VILNIAUS UNIVERSITETAS

MATEMATIKOS IR INFORMATIKOS FAKULTETAS

KOMPIUTERINIO MODELIAVIMO STUDIJŲ PROGRAMA

Magistro baigiamasis darbas

Atsitiktinių skaičių generavimo metodų įtaka optimizavimo algoritmų efektyvumui

The impact of random number generation methods on the efficiency of optimization algorithms

Martynas Kudlis

Darbo vadovas : dr. Algirdas Lančinskas

Vilnius
2025

Santrauka

Šiame straipsnyje nagrinėjama atsitiktinių skaičių generatorių įtaka optimizavimo algoritmų efektyvumui. Analizuojama, kaip skirtingi skaičių generatoriai veikia algoritmo našumą, pabrėžiamas poreikis suprasti jų savybes ir kaip jos veikia optimizavimo rezultatus. Tyrime taip pat aptariamas Kolmogorov-Smirnov testo taikymas atsitiktinių skaičių generatorių kokybei įvertinti ir pateikiama įvairių testinių funkcijų analizė, skirta algoritmo patikimumui įvertinti. Palyginami skirtingi atsitiktinių skaičių generatoriai ir jų tikslumas generuojant reikšmes, pateikiant įžvalgų apie jų tinkamumą praktikoje ir tolimesniems tyrimams.

Summary

This paper explores the influence of random number generation methods on the efficiency of optimization algorithms. It delves into the impact of different generators on algorithm performance, emphasizing the need to understand their properties and how they affect optimization results. The research also discusses the application of the Kolmogorov-Smirnov test to evaluate the quality of random number generators and presents an analysis of various test functions to assess algorithm reliability. Additionally, it compares different random number generators and their precision in generating values, providing insights into their suitability for practical applications and future research.

Iliustracijų sąrašas

Lentelių sąrašas

Turinys

Žymėjimai

- KS - Kolmogorov-Smirnov
- LCG - tiesinis kongruentinis generatorius (angl. *Linear Congruential Generator)*
- PCG - permutacinis kongruentinis genetarorius (angl. *Permuted Congruential Generator)*
- MT - Mersenne Twister
- PRS - paprastoji atsitiktinė paieška (angl. *Pure random search*)
- PSO - dalelių spiečiaus optimizavimas (angl. *Particle swarm optimization*)
- GA - genetinis algoritmas (angl. *Genetic algorithm*)
- SA - modeliuojamas atkaitinimas (angl. *Simulated annealing*)
- RNG - atsitiktinių skaičių generatorius (angl. *Random number generator*)
- SMA - gleivūnų algoritmas (angl. *Slime mold algorithm*)
- AOA - aritmetinis optimizavimo algoritmas (angl. *Arithmetic Optimization Algorithm*)

Įvadas

Optimizavimo problemų gausu įvairiose srityse, tačiau dažnai jos būna sudėtingos, kad negalima tiksliai įvertinti daugelio įtakos turinčių kintamųjų, o kartais jų vertės netgi nežinomos. Tokiu atveju atsitiktinumas tampa lemiamu veiksniu sprendžiant problemas, ir atsitiktinio skaičiavimo metodai gali leisti išsamiau tyrinėti galimus sprendimus. Šis magistrantūros darbas sutelktas į tai, kaip atsitiktinių skaičių generavimo metodai veikia optimizavimo algoritmų efektyvumą.

Optimizavimo algoritmų efektyvumas yra glaudžiai susijęs su metodais, naudojamais atsitiktinių skaičių generavimui. Atsitiktinių skaičių generatoriai yra esminiai komponentai įvairiuose skaitiniuose procesuose, generuojantys skaičių ar simbolių sekas be jokio pastebimo modelio. Tuo tarpu optimizavimo algoritmai veikia kaip skaitiniai įrankiai, skirti rasti geriausius sprendimus apibrėžtoms problemoms, atsižvelgiant į suteiktus apribojimus.

Suprasti, kaip skirtingi atsitiktinių skaičių generavimo metodai veikia optimizavimo algoritmus, yra labai svarbu dėl praktinių pasekmių realaus pasaulio scenarijose. Pavyzdžiui, finansiniame modeliavime rizikos vertinimo ir investicinių strategijų tikslumas priklauso nuo kokybiškų atsitiktinių skaičių, naudojamų simuliuoti rinkos elgesį. Taip pat moksliniuose modeliavimuose, tokiose kaip klimato modeliavimas ar molekulių dinamika, atsitiktinių skaičių neprognozuojamumas yra būtinas, siekiant gauti realistiškus ir patikimus rezultatus. Todėl analizuojamų realių problemų, paremtų atsitiktiniais generatoriais, svarbą pabrėžia poreikis suprasti, kaip skirtingi generavimo metodai veikia optimizavimo rezultatus.

Su technologijų pažanga optimizavimo algoritmų ir atsitiktinių skaičių generatorių pokyčiai tęsiasi. Naujausių metodų, tokių kaip: genetiniai algoritmai (angl. *Genetic algorithm, GA*), dalelių spiečių optimizavimas (angl. *Particle swarm optimization, PSO*) ir sudėtingesni atsitiktinių generatorių metodai, pvz. Permutacinis kongruentinis generatorius (angl. *Permuted Congruential Generator, PCG*), atspindi šioje srityje vykstantį nuolatinį inovacijų procesą. Tačiau, nors naujų optimizavimo algoritmų ir atsitiktinių generatorių atsiradimas žada sprendimus sudėtingoms problemoms, tai taip pat reikalauja nuolatinio analizavimo ir įvertinimo, siekiant užtikrinti optimalų jų panaudojimą.

Keletas tyrimų nagrinėjo atsitiktinių skaičių generavimo metodų poveikį optimizavimo algoritmų efektyvumui. Pavyzdžiui, Blum ir Roli [BR18] apžvelgė atsitiktinių skaičių generatorių vaidmenį metaeuristiniuose algoritmuose, pabrėždami, kad atsitiktinumo kokybė reikšmingai veikia algoritmo veikimą. Kitas tyrimas, kurį atliko Schumann ir Schürmann [SS16], nagrinėjo Monte Karlo simuliacijas optimizavimo kontekstuose, nustatydamas, kad pseudo-atsitiktinių skaičių generatoriai dažnai užtikrina geresnį nuoseklumą ir patikimumą. Šie tyrimai paprastai nustato, kad atsitiktinių skaičių generatorių pasirinkimas ir įgyvendinimas yra esminiai optimizavimo algoritmų efektyvumui ir veiksmingumui.

Šio tyrimo metu bus analizuojami skirtingi atsitiktinių skaičių generatoriai, siekiant nustatyti, kuris yra efektyviausias. Taip pat bus tiriama, kaip skirtingi generatoriai daro įtaką optimizavimo algoritmų rezultatams, veiksmingumui ir ar yra kokių nors koreliacijų tarp jų savybių ir algoritmų efektyvumo.

Pagrindinis šio darbo tikslas – naudojant įvairius atsitiktinius generatorius, optimizavimo algoritmų testavimo funkcijas nustatyti, ar atsitiktiniai skaičių generatoriai turi įtakos optimizavimo algoritmams. Darbo tikslui pasiekti yra iškelti uždaviniai:

- Pasirinkus keletą atsitiktinių skaičių generatorių nustatyti kuris yra geresnis pagal tam tikrą testą
- Pasirinkti ir sudaryti optimizavimo algoritmų testo funkcijų rinkinį
- Pasirinkti ir įgyvendinti kelis optimizacijos algoritmus
- Ištirti ir atlikti atsitiktinių generatorių įtakos analizę

1. Susijusių darbų analizė

[LJC19] straipsnyje yra sprendžiamas parduotuvės lankstaus grafiko planavimo uždavinys. Tai uždavinių tipas, kur kiekvienos užduoties sprendimas reikalauja tam tikros veiksmų sekos, kelio, kad būtų atliktas su skirtingomis mašinomis [DDS+24]. Šiai problemai spręsti buvo pasitelktas plačiai žinomas genetinis algoritmas (angl. *Genetic Algorithm, GA*). Autoriai aiškina, kaip skirtingai sugeneruotos skaičių sekos pasinaudojant atsitiktiniais skaičių generatoriais keičia optimizacijos algoritmo optimalaus arba beveik optimalaus sprendimo radimą. Straipsnyje analizuojami du pagrindiai faktoriai: kokybinis ir efektyvumo. Kokybinis faktorius padeda nustatyti, kaip greitai veikia optimizacijos algoritmas, o efektyvumo faktorius - kiek iteracijų prireikė norint surasti gerą sprendimą. [MV12] straipsnyje tiriamas dalelių spiečiaus optimizavimo (PSO) algoritmo efektyvumas, priklausomai nuo skirtingų atsitiktinių skaičių generatorių (angl. *Random number generator, RNG*) naudojimo. PSO yra biologiškai įkvėptas optimizavimo metodas, taikomas sprendžiant sudėtingas inžinerines problemas, dažnai pasitelkiant atsitiktinius skaičius. Nors tokių generatorių įtaka PSO veikimui yra svarbi, tyrimų šioje srityje atlikta nedaug. Šiame darbe nagrinėjamas skirtingų RNG poveikis tiek dalelių spiečiaus inicializacijai (pradinėms padėtims ir greičiams), tiek jų greičių atnaujinimui. Trijose realiose elektromagnetinėse problemose, susijusiose su antenų projektavimu, išbandomas PSO veikimo su skirtingais ASG efektyvumas. Nors visi tirti generatoriai sugebėjo pasiekti projektavimo tikslus, tyrimas atskleidė, kad Gauso generatorius (angl. *Gaussian random number generator*) konverguoja (randa sprendimą) gerokai sparčiau nei kiti naudotieji variantai.

Dar vienas iš reikšmingų tyrimų, kuriame buvo naudojami atsitiktinių skaičių generatoriai ir optimizavimo algoritmai, nagrinėja hibridinio gleivūnų (angl. *Slime mold algorithm, SMA*) ir aritmetinio optimizavimo (angl. *Arithmetic optimization Algorithm, AOA*) algoritmus. Šiame tyrime naudojami atsitiktinių skaičių generatoriai, siekiant pagerinti optimizavimo proceso efektyvumą. gleivūnų algoritmas (SMA) pasižymi stipriomis globalios paieškos savybėmis, tačiau vėlesnėse iteracijų stadijose jo optimizavimo galimybės silpnėja. Norint tai išspręsti, buvo integruotas aritmetinis optimizavimo algoritmas (AOA), kuris naudoja daugybos ir dalybos operacijas pozicijų atnaujinimui, taip suteikdamas stiprią atsitiktinumo ir geras konvergavimo savybes. Tyrimo metu buvo pritaikyta atsitiktinio centro sprendimo strategija (angl. *Random central solution strategy*), mutacijos strategija ir atnaujinimo strategija, siekiant pagerinti globalios paieškos efektyvumą ir algoritmo populiacijos įvairovę. Eksperimentai parodė, kad ši hibridinė metodika, derinanti SMA ir AOA algoritmus, efektyviai sprendžia optimizavimo uždavinius ir pagerina konvergavimo tikslumą. Šie rezultatai rodo, kad atsitiktinių skaičių generatoriai gali žymiai prisidėti prie sudėtingų optimizavimo problemų sprendimo, suteikdami algoritmams reikalingą lankstumą ir efektyvumą [Abu+23].

Daugelis autorių moksliniuose straipsniuose nagrinėja tik vieną, tam tikram uždaviniui spręsti pritaikytą, optimizacijos algoritmą ir kaip atsitiktinai sugeneruotos skaičių sekos gali privesti prie skirtingų rezultatų ir išvadų. Kaip ir Resende, F. J. ir Costa, B. V. [RC98] darbe analizuojama Monte-Karlo simuliacija pasinaudojant atsitiktinių skaičių generatoriais. Šiame tyrime autoriai naudojo tik 2 atsitiktinių skaičių generatorius sprendžiant problemą: $RAN2$ ir $RANLUX$. Tyrimo rezultatai parodė potencialią problemą, kuri susijusi su atsitiktinių skaičių generatoriais - pastebėtas generatorių nepastovumas sukelia potencialią grėsmę simuliacijoms. Taip pat tyrimo metu pastebėta, kad nepakanka patikrinti atsitiktinių skaičių generatorių Aizingo modelyje (angl. *Ising model*), kaip yra įprasta simuliacijų bendruomenėje.

Šiame darbe bus pasirinkti keli atsitiktinių skaičių generatoriai, tuomet pasinaudojant Kolmogorov - Smirnov testu [Jr51] bus įvertintas jų gerumas. Atsitiktinių skaičių generatoriai bus integruoti į kelis optimizacijos algoritmus ir palyginamas sąryšis tarp jų ir optimizacijos algoritmų gerumo.

2. Atsitiktiniai skaičių generatoriai

2.1. Tiesinis kongruentinis generatorius

Tiesinis kongruentinis generatorius (angl. *Linear Congruential Generator*, LCG) yra vienas iš seniausių ir paprasčiausių pseudo-atsitiktinių skaičių generatorių. Jis veikia naudodamas tiesinę pasikartojančią formą, kuri apibrėžiama kaip:

$$X_{n+1} = (aX_n + c) \mod m, \tag{1}$$

kur X yra generuojamų skaičių seka, a, c ir m yra konstantos, kurios atitinkamai vadinamos daugikliu, poslinkiu ir moduliu [Knu97]. Šis paprastas algoritmas leidžia lengvai įgyvendinti ir suprasti generatorius.

LCG praktiniame naudojime rodo, kad pasirinkus tinkamas konstantas, galima užtikrinti ilgą periodą ir gerą atsitiktinių skaičių kokybę [LEc88]. LCG naudojamas įvairiose srityse, įskaitant simuliacijas, statistinę mėginių atranką ir kriptografiją. Tačiau dėl savo paprastumo LCG turi tam tikrų trūkumų, tokių kaip ribotas periodas ir polinkis generuoti skaičių sekas, kurios turi tam tikrus pasikartojimus ir koreliacijas [PM88].

LCG yra įdiegtas daugelyje programavimo kalbų ir programinėse įrangose: C ir C++ standartinė biblioteka ir MATLAB [Mat23], tačiau norint naudoti LCG dirbant su Python, vartotojas turi savarankiškai apsibrėžti tiesinį kongruentinį generatorių. Nepaisant tam tikrų apribojimų, LCG yra plačiai naudojamas dėl savo paprastumo ir efektyvumo.

2.2. Permutacinis kongruentinis generatorius

Permutacinis kongruentinis generatorius (angl. *Permuted Congruential Generator*, PCG) yra šeima paprastų, greitų ir aukštos kokybės pseudo-atsitiktinių skaičių generatorių, sukurtų Melisos O'Neill [ONe14] 2014 metais. PCG garsėja savo veikimu ir statistine kokybe, siūlydami gerą atsitiktinumą su santykinai paprastu įgyvendinimu. Pagrindinė PCG idėja yra sujungti LCG su elementų sukeitimo funkcija, kad pagerėtų išvesties kokybė.

Pagrindinė PCG forma apima LCG, kuris generuoja skaičių seką, kuri vėliau yra mutuojama naudojant elementų sukeitimo funkciją, kad bitai būtų sumaišyti ir sumažėtų koreliacijos. Šis požiūris padeda pasiekti geresnes statistines savybes, palyginti su tradiciniais LCG. Standartinis PCG algoritmas gali būti apibrėžtas kaip:

$$X_{n+1} = (aX_n + c) \mod 2^k, \tag{2}$$

kur a ir c yra konstantos, o k yra bitų skaičius būsenoje. Išvesties elementai tuomet yra sukeičiami prieš grąžinant ją kaip atsitiktinį skaičių [ONe14].

PCG turi kelis variantus, kiekvienas skirtas optimizuoti skirtingus veikimo ir kokybės aspektus: PCG-XSH-RR, PCG-XSH-RS, PCG-RXS-M-XS ir kitus. Kiekvienas kokybės ir veikimo aspektas siūlo skirtingus kompromisus tarp greičio, atminties naudojimo ir atsitiktinumo kokybės. PCG yra ypač populiarūs programose, kuriose reikalingas aukštos kokybės atsitiktinumas, pvz., simuliacijose, žaidimuose ir statistinėje atrankoje.

PCG yra įdiegtas keliose programavimo kalbose, įskaitant C, C++ ir Python, ir yra plačiai naudojamas dėl savo paprastumo, efektyvumo ir puikių statistinių savybių.

2.3. Mersenne Twister

Mersenne Twister (MT) yra plačiai naudojamas pseudo-atsitiktinių skaičių generatorius, sukurtas 1997 m. Makoto Matsumoto ir Takuji Nishimura [MN98]. Jis garsėja aukšta kokybe ir greitu atsi-

tiktinių skaičių generavimu, veikia remiantis matricine tiesine rekurencija baigtiniame dvejetainiame lauke. Jis pavadintas Mersenne pirminio skaičiaus vardu, konkrečiai $2^{19937} - 1$, kuris atspindi jo generuojamos sekos periodą.

Vienas iš pagrindinių Mersenne Twister bruožų yra itin ilgas $2^{19937} - 1$ periodas, todėl jis tinka programoms, kurioms reikia daug atsitiktinių skaičių. Be to, jis turi 623 dimensijų tolygų pasiskirstymą, kuris užtikrina, kad generuojami skaičiai yra tolygiai pasiskirstę aukštos dimensijos erdvėse. Šios savybės daro jį ypač naudingą srityse, kuriose svarbi atsitiktinių skaičių kokybė ir nenuspėjamumas, pvz. kriptografijoje [LEc17].

Mersenne Twister generatorius yra itin vertinamas dėl savo efektyvumo ir patikimumo simuliacijose ir statistiniame mėginyje. Jis plačiai naudojamas Monte-Karlo metoduose, kurie taikomi įvairiose mokslinėse ir inžinerinėse disciplinose modeliuojant sudėtingas sistemas ir procesus. Jis taip pat naudingas ir finansiniame modeliavime, nes tiksli ir patikima atsitiktinių skaičių generacija yra būtina rinkos elgesio modeliavimui ir rizikos vertinimui [Nis00].

