n\text{-samples} = 40000$. Για την ώρα ας αφήσουμε $L = 1$. Το αποτέλεσμα φαίνεται στην παρακάτω εικόνα.



Σχήμα 2.33: Το αποτέλεσμα της δειγματοληψίας 40000 σημείων στο πολύτοπο του παραδείγματος με την μέθοδο Billiard Walk.

Η μέθοδος έχει λειτουργήσει άψογα και έχει σχηματίσει μια ομοιόμοφη κατανομή δειγμάτων στο σύνολο της επιφάνειας του πολυτόπου. Όσον αφορά το L -scale που δέχεται ως παράμετρο η συνάρτηση, πρόβλημα πορεύεται μόνο για πολύ μικρά μήκη καθώς ο περίπατος δεν προλαβαίνει να καλύψει όλο το πολύτοπο. Αντιθέτως, μεγάλα μήκη θα καλύψουν ομοιόμορφα το πολύτοπο, αλλά θα αυξήσουν τον χρόνο της μεθόδου καθώς θα πραγματοποιηθούν πολλές ανακλάσεις.

Η συνάρτηση billiard-walk μπορεί να χρησιμοποιηθεί και σε περισσότερες από δυο διαστάσεις. Σε κανένα σημείο δεν ορίζουμε τις διαστάσεις στις οποίες εργαζόμαστε. Αρκεί λοιπόν, να ορίσουμε το πολύτοπο σε όσες διαστάσεις θέλουμε και να τρέξουμε την συνάρτηση. Ας δοκιμάσουμε αρχικά 3 διαστάσεις. Το αποτέλεσμα φαίνεται στην συνέχεια.



Σχήμα 2.34: Το αποτέλεσμα της δειγματοληψίας 40000 σημείων στο πολύτοπο του παραδείγματος με την μέθοδο Billiard Walk σε 3 διαστάσεις.

Ας δοκιμάσουμε επίσης 6 διαστάσεις, με το πολύτοπο του Listing 2.19. Τυπώνουμε τα τελευταία 5 δείγματα.

Last 5 samples:

```
[0.34140269 0.94313671 0.99271327 0.85519785 0.68699482 0.98557651]
[0.47232794 0.78231602 0.71675207 0.54859619 0.90744534 0.7238048 ]
[0.08106716 0.5897296 0.97076318 0.56439334 0.83822999 0.64571987]
[0.14803628 0.6299687 0.89828939 0.71725236 0.64312259 0.80508464]
[0.20821193 0.55092719 0.94187131 0.74486167 0.6372729 0.79616081]
```

Τα παραπάνω σημείο εύκολα μπορούμε να ελέγξουμε πως ικανοποιούν τις ανισότητες του πολυτόπου.

2.4.12 Χαμιλτονιανή Μηχανική

Η επόμενη μέθοδος προχωρά την ιδέα της αντιμετώπισης της δειγματοληψίας μέσω ενός σωματιδίου που κινείται εντός του πολυτόπου ένα βήμα παραπέρα. Το σωματίδιο τώρα δεν ανακλάται απλά στα σύνορα του πολυτόπου, αλλά ακολουθεί κατά την κίνησή του τους φυσικούς νόμους που διέπουν τα πραγματικά σωματίδια. Πιο συγκεκριμένα, η κίνηση του δοκιμαστικού σωματιδίου ακολουθεί τις εξισώσεις του Χάμιλτον που μπορούμε να τις δούμε ως την γενίκευση του δεύτερου νόμου του Νεύτωνα. Για τον σκοπό αυτό, κρίνεται αναγκαίο να δωθεί μια μικρή εισαγωγή στην Χαμιλτονιανή δυναμική.

Η δεύτερη εξίσωση του Νεύτωνα περιγράφει τον τρόπο που κινούνται τα σωματίδια στον χώρο δεδομένης της δύναμης που ασκείται σε αυτά και των αρχικών συνθηκών:

$$\vec{F} = m \ddot{\vec{x}}$$

Η παραπάνω θεώρηση έχει κάποια προβλήματα. Πρώτον υπάρχει ανάγκη για μια αναθεώρησή της στα πλαίσια της σχετικότητας αλλά και της κβαντομηχανικής. Δεύτερον, πρέπει να γνωρίζουμε την δύναμη που ασκείται στο σύστημα κάτι που πολλές φορές δεν είναι απλό. Τρίτον, η μορφή των εξισώσεων κίνησης εξαρτάται έντονα από το σύστημα συντεταγμένων που χρησιμοποιούμε.

Μια εναλλακτική πορεία σε σχέση με την νευτώνεια μηχανική δίνει η αρχή του Hamilton. Σύμφωνα με την αρχή αυτή, υπάρχει μια θεμελιώδης συνάρτηση

των θέσεων και των ταχυτήτων που χαρακτηρίζει το εκάστοτε φυσικό σύστημα. Η συνάρτηση αυτή είναι η λαγκρανζιανή συνάρτηση L . Για μηχανικά συστήματα υπό την επίδραση συντηρητικών δυνάμεων, όπως στην περίπτωση μας, η λαγκρανζιανή συνάρτηση είναι η διαφορά μεταξύ της κινητικής και της δυναμικής ενέργειας του συστήματος:

$$L = T - U$$

Σύμφωνα με την αρχή του Χάμιλτον λοιπόν, ένα σωματίδιο που ξεκινά από το σημείο A την χρονική στιγμή t_A και φτάνει στο σημείο B την χρονική στιγμή t_B , ακολουθεί στο ενδιάμεσο χρονικό διάστημα την διαδρομή εκείνη για την οποία η δράση, δηλαδή η ποσότητα:

$$S = \int_{t_A}^{t_B} L dt$$

καθίσταται στάσιμη.

Η δράση αποτελεί ένα συναρτησοειδές υπό την έννοια πως δεν εξαρτάται από κάποιες μεταβλητές αλλά από την διαδρομή που ακολουθεί το μηχανικό σύστημα. Μια τέτοια ποσότητα καθίσταται στάσιμη, αν για μια μικρή μεταβολή ϵ της διαδρομής, η δράση δεν μεταβάλλεται σε πρώτη τάξη αλλά μόνο σε τάξη $O(\epsilon^2)$. Δείτε τώρα πως η παραπάνω ιδέα οδηγεί αμέσως στο νόμο του Νεύτωνα.

Έστω $x_0(t)$ η φυσική διαδρομή που ακολουθεί ένα σωματίδιο, δηλαδή η διαδρομή που καθιστά την δράση που αντιστοιχεί στην λαγκρανζιανή

$$\frac{1}{2}mu^2 - V(x)$$

στάσιμη. Αν παρεκκλίνουμε λίγο από την διαδρομή αυτή, η δράση δεν θα αλλάξει αισθητά (θα παραμείνει αμετάβλητη σε όρους τάξης ϵ). Πρέπει δηλαδή να ισχύει:

$$S(x_0(t) + \epsilon\eta(t)) - S(x_0(t)) = O(\epsilon^2)$$

όπου $\eta(t)$ είμαι μια αυθαίρετη συνεχώς παραγωγίσιμη συνάρτηση που δείχνει με ποιον τρόπο η καινούργια διαδρομή αποκλίνει από την φυσική διαδρομή. Η μο-

ναδική απαίτηση που πρέπει να ικανοποιεί η $\eta(t)$ είναι να ισχύει πως $\eta(t_A) = \eta(t_B) = 0$, δηλαδή το σωματίδιο να ξεκινά και να καταλήγει στα ίδια σημεία. Υπολογίζουμε την διαφορά στις δυο δράσεις:

$$\begin{aligned}
 S(x_0 + \varepsilon\eta) - S(x_0) &= \left[\int_{t_A}^{t_B} \frac{1}{2} m (\dot{x}_0 + \varepsilon\dot{\eta})^2 dt - \int_{t_A}^{t_B} V(x_0 + \varepsilon\eta) dt \right] \\
 &\quad - \left[\int_{t_A}^{t_B} \frac{1}{2} m (\dot{x}_0)^2 dt - \int_{t_A}^{t_B} V(x_0) dt \right] \\
 &= \int_{t_A}^{t_B} \left[\frac{1}{2} m \left((\dot{x}_0)^2 + 2\varepsilon\dot{x}_0\dot{\eta} + \varepsilon^2(\dot{\eta})^2 \right) - \left(V(x_0) + \varepsilon V'(x_0)\eta + O(\varepsilon^2) \right) \right] dt \\
 &\quad - \int_{t_A}^{t_B} \left[\frac{1}{2} m (\dot{x}_0)^2 - V(x_0) \right] dt \\
 &= \varepsilon \int_{t_A}^{t_B} \left[m\dot{x}_0\dot{\eta} - V'(x_0)\eta \right] dt + O(\varepsilon^2).
 \end{aligned}$$

Με ολοκλήρωση κατά παράγοντες το πρώτο ολοκλήρωμα αποκτά την μορφή:

$$\int_{t_A}^{t_B} m\dot{x}_0\dot{\eta} dt = m\dot{x}_0\eta|_{t_A}^{t_B} - \int_{t_A}^{t_B} m\ddot{x}_0\eta dt = - \int_{t_A}^{t_B} m\ddot{x}_0\eta dt$$

γιατί ισχύει πως $\eta(t_A) = \eta(t_B) = 0$. Συνεπώς, η δράση είναι στάσιμη όταν ο όρος που αποτελεί τον συντελεστή του ε γίνεται μηδενικός:

$$\int_{t_A}^{t_B} \left[m\ddot{x}_0 + V'(x_0) \right] \eta dt = 0$$

Επειδή το παραπάνω πρέπει να ισχύει για κάθε αυθαίρετη συνάρτηση η πρέπει αναγκαστικά η ποσότητα στις αγκύλες να είναι ταυτοτικά μηδενική, δηλαδή να ισχύει πως:

$$m\ddot{x}_0 = -V'(x_0)$$

που είναι ακριβώς ο δεύτερος νόμος του Νεύτωνα αφού για συντηρητικές δυνάμεις ισχύει πως $F = -dV/dx$.

Εφαρμόζοντας την παραπάνω λογική στη γενική περίπτωση μιας λαγκρανζιανής, προσπαθώντας δηλαδή να στασιμοποιήσουμε την δράση για ένα τυχαίο σύστημα, καταλήγουμε στις εξισώσεις Euler-Lagrange οι οποίες δίνονται ως:

$$\frac{\partial L}{\partial x_i} - \frac{d}{dt} \left(\frac{\partial L}{\partial \dot{x}_i} \right) = 0$$

για κάθε i , όπου i οι συντεταγμένες. Η λύση των παραπάνω εξισώσεων δεύτερης τάξης καθιστά την δράση στάσιμη και άρα μας δίνουν τις εξισώσεις κίνησης του συστήματος.

Το βασικό πρόβλημα λοιπόν είναι να γράψουμε την κατάλληλη λαγκρανζιανή για κάθε μηχανικό σύστημα. Οι εξισώσεις της κίνηση θα δοθούν πολύ απλά με μερικές παραγωγίσεις. Ένα απλό παράδειγμα αποτελεί ένα σωματίδιο που κινείται στο βαρυτικό πεδίο της Γης. Όπως έχουμε δει, για μηχανικά συστήματα η λαγκρανζιανή είναι η διαφορά κινητικής και δυναμικής ενέργειας. Συνεπώς στο παράδειγμα αυτό:

$$L = \frac{1}{2}m|\dot{\vec{x}}|^2 + m\vec{g} \cdot \vec{x} = \frac{1}{2}m(\dot{x}^2 + \dot{y}^2 + \dot{z}^2) - mgz$$

Σύμφωνα με την αρχή ελάχιστης δράσης η φυσική διαδρομή πρέπει να είναι τέτοια ώστε η δράση να καθίσταται στάσιμη και συνεπώς να ικανοποιεί τις εξισώσεις Euler-Lagrange. Συνεπώς, έχουμε πως:

$$\frac{\partial L}{\partial x} - \frac{d}{dt} \left(\frac{\partial L}{\partial \dot{x}} \right) = 0$$

$$\frac{\partial L}{\partial y} - \frac{d}{dt} \left(\frac{\partial L}{\partial \dot{y}} \right) = 0$$

$$\frac{\partial L}{\partial z} - \frac{d}{dt} \left(\frac{\partial L}{\partial \dot{z}} \right) = 0$$

οι οποίες δίνουν:

$$m\ddot{x} = 0$$

$$m\ddot{y} = 0$$

$$m\ddot{z} + mg = 0$$

με λύση την:

$$\begin{aligned}x(t) &= x_0 + u_x t \\y(t) &= y_0 + u_y t \\z(t) &= z_0 + u_z t - \frac{1}{2}gt^2\end{aligned}$$

Στην χαμιλτονιανή θεώρηση της κίνησης ενός συστήματος, η βασική συνάρτηση που περιγράφει το σύστημα ονομάζεται χαμιλτονιανή. Η χαμιλτονιανή με μεταβλητές τις θέσεις και τις ορμές $H(q, p, t)$ μπορεί να προκύψει από την λαγκρανζιανή $L(q, \dot{q}, t)$ μέσω ενός μετασχηματισμού Legendre. Έστω μια συνάρτηση $f(x)$. Η παράγωγός της ορίζει μια νέα συνάρτηση

$$p(x) = \frac{df}{dx}$$

Ο μετασχηματισμός Legendre της $f(x)$ είναι εκείνη η συνάρτηση της p , $g(p)$, η οποία έχει την ιδιότητα να έχει ως παράγωγο την αντίστροφη συνάρτηση της $p(x)$, την

$$x(p) = \frac{dg}{dp}$$

Η συνθήκη ύπαρξης μετασχηματισμού Legendre είναι η

$$\frac{dp}{dx} = \frac{d^2 f}{dx^2} \neq 0$$

γιατί μόνο τότε υπάρχει η αντίστροφη της p . Πρέπει δηλαδή η καμπυλότητα της $f(x)$ στο διάστημα που μας ενδιαφέρει να έχει σταθερό πρόσημο. Ο γενική σχέση λοιπόν που προσδιορίζει τον μετασχηματισμό Legendre είναι η:

$$g(p) = x(p)p - f(x(p))$$

Μπορούμε εύκολα να επιβεβαιώσουμε τον παραπάνω ισχυρισμό παραγωγίζοντας:

$$\frac{dg}{dp} = \frac{dx(p)}{dp}p + x(p) - \frac{df}{dx} \frac{dx(p)}{dp} = x(p)$$

αφού εξ'ορισμού είναι $p = df/dx$.

Αντιστρόφως, αν υπάρχει μετασχηματισμός Legendre της $f(x)$ και είναι ο $g(p)$, τότε ο μετασχηματισμός Legendre της $g(p)$ υπάρχει και είναι ο:

$$f(x) = px - g(p)$$

Έτσι λοιπόν, ξεκινώντας από μια λαγκρανζιανή $L(q_1, \dots, q_n, \dot{q}_1, \dots, \dot{q}_n, t)$ αρχικά οι ορμές ορίζονται ως:

$$p_i = \frac{\partial L}{\partial \dot{q}_i}$$

χωρίς καμία αναφορά στη φυσική κίνηση του συστήματος. Η Χαμιλτονιανή ορίζεται τώρα ως ο μετασχηματισμός Legendre της Λαγκρανζιανής ως προς όλες τις γενικευμένες ταχύτητες, δηλαδή είναι η συνάρτηση:

$$H(q_1, \dots, q_n, p_1, \dots, p_n, t)$$

η οποία ικανοποιεί τη σχέση του μετασχηματισμού Legendre:

$$\dot{q}_i = \frac{\partial H}{\partial p_i}$$

Συνεπώς, από την γενική σχέση του μετασχηματισμού, η χαμιλτονιανή δίνεται ως:

$$H(q_1, \dots, q_n, p_1, \dots, p_n, t) = \sum_{i=1}^n p_i \dot{q}_i - L$$

Γράφοντας τις εξισώσεις Euler-Lagrange μέσω της χαμιλτονιανής και χρησιμοποιώντας τον ορισμό μέσω του μετασχηματισμού Legendre, έχουμε τις δυο παρακάτω εξισώσεις του Χάμιλτον:

$$\dot{q}_i = \frac{\partial H}{\partial p_i}$$

$$\dot{p}_i = -\frac{\partial H}{\partial q_i}$$

Οι επίλυση των παραπάνω εξισώσεων δίνει την τροχία του σωματιδίου στον χώρο των φάσεων $2n$ διαστάσεων. Οι $2n$ εξισώσεις του Χάμιλτον είναι διαφορικές εξισώσεις πρώτης τάξης ως προς το χρόνο, ισοδύναμες με τις n εξισώσεις Euler-Lagrange οι οποίες όμως είναι δεύτερης τάξης ως προς τον χρόνο.

Σε ένα σύστημα το οποίο περιγράφεται από μια χρονοανεξάρτητη Λαγκρανζιανή, η Χαμιλτονιανή προκύπτει να είναι σταθερά της κίνησης και περιγράφει την ενέργεια του συστήματος. Είναι δηλαδή το άθροισμα κινητικής και δυναμικής ενέργειας. Σε κάθε περίπτωση όμως, τον βασικό ρόλο έχει η Λαγκρανζιανή, η οποία μπορεί να κατασκευαστεί άμεσα από τις συμμετρίες του συστήματος, ενώ η ενέργεια δεν προκύπτει άμεσα.

2.4.13 Σωματίδιο σε κουτί 1 διάστασης

Για την καλύτερη κατανόηση της εφαρμογής των εξισώσεων ας ξεκινήσουμε από την μονοδιάστατη περίπτωση. Έστω ένα κουτί μιας διάστασης από το 0 έως το L όπου ζει το σωματίδιο. Εκτός του κουτιού, το δυναμικό είναι άπειρο. Θα βρούμε την εξίσωση κίνησης καθώς και πότε ανακλάται το σωματίδιο στα τοιχώματα.

Έστω το κουτί $[0, L]$ και το δυναμικό:

$$U(q) = \begin{cases} 0, & q \in (0, L), \\ +\infty, & q \notin [0, L]. \end{cases}$$

Η Χαμιλτονιανή είπαμε πως είναι η ενέργεια του σωματιδίου, δηλαδή το άθροισμα κινητικής και δυναμικής ενέργειας. Έτσι στο εσωτερικό του κουτιού είναι:

$$H(q, p) = \frac{p^2}{2m} + U(q) = \frac{p^2}{2m}$$

Οι εξισώσεις του Χάμιλτον είναι λοιπόν:

$$\dot{q} = \frac{\partial H}{\partial p} = \frac{p}{m}$$

$$\dot{p} = -\frac{\partial H}{\partial q} = 0$$

Θεωρούμε ως αρχικές συνθήκες τις $q(0) = L/2, u(0) = u_0$. Έτσι $p_0 = mu_0$. Η λύση των εξισώσεων Χάμιλτον, είναι πολύ απλά:

$$\begin{aligned} q(t) &= q_0 + \frac{p_0}{m}t = q_0 + u_0t \\ p(t) &= p_0 \end{aligned}$$

Όταν τώρα το σωματίδιο χτυπά σε κάποιο σύνορο, έστω στο $x = L$, τότε προφανώς δεν μπορεί να περάσει αφού το δυναμικό είναι άπειρο, αλλά ανακλάται. Η ανάκλαση είναι άμεση και έχει ως συνέπεια την στροφή της κατεύθυνσης της ορμής κατά 180 μοίρες $p \rightarrow p' = -p$. Η κρούση είναι ελαστική με συνέπεια η κινητική ενέργεια να διατηρείται.

Η πρώτη κρούση θα γίνει σε χρόνο:

$$t_1 = \frac{L - q_0}{u_0}$$

ενώ η δεύτερη κρούση στο αριστερό τοίχωμα σε χρόνο:

$$t_2 = t_1 + \frac{L}{u_0}$$

Με την ίδια λογική, η k κορύση θα γίνει σε χρόνο:

$$t_k = t_1 + (k - 1) \frac{L}{u_0}$$

Μια πιο χρήσιμη εκδοχή για την κωδικοποίηση της κίνησης, είναι να γράψουμε την θέση του σωματιδίου σε κλειστή μορφή. Αυτό μπορεί να γίνει ως εξής:

$$q(t) = \begin{cases} r(t), & r(t) \leq L, \\ 2L - r(t), & r(t) > L, \end{cases} \quad r(t) = (q_0 + v_0t) \bmod (2L).$$

με ταχύτητα:

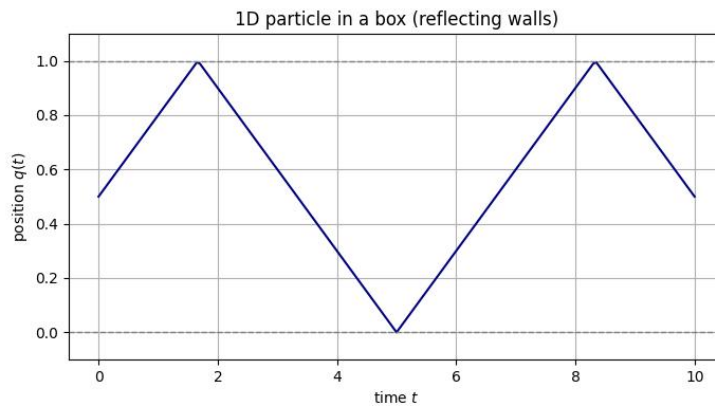
$$v(t) = \begin{cases} +v_0, & r(t) \leq L, \\ -v_0, & r(t) > L. \end{cases}$$

Κατά αυτόν τον τρόπο, μπορούμε εύκολα να γράψουμε ένα κώδικα που να υπολογίζει την θέση σε διάφορες χρονικές στιγμές.

```
1 def box_motion(q0, v0, L, t):
2     q0 = float(q0)
3     v0 = float(v0)
4     t = np.asarray(t, dtype=float)
5     s = q0 + v0 * t
6     r = np.mod(s, 2.0 * L)
7     q = np.where(r <= L, r, 2.0 * L - r)
8     v = np.where(r <= L, v0, -v0)
```

Listing 2.30: Simple function to find the position of the particle for different time values.

Σχεδιάζοντας το γράφημα χρόνου-θέσης, η εικόνα είναι η παρακάτω:



Σχήμα 2.35: Η γραφική παράσταση της κίνησης του σωματιδίου στο μονοδιάστατο κουτί.

Για περισσότερες διαστάσεις, τα πράγματα δεν αλλάζουν σημαντικά. Με το δυναμικό 0 μέσα στο κουτί, μπορούμε να επεξεργαστούμε κάθε συντεταγμένη ανεξάρτητα από τις άλλες. Για κάθε συντεταγμένη λοιπόν θα ισχύουν οι παραπάνω εξισώσεις που είδαμε για την 1D περίπτωση. Αρκεί να συνδυάσουμε όλες τις θέσεις ώστε να βρούμε την πραγματική θέση του σωματιδίου εντός του κουτιού.

2.4.14 Reflective Hamiltonian Monte Carlo method

Στην περίπτωση που το δυναμικό εντός του πολυτόπου είναι μηδενικό όπως στο παράδειγμα της προηγούμενης ενότητας, η κίνηση του σωματιδίου δεν διαφέρει από την κίνηση που ακολουθεί ένα σωματίδιο κατά την μέθοδο Billiard walk που είδαμε σε προηγούμενη ενότητα. Το σωματίδιο κινείται ελεύθερα σε ευθεία γραμμή έως ότου χτυπήσει κάποιο σύνορο άπειρου δυναμικού και ανακλαστεί ελαστικά, δηλαδή χωρίς απώλειες ενέργειας. Μια τέτοια ανάκλαση ακολουθεί προφανώς τον νόμο της ανάκλασης.

Αντιθέτως, κατά την Hamiltonian μέθοδο που θέλουμε να ακολουθήσουμε εδώ, το δυναμικό εντός του πολυτόπου δεν θα είναι πλέον μηδενικό, αλλά μια συνάρτηση $f(x)$. Γενικά λοιπόν, έστω ένα πολύτοπο της γνωστής μορφής των ανισοτήτων. Στην γενική περίπτωση, δεν μας ενδιαφέρει μόνο η ομοιόμορφη δειγματοληψία εντός πολυτόπου, αλλά μπορεί πιθανόν να ψάχνουμε κάποια πολύ ειδική κατανομή. Οι μέθοδοι που είδαμε σε προηγούμενες ενότητες χρησιμοποιούνται κυρίως για ομοιόμορφη δειγματοληψία. Η Hamiltonian μέθοδος ωστόσο, μπορεί εύκολα να χρησιμοποιηθεί για μια συγκεκριμένη περιέργη δειγματοληψία, ανάλογα με την συνάρτηση δυναμικού που θα επιλέξουμε. Όταν η συνάρτηση δυναμικού δεν είναι μηδενική, το σωματίδιο ακολουθεί κάποια καμπύλη τροχία και περνά διαφορετικό ποσοστό του χρόνου σε διαφορετικά σημεία του πολυτόπου.

Ας θεωρήσουμε λοιπόν, πως η κατανομή στόχος από την οποία θέλουμε να γίνει η δειγματοληψία είναι η παρακάτω

$$\pi(x) \propto e^{-f(x)}$$

Παρατηρήστε πως η κατανομή στόχος ρυθμίζεται από την συνάρτηση δυναμικού εντός του πολυτόπου (στα σύνορα και έξω εξακολουθεί το δυναμικό να είναι άπειρο). Πρόκειται ουσιαστικά για την κατανομή Boltzmann. Η κατανομή Boltzmann είναι μια κατανομή πιθανότητας που δίνει την πιθανότητα μιας συγκεκριμένης κατάστασης ως συνάρτηση της ενέργειας αυτής της κατάστασης και της θερμοκρασίας του συστήματος στο οποίο εφαρμόζεται η κατανομή. Η ενέργεια στην περίπτωσή μας είναι ακριβώς η συνάρτηση Χάμιλτον του σωματιδίου.

Σε περιοχές όπου η συνάρτηση $f(x)$ παίρνει μικρές τιμές το εκθετικό μεγαλώνει και άρα αντιστοιχεί σε περιοχή μεγάλης πιθανότητας για την κατανομή. Στην αντίθετη περίπτωση, όταν η $f(x)$ λαμβάνει μεγάλες τιμές το εκθετικό μικραίνει και άρα αντιστοιχεί σε περιοχές μικρής πιθανότητας για την κατανομή. Συνεπώς, το που θα βρούμε τα περισσότερα δείγματα εξαρτάται άμεσα από το δυναμικό.

Στη γλώσσα της φυσικής, όταν το δυναμικό είναι μεγάλο βρισκόμαστε σε περιοχή όπου η κατάσταση του συστήματος έχει μεγάλη ενέργεια και το αντίθετο. Ωστόσο, όπως γνωρίζουμε, καθέ σύστημα τελικά καταλήγει στην κατάσταση ελάχιστης ενέργειας και μέγιστης εντροπίας. Συνεπώς η πιθανότητα να βρούμε το σύστημα σε κάποια κατάσταση χαμηλής ενέργειας είναι μεγαλύτερη. Επιπλέον, η μορφή αυτή της κατανομής είναι χρήσιμη όταν εμφανιστούν λογαριθμικές ποσότητες. Συνήθως, για την αποφυγή ακραίων τιμών σε μεταβλητές και μεγάλων σφαλμάτων χρησιμοποιούμε το λογάριθμο της κατανομής πιθανότητας. Τότε όμως το εκθετικό καταργείται και μας μένει μόνο η συνάρτηση δυναμικού $\log \pi(x) = -f(x)$.

Όπως συνηθίζουμε έως τώρα, ένα παράδειγμα θα βοηθήσει. Ας θεωρήσουμε ως συνάρτηση δυναμικού την:

$$f(x) = \frac{1}{2}\|x\|^2 = \frac{1}{2}x^T x, x \in R^d$$

ενώ για την κινητική ενέργεια θα έχουμε (θεωρούμε μάζα $m = 1$):

$$K(p) = \frac{1}{2}\|p\|^2$$

Μπορούμε τώρα να γράψουμε την Χαμιλτονιανή του σωματιδίου:

$$H(x, p) = \frac{1}{2}\|p\|^2 + \frac{1}{2}\|x\|^2$$

και να βρούμε τις εξισώσεις του Χάμιλτον:

$$\begin{aligned}\dot{x} &= \frac{\partial H}{\partial p} = p \\ \dot{p} &= -\frac{\partial H}{\partial x} = -x\end{aligned}$$

Για κάποιες αρχικές συνθήκες $x(0) = x_0, p(0) = p_0$, οι παραπάνω δυο εξισώσεις αναγνωρίζονται εύκολα ως οι εξισώσεις ενός αρμονικού ταλαντωτή:

$$x(t) = x_0 \cos t + p_0 \sin t$$

με γωνιακή συχνότητα $\omega = 1$. Συνεπώς, εντός του πολυτόπου, η κίνηση είναι περιοδική με περίοδο 2π . Θυμηθείτε πως το x εδώ είναι ένα διάνυσμα, στις δυο διαστάσεις $x = (x_1, x_2)$ και $\|x\|^2 = x_1^2 + x_2^2$. Κατά αυτόν τον τρόπο, η κίνηση ακολουθεί την περιοδική τροχία σε κάθε μια εκ των δυο διαστάσεων.

Η συνάρτηση δυναμικού που έχουμε ορίσει στο παράδειγμα αυτό αυξάνει όσο απομακρυνόμαστε από το αρχικό σημείο. Συνεπώς, τα δείγματα έχουν αυξημένη πιθανότητα να βρεθούν κοντά στο αρχικό σημείο, και μικρή πιθανότητα να βρεθούν πιο μακριά. Συνεπώς η κατανομή που θα λάβουμε δεν είναι πλέον ομοιόμορφη.

Στα παραπάνω προκύπτει το ερώτημα από ποιο σημείο αναφοράς μετράμε την απόσταση. Το x εκφράζει την απόσταση από την αρχή του συστήματος αναφοράς έως τη θέση του σημείου. Επειδή το πολύτοπο θεωρητικά μπορεί να τοποθετηθεί οσοδήποτε μακριά από την αρχή του συστήματος αξόνων, μια ιδέα που μπορούμε να χρησιμοποιήσουμε είναι να κανονικοποιούμε την απόσταση κάθε φορά σε σχέση με το αρχικό σημείο που δίνει ο γραμμικός προγραμματισμός εντός του πολυτόπου. Ουσιαστικά να τοποθετήσουμε το σύστημα των αξόνων πάνω στο αρχικό σημείο του πολυτόπου. Στην περίπτωση αυτή, για την αρχική θέση θα ισχύει πως:

$$x_{new}(t) = x(t) - x_0 \rightarrow x_{new}(0) = x_0 - x_0 = 0$$

με συνέπεια οι εξισώσεις κίνησης να γίνουν:

$$x_{new}(t) = p_0 \sin t$$

$$p_{new}(t) = p_0 \cos t$$

και συνεπώς η περιοδική κίνηση να εξαρτάται μόνο από την αρχική ορμή του σωματιδίου.

Θυμηθείτε πως η ορμή είναι διανυσματικό μέγεθος. Έχει κάποια κατεύθυνση και κάποιο μέτρο. Αν η ορμή παραμείνει σταθερή καθόλη την διάρκεια της δειγματοληψίας, τότε το σωματίδιο ακολουθεί μια συγκεκριμένη κίνηση και επιστρέφει στο αρχικό σημείο. Η δειγματοληψία θα δώσει δείγματα πάνω σε μια συγκεκριμένη καμπύλη. Για τον λόγο αυτό, καλό είναι να δειγματοληπτούμε μια νέα αρχική ορμή στην αρχή κάθε βήματος. Σε περίπτωση τώρα, το σωματίδιο κατά την πορεία του χτυπήσει κάποιο σύνορο του πολυτόπου, εφαρμόζουμε τον νόμο της ανάκλασης και συνεχίζουμε την κίνηση.

2.4.15 Εφαρμογή της Hamiltonian μεθόδου

Θα προσπαθήσουμε να αναπαράγουμε την λογική της μεθόδου. Η γενική εικόνα είναι η εξής:

1. Έχουμε το πολύτοπο ως ένα σύνολο ανισοτήτων. Ορίζουμε αρχικά μια συνάρτηση $f(x)$ η οποία αντιπροσωπεύει το δυναμικό εντός του πολυτόπου και δίνει την κατανομή που θέλουμε να επιτύχουμε σύμφωνα με το εκθετικό που είδαμε στην θεωρία. Το δυναμικό εκτός είναι άπειρο.
2. Βρίσκουμε ένα αρχικό σημείο, είτε με γραμμικό προγραμματισμό, είτε χειροκίνητα. Αυτή θα είναι και η αρχική θέση για την κίνηση του σωματιδίου. Επειδή θα μεταφέρουμε το σύστημα αξόνων στο σημείο αυτό, η αρχική θέση του σωματιδίου θα είναι 0. Για την αρχική ορμή, δειγματοληπτούμε μια τυχαία τιμή σε κάθε βήμα.
3. Χρησιμοποιούμε τις εξισώσεις του Χάμιλτον ώστε να αναπτύξουμε την τροχία του σωματιδίου. Θα χρησιμοποιήσουμε την Leapfrog μέθοδο κατά την οποία διακριτοποιούμε τις εξισώσεις και προχωρούμε σε μικρά βήματα για τα οποία η τροχία θεωρείται ευθύγραμμη.
4. Αν συγκρουστούμε με κάποιο σύνορο του πολυτόπου, εφαρμόζουμε τον νόμο της ανάκλασης. Βρίσκουμε τα υποψήφια σύνορα με την μέθοδο που είδαμε αναλυτικά κατά την εφαρμογή της μεθόδου Billiard walk. Ανατρέξτε στο αντίστοιχο εδάφιο. Έχουμε το δικαίωμα αυτό γιατί κατά τα μικρά βήματα η τροχία θεωρείται ευθύγραμμη.
5. Σε κάθε βήμα της μεθόδου, δειγματοληπτούμε μια αρχική ορμή, τρέχουμε την μέθοδο Leapfrog για κάποιο αριθμό βημάτων και αποθηκεύουμε την τελική θέση του σημείου. Αυτό είναι και το δείγμα μας. Συνεχίζουμε την διαδικασία με βάση το νέο δείγμα.

Ξεκινούμε από κάποιες απλές συναρτήσεις τις οποίες μπορούμε εύκολα να κωδικοποιήσουμε και οι οποίες είναι απαραίτητες για την συνέχεια. Αρχικά, για κάποιο δυναμικό $f(x)$ πρέπει να υπολογίσουμε την παράγωγό του ως προς x αφού αυτή εμφανίζεται στην εξίσωση του Χάμιλτον. Η χαμιλτονιανή του σωματιδίου είναι η ενέργεια:

$$H = \frac{\|p\|^2}{2m} + f(x)$$

και συνεπώς η εξίσωση του Χάμιλτον γράφεται ως:

$$\dot{p} = -\nabla_p H(x, p) = -\nabla f(x)$$

όπου έχουμε χρησιμοποιήσει τον πιο επίσημο συμβολισμό με το ανάδελτα που συμβολίζει την κλίση της συνάρτησης, δηλαδή την παράγωγο ως προς κάθε συνιστώσα του διανύσματος x . Όσον αφορά την άλλη εξίσωση είναι πολύ απλά πως:

$$\dot{x} = \frac{\partial H}{\partial p} = p$$

Η λύση μπορεί να βρεθεί είτε υπολογίζοντας πρώτα το p από την πρώτη εξίσωση και αντικαθιστώντας στην δεύτερη, είτε παραγωγίζοντας την δεύτερη εξίσωση και αντικαθιστώντας στην πρώτη να βρούμε την θέση.

Η πιο σωστή αντιμετώπιση του υπολογισμού της κλίσης του δυναμικού θα ήταν η χρήση κάποιας αριθμητικής μεθόδου, όπως της Runge-Kutta 4ης τάξης. Ωστόσο, αυτό θα προσθέσει ένα επιπλέον βαθμό πολυπλοκότητας που δεν μας χρειάζεται για αυτή την εισαγωγή στην μέθοδο. Για τον σκοπό αυτό θα περιοριστούμε σε δυναμικά για τα οποία μπορούμε εύκολα να υπολογίσουμε θεωρητικά την κλίση τους, όπως στο παράδειγμα που είδαμε στην προηγούμενη ενότητα. Αφού υπολογίσουμε την κλίση, την εισάγουμε χειροκίνητα στον κώδικα και συνεχίζουμε. Συνεπώς, το μόνο που έχουμε να κάνουμε είναι να ορίσουμε μια συνάρτηση που για κάποιο x επιστρέφει την τιμή της κλίσης.

```
1 def grad_f(x, x_start):
2     diff = x - x_start
3     return diff
```

Listing 2.31: We will use simple potentials that we can find the gradient on paper and plug it in.

Στο παραπάνω παράδειγμα έχουμε θεωρήσει την σύμβαση κατά την οποία μεταφέρουμε την αρχή των αξόνων στο αρχικό σημείο και το δυναμικό που είδαμε στο παράδειγμα της προηγούμενης ενότητας

$$f(x) = \frac{1}{2}\|x\|^2$$

με παράγωγο

$$\nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_d} \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix} = x$$

Αυτό που έχει σημασία είναι πως για κάθε συνάρτηση δυναμικού, υπολογίζουμε την παράγωγο και την εισάγουμε στην παραπάνω συνάρτηση `grad-f`. Μια ακόμη χρήσιμη συνάρτηση είναι αυτή που κανονικοποιεί ένα διάνυσμα ώστε να έχει μοναδιαίο μέτρο. Για διανύσματα που μας δείχνουν κατεύθυνση, δεν μας ενδιαφέρει το μέτρο τους. Πολύ απλά:

```
1 def unit_normal(a):
2     return a / np.linalg.norm(a)
```

Listing 2.32: Normalize a vector.

Η μέθοδος του NumPy υπολογίζει το μέτρο του διανύσματος

$$\|a\| = \sqrt{a_1^2 + a_2^2 + \dots + a_d^2}$$

και διαιρεί με την τιμή αυτή ώστε να διάνυσμα να γίνει μοναδιαίο.

Ένα ακόμη χρήσιμο διάνυσμα είναι αυτό που μας δίνει την ανακλώμενη κατεύθυνση του σωματιδίου όταν χτυπήσει κάποιο τοίχωμα. Όπως είδαμε στην μελέτη της μεθόδου `Billiard walk`, χρησιμοποιώντας τον νόμο της ανάκλασης κατά την πτώση του σωματιδίου στο φράγμα άπειρου δυναμικού που είναι κάποιο σύνορο του πολυτόπου, η νέα κατεύθυνση δίνεται από την σχέση:

$$u' = u - 2(u \cdot n)n$$

όπου με u' η νέα κατεύθυνση, u η τωρινή κατεύθυνση και n το κάθετο μοναδιαίο διάνυσμα πάνω στο σύνορο που χτυπάμε. Συνεπώς:

```
1 def reflect_vector(current_u, wall_n):
2     n = unit_normal(wall_n)
3     return current_u - 2.0 * np.dot(current_u, n) * n
```

Listing 2.33: Find the reflected vector of momentum.

Βέβαια, η χρήση της συνάρτησης ανάκλασης προϋποθέτει πως ξέρουμε το τοίχωμα που θα χτυπήσουμε στην συνέχεια της κίνησης. Πρέπει με κάποιον τρόπο να αντιμετωπίσουμε το πρόβλημα αυτό. Ευτυχώς για εμάς έχουμε αντιμετωπίσει το πρόβλημα αυτό ξανά στη μέθοδο Billiard walk. Για ένα αρχικό σημείο x και μια τυχαία κατεύθυνση u , πρέπει να ακολουθήσουμε την πορεία αυτή και να βρούμε το πρώτο σύνορο που συναντάμε:

$$a_i^T (x + ut) = b_i$$

Η ημιευθεία αυτή πιθανώς τέμνει πολλά σύνορα, αλλά μόνο η πρώτη τομή βρίσκεται πάνω στα όρια του πολυτόπου. Οι υπόλοιπες τομές συμβαίνουν εκτός του πολυτόπου. Για τον λόγο αυτό, ψάχνουμε το ελάχιστο t για το οποίο ικανοποιείται η ισότητα δοκιμάζοντας όλα τα σύνορα. Δείτε την ενότητα για την εφαρμογή της Billiard walk όπου έχουμε θεωρήσει όλες τις περιπτώσεις.

Επειδή στην περίπτωση πολλών διαστάσεων και πολλών facets η επανάληψη πάνω σε όλα γίνεται κάπως αργή, θα γράψουμε την συνάρτηση χρησιμοποιώντας διανυσματική λογική, αφήνοντας στο NumPy τις επαναλήψεις. Ας δούμε την συνάρτηση:

```

1 def intersection_candidates(A, b, x, u, tol=1e-12):
2     # Remember t = (b_i - aT_i * x0) / (aT_i * u)
3     a_dot_u = A.dot(u) # denominator
4     s = b - A.dot(x) # numerator
5     t_candidates = np.full(A.shape[0], np.inf) # Start with
        infinite t. No wall hit yet
6     mask = a_dot_u > tol # Create a mask that will ignore
        negative t. We move away from these borders.
7     t_candidates[mask] = s[mask] / a_dot_u[mask] # Apply the mask
8     valid = (t_candidates > tol) & (t_candidates <= 1.0 + tol) #
        We only care about solutions during our finite step from
        x to x+u ie. for t:(0,1]
9     if not np.any(valid): # If no valid t, we hit no border.
        Return nothing.
10    return None, None
11    t_hit = np.min(t_candidates[valid]) # Otherwise, we hit the
        border for the lowest t.
12    hit_idx = np.nonzero(np.abs(t_candidates - t_hit) <= max(tol,
        1e-14))[0] # Special case where we hit a corner. We
        return all indices.
13    return float(t_hit), hit_idx

```

Listing 2.34: Find if and which border we hit by checking all of them and comparing them with the direction we move in.