Mersenne Twister yra įdiegtas daugelyje programavimo kalbų ir programinės įrangos: Python `random` modulis [Pyt23], MATLAB [Mat23], R ir C++. Jo universalumas ir patikimumas daro jį standartiniu pasirinkimu atsitiktinių skaičių generacijai šiose aplinkose. Be to, jo vaidmuo kompiuterinėje grafikoje, ypač procedūriniame generavime ir natūralių reiškinių simuliacijoje, demonstruoja jo praktinius pritaikymus įvairiose srityse [MN98].

Tyrimai plačiai pademonstravo Mersenne Twister veikimą, patvirtindami jo tinkamumą įvairioms programoms. Mersenne Twister išlieka vienu patikimiausių ir plačiausiai naudojamų atsitiktinių skaičių generatorių.

2.4. Kolmogorov-Smirnov testas

Kolmogorov-Smirnov (KS) testas yra plačiai naudojamas statistinis metodas, kuris yra skirtas dviejų kumuliacinių pasiskirstymo funkcijų palyginimui arba vienos funkcijos palyginimui su teorine pasiskirstymo funkcija. KS testas ypač naudingas vertinant atsitiktinių skaičių generatorių kokybę, nes jis leidžia patikrinti, ar sugeneruoti atsitiktiniai skaičiai atitinka norimą statistinį skirstinį. Ši savybė yra itin svarbi optimizavimo algoritmų efektyvumui, nes geros kokybės atsitiktiniai skaičiai gali ženkliai pagerinti šių algoritmų veikimą.

Atsitiktinių skaičių generatorių kokybės vertinimas yra kritinis žingsnis kuriant efektyvius optimizavimo algoritmus. Naudojant KS testą, galima nustatyti, ar generatoriaus sugeneruoti skaičiai iš tiesų yra atsitiktiniai ir ar jie tinkamai pasiskirstę. Vienos imties KS testas tikrina hipotezę H_0: atsitiktinė skaičių seka yra pasiskirsčiusi pagal tolygųjį skirstinį, o alternatyva šiai hipotezei – H_1: atsitiktinė skaičių seka nėra pasiskirsčiusi pagal tolygųjį skirstinį. Jei generatorius tenkina KS testą, tai reiškia, kad sugeneruoti atsitiktiniai skaičiai tikėtina turės mažiau koreliacijų ir bus labiau tinkami sudėtingoms optimizavimo užduotims spręsti [Knu97].

KS testas yra pagrįstas didžiausiu atstumu tarp dviejų kumuliacinių pasiskirstymo funkcijų. Tarkime, kad turime empirinę kumuliacinę pasiskirstymo funkciją $F_n(x)$ iš mėginio dydžio n ir teorinę kumuliacinę pasiskirstymo funkciją $F(x)$. KS statistika apibrėžiama kaip:

$$D_n = \sup_x |F_n(x) - F(x)|, \tag{3}$$

čia *sup* reiškia supremumo funkciją, kuri suranda didžiausią atstumą tarp dviejų funkcijų. D_n reikšmė yra lyginama su KS testo kritine reikšme – D_α, kuri apskaičiuojama pagal formulę:

$$D_\alpha = \frac{K_\alpha}{\sqrt{n}}, \tag{4}$$

čia n yra generuojamų stebėjimų skaičius, o K_α yra konstanta, kuri priklauso nuo pasirinkto

reikšmingumo lygmens. Dažniausiai tyrimuose pasirenkamas reikšmingumo lygmuo yra 0.05, o naudojama konstanta yra lygi 1.36, tačiau vartotojas turi galimybę reikšmingumo lygmenį pasirinkti pats.

Kuo didesnė D_n vertė už D_α, tuo didesnis neatitikimas tarp stebimų ir numatomų skirstinių, t.y. atmetame nulinę hipotezę. Priešingai, kuo mažesnė D_n vertė už D_α, tuo arčiau empirinė pasiskirstymo funkcija yra teorinei funkcijai, o tai reiškia, priimame nulinę hipotezę ir galime teigti, kad generatorius imituoja tolygiojo skirstinio pasiskirstymą [Jr51].

Apskaičiuota p-reikšmė nurodo tikimybę, kad stebimi duomenys galėtų būti gauti pagal tolygiojo skirstinio modelį. Jei p-reikšmė yra didesnė nei pasirinktas reikšmingumo lygmuo, nulinė hipotezė neatmetama, tai yra, pripažįstame, kad duomenys atitinka tolygųjį skirstinį.

Vienas iš svarbiausių KS testo privalumų yra jo jautrumas skirtumams tarp empirinės ir teorinės pasiskirstymo funkcijos. Tai leidžia tiksliai nustatyti net smulkius neatitikimus, kurie gali turėti didelę įtaką optimizavimo algoritmų veikimui. Pavyzdžiui, optimizavimo algoritmai, tokie kaip genetinis algoritmas ar dalelių spiečiaus optimizavimas, labai priklauso nuo atsitiktinių skaičių kokybės. Net ir nedideli neatitikimai pasiskirstyme gali lemti prastesnę sprendimų kokybę ar lėtesnį algoritmų konvergavimą [GDT+09].

KS testas taip pat naudojamas įvairiuose tyrimuose, siekiant įvertinti atsitiktinių skaičių generatorių tinkamumą specifinėms taikymo sritims. Pavyzdžiui, atliktas tyrimas parodė, kad naudojant Mersenne Twister (MT) generatorių, optimizavimo algoritmai veikia efektyviau, palyginus su kitais generatoriais, kurie netenkina KS testo [MN98]. Tai rodo, kad generatoriaus pasirinkimas gali turėti didelę įtaką optimizavimo algoritmų veikimui ir jų galutiniams rezultatams.

KS testas yra naudingas ne tik patikrinant atsitiktinių skaičių generatorių kokybę, bet ir lyginant skirtingų optimizavimo algoritmų veikimą. Pavyzdžiui, galima naudoti KS testą, norint palyginti dviejų skirtingų algoritmų sprendimų pasiskirstymą ir įvertinti, kuris algoritmas generuoja geresnius sprendimus pagal tam tikrus kriterijus [PTV+07].

Apibendrinant, Kolmogorov-Smirnov testas yra svarbus įrankis atsitiktinių skaičių generatorių kokybės vertinimui, kuris tiesiogiai veikia optimizavimo algoritmų efektyvumą. Tinkamai parinktas generatorius, tenkinantis KS testą, gali žymiai pagerinti optimizavimo procesų rezultatus, tuo tarpu netinkamas generatorius gali lemti prastus sprendimus ir lėtą konvergavimą. Todėl KS testo taikymas yra esminis žingsnis kuriant ir vertinant optimizavimo algoritmus.

2.5. Skaičiaus PI Generavimas

Skaičiaus PI generavimas naudojant atsitiktinius taškus yra vienas iš būdų įvertinti skaičiavimo tikslumą ir aptarti atsitiktinių skaičių generatorių veikimą. Šiame skyriuje aprašysime metodiką, kaip galime generuoti skaičių PI naudodami atsitiktinių skaičių generatorius.

PI skaičiaus generavimas vyksta taikant Monte-Karlo simuliacijos principą [KW08]. Pagrindinė idėja yra simuliuoti atsitiktinius taškus kvadrate, po to nustatyti, kiek iš tų taškų patenka į ketvirtį apskritimo, įbrėžto į tą kvadratą. Tai atliekama vykdant kelis žingsnius. PI generavimo metodika naudojant atsitiktinius skaičius yra aprašyta pseudo kodu (1 algoritmas).

Atsitiktinių taškų generavimas vyksta parenkant tam tikrą skaičių atsitiktinių taškų, kurie bus simuliuojami. Atsitiktiniai skaičiai yra sugeneruojami (x, y) porų pavidalu, kur x ir y yra atsitiktiniai skaičiai tarp -1 ir 1. Kiekvienas sugeneruotas atsitiktinis taškas (x, y) tikrinamas, ar jis patenka į apskritimą, kurio centras yra kvadrato centre ir spindulys yra vienetas. Tai nustatoma apskaičiavus atstumą nuo taško iki koordinatės pradžios taško ir tikrinant, ar atstumas yra mažesnis nei vienetas.

1 algoritmas PI generavimas naudojant atsitiktinius skaičius

Funkcija `GeneruotiPi()`
`# Apsibrėžti kintamuosius`
Įeitis: Išviso taškų $= 0$
Įeitis: Taškai apskritimo viduje $= 0$
Įeitis: Iteracijų skaičius $= 10000$
for $i = 1$ **iki** $num\_iterations$ **do**
    Sugeneruoti atsitiktinį tašką (x, y) tarp -1 ir 1
    if $\sqrt{x^2 + y^2} \leq 1$ **then**
        Taškai apskritimo viduje $=$ Taškai apskritimo viduje $+1$
    end if
    Išviso taškų $=$ Išviso taškų $+1$
end for
Apskaičiuoti PI reikšmę:
    $pi\_artima = 4 \times \frac{\text{Taškai apskritimo viduje}}{\text{Išviso taškų}}$
Išvesti pi_artima
Palyginti su tikrąja π reikšme

Apskaičiuojama, kiek taškų patenka į apskritimą ir pagal Monte-Karlo simuliacijos principą įvertinama PI reikšmė. Taip pat apskaičiuojama absoliutinė paklaida tarp įvertinto PI ir tikrojo PI (kuris yra žinomas ir lygus matematinei π reikšmei). Rezultatai, gauti naudojant tris skirtingus atsitiktinių skaičių generatorius (Mersenne Twister, permutacinį kongruentinį ir tiesinį kongruentinį), rodo įvairius PI reikšmės įvertinimus ir jų tikslumą. Kiekvienam generatoriui buvo vykdomos kelios simuliacijos, o gautos PI reikšmės buvo lyginamos su tikrąja PI reikšme.

3. Atsitiktinės paieškos optimizavimo algoritmai

Optimizavimo algoritmai yra skaičiavimo metodai, skirti rasti geriausią sprendimą konkrečiai problemai, atsižvelgiant į nustatytus apribojimus ir kriterijus. Šios problemos dažnai apima tam tikro objekto funkcijos maksimizavimą ar minimizavimą, kas gali reikšti kaštų, efektyvumo, klaidos ar bet kokio kito su užduotimi susijusio našumo rodiklio optimizavimą.

Optimizavimo problemas galima suskirstyti į įvairias kategorijas pagal objekto funkcijos ir apribojimų pobūdį. Tai apima tiesinį ir netiesinį optimizavimą, diskrečią ir nepertraukiamą optimizaciją, ribotą ir neribotą optimizaciją. Kiekvienai problemų klasei reikalingi skirtingi metodai ir algoritmai.

Optimizavimo algoritmus galima suskirstyti į dvi pagrindines kategorijas: deterministinius ir stochastinius [Cav13]. Deterministiniai algoritmai vykdo iš anksto nustatytą operacijų seką ir pateikia tą patį rezultatą, kai įvedamas tas pats pradinis duomenų rinkinys. Pavyzdžiai apima gradientų nusileidimą ir simplex metodą. Stochastiniai algoritmai, kita vertus, įtraukia atsitiktinumą į savo procesus, todėl jie yra tinkami sudėtingoms, multimodalinėms arba blogai suprantamoms aplinkoms. Pavyzdžiai apima genetinius algoritmus ir modeliuojamą atkaitinimą.

Šiuolaikiniai optimizavimo algoritmai semiasi įkvėpimo iš įvairių sričių. Kai kurie iš jų remiasi biologiniais procesais, pvz.: genetinis algoritmas (GA), kuris imituoja natūralią atranką ir dalelių spiečiaus optimizavimas (PSO), kuris įkvėptas paukščių ir žuvų socialinio elgesio [Tro23]. Kiti remiasi fizikiniais procesais, tokiais kaip modeliuojamas atkaitinimas (SA), kuris remiasi metalurgijos atkaitinimo procesu.

Optimizavimo algoritmų kūrimą ir taikymą skatino poreikis spręsti realaus pasaulio problemas įvairiose srityse. Inžinerijoje jie naudojami efektyvių struktūrų ir sistemų projektavimui. Ekonomikoje jie padeda išteklių paskirstymui ir rizikos valdymui. Mašininio mokymosi srityje jie optimizuoja modelio parametrus, siekiant pagerinti prognozavimo našumą. Kadangi skaičiavimo galia toliau auga, gebėjimas spręsti vis sudėtingesnes optimizavimo problemas taip pat didėja, todėl optimizavimo algoritmai yra būtinas modernus analitinis įrankis.

3.1. Paprastoji atsitiktinė paieška

Paprastoji atsitiktinė paieška (angl. *Pure random search, PRS*) yra pagrindinis optimizavimo algoritmas, kuriame kandidatai yra generuojami atsitiktinai paieškos erdvėje, o gautas sprendimas, su kandidatų pora, po kiekvienos iteracijos yra lyginamas su globaliu minimumu. Ir jei skirtumas atitinka pasirinktas sąlygas jis yra traktuojamas kaip sėkmingai rastas sprendimas. Nepaisant jos paprastumo, PRS gali būti veiksminga tam tikrose scenarijose, ypač kai paieškos erdvė yra didelė ir sudėtinga, o apie objektyvios funkcijos struktūra mažai arba visiškai nėra žinoma.

Pagrindinis PRS principas yra atsitiktinė paieška erdvėje. Matematiškai PRS gali būti aprašoma taip:

1. Apibrėžkite paieškos erdvę S ir objektyvios funkcijos $f : S \to \mathbb{R}$.
2. Nustatykite iteracijų skaičių N.
3. Inicializuoti sprendimą x ir jo funkcijos vertę f_x.
4. Nuosekliai nuo $i = 1$ iki N:
    - Sugeneruokite atsitiktinį sprendimą $x_i \in S$.
    - Įvertinkite $f(x_i)$.
5. Gražinkite x jei $f(x_i)$ tenkina paklaidos sąlygas

PRS yra viena iš ankstyviausių optimizavimo technikų, kurios šaknys kyla iš Monte-Karlo metodų. Monte-Karlo metodus pristatė Nicholas Metropolis ir Stanislaw Ulam 1949 metais [MU49]. PRS paprastumas daro jį patraukliu pradiniu tašku optimizavimo problemoms, ypač kai skaičiavimo ištekliai riboti arba problemos struktūra nežinoma.

Pagrindinis PRS principas yra atsitiktinumas. Skirtingai nei sudėtingesni algoritmai, PRS nenaudoja gradiento informacijos ar kitų heuristinių priemonių paieškai nukreipti. Šis krypties nebuvimas gali būti tiek privalumas, tiek trūkumas. Pagrindinis PRS privalumas yra paprastumas, šį algoritmą lengva įgyvendinti ir suprasti, taip pat jis nereikalauja papildomų prielaidų ar informacijos apie gradientą ar kitas problemas. Didžiausias PRS trūkumas - neefektyvus algoritmo veikimas. Įgyvendinti šį algoritmą reikalingi papildomi funkcijų įvertinimai, o tai padidina skaičiavimo sąnaudas. Taip pat algoritmo našumas mažėja didėjant paieškos erdvės dimensijai, o tai apsunkina sprendimų paiešką sudėtingesnėse situacijose. Be to, trūksta garantijos, kad metodas pasieks optimalų sprendimą, ypač aukštų dimensijų erdvėse, kur paieška tampa dar sudėtingesnė.

Nepaisant savo trūkumų, PRS buvo taikoma įvairiose srityse. Pavyzdžiui, ji buvo naudojama:

- **Parametrų derinimo mašininio mokymo metoduose**: tyrimų darbuose dažnai naudojama PRS, kad būtų sureguliuoti hiperparametrai mašininio mokymo modeliams, kai nėra išankstinės informacijos apie optimalius parametrų nustatymus [BB12].
- **Inžinerijos projektavimas**: PRS gali būti naudinga inžinerijos projektavimo pradžioje, kad būtų ištirtas įvairių potencialių sprendimų diapazonas prieš taikant sudėtingesnius optimizavimo metodus [Sim15].
- **Aplinkosaugos modeliavimas**: PRS buvo naudojama sudėtingų aplinkos modelių kalibravimui, kur santykis tarp modelio parametrų ir rezultatų yra labai netiesinis ir prastai suprantamas [RTB12].

3.2. Dalelių spiečiaus optimizavimas

Dalelių spiečiaus optimizavimas (angl. *Particle swarm optimization, PSO*) yra optimizavimo algoritmas, kuris modeliuoja bandomų dalelių elgesį, siekiant rasti optimalų sprendimą. PSO yra populiarus algoritmas, naudojamas spręsti įvairias optimizavimo problemas, įskaitant neapibrėžtas funkcijas.

PSO algoritmo pagrindinis principas yra modeliuoti bandomų dalelių judėjimą paieškos erdvėje. Kiekviena dalelė yra potencialus sprendimas, o jos judėjimas yra paremtas geriausios dalelės ir geriausio vietinio sprendimo pozicijos atnaujinimu. Dalelių spiečiaus algoritmo pradžioje yra atsitiktinai inicializuojamos dalelių pozicijos ir greičiai paieškos erdvėje. Tuomet yra atnaujinami kiekvienos dalelės greičiai pagal geriausiai rastus globalius ir lokalius sprendimus. Galiausiai yra atnaujinamos dalelių pozicijos pagal naujus greičius. Paskutiniai du žingsniai yra kartojami, kol yra pasiekiama nustatyta sustojimo sąlyga.

Dalelių spiečiaus optimizavimas yra įkvėptas paukščių pulkų ar žuvų būrių socialinio elgesio. Jis apima kandidatų sprendimų populiaciją, vadinamą dalelėmis, kurios juda per paieškos erdvę. Kiekviena dalelė koreguoja savo poziciją pagal savo patirtį ir kaimynų patirtį. PSO yra žinomas dėl savo paprastumo ir efektyvumo sprendžiant nepertraukiamo optimizavimo problemas.

PSO buvo sukurtas Kennedy ir Eberhart 1995 metais, remiantis socialinio elgesio ir žmonių bandymo stiliaus idėjomis [KE95]. Šis algoritmas iš pradžių buvo sukurtas modeliuoti bandos elgesį, tačiau vėliau buvo pritaikytas spręsti optimizavimo problemas.

Pagrindiniai PSO privalumai yra lengvas įgyvendinimas ir greitas konvergavimas į gerus sprendimus. Tačiau PSO kartais gali užstrigti lokaliuose minimumuose arba maksimumuose ir gali prireikti parametrų, tokių kaip inercijos svoris ir greitėjimo koeficientai, derinimo [ES98].