Παρατηρήστε πως δεν έχουμε καμία επανάληψη for, αλλά εφαρμόζουμε τις πράξεις πάνω σε όλα τα στοιχεία του πίνακα του πολυτόπου. Έτσι, η λογική τρέχει πιο γρήγορα σε σχέση με τις επαναλήψεις κατά την εφαρμογή της Billiard Walk. Συνίστουμε ξανά να δείτε την ενότητα με την εφαρμογή της Billiard walk όπου εξηγούμε αναλυτικά το πως βρίσκουμε το σύνορο που θα χτυπήσουμε στην τυχαία κατεύθυνση u . Σε γενικές γραμμές, λύνουμε την εξίσωση ως προς t

$$t = \frac{b - A^T x_0}{A^T u}$$

και υπολογίζουμε αριθμητή και παρονομαστή. Ο αριθμητής είναι πάντα θετικός εντός του πολυτόπου και άρα το πρόσημο καθορίζεται από τον παρονομαστή. Θετικό πρόσημο σημαίνει πως κινούμαστε προς το σύνορο, ενώ αρνητικό πως κινούμαστε από το σύνορο. Για τον λόγο αυτό απορρίπτουμε τις αρνητικές λύσεις για το t εφαρμόζοντας μια συνθήκη (mask) σε όλα τα στοιχεία του πίνακα $a \cdot u$ ώστε ο υπολογισμός να γίνει μόνο για τα θετικά. Στη συνέχεια μας ενδιαφέρει να βρούμε κάποια τομή μόνο για το τρέχον βήμα και όχι γενικά την τομή με το κοντινότερο σύνορο. Για αυτό περιορίζουμε τις λύσεις από το 0 ως το 1. Επιστρέφουμε none αν δεν υπάρχει σύγκρουση, ενώ επιστρέφουμε το t και τον δείκτη του συνόρου που χτυπάμε.

Ας βέβαια σκεφτεί κανείς λιγό περισσότερο την παραπάνω συνάρτηση, προκύπτει το εξής ερώτημα. Η διαδικασία που ακολουθήσαμε είναι ίδια με την Billiard Walk, απλά τώρα δουλέψαμε διανυσματικά που είναι πιο γρήγορο. Κατά αυτόν τον τρόπο, η μέθοδος με την οποία υπολογίζουμε το ποιο σύνορο χτυπάμε στην τυχαία κατεύθυνση u είναι να θεωρήσουμε την ευθεία που ορίζει η κατεύθυνση αυτή και να βρούμε την απόσταση μεταξύ αρχικού σημείου και κοντινότερου συνόρου. Η κίνηση δηλαδή ενός υποθετικού σωματιδίου είναι ευθύγραμμη. Ωστόσο, στην περίπτωση της Χαμιλτονιανής κίνησης, η κίνηση δεν είναι κατ'ανάγκη ευθύγραμμη, αλλά καμπύλη που εξαρτάται από το δυναμικό εντός του πολυτόπου. Για τον σκοπό αυτό, ο παραπάνω υπολογισμός του σημείου κρούσης δεν έχει ιδιαίτερο νόημα για εμάς. Το σωματίδιο δεν θα κινείται ευθεία προς το σύνορο που θα χτυπήσει, αλλά θα ακολουθήσει μια καμπύλη με συνέπεια το υποψήφιο προς σύγκρουση σύνορο να μεταβάλλεται από στιγμή σε στιγμή.

Η λύση στο παραπάνω πρόβλημα είναι να διακριτοποιήσουμε τις εξισώσεις του Χάμιλτον. Αντί να υπολογίσουμε συνολικά την τροχία του σωματιδίου εντός του πολυτόπου, χωρίζουμε τον υπολογισμό της τροχίας σε μικρά βήματα, που όμως διατηρούν την συνολική ενέργεια και τον όγκο της τροχιάς στο χώρο των φάσεων (διάγραμμα θέσης-ταχύτητας). Για τον σκοπό αυτό σπάμε τον υπολογισμό σε ενδιάμεσα ημιβήματα για την ορμή και σε βήματα για την θέση:

$$p_{n+\frac{1}{2}} = p_n - \frac{\eta}{2} \nabla f(x_n)$$

$$x_{n+1} = x_n + \eta p_{n+\frac{1}{2}}$$

$$p_{n+1} = p_{n+\frac{1}{2}} - \frac{\eta}{2} \nabla f(x_{n+1})$$

όπου η το μέγεθος του βήματος, ένας μικρός αριθμός ανάλογα με την ακρίβεια που θέλουμε. Η λογική εδώ είναι να χωρίσουμε την κατά τα άλλα καμπύλη τροχία του σωματιδίου σε μικρά, σχεδόν ευθύγραμμα τμήματα για τα οποία μπορούμε να εφαρμόσουμε τις μεθόδους που είδαμε πιο πάνω. Η ορμή αναβαθμίζεται πριν και μετά από την το βήμα της θέσης και για τον λόγο αυτό η μέθοδος αυτή ονομάζεται Leapfrog. Ορμή και θέση εναλλάσσονται στους υπολογισμούς

$$p_{n+\frac{1}{2}} \rightarrow x_{n+1} \rightarrow p_{n+1}$$

και άρα το βήμα από ορμή σε ορμή και θέση σε θέση θυμίζει το βήμα ενός βατράχου.

Ας δούμε πως θα εφαρμόσουμε την Leapfrog μέθοδο σε απλά βήματα. Δείτε τον παρακάτω κώδικα.

```

1 def leapfrog_reflect(x, v, eta, A, b):
2     """
3     v: current momentum
4     v_hat: momentum at half step
5     grad_f: The gradient of f
6     u: distance moved this step eta * p_(n+1/2)
7     x_new: new position after considering reflections, also
          update momentum
8     v_new: Second momentum update
9     """
10    # half momentum update
11    v_hat = v - 0.5 * eta * grad_f(x, x0)
12    # position update (with reflections) and reflect v_hat
          accordingly
13    u = eta * v_hat
14    x_new, v_hat_new, n_refl = reflect_segment_with_velocity(x,
          u, v_hat, A, b)
15    # final half momentum update at new position
16    v_new = v_hat_new - 0.5 * eta * grad_f(x_new, x0)
17    return x_new, v_new

```

Listing 2.35: Apply the leapfrog method. Split the curved trajectory into really small, almost linear segments. Update each step using the discrete Hamiltonian equations.

Υπολογίζουμε αρχικά το ημι-βήμα της ορμής. Η κλίση του δυναμικού υπολογίζεται στο αρχικό σημείο. Στη συνέχεια υπολογίζουμε το βήμα της θέσης. Υπάρχει ωστόσο περίπτωση να έχουμε ανάκλαση σε κάποιο σύνορο και για αυτό χειριζόμαστε την θέση σε ξεχωριστή συνάρτηση που θα δούμε στην συνέχεια. Τέλος, αναβαθμίζουμε εκ νέου την ορμή σύμφωνα με την μέθοδο.

Για να έχουμε μια αληθοφανή τροχία πρέπει να τρέξουμε την παραπάνω συνάρτηση πολλές φορές, ανάλογα με το βήμα η . Πρέπει δηλαδή να αφήσουμε χρόνο στην τροχιά να εξελιχθεί. Κατά την διάρκεια αυτών των μικρών βημάτων μπορεί να συμβούν μια ή περισσότερες ανακλάσεις στα φράγματα δυναμικού των συνόρων. Αυτά τα αντιμετωπίζουμε με την συνάρτηση `reflect-segment-with-velocity` ώστε να μας θυμίζει να αναβαθμίσουμε και την ταχύτητα (ορμή). Η λογική είναι απλή. Σε κάθε μικρό βήμα βρίσκουμε το υποψήφιο σημείο κρούσης από την `intersection-candidates` και ελέγχουμε αν έχουμε μετακινηθεί κατά το βήμα αυτό αρκετά ώστε να το χτυπήσουμε. Δείτε την αναλυτική συνάρτηση:

```

1 def reflect_segment_with_velocity(x, u, v_hat, A, b, tol=1e-12,
2   max_reflections=200):
3     x_curr = x.copy() # current particle position
4     u_curr = u.copy() # remaining displacement vector for this
5       leapfrog step
6     v_curr = v_hat.copy() # current momentum (velocity) to be
7       reflected
8     reflections = 0 # reflection counter, steps are small but we
9       might be close to a corner
10    # Move until there is displacement left to travel.
11    # We also set a maximum number of reflections in order to
12      avoid corner traps.
13    while np.linalg.norm(u_curr) > tol and reflections <
14      max_reflections:
15      # Find the candidate facet we are about to hit
16      t_hit, hit_idx = intersection_candidates(A, b, x_curr,
17        u_curr, tol=tol)
18      # If t_hit is too far, we are safe to move the rest of
19        the distance. End while loop.
20      if t_hit is None:
21        x_curr = x_curr + u_curr
22        u_curr = np.zeros_like(u_curr)
23        break
24      move = t_hit * u_curr # Movement until first hit
25      x_curr = x_curr + move # New position at the wall
26      u_curr = u_curr - move # Movement left after reflection
27      normals = A[hit_idx]
28      # If multiple walls are hit simultaneously, get a
29        combined reflection direction
30      combined = np.sum(normals, axis=0)
31      # If the sum is near zero (numerical issue), just use the
32        first value.

```

```

23         if np.linalg.norm(combined) < 1e-16:
24             combined = normals[0]
25             # reflect remaining displacement and velocity vectors
26             u_curr = reflect_vector(u_curr, combined)
27             v_curr = reflect_vector(v_curr, combined)
28             reflections += 1
29             # Tiny nudge away from the wall to avoid hitting the same
            wall again due to numerical rounding.
30             x_curr = x_curr + 1e-14 * (combined /
                np.linalg.norm(combined))
31         if reflections >= max_reflections:
32             # Keep track to avoid infinite loops if stuck in a corner.
33             raise RuntimeError("Too many reflections in one position
                update.")
34         """
35         ***Return variables***
36         x_curr: final position after all reflections in this step
37         v_curr: updated momentum after reflections
38         reflections: number of reflections occurred
39         """
40         return x_curr, v_curr, reflections

```

Listing 2.36: Check at every Leapfrog step if we hit any wall. To do this, find the distance to the closes border and compare it to the distance we will travel this step. If we dont travel far enough, just update the position. If we do manage to hit the wall, reflect position and momentum and travel the remaining distance to the new reflected direction. Return final position and momentum.

Η συνάρτηση δέχεται ως είσοδο την τωρινή θέση και ορμή, καθώς και τα σύνορα του πολυτόπου. Θεωρώντας ευθύγραμμη κίνηση (κάτι το οποίο ισχύει για τα Leapfrog βήματα) βρίσκουμε πόσο μακριά είναι ο τοίχος και συγκρίνουμε την απόσταση αυτή με το πόσο θα κινηθούμε σε αυτό το βήμα. Αν δεν φτάνουμε το σημείο κρούσης, το μόνο που έχουμε να κάνουμε είναι να κινηθούμε στην κατεύθυνση της ορμής κατά τν απόσταση του βήματος. Σε περίπτωση όμως που τύχει να συγκρουστούμε, βρίσκουμε τα ανακλώμενα διανύσματα θέσης και ορμής και συνεχίζουμε το υπόλοιπο της κίνησης κατά την κατεύθυνση αυτή.

Κατά την διάρκεια της διαδικασίας προσπαθούμε να λάβουμε μέτρα απέναντι σε ακραίες περιπτώσεις. Αν για παράδειγμα βρισκόμαστε κοντά σε κάποια κορυφή του πολυτόπου όπου συναντώνται δυο σύνορα (ή περισσότερα σε περισσότερες διαστάσεις), υπάρχει περίπτωση να κολλήσουμε πραγματοποιώντας συνεχόμενες επαναλήψεις χωρίς να προχωρούμε καθόλου. Επιπλέον, υπάρχει το ενδεχόμενο η κίνηση να σταματήσει τόσο κοντά στο σύνορο ώστε ο κώδικας να θεωρήσει πως βρισκόμαστε πάνω σε αυτό και να πραγματοποιήσει μια αναίτια ανάκλαση. Για τους παραπάνω λόγους έχουμε εισάγει κάποιες ανοχές στις τιμές των μεταβλητών.

Απομένει να εφαρμόσουμε όλα τα παραπάνω. Χρειαζόμαστε ένα αρχικό σημείο, αριθμό βημάτων, αριθμό βημάτων Leapfrog και μέγεθος βήματος η . Θεωρούμε τυχαία αρχική ορμή σε κάθε επανάληψη και αρχική θέση το προηγούμενο δείγμα.

```

1 def simulate_reflective_hamiltonian(x0, n_steps=200, w=20,
2   eta=0.1):
3     # Lets simulate a single chain.
4     # At every iteration, sample momentum ~ N(0,I) and run w
5     leapfrog steps.
6     x = x0.copy()
7     traj = [x.copy()]
8     for it in range(n_steps):
9         # sample momentum
10        v = np.random.normal(size=x.shape)
11        # integrate w leapfrog steps
12        for _ in range(w):
13            x, v = leapfrog_reflect(x, v, eta, A, b)
14            traj.append(x.copy())
15    return np.array(traj)
16
17 # ---- Initial parameters ----
18 x0 = np.array([2, 1.8])
19 traj = simulate_reflective_hamiltonian(x0, n_steps=5000, w=10,
20   eta=0.15)

```

Listing 2.37: Set random walk parameters and apply the method.

Εδώ θέτουμε το αρχικό σημείο του πολυτόπου με το χέρι ώστε να δοκιμάσουμε διάφορες περιπτώσεις. Θα μπορούμε να έχουμε χρησιμοποιήσει τον επαυξημένο γραμμικό προγραμματισμό για την εύρεση σημείου στο εσωτερικό του πολυτόπου. Οι μεταβλητές n -steps, w , η ορίζουν τις παραμέτρους για τον συνολικό αριθμό δειγμάτων, επαναλήψεις της Leapfrog και το βήμα της Leapfrog. Εκτελούμε όσες επαναλήψεις απαιτεί η παράμετρος. Ξεκινάμε αρχικά από το x_0 της αρεσκείας μας και σε κάθε επόμενο βήμα χρησιμοποιούμε ως αρχική θέση το προηγούμενο x ενώ για την ορμή δειγματοληπτούμε από την κανονική κατανομή. Προφανώς ικανοποιούμε τις απαιτήσεις μιας αλυσίδας Markov όπου κάθε επόμενο δείγματα εξαρτάται από το προηγούμενο και μόνο. Δείτε τα αντίστοιχα κεφάλαια.

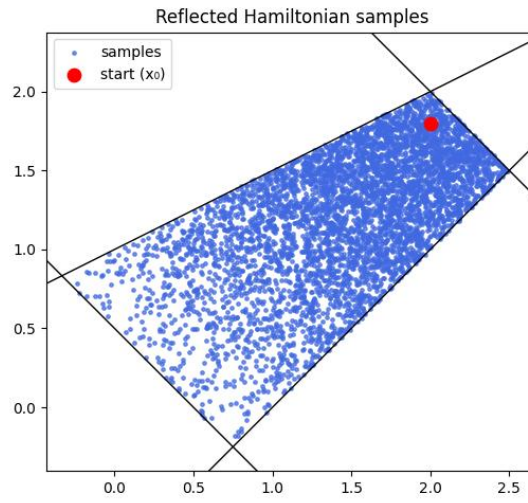
Ήρθε λοιπόν η ώρα να παρουσιάσουμε κάποια από τα αποτελέσματα. Το πρώτο δυναμικό που δοκιμάζουμε είναι βέβαια αυτό που χρησιμοποιήσαμε στο παράδειγμα

$$f(x) = \frac{1}{2}\|x\|^2 \rightarrow \nabla f(x) = x$$

Τι αναμένουμε από το δυναμικό αυτό; Όσο το x αυξάνει, η πιθανότητα της κα-

τανομής $\pi(x) \propto e^{-f(x)}$ να τύχει το σημείο στη θέση αυτή μειώνεται. Η μεγαλύτερη πιθανότητα εντοπίζεται στο να βρούμε το x κοντά στο αρχικό σημείο. Περιμένουμε λοιπόν μια κατανομή για την οποία τα περισσότερα δείγματα θα βρίσκονται κοντά στο αρχικό σημείο x_0 , ενώ θα αραιώνουν μακριά από αυτό.

Ως παράδειγμα, χρησιμοποιούμε το γνωστό πολύτοπο που έχουμε επεξεργαστεί έως τώρα στις δυο διαστάσεις. Θα δούμε δυο γραφήματα, στο πρώτο ορίζουμε το αρχικό σημείο στα αριστερά του πολύτοπου ενώ στη δεύτερη στα δεξιά. Δείτε το αποτέλεσμα στα σχήματα 2.36 και 2.37.

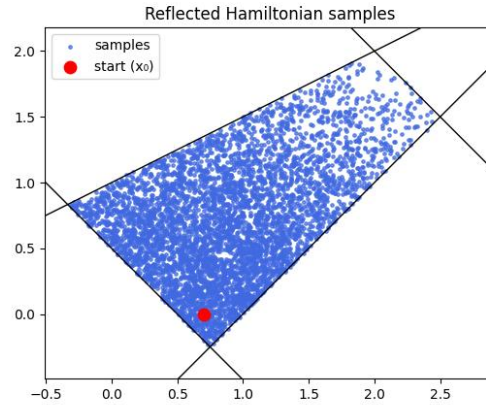


Σχήμα 2.36: Δειγματοληψία με την μέθοδο Reflective Hamiltonian και δυναμικό $f(x) = \frac{1}{2}\|x\|^2$. Αρχικό σημείο το $[2, 1.8]$.

Τα δείγματα ακολουθούν την συμπεριφορά που αναμένουμε. Η κατανομή τους είναι πιο πυκνή κοντά στο αρχικό σημείο, δηλαδή στο σημείο όπου το δυναμικό είναι μηδέν και άρα αποτελεί ένα πηγάδι δυναμικού, και πιο αραιά στον χώρο όπου το δυναμικό είναι μεγάλο και άρα η ενέργεια είναι επίσης περισσότερη.

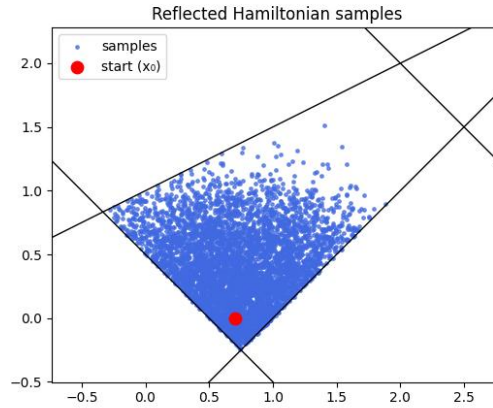
Ας δοκιμάσουμε ένα πιο ακραίο δυναμικό, εκθετικής μορφής ώστε να διαπιστώσουμε αν τα δείγματα θα συγκεντρωθούν ακόμη πιο κοντά στο πηγάδι δυναμικού:

$$f(x) = e^{x^2} \rightarrow \nabla f(x) = 2xe^{x^2}$$

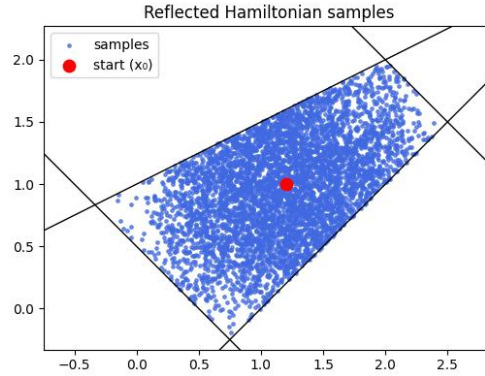


Σχήμα 2.37: Δειγματοληψία με την μέθοδο Reflective Hamiltonian και δυναμικό $f(x) = \frac{1}{2}\|x\|^2$. Αρχικό σημείο το $[0.7, 0]$.

Το αποτέλεσμα της δειγματοληψίας φαίνεται στο σχήμα 2.38 όπου έχουμε ίδιο αρχικό σημείο με το 2.37 ώστε να γίνει η σύγκριση και 2.39.



Σχήμα 2.38: Δειγματοληψία με την μέθοδο Reflective Hamiltonian και δυναμικό $f(x) = e^{x^2}$. Αρχικό σημείο το $[0.7, 0]$, όπως με το προηγούμενο δυναμικό. Η κατανομή έχει αλλάξει και έχει προσεγγίσει το αρχικό σημείο ακόμη περισσότερο.



Σχήμα 2.39: Δειγματοληψία με την μέθοδο Reflective Hamiltonian και δυναμικό $f(x) = e^{x^2}$. Αρχικό σημείο το $[1.2, 1]$.

Όσον αφορά τις παραμέτρους της μεθόδου, γίνεται κατανοητό πως η κατανομή εξαρτάται από το αρχικό σημείο. Από εκεί και πέρα, ιδιαίτερη σημασία έχει η παράμετρος η . Πολύ μικρή τιμή της παραμέτρου, δεν αφήνει την Leapfrog μέθοδο να κινιθεί αρκετά. Αν δοκιμάσετε για παράδειγμα τιμή $\eta = 0.001$ ο περίπατος που θα δείτε δεν θα είναι και τόσο τυχαίος. Θα μπορούσαμε να πούμε πως το η έχει ανάλογο ρόλο με το μήκος της μπάλας στη μέθοδο Ball Walk. Τέλος, για την παράμετρο w , δεν φαίνεται κάποια διαφορά στην κατανομή όποια τιμή και αν επιλέξουμε, εκτός από το 0 φυσικά όπου δεν παίρνουμε κανένα δείγμα.

Ένα σημείο το οποίο παραλείψαμε κατά την δοκιμή της μεθόδου ώστε να απλοποιήσουμε την διαδικασία, αφορά το βήμα διόρθωσης Metropolis–Hastings. Στην 2D περίπτωση δεν υπήρχε κάποιο πρόβλημα, αλλά σε πολλές διαστάσεις είναι μάλλον χρήσιμο να υπάρχει. Η Χαμιλτονιανή, δηλαδή η ενέργεια του σωματιδίου παραμένει σταθερή καθ'όλη την διάρκεια της κίνησης ή τουλάχιστον πρέπει να παραμένει σταθερή. Ωστόσο, από την στιγμή που δεν έχουμε λύσει αναλυτικά τις εξισώσεις αλλά έχουμε εφαρμόσει την Leapfrog μέθοδο, μεταφέρουμε σε κάθε βήμα κάποιο ελάχιστο σφάλμα, τέτοιο ώστε να ισχύει

$$H(x', p') \neq H(x, p)$$

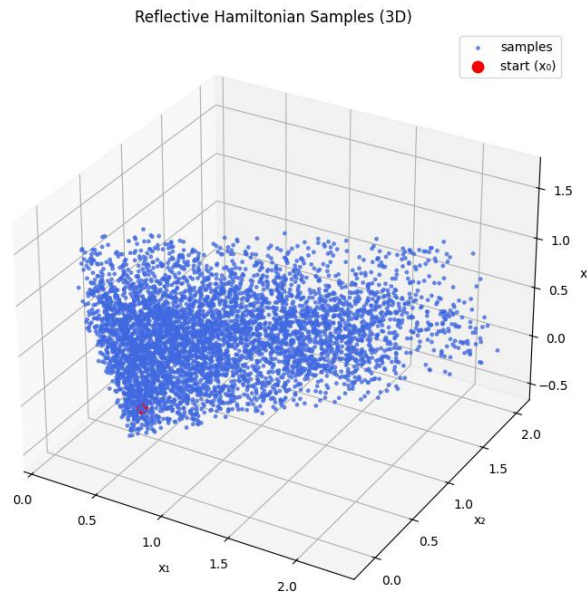
Αυτό, συσσωρευτικά οδηγεί στην ολίσθηση του σωματιδίου σε περιοχές διαφορε-

τικής ενέργειας και εν τέλει στην αλλοίωση της κατανομής. Για τον λόγο αυτό, σε κάθε διακριτό βήμα υπολογίζουμε μια πιθανότητα αποδοχής της μορφής

$$a = \min(1, \exp(-H(x', p') + H(x, p)))$$

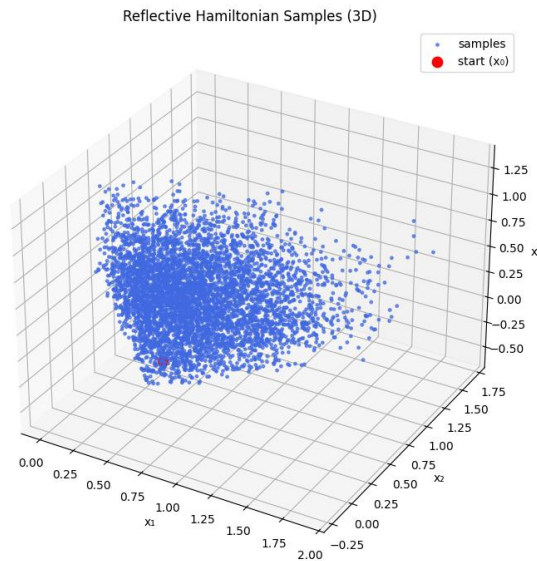
η οποία μας καθορίζει το αν θα αποδεχθούμε το νέο βήμα ή όχι. Σε περίπτωση απόρριψης, παραμένουμε στο ίδιο σημείο. Από την στιγμή που δειγματοληπτούμε την ορμή σε κάθε βήμα, μόνο η θέση έχει ρόλο στην παραπάνω πιθανότητα.

Ας δούμε τώρα μερικές δειγματοληψίες σε 3 διαστάσεις. Σε κανένα σημείο του κώδικα δεν έχει γίνει αναφορά στις διαστάσεις των A και b που ορίζουν το πολύτοπο. Το NumPy εκτελεί τις ίδιες πράξεις για όλα τα μεγέθη των πινάκων εισόδου. Αρκεί λοιπόν να επεκτείνουμε το πολύτόπό μας σε μια επιπλέον διάσταση και να το εισάγουμε στον κώδικα. Δείτε το αποτέλεσμα στις εικόνες 2.40 και 2.41.



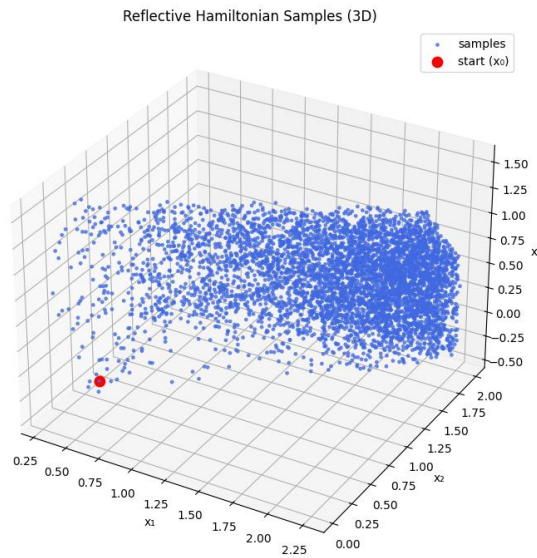
Σχήμα 2.40: Δειγματοληψία με την μέθοδο Reflective Hamiltonian και δυναμικό $f(x) = \frac{1}{2}\|x\|^2$. Αρχικό σημείο το $[0.7, 0, 0.1]$.

Παρατηρείστε πως και στις δυο περιπτώσεις η κατανομή βρίσκεται συγκεντρωμένη γύρω από το αρχικό σημείο που είναι τοποθετημένο κάτω αριστερά στο σχήμα, χωρίς να παραβιάζονται τα όρια του πολύτπου. Μπορούμε λοιπόν τώρα, να ρυθμίζουμε την κατανομή κατά το δοκούν, ανάλογα με το δυναμικό που ορίζουμε. Αν θέλουμε για παράδειγμα να σπρώξουμε τα δείγματα μακριά από το αρχικό σημείο



Σχήμα 2.41: Δειγματοληψία με την μέθοδο Reflective Hamiltonian και δυναμικό $f(x) = e^{x^2}$. Αρχικό σημείο το $[0.7, 0, 0.1]$.

και πάνω αριστερά στο πολύτοπο, αρκεί να προσθέσουμε ένα αρνητικό πρόσημο στην εκθετική συνάρτηση δυναμικού.



Σχήμα 2.42: Δειγματοληψία με την μέθοδο Reflective Hamiltonian και δυναμικό $f(x) = e^{-x^2}$. Αρχικό σημείο το $[0.7, 0, 0.1]$.

2.5 Μετρικές σύγκλισης

Έως τώρα δεν έχουμε μιλήσει καθόλου για το αν τελικά τα δείγματα της αλυσίδας Markov που δίνει ο τυχαίος περίπατος, πράγματι ικανοποιούν τις ιδιότητες που πρέπει να χαρακτηρίζουν μια αλυσίδα Markov. Μετά το πέρας της δειγματοληψίας, τα δείγματά μας είναι πράγματι ανεξάρτητα από την θέση του αρχικού σημείου και επιπλέον έχουν καταλήξει στην στατική κατανομή; Δείτε για παράδειγμα το σχήμα 2.24 με τα αποτελέσματα από την μέθοδο ball walk. Εκεί, ανάλογα με την παράμετρο r της ακτίνας της μπάλας, η ανάμειξη των σημείων βρίσκεται σε διαφορετικό στάδιο σε κάθε περίπτωση. Σε πολλές διαστάσεις όπου δεν υπάρχει οτική επιβαβαίωση των αποτελεσμάτων, πρέπει να είμαστε σε θέση να διακρίνουμε με κάποιον τρόπο αν η ακτίνα που έχουμε επιλέξει έχει δώσει σωστά αποτελέσματα. Για τον σκοπό αυτό, θα χρησιμοποιήσουμε δυο μετρικές που απαντούν στο ερώτημα αυτό τις Potential Scale Reduction Factor ή PSRF και την Effective Sample Size ή ESS.

2.5.1 PSRF

Ο Potential Scale Reduction Factor αποτελεί έναν από τους πιο διαδεδομένους ελέγχους για την σύγκλιση μιας αλυσίδας Markov και κατ'επέκταση μιας μεθόδου MCMC. Η λογική είναι απλή: αν και δεν ξέρουμε το αν μια αλυσίδα έχει καταλήξει στην στατική της κατανομή μετά το πέρας ενός πειράματος, η υλοποίηση της αλυσίδας πολλές φορές υπό τις ίδιες συνθήκες θα πρέπει να δώσει σε κάθε επανάληψη την ίδια κατανομή, η οποία θα είναι προφανώς η στατική. Σε διαφορετική περίπτωση, τα δείγματα που επιστρέφει η κάθε αλυσίδα δεν έχουν προλάβει να διαχυθούν σε όλο τον χώρο και έτσι να ανεξαρτητοποιηθούν όπως συμβαίνει στην στατική κατανομή. Στην ουσία συγκρίνουμε τις διακυμάνσεις κάθε αλυσίδας με τις υπόλοιπες.

Στο παράδειγμα του σχήματος 2.24, στο πάνω αριστερά γράφημα όπου η επιλογή της ακτίνας της μπάλας δειγματοληψίας είναι λανθασμένη, βλέπουμε ξεκάθαρα τα μονοπάτια που έχει σχηματίσει ο τυχαίος περίπατος. Η επανάληψη του αλγορίθμου θα δώσει ξανά τέτοια μονοπάτια, αλλά σε διαφορετικές θέσεις στον χώρο. Αυτή η διαφορά θα φανεί στον υπολογισμό της μετρικής PSRF με αποτέλεσμα να βεβαιωθούμε πως ο περίπατός μας δεν έχει καταλήξει στην στατική του κατάσταση.

Ας δούμε τα πράγματα πιο αναλυτικά. Έστω πως τρέχουμε m παράλληλες αλυσίδες, κάθε μια εκ των οποίων παράγει n δείγματα κάποιας μεταβλητής X . Έστω x_{ij} το j -στο δείγμα της i -στης αλυσίδας, για $i = 1, \dots, m$ και $j = 1, \dots, n$. Για κάθε αλυσίδα ορίζουμε τη μέση τιμή και την διακύμανση:

$$\bar{x}_i = \frac{1}{n} \sum_{j=1}^n x_{ij},$$

$$s_i^2 = \frac{1}{n-1} \sum_{j=1}^n (x_{ij} - \bar{x}_i)^2.$$

Η συνολική μέση τιμή για όλες τις αλυσίδες θα είναι

$$\bar{x}_{..} = \frac{1}{m} \sum_{i=1}^m \bar{x}_i.$$

Ορίζουμε την διακύμανση μεταξύ των αλυσίδων B η οποία μετρά το πόσο μακριά βρίσκονται οι αλυσίδες:

$$B = \frac{n}{m-1} \sum_{i=1}^m (\bar{x}_i - \bar{x}_{..})^2$$

ενώ η μέση τιμή της διακύμανσης για κάθε αλυσίδα W είναι:

$$W = \frac{1}{m} \sum_{i=1}^m s_i^2.$$

Υπολογίζουμε τέλος την οριακή οπίσθια διακύμανση:

$$\hat{V} = \left(1 - \frac{1}{n}\right) W + \frac{1}{n} B.$$

Η μετρική PSRF ορίζεται ως:

$$\hat{R} = \sqrt{\frac{\hat{V}}{W}}.$$

Στην περίπτωση που όλες οι αλυσίδες έχουν συγκλίνει στην ίδια στατική κατανομή, οι δυο παραπάνω διακυμάνσεις είναι ίδιες και άρα ισχύει πως $\hat{R} \approx 1$. Μια

τιμή $\hat{R}$ κοντά στο 1 υποδηλώνει ότι όλες οι αλυσίδες λαμβάνουν δείγματα από την ίδια κατανομή και ότι η αλυσίδα MCMC έχει φτάσει σε στασιμότητα. Τιμές πολύ μεγαλύτερες από 1 υποδηλώνουν ότι μάλλον απαιτούνται περαιτέρω επαναλήψεις. Ακομή και τιμές κοντά στο 1.10 μπορούν να θεωρηθούν οριακά αποδεκτές. Από εκεί και έπειτα όμως, οι αλυσίδες διαφέρουν αρκετά.

2.5.2 ESS

Η Effective Sample Size αποτελεί βασικό δείκτη ποιότητας στην ανάλυση MCMC μεθοδων. Σκοπός της είναι να μετρήσει πόσα από τα δείγματα μιας αλυσίδας μπορούν να θεωρηθούν στατιστικά ανεξάρτητα, λαμβάνοντας υπόψη την συσχέτιση που υπάρχει μεταξύ διαδοχικών δειγμάτων.

Σε μια τέλεια ανεξάρτητη δειγματοληψία, κάθε δείγμα προέρχεται από την κατανομή στόχο χωρίς καμία εξάρτηση από τα προηγούμενα. Στις MCMC μεθόδους, όμως, έχουμε αναλύσει πως κάθε νέο δείγμα εξαρτάται από το προηγούμενο, οπότε τα δεδομένα παρουσιάζουν αυτοσυσχέτιση και η πραγματική πληροφορία που φέρουν είναι μικρότερη από τον αριθμό των συνολικών δειγμάτων.

Θεωρούμε ότι διαθέτουμε N δείγματα $\{x_1, x_2, \dots, x_N\}$ από μια αλυσίδα Markov. Η αυτοσυσχέτιση πρώτης τάξης ορίζεται ως:

$$\rho_k = \frac{\text{Cov}(x_t, x_{t+k})}{\text{Var}(x_t)},$$

όπου ρ_k είναι ο συντελεστής αυτοσυσχέτισης για χρονική υστέρηση k . Όσο μεγαλύτερη είναι η θετική αυτοσυσχέτιση, τόσο μικρότερη είναι η στατιστική ανεξαρτησία των δειγμάτων.

Η μετρική ESS ορίζεται ως:

$$ESS = \frac{N}{1 + 2 \sum_{k=1}^{\infty} \rho_k},$$

όπου το άθροισμα εκτείνεται μέχρι το σημείο που οι αυτοσυσχετίσεις ρ_k καθίστανται ασήμαντες. Αν όλα τα δείγματα ήταν ανεξάρτητα, τότε $\rho_k = 0$ για κάθε $k > 0$ και επομένως $ESS = N$. Αντίθετα, όσο πιο ισχυρή είναι η αυτοσυσχέτιση, τόσο μικρότερο γίνεται το ESS .

Στην πράξη, ο υπολογισμός του ESS γίνεται χρησιμοποιώντας περιορισμένο αριθμό χρονικών υστερήσεων (lags), δηλαδή λαμβάνονται υπόψη μόνο οι πρώτες αυτοσυσχετίσεις, μέχρι το σημείο όπου αυτές καθίστανται αμελητέες. Το ESS υπολογίζεται ξεχωριστά για κάθε διάσταση του διανύσματος παραμέτρων. Για παράδειγμα, σε μια τρισδιάστατη αλυσίδα οι τιμές του ESS μπορεί να διαφέρουν για τις συνιστώσες x_1 , x_2 και x_3 , ανάλογα με το πόσο ελεύθερα κινείται η αλυσίδα σε κάθε κατεύθυνση.

Η ερμηνεία του δείκτη είναι απλή: το ESS ισοδυναμεί με τον αριθμό ανεξάρτητων δειγμάτων που θα παρήγαγαν την ίδια στατιστική ακρίβεια με τα N συσχτισμένα δείγματα της MCMC. Για παράδειγμα, αν μια αλυσίδα των 40,000 δειγμάτων έχει $ESS = 600$ σε κάποια διάσταση, αυτό σημαίνει ότι η πληροφορία που περιέχεται αντιστοιχεί περίπου σε 600 ανεξάρτητα δείγματα.

Είναι σημαντικό να σημειωθεί ότι το ESS μπορεί, σε ορισμένες περιπτώσεις, να υπερβεί το N . Αυτό συμβαίνει όταν οι διαδοχικές τιμές παρουσιάζουν αρνητική αυτοσυσχέτιση, δηλαδή όταν η αλυσίδα κινείται εναλλάξ σε αντίθετες κατευθύνσεις και παράγει αποτελεσματικά πιο «ανεξάρτητα» δείγματα. Η συμπεριφορά αυτή υποδηλώνει πολύ καλή ανάμιξη (mixing) της αλυσίδας.

2.5.3 Εφαρμογή των PSRF και ESS

Θα εφαρμόσουμε ότι παρουσιάσαμε στις προηγούμενες ενότητες σε δυο από τις πιο βασικές μεθόδους δειγματοληψίας, τις Ball Walk και hit-and-run. Ο κώδικας για την κάθε μέθοδο είναι ήδη έτοιμος και δοκιμασμένος (δείτε τα αντίστοιχα κεφάλαια), συνεπώς το μόνο που έχουμε να κάνουμε είναι να προσθέσουμε την λογική υπολογισμού της μετρικής. Όπως πάντα, είμαστε τυχεροί καθώς η δουλειά αυτή έχει ήδη γίνει και περιλαμβάνεται στο πακέτο `arviz` της `python`. Το πακέτο αυτό προσφέρει έτοιμα εργαλεία για τον υπολογισμό της PSRF και της ESS και μάλιστα χρησιμοποιεί έναν πιο εκλεπτυσμένο αλγόριθμο από ότι είδαμε στην θεωρία.

Για το PSRF, μερικές από τις ανώτερες τεχνικές που χρησιμοποιούνται είναι οι εξής: (να κάνω reference to Vehtari et al., 2021) κάθε αλυσίδα χωρίζεται στη μέση, διπλασιάζοντας τον αριθμό των αλυσίδων. Αν κάθε τμήμα της αλυσίδας έχει διαφορετικές μέσες τιμές και αποκλίσεις, η απουσία στασιμότητας γίνεται εμφανής. Στη συνέχεια, δεν χρησιμοποιεί τα ίδια τα δείγματα, αλλά μια κανονικοποιημένη μορφή τους η οποία βοηθά στους υπολογισμούς και αποκλείει ακραίες περιπτώσεις. Τέλος υπολογίζει το $\hat{R}$ πάνω στις νέες τιμές και αλυσίδες. Ομοίως, για την ESS κάποιες λεπτομέρειες διαφέρουν σε σχέση με την απλή εφαρμογή του τύπου που είδαμε.

Σε κάθε περίπτωση, η χρήση του πακέτου είναι αρκετά απλή. Αν ανατρέξετε στον κώδικά μας, εκεί κατά την δειγματοληψία τρέχουμε μια αλυσίδα μια φορά. Ωστόσο, θα πρέπει τώρα να έχουμε στα χέρια μας ένα πλήθος από αλυσίδες. Για τον λόγο αυτό, εισάγουμε όλη της λογική του κάθε τυχαίου περιπάτου σε μια συνάρτηση την οποία μπορούμε στη συνέχεια να καλέσουμε εντός μιας επανάληψης. Η εν λόγω επανάληψη θα καθορίσει τον αριθμό των αλυσίδων που θα τρέξουμε, 4 φαίνεται ένας καλός αριθμός.