PSO buvo sėkmingai pritaikytas įvairiose srityse, įskaitant inžineriją, mašininį mokymą ir ekonomiką. Pavyzdžiui, jis buvo naudojamas parametrų optimizavimui neuroninių tinklų mokyme ir taikytas ekonomikos modeliavime.

3.3. Genetinis algoritmas

Genetinis algoritmas (angl. *Genetic algorithm, GA*) yra populiacijos pagrindu veikiantis optimizavimo metodas, įkvėptas natūralios atrankos ir genetikos principų. GA palaiko kandidatų sprendimų populiaciją ir iteratyviai vykdo atrankos, kryžminimo bei mutacijos operacijas, siekiant sukurti naujas generacijas, kol randamas tinkamas sprendimas. Šis algoritmas yra itin naudingas, kai paieškos erdvė yra didelė ir tradiciniai optimizavimo metodai (pvz., gradientų metodai) nėra efektyvūs arba tinkami.

Pagrindinis GA principas yra imituoti evoliuciją naudojant genetinius operatorius, tokius kaip kryžminimas, mutacija ir atranka. Kiekvienas potencialus sprendimas yra koduojamas genetine informacija ir vertinamas pagal objektyviąją funkciją. Genetinio algoritmo pradžioje yra atsitiktinai inicializuojama pradinė populiacija. Kiekvienas individas populiacijoje yra atskirai įvertinamas pagal pasirinktą funkciją. Įvertinus kiekvieną individą, yra taikomi genetinio algoritmo operatoriai (kryžminimas, mutacija, atranka) populiacijai, generuojant naują individų generaciją (angl. *Generation*). Paskutiniai du žingsniai yra kartojami nustatytą generacijų skaičių arba kol yra pasiekiama sustojimo sąlyga.

Genetinio algoritmo ištakos siekia 1960-uosius metus, kai Johnas Hollandas, amerikiečių biologas ir kompiuterių mokslininkas, pradėjo nagrinėti biologinės evoliucijos procesus kaip modelį problemų sprendimui kompiuterinėse sistemose. 1975 metais jis išleido knygą ”Adaptacija natūraliuose ir dirbtiniuose sistemose” (angl. *Adaptation in Natural and Artificial Systems*) [Hol75], kurioje išsamiai aptarė evoliucijos principų taikymą sprendimų paieškoje ir optimizavime. Nors pirminis genetinio algoritmo taikymas buvo ribotas, 1980-aisiais ir 1990-aisiais GA pradėjo plačiau naudoti inžinerijos, mašininio mokymosi, ekonomikos bei kitose srityse. GA patrauklumas ir lankstumas paskatino didelę mokslininkų ir praktikų bendruomenę tobulinti šį metodą, kuris dabar yra naudojamas įvairiose srityse ir yra vienas iš pagrindinių evoliucinių algoritmų tipų.

Genetinio algoritmo veikimas remiasi keliais pagrindiniais principais:

1. **Inicializacija**: Sugeneruojama pradinė populiacija – atsitiktinai parinktas galimų sprendimų rinkinys. Populiacijos dydis paprastai nustatomas atsižvelgiant į problemos sudėtingumą ir prieinamus resursus.

2. **Atranka**: Populiacijos individai (sprendimai) yra įvertinami pagal tinkamumo (angl. *fitness*) funkciją, kuri matuoja, kaip gerai sprendimas atitinka problemos reikalavimus. Atrankos procesas parenka tinkamiausius individus, kurie bus naudojami kitos generacijos sprendimų kūrimui. Dažnai naudojami tokie metodai kaip turnyro atranka.

3. **Kryžminimas (angl. *crossover)***: Atrinkti individai suporuojami ir kryžminami, kad būtų generuojami nauji individai (palikuonys). Kryžminimo metu tėvų genai yra sujungiami, sudarant naujus sprendimus. Dažnai naudojamas vienos arba dviejų taškų kryžminimas, kai naujasis individas paveldi dalį genų iš kiekvieno tėvo.

4. **Mutacija**: Kiekvienas naujas individas gali patirti mutaciją – atsitiktinį genų pasikeitimą, kuris padeda išvengti lokalių minimumų ar maksimumų paieškos erdvėje. Mutacijos dažnis paprastai yra mažas, bet svarbus populiacijos įvairovei palaikyti.

5. **Elitiškumas**: Siekiant užtikrinti, kad geriausi individai nebūtų prarasti, kartais į kitos generacijos tiesiogiai perkeliami kai kurie geriausi individai.

6. **Kartojimas**: Procesas kartojamas, kol pasiekiamas sprendimas, kuris atitinka optimizavimo tikslus, arba kol praeina nustatytas generacijų skaičius.

Genetinis algoritmas turi daug privalumų, dėl kurių jis yra plačiai pritaikomas įvairiose srityse. Dėl lankstumo genetiniai algoritmai tinka įvairioms problemoms – tiek nepertraukiamo, tiek diskretaus optimizavimo. Genetinis algoritmas nėra linkęs įstrigti lokaliuose minimumuose, todėl dažnai randa globalų sprendimą. Taip pat jis reikalauja tik tinkamumo funkcijos, o ne specifinės matematinės informacijos apie problemą, todėl yra universalus ir lengvai pritaikomas įvairiems tikslams.

Nepaisant privalumų, genetinis algoritmas taip pat turi keletą trūkumų, kurie gali apsunkinti jo taikymą tam tikrose situacijose. Didžiausias trūkumas, kuris yra akcentuojamas, tai didelės skaičiavimo sąnaudos. GA yra skaičiavimo požiūriu brangus, ypač kai individų skaičius populiacijoje arba generacijų skaičius yra didelis. Taip pat pasirinktas populiacijos dydis, mutacija ir kiti genetinio algoritmo operatoriai dažnai turi didelę įtaką algoritmo veikimui ir reikalauja atidžios optimizacijos. Galiausiai pats algoritmas nėra pilnai pritaikytas spręsti mažas problemas - kai problemos sprendimo paieškos erdvė yra maža, tradiciniai metodai dažnai būna efektyvesni nei GA.

Genetinis algoritmas yra plačiai taikomas įvairiose srityse:

- **Inžinerinis projektavimas**: GA naudojamas optimizuoti konstrukcijų dizainą, pavyzdžiui, statinių ar mechaninių komponentų struktūras. Pritaikant genetinį algoritmą, galima optimizuoti tokias savybes kaip medžiagų sąnaudos ar konstrukcijos patvarumas.
- **Ekonomika ir finansai**: Portfelio optimizavimas, prekybos strategijų kūrimas ir rizikos valdymas yra sritys, kuriose GA padeda surasti optimalias investicijų kombinacijas arba prekybos modelius, atsižvelgiant į rinkos sąlygas ir investavimo riziką.
- **Mašininis mokymasis ir dirbtinis intelektas**: GA dažnai naudojami mašininio mokymosi modelių ypatybių parinkimui. Pavyzdžiui, Siedlecki ir Sklansky [SS89] naudojo GA, kad pasirinktų svarbiausias ypatybes raštų klasifikavimo problemoms, padedant pagerinti klasifikatoriaus efektyvumą. GA taip pat naudojamas neprižiūrimame mokyme, pavyzdžiui, klasterizacijos uždaviniams spręsti.
- **Tvarkaraščių sudarymas**: GA taikomas tvarkaraščių sudarymui universitetuose, gamybos linijose ar transporto sektoriuje, kur reikia optimizuoti išteklių naudojimą ir užtikrinti, kad visi tvarkaraščio apribojimai būtų įvykdyti.
- **Bioinformatika ir medicinos tyrimai**: GA padeda genų sekų suderinimo, vaistų kūrimo ir kitose su biologija susijusiose užduotyse. Naudojant GA, galima modeliuoti evoliucinius procesus ir rasti optimalius sprendimus net labai kompleksiškuose biologiniuose duomenyse.

Genetinis algoritmas yra universalus ir galingas metodas, leidžiantis spręsti įvairias optimizavimo problemas. Dėl savo gebėjimo dirbti su didele paieškos erdve ir globalaus optimumo siekimo galimybės, jis tapo populiariu įrankiu įvairiose mokslo ir pramonės srityse. Nepaisant tam tikrų trūkumų, tokių kaip didelės skaičiavimo sąnaudos ir parametrų derinimo sudėtingumas, genetinis algoritmas yra efektyvus įrankis sudėtingų, realaus pasaulio problemų sprendimui.

3.4. Modeliuojamas atkaitinimas

Modeliuojamas kaitinimas (angl. *Simulated annealing, SA*) yra tikimybinis optimizavimo algoritmas, įkvėptas atkaitinimo proceso metalurgijoje. Jis prasideda nuo pirminio sprendimo ir iteratyviai tyrinėja paieškos erdvę, leisdamas tam tikrus blogesnius sprendimus su tikimybe, kuri nustatoma pagal aušinimo schemą. SA palaipsniui mažina temperatūros parametrą, kontroliuodamas blogesnių sprendimų priėmimą, kai algoritmas progresuoja. Jis ypač efektyvus sprendžiant kombinatorines optimizavimo problemas.

Pagrindinis SA principas yra modeliuoti metalo atkaitinimo procesą, kuris veda prie minimalios energijos būsenos. Algoritmas pasižymi stochastiškumu, leisdamas priimti prasmingus sprendimus

net kai kuriais nepalankiais momentais. Modeliuojamo atkaitinimo algoritmas prasideda nuo atsitiktinio pradinio sprendinio, kuris yra įvertinamas pagal optimizuojamą funkciją. Nauji sprendiniai generuojami modifikuojant esamą sprendinį, o jų priėmimas priklauso nuo temperatūros ir sprendinio tinkamumo skirtumo. Aukštoje temperatūroje priimami ir blogesni sprendiniai, kad būtų išvengta užstrigimo lokaliuose minimumuose ir ištyrinėta platesnė erdvė. Temperatūra palaipsniui mažinama, leidžiant algoritmui sutelkti dėmesį į sprendimo tobulinimą. Procesas baigiasi, kai temperatūra pasiekia nustatytą ribą arba kai sprendimai nustoja gerėti.

Modeliuojamas kaitinimas buvo pristatytas Kirkpatrick, Gelatt ir Vecchi 1983 metais [KGV83]. Šis algoritmas buvo iš pradžių naudojamas spręsti kombinatorines optimizavimo problemas, tokias kaip maršrutų problema.

Pagrindinis SA privalumas yra jo gebėjimas išvengti lokalių minimumų arba maksimumų ir rasti jų globalius, jei skiriama pakankamai laiko. Tačiau SA veikimas labai priklauso nuo aušinimo schemos ir jis gali būti lėtas didelio masto problemoms.

Pagrindiniai SA algoritmo principai apima temperatūros mažinimą ir tikimybės priėmimo funkciją. Temperatūros mažinimas yra esminis SA aspektas, leidžiantis algoritmui palaipsniui pereiti nuo plačios paieškos prie tikslesnės sprendimų optimizacijos. Pradinė aukšta temperatūra leidžia priimti ir prastesnius sprendimus, o tai padeda išvengti užstrigimo lokaliuose minimumuose. Temperatūrai mažėjant algoritmas tampa efektyvesnis priimdamas tik geresnius ar nežymiai blogesnius sprendimus. Tikimybės priėmimo funkcija yra kitas svarbus principas, kuris nustato, kuriuos naujus sprendimus priimti ar atmesti, remiantis sprendimų kokybe ir esama temperatūra. Šių principų derinys leidžia SA algoritmui efektyviai spręsti sudėtingas optimizavimo problemas.

Modeliuojamas kaitinimas (SA) yra plačiai naudojamas įvairiose srityse, įskaitant kombinatorinį optimizavimą, inžineriją ir mokslinius tyrimus. Vienas iš SA panaudojimo pavyzdžių yra keliaujančio prekeivio problema (angl. *Travelling salesman problem, TSP*), kur jis buvo naudojamas rasti beveik optimalius maršrutus dideliam miestų rinkiniui. Johnson ir kt. [JAM+89] pademonstravo, kaip šis sprendimas gali efektyviai spręsti šią sudėtingą problemą, pateikdami aukštos kokybės sprendimus.

4. Testavimo funkcijos

Testavimo funkcijos yra esminė optimizacijos algoritmų vertinimo priemonė, leidžianti įvertinti jų gebėjimą spręsti sudėtingas optimizavimo užduotis. Šios funkcijos dažnai pasižymi sudėtinga struktūra, kuri sukelia iššūkių algoritmui, pavyzdžiui, daugybe lokalių minimumų, galinčių priversti algoritmą stagnuoti arba sukonverguoti ne į globalų funkcijos minimumą. Be to, jos dažnai turi aiškiai apibrėžtą ir tikslų globalų minimumą, kuris reikalauja itin tikslaus algoritmo veikimo, kad šis galėtų jį surasti. Naudojant įvairias testavimo funkcijas, galima išsamiai analizuoti algoritmų privalumus ir trūkumus bei pritaikymą sprendžiant skirtingas optimizavimo problemas.

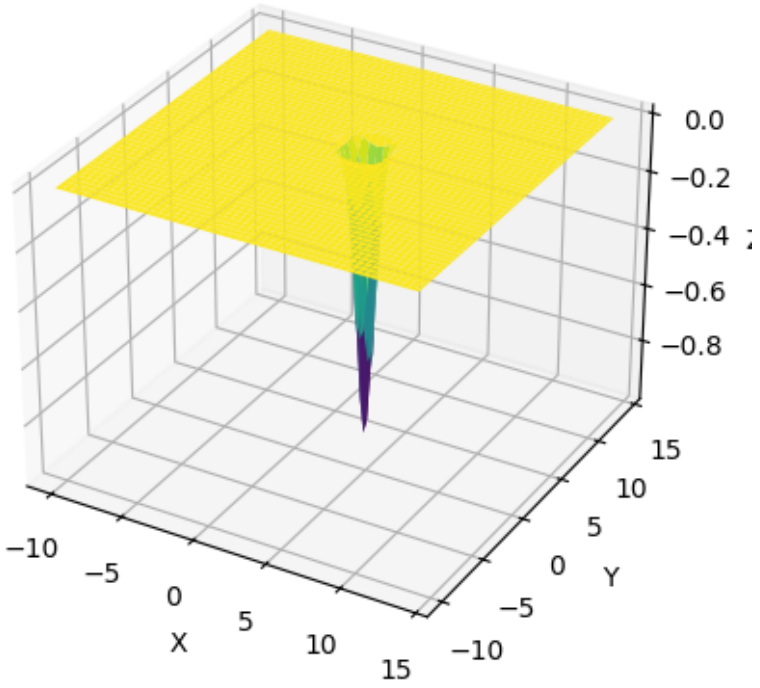
4.1. Easom funkcija

Easom funkcija, kuri aprašoma:

$$f(x,y) = -\cos(x)\cdot\cos(y)\cdot\exp\left(-(x-\pi)^2-(y-\pi)^2\right) \tag{5}$$

[SB13], yra dažnai naudojamas bandymų uždavinys optimizavimo algoritmams. Pavadinimas kilęs iš Michael E. Easom vardo, dėl savo unikalių charakteristikų ji yra naudojama įvertinant optimizavimo metodų efektyvumą ir tikslumą.

Funkcijos grafikas pavaizduotas 1 paveiksliuke rodo varpo formą su vieninteliu įtrūkiu taške (π,π), kur ji pasiekia savo globalų minimumą - 1, kitur jos reikšmė yra lygi 0. Nors formulė nėra sudėtinga, Easom funkcija kelia iššūkių optimizavimo algoritmams dėl jos siauro globalaus minimumo.



1 pav. *Easom funkcijos globalus minimumas*

Nors tiksli Easom funkcijos kilmė nėra plačiai aprašyta, ji sulaukė dėmesio optimizavimo tyrimuose dėl jos naudingumo ir paprastumo. Dažnai cituojama moksliniuose straipsniuose ir optimizavimo literatūroje, įskaitant tokius straipsnius kaip John H. Holland ”Adaptation in Natural and Artificial Systems” [Hol75] ir Michael E. Easom tyrimai dėl lanksčių gamybos sistemų modeliavimo [Eas83].

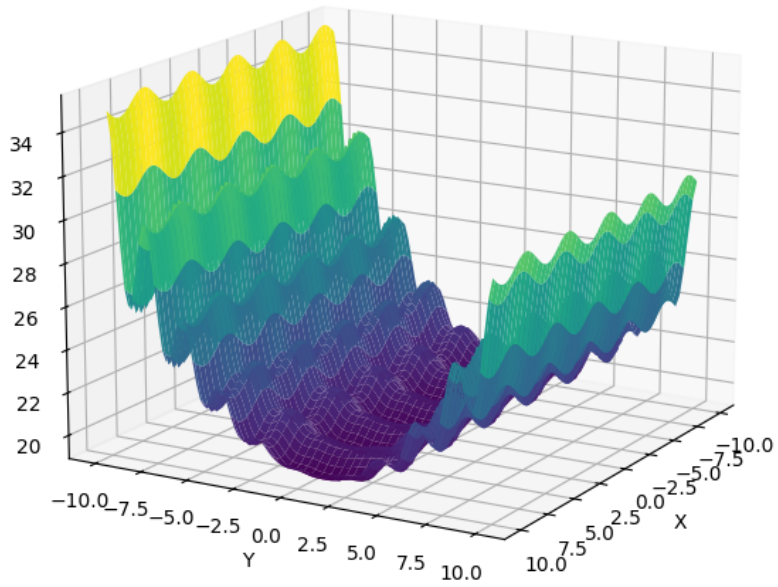
4.2. Levy Funkcija

Levy funkcija, įvesta Avriel Levy 1998 metais [Lev98], yra viena iš dažnai naudojamų funkcijų optimizavimo tyrimuose, ypač kai vertinami netiesiniai optimizavimo metodai. Ji pasižymi sudėtinga struktūra ir dažnai naudojama norint įvertinti algoritmų gebėjimą pasiekti optimalų sprendimą, neturint galimybės tiesiogiai pasikliauti paprastomis, tiesiomis paieškos erdvėmis. Funkcija turi vieną globalų minimumą, kuris pasiekiamas taške $x = (1, 1, \ldots, 1)$, ir šio minimumo reikšmė yra $f(1, 1, \ldots, 1) = 0$.