Δείτε τώρα, πως θα χρησιμοποιήσουμε το πακέτο και πως θα τυπώσουμε τα αποτελέσματα.

```

1 # -----
2 # Run multiple chains for PSRF
3 # -----
4 n_chains = 4
5 n_steps = 40000
6 all_chains = []
7
8 for i in range(n_chains):
9     start = find_feasible_point()
10    # Uncomment or comment based on the method you prefer to
11    # study.
12    # chain = ball_walk(start, n_steps=n_steps)
13    chain = hit_and_run(start, n_steps=n_steps)
14    all_chains.append(chain)
15    print(f"Chain {i+1} finished, last point: {chain[-1]}")
16
17 # -----
18 # Convert to ArviZ InferenceData
19 # -----
20 chains_array = np.stack(all_chains, axis=0) # shape: (chains,
21    draws, dims)
22 idata = az.from_dict(posterior={"x": chains_array})
23
24 # -----
25 # Compute PSRF and ESS
26 # -----
27 rhat = az.rhat(idata)['x'].values
28 ess = az.ess(idata)['x'].values
29
30 print("\nPSRF (R_hat) per dimension:")
31 for i, val in enumerate(rhat):
32     print(f"  x{i+1}: {val:.4f}")
33
34 print("\nEffective Sample Size (ESS) per dimension:")
35 for i, val in enumerate(ess):
36     print(f"  x{i+1}: {val:.1f}")

```

Listing 2.38: Apply the PSRF and ESS metrics to Ball Walk and Hit-and-Run

methods for the example polytope in 3D.

Η λογική του παραπάνω κώδικα είναι εύκολα κατανοητή. Μάλιστα, είναι ίδια για την κάθε μέθοδο αρκεί να επιλέξετε ποια από τις συναρτήσεις ball-walk ή hit-and-run θέλετε να χρησιμοποιήσετε. Όπως αναφέραμε και προηγουμένως, η κάθε μια εκ των συναρτήσεων αυτών ενθυλακώνει την λογική της αντίστοιχης μεθόδου ακριβώς όπως τις είδαμε κατά την εφαρμογή τους. Ο κώδικας επιστρέφει τα αποτελέσματα σε μια ευδιάκριτη μορφή της επιλογής μας, χωρισμένη ανά συντεταγμένη.

Ως βάση, χρησιμοποιούμε πάντα το γνωστό πολύτοπο που έχουμε δει έως τώρα σε όλα τα παραδείγματα, στις 3 διαστάσεις. Τρέχουμε τον κώδικα για τις εξής περιπτώσεις, δυο φορές για την Ball Walk, μια για την κακή επιλογή ακτίνας μπάλας $r = 0.002$ και μια για την καλή επιλογή ακτίνας μπάλας $r = 0.2$ και τέλος μια φορά για την Hit-and-Run. Για την καλύτερη παρουσίαση, τα αποτελέσματα έχουν συγκεντρωθεί στους παρακάτω δυο πίνακες.

Μέθοδος	Αποδοχή	PSRF _{<i>x</i>1}	PSRF _{<i>x</i>2}	PSRF _{<i>x</i>3}
Ball Walk ($r = 0.002$)	0.988	2.3862	2.3680	1.5582
Ball Walk ($r = 0.2$)	0.833	1.0072	1.0082	1.0009
Hit-and-Run	1.000	1.0001	1.0001	1.0000

Πίνακας 2.1: Ποσοστό αποδοχής και δείκτης σύγκλισης PSRF ανά διάσταση για τις μεθόδους Ball Walk και Hit-and-Run.

Μέθοδος	ESS _{<i>x</i>1}	ESS _{<i>x</i>2}	ESS _{<i>x</i>3}
Ball Walk ($r = 0.002$)	4.9	4.9	7.2
Ball Walk ($r = 0.2$)	549.9	640.5	2976.7
Hit-and-Run	12067.9	14729.8	41937.8

Πίνακας 2.2: Δείκτης (ESS) ανά διάσταση για τις ίδιες μεθόδους. Το *ESS* εκφράζει τον αριθμό σχεδόν ανεξάρτητων δειγμάτων που προέκυψαν από την αλυσίδα.

Ο λόγος που επιλέξαμε τα παραπάνω πειράματα είναι προφανής. Για τις 3 αλυσίδες έχουμε οπτική επιβεβαίωση της σύγκλισης ή μη των τυχαίων περιπάτων στην στατική κατανομή. Συνεπώς, μπορούμε να χρησιμοποιήσουμε τα πειράματα αυτά ώστε να επιβεβαιώσουμε τα ορθά αποτελέσματα των μετρικών. Ας αναλύσουμε τώρα, τα αποτελέσματα που προέκυψαν.

Αρχικά, θυμίζουμε τον αριθμό αποδοχής για τις δυο μεθόδους. Εκ κατασκευής, όλα τα δείγματα στην hit-and-run πέφτουν εντός του πολυτόπου, Αντιθέτως, τα δείγματα κατά την ball walk υπάρχει περίπτωση να βρεθούν εκτός του πολυτόπου όταν το κέντρο της μπάλας βρεθεί κοντά στα σύνορα και άρα ένα ποσοστό της μπάλας

βρίσκεται εκτός του πολυτόπου. Όσο μεγαλύτερη η ακτίνα, τόσο πιο πιθανό να διαπεράσουμε το σύνορο. Για τον λόγο αυτό, για την μικρή ακτίνα ελάχιστα σημεία τελικά βρίσκονται εκτός της μπάλας ενώ για την μεγάλη το ποσοστό ανεβαίνει. Σε εκατοντάδες διαστάσεις θεωρούμαστε τυχεροί αν το ποσοστό αυτό πλησιάζει το 10% με 20%. Σε πολλές διαστάσεις, το ποσοστό του συνόρου σε σχέση με το εσωτερικό του πολυτόπου αυξάνει με συνέπεια περισσότερα δείγματα να βρίσκονται κοντά στα όρια και περισσότερες μπάλες να έχουν μέρος του όγκου τους εκτός του πολυτόπου.

Όσον αφορά τον δείκτη PSRF, τα αποτελέσματα για την κακή επιλογή της ακτίνας $r = 0.002$ είναι απογοητευτικά. Τιμές μεγαλύτερες του 1.1 θεωρούνται απαράδεκτες πόσο μάλλον τιμές μεγαλύτερες του 2. Θυμηθείτε πως έχουμε τρέξει 4 αλυσίδες και έχουμε υπολογίσει τις αποκλίσεις μεταξύ τους. Οι αποκλίσεις αυτές λοιπόν είναι πολύ μεγάλες. Αν κάθε αλυσίδα είχε επιτύχει την κοινή στατική κατανομή, ο δείκτης θα ήταν 1. Αυτό σημαίνει λοιπόν, πως κάθε αλυσίδα έχει δώσει και διαφορετική κατανομή των δειγμάτων της πράγμα που μας οδηγεί στο συμπέρασμα πως η αλυσίδα δεν έχει προλάβει να φτάσει στην στατική κατανομή βάσει των παραμέτρων $n = 40000$ και $r = 0.002$. Στο σχήμα 2.24 (σε 2D) βλέπουμε τον λόγο. Ο τυχαίος περίπατος δεν είναι και τόσο τυχαίος αφού διακρίνουμε ξεκάθαρα το μονοπάτι που έχει σχηματισθεί. Σε κάθε μια από τις 4 αλυσίδες έχουμε λαβει διαφορετικά μονοπάτια τα οποία ελάχιστη σχέση έχουν μεταξύ τους. Αντιθέτως, στην καλή επιλογή $r = 0.2$ και στην hit-and-run ο δείκτης είναι τέλειος. Οι αλυσίδες δεν έχουν καμία απόκλιση αφού όλες έχουν καταλήξει στην στατική τους κατανομή.

Τέλος, για το ESS τα καλύτερα αποτελέσματα δίνει η hit-and-run επιβεβαιώνοντας έτσι πως αποτελεί την καλύτερη επιλογή σε σχέση με την ball walk. Για την κακή επιλογή ακτίνας, όλα τα 40000 δείγματα περιλαμβάνουν την ίδια πληροφορία με μόλις 5 ή 7 τυχαία δείγματα που θα είχαμε πάρει απ'ευθείας από μια ομοιόμορφη κατανομή στον όγκο του πολυτόπου. Προφανώς, αφού ο περίπατος προχωρά ελάχιστα σε κάθε βήμα, τα περισσότερα σημεία βρίσκονται το ένα πάνω στο άλλο με ισχυρή εξάρτηση μεταξύ τους. Κατά αυτόν τον τρόπο, μόλις μερικά ανεξάρτητα σημεία είναι ικανά να περιγράψουν την συμπεριφορά αυτή. Στην καλή ακτίνα υπάρχει τεράστια βελτίωση, ακόμη και έτσι όμως πολλά από τα σημεία σχετίζονται μεταξύ τους. Η βέλτιστη μετρική εντοπίζεται για την hit-and-run όπου μάλιστα ο δείκτης έχει τιμή μεγαλύτερη από τον αριθμό των δειγμάτων για την z κατεύθυνση. Εξηγήσαμε πιθανές αιτίες για την συμπεριφορά αυτή στην τελευταία παράγραφο της μεθόδου ESS.

Να σημειώσουμε τέλος, πως καμία από τις δυο μετρικές δεν επιβεβαιώνει το αν η τελική κατανομή των δειγμάτων είναι η ομοιόμορφη ή όχι. Το μόνο που υποδεικνύουν είναι αν η αλυσίδα έχει συγκλίνει στην στατική της κατανομή. Το ποιά είναι η στατική αυτή κατανομή δεν περιλαμβάνεται στον ορισμό των δεικτών.

2.5.4 Global Radial Scaling Diagnostic - KS test

We consider a convex polytope

$$\mathcal{P} = \{x \in \mathbb{R}^d : Ax \leq b\},$$

and a collection of samples $\{x_i\}_{i=1}^N \subset \mathcal{P}$, typically produced by a Markov chain Monte Carlo (MCMC) method.

The goal of the diagnostic is to assess whether the samples are uniformly distributed in volume *along rays emanating from a chosen interior point*. This diagnostic does not require computing the volume of $\mathcal{P}$.

Choice of Center

We define a reference point as the empirical mean of the samples,

$$c = \frac{1}{N} \sum_{i=1}^N x_i, \quad (2.11)$$

which corresponds to the code:

```
VT center = samples.rowwise().mean();
```

The center c is not required to be a sample itself; it serves only as an interior reference point. If c lies on or outside the boundary of $\mathcal{P}$, the diagnostic becomes invalid.

Re-centering the Polytope

Writing each sample as

$$x = c + q,$$

the polytope constraints become

$$Aq \leq b - Ac.$$

This is implemented in the code as:

```
VT b_shifted = b - A * center;
```

This change of variables expresses the polytope in coordinates centered at c , which enables a ray-based analysis.

Radial Coordinate

For each sample x_i , we define a radial coordinate $r_i \in [0, 1]$ as follows. Let

$$q_i = x_i - c,$$

and consider scaling the ray $c + tq_i$, $t > 0$, until it intersects the boundary of the polytope. The maximal admissible scaling factor satisfies

$$r_i = \max_k \frac{(A_k q_i)}{b_k - (A_k c)}.$$

In code, this is computed as:

```
VT q = samples.col(i) - center;
VT u = A * q;
r_max = max_k (u[k] / b_shifted[k]);
```

The value r_i measures how far the sample lies along its *own ray*, relative to the boundary in that direction:

- $r_i = 0$: the center c ,
- $r_i = 1$: the boundary of $\mathcal{P}$ in the direction of $x_i - c$.

Importantly, r_i is *not* a Euclidean distance.

Probability Integral Transform

If the samples are uniformly distributed in $\mathcal{P}$, then the volume of the set

$$\{x \in \mathcal{P} : r(x) \leq r\}$$

scales as r^d , independent of the shape of $\mathcal{P}$. Therefore, the transformed variable

$$u_i = r_i^d \quad (2.12)$$

is uniformly distributed on $[0, 1]$.

This transformation is implemented as:

```
uvals.push_back(std::pow(r, dim));
```

Kolmogorov–Smirnov Test

After optional thinning to reduce autocorrelation, a Kolmogorov–Smirnov (KS) test is applied to the empirical distribution of $\{u_i\}$ against the uniform distribution on $[0, 1]$.

This tests the null hypothesis:

$$H_0 : u_i \sim \text{Uniform}(0, 1),$$

which is equivalent to correct radial scaling of the samples.

Volume Shell Diagnostics

For visualization, we define volume shells corresponding to expected coverage levels

$$\alpha_k = 0.1, 0.2, \dots, 1.0,$$

with corresponding radial thresholds

$$r_k = \alpha_k^{1/d}.$$

The observed fraction of samples satisfying $r_i \leq r_k$ is compared to α_k . These shells represent *fractions of volume*, not Euclidean balls.

What the Diagnostic Detects

The test fails when the radial distribution of samples is incorrect, including:

- bias toward the center,
- over-representation near the boundary,
- direction-dependent radial truncation,
- burn-in contamination.

In these cases, the empirical distribution of u_i deviates from uniformity.

Failure Modes of the Diagnostic

The diagnostic is invariant under uniform radial contraction:

$$x = c + \alpha(x_{\text{true}}), \quad 0 < \alpha < 1.$$

If samples are uniformly distributed within a homothetic copy of $\mathcal{P}$, the distribution of u_i remains uniform and the test passes.

Consequently, the diagnostic does *not* detect:

- confinement to a strict subset of the polytope,
- missing angular regions,
- poor directional mixing (e.g., coordinate hit-and-run in elongated polytopes),
- multimodality with correct radial statistics.

Role of the Maximum over Constraints

A common source of confusion is the appearance of a maximum over constraints in the computation of the radial coordinate. One might initially expect that the diagnostic measures the Euclidean distance from the center to the boundary along the direction of the sample. This is not the case.

For a given sample $x = c + q$, each inequality

$$A_k x \leq b_k$$

induces a candidate intersection time along the ray $c + tq$,

$$t_k = \frac{b_k - A_k c}{A_k q}, \quad \text{for } A_k q > 0.$$

The ray intersects the boundary of the polytope at the *first* constraint that becomes active:

$$t_{\max} = \min_k t_k.$$

The radial coordinate used in the diagnostic is therefore

$$r = \frac{1}{t_{\max}} = \max_k \frac{A_k q}{b_k - A_k c}.$$

Thus, the maximum does not correspond to scanning multiple directions; it selects the most restrictive constraint along the *single direction* defined by the sample.

Illustrative Example: Long Thin Rectangle

Consider the two-dimensional rectangle

$$\mathcal{P} = \{(x, y) : |x| \leq 1, |y| \leq 100\},$$

with center $c = (0, 0)$.

Suppose a sampling algorithm produces samples uniformly distributed in

$$x \in [-1, 1], \quad y \in [-10, 10],$$

i.e., the chain fully explores the short dimension but only a small fraction of the long dimension.

For a sample $x = (x, y)$, the radial coordinate is

$$r = \max\left(|x|, \frac{|y|}{100}\right).$$

Since $|y|/100 \leq 0.1$ for all samples while $|x| \in [0, 1]$, the short-side constraint dominates and

$$r = |x| \quad \text{almost surely.}$$

Consequently, the radial distribution is identical to that of samples uniformly distributed in the full rectangle, despite the fact that the chain explores only 10% of the available extent in the long direction. The volume shell diagnostics therefore exhibit negligible deviation from their expected values.

Interpretation

This example illustrates a fundamental limitation of the diagnostic: the radial coordinate is determined by the *first constraint encountered along the ray*, and constraints that are not active in that direction do not influence the statistic. As a result, confinement along long directions of the polytope may remain undetected if shorter directions are fully explored.

In summary, the diagnostic verifies correct radial scaling with respect to the most restrictive constraint along each ray, but it does not detect missing volume associated with constraints that are never active.

2.6 Polytope Rounding

Σε περιπτώσεις όπου τα πολύτοπα στα οποία θέλουμε να εκτελέσουμε δειγματοληψία δεν έχουν ένα ωραίο σχήμα όπου ένας υπερκύβος ή ένα simplex, τότε οι αλγόριθμοι που είδαμε έως τώρα αρχίζουν να εμφανίζουν πορβλήματα σύγκλισης. Τέτοια πολύτοπα μπορεί να είναι εξαιρετικά επιμηκυμένα ή λεπτά σε ορισμένες κατευθύνσεις ή να έχουν οποιαδήποτε άλλη ιδιομορφία, γεγονός που καθιστά τη δειγματοληψία και τους τυχαίους περιπάτους πολύ αργούς.

Όταν ένα σώμα λοιπόν, δεν είναι καλά ισορροπημένο γεωμετρικά, οι τυχαίες κινήσεις χρειάζονται πολλές επαναλήψεις για να καλύψουν ομοιόμορφα τον χώρο. Η στρογγυλοποίηση (rounding) μέσω ενός αφινικού μετασχηματισμού που καθιστά το σώμα σχεδόν ισοτροπικό (δηλαδή φαίνεται ίδιο προς όποια κατεύθυνση και αν κοιτάξουμε) εξασφαλίζει ότι όλες οι κατευθύνσεις έχουν παρόμοια διασπορά, μειώνοντας δραστικά τον χρόνο μίξης των αλυσίδων Markov και καθιστώντας εφικτή τη δειγματοληψία και μετέπειτα βέβαια τον υπολογισμό όγκου και τη βελτιστοποίηση σε πολυδιάστατους χώρους με πολυωνυμική πολυπλοκότητα.

Στόχος μας είναι τώρα, να φέρουμε ένα κυρτό σώμα $K \subset \mathbb{R}^n$ σε σχεδόν ισοτροπική μορφή μέσω ενός αφινικού μετασχηματισμού. [1]

Η βασική ιδέα είναι η εξής: επιλέγουμε ένα μετασχηματισμό $F(x) = A(x - \mu)$ ώστε το μετασχηματισμένο σώμα

$$\tilde{K} = F(K)$$

να είναι καλά ισορροπημένο γεωμετρικά, δηλαδή να ικανοποιεί:

$$B_n \subseteq \tilde{K} \subseteq O(\sqrt{n})B_n.$$

που σημαίνει πως ταιριάζει ακριβώς μεταξύ μιας μικρής σφαίρας μονάδας και μιας μεγαλύτερης σφαίρας με ακτίνα ανάλογη προς $\sqrt{n}$.

Η συγκεκριμένη διαδικασία που θα δούμε βασίζεται στο [Dikin Walk](#), ο οποίος είναι αφινικά αναλλοίωτος και επομένως δεν απαιτεί πολλαπλές φάσεις στρογγυλοποίησης.

Ένα σώμα $K \subset \mathbb{R}^n$ με ομοιόμορφη κατανομή π ονομάζεται ισοτροπικό όταν:

$$\mathbb{E}_\pi[X] = 0, \quad \mathbb{E}_\pi[XX^\top] = I_n.$$

μέση τιμή και διασπορά. Αν $X = (X_1, \dots, X_n)$, τότε $XX^\top$ είναι ένας $n \times n$ πίνακας με στοιχεία $X_i X_j$ και ο $\mathbb{E}_\pi[XX^\top]$ δίνει ένα μέσο όρο των τιμών αυτών.

Αντίστοιχα, K είναι φ -σχεδόν ισοτροπικό αν:

$$(1 - \varphi)I_n \preceq \mathbb{E}_\pi[XX^\top] \preceq (1 + \varphi)I_n,$$

όπου $\varphi \in (0, 1)$.

Η μεταφορά ενός σώματος σε ισοτροπική θέση επιτυγχάνεται με τον αφινικό μετασχηματισμό:

$$\tilde{K} = \{\Sigma^{-1/2}(x - \mu) : x \in K\},$$

όπου μ και Σ είναι ο μέσος όρος και η συνδιακύμανση της ομοιόμορφης κατανομής πάνω στο K .

Ας θεωρήσουμε ότι το σώμα K ορίζεται μέσω ενός φραγμού $F(x)$, για παράδειγμα

$$F(x) = - \sum_{i=1}^m \log(b_i - a_i^\top x),$$

για ένα πολύτοπο $K = \{x : Ax \leq b\}$. Η παραγωγή δεύτερης τάξης δίνει τη μετρική του σώματος:

$$H(x) = \nabla^2 F(x).$$

Για κάθε $x \in \text{int}(K)$, ορίζουμε ελλειψοειδές Dikin:

$$D_x(r) = \{y : (y - x)^\top H(x)(y - x) \leq r^2\}.$$

Ο τυχαίος περίπατος Dikin Walk είναι αφινικά αναλλοίωτος και συγκλίνει στην ομοιόμορφη κατανομή του σώματος K .

Από την στατιστική, για T δείγματα $X_1, \dots, X_T \sim \pi$, εκτιμούμε:

$$\bar{X} = \frac{1}{T} \sum_{t=1}^T X_t, \quad \hat{\Sigma} = \frac{1}{T} \sum_{t=1}^T (X_t - \bar{X})(X_t - \bar{X})^\top.$$

Ο αφινικός μετασχηματισμός για τη στρογγυλοποίηση είναι:

$$F(x) = \hat{\Sigma}^{-1/2}(x - \bar{X}),$$

οπότε το νέο σώμα είναι:

$$\tilde{K} = \{F(x) : x \in K\}.$$

Για τη διασφάλιση της σχεδόν ισοτροπικής θέσης (φ -ισοτροπικότητα), απαιτείται πλήθος δειγμάτων:

$$T = \tilde{O}(n^3\nu),$$

όπου ν είναι η παραμετρική πολυπλοκότητα του φραγμού (π.χ. ίση με τον αριθμό περιορισμών για ένα πολύτοπο).

Η απόδειξη βασίζεται στην ανισότητα Bernstein για Μαρκοβιανές αλυσίδες (Paulin, 2015) και στην ύπαρξη φάσματος $\gamma = \Omega(1/(n\nu))$ για τον περίπατο Dikin (Narayanan, 2016).

Με πιθανότητα τουλάχιστον $1 - \eta$:

$$\|\hat{\Sigma}^{-1/2}(\hat{\Sigma} + (\bar{X} - \mu)(\bar{X} - \mu)^\top)\hat{\Sigma}^{-1/2} - I\| \leq \varphi.$$

Η συνολική πολυπλοκότητα είναι:

- Για πολυεδρικό σώμα με m περιορισμούς: $\nu = m$.
- Υπολογισμός εσσιανής $H(x) = A^\top S_x^{-2} A$ με κόστος $O(mn^{\omega-1})$.
- Συνολικό κόστος: $\tilde{O}(\zeta n^3\nu)$, όπου ζ το κόστος υπολογισμού της εσσιανής.

Η μέθοδος στρογγυλοποίησης με τον τυχαίο περίπατο Dikin παρέχει μια αφινικά αναλλοίωτη διαδικασία που επιτυγχάνει ισοτροπική θέση με πολυωνυμική πολυπλοκότητα. Σε αντίθεση με τις κλασικές μεθόδους (Lovász–Vempala, Kannan κ.ά.), απαιτεί μόνο μία φάση δειγματοληψίας και αξιοποιεί τις γεωμετρικές ιδιότητες του φραγμού. Σε κάθε περίπτωση δίνει την βασική ιδέα για τον πως επιτυγχάνεται η στρογγυλοποίηση ενός πολύτοπου.

2.6.1 Εφαρμογή του Polytope Rounding

Ας εφαρμόσουμε εδώ μια εξαιρετικά απλοποιημένη εκδοχή της στρογγυλοποίησης ενός πολυτόπου. Επιλέγουμε λοιπόν, έναν 3D κύβο τον οποίο όμως φροντίζουμε να διαστρεβλώσουμε ώστε να επιμηκυνθεί προς μια κατεύθυνση. Στη συνέχεια, θα δούμε πως θα μετασχηματίσουμε το πολύτοπο σε ισοτροπικό μέσω της δειγματοληψίας στο εσωτερικό του και στη συνέχεια εφαρμόζοντας την λογική του αφινικού μετασχηματισμού.

Ο κώδικας είναι ο παρακάτω:

```

1 # Start by sampling uniformly from a 3D cube
2 N = 8000
3 X = np.random.uniform(-1, 1, size=(N, 3))
4
5 # Then, apply a linear transformation to distort the cube
6 A = np.array([[2.0, 0.8, 0.0],
7               [0.0, 1.5, 0.5],
8               [0.0, 0.0, 0.5]])
9 X_distorted = X @ A.T
10
11 # Consider now that these samples belong to our unknown polytope
12 # that we need to round. We need to compute mean
13 # and covariance of distorted body
14
15 mu = np.mean(X_distorted, axis=0)
16 Sigma = np.cov(X_distorted.T)
17
18 # Finally, compute the rounding transform (whitening)
19 eigvals, eigvecs = np.linalg.eigh(Sigma)
20 Sigma_inv_sqrt = eigvecs @ np.diag(1.0 / np.sqrt(eigvals)) @
    eigvecs.T
21
22 Y = (X_distorted - mu) @ Sigma_inv_sqrt.T # rounded samples

```

Listing 2.39: Simple rounding of a distorted 3D cube.

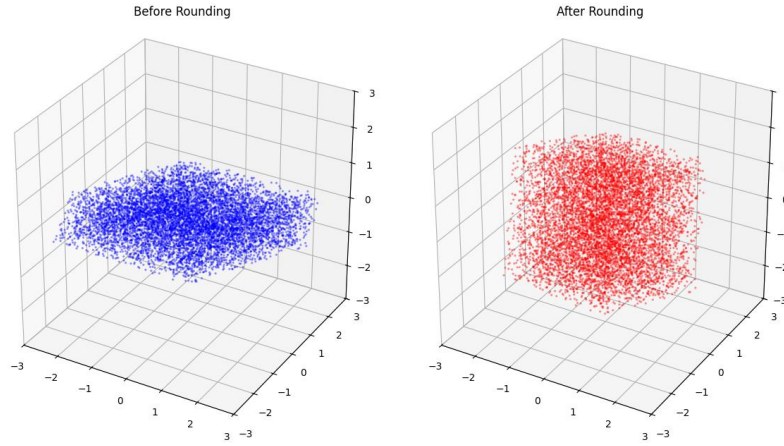
Το NumPy περιλαμβάνει έτοιμες εντολές για τον υπολογισμό των στατιστικών μεγεθών. Αναλυτικά, οι εξισώσεις που εφαρμόζουμε διαδοχικά είναι οι παρακάτω:

$$\mu = \frac{1}{n} \sum_{i=1}^n X_{\text{distorted}}^{(i)}$$

$$\Sigma = \frac{1}{n-1} \sum_{i=1}^n (X_{\text{distorted}}^{(i)} - \mu)(X_{\text{distorted}}^{(i)} - \mu)^{\top}$$

$$\begin{aligned}\Sigma &= V \Lambda V^\top, \quad \Lambda = \text{diag}(\lambda_1, \dots, \lambda_d) \\ \Sigma^{-\frac{1}{2}} &= V \text{diag}\left(\frac{1}{\sqrt{\lambda_1}}, \dots, \frac{1}{\sqrt{\lambda_d}}\right) V^\top \\ Y &= (X_{\text{distorted}} - \mu) \Sigma^{-\frac{1}{2}}\end{aligned}$$

Το αποτέλεσμα μπορούμε να το σχεδιάσουμε και φαίνεται όπως στην παρακάτω εικόνα:



Σχήμα 2.43: Το αποτέλεσμα της στρογγυλοποίησης ενός παραμορφωμένου πολύτοπου. Το αποτέλεσμα είναι ισοτροπικό. Αν και φαίνεται τα σχήματα να διαφέρουν αρκετά, σε ανώτερες διαστάσεις η διαφορά στον χώρο που καταλαμβάνουν τα δυο πολύτοπα μειώνεται συνεχώς.

2.7 Null Space Technique

To enable volumetric sampling on Netlib linear programming instances, we must first address the geometric degeneracy caused by equality constraints.

2.7.1 The Problem: Zero Volume

A standard Netlib problem is defined by a system of linear equations and bounds:

$$\mathbf{Ax} = \mathbf{b}, \quad \mathbf{lb} \leq \mathbf{x} \leq \mathbf{ub} \quad (2.13)$$

where $\mathbf{x} \in \mathbb{R}^n$ is the decision variable vector, and $\mathbf{A}$ is an $m \times n$ constraint matrix (with $m < n$).

Geometrically, the equality constraint $\mathbf{Ax} = \mathbf{b}$ restricts the feasible region to an affine subspace within the high-dimensional ambient space. As a result, the n -dimensional volume of the polytope is strictly zero. Volumetric samplers (such as Volesti) require a full-dimensional body with positive interior volume to correctly define and sample from a probability distribution.

2.7.2 The Solution: Parameterization

To extract the full-dimensional “core” of the polytope, we project the problem into its *Null Space*. We decompose the variable $\mathbf{x}$ into a fixed component and a free component:

$$\mathbf{x} = \mathbf{x}_0 + \mathbf{Z}\mathbf{v} \quad (2.14)$$

The components are defined as follows:

- **$\mathbf{x}_0$ (Particular Solution):** A strictly feasible point that satisfies $\mathbf{Ax}_0 = \mathbf{b}$ and respects the bounds $\mathbf{lb} \leq \mathbf{x}_0 \leq \mathbf{ub}$. We identify this using Linear Programming to ensure the origin of our new coordinate system lies strictly inside the polytope.
- **$\mathbf{Z}$ (Null Space Basis):** An orthonormal matrix where $\mathbf{AZ} = \mathbf{0}$. The columns of $\mathbf{Z}$ represent the “free directions” in which we can move without violating the equality constraints.
- **$\mathbf{v}$ (Reduced Coordinates):** The new variable vector. If $\mathbf{A}$ has rank m , then $\mathbf{v} \in \mathbb{R}^{n-m}$. This effectively reduces the dimension of the problem from n to $n - m$.

2.7.3 Transforming the Constraints

We substitute the parameterization $\mathbf{x} = \mathbf{x}_0 + \mathbf{Z}\mathbf{v}$ into the original bound constraints ($\mathbf{lb} \leq \mathbf{x} \leq \mathbf{ub}$):

$$\mathbf{lb} \leq \mathbf{x}_0 + \mathbf{Z}\mathbf{v} \leq \mathbf{ub} \quad (2.15)$$

We rearrange this inequality into the standard H-polytope format ($\mathcal{A}\mathbf{v} \leq \mathbf{b}$) required by the sampling algorithms. The constraints decompose into:

$$\mathbf{Z}\mathbf{v} \leq (\mathbf{ub} - \mathbf{x}_0) \quad (2.16)$$

$$-\mathbf{Z}\mathbf{v} \leq (\mathbf{x}_0 - \mathbf{lb}) \quad (2.17)$$

This yields the final system in block matrix form:

$$\begin{bmatrix} \mathbf{Z} \\ -\mathbf{Z} \end{bmatrix} \mathbf{v} \leq \begin{bmatrix} \mathbf{ub} - \mathbf{x}_0 + \epsilon \\ \mathbf{x}_0 - \mathbf{lb} + \epsilon \end{bmatrix} \quad (2.18)$$

2.7.4 Numerical Conditioning

To ensure robust sampling stability for ill-conditioned Netlib instances, we applied three pre-conditioning steps during extraction:

1. **Feasibility Check:** We solve a Phase-1 LP to find an $\mathbf{x}_0$ that is strictly interior relative to the bounds, ensuring the right-hand side vector is non-negative.
2. **Epsilon Padding:** We introduced a relaxation factor $\epsilon = 10^{-7}$ to the bounds. This prevents “zero-width” facets that cause numerical instability in rounding algorithms (e.g., John Position or Maximum Volume Ellipsoid).
3. **Infinity Handling:** Unbounded directions (where $ub_i = \infty$) were capped at 10^{10} to ensure the polytope is compact (closed and bounded).

The result is a full-dimensional polytope in $\mathbb{R}^{n-m}$ that preserves the exact combinatorial structure of the original problem while being amenable to Markov Chain Monte Carlo (MCMC) sampling.

Κεφάλαιο 3

Volesti

Όλες οι μέθοδοι δειγματοληψίας που είδαμε στην θεωρία έχουν υλοποιηθεί σε κάποιο βαθμό από την κοινότητα και περιλαμβάνονται σε διάφορα πακέτα υπολογιστικής γεωμετρίας παράλληλα με άλλα θέματα. Οι vanilla υλοποιήσεις που είδαμε στην θεωρία, αν και προσφέρουν μια πολύ καλή εισαγωγή στην φιλοσοφία και στην χρήση κάθε μεθόδου και είναι χρήσιμες παιδαγωγικά, εντούτοις τους λείπει η αυστηρότητα και η στοιβαρή υλοποίηση που απαιτούνται στην ακριβή, αποτελεσματική και αποδοτική δειγματοληψία σε όλα τα πιθανά πολύτοπα και για εκατοντάδες διαστάσεις. Για τον σκοπό αυτό, θα φύγουμε από το sandbox περιβάλλον που δουλεύουμε έως τώρα και θα μεταβούμε στον πραγματικό κόσμο όπου η υλοποίηση ενός αλγορίθμου δειγματοληψίας πρέπει να αντιμετωπίζει όλες τις πιθανές εισόδους στο κατά το δυνατόν συντομότερο χρόνο και χώρο. Την υλοποίηση αυτή προσφέρουν τα προαναφερθέντα πακέτα τα οποία θα μελετήσουμε.

3.1 Volesti Library

Volesti is an open-source C++ library for computational geometry, designed to perform high-dimensional geometric computations such as volume approximation, random sampling, and optimization over convex bodies. It supports various convex representations—including H-polytopes, V-polytopes, zonotopes, and spectrahedra—and implements efficient randomized algorithms like Hit-and-Run and Ball Walk for sampling and volume estimation. Volesti is widely used in fields such as convex optimization, probabilistic inference, and computational geometry research, and it also provides convenient interfaces for R and Python, making its advanced geometric capabilities accessible across multiple programming environments. You can find the

library here [Volesti GitHub Repository](#). The readme file contains all the information you might need, like author names, documentation and tutorials.

3.2 Installing Volesti

Volesti includes sophisticated algorithms for sampling in any dimension and for any polytope (sampling is quite useful in volume calculations, the main focus of the package, but we don't cover that here). Thus, we can experiment with different sampling algorithms and compare with the vanilla versions in regard to time and efficiency. The first order of business is to install the library itself.

Now, despite what you might think, installing the library is a huge task requiring a lot of patience. Should you ever feel the need to break your PC don't. Take a timeout and continue later. Your mental health is more important than any project.

3.2.1 Failed attempts

Unfortunately, the tutorial provided by Volesti regarding the installation is quite brief and not very useful, especially if this is your first time you attempt something like this. Another thing to worry about is your operating system. If you are on Windows, you can forget about installing the library. I personally made various failed attempts before I moved on. In the beginning I tried using Visual Studio Community 2022 and the rather useful option to clone a repository. VS community comes with integrated support for CMake so it immediately identified the CMakeLists.txt file and run it. Several problems arised while trying to build the files by clicking build-all (about 800 errors). Many of these problems came down to CMakeLists.txt file expecting linux style paths compared to windows ones, a problem that is quite common as you will see.

Since VS community acts weird, we change our plan to completing things manually. The repository clone was placed in a separate folder and custom tools were used to build it. MSYS2 was chosen, since it provide linux like commands for windows. Good thing is that cmake, vcpkg and boost (required libraries for volesti) where already installed and tested form our CGAL days so we added everything in the CMakeLists.txt file and did cmake .. Well, that was the end of it, not even the CMakeLists could be properly translated without an error. Thus, there was only one option left to explore, an option recommended by every computer geek out there: install Linux.

3.2.2 Successful attempt

The last option before formatting a disk and installing Linux was to install Windows Subsystem Linux - WSL (also recommended in the official tutorial, I credit them for that). Now, for the comfort of all those that will follow, I will provide step by step instructions on how to download and build the volesti library using WSL in Windows systems. Believe me, this will save you a lot of time and from the need to talk for hours with an AI. So follow along.

1. Make sure you have at least 12GB free. Install WSL. Open Powershell as Administrator and run:

```
wsl -- install
```

2. After it finishes, Ubuntu will run automatically (or open it yourself if it doesn't - just select Ubuntu in the start menu). It will ask you to create an account. Type a username and a password. After you are all set up, type these commands:

```
sudo apt update  
sudo apt upgrade -y
```

3. Install development tools, Git, CMake, GCC, and Boost. Just type:

```
sudo apt install -y build-essential git cmake libboost-all-dev  
sudo apt install -y ninja-build
```

4. Time to clone Volesti. Type:

```
cd ~  
git clone https://github.com/GeomScale/volesti.git  
cd volesti
```

5. You should now find yourself in your home folder inside volesti. Create a build folder:

```
mkdir build && cd build
```

and type:

```
cmake ..
```

Ideally you will see messages such as

```
-- Autodiff library not found locally, downloading it.
-- Eigen library not found locally, downloading it.
-- lp_solve library not found locally, downloading it.
```

meaning the system will download all necessary libraries.

6. Now, use `ninja` to build the library. This task is quite demanding in resources like processing power and memory. The first time I tried, the virtual environment crashed immaterially. To avoid this, we will allocate more resources to the Ubuntu virtual environment. Go to

```
C:\Users\<YourUsername>
```

Create a new `.txt` file containing these lines

```
[ws12]
memory=8GB
processors=4
```

Rename the file `.wslconfig` and ignore the warning message. Restart WSL. I do hope you have at least 5 GB of free RAM when you start building. I have 16GB on my PC and it jumped from 8 used to 12 used at some point during the process.

7. Now go back to the build folder and type:

```
ninja -j4
```

It will start building the library. This will take some time. If you crash at any point, try reallocating more resources or using less cores (the integer in the end is the cores used). Ninja is supposed to be better than `make`, that's why we used it here. The last line I saw was:

```
[46/46] Linking CXX executable test_internal_points
```

8. Now, inside the build folder we created earlier, you will see a lot of files looking like this:

```
logconcave_sampling_test
matrix_sampling_test
mcmc_diagnostics_test
...
```

These are tests that you can run by typing (while inside the build folder):

```
./mcmc_diagnostics_test
```

The result we got was:

```
[doctest] doctest version is "2.4.9"
[doctest] run with "--help" for options
--- Testing psrf on Billiard Walk and H-cube10
psrf = 1.00269
--- Testing psrf on Billiard Walk and H-cube10
univariate psrf = 1.00005 0.999911 0.999988 0.999911 0.999943 1.00023 1.00007 0.99
--- Testing psrf on Billiard Walk and H-cube10
interval psrfs = 1.00051 1.00003 1.0002 1.0007 1.00025 1.00092 1.00103 1.00041 1.00
--- Testing psrf on Billiard Walk and H-cube10
ess = 9880.89 10000 9887.83 9794.59 9943.22 7775.64 10000 10000 9703.68 9467.23
--- Testing psrf on Billiard Walk and H-cube10
geweke = 1
--- Testing psrf on Billiard Walk and H-cube10
raftery walk length to get independent samples, 1st coordinate = 1
=====
[doctest] test cases: 6 | 6 passed | 0 failed | 0 skipped
[doctest] assertions: 6 | 6 passed | 0 failed |
[doctest] Status: SUCCESS!
```

3.3 Running the tutorial example

Let's try now to run our first example, the one given in the Volesti tutorial. In the tutorial you can find some cpp code and a CMakeLists.txt and you are encouraged to copy paste them. The first question that is not answered is where exactly are we supposed to save these files? Inside the volesti library we just built or somewhere else and does this question even matter? Well, since we are left with the choice, I personally like good organization so we create a new folder in our home folder called volesti-sampling. Navigate to this folder and create the two files mentioned in the tutorial. Since I am kinda naive about Linux, I used nano but I am sure there are better solutions out there.

The procedure is known. Create a build folder, navigate to it and do cmake .. and then make. If it worked, allow me to congratulate you for making the impossible possible. We were not so lucky however as we got an error that:

```
CMake Error at CMakeLists.txt:6 (include):
  include could not find requested file:
```

```
.../.../external/cmake-files/Eigen.cmake
```

Great, there seems to be a problem with finding some of the libraries. This is where a better tutorial could be of assistance. Anyway, we ask an AI for a better CMakeLists that will run without problems. We eagerly follow its advice and end up with other mistakes:

```
fatal error: lp_lib.h: No such file or directory
fatal error: minimum_ellipsoid/khach.h: No such file or directory
...
```

All the headers mentioned above belong in the external categories used by Volesti to do a number of difficult calculations. It seems, that these headers are not correctly linked. To this end, we manually plug in ALL external header files in the CMakeLists.txt file so as to avoid further trouble. This is how our CMakeLists looks like:

```
cmake_minimum_required(VERSION 3.10)
project(my_sampler)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

set(VOL_ESTI_DIR "/home/chrthegreat/volesti")

add_definitions(-DDISABLE_NLP_ORACLES)

include_directories(
  ${VOL_ESTI_DIR}/include
  ${VOL_ESTI_DIR}/external/_deps/eigen-src
  ${VOL_ESTI_DIR}/external/Spectra/include
  ${VOL_ESTI_DIR}/external/PackedCSparse
  ${VOL_ESTI_DIR}/external/cmake-files/_deps/autodiff-src
  ${VOL_ESTI_DIR}/external/_deps/lpsolve-src
  ${VOL_ESTI_DIR}/external
  ${VOL_ESTI_DIR}/external/minimum_ellipsoid
  ${VOL_ESTI_DIR}/external/cmake-files
  ${VOL_ESTI_DIR}/external/ChebTools
```

```
)  
  
add_executable(my_sampler my_sampler.cpp)  
  
target_link_libraries(my_sampler  
    ${VOL_ESTI_DIR}/build/liblp_solve.a  
    ${VOL_ESTI_DIR}/build/libQD_LIB.a  
)
```

Finally, go back to the build folder, type:

```
rm -rf *
```

to delete everything and do again cmake .. It worked for us, I hope you are lucky too. Then type, make and enjoy the fruits of your troubles. Typing:

```
./my_sampler
```

returns the result expected by the tutorial ie:

```
Polytope:  
 6 3 double  
1 0 0 <= 1  
0 1 0 <= 1  
0 0 1 <= 1  
-1 0 0 <= 1  
0 -1 0 <= 1  
0 0 -1 <= 1
```

```
Volume of the cube: 8.22251
```

If you reached this far into reading and following this guide, congratulations for your patience. I hope the steps above will work for you on first try.