Levy funkcija yra išreiškiama lygtimi:

$$f(x) = \sin^2(\pi w_1) + \sum_{i=1}^{d-1}(w_i - 1)^2[1 + 10\ \sin^2(\pi w_i + 1)] + (w_d - 1)^2[1 + \sin^2(2\pi w_d)], \qquad (6)$$

kur $w_i = 1 + \frac{x_i - 1}{4}$, $\forall i = 1, \ldots, d$. Levy funkcijos formulė apima kelias sudėtingas sudėtines dalis, kurios sukuria daugybę lokalių minimumų, taip formuodamos sudėtingą paviršių pavaizduotą 2 paveiksliuke.



2 pav. *Levy funkcijos lokalūs optimalūs sprendiniai*

Levy funkcija dažnai naudojama testuojant įvairius optimizavimo algoritmus, nes jos savybės reikalauja itin rafinuotų metodų, kurie galėtų subalansuoti paieškos erdvės tyrinėjimo ir išnaudojimo aspektus. Tai ypač svarbu, kai reikia spręsti sudėtingas užduotis, kuriose paieškos erdvės dydis ir sudėtingumas gali smarkiai augti, didėjant dimensijų skaičiui. Tačiau taip pat svarbu pažymėti, kad, didėjant dimensijų skaičiui, šios funkcijos tyrimas tampa vis sudėtingesnis ir gali pareikalauti didelių skaičiavimo resursų, todėl ji nėra tinkama visiems optimizavimo metodams.

Nepaisant šių iššūkių, Levy funkcija lieka plačiai naudojama optimizavimo tyrimuose [Lev98]. Levy funkcija buvo naudojama eksperimentiniuose tyrimuose, tokiuose kaip Laguna ir Marti tyrimas apie išplėstų sklaidos paieškos schemų eksperimentinį išbandymą, siekiant globalaus optimizavimo multimodulinėms funkcijoms [LM02].

Levy funkcijos sudėtingos konvergavimo savybės gali kelti iššūkių optimizavimo algoritmams, dėl kurių gali kilti sunkumų pasiekiant tikslų konvergavimą. Be to, funkcijos sudėtinga struktūra gali netinkamai atspindėti realaus pasaulio optimizavimo uždavinių savybes, ribodama jos taikymą tam tikrose situacijose. Nepaisant to, ji lieka viena iš populiariausių ir naudingiausių funkcijų vertinant algoritmų gebėjimą spręsti sudėtingas netiesines optimizavimo užduotis.

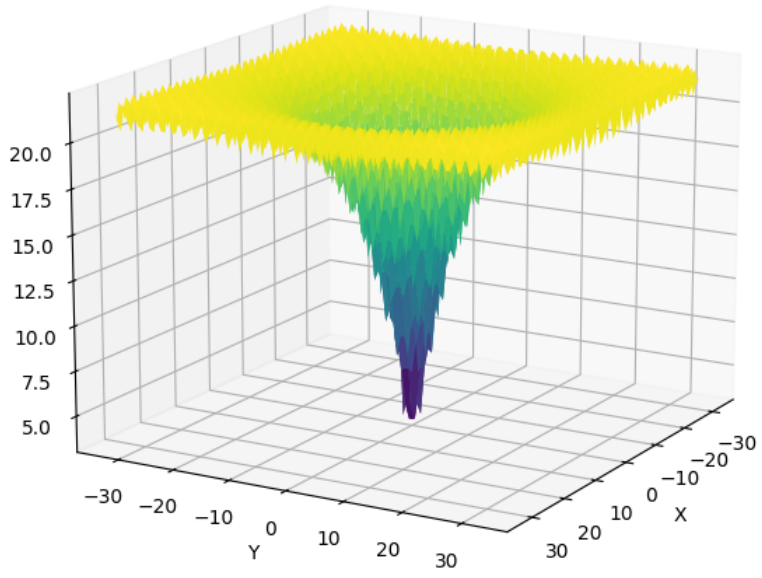
4.3. Ackley funkcija

Ackley funkcija yra plačiai naudojama optimizavimo algoritmų testavimui. Ji yra pavadinta David H. Ackley vardu ir dėl savo unikalaus charakterio yra itin naudinga įvertinti netiesinių optimizavimo metodų efektyvumą bei tikslumą. Jos formulė, išreiškiama lygtimi:

$$f(x) = -a \cdot \exp\left(-b \cdot \sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}\right) - \exp\left(\frac{1}{d}\sum_{i=1}^{d} \cos(c \cdot x_i)\right) + a + \exp(1), \tag{7}$$

kur d yra dimensijų skaičius, o a, b, ir c pasirenkami parametrai [Ado05]. Dažniausiai praktikoje rekomenduojama naudoti: $a = 20, b = 0.2, c = 2\pi$.

Svarbiausia Ackley funkcijos savybė – globalaus minimumo buvimas, apsuptas lokalių minimumų sričių, o tai leidžia efektyviai įvertinti skirtingų optimizavimo metodų gebėjimą pereiti nuo lokalių minimumų į globalų minimumą. Ackley funkcija yra dažnai minima bei pritaikoma optimizavimo literatūroje ir moksliniuose straipsniuose. Ypač plačiai ji aptariama optimizavimo ir evoliucinių algoritmų tyrimuose, kurie yra susiję su neuroniniais tinklais ir adaptacinėmis sistemomis ([Ack87]).



3 pav. *Ackley funkcijos lokalūs minimumai ir globalus minimumas*

Ackley funkcijos grafikas (3 pav.) pateikiamas dvimatėje erdvėje ir atskleidžia sudėtingą išgaubtą paviršių, kuris turi daugybę lokalių minimų ir vieną globalų minimumą, esantį taške $f(0,0) = 0$. Funkcijos konstrukcinės ypatybės, tokios kaip plokščia išorinė sritis ir gili centrinė duobė, yra svarbios,

nes jos sukuria sudėtingą struktūrą, kuri imituoja realias problemas. Šiuo sudėtingumu tikrinamas algoritmo gebėjimas išvengti vietinių minimumų ir efektyviai konverguoti prie globalaus minimumo. Be to, jo svyruojanti elgsena yra iššūkis optimizavimo strategijoms, kad būtų galima efektyviai subalansuoti tyrinėjimą ir išnaudojimą. Todėl tiriant, kaip skirtingi metodai veikia Ackley funkciją, galima gauti vertingų įžvalgų apie jų stipriąsias ir silpnąsias puses sprendžiant netiesinio optimizavimo uždavinius.

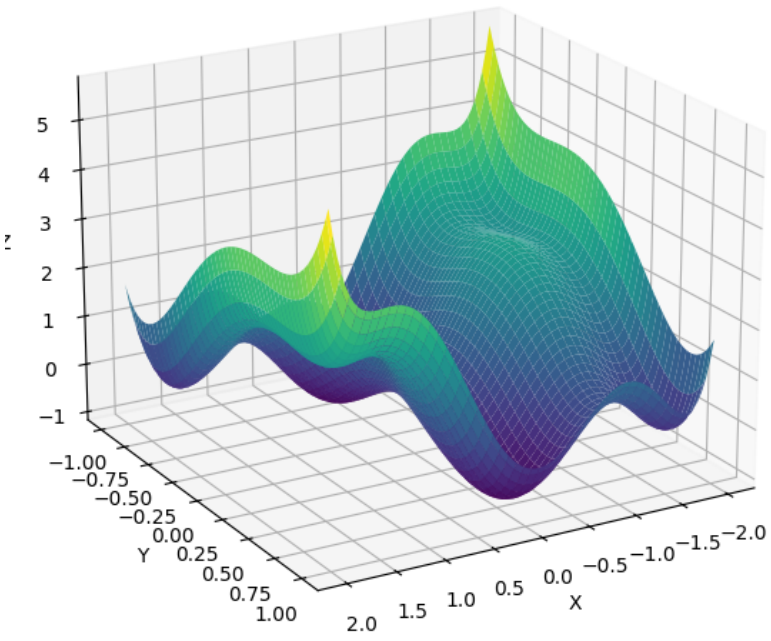
Nors Ackley funkcija yra plačiai naudojama, tačiau vienas iš šios funkcijos trūkumų - lėtas veikimas didėjant dimensijų skaičiui. Paieškos erdvė, nors dažniausiai yra dviejų dimensijų, gali būti ir keičiamo dydžio, tačiau didėjant dimensijų skaičiui tampa eksponentiškai sudėtingesnė, todėl tam tikriems algoritmams tai gali būti labai didelės skaičiavimo sąnaudos. Taip pat, nors ši funkcija yra geras daugialypės optimizacijos etalonas, tačiau juo nepatikrinamas algoritmo gebėjimas susidoroti su apribojimais, triukšmingais vertinimais ar kitais realaus pasaulio sudėtingumo aspektais.

4.4. Six-Hump Camel funkcija

Six-Hump Camel funkcija yra dar viena populiari funkcija naudojama optimizavimo algoritmų testavimui ir vertinimui. Ši funkcija dėl savo unikalios struktūros leidžia analizuoti optimizavimo algoritmų gebėjimus rasti globalų minimumą neatsižvelgiant į daugybę lokalių minimumų. Funkcijos matematinė formulė išreiškiama:

$$f(x) = \left(4 - 2.1x_1^2 + \frac{x_1^4}{3}\right) x_1^2 + x_1x_2 + \left(-4 + 4x_1^2\right) x_2^2, \tag{8}$$

kur x_1 ir x_2 yra nepriklausomi kintamieji, kurie dažniausiai apibrėžiami intervale: $x_1 \in [-3, 3]$ ir $x_2 \in [-2, 2]$. Funkcija turi du globalius minimumus, esančius taškuose $(-0.0898, 0.7126)$ ir $(0.0898, -0.7126)$, su minimaliąja verte $f(x_1, x_2) \approx -1.0316$.



4 pav. *Six-Hump Camel funkcijos globalūs minimumai*

Six-Hump Camel funkcija ypač naudinga dėl savo daugiamačių savybių, ši funkcija turi 6 lokalius

minimumus iš kurių 2 yra globalūs minimumai ([Mar05]). Šios funkcijos savybės padeda įvertinti, kaip skirtingi optimizavimo algoritmai išsprendžia uždavinius, kuriose reikia išvengti užstrigimo lokaliuose minimumuose.

4 paveiksliuke pavaizduotame grafike matyti sudėtinga dvimatė struktūra su išgaubimais, kurie lyginami su kupranugario nugaros formomis. Ši funkcija išsiskiria sudėtingais kontūrais, kurie sukuria problematišką veikimą optimizavimo algoritmams. Kaip ir kitos tokio tipo funkcijos, Six-Hump Camel funkcija turi savo apribojimų. Pavyzdžiui, nors ji efektyviai parodo algoritmų gebėjimą tvarkytis su daugiamačiais lokaliais minimumais, ji nėra pati tinkamiausia įvertinti algoritmų veikimą, kurie sprendžia uždavinius su apribojimais, duomenimis turinčiais triukšmo ar netiesiniais ryšiais. Be to, funkcija dažniausiai taikoma dvimatėje erdvėje, todėl gali būti mažiau tinkama testuoti algoritmus su dideliais dimensijų kiekiais.

4.5. Rosenbrock funkcija

Rosenbrock funkcija yra dar viena klasikinė, populiari ir unikali funkcija, kuri turi sudėtingą struktūrą ir yra plačiai naudojama optimizavimo algoritmų testavimui ir vertinimui. Funkcijos matematinė formulė yra:

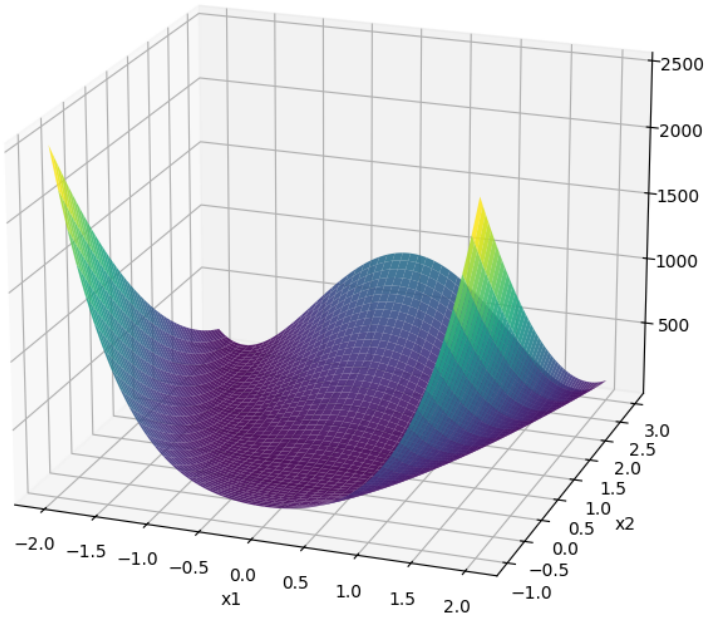
$$f(x) = \sum_{i=1}^{d-1} \left[100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2\right], \tag{9}$$

kur $d = 2$, o funkcija yra dvimatė ir apibrėžiama taip:

$$f(x_1, x_2) = (a - x_1)^2 + b(x_2 - x_1^2)^2, \tag{10}$$

kur $a = 1$ ir $b = 100$ yra standartiniai, pasirenkamieji parametrai. Šiuo atveju globalus minimumas yra taške $(x_1, x_2) = (1, 1)$, o minimali funkcijos reikšmė yra $f(x_1, x_2) = 0$.

5 paveiksliuke atvaizduota šios funkcijos dvimatė struktūra, kuriai būdingas siauras ir gilus slėnis, vedantis į globalų minimumą. Ši funkcija dažnai naudojama testuoti algoritmams, kurie turi išlaikyti pusiausvyrą tarp tyrinėjimo ir išnaudojimo, kad efektyviai rastų minimumą [Jia19].



5 pav. *Rosenbrock funkcijos globalus minimumas*

Rosenbrocko funkcijos globalus minimumas yra slėnio apačioje. Dėl mažo pateikiamos paieškos informacijos kiekio bendriesiems optimizavimo algoritmams sunku atskirti paieškos kryptį. Kartais funkcijai tampa sudėtinga rasti globalų minimumą ir lengva įstrigti lokaliame minimume vienoje slėnio pusėje. Galutinėje paieškos fazėje šie algoritmai negali iššokti iš šio optimizavimo proceso, kai optimizuoja Rosenbrocko funkciją. Šis algoritmas apjungia kelių strategijų kryžminimo operaciją ir savaime prisitaikančių dinamiškų algoritmo parametrų koregavimą. Tai leidžia algoritmui atsižvelgti tiek į globalią paiešką, tiek į lokalią paiešką. Naujas algoritmas pagerina konvergavimo greitį ir optimizavimo tikslumą Rosenbrocko funkcijos atžvilgiu.

Rosenbrock funkcija turi savo apribojimų, kurie gali pasunkinti šios funkcijos naudojimą ir pritaikymą. Ji yra tinkama vertinti algoritmus, skirtus netiesinėms optimizacijos problemoms, tačiau ji nėra idealus pasirinkimas, kai reikia įvertinti algoritmus, gebančius dirbti su triukšmingais duomenimis ar griežtais apribojimais. Be to, funkcijos sudėtingumas didėja su dimensijų skaičiumi, o tai gali lemti didesnes skaičiavimo sąnaudas, todėl funkcija dažnai taikoma žemesnės dimensijos uždaviniams.

5. Praktinė dalis

Šiame skyriuje pristatomi praktiniai rezultatai, gauti analizuojant Kolmogorov-Smirnov testą (2.4. skyrius), generuojant PI reikšmę (2.5. skyrius), naudojant optimizavimo algoritmus su skirtingais atsitiktinių skaičių generatoriais. Pagrindinis tikslas yra įvertinti, kaip įvairūs atsitiktinių skaičių generatoriai veikia optimizavimo algoritmų efektyvumą, konkrečiai taikant juos su Easom ir Levy bei kitomis funkcijomis. Kadangi Easom ir Six-Hump Camel funkcijos yra tik 2 dimensijų, visi skaičiavimai buvo atlikti tik 2 dimensijų aplinkoje, kad rezultatai galėtų būti palyginami tarpusavyje. Išanalizavus visas funkcijas dvimatėje erdvėje, skaičiavimai buvo atlikti ir su didesnės dimensijos (5) erdve, norint patikrinti, ar išvados iš dvimatės erdvės galioja ir didesnės dimensijos erdvėje, tačiau šie skaičiavimai galioja tik Levy, Ackley ir Rosenbrock funkcijoms.

5.1. Programavimo aplinkos aprašymas

Darbas buvo atliktas naudojant Python programavimo kalbą ir integruotą kūrimo aplinką Visual Studio Code. Darbas buvo vykdomas kompiuteryje su šiomis charakteristikomis:

1. Windows 11 Pro (v 23H2) x64.
2. Intel procesorius i7-10510U.
3. RAM 24 GB.

Analizėje buvo naudotos darbo autoriaus aprašytos testavimo funkcijos, atsitiktiniai skaičių generatoriai, optimizacijos algoritmai . Taip pat buvo pritaikytos šios Python bibliotekos:

1. Numpy – naudojama darbui su daugiamačiais masyvais.
2. Pandas – skirta duomenų apdorojimui ir analizei.
3. math – naudojama matematinės funkcijoms apibrėžti.

Panaudojus šias bibliotekas ir pritaikius jų funkcijas, buvo atliktas tiriamasis darbas. Darbe buvo analizuojamas vienas optimizacijos algoritmas, 5 testavimo funkcijos ir 3 atsitiktiniai generatoriai.

5.2. Kolmogorov-Smirnov testo pritaikymas

Šiame tyrime visi atsitiktinių skaičių generatoriai ir jų pasiskirstymo funkcijos lyginamos su tolygiojo skirstinio pasiskirstymo funkcija (angl. *Uniform distribution function*). Kiekvienas generatorius sugeneravo $1 \cdot 10^5$ atsitiktinių skaičių intervale $(0; 1)$, tuomet buvo paskaičiuota empirinė pasiskirstymo funkcija, kuri toliau buvo lyginama Kolmogorov-Smirnov teste. Tai buvo pakartota 1000 kartų. Tikrintos dvi reikšmės: p-reikšmė su 0.05 ir KS statistinė reikšmė, kuri lyginama su kritine reikšme, kuri lygi 0.0043.

Eksperimentų metu buvo lyginami trys skirtingi atsitiktinių skaičių generatoriai: LCG, Mersenne Twister ir PCG, siekiant nustatyti, kuris iš jų geriau modeliuoja tolygųjį skirstinį. Pritaikant LCG buvo naudojami parametrai: $a = 5$, $c = 1$, $m = 2^{32}$.