3.4 Running a sampling example with billiard walk

Volesti comes with an examples folder containing a lot of different use cases of the library. Essentially, each example is designed to demonstrate the usage of basic functions and methods of the library making it easier to use. We can use the code of each example as the basis when trying to implement the methods we care about.

In that regard, we choose an example that deals with sampling in polytopes, since its relative to the main theme of this paper. Go to examples folder and then to sampling-hpolytope-with-billiard-walks. This folder contains a cpp file with the code, a readme file with commands on how to build and run and the necessary CMakeLists.txt file with instructions to the system on how to build the files and where are the libraries included. Create a build folder as before and go in it. According to the tutorial we need to link a specific library called liblpsolve55.so. In order to find where this is located, type the following command:

```
find /usr -name "liblpsolve55.so" 2>/dev/null
```

You will get something like this in return:

```
/usr/lib/lp_solve/liblpsolve55.so
```

Now, initiate cmake by typing:

```
cmake .. -DLP_SOLVE=/usr/lib/lp_solve/liblpsolve55.so
```

Finally, to build the files, type:

```
make
```

You will probably see a ton of warning about everything, but just ignore them. The important thing is to get the end result. If you see these lines at the end of the build process:

```
[ 92%] Linking C static library liblp_solve.a
[ 92%] Built target lp_solve
[ 96%] Building CXX object CMakeFiles/sampler.dir/sampler.cpp.o
[100%] Linking CXX executable sampler
[100%] Built target sampler
```

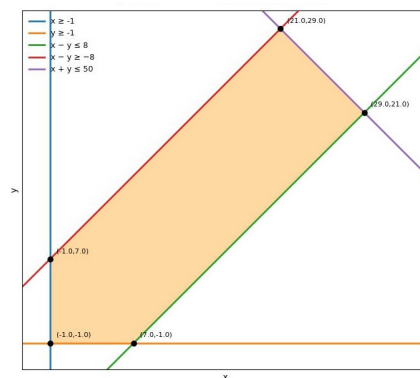
then things have gone quite nicely. The build folder is now full with all kind of files, but the important one is the one called `sampler` that is the executable. The way to run this is to just type:

```
./sampler
```

No parameters are needed since the code uses a custom defined polytope

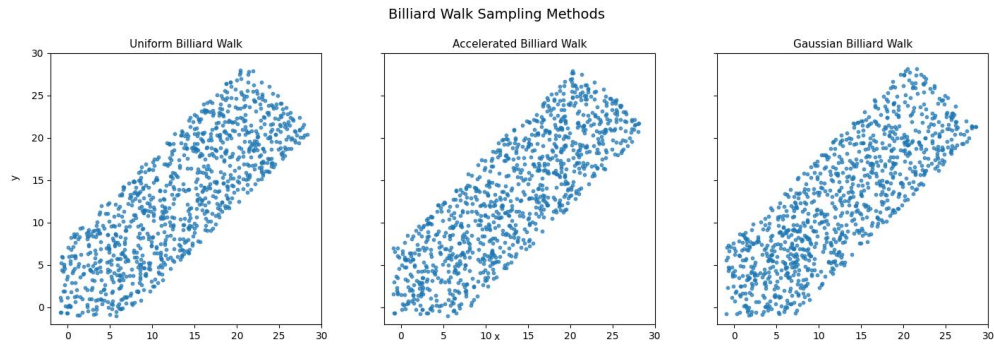
```
x >= -1
y >= -1
x-y <= 8
x-y >= -8
x+y <= 50
```

and samples over it with various variations of billiard walk. The following image shows how this example polytope looks like:



Σχήμα 3.1: The polytope used by the sampling hpolytope with billiard walks example.

Running the example doesn't return anything to the console. Instead, you will see 3 txt files created in the build folder. Each one of these contains the samples of



Σχήμα 3.2: Sampling results using the 3 variations of billiard walk.

one of the methods. In order to display these results, a segment of python code is provided by the tutorial. Make sure to have the necessary libraries and that the txt files are in the same folder as your python file. The result is shown in the next figure.

The first note is that the example works. The sampling does give results inside the requested polytope. Thus, we can copy the way the sampling methods are executed in this example, for our own sampling in the future.

3.5 Sampling example analysis

Understanding how the cpp code works, what's different from the vanilla python version we suggested earlier in the theory and how is the billiard walk implementing in comparison to the vanilla case, is quite useful and educational, especially for those who have never seen anything like it before.

To this end, we will take a look at the source code to gain a deeper understanding of how we can use the library. The way the code has been structured for this particular example of sampling with billiard walk is quite instructive for this purpose. Of course, the basic function of `sampler.cpp`, which contains the source code for the example, is to call the functions that are necessary for its purpose. The functions and their basic logic are hidden in the header files, which we will look at next.

So, the example in `sampler.cpp` obviously starts by including the necessary libraries:

```
#include <iostream>
#include <vector>
```

```
#include <fstream>
#include <boost/random.hpp>
#include "Eigen/Eigen"
```

The first libraries are external to Volesti and concern basic C++ input/output functions, a part of the boost library (on which Volesti is based) that specializes in returning high-quality random numbers, and the Eigen library, which we could say has a role almost analogous to NumPy in the sense that it deals with linear algebra.

Next, we include functions from Volesti's own ecosystem:

```
#include "cartesian_geom/cartesian_kernel.h"
#include "sampling/random_point_generators.hpp"
#include "random_walks/random_walks.hpp"
#include "preprocess/max_inscribed_ellipsoid.hpp"
#include "preprocess/inscribed_ellipsoid_rounding.hpp"
#include "convex_bodies/ellipsoid.h"
#include "convex_bodies/hpolytope.h"
```

In detail, each of these headers does the following:

- ▶ **cartesian_geom/cartesian_kernel.h**: Defines the core numeric types (NT = double), the Cartesian kernel for geometric computations, and the Point class.
- ▶ **sampling/random_point_generators.hpp**: Provides template classes such as RandomPointGenerator and MultivariateGaussianRandomPointGenerator, which are responsible for producing random samples (uniform or Gaussian) inside convex bodies.
- ▶ **random_walks/random_walks.hpp**: Defines algorithms for different random walks (e.g., BilliardWalk, AcceleratedBilliardWalk, HitAndRun, etc.) used to explore the geometry of convex sets.
- ▶ **preprocess/max_inscribed_ellipsoid.hpp**: Contains algorithms to compute the largest ellipsoid that fits inside a polytope, which is useful for rounding and Gaussian-based sampling.
- ▶ **preprocess/inscribed_ellipsoid_rounding.hpp**: Provides utilities for using the inscribed ellipsoid to round or normalize the polytope, improving numerical conditioning for sampling algorithms.

- `convex_bodies/ellipsoid.h`: Defines the `Ellipsoid` class, representing an ellipsoid as a geometric object.
- `convex_bodies/hpolytope.h`: Defines the `HPolytope` class, which models convex polytopes of the form $Ax \leq b$. This is the main convex body used in Volesti sampling examples.

We continue with the following definitions:

```
typedef double NT;
typedef Cartesian<NT> Kernel;
typedef typename Kernel::Point Point;
typedef Eigen::Matrix<NT, Eigen::Dynamic, Eigen::Dynamic> MT;
typedef Eigen::Matrix<NT, Eigen::Dynamic, 1> VT;
typedef BoostRandomNumberGenerator<boost::mt19937, NT> RNGType;
typedef HPolytope<Point> HPOLYTOPE;
```

The `typedef` keyword in C++ is used for aliasing existing data types, user-defined data types, and pointers to a more meaningful name. Typedefs allow you to give descriptive names to standard data types, which can also help you self-document your code. Mostly typedefs are used for aliasing, only if the predefined name is too long or complex to write again and again. The unnecessary use of `typedef` is generally not a good practice. In any case, if we carefully analyze these commands, we can conclude that:

- `typedef double NT;` — Defines the numeric type used throughout the code. Here ofcourse, `double` is renamed to `NT`.
- `typedef Cartesian<NT> Kernel;` — Creates a Cartesian geometry kernel using `NT` as the scalar type. It's a template class in Volesti that bundles geometric primitives (like points, vectors, operations) using a specific numeric type. For this example we consider the Cartesian system with all the known distance definitions etc.
- `typedef typename Kernel::Point Point;` — Defines a shorthand for the point type provided by the kernel. `Point` just represents a point in Euclidean space and for simplicity `Kernel::Point` (ie point in cartesian space) is renamed to `Point`. When we call `Point`, the code understands that we need a point that lives in Cartesian space.

- `typedef Eigen::Matrix<NT, Eigen::Dynamic, Eigen::Dynamic> MT;`
— Defines a dynamic-size Eigen matrix type for the constraint matrix A . It reminds us of the NumPy call, as we saw in the vanilla code. The matrix is defined to use double values and have dynamic size of rows and columns as is renamed to MT (matrix type).
- `typedef Eigen::Matrix<NT, Eigen::Dynamic, 1> VT;` — Defines a dynamic-size Eigen column vector type for the constraint vector b , renamed to VT (vector type, just a column).
- `typedef BoostRandomNumberGenerator<boost::mt19937, NT> RNGType;`
— Defines the random number generator type, based on Boost's Mersenne Twister. It returns NT values (ie double). In the python version we had used `np.random.default` function provided by NumPy.
- `typedef HPolytope<Point> HPOLYTOPE;` — Defines a polytope class (in H-representation, inequalities form) using the Point type from the geometry kernel. The polytope is defined using Points, which in our case means d-dimensional points in cartesian space.

Let's move on to this line:

```
PushBackWalkPolicy push_back_policy;
```

In Volesti, the random walk algorithms are written in a generic, policy-based style (actually its a common design pattern in modern C++ template libraries). Instead of hard-coding what to do when a new sample is found like in the vanilla code, the library accepts a policy object that defines that behavior. In this case, every new sample is pushed it into the output vector (by calling the `std::vector::push_back()` on each generated point). Remember from the theory that in billiard walk all samples are 100% inside the polytope. We can define other policies if needed like `DiscardWalkPolicy` or `StatisticsWalkPolicy`.

We now enter the important part of the logic. Some more typedef lines follow below:

```
typedef BilliardWalk::template Walk<HPOLYTOPE, RNGType> BilliardWalkType;
typedef AcceleratedBilliardWalk::template Walk<HPOLYTOPE, RNGType>
    AcceleratedBilliardWalkType;
typedef GaussianAcceleratedBilliardWalk::template Walk<HPOLYTOPE, RNGType>
    GaussianAcceleratedWalkType;
```

These three typedefs specify which kind of Billiard Walk algorithm is going to be used for sampling points inside a convex polytope. These lines are a bit long and include a lot of information so let's try to decode the big picture. The general pattern is this:

```
<Algorithm>::template Walk<BodyType, RNGType>
```

where *Algorithm* is one of Volesti's random walk algorithms (*BilliardWalk* etc.), *BodyType* is the convex body you are sampling from (in this case, *HPOLYTOPE* is a polytope in halfspace representation) and *RNGType* is the random number generator type (here the boost generator we defined earlier). So each typedef picks one specific algorithm and ties it to the geometric object we're sampling from and the random generator we'll use for randomness. So, each of the 3 lines gives a better name to one of the 3 variants of billiard walk.

As far as the 3 different variants used here, the standard Billiard Walk uses the logic we explained in the theory. The accelerated version uses numerical shortcuts or adaptive steps to reduce correlation between samples and improve convergence. In the Gaussian weighted version samples follow a multivariate normal distribution restricted to the polytope instead of being uniform.

Next, there is a write-to-file function that does exactly that, writes the samples to an output file for further processing. Why no graph you might ask? Because it's C++ and that's kinda hard to do here. Even the authors themselves provide a python script to draw the graph, the one we used to generate images 3.2. There is no point in showing the code of this write function here, go check it out in the library if you are so interested in it.

We move on to the actual functions that set things in motion. The definition of the usual billiard walk function :

```
void sample_using_uniform_billiard_walk(
    HPOLYTOPE& HP,
    RNGType& rng,
    unsigned int walk_len,
    unsigned int num_points)
```

These parameters are defined:

- **HPOLYTOPE& HP**: the convex region we want to sample from (defined by linear inequalities). This is passed by reference so the function can use the same object efficiently.
- **RNGType& rng**: the random number generator (here `BoostRandomNumberGenerator<boost::mt19937>`). Passing by reference avoids copying its internal state and ensures reproducibility.
- **walk_len**: how many reflections (steps) each random walk should take before accepting the next sample. This is different from our vanilla version. There we gave the distance travelled as a parameter (L-scale, listing 2.27) and not the number of reflections.
- **num_points**: how many random samples to generate.

The body of the function follows:

```
std::string filename = "uniform_billiard_walk.txt";
typedef RandomPointGenerator<BilliardWalkType> Generator;

std::vector<Point> randPoints;
Point q(HP.dimension()); // origin
Generator::apply(HP, q, num_points, walk_len,
                 randPoints, push_back_policy, rng);
write_to_file(filename, randPoints);
```

Begin by defining the name of the output file. Next, `RandomPointGenerator` is a template class that implements a general mechanism for generating random points inside convex bodies. It is parameterized by a specific walk type — in this case, `BilliardWalkType`, which we defined earlier as `BilliardWalk::Walk<HPOLYTOPE, RNGType>`. So `Generator` here is essentially a random point generator that uses the Billiard Walk algorithm for sampling inside a polytope. After that, prepare data structures. Define a vector of type `Points` (each `Point` is a geometric object from the Volesti kernel, Cartesian points in our case) called `randPoints` that will hold the sampled points. `Point q(HP.dimension())` initializes a starting point (usually at the origin, ie. all zeros). The Billiard Walk algorithm starts here and begins bouncing inside the polytope.

In the end, running the sampler is all that remains. So far we only defined stuff. In order to run the `Generator` call the `apply` method with the following arguments that we defined above:

- **HP**: The convex polytope to sample from.

- ▶ `q`: Starting point of the random walk.
- ▶ `num_points`: Number of random points to generate.
- ▶ `walk_len`: Number of reflections per sample.
- ▶ `randPoints`: Vector to store the results.
- ▶ `push_back_policy`: Policy that stores each sample in the vector using `push_back`.
- ▶ `rng`: Random number generator used for directions and distances.

The rest of the functions are defined in the same way, just choose the variation of Billiard walk you want. The only different part is inside the Gaussian variation where a preprocessing step is added to calculate an inscribed ellipsoid inside the polytope. In Gaussian billiard sampling, we want to draw samples proportional to a Gaussian density (rather than uniform) inside a convex polytope. However, to do this efficiently and stably, the algorithm needs to know the shape and scale of the region so it can adapt the step directions and handle correlations between dimensions properly. To that end we begin with:

```
// ----- Get inscribed ellipsoid -----
EllipsoidParams<NT> params;
params.barrier_params.maxiter = 150;
```

where a parameter object for the ellipsoid computation is created, accepting NT (double) values. Maxiter parameter sets the max amount of iterations the algorithm will run in order to converge to the exact solution. The next lines are these:

```
auto ellipsoid_result = compute_inscribed_ellipsoid
<MT, EllipsoidType::VOLUMETRIC_BARRIER>(HP.get_mat(), HP.get_vec(),
q.getCoefficients(), params);

const MT E = get<0>(ellipsoid_result);
```

The variable is defined as `auto`, which is possible only on later versions of `c++` and it allows the code itself to define the type of the variable based on the value it is assigned to it. This variable holds the result of the calculation. The template parameter `EllipsoidType::VOLUMETRIC_BARRIER` tells it which definition of inscribed ellipsoid to use. As usual there are many of these definitions, volumetric

is one of them. As for the arguments, it receives matrix A and vector b (the polytope definition) the coefficients defining the origin point and some optimization parameters, ie. the max number of iterations we defined earlier. The final line extracts the first component of a tuple, which also means that the `ellipsoid_results` is itself a tuple probably in the form of (E, center, status). E is type MT, a matrix with dynamic rows and columns and double values (check the typedef where it is defined). This E is added to the Generator arguments.

It's finally time to run the main function using an example polytope:

```
HPOLYTOPE visualization_example() {
    MT Ab(5, 3);
    Ab <<  -1,  0,  1,      // x >= -1
           0, -1,  1,      // y >= -1
           1, -1,  8,      // x-y <= 8
          -1,  1,  8,      // x-y >= -8
           1,  1, 50;      // x+y <= 50

    MT A = Ab.block(0, 0, 5, 2);
    VT b = Ab.col(2);
    return HPOLYTOPE(2, A, b);
}

int main(int argc, char const *argv[]) {
    unsigned int walk_len = 10;
    unsigned int num_points = 1000;

    HPOLYTOPE HP = visualization_example();
    HP.ComputeInnerBall();

    RNGType rng(HP.dimension());
    sample_using_uniform_billiard_walk(HP, rng, walk_len, num_points);
    sample_using_uniform_accelerated_billiard_walk(HP, rng, walk_len, num_points);
    sample_using_gaussian_billiard_walk(HP, rng, walk_len, num_points);
    return 0;
}
```

Begin by defining some of the parameters and using the example polytope defined before main. Then, a function called `ComputeInnerBall` is called on the polytope. This is probably intended for calculating a starting point on the polytope as there is no usage of LP. The code will find small sphere fully contained in the polytope and pick a starting point randomly. In addition, a random number generator is created

bases on the polytopes dimensions. This rng uses the boost random number generator we discussed earlier. Finally, the functions are called for each variation of billiard walk using the arguments we defined earlier.

3.6 Billiard walk in Volesti

In chapters 2.4.10 and 2.4.11, we presented the vanilla billiard walk in depth. Now would be a good time to revise those chapters if you want to refresh your memory regarding the core foundation of the method. In any case, here we will try to analyse the method as it is implemented in Volesti. The main objective is to understand the advanced techniques that dramatically improve the temporal complexity of the code and stability in borderline cases, compared to the simple implementation of the vanilla version.

Volesti sets up the method using the struct data structure of C++. If we think about it in terms of Python or any other language, the analogy would be a class. Therefore, each call to Billiard Walk will be an instance of this class with its own parameters and variables. Let's look at the initial definition:

```
// The main structure encapsulating the Billiard Walk algorithm
struct BilliardWalk
{
    // Constructor with explicit step length L.
    // Initializes parameter struct with L and a flag
    // indicating it's user-defined.
    BilliardWalk(double L)
        : param(L, true)
    {}

    // Default constructor: no step length provided.
    // Initializes parameter struct with L=0 and
    // flag "false" (not set by user).
    BilliardWalk()
        : param(0, false)
    {}

    // Internal parameter structure
    // This nested struct stores step length configuration.
    struct parameters
    {
```

```

    // Constructor: accepts step length L and a boolean flag.
    parameters(double L, bool set)
        :   m_L(L), set_L(set)
    {}

    double m_L;    // The walk length parameter (total travel distance)
    bool set_L;    // Flag indicating whether m_L was explicitly set
};

parameters param; // Instance of the parameters struct

```

The code start by defining a data type called `BilliardWalk` and two constructors. The one constructor is used when passing an argument to the call of the class, while the other is the default behaviour when no argument is given. The `:` syntax after the parentheses is called a member initializer list, which initializes class members directly. It is equivalent to `param = parameters(L, true)`. The final line declares that each `BilliardWalk` object has one `parameters` object inside it, named `param`. Thus, each of the constructors initializes this `param` object with some values based on the argument given, if we pass some `L` then `param` is initialized with values (`L=L`, `set=true`), otherwise we get (`L=0`, `set=false`). We can see the definition of the `parameters` object as a nested class inside `BilliardWalk`. The constructor of `parameters` demands 2 values, a double `m_L` (the values of the wealk length) and a bool `set_L` (a flag that indicates whether the length is set or not).

Next, the code continues by defining a template. C++ template is a powerful tool for creating generic classes or functions. This allows us to write code that works for any data type without rewriting it for each type. Think of a function that adds two numbers. Since c++ is statically typed (it must know the types at compile time) we would have to take all cases of whether the numbers are int, float etc. This is where templates come into play, allowing us to compile the code for any type of variable. To that end, we're defining a version of the `Walk` struct that can work with any kind of polytope representation and any kind of random number generator, not just one fixed type:

```

// The actual implementation of the walk algorithm.
// It depends on two template types:
// - Polytope: the geometric body we walk inside
// - RandomNumberGenerator: the RNG used for sampling
template
<
    typename Polytope,
    typename RandomNumberGenerator

```

```
>
}
```

There are 2 template parameters defined here. This is useful since the struct Walk that follows depends on two types, one representing the polytope and one representing the random number generator that can be anything. The angle brackets < ... > are used to define the template parameters. The struct Walk follows:

```
struct Walk
{
    // Define shorthand types for clarity
    typedef typename Polytope::PointType Point; // Type of a geometric point
    typedef typename Point::FT NT;             // Underlying numeric type (double)
```

We already explained that typedef is used to rename stuff with more conveniently. So, we essentially tell the code that inside whatever Polytope type you give me, there exists a nested type called PointType; we are going to call that Point here for convenience. In addition, look inside the Point type (double, float, ...) find its field type (FT), and call that NT (number type) locally. Typename in front is required when referring to a nested type in a template.

Lets now take a look at the Walk constructors:

```
// Constructor 1: Automatic step length computation
template <typename GenericPolytope>
Walk(GenericPolytope &P, Point const& p, RandomNumberGenerator &rng)
{
    // Compute the diameter (max distance across the polytope)
    _Len = compute_diameter<GenericPolytope>
           ::template compute<NT>(P);

    // Initialize walk from point p
    initialize(P, p, rng);
}

// Constructor 2: Uses user-specified or computed step length
template <typename GenericPolytope>
Walk(GenericPolytope &P, Point const& p, RandomNumberGenerator &rng,
     parameters const& params)
{
    // If user provided a valid step length (set_L == true),
```

```

// use it; otherwise, compute polytope diameter.
_Len = params.set_L ? params.m_L
        : compute_diameter<GenericPolytope>
        ::template compute<NT>(P);

// Initialize walk state
initialize(P, p, rng);
}

```

The Walk struct defines two template constructors that initialize a billiard walk inside a polytope. Both constructors are templated on GenericPolytope, allowing them to accept any polytope type. The first constructor takes a reference to a polytope P, a starting point p, and a random number generator rng as parameters. Inside, it computes a characteristic length *_Len* using a predefined method, which represents the polytope's diameter and scales the billiard walk steps. It then calls `initialize(P, p, rng)` to set up internal variables like the current position *_p* and direction *_v*. Note that in C++, using an underscore *_* at the beginning of a variable name is a common naming convention to indicate that the variable is a private member of a class or struct.

The second constructor adds a parameters object to its arguments. This allows the user to optionally provide a fixed step length *m_L* as discussed earlier. The ternary conditional operator `?:` is used to set *_Len*. `params.set_L` is the condition, if the user has provided a length, this condition is true and *_Len* gets the value `params.m_L`. If nothing is provided the calculation is made automatically. Finally, it calls `initialize(P, p, rng)` just like the first constructor. These template constructors provide flexibility to work with different polytope types while properly setting up the walk's length and initial state.

Finally, the core logic of billiard walk follows hiding behind some methods. The first of these is called `apply`:

```

// Core method: apply()
// Executes multiple billiard walk steps
template
<
    typename GenericPolytope
>
inline void apply(GenericPolytope &P,
                 Point& p,    // Starting point
                 unsigned int const& walk_length,
                 RandomNumberGenerator &rng)

```

```
{
    unsigned int n = P.dimension();    // Dimensionality of the polytope
    NT T = rng.sample_urdist() * _Len // Random travel distance
    const NT dl = 0.995;                // Damping factor (prevents overshooting)
```

The method is ofcourse templated on `GenericPolytope` parameter, which means it can work with any type of polytope object as long as that object implements the methods the function needs. The parameters passed on to the method are a reference `P` to the polytope inside which we are sampling, a point of type `Point` used as the starting point of the walk, the `walk_length` which is the number of reflections per sample and `rng`, the random number generator (taken by boost library). As far as inline is concerned, an inline function is a function in C++ whose code is expanded at the point of call at compile time. It reduces function-call overhead. Inlined functions have their body replace the call at compile time, so no separate symbol is created.

Next, some important parameters of the problem are defined. The number of dimensions `n` retrieved using the `dimension()` method of `GenericPolytope`. `T` which is the initial travel distance along the random direction. `_Len` is the maximum length scale of the polytope (either user-specified or computed from its diameter as discussed earlier). The call `rng.sample_urdist()` returns a random number in $[0,1]$, so `T` is scaled randomly along `_Len`. Finally, `dl` is a damping factor used in the algorithm to slightly reduce the distance after each reflection. This is a numerical safeguard to prevent the walk from sticking exactly at a face due to rounding errors (like our tolerance used in the vanilla walk).

Its time for the billiard walk:

```
// Repeat for the specified number of steps
for (auto j = 0u; j < walk_length; ++j)
{
    T = rng.sample_urdist() * _Len;    // New travel distance per step
    _v = GetDirection<Point>::apply(n, rng); // Random direction vector

    Point p0 = _p;                     // Save initial position
    int it = 0;                         // Reflection iteration counter

    // Limit reflections to 50 * n to prevent infinite loops
    while (it < 50 * n)
    {
        // Compute intersection with the next boundary
        auto pbpair = P.line_positive_intersect(_p, _v, _lambdas,
```

```

                                _Av, _lambda_prev);

    // If remaining distance T is smaller than the intersection
    // distance, move inside and stop the iteration.
    if (T <= pbpair.first) {
        _p += (T * _v);           // Move point
        _lambda_prev = T;
        break;
    }

    // Otherwise, travel until boundary, reflect, and continue.
    _lambda_prev = dl * pbpair.first; // Slightly scale down step
    _p += (_lambda_prev * _v);       // Move to boundary
    T -= _lambda_prev;               // Reduce remaining distance

    // Reflect direction vector against boundary normal
    P.compute_reflection(_v, _p, pbpair.second);

    it++;
}

// If too many reflections, revert to original position
if (it == 50 * n){
    _p = p0;
}
}

```

For each iteration j of the outer loop (the u means that j is unassigned), a random travel length $T \sim \mathcal{U}(0, L)$ is drawn, where L denotes the maximum path length. A random unit direction v is then sampled uniformly on the n -dimensional sphere. The algorithm repeatedly advances the current point p along this direction until either the total travel distance T is exhausted or a boundary of the polytope is reached. The next intersection with the boundary is obtained by calling `P.line_positive_intersect()`, which returns both the distance to the next facet and the index of the facet hit. If the remaining distance T is shorter than the distance to the next boundary, the point simply moves forward by T and the step terminates. Otherwise, the point is advanced to the boundary (slightly inside by a factor of 0.995 for numerical stability), the remaining travel distance is reduced, and the direction vector v is reflected with respect to the facet normal using `P.compute_reflection()`. This process repeats until the entire length T has been traversed or a safety limit of $50n$ reflections is reached, in which case the point is reset to its previous position. Most of these steps were applied by our vanilla version as well.

Finally, lets check the initialize method:

```
// Performs the first billiard trajectory from the initial point

template
<
    typename GenericPolytope
>
inline void initialize(GenericPolytope &P,
                      Point const& p,
                      RandomNumberGenerator &rng)
{
    unsigned int n = P.dimension();
    const NT dl = 0.995;

    // Reset helper vectors to correct dimensions
    _lambdas.setZero(P.num_of_hyperplanes());
    _Av.setZero(P.num_of_hyperplanes());

    _p = p; // Start point
    _v = GetDirection<Point>::apply(n, rng); // Random direction

    NT T = rng.sample_urdist() * _Len; // Travel distance
    Point p0 = _p;
    int it = 0;

    // Find first boundary intersection
    std::pair<NT, int> pbpair
        = P.line_positive_intersect(_p, _v, _lambdas, _Av);

    // If travel distance is less than boundary distance, move and finish
    if (T <= pbpair.first) {
        _p += (T * _v);
        _lambda_prev = T;
        return;
    }

    // Otherwise, move to boundary and reflect
    _lambda_prev = dl * pbpair.first;
    _p += (_lambda_prev * _v);
    T -= _lambda_prev;
    P.compute_reflection(_v, _p, pbpair.second);
}
```

```

// Continue walking until total distance traveled or iteration limit
while (it <= 50 * n)
{
    std::pair<NT, int> pbpair
        = P.line_positive_intersect(_p, _v, _lambdas,
                                    _Av, _lambda_prev);

    if (T <= pbpair.first) {
        _p += (T * _v);
        _lambda_prev = T;
        break;
    } else if (it == 50 * n) {
        // Fallback case if stuck: move a random small distance
        _lambda_prev = rng.sample_urdist() * pbpair.first;
        _p += (_lambda_prev * _v);
        break;
    }

    // Continue reflecting
    _lambda_prev = dl * pbpair.first;
    _p += (_lambda_prev * _v);
    T -= _lambda_prev;
    P.compute_reflection(_v, _p, pbpair.second);
    it++;
}
}

```

The `initialize()` method is responsible for setting up and executing a single billiard trajectory within a polytope. It is a templated function that accepts a generic polytope type `GenericPolytope`, a starting point `p`, and a random number generator `rng`. The procedure begins by retrieving the dimension of the polytope and initializing several internal vectors, such as `_lambdas` and `_Av`, to zero. The current point `_p` is set to the provided starting point, and a random direction vector `_v` is sampled uniformly from the unit sphere using the random generator.

Next, a total travel length T is drawn uniformly in the range $[0, L]$. The algorithm then determines the distance to the first intersection between the line $\mathbf{p} + t\mathbf{v}$ and the boundary of the polytope using the method `P.line_positive_intersect()`. If the total travel length T is smaller than the distance to the first boundary, the point simply moves forward by T units and the function terminates. Otherwise, the walker reaches the boundary before completing its intended distance. In this case, the position is updated to just inside the boundary (scaled by a damping factor $d_l =$

0.995 to avoid numerical overflow), the remaining distance is reduced accordingly, and the direction is reflected using the method `P.compute_reflection()`.

The reflection process continues iteratively while the remaining travel distance is positive and the iteration counter does not exceed a safety limit proportional to the polytope dimension ($50 \cdot n$). Each reflection is handled analytically, ensuring that the new direction maintains the billiard dynamics. If the iteration limit is reached, the algorithm performs a fallback step by advancing the point by a random fraction of the last intersection distance, preventing the walk from stalling due to numerical degeneracies.

The code ends up with some private data members that can only be accessed by methods nested inside the struct:

```
// Private data members
NT _Len;                // Step length or polytope diameter
Point _p;               // Current position
Point _v;               // Current direction vector
NT _lambda_prev;        // Distance from previous reflection
typename Point::Coeff _lambdas; // Vector for constraint distances
typename Point::Coeff _Av; // Stores A*v results for efficiency
```

3.7 Vanilla billiard walk vs Volesti

So, after all this analysis what are the main differences between the simple vanilla implementation of billiard walk and the advanced implementation of Volesti. We can sum up the most important distinctions in the following categories:

► Basic set up

- Vanilla: Python, single function, easy to read and modify.
- Volesti: C++, structured as a struct `BilliardWalk` with nested `Walk` template. Encapsulates state and logic in a class-like design.

► Travel distance selection:

- Vanilla: Each step draws an exponential distance: `remaining = rng.exponential(scale)`
- Volesti: Each step draws a uniform distance along the polytope diameter: `T = rng.sample_urdist() * _Len.`

► Direction selection:

- Vanilla: Random unit direction from a standard normal distribution.
- Volesti: Random direction computed using `GetDirection<Point>::apply(n, rng)`, optimized for uniform sampling on the unit sphere.

► Boundary intersection and reflection:

- Vanilla: Computes all forward intersections and selects the nearest; combines normals when hitting multiple faces.
- Volesti: Uses `line_positive_intersect` and `compute_reflection`; includes hard-coded iteration limits and handles reflections internally.

► Iterations:

- Vanilla: Moves and reflects until the full step distance is traveled (`while remaining > tol`).
- Volesti: Loops over `walk_length` steps; inner loop capped at `50*n` iterations; resets point if stuck.

► Handling multiple faces:

- Vanilla: Sums normals for simultaneous face hits and normalizes for reflection.
- Volesti: Reflection logic is handled internally by the polytope's method; multiple faces processed automatically.

► Initialization:

- Vanilla: Finds an interior point using `find_interior_point`.
- Volesti: Starts from a given point; initializes vectors `_v` and `_lambdas`; precomputes the first reflection if needed.

► Step scaling:

- Vanilla: No explicit damping applied.
- Volesti: Uses a damping factor `d1 = 0.995` to prevent numerical overshoot near boundaries.

3.8 Billiard Walk Variations

Volesti includes a couple of variations of the normal uniform billiard walk we analyzed earlier. First, the Accelerated Billiard Walk (lets call it ABW for short) differs from the normal Billiard Walk in several key ways. First, the travel distance T is sampled exponentially in ABW as $T = -\log(\text{rand}) * _L$ (like our vanilla version), whereas the normal walk uses a uniform step along the polytope diameter $T = \text{rand}[0, 1] * _L$. ABW checks and normalizes the polytope to improve sampling efficiency, while the normal walk does not. Reflection handling in ABW is more advanced: it uses sparse-aware reflection computations (`compute_reflection_abw_sparse`) and maintains a boundary heap (`BoundaryOracleHeap`) to optimize high-dimensional or sparse problems, whereas the normal walk performs standard dense reflections. The iteration limit is also different: ABW uses an adaptive upper bound $_rho = 1000 * \text{dim}$ for reflections, compared to a fixed $50 * n$ iterations in the normal walk. ABW precomputes $_AA = AA^T$ to speed up boundary intersections, while the normal walk has no such pre-computation. Step lengths in ABW are adaptive and can be updated during a burn-in procedure (`parameters_burnin`) to match polytope geometry, whereas the normal walk uses a fixed $_Len$. Starting points in ABW can be generated inside the inner ball, and the burn-in procedure allows for dynamic adjustment, unlike the normal walk which simply starts from a given point. Distance bookkeeping is also implemented in ABW through the moved $_dist$ variable to efficiently handle reflections in high-dimensional or sparse polytopes, while the normal walk does not track such distances. For the exact code check the Volesti github.

Second, the Gaussian Accelerated Billiard Walk (GABW) extends the ABW to sample from a Gaussian distribution over a polytope rather than a uniform distribution. The primary difference is that each step direction $_v$ is transformed using the Cholesky factor of the inverse covariance matrix, effectively sampling according to the Gaussian ellipsoid defined by $_E$, whereas ABW uses a uniform random direction. In GABW, the travel distance T is scaled dynamically by a coefficient computed from the quadratic form $v^T E v$, accounting for the local geometry of the Gaussian distribution; ABW uses fixed or exponentially sampled step lengths along the polytope diameter. Both walks handle reflections at boundaries, but GABW additionally adjusts T by this coefficient during each reflection to respect the Gaussian scaling. GABW also precomputes matrices $_AE$ and $_AEA$ to efficiently handle covariance-related computations and interactions with the polytope, while ABW only precomputes $_AA$ for standard boundary intersections. Sparse vs. dense handling is retained in both, but in GABW the Cholesky decomposition of the covariance is computed once and reused to transform directions, adding extra linear algebra operations. Other structural features, like iteration limits, boundary heap updates, and fallback

mechanisms when stuck, are inherited from ABW, ensuring robust sampling.

3.9 Creating our own example for sampling

The analysis in the previous paragraphs was deemed necessary for a better understanding of, first, the syntax of C++, which is considerably more complex than Python, and, second, the advanced techniques used for sampling with Volesti. The fact that we analysed the billiard walk method was probably a result of its use in the Volesti example. This does not negate the fact that the other sampling methods from Volesti are implemented with the same philosophy and logic. For more details on the other methods, please refer to the source code found on GitHub. If you have read the previous paragraphs, you will now be able to easily decode the other methods as well as the advanced techniques used to improve the time complexity and stability of each method.

What we will aim to do next is to design and implement our own sampling example, using the same logic we used in the vanilla version. Specifically, we will use specific polytopes on which we will run all sampling methods, extracting the convergence parameters and runtime for each method. The whole experiment will become clearer as it progresses. What matters is that we manage to call the sampling methods so that we can run them. In the Volesti example we implemented before, we saw in detail how one can call the desired variation of the billiard walk and the parameters that must be set. For the other methods, there are two strategies: either we search deep into the source code and find how each method is defined so that we know which parameters we need, or we find an example where the method has already been called and then we can simply copy the logic. The second strategy seems easier and quicker. So let's see what we can glean from the existing examples.

3.9.1 Setting up the necessary files

Remember from chapter 3.3 that we created a new folder called `volesti_sampling` and inside that we built and run the tutorial example. Lets keep working here in order to avoid damage to the main volesti source code. Create our own examples folder and inside it an example we conveniently call `general_sampling`. If you print the working directory you should see something like this:

```
$ pwd
/home/<your_name>/volesti_sampling/examples/general_sampling
```

Now, copy and paste all the files from the tutorial example to this new folder, except build and plot.py. We will use them as the basis of our own example. Rename sampler.cpp file to general_sampling.cpp and delete everything after the typedef commands in the beginning. Do not mess with the include statements. What I always believe is that its better to include more dependencies than searching for the ones you are missing. Most of them will be eventually used during our example. In addition, you can delete all contents from readme.md file as we will write our own summary and instructions. The most important part is the CMakeLists.txt file. Don't even consider deleting anything from there, as it could break it. If it works, it works. We are now ready to begin our own example.

3.9.2 Starting to work on our example using Ball Walk

First of all, we will need a polytope for sampling. We can use the well-known polytope from the vanilla method or, alternatively, define a random polyhedron, but as you will recall from the vanilla method, this is somewhat tedious procedure. The good thing about Volesti is that it offers ready classes that create instances from known polytopes such as cubes and ellipsoids.

Navigate to volesti/examples/hpolytpe-volume and check the cpp file. There, we are lucky enough that the method for creating any dimensional cubes is used. Specifically we read that:

```
// Generating a 4-dimensional cube centered at origin
HPOLYTOPE HP1 = generate_cube<HPOLYTOPE>(4, false);
```

Be careful thought. This knowledge doesn't come for free. There are probably new dependencies that we need to add to the tutorial code used as basis of this example. Checking the includes of this example vs our own, one can determine that we should probably add these two:

```
#include "hpolytope.h"
#include "known_polytope_generators.h"
```

dependencies to our project. (Since I have already experimented with all the new dependencies trust me when I say its these two exactly). It doesn't matter if extra stuff is added but its pretty easy to realize that the dependencies including the word volume are probably not needed for sampling.

We are not done yet though. We need to tell the compiler where to find these new dependencies as they are hiding somewhere inside the volesti include folder. Cmake will link them for us automatically if the correct additions are made to the CMakeLists.txt file. In order to determine what extra paths are needed to be specified we again compare the CMakeLists.txt with our own. The only path that is missing is the one leading to generators (again after trial and error this path is required). Add it to our CMakeLists.txt:

```
${VOL_ROOT}/include/generators
```

Now, lets set up an example to generate a cube. Just copy paste the code from the volume example where the cube was used to our own example. Your main should look like this:

```
int main(int argc, char const *argv[]) {
    // Generating a 4-dimensional cube centered at origin
    HPOLYTOPE CUBE = generate_cube<HPOLYTOPE>(4, false);
    std::cout<<"Polytope CUBE: \n";
    CUBE.print();
    std::cout<<"\n";
    return 0;
}
```

Create a build folder if you haven't already and enter it. Do `cmake ..` (no extra arguments needed so far) and then `make`. It should compile just fine. Run `general_sampling` to get this output:

```
Polytope CUBE:
 8 4 double
1 0 0 0 <= 1
0 1 0 0 <= 1
0 0 1 0 <= 1
0 0 0 1 <= 1
-1 0 0 0 <= 1
0 -1 0 0 <= 1
0 0 -1 0 <= 1
0 0 0 -1 <= 1
```

To get a better idea of how these cubes are created you can navigate to `include / generators / known_polytope_generators.h` and check out the `generate_cube` function.

There are two variations offered, one returns the cube in H-representation and the other in V-representation depending on whether the boolean argument is true or false. Here we chose false so that the cube is in H-rep. The generation function simply creates 2 matrices and assigns 1 and -1 to the proper positions. This header file also includes use-ready functions for other known polytopes. Let's try a simplex.

```
HPOLYTOPE SIMPLEX = generate_simplex<HPOLYTOPE>(4, false);
```

The exit was:

Polytope SIMPLEX:

```
5 4 double
1 0 0 0 <= 0
0 1 0 0 <= 0
0 0 1 0 <= 0
0 0 0 1 <= 0
-1 -1 -1 -1 <= 1
```

Using this format we can create any type of polytope that we have a function for.