1 lentelė. *Kolmogorov - Smirnov testo rezultatai*

Generatorius	**Neatmestų H_0 (%)**	**Vidutinė p-reikšmė**	**Vidutinė KS statistinė reikšmė**
LCG	89 %	0.4415	0.0029
MT	97 %	0.5151	0.0027
PCG	94 %	0.5099	0.0027

Naudojant tris skirtingus generatorius patikrinta, ar skaičiai atitinka tolygųjį skirstinį ir yra atsitiktiniai..**LCG** rezultatai pateikti 1 lentelėje rodo, kad atliekant $1 \cdot 10^5$ skaičių generavimą gauta vidutinė KS statistika yra 0.0029, o vidutinė p-reikšmė siekė 0.4415. Atsižvelgiant į gautus rezultatus, tik 89% sugeneruotų sekų negalime atmesti nulinės hipotezės, todėl galime sakyti, kad sugeneruoti skaičiai yra atsitiktiniai ir pasiskirstę pagal tolygųjį skirstinį.

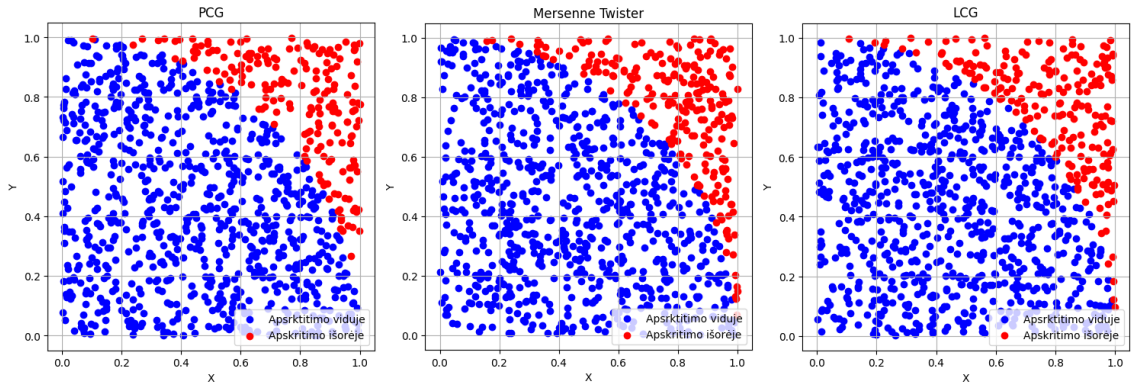
Mersenne Twister vidutinė KS statistika - 0.0027, o vidutinė p-reikšmė - 0.5151. Gauta KS statistika yra šiek tiek mažesnė nei naudojant LCG, todėl galime manyti, kad Mersenne Twister generatorius generuoja labiau atsitiktinius skaičius nei LCG. Atsižvelgiant į KS statistiką ir p-reikšmę, taip pat neatmetame nulinės hipotezės ir galime sakyti, kad sugeneruoti skaičiai yra atsitiktiniai ir atitinka tolygųjį skirstinį 97% kartų iš visų generavimo bandymų.

PCG vidutinė KS statistika - 0.0027, o vidutinė p-reikšmė - 0.5099. Šio generatoriaus generuojamos sekos atitiko prielaidas 94% kartų ir tiek kartų nebuvo atmesta hipotezė, kad skaičiai yra atsitiktiniai ir pasiskirstę pagal tolygųjį skirstinį.

Apibendrinant, nors visi trys atsitiktinių skaičių generatoriai veikia efektyviai generuojant tolygiai pasiskirsčiusius atsitiktinius skaičius, Mersenne Twister demonstruoja geriausią atitikimą tolygiajam skirstiniui, po jo seka PCG, o tada LCG. Tai rodo, kad Mersenne Twister generuoti skaičiai yra labiausiai atitinkantys tolygųjį skirstinį, tuo tarpu PCG ir LCG, nors ir gerai veikia, šiek tiek dažniau generuoja skaičius neatitinkančius tolygiojo skirstinio.

5.3. PI skaičiaus generavimas

Atsižvelgiant į 2.5. skyriuje aprašytą skaičiaus PI generavimą, atlikti skaičiavimai naudojant skirtingus generatorius. Gauti sugeneruotų taškų rezultatai pavaizduoti 6 paveiksliuke. Mersenne Twister generatorius, atlikus skaičiavimus, pasižymėjo stabiliu ir tiksliu PI reikšmės įvertinimu, kurio vidutinė paklaida siekė 0.04175, o vidutinis kodo sukimosi laikas buvo 24.05 sekundės. PCG taip pat pateikė gerus rezultatus su vidutine paklaida 0.04172, kuri buvo tik šiek tiek didesnė nei tiesinio kongruentinio generatoriaus, tačiau mažesnė nei Mersenne Twister. Vis dėlto, šio generatoriaus kodo veikimo laikas buvo ženkliai ilgesnis – 44 sekundės, kas leidžia teigti, jog PCG yra patikimas ir tikslus, tačiau lėtesnis pasirinkimas. LCG parodė mažiausią vidutinę paklaidą, siekiančią 0.04142, tačiau jo rezultatai buvo labiausiai kintantys. Nors vidutinė paklaida buvo mažiausia, kai kuriose simuliacijose klaidos reikšmės buvo didesnės, o tai gali rodyti mažesnį šio generatoriaus stabilumą. Vidutinis kodo veikimo laikas buvo 36 sekundės.



6 pav. *Atsitiktinių taškų generavimo rezultatai naudojant PCG, Mersenne Twister ir LCG*

Nedideli vidutinių paklaidų skirtumai tarp Mersenne Twister, PCG ir LCG rodo, kad jų tikslumas apskaičiuojant PI yra beveik identiškas, todėl bet kurį iš šių trijų generatorių galima naudoti Monte Karlo simuliacijoms. Tačiau efektyvumo požiūriu, Mersenne Twister yra žymiai greitesnis nei PCG ir LCG, todėl jis yra tinkamesnis pasirinkimas, kai veikimo laikas yra svarbus veiksnys, ypač didelio masto

simuliacijose ar programose, reikalaujančiose didelio atsitiktinių mėginių skaičiaus per trumpą laiką. Dėl savo greičio ir aukšto tikslumo Mersenne Twister yra geriausias pasirinkimas laiko atžvilgiu jautrioms programoms. Tuo tarpu PCG ir LCG, nors ir užtikrina panašų tikslumą, dėl ilgesnio vykdymo laiko yra mažiau efektyvūs, palyginti su Mersenne Twister. Vis dėlto jie abu gali būti tinkami specifiniuose kontekstuose, kur jų savybės yra pageidautinos arba kai vykdymo greitis nėra esminis veiksnys.

Šie rezultatai rodo, kad atsitiktinių skaičių generatoriaus pasirinkimas gali turėti įtakos skaičiavimų tikslumui, ypač kai kalbama apie jautrias skaičiavimo užduotis, tokias kaip PI reikšmės įvertinimas. Mersenne Twister yra patikimiausias atsitiktinių skaičių generatorius dėl savo stabilumo.

5.4. Genetinio algoritmo įgyvendinimas

Šiame tyrime įgyvendintas genetinis algoritmas (GA), siekiantis optimizuoti tikslines funkcijas, kaip – Easom, Levy, Ackley ir kt. GA pagrįstas evoliuciniais principais, tokiais kaip atranka, kryžminimas ir mutacija, siekiant iteratyviai tobulinti pradinę populiaciją. Toliau pateikiama implementuoto genetinio algoritmo apžvalga.

Algoritmas prasideda sukuriant pradinę sprendimų (individų) populiaciją, kur kiekvienas individas yra reprezentuojamas chromosoma. Chromosomos yra dviejų (gali būti ir daugiau) realių skaičių masyvai, atitinkantys funkcijų įvestis. Naudojami skirtingi atsitiktinių skaičių generatoriai, norint įvertinti jų įtaką optimizavimo algoritmui: PCG, LCG ir Mersenne Twister.

Kiekvieno individo chromosomos įvertinimas atliekamas naudojant algoritmui duotą funkciją. Funkcijos rezultatas grąžina tinkamumo vertę, kur artimesnės reikšmės globaliam minimumui rodo geresnius sprendimus, kadangi GA tikslas yra minimizuoti funkcijos rezultatą. Tinkamumo funkcija padeda nustatyti tuos individo sprendimus, kurie bus pasirinkti reprodukcijai, suteikiant pirmenybę geresniems sprendimams.

Atranka atliekama naudojant elitizmą ir turnyro atranką. Geriausi 10% generacijos, arba dar kitaip vadinamas - elitas, tiesiogiai pereina į kitą generaciją, kad išsaugotų geriausius sprendimus. Likę 90% patiria turnyro atranką: atsitiktinai iš likusiųjų parenkama grupė individų, iš kurios pasirenkamas geriausias, pagal algoritmo funkcijos rezultatą. Šis procesas padidina tikimybę atrinkti geresnius individų sprendimus ir tuo pačiu palaiko įvairovę, užtikrinant, kad algoritmas neįstrigtų lokaliuose minimumuose.

Kryžminimas vyksta tarp atrinktų tėvų (individų), kurie buvo atrinkti turnyro metu, kad sukurtų naujas atžalas. Tėvų chromosomos sujungiamos, kad suformuotų vaikų chromosomas. Kiekvienam genui (dimensijai) yra 50% tikimybė paveldėti vieno iš tėvų geną, kas įneša įvairovės į atžalas. Šis rekombinacijos mechanizmas leidžia perduoti gerus bruožus kitai generacijai ir padeda atrasti geresnius sprendimus laikui bėgant.

Kiekviena atžala pereina mutacijos etapą, kur atsitiktinai pasirinktam genui pridedama maža atsitiktinai pasirinkta reikšmė (mutacijos stiprumas). Mutacija įneša papildomą įvairovę, užtikrinant, kad algoritmas neįstrigtų lokaliuose minimumuose. Algoritmas taip pat užtikrina, kad mutavę genai liktų ribose, kurios leistinos pagal duotos funkcijos parametrus, kad būtų išvengta verčių, viršijančių ribas.

Norint išvengti stagnacijos, kai generacijos geriausias sprendinys nesikeičia per kelias iteracijas, įdiegtas atstatymo mechanizmas. Jei per 20 generacijų nepasiekiamas joks patobulinimas, dalis generacijos yra atsitiktinai vėl sugeneruojama. Jei stagnacija tęsiasi ir pasiekiamas atstatymo limitas (30 atstatymų), algoritmas sustabdomas, ir fiksuojamas geriausias rastas sprendimas. Šis mechanizmas padeda palaikyti įvairovę ir užtikrinti, kad algoritmas nesutriktų ir neviršytų per ilgo vykdymo laiko, net jei problema yra sudėtinga.

Algoritmas tęsiasi tol, kol pasiekiamas optimizuotas sprendimas (tinkamumo riba), arba pasiekiamas atstatymo limitas. Kelios metrikos yra matuojamos siekiant analizuoti algoritmo veikimą:

- **Sprendinių skaičius:** sėkmingai surastų sprendinių skaičius, su kuriais pasiektas globalus minimumas su duota minimalia paklaida $1 \cdot 10^{-8}$.

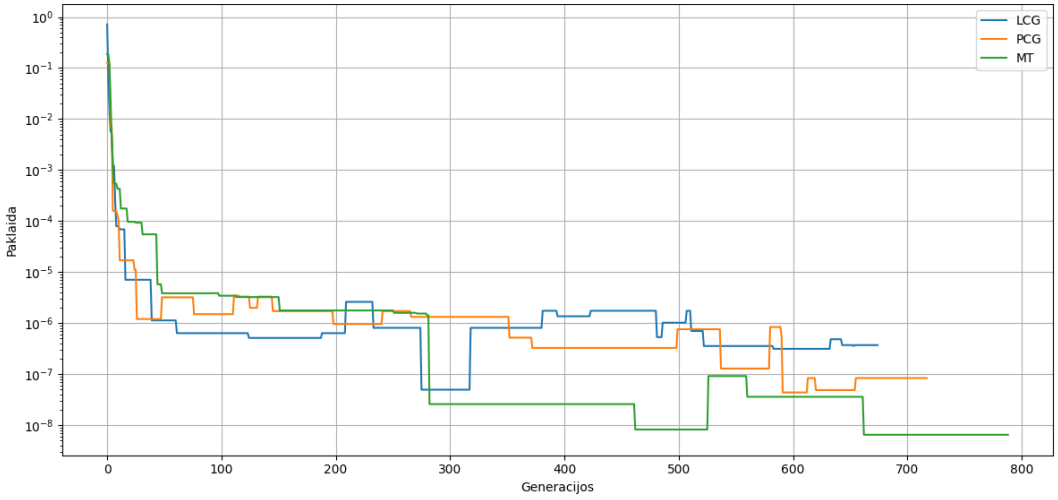
- **Vidutinis generacijų skaičius sprendimui rasti:** vidutiniškai kurioje generacijoje buvo pasiektas globalus minimumas - rastas geriausias sprendinys.
- **Vidutinė sprendimo paklaida nuo globalaus minimumo:** vidutinė sėkmingai rasto sprendimo reikšmė.
- **Vidutinis generacijų skaičius, kai buvo pasiektas atstatymo limitas:** vidutinis generacijų skaičius po kurių buvo pasiektas atsistatymo limitas.
- **Vidutinė geriausia sprendimo reikšmė, kai buvo pasiektas atstatymo limitas:** vidutinė geriausia sprendimo reikšmė, kai buvo pasiektas atstatymo limitas.

Šis genetinis algoritmas buvo implementuotas taip, kad būtų efektyvus optimizuojant tiek Easom, tiek Levy ar kitas funkcijas, kurios pasižymi siaurais ir sudėtingais optimalių sprendimų paieškos intervalais. Algoritmas sugeba pasiekti gerus rezultatus net ir sudėtingose problemose, naudodamas tinkamą evoliucijos mechanizmų derinį, derinant atranką, kryžminimą, mutaciją ir stagnacijos valdymą.

Toliau yra pateikti rezultatai, naudojant anksčiau aprašytą genetinį algoritmą, naudojant 3 skirtingus ir šiame tyrime aprašytus atsitiktinių skaičių generatorius: LCG (2.1. skyrius), PCG (2.2. skyrius) ir Mersenne Twister (2.3. skyrius). Kiekvienas iš šių skaičių generatorių buvo panaudotas, siekiant rasti geriausią sprendinį 4. skyriuje aprašytoms funkcijoms.

Atsitiktinių skaičių generatoriai buvo naudojami įvairiuose optimizavimo algoritmo etapuose, įskaitant pradinės generacijos generavimą, palikuonių kūrimą paveldint genus iš tėvų, naujai sudarytų palikuonių mutaciją, turnyro atrankos metodo taikymą ir naujų individų generavimą stagnacijos metu. Kiekvienoje generacijoje buvo 100 individų, kurie buvo sudaryti iš dviejų genų. PCG ir Mersenne Twister generatorių parametrai buvo nekeičiami ir naudojami numatytieji, o LCG parametrai buvo modifikuoti, kaip ir 5.2. skyriuje aprašyto Kalmogorov-Smirnov testo metu: $a = 5, c = 1, m = 2^{32}$.

Grafikas pavaizduotas 7 paveiksliuke parodo vieno atsitiktinio pakartojimo geriausio sprendimo evoliuciją kiekvienoje iš generacijų. 3 skirtingos linijos rodo skirtingus atsitiktinių skaičių generatorius, integruotus į genetinį algoritmą sprendžiant Easom funkciją. Principas, kaip genetinis algoritmas sprendžia kiekvieną funkciją, išlieka toks pat ir grafikai yra panašūs.



7 pav. *Genetinio algoritmo geriausio sprendimo paieška Easom funkcijai*

Gauti rezultatai pateikti žemiau esančiose lentelėse kartu su kiekvienos funkcijos ir atsitiktinių skaičių generatoriaus pora.

Gauti rezultatai sprendžiant Levy testavimo funkciją pateikti 2 lentelėje.

2 lentelė. *Rezultatai sprendžiant Levy funkciją naudojant genetinį algoritmą*

Stebima metrika	**LCG**	**MT**	**PCG**
Sėkmingų sprendimų skaičius	692	741	740
Vidutinis generacijų skaičius sprendimui rasti	332	337	329
Vidutinė sprendimo paklaida nuo globalaus minimumo	$5.15 \cdot 10^{-9}$	$5.39 \cdot 10^{-9}$	$5.09 \cdot 10^{-9}$
Vidutinis generacijų skaičius kai buvo pasiektas atstatymo limitas	726	724	723
Vidutinė geriausia sprendimo reikšmė, kai buvo pasiektas atstatymo limitas	$2.38 \cdot 10^{-8}$	$2.41 \cdot 10^{-8}$	$2.23 \cdot 10^{-8}$

Atitinkamai genetinis algoritmas buvo testuotas ir su kita optimizacijos algoritmų testavimo funkcija - Easom. 3 lentelėje pateikti rezultatai su anksčiau minėtais atsitiktiniais skaičių generatoriais.