Now that we have our polytope, it is time to sample over its volume. Billiard walk is already known and we can simply copy the code from before and call the function using the generated cube. However, we want to try out all the sampling methods we saw in their vanilla version, as well as what extra Volesti has to offer, methods that are mainly variations of the methods we studied in theory and try to improve some important aspect like the mixing time. We will try them all. In addition, the sampling distribution for now will be uniform as it is the simpler and works with any sampling method.

Let's begin with the ball walk. Go to include `/random_walks/uniform_ball_walk.hpp`. The basic format of the ball walk is quite similar to the billiard walk one. To get a better understanding of the source code, read chapter 3.6 where the billiard walk source code was explained in detail. In any case, what we are looking for is the apply method of the walk struct inside ball walk in order to know what arguments it requires. Lucky for us, the arguments are the same, so we can call it by repeating the billiard walk call. The `walk_length` in this case tells us after how many balls we keep the sample. Our code looks like this:

```
typedef BallWalk::template Walk<HPOLYTOPE, RNGType> BallWalkType;
```

```

void sample_using_ball_walk(HPOLYTOPE& CUBE, RNGType& rng,
                           unsigned int walk_len, unsigned int num_points) {
    std::string filename = "uniform_ball_walk.txt";
    typedef RandomPointGenerator<BallWalkType> Generator;

    std::vector<Point> randPoints;
    Point q(CUBE.dimension());
    Generator::apply(CUBE, q, num_points, walk_len,
                    randPoints, push_back_policy, rng);
    write_to_file(filename, randPoints);
    std::cout << "Saved " << randPoints.size()
              << " points sampled by Ball Walk to " << filename << std::endl;
}

int main(int argc, char const *argv[]) {
    std::cout << "Starting Process" << std::endl;

    std::cout << "Creating 3D CUBE" << std::endl;
    HPOLYTOPE CUBE = generate_cube<HPOLYTOPE>(3, false);
    std::cout << "CUBE Ready" << std::endl;

    unsigned int walk_len = 1;
    unsigned int num_points = 3000;
    RNGType rng(CUBE.dimension());

    CUBE.ComputeInnerBall();
    auto innerBall = CUBE.ComputeInnerBall();
    auto center = innerBall.first;
    auto radius = innerBall.second;

    std::cout << "Inner ball radius: " << radius << std::endl;
    std::cout << "Inner ball center: ";
    center.print();           // prints the point to stdout
    std::cout << std::endl;

    std::cout << "Ball Walk Sampling" << std::endl;
    sample_using_ball_walk(CUBE, rng, walk_len, num_points);

    return 0;
}

```

As you can see, we have copied the technique used to perform the billiard walk. First, we create a shortcut for creating ball walk objects. All these structs that

represent the various sampling methods include a nested struct called Walk. This design choice really helps consistency and efficiency plus it matches the Random Point Generator engine. Generally speaking, the outer BallWalk is used to configure the parameters of the walk, while the walk nested struct links it to a specific polytope and a random generator. The apply method of the Walk struct runs the logic of the sampling algorithm.

Inside the sample-using-ball-walk function, a filename for the output is defined and then we link the BallWalkType to a random point generator (from boost library). This holds the logic for creating random numbers. Then all that needs to be done is to call the apply method of the Walk struct that runs the sampling algorithm for ball walk using rng to generate a random number when needed. We write the results to a txt file using the example function.

Inside main, a lot of messages have been added to check the progress of the execution. In addition, effort has been directed towards understanding what the call of ComputeInnerBall does. You can find this method by navigating to include / convex_bodies / zpolytope.h (line 362) where it returns this:

```
std::pair<Point, NT> (center, radius)
```

That way, it is possible to print these values with the lines we have added to the code and check that they indeed return the radius and the center of a ball. This ball is the largest possible ball fully contained inside the polytope and its used to get a starting point for the walk (its center). Just a call to this method is required since it updates an internal cached value inside CUBE saving the starting point and the ball radius for later. This radius is used immediately to compute the proper radius of the sampling ball (in python we gave that by hand) using this formula:

$$\delta = \frac{4 \cdot r}{\sqrt{d}}$$

If you run the code without calling ComputeInnerBall there will be no errors but you are in for a surprise when you check the txt file. All samples will be zero since probably all values have defaulted to zero.

If you kept the includes and typedefs from the initial example and added those that were discussed earlier this main should compile without the need to do cmake .. again. Just type make. The result should look like this:

Starting Process

Creating 3D CUBE

CUBE Ready

Inner ball radius: 1

Inner ball center: 0 0 0

Ball Walk Sampling

Saved 1000 points sampled by Ball Walk to uniform_ball_walk.txt

In order to be sure of the correct implementation of the method we should check the txt file with the samples. I will show you the first 10 samples below:

```
0 0 0
0 0 0
0 0 0
0 0 0
-0.0272577 0.624748 -0.522942
-0.0272577 0.624748 -0.522942
0.203095 0.93175 -0.957666
0.203095 0.93175 -0.957666
0.203095 0.93175 -0.957666
0.203095 0.93175 -0.957666
0.203095 0.93175 -0.957666
```

Well, that's unexpected. I told you I will present the 10 first samples but it looks like there are only 3 different samples here and one of them is the starting point. The first suspect of this problem is the `walk_len=1`. That is what we changed compare to the billiard walk. Back to the source ball walk code, the part where `walk_len` is used is this for loop inside apply method of Walk struct:

```
for (auto j = 0u; j < walk_length; ++j)
{
    Point y = GetPointInDsphere<Point>::apply(P.dimension(),
                                              _delta,
                                              rng);

    y += p;
    if (P.is_in(y) == -1) p = y;
}
```

The sampler returns a random point `y` in the ball with radius `delta` calculated from the math formula we saw earlier. Then, current point `p` is added to `y` and a

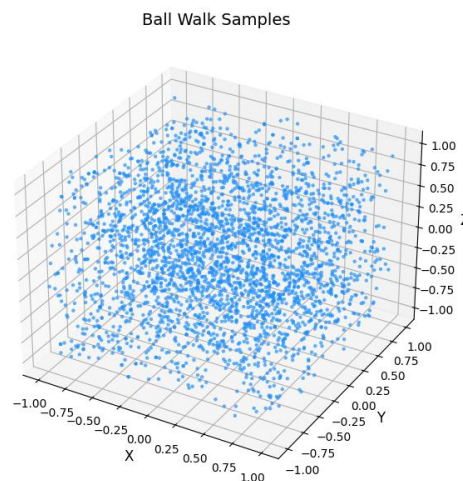
check is made of whether this new point is inside the polytope. If the result is not -1, then the point is outside the polytope and we do not update current point. There are a couple of reasons of why this condition should be false again and again. Firstly, the geometric one: the current point could be really close to the boundary of the polytope so that the ball is mostly outside the borders. This way the sample has a high chance to find itself outside the polytope and the current point fails to update for many steps. Secondly the programming one: maybe the new point is randomly chosen so close to zero that the code rounds up the displacement to what it was. Whatever the case, we should increase the `walk_len`.

```
unsigned int walk_len = 20;
```

After we re-compile and run again the new 5 first samples are these:

```
0.220432 -0.346057 -0.430803
-0.749733 0.819683 -0.227501
0.0155678 0.539782 -0.769278
-0.529927 0.897745 0.151633
-0.956973 -0.334199 -0.355999
```

The results look far better, but are they compose a uniform distribution inside a cube? To find out, we design the scatter plot of these samples. This is the graph:



Σχήμα 3.3: Scatter plot of 3000 samples in a 3D cube using Volesti Ball Walk with `walk_len` 20.

The results are now confirmed. We are confident that things work as intended and the code is correct.

3.9.3 Applying the convergence metrics

The scatter graph was useful indeed but in higher dimension we need to use the convergence metrics. Check chapter 2.5 for analytical explanation on what they are and how to use them. Again, Volesti does implement these metrics in a more advanced and elegant way. Thus, we only need to find how to call upon them using the sample results from the random walks. To that end, check examples / correlation_matrices / sampler.cpp (lines 203 - 213). There the metrics are calculated with the only argument being a matrix of the sampled points.

We gonna need more headers to our source file. In order to accommodate the new functions add these extra includes AFTER our current includes:

```
// These are related to PSRF and ESS
#include "sampling/sample_correlation_matrices.hpp"
#include "matrix_operations/EigenvaluesProblems.h"
#include "diagnostics/effective_sample_size.hpp"
#include "diagnostics/univariate_psrfs.hpp"
```

I deliberately wrote the word after with capital letters. Believe me when I say that it will cause an error if you add them before, I didn't had the courage to test out why. The CMakeLists.txt needs no changes. Everything needed is already linked to our source file. Now, it is time to update our sample-using-ball-walk function in order to calculate the metrics. All we need to do is copy the code from the volesti example.

As usual there is a trick. The metrics expect an Eigen object (Eigen is the NumPy for c++) but are samples are returned as the custom Points type. That is why they must be converted. The code of the example though will give you errors. That's why you should add this code in order to do the conversion:

```
// getCoefficientsFromMatrix<NT, MT> a compatible
// Eigen object, not a Volesti Point. We need to convert randPoints.
MT samples(CUBE.dimension(), num_points);
for (unsigned int jj = 0; jj < randPoints.size(); ++jj) {
    samples.col(jj) = randPoints[jj].getCoefficients();
}
```

Then, simply add these lines:

```

//calculate psrf
std::cout << "Calculating PSRF" << std::endl;
VT psrf = univariate_psrfs<NT, VT, MT>(samples);
double max_psrfs = psrf.maxCoeff();
std::cout << "PSRF = " << max_psrfs << std::endl;

//calculate ess
std::cout << "Calculating ESS" << std::endl;
unsigned int min_ess = 0;
VT ess_vector = effective_sample_size<NT, VT, MT>(samples, min_ess);
std::cout << "Effective Sample Size = " << min_ess << std::endl;
std::cout << "Average Effective Sample Size = " << min_ess/(double)num_points << std::endl;
std::cout << std::endl;
}

```

After running the code the result looks like this:

```

Ball Walk Sampling
Saved 3000 points sampled by Ball Walk to uniform_ball_walk.txt
Saved Eigen matrix representation to samples_matrix.txt
Calculating PSRF
PSRF = 1.00068
Calculating ESS
Effective Sample Size = 2351
Average Effective Sample Size = 0.783667

```

Note that we printed the samples matrix in order to check if the samples are translated correctly. This was super useful during debugging but is not needed any more. In any case, we got some nice results, since the values of the metrics are excellent and verify the success of the sampling. To learn more about the meaning of these values check chapter 2.5.

3.9.4 Generating samples dynamically based on ESS

Briefly, ESS tells us how many of the dependent Markov chain samples can be considered independent. If we sample 10000 points but ESS is 10 or something, then we are wasting our time. To that end, a good idea is to keep generating samples until ESS reaches a target value while PSRF is close to 1 of course. It will be necessary to modify our existing function a bit.

Our strategy consists of these steps: we will add an extra parameter called `target_ess` to replace number of samples. The sampling will take place in small batches, of size 100 for example. After every 100 samples collected with the same method used so far, ESS will be calculated again. While the new ESS is lower than our target, we keep sampling until we have enough samples. These modifications will turn the code to this:

```
void sample_using_ball_walk(HPOLYTOPE& CUBE, RNGType& rng,
                           unsigned int walk_len,
                           unsigned int target_ESS) {

    std::string filename = "uniform_ball_walk.txt";
    typedef RandomPointGenerator<BallWalkType> Generator;

    std::vector<Point> randPoints;
    Point q(CUBE.dimension()); // origin

    unsigned int batch_size = 100; // number of points to generate
                                   per batch
    MT samples(CUBE.dimension(), 0); // dynamic number of columns

    unsigned int total_points = 0; //Total points so far
    unsigned int current_ESS = 0; //Starting ESS vvalue

    while (current_ESS < target_ESS) {
        std::vector<Point> batchPoints;
        Generator::apply(CUBE, q, batch_size, walk_len,
                        batchPoints, push_back_policy, rng);

        // Append batchPoints to randPoints
        randPoints.insert(randPoints.end(), batchPoints.begin(),
                        batchPoints.end());

        // Convert batch to Eigen and append as new columns to samples
        MT batchMat(CUBE.dimension(), batchPoints.size());
        for (unsigned int jj = 0; jj < batchPoints.size(); ++jj) {
            batchMat.col(jj) = batchPoints[jj].getCoefficients();
        }

        // Append batchMat to samples
        MT tmp(samples.rows(), samples.cols() + batchMat.cols());
        if (samples.cols() > 0) tmp << samples, batchMat;
        else tmp = batchMat;
        samples = tmp;
    }
}
```

```

        total_points += batchPoints.size();

        // Calculate ESS
        unsigned int min_ess = 0;
        VT ess_vector = effective_sample_size<NT, VT, MT>
                        (samples, min_ess);
        current_ESS = min_ess;

        std::cout << "Generated " << total_points
                  << " points, current ESS = " << current_ESS
                  << std::endl;
    }

    // Save all sampled points to file
    write_to_file(filename, randPoints);
    std::cout << "Finished sampling. Total points: " << total_points
              << ", ESS reached: " << current_ESS
              << ". Saved to " << filename << std::endl;

    //calculate psrf
    std::cout << "Calculating PSRF" << std::endl;
    VT psrf = univariate_psrfs<NT, VT, MT>(samples);
    double max_psrfs = psrf.maxCoeff();
    std::cout << "PSRF = " << max_psrfs << std::endl;
}

int main(int argc, char const *argv[]) {
    std::cout << "Starting Process" << std::endl;

    std::cout << "Creating 3D CUBE" << std::endl;
    HPOLYTOPE CUBE = generate_cube<HPOLYTOPE>(3, false);
    std::cout << "CUBE Ready" << std::endl;

    unsigned int walk_len = 20;
    unsigned int target_ESS = 3000;

    RNGType rng(CUBE.dimension());

    CUBE.ComputeInnerBall();

    std::cout << "Ball Walk Sampling" << std::endl;
    sample_using_ball_walk(CUBE, rng, walk_len, target_ESS);

    return 0;
}

```

```
}
```

The code is simple but a bit inefficient since we create new temp matrices every step in order to copy paste the new values, but it works without the need for extra headers. The result of this code is the following:

```
Starting Process
Creating 3D CUBE
CUBE Ready
Ball Walk Sampling
Generated 100 points, current ESS = 85
Generated 200 points, current ESS = 157
.....
Generated 3400 points, current ESS = 2934
Generated 3500 points, current ESS = 3093
Finished sampling. Total points: 3500, ESS reached: 3093.
                        Saved to uniform_ball_walk.txt
Calculating PSRF
PSRF = 1.0001
```

Everything seems to work fine and the debugging messages are confirming it. There are a number of tasks left, first to expand the number of dimensions, second to apply the rest of the random walk methods and third to explore different distributions. One task at a time.

3.9.5 Applying for all random walk methods and adding time

In order to generalize the experiment for any of the offered uniform methods, it is best to create a template function with a template parameter referring to the preferred walk method. Again, we can have a look to the offered examples in order to help us with this task. Go back to the example that the metrics were found (examples / correlation matrices / sampler.cpp) and check the templates. They all come with a parameter called WalkType that represents an object of any given random walk struct. We can use this idea for our experiment.

We can begin by altering the name of our template to a more general description and adding these new parameters:

```
template <typename WalkType>
```

```

void sample_using_walk(HPOLYTOPE& Polytope, RNGType& rng,
                      unsigned int walk_len,
                      unsigned int target_ESS,
                      const std::string& walk_name)
{

```

All that needs to be done now is to pass this argument to the typedef line where we defined the short name for our walk:

```

typedef RandomPointGenerator<WalkType> Generator;

```

Generator will now be an instance of any WalkType we pass as argument when calling the template. So, later in the function where we do Generator::apply, the code will use the apply method that corresponds to the walk type chosen, thus applying any random walk we need without creating such a function for every single type of walk.

We present here what the end product looks like with all the mentioned changes applied. We also count the time it takes for every generator to run. The clock starts and ends right before and after the Generator apply line where the actual random walk runs. Copying new batches and calculating ESS is not included in the time. Finally, we change the name of the polytope from CUBE to just Polytope since we will use more shapes in the future.

```

// This template can be used with any random walk
template <typename WalkType>
void sample_using_walk(HPOLYTOPE& Polytope, RNGType& rng,
                      unsigned int walk_len,
                      unsigned int target_ESS,
                      const std::string& walk_name)
{
    std::string filename = "uniform_" + walk_name + ".txt";
    typedef RandomPointGenerator<WalkType> Generator;

    std::vector<Point> randPoints;
    Point q(Polytope.dimension()); // start point

    unsigned int batch_size = 100; // number of points to generate
                                   per batch
    MT samples(Polytope.dimension(), 0); // dynamic number of columns

```

```

unsigned int total_points = 0;
unsigned int current_ESS = 0;
double total_sampling_time = 0.0;

std::cout << "[" << walk_name << "]" Starting sampling...\n";

// Continue sampling until we hit ESS target
while (current_ESS < target_ESS) {

    std::vector<Point> batchPoints;

    // Time only the generation of samples
    auto start = std::chrono::steady_clock::now();
    Generator::apply(Polytope, q, batch_size, walk_len, batchPoints,
                    push_back_policy, rng);
    auto end = std::chrono::steady_clock::now();

    double batch_time = std::chrono::duration_cast<std::chrono::
        ::microseconds>(end - start).count();
    total_sampling_time += batch_time;

    // Append batch points
    randPoints.insert(randPoints.end(), batchPoints.begin(),
                    batchPoints.end());

    // Convert to Eigen matrix, needed for the ess function
    MT batchMat(Polytope.dimension(), batchPoints.size());
    for (unsigned int jj = 0; jj < batchPoints.size(); ++jj)
        batchMat.col(jj) = batchPoints[jj].getCoefficients();

    // Append columns to samples
    MT tmp(samples.rows(), samples.cols() + batchMat.cols());
    if (samples.cols() > 0) tmp << samples, batchMat;
    else tmp = batchMat;
    samples = tmp;

    total_points += batchPoints.size();

    // Compute ESS for the samples so far
    unsigned int min_ess = 0;
    VT ess_vector = effective_sample_size<NT, VT, MT>(samples, min_ess);
    current_ESS = min_ess;

    // These extra print lines shows the ESS for every new batch.

```

```

        //std::cout << "[" << walk_name << "]" Generated "
        //                                << total_points
        //                                << " points, current ESS = " << current_ESS
        //                                << ", batch time = " << batch_time << " micros"
        //                                << std::endl;
    }

    // Save samples to a txt file. One column for every dimension.
    write_to_file(filename, randPoints);

    // Print final results
    std::cout << "[" << walk_name << "]" Finished sampling.\n"
              << "[" << walk_name << "]" Total points: "
              << total_points << "\n"
              << "[" << walk_name << "]" ESS reached: "
              << current_ESS << "\n"
              << "[" << walk_name << "]" Saved to: " <<
              filename << std::endl;

    // Calculate PSRF
    // std::cout << "[" << walk_name << "]" Calculating PSRF..."
    //                                << std::endl;
    VT psrf = univariate_psrfs<NT, VT, MT>(samples);
    double max_psrfs = psrf.maxCoeff();
    std::cout << "[" << walk_name << "]" PSRF = " << max_psrfs
              << std::endl;

    // Print timing summary at the end
    std::cout << "[" << walk_name << "]" Total sampling time: "
              << total_sampling_time << " micros" << std::endl;
}

```

In order to use this new upgraded function, just add these lines inside your main:

```

typedef BallWalk::template Walk<HPOLYTOPE, RNGType> BallWalkType;
sample_using_walk<BallWalkType>(Polytope, rng, walk_len,
                                target_ESS, "BallWalk");

```

The output looks like this:

Starting Process...

As you may have noticed most of the cout messages have been removed. The same plan is applied for the sample function. Writing to a file is also commented out since there is no point in creating a graph for more than 3 dimensions. Lastly, the time is now calculated in seconds as it will become significant for many dimensions. We can now run this version:

```
Polytope Ready. Dimensions: 10
[BallWalk] Total points: 11400
[BallWalk] ESS reached: 3036
[BallWalk] PSRF = 1.00074
[BallWalk] Total sampling time: 0.0427142 s
Polytope Ready. Dimensions: 15
[BallWalk] Total points: 21600
[BallWalk] ESS reached: 3045
[BallWalk] PSRF = 1.00094
[BallWalk] Total sampling time: 0.12277 s
Polytope Ready. Dimensions: 20
[BallWalk] Total points: 35600
[BallWalk] ESS reached: 3000
[BallWalk] PSRF = 1.00124
[BallWalk] Total sampling time: 0.316353 s
Polytope Ready. Dimensions: 25
[BallWalk] Total points: 53000
[BallWalk] ESS reached: 3011
[BallWalk] PSRF = 1.00132
[BallWalk] Total sampling time: 0.643154 s
```

Unfortunately, while these results show that the actual sampling doesn't take that long, the runtime of the code is super high. That's why we only show the results for up to 25 dimensions. Also, notice how many more samples the code has to find in order to reach the ESS threshold. From 10 to 20 dimension the necessary point have more than tripled and this effect will only grow.

3.9.7 Optimizing the code

It will probably be a good idea to increase the `batch_size` in order to calculate ess fewer times. According to the data we collected in the previous chapter for ball walk, there is no point in calculating the ess every 100 or even 1000 points since the amount of samples needed is too great. Following this line of thought, an adaptive `batch_size` sounds like the best idea. To that end we create the following function:

```
// Function to dynamically compute the batch size
// based on the walk type and dimension
unsigned int compute_batch_size(const std::string& walk_name,
                                unsigned int dim) {

    // Default base size
    unsigned int batch_size = 500;

    // Adaptive logic by walk type
    if (walk_name == "BallWalk") {
        batch_size = dim * 100;
    }
    else if (walk_name == "AcceleratedBilliardWalk") {
        batch_size = dim * 1 + 200;
    }
    else if (walk_name == "BilliardWalk") {
        batch_size = dim * 2 + 200;
    }
    else if (walk_name == "DikindWalk") {
        batch_size = dim * 100;
    }
    else if (walk_name == "JohnWalk") {
        batch_size = dim * 15;
    }
    else if (walk_name == "VaidyaWalk") {
        batch_size = dim * 15;
    }
    else if (walk_name == "CDHRWalk") {
        batch_size = dim * 2 + 200;
    }
    else if (walk_name == "RDHRWalk") {
        batch_size = dim * 50;
    }
    return batch_size;
}
```

What we did here is set the `batch_size` in relation to the dimension and the method used. These values have been selected after a lot of trials for the different methods. For ball walk, the number of points needed is too great so we are more precise in setting up the correct number. For the billiard walk methods, points come out quite independent so the batch size doesn't have to be too large.

The same variable behaviour needs to be implemented for the walk length as well. For ball walk, a big walk length is needed for the method to converge

properly, while for the billiard walks walk length of 1 is sufficient for the method to work as intended. To that end we introduce another helper function responsible for manipulating the walk length based on the walk name.

```
// We choose the walk_len appropriate for each method
unsigned int set_walk_len(const std::string& walk_name,
                          unsigned int dim) {
    static const std::unordered_map<std::string,
    std::function<unsigned int(unsigned int)>> walk_len_map = {

        {"BallWalk", [] (unsigned int dim){ return dim*2; }},
        {"BilliardWalk", [] (unsigned int){ return 1; }},
        {"AcceleratedBilliardWalk", [] (unsigned int){ return 1; }},
        {"CDHRWalk", [] (unsigned int dim){ return dim*2; }},
        {"RDHRWalk", [] (unsigned int dim){ return dim*2; }},
        {"DikinWalk", [] (unsigned int){ return 10; }},
        {"JohnWalk", [] (unsigned int){ return 10; }},
        {"VaidyaWalk", [] (unsigned int){ return 10; }}

    };

    if (auto it = walk_len_map.find(walk_name); it != walk_len_map.end())
        return it->second(dim);

    return 10; // default fallback
}
```

Further experiments can tune these numbers even better, but for now there are more important matters to attend to.

Another problem is how to store the batches. To that end, a vector is chosen to store the values. A vector in C++ is like a resizable array. Both vectors and arrays are data structures used to store multiple elements of the same data type. The difference between an array and a vector, is that the size of an array cannot be modified (you cannot add or remove elements from an array). A vector however, can grow or shrink in size as needed. This property is exactly what we need here.

For every step of the loop, we create a specific amount of samples equal to batch size. For these samples ESS is calculated (only after there are atleast `ess-threshold` number of samples created, no reason to calculate ESS before that). If the threshold is not reached, new samples are created in the next loop and are added to the vector. This way, all samples generated so far are stored and contribute in calculating the ESS. This vector will hold data of type `Point`, that represent our samples in Cartesian space:

```
// Dynamic vector that will grow automatically
std::vector<Point> allSamples;
allSamples.reserve(4000);
.....
std::vector<Point> batchPoints;
.....
Generator::apply(Polytope, q, num_points, walk_len, batchPoints,
                 push_back_policy, rng);
.....
allSamples.insert(allSamples.end(), batchPoints.begin(), batchPoints.end());
```

Next, we need a timer to count the work load of every process; the samples generation, the ess calculation and the psrf calculation. We could type `std::chrono` before and after each command but why not create a class and refresh our memory of object oriented programming:

```
class Timer {
public:
    Timer(const std::string& name = "") : walk_name(name), total_time(0.0) {}

    void start() { start_time = std::chrono::steady_clock::now(); }

    // Stop and accumulate
    double stop(const std::string& label = "") {
        auto end_time = std::chrono::steady_clock::now();
        double elapsed = std::chrono::duration<double>(end_time
                                                         - start_time).count();

        total_time += elapsed;
        if (!label.empty())
            std::cout << "[" << walk_name << "]" " << label << " = "
                      << elapsed << " s\n";

        return elapsed;
    }
    double get_total_time() const { return total_time; }

private:
    std::string walk_name;
    std::chrono::steady_clock::time_point start_time;
    double total_time;
};
```

There are 2 methods, start and stop each one working as named. The stop

method can accept a message that will cause it to output sad message along with time elapsed since the last start. Its easy now to count times:

```
Timer ess_timer(walk_name);
.....
// Compute current ESS
ess_timer.start();
current_ESS = compute_ess<NT, VT, MT>(samples);
ess_timer.stop("");
```

Finally, we need to be careful with the starting point. So far, we only experimented with a hypercube so taking (0,0,0,...,0) as a starting point is right on track. However, for more complex polytopes it is essential for the stability of the methods that we use an inner point as the start. To that end, some extra lines of code are added to take advantage of the ComputeInnerBall method that calculates an inscribed ball (hypersphere) and returns its center among others.

```
//These lines will return the coordinates of the center of
//the inner ball that will be used as our starting point
auto inner = Polytope.ComputeInnerBall();
Point center = inner.first;
```

Then, we pass this starting point as an argument to our sampling function:

```
sample_using_walk<BallWalkType>(Polytope, center, rng,
                                target_ESS, "BallWalk");
...
(inside the function)
Point starting_point = start_point; // start point
...
Generator::apply(Polytope, starting_point, num_points, walk_len,
                 batchPoints, push_back_policy, rng);
```

3.10 Running our example

Our example is complete, tested and ready to execute. The main purpose is to test Volesti's sampling algorithms regarding the required time to reach a certain

ess value on some polytope. Our code doesn't come with any user input so each experiment must be carefully prepared by changed a few lines of code or commenting and un commenting some other lines. Each test will vary based on one of 4 things, the type of polytope used, the random walk used and the ess threshold used and the number of dimensions.

3.10.1 Time trial for d-hypercube and constant ess threshold

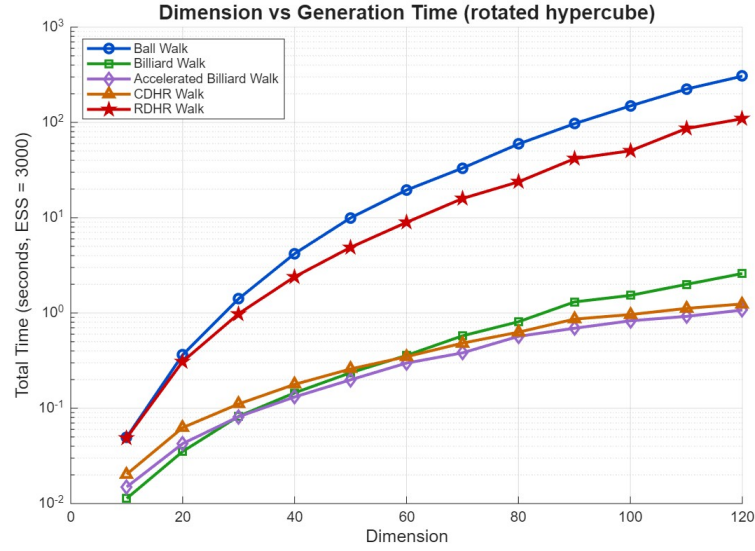
For this first experiment we will test the time each random walk needs in order to reach 3000 Ess score for a hypercube. The identity of this experiment is:

- ▶ Polytope: Hypercube (rotated by 53° over all planes)
- ▶ Dimensions: 10 to 120
- ▶ ESS threshold: 3000
- ▶ ESS tolerance: up to +100
- ▶ Final PSRF: <1.003
- ▶ Distribution: Uniform
- ▶ Random Walk: Varied
- ▶ Output: Graph of dimension vs time

As shown in [Figure 3.4](#), the sampling time increases with the dimension of the polytope for all methods. Among the tested algorithms, the BilliardWalk and AcceleratedBilliardWalk consistently exhibit the lowest sampling times, making them the most efficient in higher dimensions. In contrast, the BallWalk method struggles in higher dimensions, with its runtime growing rapidly as the dimension increases. The same is true for RDHRWalk, while CDHRWalk seems to be the fastest. Check out the great difference between Coordinate Direction Hit-and-Run and Random Direction Hit-and-Run. CDHR is the best while RDHR competes with ball walk for last place.

Lets focus on the 2 fastest walks, Accelareated Billiard and CDHR and push on to 1000 dimensinos. The results are displayed in [Figure 3.5](#).

Is is quite clear that Accelerated Billiard grows way faster than CDHR while dimensions are increasing. Lets try fitting a power law $y = ax^b$ to find the best equation describing this behaviour. The results are these:



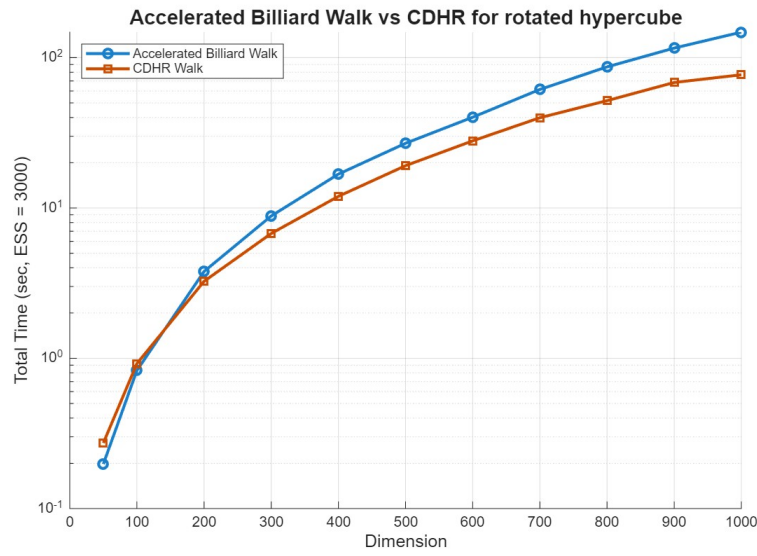
Σχήμα 3.4: This graph shows the results of running our random walk example. The polytope used is a hypercube rotated by 53° over all planes to reduce unfair advantage of some methods over a nicely centred cube. In x axis we have dimensions of the cube and in y axis the time it took the generator to create enough samples to reach 3000 ESS threshold (more accurately 3000-3100 ESS, so there is some infinitesimal error). Axis y is in log-scale and walk_len for non billiard methods is $\text{dim} \times 2$.

```
AcceleratedBilliardWalk: time = 0.000032 * dim ^ 2.21
CDHRWalk:               time = 0.000191 * dim ^ 1.91
```

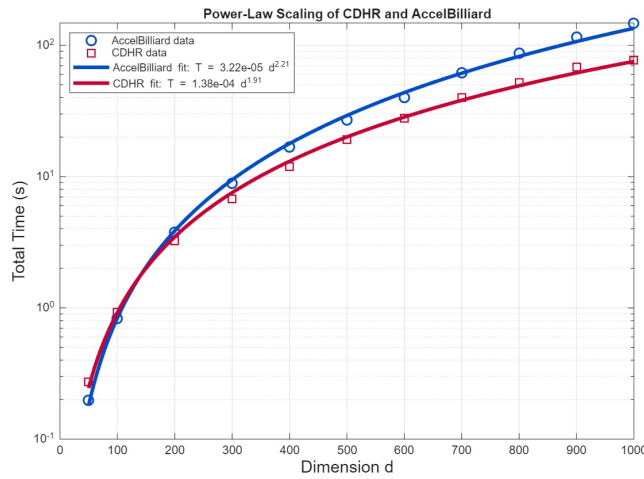
We draw these lines along with the experimental points to check the fitness. The following graph holds the results [Figure 3.6](#). The difference between the methods is now clear, ABWalk grows to the power of 2.1 while CDHR grows to the power of 1.9. The expected advantage of coordinate direction hit and run when sampling from a nice cube is verified.

3.10.2 Time trial for 500-hypercube and varied ess threshold

In this section, we aim to study the time it takes for some method to reach a specific ESS threshold. Of course linear dependence is expected i.e. we will reach 4k ess in double the time it takes reach 2k ess. All we have to do is draw a graph of ess threshold vs time to generate enough points. Lets choose Accelerated Billiard walk that is considered on of the best sampling methods.



Σχήμα 3.5: Accelerated Billiard Walk vs Coordinate Direction Hit-and-Run for up to 1000 dimensions hypercube. Same parameters as before.

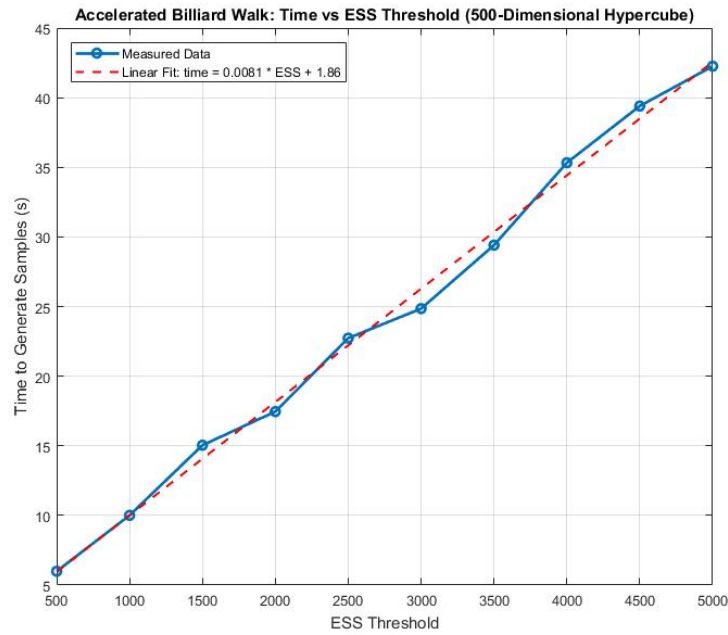


Σχήμα 3.6: Sampling times (needed to reach 3k ess for a hypercube) as a function of dimension for Accelerated Billiard Walk $time = 3.22 \cdot 10^{-5} \cdot \dim^{2.21}$ and CDHR Walk $time = 1.38 \cdot 10^{-4} \cdot \dim^{1.91}$.

- Polytope: Hypercube
- Dimensions: 500
- ESS threshold: Vaired

- ESS tolerance: up to +100
- Final PSRF: <1.003
- Distribution: Uniform
- Random Walk: Accelerated Billiard Walk
- Output: Graph of ess threshold vs time

Graph [Figure 3.7](#) confirms our results.



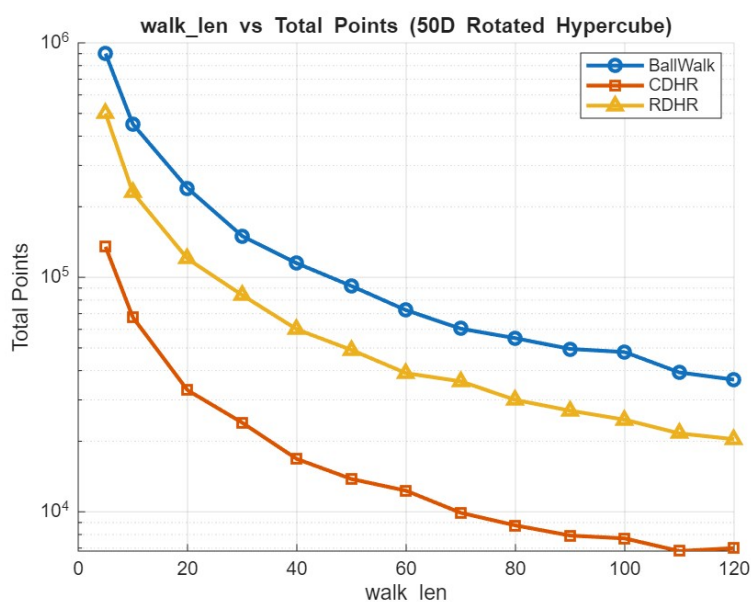
Σχήμα 3.7: Sampling time as a function of ess threshold for Accelerated Billiard Walk. The linear equation fitted is $time = 8.1 \cdot 10^{-3} \cdot ESS + 1.86$. The hypercube has 500 dimensions.

3.10.3 Walk length vs total points for constant ess threshold

Walk length is quite an important parameter as it controls how well each subsequent sample is mixed compared to its previous neighbour. Selecting a good walk length for each method is crucial for converging as fast as possible. For the billiard walk methods walk length must be equal to 1 by default. But, for ball walk and hit and run methods different walk length provide different results. Lets focus on a 50 dimensional hypercube, rotated and ess threshold of 3000.

- Polytope: Hypercube (rotated)
- Dimensions: 50
- ESS threshold: 3000
- ESS tolerance: up to +100
- Final PSRF: <1.003
- Distribution: Uniform
- Random Walk: Ball, CDHR, RDHR
- Output: Graph of walk length vs total points

You can check out the results at graph [Figure 3.8](#).



Σχήμα 3.8: Total points required to reach 3k ess vs walk length. Time it took to generate enough samples in independent of walk length, around 0.2s for CDHR, around 5s for RDHR and around 10s for ball walk. The trend is the same for all 3 methods. As walk length increases, fewer samples are required to reach target ess but it takes slightly longer to generate them. Thus, time remains constant. What is important though is that the fewer the points the faster it is to calculate ess at every iteration. Thus a big walk length is preferable. Choosing $\text{dim} \times 2$ like we did during sampling sounds like a perfect choice.

3.10.4 Time trial for barrier functions walks for hypercube

For this experiment we will test the three random walks offered by Volesti that use geometric objects to determine where they will sample next. These are Dikin, John and Vaidya walks. For information about Dikin walk you can find in this section [2.4.8](#). Lets summarize all three methods here. Each random walk is defined by a different interior geometry:

- **Dikin Walk.** Uses the classical log-barrier

$$\phi(x) = - \sum_{i=1}^m \log(b_i - a_i^\top x) ,$$

whose Hessian

$$H(x) = \nabla^2 \phi(x)$$

defines the Dikin ellipsoid. Steps are drawn from a Gaussian scaled by $H(x)^{-1}$, corresponding to the Riemannian metric induced by the log-barrier.

- **John Walk.** Based on the geometry of the John ellipsoid, the maximum-volume ellipsoid inscribed in the polytope. The method approximates the isotropic shape of this ellipsoid through the contact points, yielding a globally well-conditioned proposal distribution.
- **Vaidya Walk.** Uses the Vaidya barrier, a leverage-score weighted log-barrier

$$\phi_V(x) = \sum_{i=1}^m w_i(x) \log(b_i - a_i^\top x) ,$$

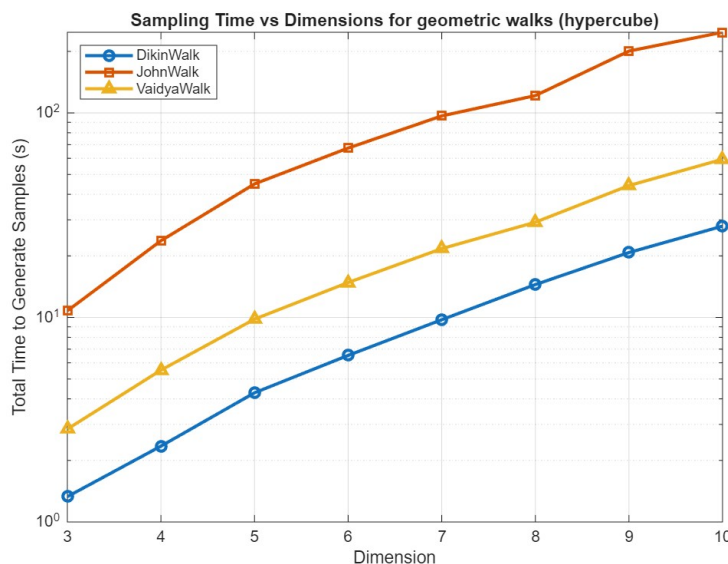
whose Hessian produces a more isotropic local metric than the classical Dikin barrier. This results in improved mixing while still retaining barrier-based geometry.