3 lentelė. *Rezultatai sprendžiant Easom funkciją naudojant genetinį algoritmą*

Stebima metrika	**LCG**	**MT**	**PCG**
Sėkmingų sprendimų skaičius	183	193	216
Vidutinis generacijų skaičius sprendimui rasti	424	419	406
Vidutinė sprendimo paklaida nuo globalaus minimumo	$5.28 \cdot 10^{-9}$	$5.17 \cdot 10^{-9}$	$5.17 \cdot 10^{-9}$
Vidutinis generacijų skaičius kai buvo pasiektas atstatymo limitas	718	721	719
Vidutinė geriausia sprendimo reikšmė, kai buvo pasiektas atstatymo limitas	$8.33 \cdot 10^{-8}$	$6.68 \cdot 10^{-8}$	$6.91 \cdot 10^{-8}$

Genetinis algoritmas buvo pritaikytas ir Ackley funkcijai, su kuria buvo gauti 4 lentelėje pateikti rezultatai:

4 lentelė. *Rezultatai sprendžiant Ackley funkciją naudojant genetinį algoritmą*

Stebima metrika	**LCG**	**MT**	**PCG**
Sėkmingų sprendimų skaičius	0	0	0
Vidutinis generacijų skaičius sprendimui rasti	-	-	-
Vidutinė sprendimo paklaida nuo globalaus minimumo	-	-	-
Vidutinis generacijų skaičius kai buvo pasiektas atstatymo limitas	720	718	720
Vidutinė geriausia sprendimo reikšmė, kai buvo pasiektas atstatymo limitas	$5.1 \cdot 10^{-4}$	$4.49 \cdot 10^{-4}$	$4.71 \cdot 10^{-4}$

Sprendžiant Ackley funkciją globalus minimumas nebuvo rastas su nustatyta paklaida, tačiau iš rezultatų 4 lentelėje galima matyti, kad Mersenne Twister generatorius ir PCG buvo vidutiniškai arčiau globalaus minimumo nei LCG. Atsistatymo limito rodiklis su šia funkcija taip pat nenusako, kokią įtaką daro atsitiktinių skaičių generatorius, nes su visais rodiklis yra labai panašus. Funkcija buvo ištestuota padidinant paklaidą, su kuria ieškomas sprendimas ir padidinamas stagnacijos limitas. Be to, buvo pridėtas dar vienas papildomas žingsnis, kuris kas 100 generacijų prie gautų chromosomų prideda atsitiktinį skaičių iš intervalo $[1 \cdot 10^{-8}, 1 \cdot 10^{-6}]$. Naujai gauti rezultatai atvaizduoti 5 lentelėje.

5 lentelė. *Rezultatai sprendžiant Ackley funkciją naudojant modifikuotą genetinį algoritmą*

Stebima metrika	**LCG**	**MT**	**PCG**
Sėkmingų sprendimų skaičius	0	36	33
Vidutinis generacijų skaičius sprendimui rasti	-	1651	1594
Vidutinė sprendimo paklaida nuo globalaus minimumo	-	$6.73 \cdot 10^{-7}$	$7.11 \cdot 10^{-7}$
Vidutinis generacijų skaičius kai buvo pasiektas atstatymo limitas	2001	2001	2001
Vidutinė geriausia sprendimo reikšmė, kai buvo pasiektas atstatymo limitas	$3.81 \cdot 10^{-1}$	$4.78 \cdot 10^{-1}$	$4.88 \cdot 10^{-1}$

Dar viena funkcija, kuri buvo implementuota, kad būtų patikrinta atsitiktinių skaičių įtaka optimizacijos algoritmams, yra Rosenbrock. 6 lentelėje surašyti gauti rezultatai.

6 lentelė. *Rezultatai sprendžiant Rosenbrock funkciją naudojant genetinį algoritmą*

Stebima metrika	**LCG**	**MT**	**PCG**
Sėkmingų sprendimų skaičius	2	9	3
Vidutinis generacijų skaičius sprendimui rasti	498	371	369
Vidutinė sprendimo paklaida nuo globalaus minimumo	$7.23 \cdot 10^{-9}$	$4.29 \cdot 10^{-9}$	$6.88 \cdot 10^{-9}$
Vidutinis generacijų skaičius kai buvo pasiektas atstatymo limitas	725	724	726
Vidutinė geriausia sprendimo reikšmė, kai buvo pasiektas atstatymo limitas	$1.43 \cdot 10^{-5}$	$1.46 \cdot 10^{-5}$	$1.44 \cdot 10^{-5}$

Paskutinė funkcija, kuri buvo įgyvendinta tyrime - Six-Hump Camel. Rezultatai gauti su šia funkcija ir visais atsitiktinių skaičių generatoriais yra aprašyti 7 lentelėje:

7 lentelė. *Rezultatai sprendžiant Six-Hump Camel funkciją naudojant genetinį algoritmą*

Stebima metrika	**LCG**	**MT**	**PCG**
Sėkmingų sprendimų skaičius	33	41	39
Vidutinis generacijų skaičius sprendimui rasti	366	374	368
Vidutinė sprendimo paklaida nuo globalaus minimumo	$4.83 \cdot 10^{-9}$	$4.45 \cdot 10^{-9}$	$5.27 \cdot 10^{-9}$
Vidutinis generacijų skaičius kai buvo pasiektas atstatymo limitas	709	708	709
Vidutinė geriausia sprendimo reikšmė, kai buvo pasiektas atstatymo limitas	$3 \cdot 10^{-7}$	$2.99 \cdot 10^{-7}$	$2.97 \cdot 10^{-7}$

Genetinio algoritmo rezultatai parodė, kad MT ir PCG pralenkė LCG su visomis testavimo funkcijomis. Nors ir su MT generatoriumi genetinis algoritmas gavo geriausius rezultatus su sudėtingomis funkcijomis, tokiomis kaip Rosenbrock arba Ackley, tačiau PCG sugebėjo rasti sprendimą per mažesnį generacijų skaičių. Nors ir buvo pasirinktos skirtingos funkcijos su skirtingomis savybėmis, bendras PCG ir MT pranašumas yra akivaizdus. Genetinis algoritmas su visais generatoriais, nors ir elgėsi panašiai ties stagnacijos ar atsistatymo riba, pažangesni generatoriai leido algoritmui konverguoti daugiau kartų link globalaus minimumo. Rezultatai parodė, kad tinkamo atsitiktinių skaičių generatoriaus pasirinkimas gali pagerinti algoritmo našumą ir stabilumą.

Toliau atliekami skaičiavimai buvo daromi penkiamatėje erdvėje. Parametrų modifikacijos buvo atliekamos pagal sprendžiamą funkciją, kad būtų pasiekti norimi rezultatai. Pirmiausia yra sprendžiama Levy funkcija su padidintu tikslumu iki $1 \cdot 10^{-7}$. Pirminiame kodo leidime stagnacijos limitas paliekamas kaip buvo ir dvimatėje erdvėje. Atlikus pakeitimus ir naudojant aukštesnės dimensijos Levy funkciją rezultatai aprašyti 8 lentelėje.

8 lentelė. *Rezultatai sprendžiant penkiamatę Levy funkciją naudojant genetinį algoritmą*

Stebima metrika	**LCG**	**MT**	**PCG**
Sėkmingų sprendimų skaičius	329	479	464
Vidutinis generacijų skaičius sprendimui rasti	598	568	575
Vidutinė sprendimo paklaida nuo globalaus minimumo	$7.4 \cdot 10^{-8}$	$7.37 \cdot 10^{-8}$	$7.52 \cdot 10^{-8}$
Vidutinis generacijų skaičius kai buvo pasiektas atstatymo limitas	823	822	826
Vidutinė geriausia sprendimo reikšmė, kai buvo pasiektas atstatymo limitas	$2.22 \cdot 10^{-7}$	$1.96 \cdot 10^{-7}$	$1.89 \cdot 10^{-7}$

Sprendžiant penkiamatę Ackley funkciją paklaida, su kuria priimamas sprendimas, buvo padidinta iki $1 \cdot 10^{-5}$. Tokios korekcijos buvo atliktos, nes algoritmas nesugebėjo konverguoti į globalų minimumą dvimatėje erdvėje. Atlikus modifikacijas Ackley funkcijos rezultatai aprašyti 9 lentelėje.

9 lentelė. *Rezultatai sprendžiant penkiamatę Ackley funkciją naudojant genetinį algoritmą*

Stebima metrika	**LCG**	**MT**	**PCG**
Sėkmingų sprendimų skaičius	0	28	19
Vidutinis generacijų skaičius sprendimui rasti	-	4690	4705
Vidutinė sprendimo paklaida nuo globalaus minimumo	-	$7.79 \cdot 10^{-6}$	$7.83 \cdot 10^{-6}$
Vidutinis generacijų skaičius kai buvo pasiektas atstatymo limitas	5001	5001	5001
Vidutinė geriausia sprendimo reikšmė, kai buvo pasiektas atstatymo limitas	1.38	0.82	0.71

Pirminiai rezultatai su Rosenbrock funkcija penkiamatėje erdvėje su $1 \cdot 10^{-7}$ tikslumu nedavė jokių rezultatų. Algoritmas stagnuodavo nepasiekęs norimo tikslumo. Padidinus tikslumą iki pat $1 \cdot 10^{-4}$, atsistatymo limitus iki 50, o stagnacijos limitą iki 70, algoritmas sugebėjo rasti keletą sprendinių kurie pateikti 10 lentelėje.

10 lentelė. *Rezultatai sprendžiant penkiamatę Rosenbrock funkciją naudojant genetinį algoritmą*

Stebima metrika	**LCG**	**MT**	**PCG**
Sėkmingų sprendimų skaičius	0	5	2
Vidutinis generacijų skaičius sprendimui rasti	-	1442	1226
Vidutinė sprendimo paklaida nuo globalaus minimumo	-	$8.32 \cdot 10^{-5}$	$6.83 \cdot 10^{-5}$
Vidutinis generacijų skaičius kai buvo pasiektas atstatymo limitas	4789	8906	9102
Vidutinė geriausia sprendimo reikšmė, kai buvo pasiektas atstatymo limitas	$8.83 \cdot 10^{-1}$	$2.27 \cdot 10^{-1}$	$2.25 \cdot 10^{-1}$

Panašiai kaip ir dviejų dimensijų atveju, penkių dimensijų genetinio algoritmo rezultatai parodė aiškų MT ir PCG pranašumą prieš LCG. MT generatorius rado daugiausia sėkmingų sprendimų funkcijose ir dažnu atveju turėjo mažesnę sprendimų paklaidą. Nors ir sudėtingesnėmis sąlygomis, pažangesni generatoriai, tokie kaip MT ir PCG, leido algoritmui dažniau rasti sėkmingą sprendimą nei LCG. Šie rezultatai dar kartą pabrėžia, kaip tinkamas atsitiktinių skaičių generatoriaus pasirinkimas gali turėti reikšmingą įtaką genetinio algoritmo rezultatams ne tik dvimatėje, bet ir aukštesnėje erdvėje.

5.5. Paprastosios atsitiktinės paieškos algoritmo įgyvendinimas

Šiame magistrantūros darbe taip pat buvo įgyvendintas ir paprastosios atsitiktinės paieškos optimizavimo algoritmas. Šis algoritmas yra pats paprasčiausias ir lengviausiai įgyvendinamas. Algoritmo veikimo metu yra generuojami atsitiktiniai skaičiai, su kuriais yra tikrinama funkcijos reikšmė. Algoritmo pagalba buvo sprendžiami 3 skirtingi ir šiame tyrime aprašyti atsitiktinių skaičių generatoriai: LCG (2.1. skyrius), PCG (2.2. skyrius) ir Mersenne Twister (2.3. skyrius). Kiekvienas iš šių skaičių generatorių buvo panaudotas, siekiant rasti geriausią sprendinį 4. skyriuje aprašytoms funkcijoms. Naudojant paprastąjį atsitiktinį optimizavimo algoritmą buvo stebimos kelios metrikos, norint suprasti, kaip gerai ir tiksliai optimizavimo algoritmas sugeba surasti funkcijos sprendinį. Buvo pasirinkta stebėti vidutinę paklaidą, vidutinį iteracijų skaičių ir kitas metrikas. Buvo generuojama 1000000 skaičių porų (dvimatėje erdvėje reikia 2 skaičių, norint įvertinti funkciją), o sprendimas yra priimamas, jei sprendimo paklaida nuo globalaus minimumo yra ne didesnė nei $1 \cdot 10^{-5}$. Rezultatai buvo analizuojami šių metrikų pagalba:

1. **Vidutinė sprendimo paklaida**: paklaidos vidurkis, gautas visuose bandymuose su konkretaus atsitiktinių skaičių generatoriaus ir funkcijos deriniu. Apskaičiuojamas imant skirtumą tarp optimizacijos algoritmo reikšmės ir globalaus minimumo.

2. **Vidutinis iteracijų skaičius sprendimui rasti**: vidutinis iteracijų skaičius, per kurį buvo rastas sprendimas.

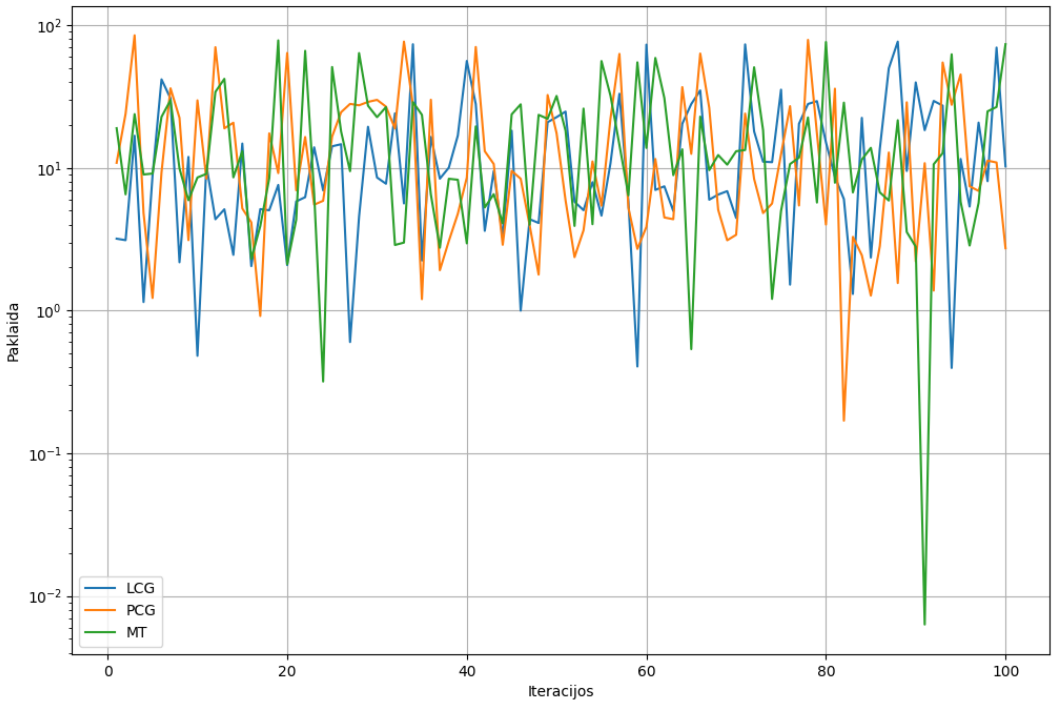
3. **Sėkmingai surasti sprendiniai**: sėkmingai rastų sprendinių skaičius po visų skaičių generavimo.

4. **Vidutinė sprendinio paklaida kai nebuvo rastas sprendimas**: vidutinė nesėkmingų sprendimų paklaida nuo globalaus minimumo.

5. **Mažiausia rasta paklaida (nesėkmingų sprendimų)**: geriausias sprendimas arba mažiausia paklaida, kuri buvo rasta po visų iteracijų.

PCG ir Mersenne Twister generatorių parametrai buvo nekeičiami ir naudojami numatytieji, o LCG parametrai buvo modifikuoti, kaip ir Kalmogorov-Smirnov testo metu: $a = 5, c = 1, m = 2^{32}$.

8 grafikas parodo, kaip paprastosios atsitiktinės paieškos algoritmas generuojant naujus skaičius netobulėja ir sprendimas neartėja link globalaus minimumo. 3 skirtingos linijos rodo skirtingus atsitiktinių skaičių generatorius, integruotus į algoritmą, sprendžiant Levy funkciją. Principas, kaip algoritmas sprendžia kitas funkcijas, išlieka toks pat ir grafikai yra labai panašūs.



8 pav. *Paprastosios atsitiktinės paieškos atsitiktinis sprendimas*

Rezultatai sprendžiant Levy funkciją aprašyti 11 lentelėje.

11 lentelė. *Rezultatai sprendžiant Levy funkciją naudojant paprastosios atsitiktinės paieškos algoritmą*

Stebima metrika	**LCG**	**MT**	**PCG**
Sėkmingų sprendimų skaičius	0	3	1
Vidutinis iteracijų skaičius sprendimui rasti	-	682423	804861
Vidutinė sprendimo paklaida	-	$9.18 \cdot 10^{-7}$	$6.06 \cdot 10^{-6}$
Vidutinė sprendimo paklaida, kai nebuvo rastas sprendinys	17	17.01	17.01
Mažiausia rasta paklaida (nesėkmingų sprendimų)	$3.59 \cdot 10^{-1}$	$3.34 \cdot 10^{-5}$	$9.68 \cdot 10^{-5}$

Atitinkamai keičiant funkciją į Easom, buvo gauti rezultatai, kurie pateikti 12 lentelėje.

12 lentelė. *Rezultatai sprendžiant Easom funkciją naudojant paprastosios atsitiktinės paieškos algoritmą*

Stebima metrika	**LCG**	**MT**	**PCG**
Sėkmingų sprendimų skaičius	0	1	0
Vidutinis iteracijų skaičius sprendimui rasti	-	898174	-
Vidutinė sprendimo paklaida	-	$2.07 \cdot 10^{-6}$	-
Vidutinė sprendimo paklaida, kai nebuvo rastas sprendinys	$9.97 \cdot 10^{-1}$	$9.96 \cdot 10^{-1}$	$9.97 \cdot 10^{-1}$
Mažiausia rasta paklaida (nesėkmingų sprendimų)	$7.17 \cdot 10^{-1}$	$2.43 \cdot 10^{-5}$	$1.71 \cdot 10^{-4}$

Rezultatai gauti su Ackley funkcija pateikti 13 lentelėje.

13 lentelė. *Rezultatai sprendžiant Ackley funkciją naudojant paprastosios atsitiktinės paieškos algoritmą*

Stebima metrika	**LCG**	**MT**	**PCG**
Sėkmingų sprendimų skaičius	0	0	0
Vidutinis iteracijų skaičius sprendimui rasti	-	-	-
Vidutinė sprendimo paklaida	-	-	-
Vidutinė sprendimo paklaida, kai nebuvo rastas sprendinys	9.68	9.7	9.69
Mažiausia rasta paklaida (nesėkmingų sprendimų)	$1.5 \cdot 10^{-4}$	$7.63 \cdot 10^{-3}$	$5.41 \cdot 10^{-3}$

Implementuotos Rosenbrock funkcijos į paprastosios atsitiktinės paieškos algoritmą gauti rezultatai aprašyti 14 lentelėje.

14 lentelė. *Rezultatai sprendžiant Rosenbrock funkciją naudojant paprastosios atsitiktinės paieškos algoritmą*

Stebima metrika	**LCG**	**MT**	**PCG**
Sėkmingų sprendimų skaičius	0	1	0
Vidutinis iteracijų skaičius sprendimui rasti	-	852613	-
Vidutinė sprendimo paklaida	-	$9.47 \cdot 10^{-6}$	-
Vidutinė sprendimo paklaida, kai nebuvo rastas sprendinys	123773	127781	127476
Mažiausia rasta paklaida (nesėkmingų sprendimų)	3.44	$1.73 \cdot 10^{-3}$	$2.22 \cdot 10^{-3}$

Ir galiausiai gauti rezultatai su Six-Hump Camel funkcija matomi 15 lentelėje.

15 lentelė. *Rezultatai sprendžiant Six-Hump Camel funkciją naudojant paprastosios atsitiktinės paieškos algoritmą*

Stebima metrika	**LCG**	**MT**	**PCG**
Sėkmingų sprendimų skaičius	0	2	1
Vidutinis iteracijų skaičius sprendimui rasti	-	920109	940681
Vidutinė sprendimo paklaida	-	$4.39 \cdot 10^{-6}$	$5.01 \cdot 10^{-6}$
Vidutinė sprendimo paklaida, kai nebuvo rastas sprendinys	21.59	21.21	21.2
Mažiausia rasta paklaida (nesėkmingų sprendimų)	$3.32 \cdot 10^{-1}$	$5.33 \cdot 10^{-4}$	$9.44 \cdot 10^{-5}$

Paprastosios atsitiktinės paieškos algoritmas šiame magistrantūros darbe buvo implementuotas norint parodyti, kad skirtingi rezultatai yra gaunami, tiesiog remiantis atsitiktinių skaičių generatorių savybėmis be papildomų modifikacijų ar patobulinimų. Iš rezultatų galime aiškiai matyti, kad nors ir sėkmingai surastų sprendimų buvo vienetai, tačiau galime pastebėti, kad LCG nesugebėjo rasti nei vieno sprendinio, o rastas sprendinys su mažiausia paklaida neprilygsta nei PCG, nei MT. Lyginant MT ir PCG - tiek sėkmingų sprendimų skaičiumi, tiek tikslumu, MT generatorius pasirodė efektyviau, pasiekdamas daugiau sėkmingų sprendimų ir turėdamas mažesnę paklaidą tarp geriausio nesėkmingo sprendinio. Net ir paprastosios atsitiktinės paieškos algoritmo realizacijoje matosi ryškūs skirtumai tarp generatorių ir algoritmo gautų rezultatų. Tai parodo, kad generatoriaus pasirinkimas gali nulemti optimizacijos algoritmo efektyvumą.

5.6. Dalelių spiečiaus optimizavimo algoritmas

Šiame magistrantūros darbe taip pat buvo įgyvendintas dalelių spiečiaus optimizavimo (PSO) algoritmas. Kaip su anksčiau įgyvendintais algoritmais, taip ir šiuo atveju buvo naudojamos Easom, Levy ir kitos testavimo funkcijos. Toliau pateikta naudojamo algoritmo apžvalga.

Dalelių spiečiaus optimizavimo algoritmas prasideda nustatant parametrus ir sukuriant dalelių spiečių (populiaciją) erdvėje, kurią apibrėžia vartotojo nurodytos ribos, kurios yra koreguojamos priklausomai nuo sprendžiamos funkcijos. Šios dalelės reprezentuoja galimus sprendimus. Kiekviena jų turi poziciją, greitį ir geriausią iki šiol pasiektą poziciją. Algoritmo tikslas – surasti geriausią poziciją (sprendimą), minimizuojant pasirinktą funkciją, kuri yra naudojama vertinant kiekvieno sprendimo kokybę.

Pirmiausia, kiekviena dalelė gauna pradinę atsitiktinę poziciją ir greitį, o tada apskaičiuojamas jų tinkamumas pagal duotą funkciją. Funkcijos rezultatas rodo, kiek sprendimas artimas globaliam minimumui, o mažesnės reikšmės signalizuoja geresnį rezultatą.

Kiekviename algoritmo cikle (iteracijoje) kiekvienos dalelės greitis atnaujinamas, atsižvelgiant į geriausią jos pačios ir visos populiacijos poziciją. Greičio atnaujinimas remiasi trijų komponentų suma:

- **Inercijos komponentas**: palaiko ankstesnį dalelės greitį.
- **Kognityvinis komponentas**: skatina dalelę grįžti į savo geriausią poziciją.
- **Socialinis komponentas**: traukia dalelę link spiečiaus geriausio sprendimo.

Šie trys veiksniai padeda dalelėms balansuoti tarp paieškos ploto tyrinėjimo ir susitelkimo į jau surastus gerus sprendimus. Dalelių greitis ribojamas, kad būtų išvengta per didelio greičio, kuris galėtų sukelti chaotišką elgesį, taip išlaikant stabilumą ir tikslumą.

Kiekviena dalelė taip pat atnaujina savo poziciją pagal savo greitį ir patikrina, ar nepasiekė ribų. Jeigu pozicija yra ribose, ji laikoma galiojančia ir dalelės istorijoje registruojama ši nauja pozicija. Po šių veiksmų vėl skaičiuojamas kiekvienos dalelės tinkamumas pagal naudojamą funkciją ir jeigu šis rezultatas geresnis už ankstesnį, pozicija atnaujinama kaip nauja geriausia.

PSO algoritmas taip pat turi mechanizmą, skirtą užtikrinti, kad algoritmas neįgytų reikšmių iš lokalių minimumų ir neperžengtų tam tikro vykdymo laiko. Kiekvienoje iteracijoje skaičiuojama, ar buvo pasiektas pagerėjimas. Jei keliose iš eilės iteracijose nėra jokių pagerėjimų, algoritmas baigiasi ir fiksuojama geriausia pasiekta vertė. Taip algoritmas sugeba išlaikyti sprendimų įvairovę ir išvengti per ilgo vykdymo, jei problema yra sudėtinga.

Algoritmas tęsiasi tol, kol pasiekiamas optimizuotas sprendimas (tinkamumo riba), arba po nustatyto iteracijų skaičiaus algoritmas nebegali rasti geresnio sprendimo arba pasiekiamas leistinas iteracijų skaičius. Kelios metrikos yra matuojamos siekiant analizuoti algoritmo veikimą:

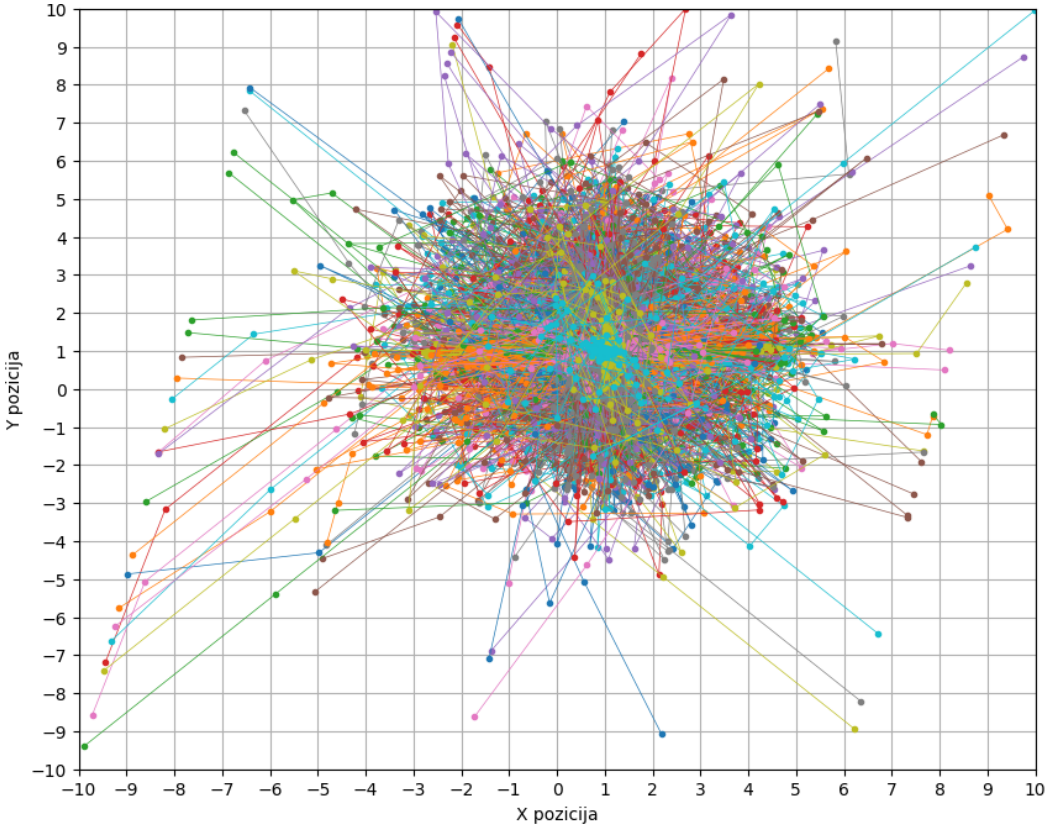
- **Sėkmingų sprendimų skaičius:** sėkmingai surastų sprendinių skaičius su kuriais pasiektas globalus minimumas su duota minimalia paklaida $1 \cdot 10^{-8}$.
- **Vidutinis iteracijų skaičius sprendimui rasti:** vidutiniškai kurioje iteracijoje buvo pasiektas globalus minimumas - rastas geriausias sprendinys.
- **Vidutinė sprendimo paklaida:** vidutinė sėkmingai rasto sprendimo paklaida.
- **Vidutinis iteracijų skaičius kai buvo pasiektas atstatymo limitas:** vidutinis iteracijų skaičius po kurių buvo pasiektas atsistatymo limitas.
- **Vidutinė geriausia sprendimo paklaida, kai algoritmas sustojo dėl stagnacijos:** vidutinė geriausia sprendimo reikšmė, kai algoritmas sustojo,nes nebegalėjo rasti geresnio sprendimo.

Toliau yra pateikti rezultatai, naudojant anksčiau aprašytą dalelių spiečiaus algoritmą, naudojant 3 skirtingus ir šiame tyrime aprašytus atsitiktinių skaičių generatorius: LCG (2.1. skyrius), PCG (2.2. skyrius) ir Mersenne Twister (2.3. skyrius). Kiekvienas iš šių skaičių generatorių buvo panaudotas, siekiant rasti geriausią sprendinį 4. skyriuje aprašytoms funkcijoms.

Su kiekvienu skaičių generatoriaus ir optimizacijos algoritmo testavimo funkcijos pora buvo atlikta 1000 pakartojimų, kurių metu buvo stebimos anksčiau aprašytos metrikos.

Kiekvienoje iteracijoje buvo 100 dalelių. PCG ir Mersenne Twister generatorių parametrai buvo nekeičiami ir naudojami numatytieji, o LCG parametrai buvo modifikuoti, kaip ir Kalmogorov-Smirnov testo metu: $a = 5, c = 1, m = 2^{32}$.

9 grafikas parodo PSO algoritmo vieno atsitiktinio pakartojimo sprendimą, kaip dalelės, pradėjusios atsitiktinėje vietoje, randa globalų minimumą. Grafike aiškiai matosi, kad algoritmas konverguoja į tam tikrą tašką, tai yra visi taškai pradeda judėti link globalaus minimumo (pavaizduotu atveju tai yra 0 su koordinatėmis $(1, 1)$). Principas, kaip dalelių spiečiaus algoritmas sprendžia kiekvieną funkciją, išlieka toks pat ir grafikai yra panašūs.



9 pav. *Dalelių spiečiaus algoritmo atsitiktinis sprendimas*

Rezultatai gauti sprendžiant Levy testavimo funkciją aprašyti 16 lentelėje.

16 lentelė. *Rezultatai sprendžiant Levy funkciją naudojant dalelių spiečiaus algoritmą*

Stebima metrika	**LCG**	**MT**	**PCG**
Sėkmingų sprendimų skaičius	851	872	862
Vidutinis iteracijų skaičius sprendimui rasti	135	133	134
Vidutinė sprendimo paklaida	$5 \cdot 10^{-9}$	$4.5 \cdot 10^{-9}$	$4.78 \cdot 10^{-9}$
Vidutinis iteracijų skaičius kai buvo pasiekta stagnacija	94	92	91
Vidutinė geriausia sprendimo paklaida, kai buvo pasiekta stagnacija	$4.8 \cdot 10^{-6}$	$6.17 \cdot 10^{-6}$	$5.06 \cdot 10^{-6}$

Rezultatai gauti su implementuota Easom funkcija matomi 17 lentelėje.

17 lentelė. *Rezultatai sprendžiant Easom funkciją naudojant dalelių spiečiaus algoritmą*

Stebima metrika	**LCG**	**MT**	**PCG**
Sėkmingų sprendimų skaičius	828	855	847
Vidutinis iteracijų skaičius sprendimui rasti	156	151	153
Vidutinė sprendimo paklaida	$4.89 \cdot 10^{-9}$	$4.45 \cdot 10^{-9}$	$4.71 \cdot 10^{-9}$
Vidutinis iteracijų skaičius kai buvo pasiekta stagnacija	98	101	99
Vidutinė geriausia sprendimo paklaida, kai buvo pasiekta stagnacija	$2.85 \cdot 10^{-5}$	$3.17 \cdot 10^{-5}$	$3.33 \cdot 10^{-5}$

Kaip ir su kitais algoritmais, buvo implementuotos ir Ackley, Rosenbrock ir Six-Hump camel funkcijos. Šių funkcijų rezultatai atitinkamai yra aprašyti 18, 19 ir 20 lentelėse.

18 lentelė. *Rezultatai sprendžiant Ackley funkciją naudojant dalelių spiečiaus algoritmą*

Stebima metrika	**LCG**	**MT**	**PCG**
Sėkmingų sprendimų skaičius	721	844	804
Vidutinis iteracijų skaičius sprendimui rasti	313	307	308
Vidutinė sprendimo paklaida	$7.41 \cdot 10^{-9}$	$6.74 \cdot 10^{-9}$	$6.89 \cdot 10^{-9}$
Vidutinis iteracijų skaičius kai buvo pasiekta stagnacija	111	114	122
Vidutinė geriausia sprendimo paklaida, kai buvo pasiekta stagnacija	$2.17 \cdot 10^{-2}$	$2.52 \cdot 10^{-2}$	$1.95 \cdot 10^{-2}$

19 lentelė. *Rezultatai sprendžiant Rosenbrock funkciją naudojant dalelių spiečiaus algoritmą*

Stebima metrika	**LCG**	**MT**	**PCG**
Sėkmingų sprendimų skaičius	497	523	517
Vidutinis iteracijų skaičius sprendimui rasti	262	260	259
Vidutinė sprendimo paklaida	$5.41 \cdot 10^{-9}$	$5.01 \cdot 10^{-9}$	$5.13 \cdot 10^{-9}$
Vidutinis iteracijų skaičius kai buvo pasiekta stagnacija	120	122	120
Vidutinė geriausia sprendimo paklaida, kai buvo pasiekta stagnacija	$4.31 \cdot 10^{-4}$	$3.86 \cdot 10^{-4}$	$4.3 \cdot 10^{-4}$

20 lentelė. *Rezultatai sprendžiant Six-Hump Camel funkciją naudojant dalelių spiečiaus algoritmą*

Stebima metrika	**LCG**	**MT**	**PCG**
Sėkmingų sprendimų skaičius	0	2	1
Vidutinis iteracijų skaičius sprendimui rasti	-	70	54
Vidutinė sprendimo paklaida	-	$4.12 \cdot 10^{-9}$	$8.31 \cdot 10^{-9}$
Vidutinis iteracijų skaičius kai buvo pasiekta stagnacija	308	305	309
Vidutinė geriausia sprendimo paklaida, kai buvo pasiekta stagnacija	$1.21 \cdot 10^{-5}$	$1.23 \cdot 10^{-5}$	$1.23 \cdot 10^{-5}$

Dalelių spiečiaus optimizacijos (PSO) algoritmo rezultatai, analizuojant įvairias testavimo funkcijas ir atsitiktinių skaičių generatorius, parodė šio algoritmo patikimumą bei jautrumą pasirinktų generatorių kokybei. MT generatorius dažniausiai pasiekė didesnį sėkmingų sprendimų skaičių, ypač sprendžiant sudėtingesnes funkcijas, tokias kaip Ackley ar Rosenbrock. LCG, nors ir veiksmingas, paprastai nusileido tiek sėkmingų sprendimų skaičiaus rodikliais, tiek tikslumu, pabrėžiant jo ribotumus, lyginant su pažangesniais generatoriais. Analizuojant algoritmo konvergavimą, pastebėta, kad MT generatorius ir PCG dažnai pasiekdavo sprendimus su mažesniu iteracijų skaičiumi, išlaikant aukštą tikslumą, o tai įrodo jų efektyvumą. Stagnacijos metu PCG neretai pasižymėjo mažesne vidutine sprendimo paklaida, pabrėžiant jo gebėjimą išlaikyti kokybiškus rezultatus net sudėtingomis sąlygomis. Šie rezultatai rodo, kad PSO algoritmas, naudojant pažangesnius generatorius: MT ar PCG, yra itin veiksmingas sprendžiant sudėtingas optimizavimo užduotis.

Kaip ir su genetiniu optimizacijos algoritmu, dalelių spiečiaus optimizacijos algoritmas buvo skaičiuojamas ir aukštesnėje dimensijoje - 5. Sprendžiant Levy funkciją paklaida, su kuria priimamas sprendimas, nekeičiama.

Levy funkcijos penkiamatėje erdvėje gauti rezultatai aprašyti 21 lentelėje.

21 lentelė. *Rezultatai sprendžiant penkiamatę Levy funkciją naudojant dalelių spiečiaus algoritmą*

Stebima metrika	**LCG**	**MT**	**PCG**
Sėkmingų sprendimų skaičius	981	989	984
Vidutinis iteracijų skaičius sprendimui rasti	228	228	229
Vidutinė sprendimo paklaida	$7.18 \cdot 10^{-9}$	$7.07 \cdot 10^{-9}$	$7.26 \cdot 10^{-9}$
Vidutinis iteracijų skaičius kai buvo pasiekta stagnacija	105	106	102
Vidutinė geriausia sprendimo paklaida, kai buvo pasiekta stagnacija	$6.42 \cdot 10^{-4}$	$7.62 \cdot 10^{-4}$	$1.31 \cdot 10^{-3}$

Padidinus erdvės dimensiją Ackley funkcijos algoritmas nebuvo modifikuotas, o gauti rezultatai su šia funkcija yra surašyti 22 lentelėje.