Experiment id:

- Polytope: Hypercube (rotated)
- Dimensions: 3 to 10
- ESS threshold: 3000
- ESS tolerance: up to +100

- Final PSRF: <1.003
- Distribution: Uniform
- Random Walk: Dikin, John, Vaidya

Results at [Figure 3.9](#).



Σχήμα 3.9: Total time to generate enough samples to reach 3k ess for Dikin, John and Vaidya walks. These methods rely on geometric objects for sampling and are thus quite expensive. The reason we study them separately from the rest of the methods is that they are orders of time more expensive to sample from a hypercube.

If we try to fit a power law using the experimental samples, the result is:

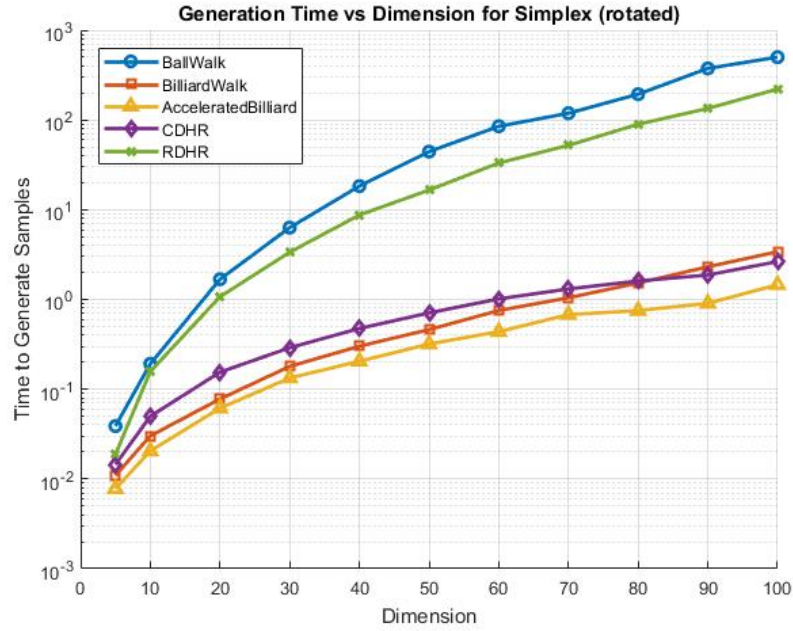
```
Dikin: time = 0.0566 * dim ^ 2.17
John: time = 0.5737 * dim ^ 2.74
Vaidya: time = 0.1689 * dim ^ 2.05
```

3.10.5 Time trial for Simplex and 3k ess threshold

All of the previous discussions were based on hypercubes. It is the most simple of polytopes to sample from and thus is useful to develop and optimize the code. Plus, the expected results are well known so the reliability of our code is also confirmed. Time to move to more complicated polytopes, like the simplex:

- Polytope: Simplex (rotated)
- Dimensions: 5 to 100
- ESS threshold: 3000
- ESS tolerance: up to +100
- Final PSRF: <1.003
- Distribution: Uniform
- Random Walk: Varied

The batch sizes needs some twining since each walk now needs quite more samples to reach 3k ess. The results are presented in the following graph [Figure 3.10](#).



Σχήμα 3.10: Total time to generate enough samples to reach 3k for various methods. Polytope is a simplex as generated by Volesti, but rotated by 45° degrees to avoid over fitting of any particular walk. The graph uses log y axis.

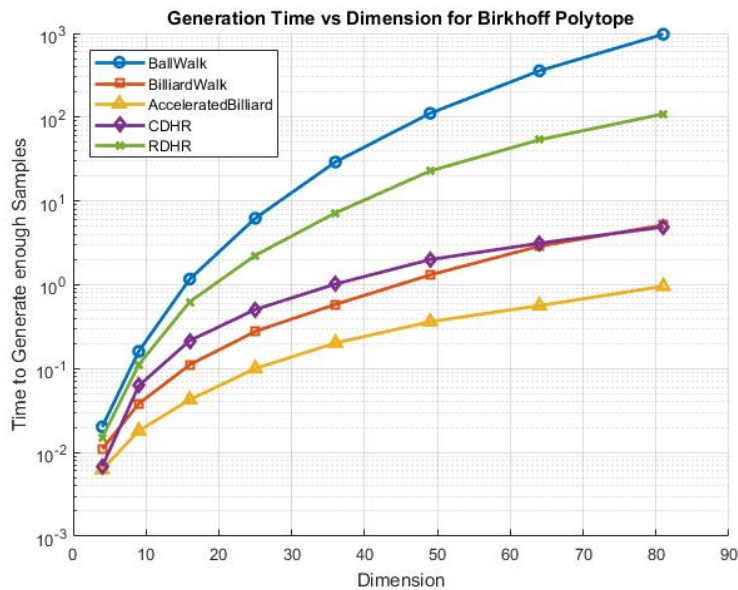
3.10.6 Time trial for Birkhoff and 3k ess threshold

In this section we try a birkhoff polytope for all our methods. The Birkhoff polytope B_n is a set of all $n \times n$ doubly stochastic matrices. The dimension of such

a polytope is $\dim(B_n) = (n - 1)^2$, where n will be the number of dimension we pass as parameter. So when we give $\dim = 3$, the polytope that is created will have 4 dimensions, while if we give 10 input the polytope will have 81 dimensions.

- Polytope: Birkhoff (rotated)
- Dimensions: 4 up to 100
- ESS threshold: 3000
- ESS tolerance: up to +100
- Final PSRF: <1.003
- Distribution: Uniform
- Random Walk: Varied

Results here [Figure 3.11](#).



Σχήμα 3.11: Total time to generate enough samples to reach 3k for various methods. Polytope is a Birkhoff as generated by Volesti, but rotated by 45° degrees to avoid over fitting of any particular walk. The graph uses log y axis.

Κεφάλαιο 4

PolytopeWalk

4.1 PolytopeWalk Library

In this section, we will explore another package dealing with polytope sampling using random walks. As we read in the documentation, PolytopeWalk is a modern computational framework designed for efficient sampling within convex polytopes using MCMC methods. Developed in C++ for performance and equipped with a convenient Python interface, the library provides a unified environment for generating samples uniformly distributed over polytopes of arbitrary shape and dimension.

The package not only implements a variety of state-of-the-art random walk algorithms, but also automates key preprocessing and initialization steps that are critical for reliable high-dimensional sampling. In particular, PolytopeWalk performs Facial Reduction to transform potentially degenerate or lower-dimensional constraint systems into full-dimensional form, and employs interior-point initialization strategies to ensure the random walk begins in a feasible region of the polytope.

Implementations of several basic and advanced MCMC random walks are included:

- ▶ Ball Walk
- ▶ Hit-and-Run
- ▶ Dikin Walk
- ▶ Vaidya Walk

- John Walk
- Lee–Sidford Walk

These are provided in both dense (full-dimensional) and sparse (constraint-based) formulations, allowing users to balance numerical efficiency and memory usage depending on problem structure. PolytopeWalk thus serves as a flexible, high-performance tool for exploring and analyzing high-dimensional convex bodies through random walks. We will use PolytopeWalk for sampling from high-dimensional polytopes, comparing the behavior and convergence properties of its various walk implementations and organize a standoff against Volesti.

Further information and source code are available in the official GitHub repository: [PolytopeWalk GitHub Repository](#).

4.2 Installing PolytopeWalk

As mentioned, Polytope Walk is equipped with a convenient Python interface so its ready to use without the need to compile any `c++` files ourselves. For this reason, we will use the most guaranteed and basic way to use it. Follow the steps below. These commands will work on Windows and git bash terminal. You might need variations in order to run in the terminal or system of your choice.

1. Create a new folder. I named mine PolytopeWalk for reasons beyond your understanding.
2. Create a python virtual environment using:

```
-python -m virtualenv (name)
```

If you are looking for originals names, call it PolyWalk.

3. To activate this virtujal environment type:

```
source PolyWalk/Scripts/activate
```

4. Install the necessary libraries:

```
pip install -U polytopewalk
pip install numpy
pip install matplotlib
pip install scipy
pip install arviz
```

I might be forgetting something, so if you get a module not found error then you might need to install something more

5. We are now ready to use PolytopeWalk.

4.3 Running polytope walk tutorial example

To understand the library and its usages, its always useful to analyse the examples provided by the authors. To that end, there is a folder called examples containing a couple of very interesting jupyter files using the main functions offered by the package. Lets have a look at the dense walks example.

We begin by importing the necessary classes and functions:

```
# Import Standard Random Walks
from polytopewalk.dense import BallWalk, HitAndRun
# Import Barrier Random Walks
from polytopewalk.dense import DikinWalk, VaidyaWalk, JohnWalk,
                                DikinLSWalk

# Import Initialization Algorithm
from polytopewalk.dense import DenseCenter
# Import Facial Reduction Algorithm
from polytopewalk import FacialReduction
# Import End-to-End Pipeline
from polytopewalk import denseFullWalkRun
```

We are already familiar with ball-walk ([2.4.4](#)), hit-and-run ([2.4.6](#)) and dikin-walk ([2.4.8](#)). The rest of the barrier random walks uses the same logic as dikin walk. They exploit barrier functions (smooth convex functions that blow up at the boundary) to define local geometry, improving movement and mixing in high-dimensional spaces. Remember that Dikin walk uses the Hessian of the logarithmic barrier. VaidyaWalk adapts the local metric further by incorporating leverage scores, while JohnWalk uses an approximation of the John ellipsoid (max-volume inscribed ellipsoid),

providing yet another adaptive step distribution. Lastly, DikinLSWalk is a Lee–Sidford variant of the Dikin walk with an improved theoretical mixing bound.

Some generation functions follow, setting up known polytopes in H-representation in order to be used by the walks. For example, the following function creates dim dimensional cubes:

```
def generate_hc(dim):
    dim = dim // 3
    col = np.concatenate((np.arange(0, dim), np.arange(0, dim), np.arange(dim, 3 *
    row = np.concatenate((np.arange(0, 2 * dim), np.arange(0, 2 * dim)))
    ones = np.ones(dim)
    negative_ones = np.repeat(-1, dim)
    data = np.concatenate((ones, negative_ones, ones, ones))
    A = sparse.csr_array((data, (row, col)), shape=(2 * dim, 3 * dim))
    b = np.array([[1]] * (2 * dim))
    x = np.concatenate((np.array([[0]] * (dim)), np.array([[1]] * (2 * dim))))
    return x, A, b, 2 * dim, 'hypercube'
```

The function internally builds an embedding of the cube in a higher dimensional space, specifically a space of size $3 \cdot \text{dim}$. dim is divided by 3 at the beginning because later it is multiplied by 3 when constructing the constraint matrix shape $(2 \cdot \text{dim}, 3 \cdot \text{dim})$. The matrix representing a hypercube is quite sparse. Most values are 0 except one value per line corresponding to the variable describing each dimension. The resulting shape of A is $(2 \cdot \text{dim}, 3 \cdot \text{dim})$ ie $3 \cdot \text{dim}$ dimension, the same value passed as argument.

Next, the parameters of the walks are defined:

```
THIN = 10          # keep every 10th sample
ITER = 100_000    # total samples
BURNIN = 1_000    # discard initial samples
SEED = 8888       # Used for pseudo number generation
```

These parameters are cleverly used to improve the independence of the produced samples. First of all, some initial points are discarded in order to avoid these points being correlated to the starting point inside the polytope. Then, there is a thin parameter that tells us after how many samples to store the point. This way, each stored sample is further away in the walk from the previous one decreasing autocorrelation. Ofcourse, there is a cost we pay regarding the total samples returned by the walk but it is

generally better to have less non-correlated samples than many correlated ones. Total number of generated samples are defined by iter.

These lines follow:

```
dikin = DikinWalk(r=0.5)
vaidya = VaidyaWalk(r=0.5)
john = JohnWalk(r=0.5)
```

Each of these lines creates an instance of a barrier-based random walk. The parameter r controls the step radius.

A quite important line follows:

```
fr = FacialReduction()
```

Before running any random walk, we apply a facial reduction algorithm. But what is this algorithm about? We have not seen anything similar so far. Lets check the source file. Go to `src/Utils/FacialReduction.cpp`. There, an algorithm is implemented manipulating the polytope. We will not show the code here, go check out github. We will make an attempt to explain what this Facial Reductio algorithm does:

1. The algorithm first attempts to find a vector y satisfying

$$A^T y = [0, z], \quad \langle b, y \rangle = 0, \quad z \geq 0, \quad z \neq 0,$$

with $x_{\dim} = d - k$ being the expected dimension of the reduced system. The existence of such a vector indicates that the current system is not strictly feasible, and that certain constraints are redundant or linearly dependent. This step is formulated and solved as a sparse linear program (LP).

2. Once a nonzero vector z is found, the algorithm constructs a sparse selection matrix V that extracts the coordinates corresponding to zeros in z . This effectively defines a projection onto the subspace where the system remains feasible.
3. The reduced system AV is analyzed using a sparse QR decomposition to identify linearly independent rows. A projection matrix P is formed to retain only the independent constraints, removing redundancy and ensuring numerical stability.

4. The process of finding z , constructing V , and applying P is repeated recursively until no further degeneracy is detected (i.e., the LP in `findZ` fails to produce a valid z). The product of all V matrices is accumulated in `savedV`, representing the total transformation applied to the original system.
5. Finally, the algorithm combines all reduction steps and, if a dense formulation is requested, performs a QR decomposition to transform the reduced sparse system into a full-dimensional dense system (A', b') suitable for dense random walk samplers. The procedure also stores auxiliary matrices Q and z_1 that allow converting between the reduced and original coordinate systems.

In summary, the facial reduction algorithm systematically projects the original constraint system (A, b) into its minimal representation by removing redundant and dependent inequalities. This guarantees that the resulting polytope is strictly feasible and full-dimensional, which is a necessary condition for the convergence and numerical stability of the subsequent random walk sampling algorithms. In many cases, the constraint system (A, b) can be degenerate, meaning that the feasible region lies in a lower-dimensional affine subspace of $\mathbb{R}^d$. If this degeneracy is not removed, the barrier-based random walks (such as Dikin, Vaidya, or John Walks) may fail because the local metric defined by the barrier Hessian becomes singular.

Back to the example, time to create the polytope, that is later fixed using facial reduction:

```
_, A, b, k, name, dim = generate_simplex(5)
polytope = fr.reduce(A, b, k, sparse=False)
dense_A = polytope.dense_A
dense_b = polytope.dense_b

dc = DenseCenter()
init = dc.getInitialPoint(dense_A, dense_b)
```

These lines are more familiar for us. A polytope is created using some generator function that is then passed by the facial reduction class. Then, A and b are retrieved and are passed to another class, destined to find an interior point by solving the linear program. That all. You can check out `DenseCenter` at `src/Utils/DenseCenter.cpp` for further information.

Time to run the walks by calling:

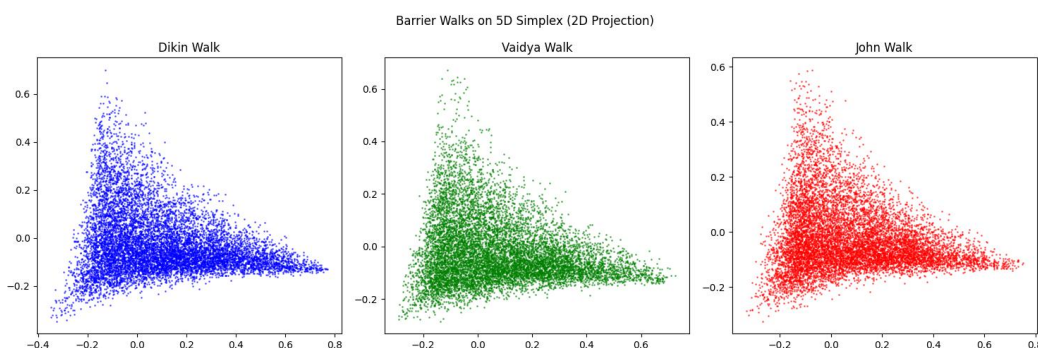
```
dikin_res = dikin.generateCompleteWalk(ITER, init, dense_A, dense_b,
```

```

burnin=BURNIN, thin=THIN, seed=SEED)
vaidya_res = vaidya.generateCompleteWalk(ITER, init, dense_A, dense_b,
burnin=BURNIN, thin=THIN, seed=SEED)
john_res = john.generateCompleteWalk(ITER, init, dense_A, dense_b,
burnin=BURNIN, thin=THIN, seed=SEED)

```

Each method needs the number of iterations, the initial point, matrices A and b, and the rest of the parameters we defined before. The samples will be returned in a list and are ready to be used. Lets check out the exit of this example. We actually copied the jupyter code to a python file so we just need to type: python dense-example.py to run it. You can use the jupyter file if you like. In any case, this is the exit.



Σχήμα 4.1: Exit of the dense example of PolytopeWalk. We did made some minor changes like printing all walks together.

4.4 Creating our own polytope walk example

The knowledge gained by the tutorial will now be used to code our own example. Our main goal is to use the barrier function walks to sample from polytopes and explore the timings and the convergence of the methods.

4.4.1 Setting up the diagnostic functions

An important observation that can be made for Polytope Walk, is that it does not implement its own ess and psrf function, but they rather use a python package called arviz. The thing is that arviz expects multiple Markov chains in order to calculate

these metrics which of course is the correct way to work. That means that we will have to run the random walks multiple times and we choose to avoid this fact.

As demonstrated in Volesti, these metrics can be evaluated by manipulating the samples of a single chain. For ess an autocorrelation factor can be calculated, while for psrf the single chain can be randomly split in half and these two halves will be compared in order to determine stationary convergence. Using this logic, we can make the use of the arviz ourselves with only one chain.

The end product is the following:

```
import numpy as np
from numpy.fft import fft, ifft
import arviz as az

def ess(samples):

    ds = az.convert_to_dataset(samples[None, :, :])
    ess_vals = az.ess(ds).to_array().values
    ess_min = float(np.nanmin(ess_vals))
    return ess_min

def psrf(samples):

    arr = np.asarray(samples)
    N, d = arr.shape
    half = N // 2

    # Split the single chain into two halves
    chains = np.stack([arr[:half, :], arr[half:, :]], axis=0)

    # Convert to ArviZ dataset and compute R-hat
    ds = az.convert_to_dataset(chains)
    rhat_vals = az.rhat(ds).to_array().values

    # Handle NaNs safely
    if np.all(np.isnan(rhat_vals)):
        return np.nan

    return float(np.nanmax(rhat_vals))
```

For more information on these metrics check [2.5](#). Also check the Volesti example

to remember how the metrics are used.

4.4.2 Setting up the polytope generators

There is not to say in this section. For now we will use the functions provided by the tutorials. We presented the hypercube in the previous sections. In addition, these are also available:

```
def generate_hc(dim)
def generate_simplex(dim)
def generate_birkhoff(dim)
```

4.4.3 Basic set-up of our example

Great, we are now ready to apply the random walks and measure their runtime plus check their convergence. To run the walks, we use the commands presented in the tutorial. The end product looks like this:

```
import time

from polytope_generators import *
from polytope_diagnostics import *

from polytopewalk.dense import BallWalk, HitAndRun
from polytopewalk.dense import DikinWalk, VaidyaWalk, JohnWalk,
                                DikinLSWalk

from polytopewalk.dense import DenseCenter
from polytopewalk import FacialReduction
from polytopewalk import denseFullWalkRun

# Parameters of our walk
THIN = 10          # keep every thin-th sample
ITER = 100_000     # total samples
BURNIN = 1_000     # discard initial samples
SEED = 8888        # Used for pseudo number generation

total_samples = int((ITER - BURNIN) / THIN)
print(f"\nFinal total samples: {total_samples:,}")
```

```

dikin = DikinWalk(r=0.5)
vaidya = VaidyaWalk(r=0.5)
john = JohnWalk(r=0.5)

fr = FacialReduction()

# Polytope generation. Use one of the fucntions in generators.py
_, A, b, k, name, dim = generate_cube(5)
print(f"Polytope {name} ready. Number of dimensions: {dim}")

polytope = fr.reduce(A, b, k, sparse=False)
dense_A = polytope.dense_A
dense_b = polytope.dense_b

# Find initial polytope point
dc = DenseCenter()
init = dc.getInitialPoint(dense_A, dense_b)

# For each method measure the runtime.
# Dikin Walk
start = time.time()
dikin_res = dikin.generateCompleteWalk(ITER, init, dense_A, dense_b,
                                       burnin=BURNIN, thin=THIN, seed=SEED)
runtime_dikin = time.time() - start

# Vaidya Walk
start = time.time()
vaidya_res = vaidya.generateCompleteWalk(ITER, init, dense_A, dense_b,
                                       burnin=BURNIN, thin=THIN, seed=SEED)
runtime_vaidya = time.time() - start

#John Walk
start = time.time()
john_res = john.generateCompleteWalk(ITER, init, dense_A, dense_b,
                                       burnin=BURNIN, thin=THIN, seed=SEED)
runtime_john = time.time() - start

# Find ess using our Volesti like function
ess_dikin = ess(dikin_res)
ess_vaidya = ess(vaidya_res)
ess_john = ess(john_res)

# Find PSRF (or r_hat) using ou Volesti like function
psrf_dikin = psrf(dikin_res)
psrf_vaidya = psrf(vaidya_res)

```

```

psrf_john = psrf(john_res)

# Print results
print(f"Dikin Walk:  ESS = {ess_dikin:.0f}, PSRF = {psrf_dikin:.5f},
      Time = {runtime_dikin:.8f}s")
print(f"Vaidya Walk: ESS = {ess_vaidya:.0f}, PSRF = {psrf_vaidya:.5f},
      Time = {runtime_vaidya:.8f}s")
print(f"John Walk:   ESS = {ess_john:.0f}, PSRF = {psrf_john:.5f},
      Time = {runtime_john:.8f}s")

```

Note that we have not included the code for the rest of the methods since its quite repetitive. The basic logic is simple. Set up the parameters, create the walk instances, use Facial Reduction as presented before, find initial polytope point, run the walks, calculate the runtime and the metrics, return the results. Confirm that the walks have converged by hoping that psrf value is close to 1, check the minimum independent points per some dimension by looking at ess and save the time value for some later graph.

The output of this basic example is the following:

```

Final total samples: 9,900
Polytope hypercube ready. Number of dimensions: 3
Ball Walk:      ESS = 4893, PSRF = 1.00018, Time = 0.03235745s
Hit-and-Run:    ESS = 9900, PSRF = 0.99991, Time = 0.31133127s
Dikin Walk:     ESS = 1207, PSRF = 1.00324, Time = 0.23393559s
Vaidya Walk:    ESS = 836,  PSRF = 1.00075, Time = 0.42393088s
John Walk:      ESS = 1129  PSRF = 1.00114, Time = 1.95301199s

```

So, we set 100000 iterations, with thin 10 and burn 1000. That will give us a total of 9900 samples at the end for each method. Surprisingly, hit and run ess have returned a perfect score. We shall see if this nice score will remain in higher dimensions.

4.4.4 Run for multiple dimensions and up to 3k ESS

We modify the code to run each method for up to 3k ESS. The function that does exactly that is the following:

```

def run_until_target_ess(walk, name, start_point, A, b,
                        target_ess=3000, batch_iter=10000,
                        r=0.5, thin=10):
    """
    Runs the random walk in batches until the Effective Sample
    Size (ESS) reaches the target_ess.
    """
    print(f"[{name:<11}] Batch size: {batch_iter} , r: {r} , thin: {thin}")

    # Storage for the cumulative chain
    all_samples = None
    total_time = 0.0

    # State tracking
    current_init = start_point.copy()
    current_ess = 0.0
    batch_count = 0
    max_batches = 200 # Safety break: stop if we loop 100 times

    while current_ess < target_ess and batch_count < max_batches:
        batch_count += 1

        # We only use burn-in for the very first batch.
        # Subsequent batches are continuations, so burnin=0.
        current_burnin = 100 if batch_count == 1 else 0

        # Start timer
        start_t = time.time()

        # Run the C++ backend sample generator
        new_batch = walk.generateCompleteWalk(
            batch_iter,      # How many samples
            current_init,    # Starting point
            A, b,            # Matrices
            burnin=current_burnin, # Burn first points
            thin=thin,       # Only keep 1 in 10 samples
            seed=8888 + batch_count # Change seed slightly
                                   # to ensure randomness
        )

        # Calculate time the generator is running
        batch_time = time.time() - start_t
        total_time += batch_time

        # Stack new samples onto the history

```

```

    if all_samples is None:
        all_samples = new_batch
    else:
        all_samples = np.vstack([all_samples, new_batch])

    # UPDATE STATE
    # The next batch must start where this one ended
    current_init = all_samples[-1, :]

    # Calculate ESS
    try:
        current_ess = ess_arviz(all_samples)
    except Exception as e:
        # If ESS crashes
        current_ess = 0

    # Log progress
    print(f"[{name:<11}] Batch {batch_count}: Samples={len(all_samples):,},//
          // ESS={current_ess:>5.0f}, TotalTime={total_time:>6.5f}s", end='\r')

    # Print Done
    print(f"\n[{name:<11}]    -> DONE. Final ESS: {current_ess:.0f}")

    return current_ess, total_time, all_samples

```

As parameters to this function we pass the walk, the name of the walk, the starting point of the random walk (found using the the `getInitialPoint` provided by the library), `A` and `b` the matrices of the polytope created by our `generate_polytope` functions, the `ess_threshold`, the batch size and finally the `thin` parameter. The code works with the same logic as in *Volesti*. Start sampling points from the polytope until their collective ESS becomes more than 3k.

Special attention is paid to every batch of samples in order to continue expand the initial chain. To that end, we pass as initial point of the next batch of points, the last sample of the previous batch. We also change the seed of the random numbers to avoid repeating the same random sequence. This way, all samples ends up belonging to a big connected MCMC chain and the diagnostics can run accurately on them.

Inside main, we run this function like this:

```

# Here type the dimensions you want to test
dims = [20]
results = []

```

```

for dim in dims:

    print(f"\n{' '*40}")
    print(f"***Running for dimension {dim} and ESS {TARGET_ESS}***")

    # Calculate radii for each method
    r_ball    = get_radius("ball", dim)
    r_hit     = get_radius("hit", dim)
    ....
    # Get thin parameter
    thin_ball = get_thin("ball", dim)
    thin_hit  = get_thin("hit", dim)
    ....
    # Construct walk objects
    ball      = BallWalk(r=r_ball)
    hitrun    = HitAndRun(r=r_hit)
    ....
    # Generate rotated polytope hypercube
    init, dense_A, dense_b, name = generate_rotated_hypercube_direct //
                                   // (dim, angle_deg=53)
    print(f"Generated {name}.")

    dc = DenseCenter()
    init = dc.getInitialPoint(dense_A, dense_b)

    # BALL WALK
    #bs_ball = get_batch_size("ball", dim)
    #e_ball, t_ball, samples_ball = run_until_target_ess(ball, //
    // "Ball Walk", init, dense_A, dense_b, TARGET_ESS,
    // bs_ball, r_ball, thin_ball)
    #psrf_ball = psrf_arviz(samples_ball)
    ....
    results.append({
        "dim": dim,

        "Ball_ESS": e_ball,
        "Ball_PSFR": psrf_ball,
        "Ball_Time": t_ball,
        ....

```

4.5 Running our example

4.5.1 Walks vs r parameter

Lets examine `ball_walk.cpp` where the code for the (dense) ball walk is located. In there we see this line:

```
VectorXd new_x = generateGaussianRVNorm(n, gen) * R/sqrt(d) + x;
```

This implies that:

$$x_{new} = x_{old} + \frac{R}{\sqrt{d}} \cdot \xi$$

where ξ is a standard normal vector. R dictates how far the walker attempts to jump in each step. This R is the same r we pass as argument to the walker.

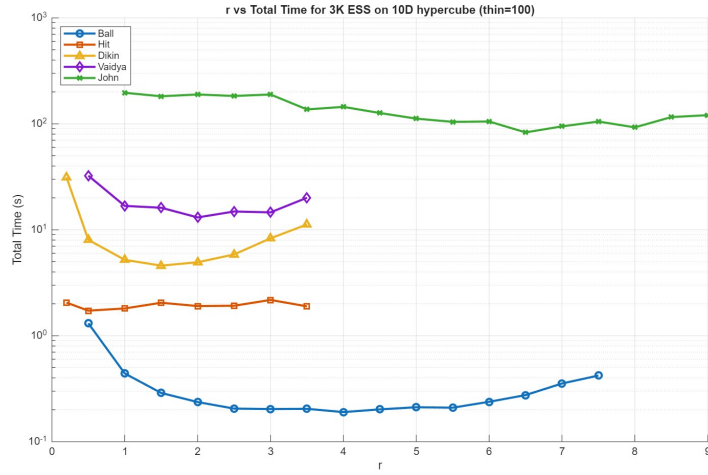
Sometimes, based on polytope dimensions and the value of r (whether it is too big) we can get the following error while trying to calculate ESS:

```
RuntimeWarning: invalid value encountered in scalar divide
(between_chain_variance / within_chain_variance + num_samples - 1) / (num_samples)
```

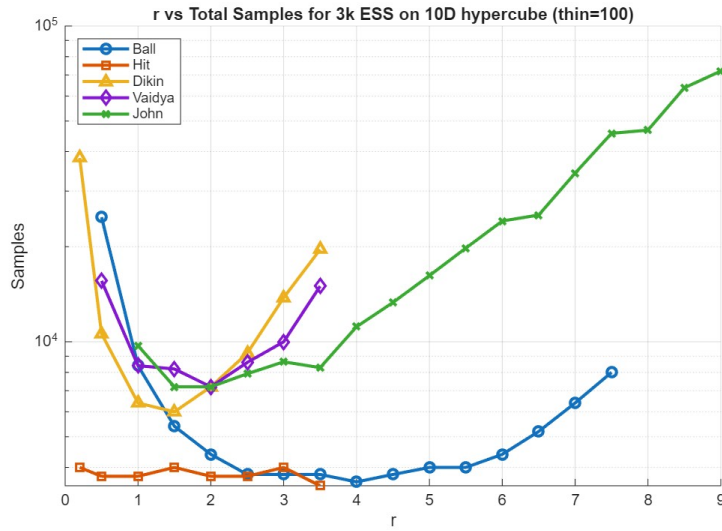
One example of this error was for $r = 3.5$ and $d = 3$ or $r = 8.5$ and $r = 5$.

The distance of each new point is analogous to R . If R is too big compared to the polytope, then the new point almost always lands outside the boundaries $Ax \leq b$. Thus, it is rejected and you add the previous point to the list of samples (the if statement in the ball walk code is skipped). If this happens too many times in a row, then all the samples will be the same, so arviz tries to calculate the ESS, the variance is zero and thus we get the error.

All of this discussion leads us to a dynamic r . For each dimension, a different r seems to be the best choice. For a 10D hypercube, check [Figure 4.2](#) and [Figure 4.4](#).



Σχήμα 4.2: PolytopeWalk vs r parameter for 10D rotated hypercube, ESS threshold 3000 and thin parameter 100. r parameter is usually used to determine the range of a candidate new sample. Small r lead to the chains getting stuck in one place while big r ends up rejecting most of the new samples. To that end, there seems to be a sweet spot for r where it offers the best mixing time and the most independent samples. Based on this graph, check 4.5.2 for the choices we made.

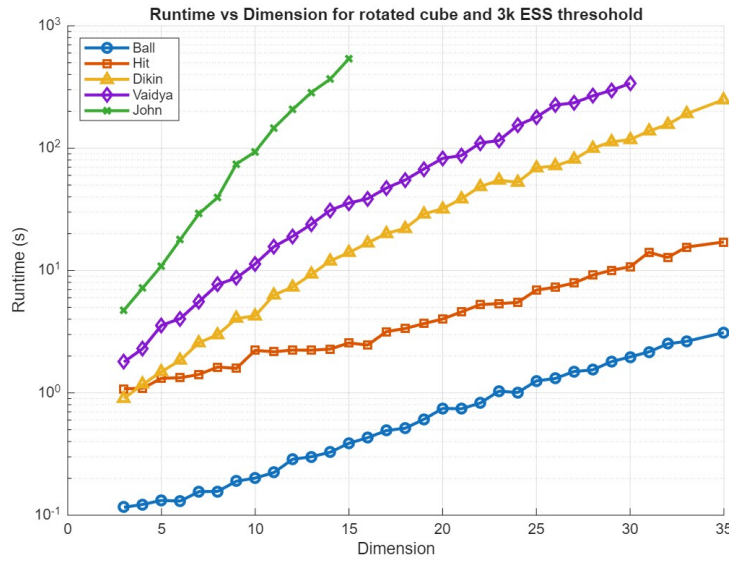


Σχήμα 4.3: PolytopeWalk vs r parameter for 10D rotated hypercube, ESS threshold 3000 and thin parameter 100. In this graph we see total number of samples while in Figure 4.2 we see Total Time.

4.5.2 Choosing r parameter

- **Ball Walk:** Based on the optimum value of r for 10 dimensions and the failure of this optimum at 3 dimensions, after a series of experiments we choose $r = \sqrt{d}$ for the ball walk.
- **Hit and Run:** For hit and run, r parameter has no significant part to play. It is actually ignored by the source code. That is to be expected from hit and run since we choose a uniform random point on a chord of the polytope. R in this case is linked to the distance of this cord.
- **Dikin/Vaidya/John Walk:** These methods scan the local geometry. If the polytope gets narrow, they automatically shrink their step. If it gets wide, they expand. Because they effectively normalize the space themselves, some specific r always means a safe step to the nearest wall, regardless of whether you are in 3 dimensions or 100 dimensions. To that end, based on the 10D experiment we choose $r_{dikin} = 1.5$, $r_{vaidya} = 2.0$ and $r_{john} = 5.0$.

4.5.3 Time trial for hypercube



Σχήμα 4.4: Time trial for rotated hypercube. ESS threshold is 3k and thin parameter is 100. The plot is dimension vs time to reach this 3k threshold. Axis y is log.

Κεφάλαιο 5

PolytopeSampler

As we read from the readme file, PolytopeSampler is a Matlab implementation of constrained Riemannian Hamiltonian Monte Carlo for sampling from high dimensional distributions on polytopes. It is able to sample efficiently from sets and distributions with more than 100K dimensions. Check it out here [PolytopeSampler GitHub Repository](#). To learn more about how a Hamiltonian Monte Carlo method works, check this out [2.4.14](#).

5.1 Installing PolytopeSampler

In order to use this MatLab library you need Matlab. Who would have guessed right? If you are part of any academic institution, there is high chance that you have free access to MatLab. Check availability with your institution. In any case, find a way to install Matlab. For this project, we use Matlab R2025b.

Next, create a folder somewhere, but make sure its path contains only ASCII characters. Inside this folder download the source code from the github repository (link in introduction of this chapter). Either use git and clone it or go to the website and download and extract the zip.

After you are finished downloading, open Matlab and select this folder you created and contains the library. You will see two .m files. One of them is called `initSampler.m` and it is essential to tell Matlab where it can find the library whenever you call it by adding the package folders to Matlab's path. The other file is called `demo.m` and contains some useful examples. If you run this file some graphs will

pop up and a useful output will be displayed in the command window. Lets check out this demo so that we understand how the library is used.

5.2 Running demo tutorial

This demo is split into several parts, each part executing sampling over some polytope. Lets begin with the first part of the demo.

```
% Example 1: Sample uniform from a simplex
initSampler

P = struct; d = 100;
P.Aineq = ones(1, d);
P.bineq = 1;
P.lb = zeros(d, 1);
o = sample(P, 2000);
s = o.samples;
histogram(sum(s), 0.9:0.005:1)
title('distribution of l1 norm of simplex');
distribution_test(o, struct('toPlot', true));
drawnow()
```

First, it adds the package folders to MATLAB's path. You can run `initSampler.m` once and forget about it or you can leave it like this. It doesn't matter that much. Next, an empty struct (or class) P is created and a dimension is chosen. The polytope follows. These two lines:

```
P.Aineq = ones(1, d);
P.bineq = 1;
P.lb = zeros(d, 1);
```

create a simplex as they set the polytope inequalities $A \cdot x \leq b$ to $A = [1, 1, 1, \dots, 1]$, $b = 1$ and the linear bounds to zeroes ie. all x must be non-negative $x_i \geq 0$. The sampler is call by the command:

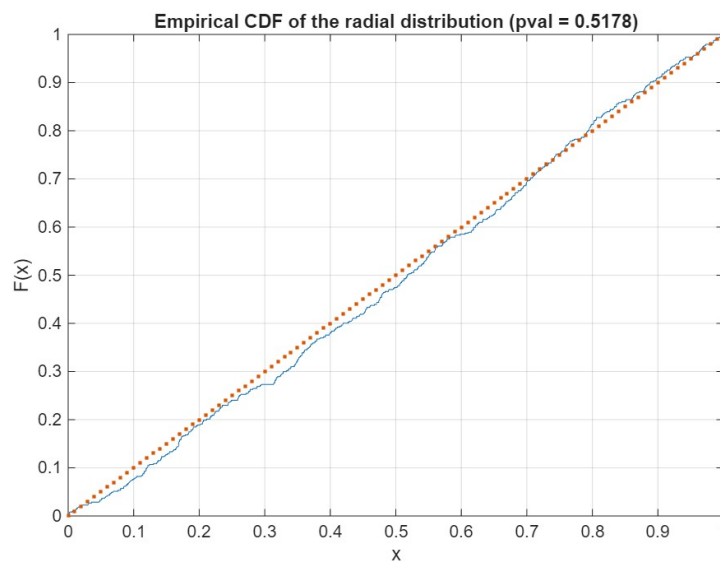
```
o = sample(P, 2000);
```

The function sample will: 1) Find an initial feasible point, 2) Run constrained Hamiltonian Monte Carlo and 3) Test convergence of the sampling algorithm by computing Effective Sample Size (ESS) and terminate when $ESS \geq N$. If the target distribution is uniform, a uniformity test will also be performed. The output is a struct `o`, which stores samples generated in `o.samples` and a summary of the sample in `o.summary`. `o.samples` is an array of size `dim x #steps`.

Running this demo will yield the following outputs. First, during execution there is an output on the command window informing us of the progress. Here is a snapshot of this progress bar:

```
>> demo
Time spent | Time left | Progress | Est Samples | ...
00d:00:00:14 | 00d:00:00:01 | ##### | 1832 / 2000 | ...
```

The demo also returns a useful convergence metric in the form of a graph. Check it out here [Figure 5.1](#)



Σχήμα 5.1: Empirical CDF of the transformed radial variable for samples drawn from the 100-dimensional simplex by the demo mentioned above. The empirical curve closely follows the theoretical uniform CDF and the Kolmogorov–Smirnov test yields a non-small p-value, indicating that the sampler produces samples consistent with the uniform distribution on the simplex.

5.2.1 Understanding the result - the K-S statistic

Let us give some more background to what [Figure 5.1](#) tells us. This is the first time we encounter such a diagnostic metric, and it is worth understanding precisely what is being tested [\[2\]](#).

Let

$$\Delta_d = \left\{ x \in \mathbb{R}^d : x_i \geq 0, \sum_{i=1}^d x_i \leq 1 \right\}$$

denote the standard d -dimensional probability simplex. For a point $x \in \Delta_d$, we define its radial position r as its L_1 norm:

$$r = \sum_{i=1}^d x_i.$$

Geometrically, r measures how far x lies from the origin relative to the simplex boundary. The volume of a simplex scaled by a factor r is proportional to r^d . Therefore, when x is drawn uniformly from Δ_d , the random variable r follows a power-law distribution with cumulative distribution function (CDF):

$$F_r(r) = r^d, \quad 0 \leq r \leq 1.$$

The corresponding probability density function is:

$$f_r(r) = d r^{d-1}.$$

For a real-valued random variable X , the cumulative distribution function (CDF) is defined as

$$F_X(x) = \Pr(X \leq x).$$

The CDF fully characterizes the distribution of X . In particular, the Probability Integral Transform theorem states that if X has a continuous CDF F_X , then the random variable $U = F_X(X)$ follows a standard uniform distribution:

$$U \sim \text{Uniform}(0, 1).$$

In our setting, this implies that if we transform the observed radii using the theoretical CDF:

$$u = F_r(r) = r^d,$$

the resulting values should be uniformly distributed on $[0, 1]$. Graphically, the empirical CDF of these transformed values should lie on the diagonal line $y = x$.

PolytopeSampler computes the radii $r^{(1)}, \dots, r^{(N)}$ of the generated samples, applies the transformation $u^{(k)} = (r^{(k)})^d$, and plots the empirical cumulative distribution function $\hat{F}(u)$. To formally assess agreement with the target distribution, the sampler performs a Kolmogorov–Smirnov (K–S) test. The K–S statistic is defined as

$$D = \sup_{u \in [0,1]} |\hat{F}(u) - u|,$$

which represents the maximum vertical deviation between the empirical CDF and the theoretical uniform line.

A plot in which the empirical CDF tracks the diagonal closely, yielding a small D statistic and a non-significant p -value, indicates that the sampled points exhibit the correct radial distribution. This provides evidence that the Markov chain has mixed well and that the sampler is correctly exploring the "onion layers" of the high-dimensional body. Conversely, systematic deviations from the diagonal signal that the walker is getting stuck near the center (if the curve is above the diagonal) or near the boundary (if below).

5.2.2 Methodology and Implementation

The Geometry of High-Dimensional Volume

For any convex body $K \subset \mathbb{R}^d$ containing the origin, the volume scales as a power of the dimension. Specifically, if we scale the body by a factor $r \in [0, 1]$, the volume of the shrunken body rK is given by:

$$\text{Vol}(rK) = r^d \cdot \text{Vol}(K) \quad (5.1)$$

The Radial Distribution

If a point X is drawn uniformly at random from K , its "radial position" r (defined as its relative distance from the center to the boundary along the ray passing through X) follows a specific probability distribution. Since volume concentrates near the boundary in high dimensions, the probability of finding a point within a relative distance r corresponds exactly to the volume fraction at that distance:

$$P(R \leq r) = \frac{\text{Vol}(rK)}{\text{Vol}(K)} = r^d \quad (5.2)$$

Thus, the Cumulative Distribution Function (CDF) of the radial position is:

$$F_R(r) = r^d, \quad 0 \leq r \leq 1 \quad (5.3)$$

The Probability Integral Transform

To statistically test if samples are truly uniform, we utilize the Probability Integral Transform theorem. If a random variable R has a continuous CDF F_R , then the transformed variable $U = F_R(R)$ must follow a standard Uniform distribution on $[0, 1]$. Applying this to our setting:

$$U = R^d \sim \text{Uniform}(0, 1) \quad (5.4)$$

This transformation effectively collapses the complex high-dimensional geometry into a 1D variable that can be tested against a known standard distribution.

Algorithmic Implementation

Our implementation performs this diagnostic in distinct steps:

Empirical Centering

Since the analytic center of the polytope is often computationally expensive to compute, we approximate the center using the empirical mean of the generated samples:

$$c = \frac{1}{N} \sum_{i=1}^N x_i$$

We then conceptually shift the polytope constraints $Ax \leq b$ relative to this center to measure the "outward" distance of each sample.

Computing the Radial Gauge

For every sample x_i , we calculate its radial coefficient (gauge function) r_i . This involves projecting the vector $(x_i - c)$ onto all constraint normals. The value r_i is determined by the constraint that is "closest" to the point relative to the center:

$$r_i = \max_j \left(\frac{a_j^T (x_i - c)}{b_j - a_j^T c} \right) \quad (5.5)$$

where $r_i \approx 0$ implies the point is near the center, and $r_i \approx 1$ implies the point is near the boundary.

Statistical Independence

Markov Chain Monte Carlo (MCMC) samples are inherently correlated. Standard statistical tests assume independent data. To satisfy this requirement, we apply a thinning strategy:

- We calculate a thinning factor τ based on the Effective Sample Size (ESS) or use a conservative fixed factor.
- We extract a subset of samples $S_{thin} = \{u_k, u_{k+\tau}, u_{k+2\tau}, \dots\}$.

This ensures the P-value calculation is statistically valid and not artificially inflated by autocorrelation.

Kolmogorov-Smirnov (KS) Test

We compare the empirical distribution of our thinned, transformed samples (u) against the theoretical Uniform(0,1) distribution. We compute the KS statistic D :

$$D = \sup_{x \in [0,1]} |\hat{F}(x) - x| \quad (5.6)$$

where $\hat{F}(x)$ is the empirical CDF. The associated P-value is calculated using the Kolmogorov distribution. A P-value > 0.05 indicates consistency with the uniform distribution.

Volume Shell Analysis

As a secondary diagnostic, we calculate the percentage of points falling into theoretical "volume shells" (e.g., the inner 10% volume, 20% volume, etc.) using the full dataset. We compare the Expected Volume vs. the Observed Volume to visualize systematic biases (e.g., sampler getting stuck in the center vs. the boundary).

5.3 Creating our own demo

The basic logic of what we indent should now be clear to anyone that managed to read any of the previous chapters. We want to keep sampling from the polytope

of choice until we reach some ESS threshold. PolySampler package brings another random walk to the table and we intend to take advantage of that.

Lets begin with a hypercube. In example 3 of the demo.m a hypercube is created like this:

```
P.lb = -ones(d,1);
P.ub = ones(d,1);
```

where the upper and lower bounds of our variables are set to -1 and 1. This is exactly the H-representation of a hypercube. As discussed in the Volesti section, we need to rotate the cube from this perfect position in order to avoid some method over fitting and thus gaining a temporary advantage. Matlab provide a way to randomly rotate any polytope of choice:

```
% Rotation Function
function R = random_rotation(d)
    [Q, ~] = qr(randn(d));
    R = Q;
end
```

Using this function, the hypercube can be easily rotated. We will use a boolean to control whether this rotation is applied. Up to the good stuff:

```
initSampler

% Experiment Parameters
dims = [3];
targetESS = 3000;
times = zeros(size(dims));
USE_ROTATION = true; % Boolean controlling rotation

% Run the experiment for multiple dimensions
for i = 1:length(dims)
    d = dims(i);
    fprintf('Running dimension d = %d...\n', d);

    % First, build an axis-aligned cube as linear inequalities
    % eye is just the unit matrix I(d).
    A = [ eye(d) ; -eye(d) ]; % 2d × d
```

```

b = ones(2*d,1);

% Then choose whether to rotate or not
if USE_ROTATION
    R = random_rotation(d); % orthogonal matrix
    A_rot = A * R';          % rotate constraints: A_rot * y = b
    A_use = A_rot;           % rotated version
else
    A_use = A;               % unrotated axis-aligned cube
end

% Build polytope struct
P = struct;
P.Aineq = A_use;
P.bineq = b;

% Sample until ESS >= targetESS
% Remember that this is exactly how sample struct
% works as explained in the library documentation
o = sample(P, targetESS);
times(i) = o.sampleTime;

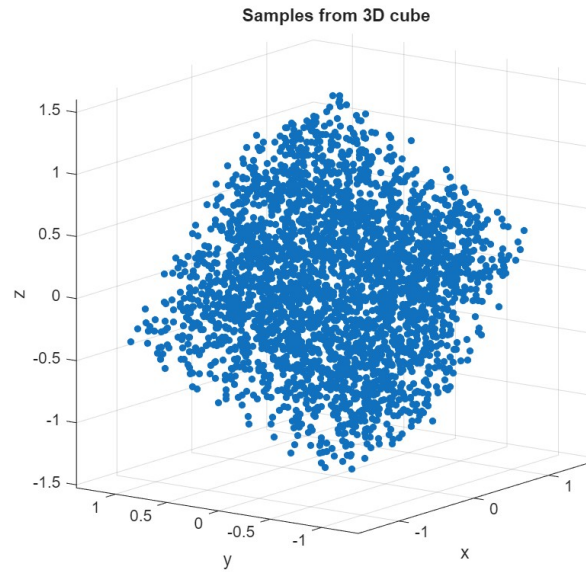
fprintf(' Time: %.4f seconds\n', times(i));

% Diagnostics for this d
% Built-in KS test + autocorrelation diagnostics
figure;
distribution_test(o, struct('toPlot', true));
sgtitle(sprintf('Diagnostic test for dimension d = %d', d));
drawnow();
end

```

We enter the desired parameters. Dims is an array holding all the dimensions the hypercube will have. Then, using the for loop we sample over all hypercubes of every dimension. The cube is created based on dim and rotation and then sample is called to sample from it. Results are retrieved and stored for plotting. The lines of code that plot the results are not presented here. They are trivial and not particularly interesting.

Lets plot the code results for sampling in a 3D rotated cube in order to verify its results. You can see the scatter plot of these samples in [Figure 5.2](#).



Σχήμα 5.2: Scatter plot of samples created from our demo. The shape of a 3D cube is clearly visible.

5.4 Running our demo

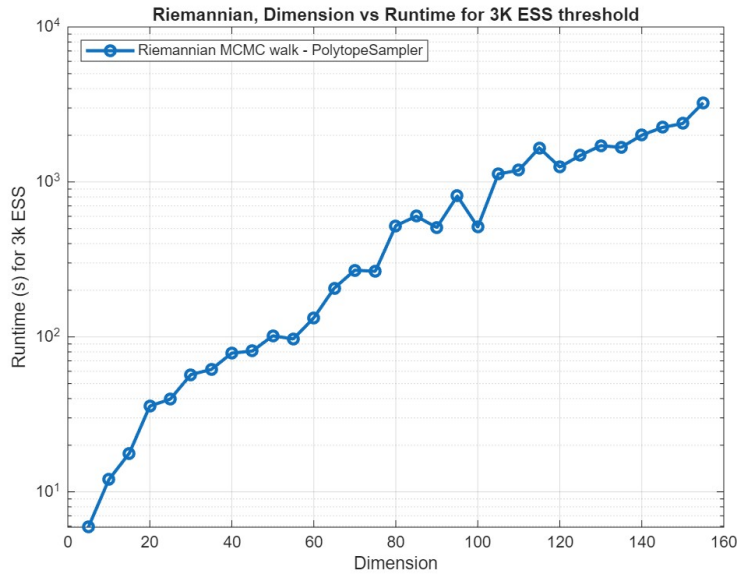
5.4.1 Time trial for d-hypercube using 3k ess threshold

We run the following experiment experiment:

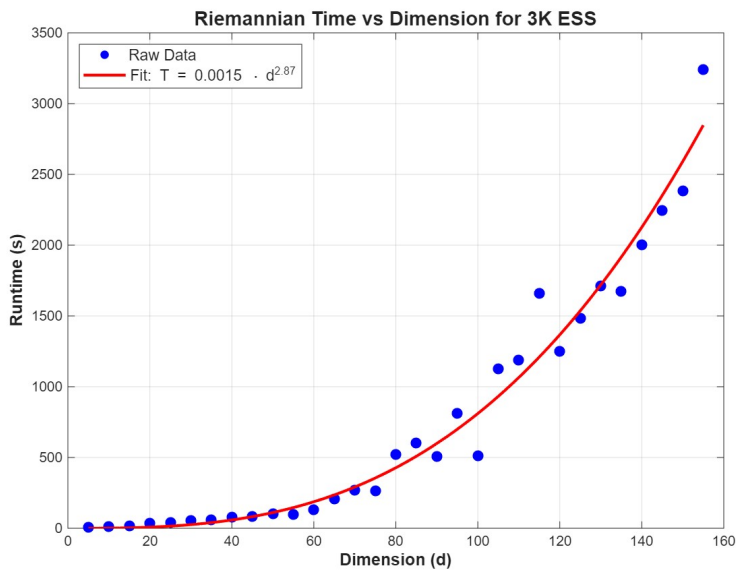
- Polytope: Hypercube (rotated)
- Dimensions: 10 up to 155
- ESS threshold: 3000
- Distribution: Uniform
- Random Walk: Riemannian Hamiltonian Monte Carlo

Results in the followind [Figure 5.3](#).

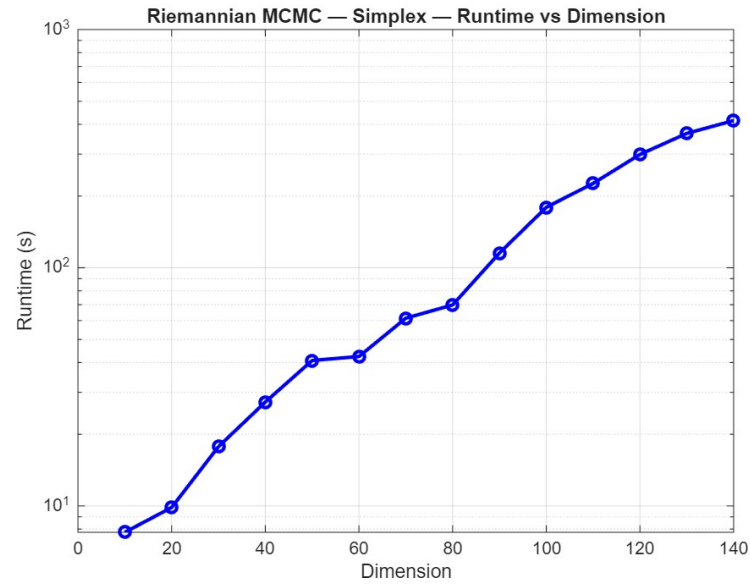
The best fit for these values seems to be a quadratic equation that you can see in the following [Figure 5.4](#).



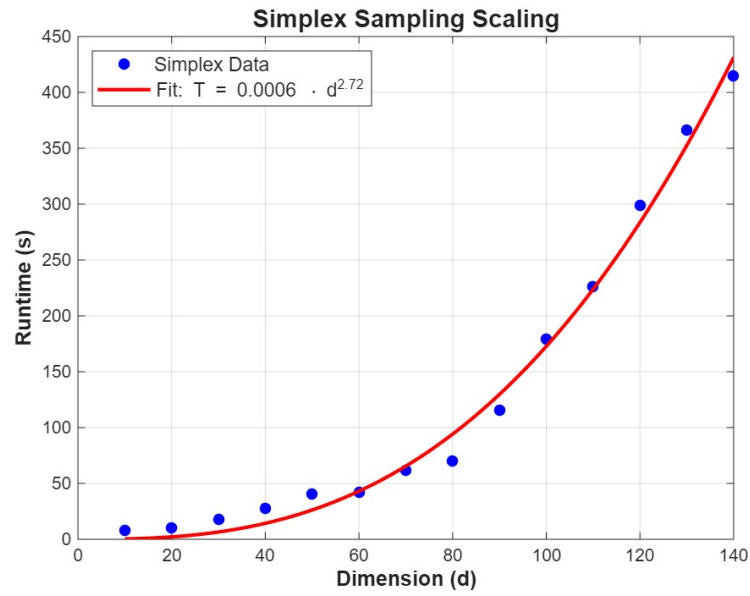
Σχήμα 5.3: Time to generate enough samples to reach 3k ess vs dimension for RHMC method using PolytopeSampler. Polytope used is a hypercube randomly rotated using Matlab's random function. Y axis is log-scale.



Σχήμα 5.4: Power law fit for hypercube results. Y axis is now linear not log-scale. The equation is shown in the legend top left on the graph.



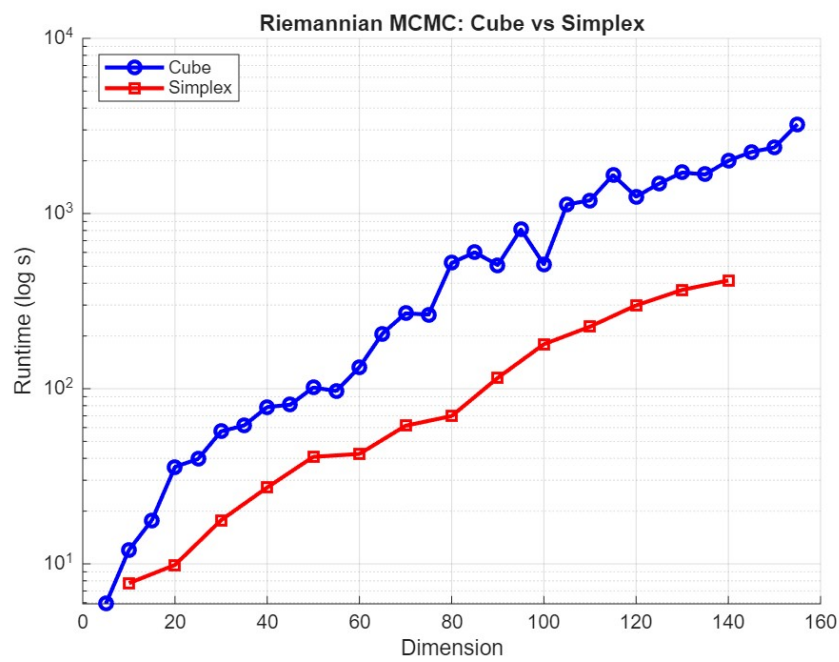
Σχήμα 5.5: Time Trial on Simplex polytope using Riemannian MCMC from PolytopeSamplera for 3K ESS threshold. Y axis is log.



Σχήμα 5.6: Power law fit for the time.

5.4.2 Time Trial for d-Simplex and 3k ESS threshold

5.4.3 Cube vs Simplex

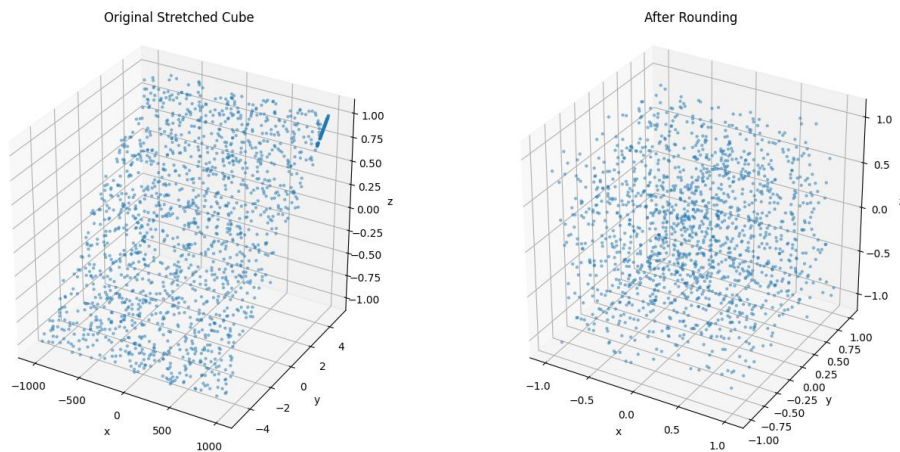


Σχήμα 5.7: Simplex vs Cube for PolytopeSampler Riemannian MCMC and 3k ESS threshold.

Κεφάλαιο 6

Dingo

Dingo is a Python package that analyzes metabolic networks. It relies on high dimensional sampling with Markov Chain Monte Carlo (MCMC) methods and fast optimization methods to analyze the possible states of a metabolic network. To perform MCMC sampling, dingo relies on the C++ library volesti, which provides several algorithms for sampling convex polytopes. dingo also performs two standard methods to analyze the flux space of a metabolic network, namely Flux Balance Analysis and Flux Variability Analysis.



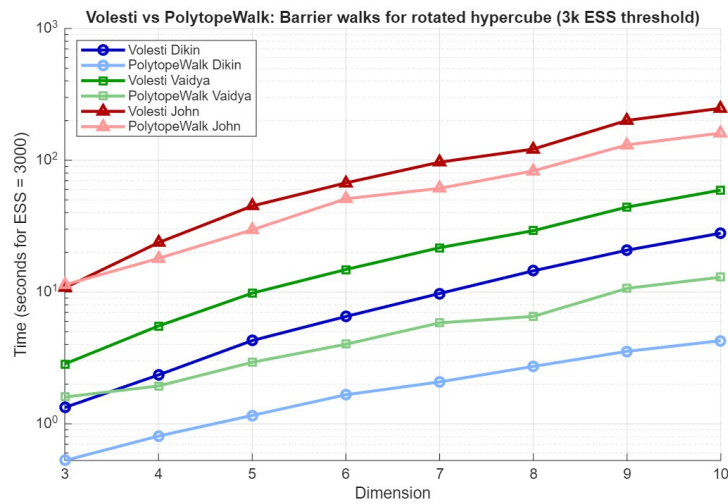
Σχήμα 6.1: Samples in original polytope and in the polytope after rounding with dingo.

Κεφάλαιο 7

PvP

7.1 Barrier Walks

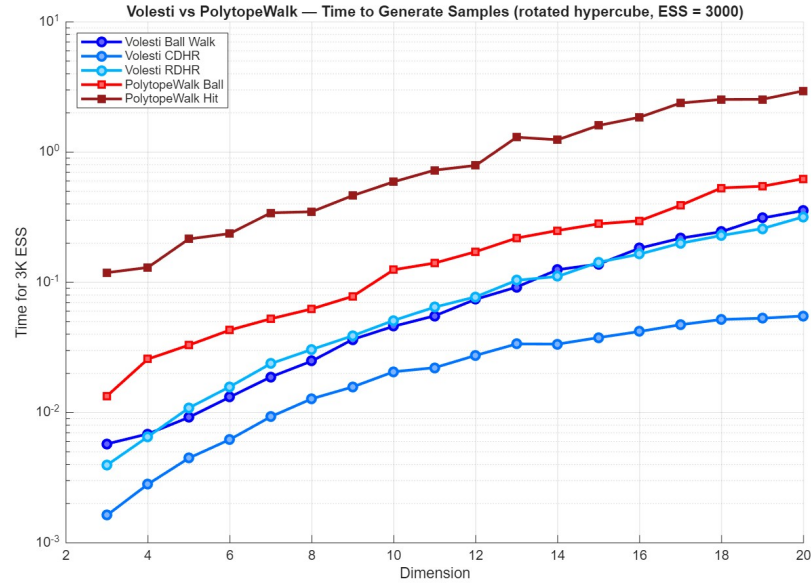
Check graph [Figure 7.1](#).



Σχήμα 7.1: Volesti vs PolytopeWalk runtime for barrier walks. Polytope is a rotated hypercube (53 degrees) for 3 up to 10 dimensions. ESS threshold is 3k as usual. Each walk has a different color and each package has a different shade of this color. It's quite clear that PolytopeWalk has an edge over Volesti.

7.2 Ball Walk, Hit and Run

Check graph [Figure 7.2](#).



Σχήμα 7.2: Volesti vs PolytopeWalk runtime for ball walk and hit and run. Polytope is a rotated hypercube (53 degrees) for 3 up to 20 dimensions. ESS threshold is 3k as usual. Each walk has a different color and each package has a different shade of this color. It's quite clear that Volesti has an edge over PolytopeWalk.

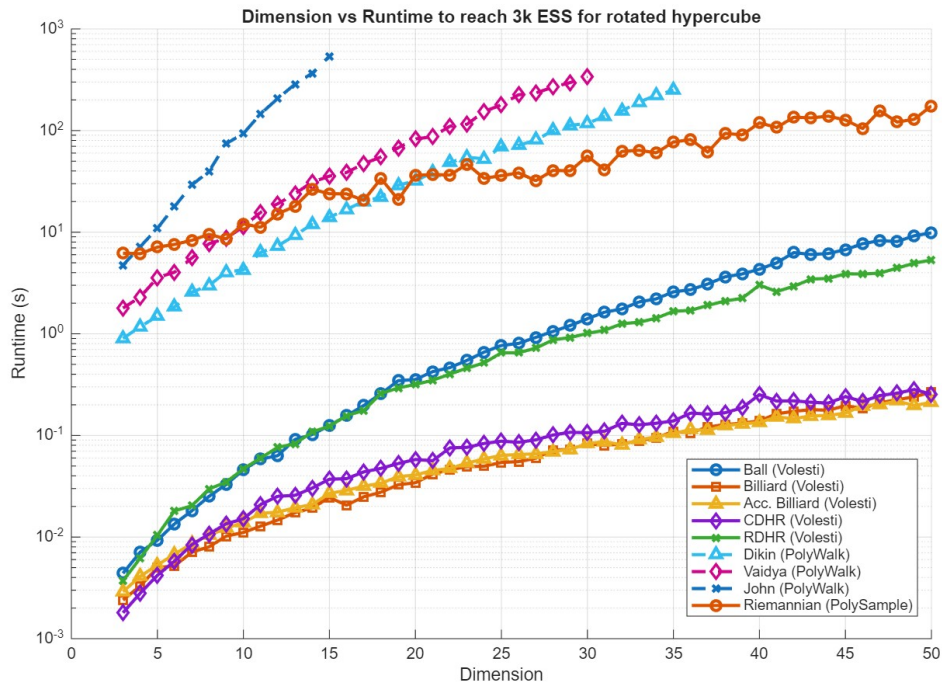
7.3 Free for All

7.3.1 Cube for 3k ESS

Fit power laws. Check graph [Figure 7.3](#).

Algorithm	Exponent	Coeff
Ball Walk (Volesti)	3.7178	0.000005
Billiard (Volesti)	2.2867	0.000032

Acc. Billiard (Volesti)	1.9067	0.000123
Coord HR (Volesti)	1.7139	0.000340
Random HR (Volesti)	3.1915	0.000020
Ball Walk (PolyWalk)	2.7322	0.000187
Hit&Run (PolyWalk)	2.6148	0.001578
Dikin Walk (PolyWalk)	3.8617	0.000259
Vaidya Walk (PolyWalk)	3.5064	0.002242
John Walk (PolyWalk)	4.1712	0.006485
Riemannian (PolySample)	1.9668	0.070703



Σχήμα 7.3: Volesti vs PolytopeWalk vs PolytoSampler for rotated hypercube and 3K ESS threshold.

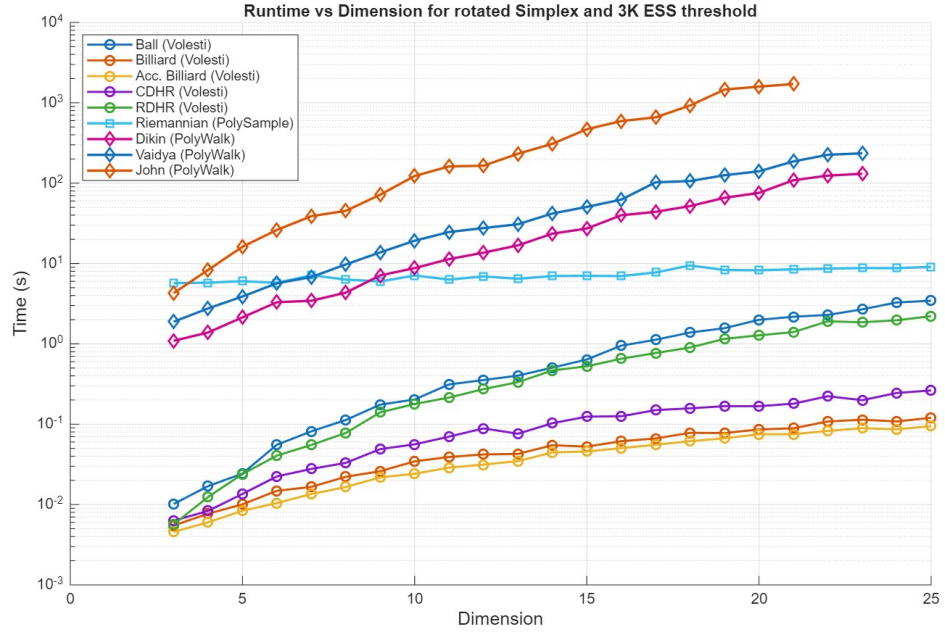
7.3.2 Simplex for 3k ESS

Fit power law. Check graph [Figure 7.4](#).

Algorithm	Exponent b	Coeff a	R ²
-----------	------------	---------	----------------

Ball (Volesti)	3.0675	0.000182	0.993
Billiard (Volesti)	1.4571	0.001104	0.984
Acc Billiard (Vol)	1.5240	0.000735	0.989
Coord HR (Volesti)	1.6718	0.001203	0.974
Random HR (Volesti)	2.8969	0.000212	0.985
Riemannian (PolySampler)	0.3673	2.755148	0.796
Dikin Walk (PolyWalk)	3.2355	0.004883	0.976
Vaidya Walk (PolyWalk)	3.0697	0.014743	0.977
John Walk (PolyWalk)	3.7888	0.016567	0.968

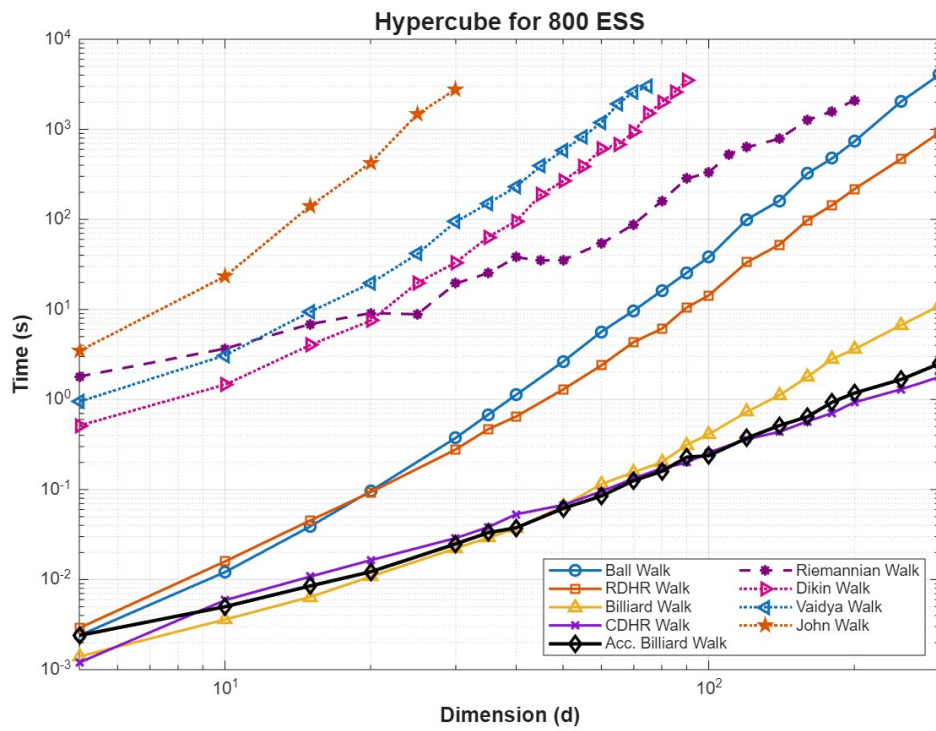
Riemannian exponent increases with dimensions. Check Chapter 5.



Σχήμα 7.4: Volesti vs PolytopeWalk vs PolytoSampler for rotated Simplex and 3K ESS threshold.

7.3.3 Cube for 800 ESS

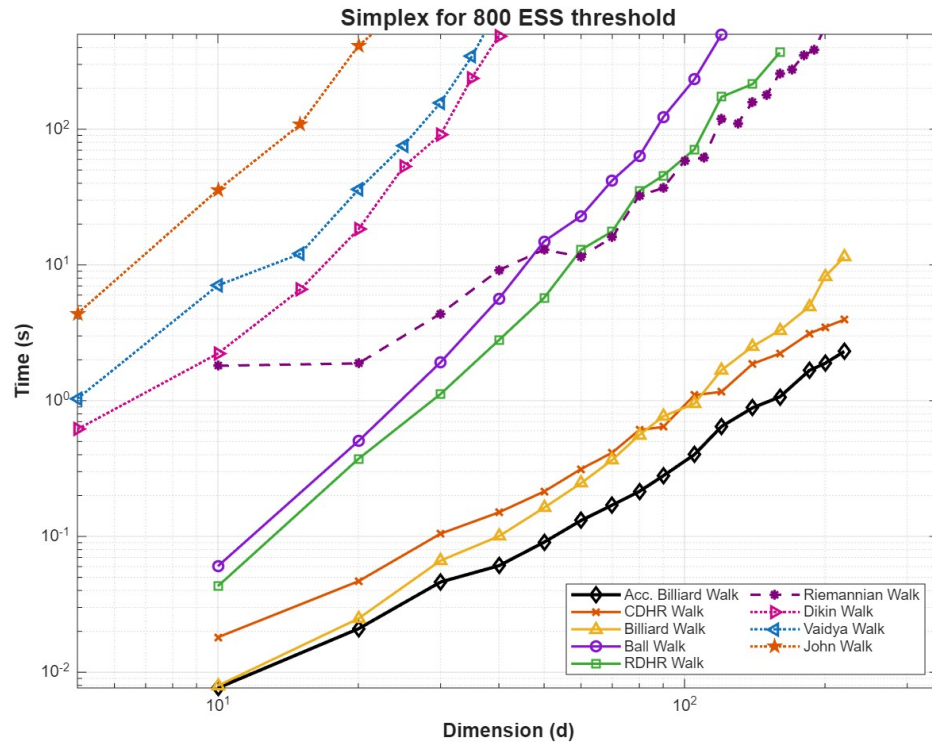
For 800 ESS hypercube check graph [Figure 7.5](#).



Σχήμα 7.5: Volesti vs PolytopeWalk vs PolytoSampler for rotated Cube and 800 ESS threshold.

7.3.4 Simplex for 800 ESS

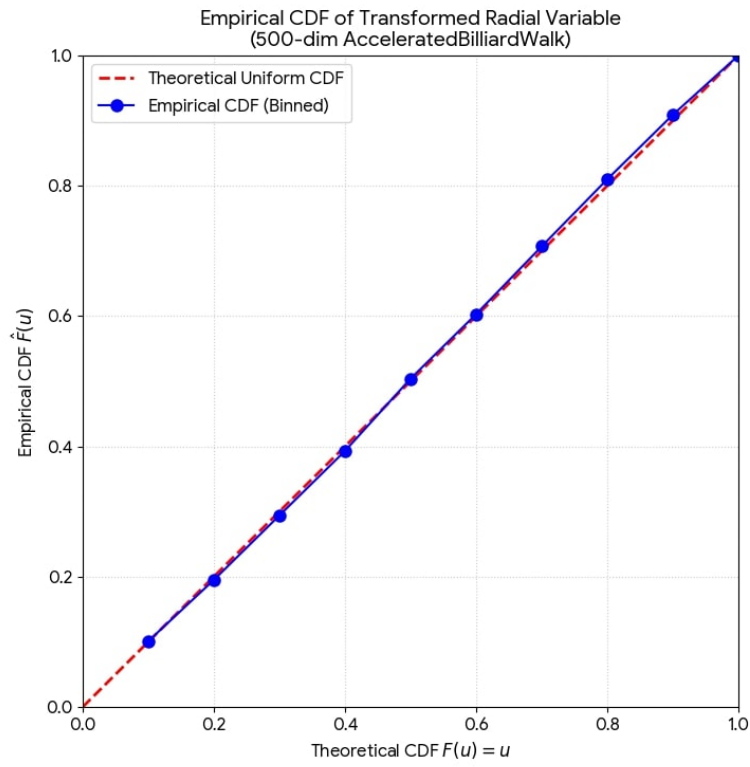
For 800 ESS Simplex check graph [Figure 7.6](#).



Σχήμα 7.6: Volesti vs PolytopeWalk vs PolytoSampler for rotated Simplex and 800 ESS threshold.

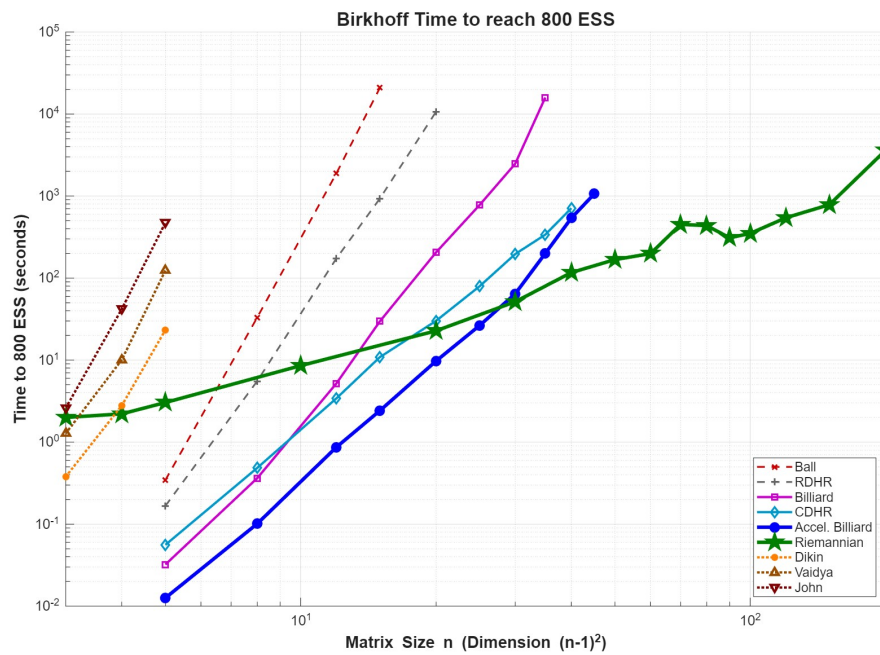
7.3.5 KS test for 500 Hypercube

KS test for 500 dimension rotated hypercube, 3k ESS and Accelerated Billiard Walk. Check [Figure 7.7](#).



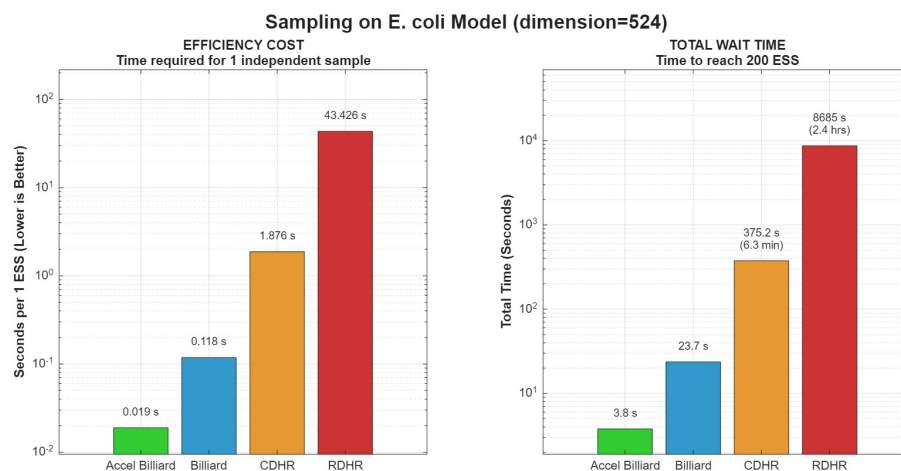
Σχήμα 7.7: Empirical CDF of transformed radial variable. Hypercube, 500 dim, AccBilliard, 3k ESS.

7.3.6 Birkhoff for 800 ESS



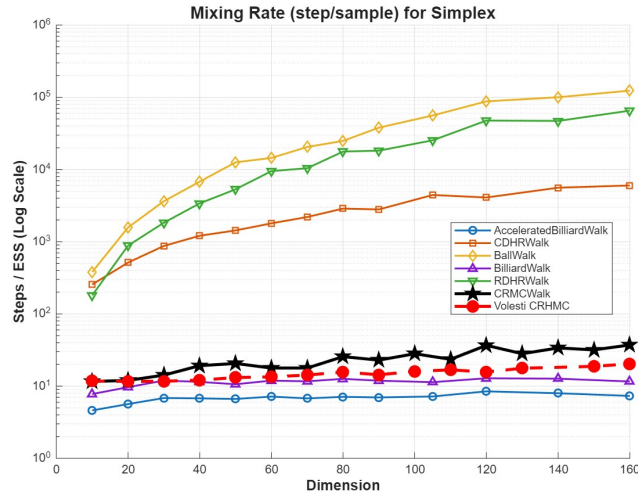
Σχήμα 7.8: Volesti vs PolytopeWalk vs PolytoSampler for Birkhoff and 800 ESS threshold.

7.3.7 E.Coli polytope for 200 ESS

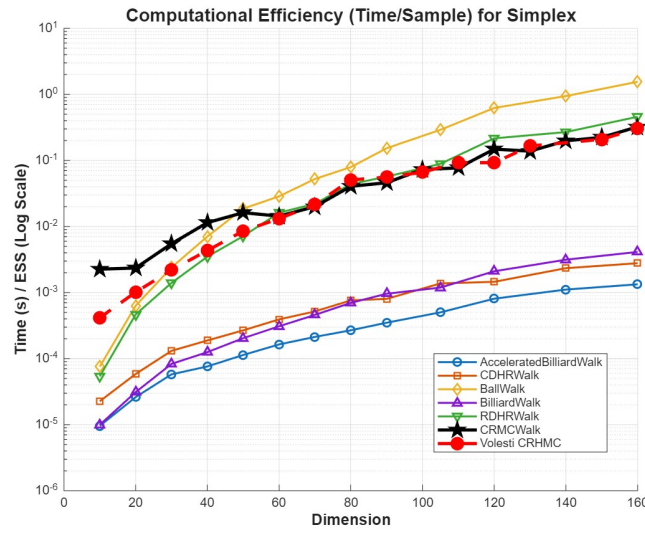


Σχήμα 7.9: The polytope used in this graph comes from biology. It is a mathematical reconstruction of the metabolism of the E. coli bacterium (specifically strain K-12 MG1655). The polytope is rounded but a transformation matrix is provided to reconstruct the original. Using the H-representation, we sampled over it using the methods we talked about so far. Threshold is now at 200 ESS since sampling from such a complex polytope takes a lot of time.

7.3.8 Mixing Time and Time efficiency for Simplex

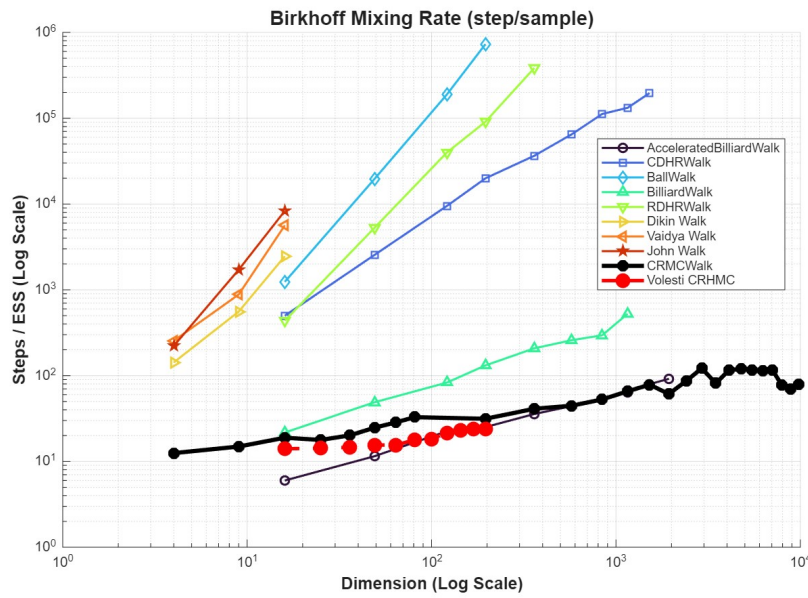


Σχήμα 7.10: We use the results of the 800 ESS simplex experiment to find the mixing time vs dimension. The step/sample ratio is calculated. The number of the steps an algorithm took is the final number of samples returned multiplied by the walk length (since for thinning purposes we ignore some steps). This result is divided by the final ESS to get the ratio. It tells us how many steps we need to move before generating an independent sample inside the polytope.

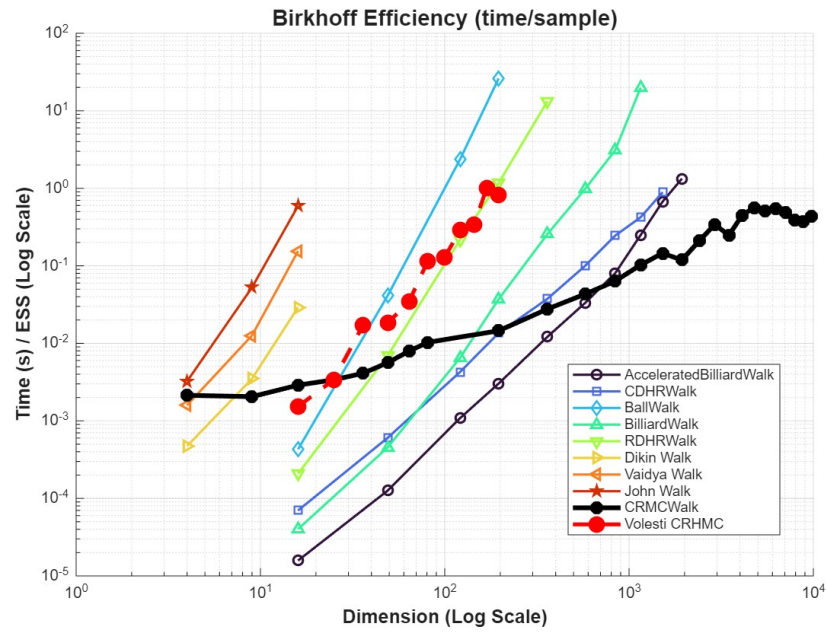


Σχήμα 7.11: This graph is just the time to reach the threshold ESS for the simplex with each method divided by the final ESS. How much time we took to generate one independent sample.

7.3.9 Mixing Time and Time efficiency Birkhoff

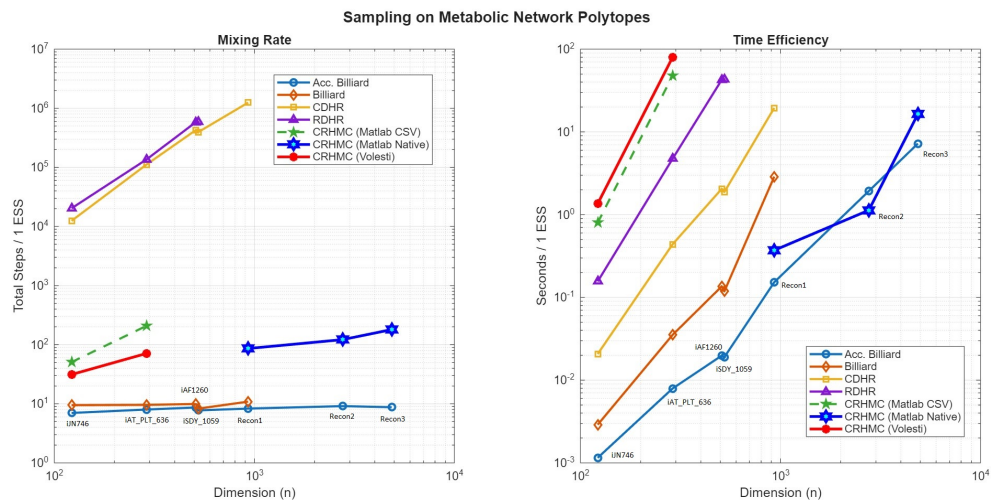


Σχήμα 7.12: Mixing time for Birkhoff.



Σχήμα 7.13: Time per ess for Birkhoff

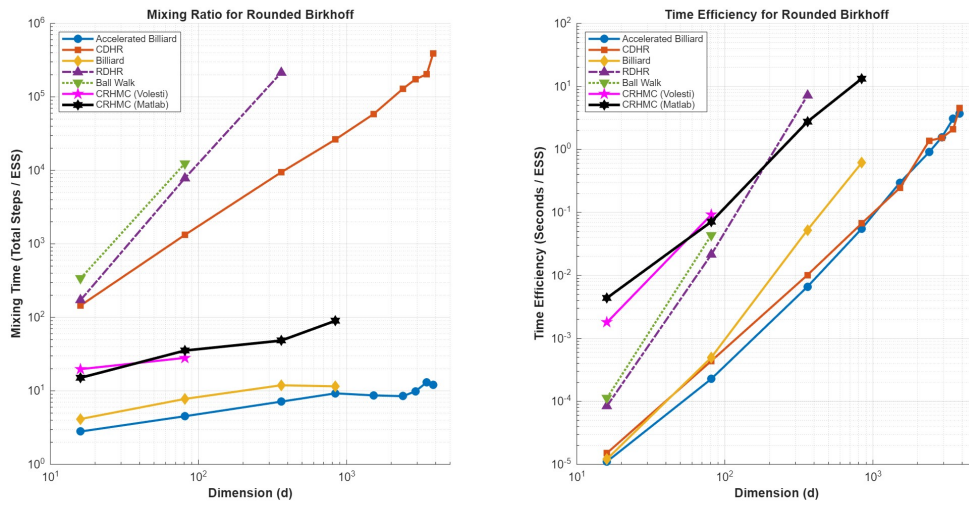
7.3.10 Mixing Time and Time efficiency for Biological Polytopes



Σχήμα 7.14: Mixing time and time efficiency for the biological polytopes.

7.3.11 Mixing Time and Time efficiency for Rounded Birkhoff

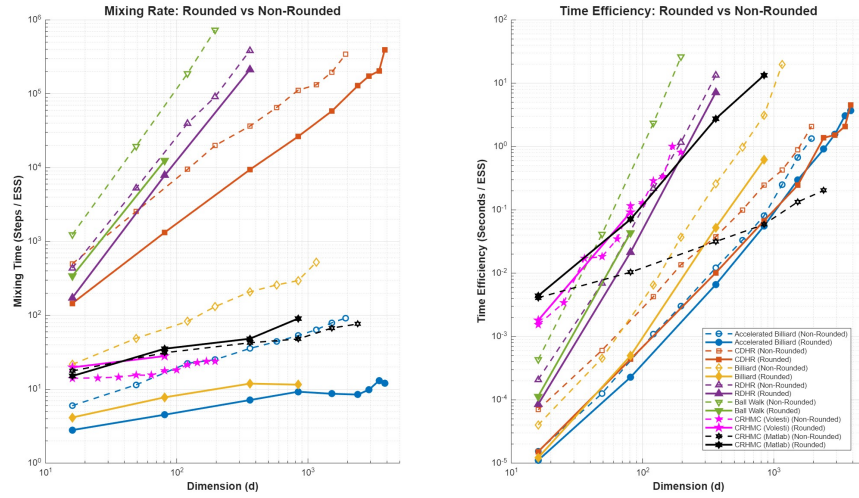
In this section, we have used Dingo to create and round a Birkhoff polytope. The resulting polytope is saved in .csv files and is then passed onto Volesti and PolytopeSampler packages. You can see the results in [Figure 7.15](#).



Σχήμα 7.15: Mixing time and time efficiency for rounded Birkhoff polytope (rounded using Dingo and saved in csv files).

7.3.12 Comparing rounded vs original Birkhoff

We now place the results for the rounded Birkhoff in the same graph as the non-rounded. This way the effects of rounding will become crystal clear. See [Figure 7.16](#).



Σχήμα 7.16: Mixing time and time efficiency for rounded Birkhoff polytope (rounded using Dingo and saved in csv files) vs non rounded natively generated Birkhoff in each package. Continuous lines are for the rounded results while dotted lines are for the non rounded results. Each colour represents a method. As it is clearly visible, all the continuous lines are below dotted lines, showcasing the improvement in performance for the rounded case. The only exception is the CRHMC method of PolytopeSampler package that is dropping in performance. Barrier methods are heavily relying on the sparseness of the matrices to reduce workload. Since rounding often turns a sparse matrix into a dense matrix, this result confirms this.

7.3.13 Empirical CDF για την CDHR

Πίνακας 7.1: Volume Shells Analysis, 500D rotated cube, 3K ESS

Exp Vol %	Obs Vol %	Deviation
10.0%	10.2%	+0.16%
20.0%	20.1%	+0.09%
30.0%	30.1%	+0.14%
40.0%	40.2%	+0.24%
50.0%	50.0%	+0.02%
60.0%	60.1%	+0.10%
70.0%	70.1%	+0.11%
80.0%	80.2%	+0.16%
90.0%	90.0%	+0.02%
100.0%	100.0%	+0.00%

7.3.14 Dingo vs Volesti Sampling time

Πίνακας 7.2: Performance Comparison: 50D rotated Simplex

Method	Samples	Thinning	Time (s)		Min ESS	
			C++	Dingo	C++	Dingo
Acc. Billiard	7,750	1	0.146	0.148	1152	1121
RDHR	76,000	100	10.150	10.350	1209	1141
Ball Walk	132,500	100	18.760	20.010	1226	1134
CDHR	15,000	100	0.278	0.3050	1036	981

Πίνακας 7.3: Performance Comparison: 100D rotated Cube

Method	Samples	Thinning	Time (s)		Min ESS	
			C++	Dingo	C++	Dingo
Acc. Billiard	3,200	1	0.365	0.440	1,199	1,235
CDHR	3,000	200	0.318	0.382	1,040	1,026
RDHR	18,000	200	20.088	20.330	1,104	1,019
Ball Walk	49,500	200	80.986	85.464	1,616	1,604

Πίνακας 7.4: Performance Comparison: 3D rotated cube

Method	Samples	Thinning	Time (s)		Min ESS	
			C++	Dingo	C++	Dingo
Acc. Billiard	200	1	0.0002	0.0017	130	120
CDHR	200	6	0.0001	0.0013	113	113
RDHR	150	6	0.0002	0.0015	116	111
Ball Walk	350	6	0.0003	0.0015	106	113

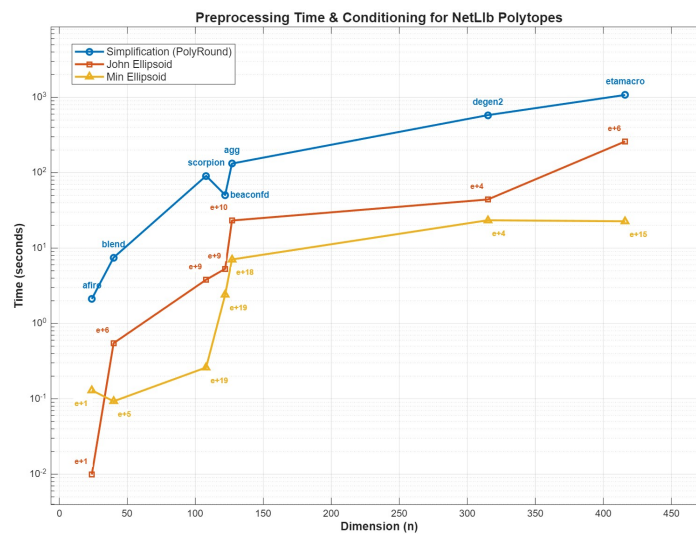
Results are mostly the same but notice that for 3D cube where sampling is negligible, Dingo is one order of magnitude slower, a price we have to pay for the Dingo wrapper.

7.3.15 NetLib Polytopes

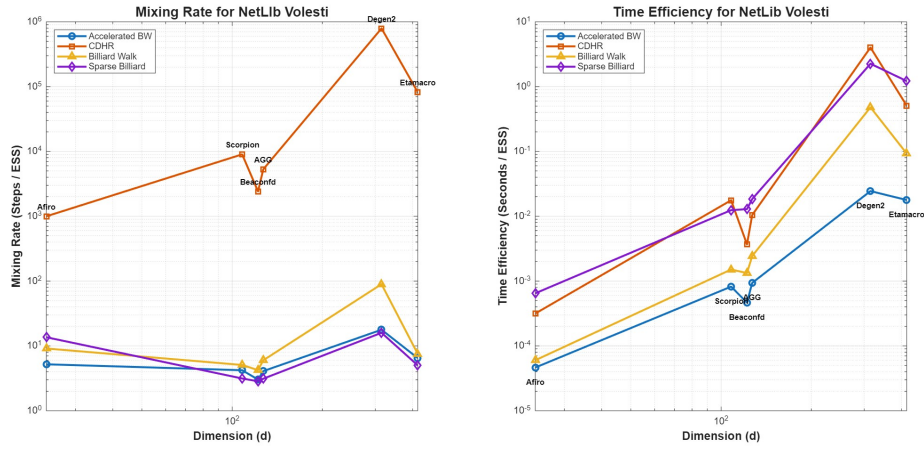
NetLib polytopes were extracted from their .mat format inside the PolytopeSampler package using the Null space technique to convert the equalities plus bounds to inequalities. These polytopes though are way too edgy to sample with using the

random walks of Volesti. To that end, first they are simplified using a package called Polyround found here [PolyRound GitLab](#). This will help the polytopes become a bit more wide, so that their volume at any given dimension is not close to zero as that will confuse the rounding process. Then the polytopes are passed through Dingo to be rounded using John ellipsoid method. This will stretch them in order to make them behave more smoothly in all dimensions. Finally, these polytopes are fed to Volesti random walks to be sampled from.

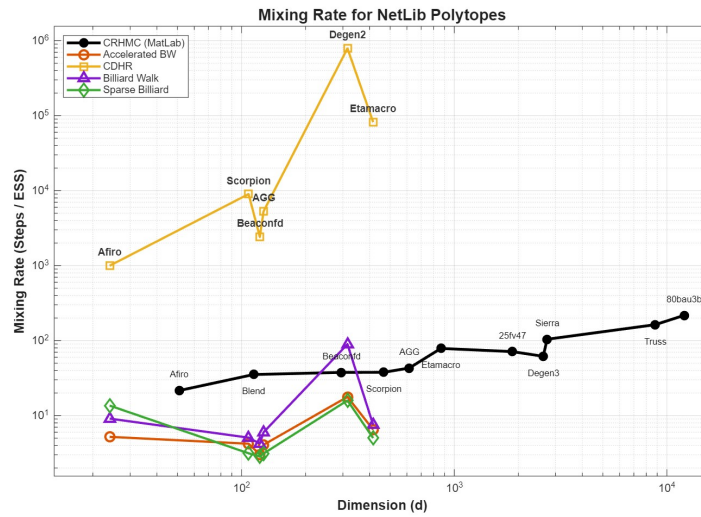
Notes: Only some of the simpler NetLib polytopes have been sampled, those that needed less than 30min to be simplified. In addition, blend polytope even though it was simplified in less than 30min, even after rounding Volesti methods got stuck near the center or in some corner. ESS couldn't grow at all for any method. Maybe round it a second time?



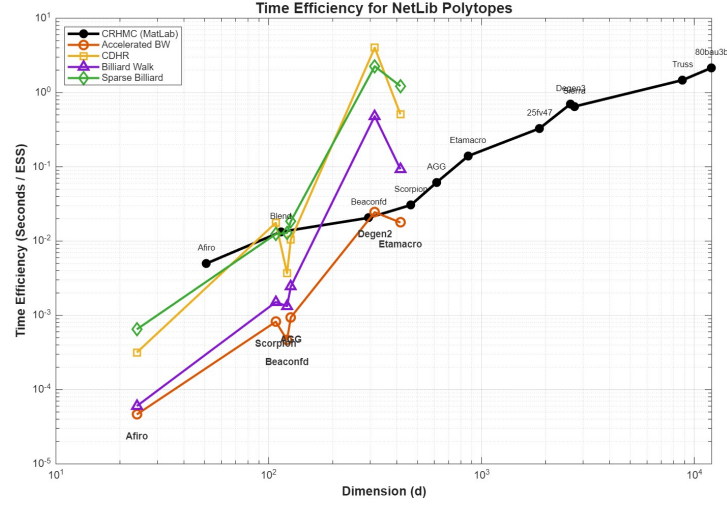
Σχήμα 7.17: Time it took to simplify and round each of the NetLib polytopes. Blue line is the simplify time and the rest represent the rounding times for the John ellipsoid and min ellipsoid methods. No other method was recognised by Dingo. The number you see above each rounding point represents the ratio of the inner ball vs the outer ball.



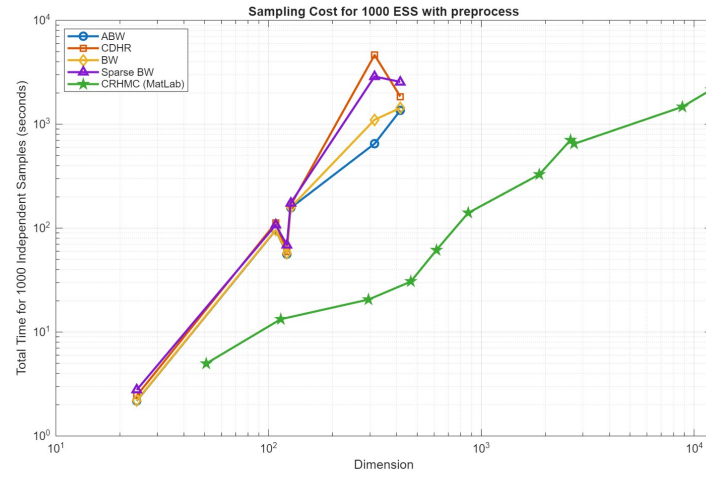
Σχήμα 7.18: Mixing rate and time efficiency for the simplified rounded NetLib polytopes sampled with Volesti methods.



Σχήμα 7.19: Mixing rate comparison between all methods. MatLab polytopes have higher dimension because they are still in their original .mat form with equalities and boundaries.



Σχήμα 7.20: Time efficiency comparison between all methods. Preprocessing time for the Volesti methods is not included.

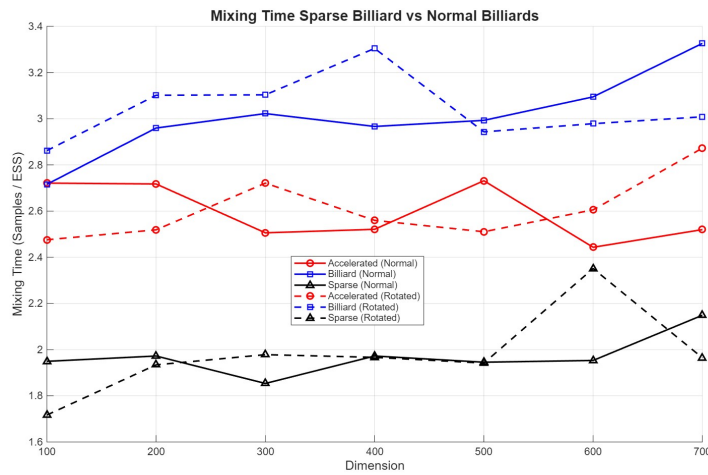


Σχήμα 7.21: Time required to reach 1000 ESS based on the time efficiency of the previous graph. Preprocess time is now included for the Volesti methods (simplify time and john rounding time).

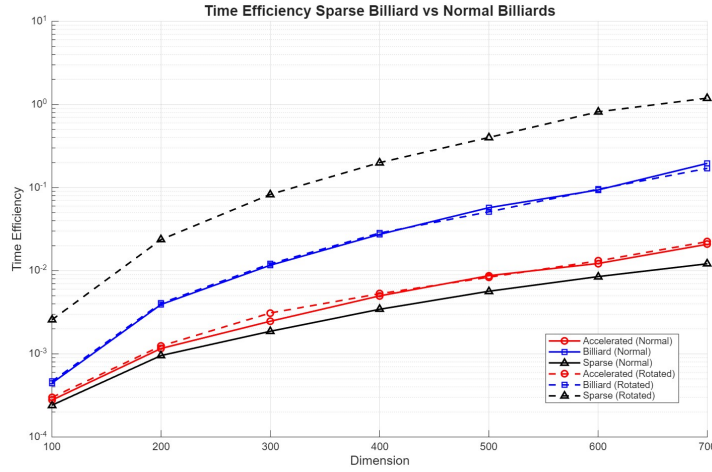
7.3.16 Sparse Billiard vs Normal Billiards

In this section we test a brand new Volesti method, the sparse Billiard walk. The method takes advantage of sparse matrices to reduce calculations load and improve

the time efficiency of the billiard walks. In order to test the method against the regular billiard walks, the most obvious sparse polytope matrix was used, that of a cube. In the first experiment, the cube was used in its infant axis aligned form where most entries of the matrix are 0. In the second experiment, the cube was rotated by some angle (53 deg), thus turning the matrix is a dense one. In theory, the sparse billiard efficiency is expected to drop significantly in the second experiment. See the results in [Figure 7.22](#) and [Figure 7.23](#).



Σχήμα 7.22: Mixing Time for sparse billiard vs normal billiard walks. Continuous lines are for the axis aligned cube (sparse matrix) while the dotted lines are for the rotated cube (dense matrix).

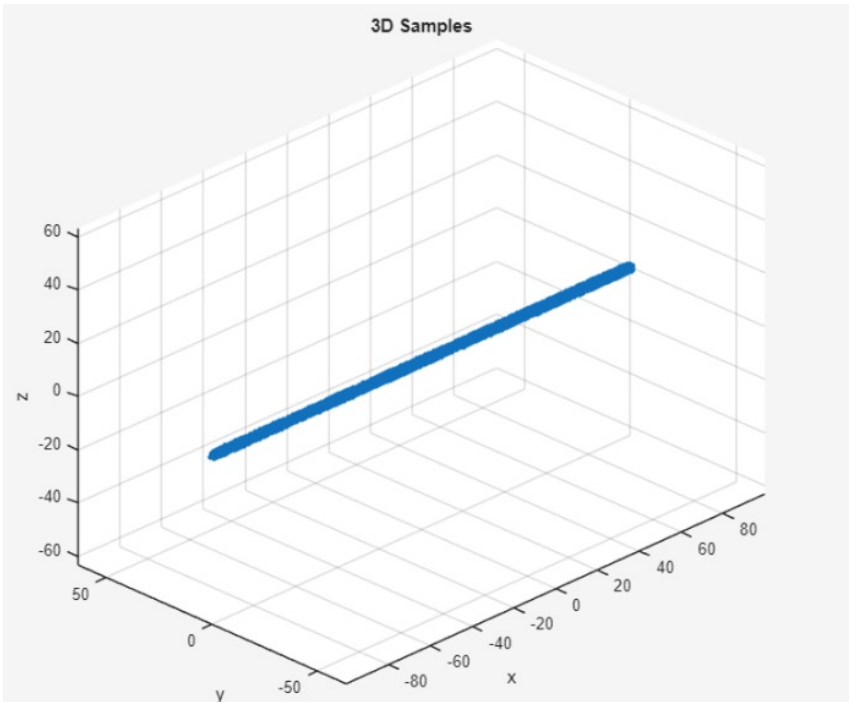


Σχήμα 7.23: Time efficiency for sparse billiard vs normal billiard walks. Continuous lines are for the axis aligned cube (sparse matrix) while the dotted lines are for the rotated cube (dense matrix).

7.3.17 Breaking the KS test

For a better understanding of the KS test read the theory. In this section we will try to create a case where the test will save us from a faulty sampling over some polytope. By faulty we mean that despite the ESS being quite good and increasing with every sample, the samples generated are not uniformly distributed in the polytopes volume. The KS test is having difficulties ensuring uniform sampling over wide and short polytopes in a needle like shape (ESS will save the sampling in these cases), but for convex rounded polytopes it is quite useful.

We tried a variety of polytopes in all shapes and dimensions. One of the main focuses was the skinny cube provided by Volesti generators. Check [Figure 7.24](#) for the samples results.



Σχήμα 7.24: Ball walk sampling in a 3D skinny cube (normal cube with one dimension extended). 50k samples taken.

While we see the samples spread uniformly all over the skinny cube, nevertheless the uniform walk results couldn't confirm it:

[BallWalk] Samples: 50000 | ESS: 15 | Time: 0.0335417s

[BallWalk] DONE. Final ESS: 15

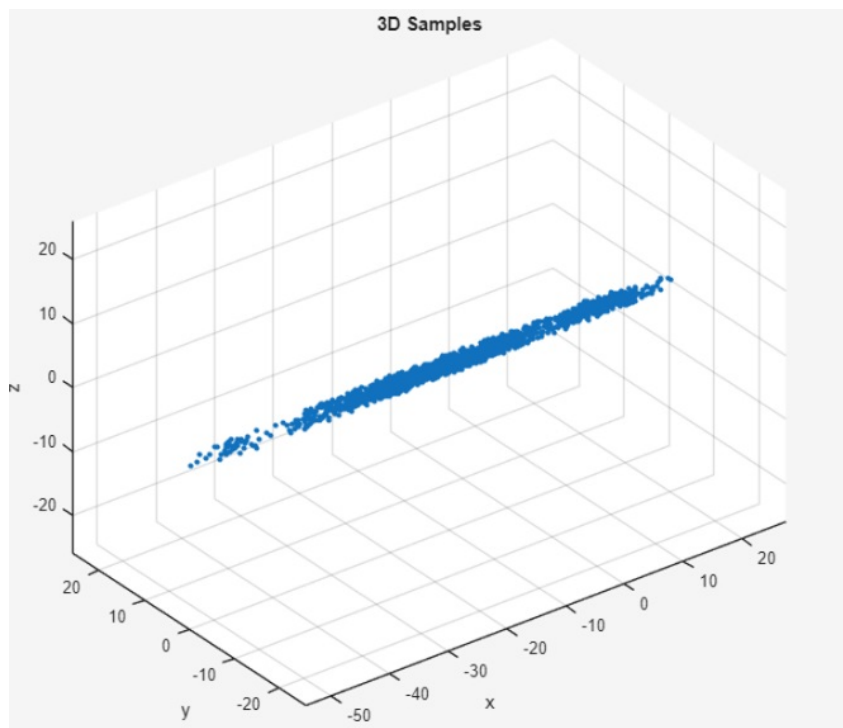
Volume Shells Analysis (Expected vs Observed):

Exp Vol% | Obs Vol% | Deviation

10.0%		9.3%		-0.73%
20.0%		19.2%		-0.82%
30.0%		29.8%		-0.21%
40.0%		40.1%		+0.06%
50.0%		50.3%		+0.26%
60.0%		60.3%		+0.33%
70.0%		70.3%		+0.35%
80.0%		80.4%		+0.36%
90.0%		90.4%		+0.45%
100.0%		100.0%		+0.00%

You can see that the Volume Shell analysis does confirm the uniformity of the samples but the ESS couldnt confirm this. Thus, the ESS metric failed while the KS test worked. And yes we did use the ball walk as it is the least efficient walk.

In the second experiment, we deliberately asked for fewer samples so that we would not cover the polytope completely. Check [Figure 7.25](#).



Σχήμα 7.25: Ball walk sampling in a 3D skinny cube (normal cube with one dimension extended). Only 2k samples taken.

The sampling metrics are these:

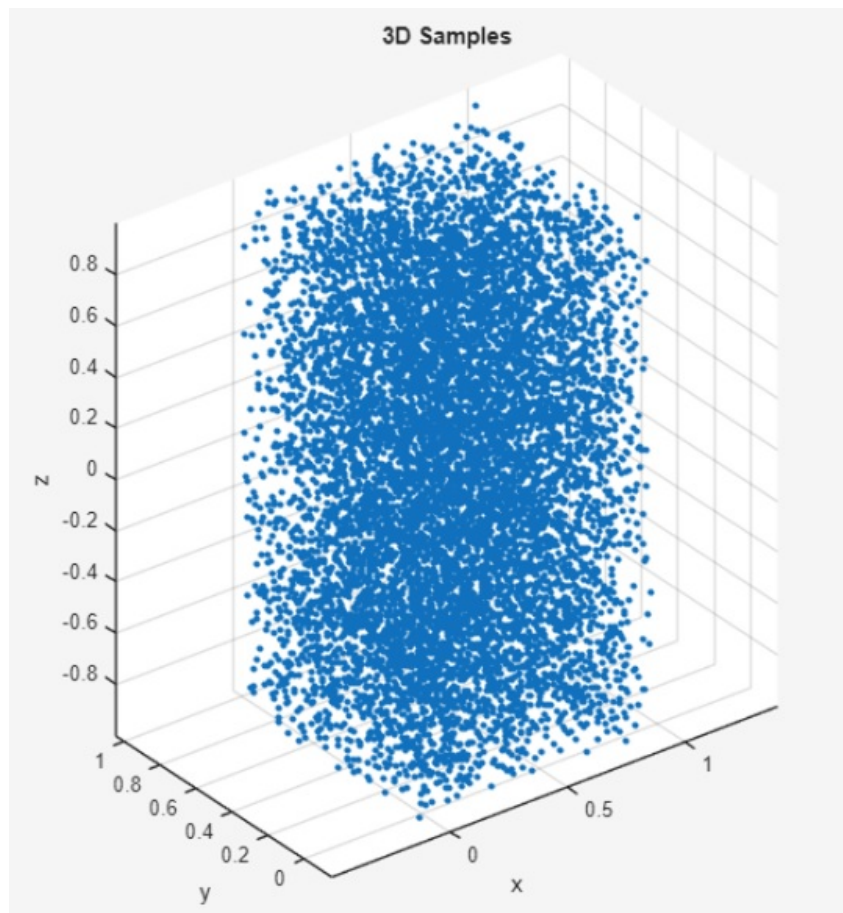
```
[BallWalk] Samples: 2000 | ESS: 11 | Time: 0.00201971s
[BallWalk] DONE. Final ESS: 11
Volume Shells Analysis (Expected vs Observed):
Exp Vol% | Obs Vol% | Deviation
```

```
-----
10.0% | 20.5% | +10.50%
20.0% | 32.6% | +12.60%
30.0% | 44.1% | +14.15%
40.0% | 53.7% | +13.70%
```

50.0%		63.7%		+13.70%
60.0%		72.7%		+12.65%
70.0%		80.5%		+10.55%
80.0%		87.4%		+7.35%
90.0%		94.0%		+4.00%
100.0%		100.0%		+0.00%

ESS is still broken in this case, but now we see the KS test start to break as well. Checking the radial values of the samples, we see that in the left side of the polytope, fewer samples exist and thus more samples are located in smaller shells. Anything more than 1 to 2 percent in this shell analysis is beginning to look weird.

Still, the question about how to have a good ESS value but break the shells remain. To that end we execute a final experiment. We will cheat in order to achieve our goal. We now sample from a normal cube but we only keep samples located in the upper quarter of the cube. This way, the samples will be well spread in the upper quarter and the ESS will be good, while the KS test will fail since the border are not equally distributed around the samples. Check [Figure 7.26](#).



Σχήμα 7.26: Ball walk samples in a normal cube but only those in the upper quarter of the cube.

The sampling results are these:

[BallWalk] Samples: 15450 | ESS: 5985 | Time: 0.010669s

[BallWalk] DONE. Final ESS: 5985

Volume Shells Analysis (Expected vs Observed):

Exp Vol% | Obs Vol% | Deviation

10.0%		25.8%		+15.83%
20.0%		39.0%		+18.96%
30.0%		49.9%		+19.89%
40.0%		60.2%		+20.18%
50.0%		69.3%		+19.33%
60.0%		78.1%		+18.12%

70.0%		86.8%		+16.77%
80.0%		92.9%		+12.94%
90.0%		96.8%		+6.78%
100.0%		100.0%		+0.00%

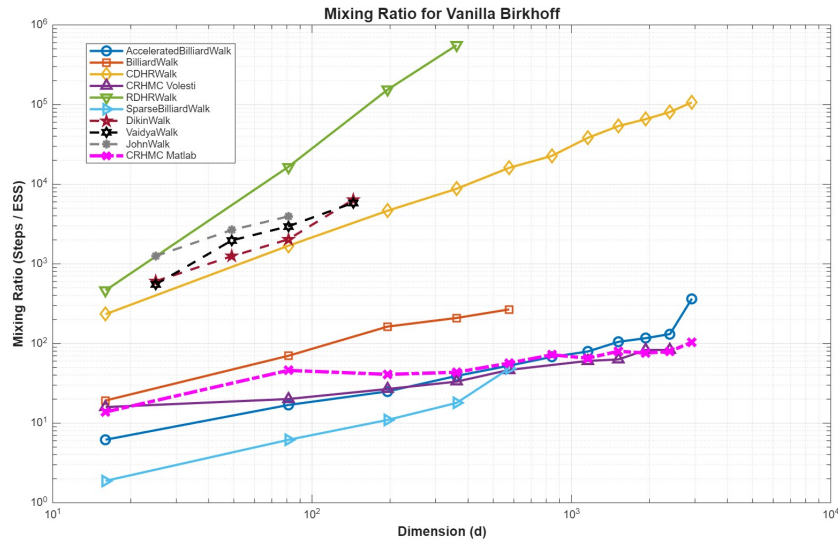
The 15450 samples gave an awesome 6k ESS score, 1 in 3 samples in independent. Yet, the KS test saves us from an obvious failure since the samples are located in the upper quarter of the cube. The test can detect this deficiency. Thus, we found a case where ESS is good but the KS test fails.

7.3.18 Vanilla Birkhoff benchmark

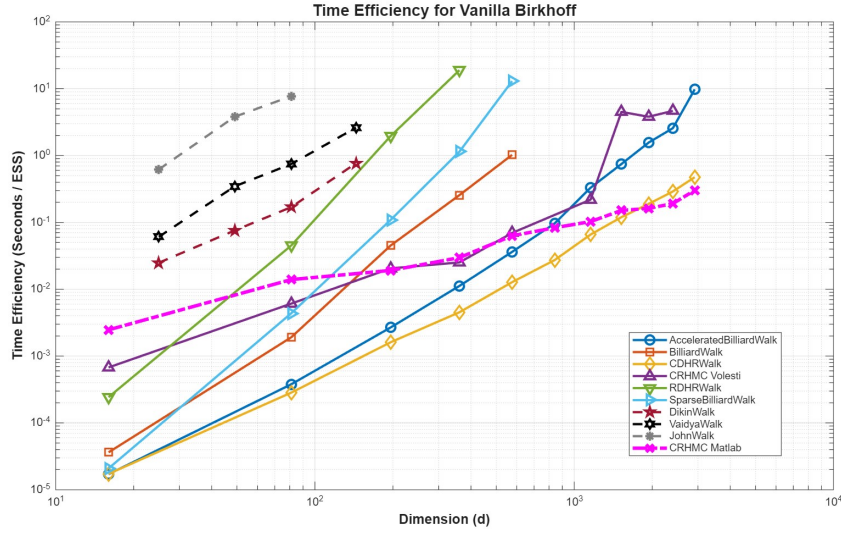
First of all, by the term vanilla we imply that the Birkhoff we sampled from is native to each package and method, we didn't change anything regarding rotation or alignment. The main purpose for this design choice was to test whether the sparseness of the Birkhoff definition in the $A \times x \leq b$ form has any effect for some of the methods used compared to the dense matrices we used before.

Some of the packages generated the Birkhoff in the equalities form plus some constraints since that is the input they expect for the sampling methods to work. This discrepancy in the definition of the Birkhoff is not expected to alter the main logic of the experiment in any significant way since the in both cases the polytope is the same. In a different read of the situation, each packages performance will be compared to the rest of them and their choice of definition will be evaluated.

The results of the benchmark can be seen in the following graphs [Figure 7.27](#) and [Figure 7.28](#).



Σχήμα 7.27: Mixing ratio for the vanilla Birkhoff (see section intro for more details). We can see some interesting results. First of all, the barrier walks from PolytopeWalk are lagging way behind even though the sparse version of the sampling is quite more efficient than the dense one. Secondly, the sparse billiard, accelerated billiard and the two CRHMC methods are all competing for the first place. Finally, the hit and run methods apply many steps in order to achieve independent samples.



Σχήμα 7.28: Time efficiency for the vanilla Birkhoff (see section intro for more details). Firstly, the sparse billiard walk is quite disappointing in this benchmark, achieving the same time efficiency as the normal billiard walk. On another note, CDHR method turns out to be the king of the sparse vanilla sampling. The nice aligned vanilla Birkhoff entertains the method, same with the sparse cube we saw earlier, while our rotated polytopes gave the method troubles. Additionally, the barrier methods looks like they will overthrow the geometric methods in huge dimensions, as is expected by the theory. Finally, the Volesti CRHMC method challenges the Matlab method up to a point, but is then slowed down before it continues its normal trend. Eventually the CRHMC methods will become the most efficient ones in the long run.

7.4 Biological Polytopes Preprocessing

Genome-scale metabolic models (GEMs) were obtained from the BiGG Models database in SBML (.xml) format. The steady-state solution space of each metabolic network is inherently defined as a convex polytope:

$$\mathcal{P} = \{x \in \mathbb{R}^n \mid Sx = 0, lb \leq x \leq ub\} \quad (7.1)$$

where S is the sparse stoichiometric matrix, x represents the flux vector, and lb, ub are the lower and upper thermodynamic flux bounds, respectively.

To prepare these highly ill-conditioned spaces for uniform sampling, we utilized the PolyRound Python package. First, the models were simplified to remove redundant

constraints and fixed variables, drastically reducing the dimensionality of the constraint system. Next, the equality constraints ($Sx = 0$) were eliminated by projecting the polytope into a full-dimensional basis within the null space of S . This transformation yields a strict inequality system:

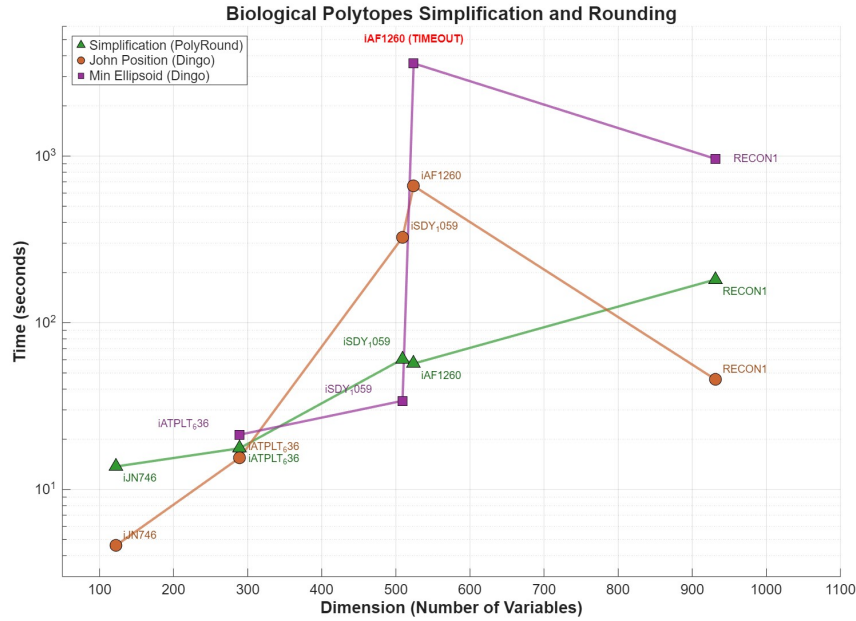
$$\mathcal{P}_{trans} = \{y \in \mathbb{R}^k \mid Ay \leq b\} \quad (7.2)$$

where $k < n$.

Finally, because biological polytopes frequently exhibit extreme aspect ratios due to disparate flux bounds (forming “needle-like” geometries), geometric rounding was required. We applied rounding algorithms from Dingo to compute an affine transformation matrix, T . Applying this transformation scales the polytope into a well-conditioned, isotropic shape:

$$\mathcal{P}_{round} = \{z \in \mathbb{R}^k \mid (AT)z \leq b\} \quad (7.3)$$

This rounding step mitigates the numerical challenges of floating-point arithmetic and is strictly necessary to ensure the convergence and mixing efficiency of subsequent Markov Chain Monte Carlo (MCMC) flux sampling algorithms.



Σχήμα 7.29: Simplify and rounding for the biological polytopes. We downloaded as .xml from BIGG, fed to Polyround to simplify, then fed to Dingo to transform and round.

Βιβλιογραφία

- [1] Adam M. Gustafson. Topics in statistics and convex geometry: Rounding, sampling and interpolation. Technical report, University of Washington, 2018. A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.
- [2] F. J. Massey Jr. The Kolmogorov–Smirnov test for goodness of fit. *Journal of the American Statistical Association*, 46(253):68–78, 1951.