22 lentelė. *Rezultatai sprendžiant penkiamatę Ackley funkciją naudojant dalelių spiečiaus algoritmą*

Stebima metrika	**LCG**	**MT**	**PCG**
Sėkmingų sprendimų skaičius	786	832	803
Vidutinis iteracijų skaičius sprendimui rasti	308	308	309
Vidutinė sprendimo paklaida	$6.72 \cdot 10^{-9}$	$6.74 \cdot 10^{-9}$	$6.8 \cdot 10^{-9}$
Vidutinis iteracijų skaičius kai buvo pasiekta stagnacija	117	122	116
Vidutinė geriausia sprendimo paklaida, kai buvo pasiekta stagnacija	$2.18 \cdot 10^{-2}$	$2.51 \cdot 10^{-2}$	$2.27 \cdot 10^{-2}$

Galiausiai buvo sprendžiama Rosenbrock funkcija. Padidinus tikslumą, su kuriuo priimamas sprendinys, iki $1 \cdot 10^{-5}$, o gauti rezultatai pateikti 23 lentelėje.

23 lentelė. *Rezultatai sprendžiant penkiamatę Rosenbrock funkciją naudojant dalelių spiečiaus algoritmą*

Stebima metrika	**LCG**	**MT**	**PCG**
Sėkmingų sprendimų skaičius	5	8	7
Vidutinis iteracijų skaičius sprendimui rasti	778	682	450
Vidutinė sprendimo paklaida	$9.94 \cdot 10^{-6}$	$9.95 \cdot 10^{-6}$	$9.43 \cdot 10^{-6}$
Vidutinis iteracijų skaičius kai buvo pasiekta stagnacija	937	957	930
Vidutinė geriausia sprendimo paklaida, kai buvo pasiekta stagnacija	$2{,}27 \cdot 10^{-1}$	$1{,}81 \cdot 10^{-1}$	$2{,}75 \cdot 10^{-1}$

Dalelių spiečiaus optimizacijos (PSO) algoritmo rezultatai penkių dimensijų atveju, naudojant skirtingus atsitiktinių skaičių generatorius (LCG, MT ir PCG), parodė, kad MT ir PCG generavo geresnius rezultatus. Abu šie generatoriai pasiekė didesnį sėkmingų sprendimų skaičių visose funkcijose, tačiau MT buvo šiek tiek geresnis pagal vidutinę paklaidą ir iteracijų skaičių, reikalingą sprendimui rasti. LCG, kaip ir dvimatėje erdvėje, rodė prastesnius rezultatus su visomis funkcijomis. Nors LCG generavo panašius rezultatus sprendžiant Levy funkciją, kitose funkcijose jis atsiliko tiek pagal sėkmingų sprendimų skaičių, tiek pagal vidutinę paklaidą. Tai rodo, kad MT ir PCG yra aiškiai efektyvesni ir patikimesni sprendžiant net ir aukštesnės dimensijos funkcijas.

5.7. Modeliuojamo atkaitinimo algoritmas

Paskutinis optimizacijos algoritmas, kuris buvo įgyvendintas su visomis 4. skyriuje minėtomis funkcijomis - modeliuojamo atkaitinimo (SA) algoritmas. Šis algoritmas yra įkvėptas fizinio medžiagų atkaitinimo proceso, kuriame medžiaga lėtai aušinama, siekiant pasiekti minimalios energijos būseną.

Kaip ir daugelyje kitų algoritmų, SA prasideda atsitiktinai sugeneruojant pirminį sprendinį arba, dar kitaip sakant, atskaitos tašką leistinų reikšmių intervale. Taškas atspindi galimą funkcijos sprendimą, kurio kokybė yra įvertinama pagal analizuojamos funkcijos rezultatą.

Pagrindiniai kriterijai modeliuojamo kaitinimo algoritme yra temperatūra ir aušinimo greitis. Temperatūra naudojama padedant algoritmui nuspręsti, ar gali būti priimtas blogesnis sprendimas nei prieš tai buvęs. Taip pat aukšta temperatūra leidžia algoritmui laisviau tyrinėti reikšmes leistinuose intervaluose. Iteruojant, temperatūra yra mažinama su kiekviena iteracija pagal nustatytą aušinimo greitį, o tai padeda algoritmui nebepriimti blogesnių sprendinių (tikimybė, kad bus priimtas blogesnis sprendinys yra mažinama) ir užtikrinti konvergavimą į globalų minimumą.

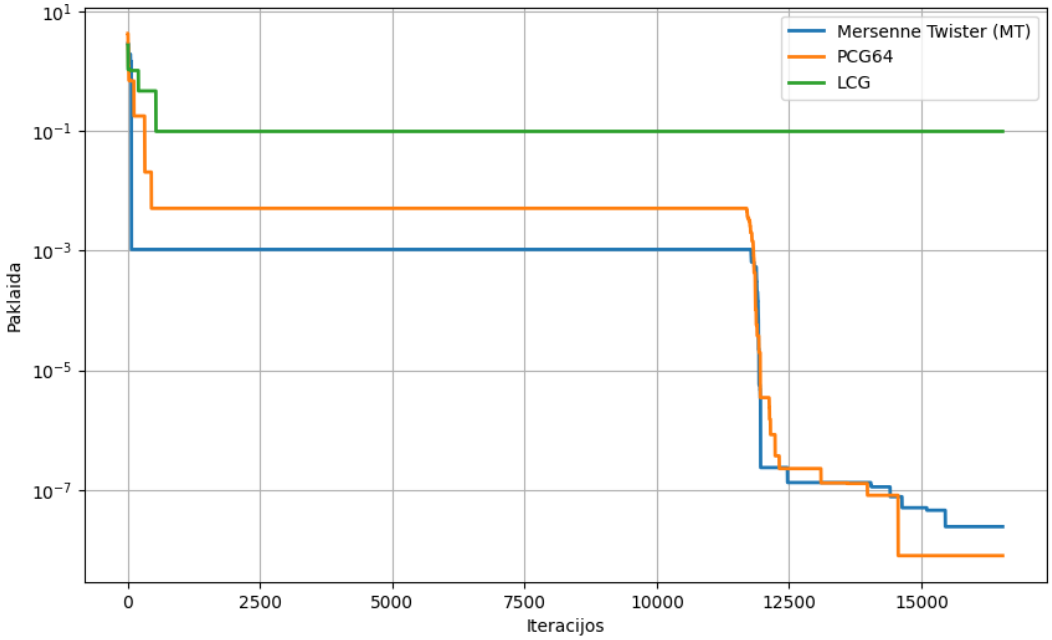
Kiekvienoje iteracijoje gaunamas naujas funkcijos sprendinys, atlikus nedidelį pakeitimą dabartiniame sprendinyje - tai vadinama perturbacija (angl. *perturbation*). Jei naujas sprendinys yra geresnis, jis automatiškai yra priimamas ir toliau traktuojamas kaip dabartinis sprendinys. Atsitinka taip, kad naujas sprendinys yra blogesnis, tokiu atveju jis irgi gali būti priimtas su tam tikra tikimybe, kuri apskaičiuojama pagal temperatūrą ir sprendinių skirtumą. Ši tikimybė tuomet yra lyginama su atsitiktinai sugeneruotu skaičiumi, jei tikimybė didesnė - sprendinys priimamas. Taip algoritmas neužsilieka lokaliuose minimumuose, o toliau tyrinėja paieškos erdvę.

Algoritmui toliau generuojant naujus sprendinius, jei po nustatyto skaičiaus iteracijų nepavyksta rasti geresnio sprendimo, procesas sustabdomas, išvengiant bereikalingo laiko eikvojimo. Naudojant anksčiau aprašytą modeliuojamo atkaitinimo algoritmą ir naudojant 3 skirtingus šiame tyrime aprašytus atsitiktinių skaičių generatorius: LCG (2.1. skyrius), PCG (2.2. skyrius) ir Mersenne Twister (2.3. skyrius). Kiekvienas iš šių skaičių generatorių buvo panaudotas, siekiant rasti geriausią sprendinį 4. skyriuje aprašytoms funkcijoms.

Algoritmo veikimas galiausiai yra įvertinamas pagal kelias pasirinktas metrikas:

- **Sprendinių skaičius:** sėkmingai surastų sprendinių skaičius, su kuriais pasiektas globalus minimumas su duota minimalia paklaida $1 \cdot 10^{-8}$.
- **Vidutinis iteracijų skaičius sprendimui rasti:** vidutiniškai kurioje iteracijoje buvo pasiektas globalus minimumas - rastas geriausias sprendinys.
- **Vidutinė sprendimo paklaida nuo globalaus minimumo:** vidutinė sėkmingai rasto sprendimo reikšmė.
- **Vidutinis iteracijų skaičius, kai buvo pasiekta stagnacija:** vidutinis iteracijų skaičius po kurių buvo pasiektas atsistatymo limitas.
- **Vidutinė geriausia sprendimo paklaida, kai buvo pasiekta stagnacija:** vidutinė geriausia sprendimo paklaida, kai buvo pasiektas atstatymo limitas.

Vieno atsitiktinio pakartojimo geriausio sprendimo evoliucija yra pavaizduota 10 grafike. 3 skirtingos linijos rodo skirtingus atsitiktinių skaičių generatorius, integruotus į modeliuojamo atkaitinimo algoritmą, sprendžiant Levy funkciją. Principas, kaip algoritmas sprendžia kitas funkcijas, išlieka toks pat ir grafikai yra panašūs.



10 pav. *Modeliuojamo atkaitinimo algoritmo atsitiktinis sprendimas*

Gauti rezultatai sprendžiant Levy funkciją aprašyti 24 lentelėje.

24 lentelė. *Rezultatai sprendžiant Levy funkciją naudojant modeliuojamo atkaitinimo algoritmą*

Stebima metrika	**LCG**	**MT**	**PCG**
Sėkmingų sprendimų skaičius	8	55	38
Vidutinis iteracijų skaičius sprendimui rasti	6688	8565	8514
Vidutinė sprendimo paklaida	$5.21 \cdot 10^{-9}$	$4.46 \cdot 10^{-9}$	$5.4 \cdot 10^{-9}$
Vidutinis iteracijų skaičius kai buvo pasiekta stagnacija	5951	5666	5717
Vidutinė geriausia sprendimo paklaida, kai buvo pasiekta stagnacija	$4.24 \cdot 10^{-2}$	$4.76 \cdot 10^{-2}$	$5.01 \cdot 10^{-2}$

Easom funkcijos rezultatai matomi 25 lentelėje.

25 lentelė. *Rezultatai sprendžiant Easom funkciją naudojant modeliuojamo atkaitinimo algoritmą*

Stebima metrika	**LCG**	**MT**	**PCG**
Sėkmingų sprendimų skaičius	4	11	6
Vidutinis iteracijų skaičius sprendimui rasti	9087	8693	8621
Vidutinė sprendimo paklaida	$5.22 \cdot 10^{-9}$	$5.11 \cdot 10^{-9}$	$5.32 \cdot 10^{-9}$
Vidutinis iteracijų skaičius kai buvo pasiekta stagnacija	5665	5655	5663
Vidutinė geriausia sprendimo paklaida, kai buvo pasiekta stagnacija	$2.56 \cdot 10^{-1}$	$2.64 \cdot 10^{-1}$	$2.72 \cdot 10^{-1}$

Ackley, Rosenbrock ir Six-Hump camel funkcijų rezultatai su modeliuojamo atkaitinimo algoritmu aprašyti 26, 27, 28 lentelėse atitinkamai.

26 lentelė. *Rezultatai sprendžiant Ackley funkciją naudojant modeliuojamo atkaitinimo algoritmą*

Stebima metrika	**LCG**	**MT**	**PCG**
Sėkmingų sprendimų skaičius	6	11	10
Vidutinis iteracijų skaičius sprendimui rasti	4796	4905	4838
Vidutinė sprendimo paklaida	$7.89 \cdot 10^{-9}$	$7.22 \cdot 10^{-9}$	$7.67 \cdot 10^{-9}$
Vidutinis iteracijų skaičius kai buvo pasiekta stagnacija	5513	5513	5513
Vidutinė geriausia sprendimo paklaida, kai buvo pasiekta stagnacija	2.29	2.01	2.03

27 lentelė. *Rezultatai sprendžiant Rosenbrock funkciją naudojant modeliuojamo atkaitinimo algoritmą*

Stebima metrika	**LCG**	**MT**	**PCG**
Sėkmingų sprendimų skaičius	1	13	7
Vidutinis iteracijų skaičius sprendimui rasti	8622	10015	10422
Vidutinė sprendimo paklaida	$3.91 \cdot 10^{-9}$	$4.71 \cdot 10^{-9}$	$5.32 \cdot 10^{-9}$
Vidutinis iteracijų skaičius kai buvo pasiekta stagnacija	9122	8369	8427
Vidutinė geriausia sprendimo paklaida, kai buvo pasiekta stagnacija	$1{,}07 \cdot 10^{-1}$	$1{,}69 \cdot 10^{-1}$	$1{,}75 \cdot 10^{-1}$

28 lentelė. *Rezultatai sprendžiant Six-Hump Camel funkciją naudojant modeliuojamo atkaitinimo algoritmą*

Stebima metrika	**LCG**	**MT**	**PCG**
Sėkmingų sprendimų skaičius	1	3	2
Vidutinis iteracijų skaičius sprendimui rasti	6330	6423	6745
Vidutinė sprendimo paklaida	$3.22 \cdot 10^{-9}$	$1.42 \cdot 10^{-9}$	$4.88 \cdot 10^{-9}$
Vidutinis iteracijų skaičius kai buvo pasiekta stagnacija	8969	8700	8902
Vidutinė geriausia sprendimo paklaida, kai buvo pasiekta stagnacija	$7.8 \cdot 10^{-3}$	$1.01 \cdot 10^{-2}$	$9.67 \cdot 10^{-3}$

Atlikus modeliuojamo atkaitinimo algoritmo (SA) taikymo analizę, naudojant skirtingus atsitiktinių skaičių generatorius, matyti, kad algoritmo našumas stipriai priklauso nuo generatoriaus pasirinkimo. Daugumoje situacijų MT generatorius parodė aukštesnį sėkmingų sprendimų skaičių, ypač sprendžiant sudėtingas funkcijas, tokias kaip Levy, Easom ir Ackley. Nors PCG taip pat pasiekė konkurencingų rezultatų, jis dažniausiai nežymiai nusileido MT pagal sėkmės rodiklius, tačiau sprendimų tikslumo lygis buvo panašus.

LCG akivaizdžiai pasirodė prasčiausiai, sugebėdamas rasti labai mažai sėkmingų sprendimų visuose testuose. Tai rodo, kad šis generatorius nėra tinkamas naudoti kartu su SA algoritmu sprendžiant sudėtingus optimizavimo uždavinius. Kalbant apie konvergavimą, MT generatorius ir PCG pasiekė panašų vidutinį iteracijų skaičių iki sprendimo, tačiau MT dažniau užtikrino geresnį tikslumo ir efektyvumo balansą. Stagnacijos metu MT ir PCG generuoti sprendimai pasiekė panašias vidutines paklaidas, nors kai kuriais atvejais PCG pavyko išlaikyti kiek mažesnę paklaidą.

Apibendrinant, modeliuojamo atkaitinimo efektyvumas stipriai priklauso nuo atsitiktinių skaičių generatoriaus. MT išsiskiria kaip patikimiausias pasirinkimas, nes jis užtikrina gerą sėkmingų sprendimų, tikslumo ir efektyvumo pusiausvyrą daugelyje testavimo funkcijų. PCG gali būti tinkama alternatyva, kai prioritetas teikiamas šiek tiek geresniam tikslumui, tačiau kai kuriais atvejais tai kainuoja mažesnį sėkmingai rastų sprendimų rodiklį. Tuo tarpu LCG akivaizdžiai atsilieka pagal visus stebėtus rodiklius.

Galiausiai ir modeliuojamo atkaitinimo algoritmas buvo testuotas aukštesnės dimensijos erdvėje. Sprendžiant Levy funkciją, paklaida, su kuria buvo priimtas sprendimas, buvo padidinta iki $1 \cdot 10^{-6}$, kad sėkmingų sprendimų būtų daugiau ir padėtų analizei. Gauti rezultatai su penkiamate Levy funkcija aprašyti 29 lentelėje.

29 lentelė. *Rezultatai sprendžiant penkiamatę Levy funkciją naudojant modeliuojamo atkaitinimo algoritmą*

Stebima metrika	**LCG**	**MT**	**PCG**
Sėkmingų sprendimų skaičius	0	3	2
Vidutinis iteracijų skaičius sprendimui rasti	-	12183	11466
Vidutinė sprendimo paklaida	-	$8.55 \cdot 10^{-7}$	$9.33 \cdot 10^{-7}$
Vidutinis iteracijų skaičius kai buvo pasiekta stagnacija	6044	6101	6084
Vidutinė geriausia sprendimo paklaida, kai buvo pasiekta stagnacija	3.79	3.94	3.87

Su Ackley funkcija paklaida, su kuria buvo priimtas sprendimas, taip pat buvo padidinta iki $1 \cdot 10^{-6}$. 30 lentelėje pateikti rezultatai su penkiamate Ackley funkcija.

30 lentelė. *Rezultatai sprendžiant penkiamatę Ackley funkciją naudojant modeliuojamo atkaitinimo algoritmą*

Stebima metrika	**LCG**	**MT**	**PCG**
Sėkmingų sprendimų skaičius	17	24	21
Vidutinis iteracijų skaičius sprendimui rasti	4389	5513	4330
Vidutinė sprendimo paklaida	$7.71 \cdot 10^{-7}$	$5.73 \cdot 10^{-7}$	$8.59 \cdot 10^{-7}$
Vidutinis iteracijų skaičius kai buvo pasiekta stagnacija	5513	5512	5513
Vidutinė geriausia sprendimo paklaida, kai buvo pasiekta stagnacija	15.39	$15.4`2$	15.52

Rosenbrock funkcijai rasti sprendimą buvo keičiami visi parametrai: temperatūra, maksimalus

įteracijų skaičius, aušinimo greitis, paklaida, su kuria priimamas sprendinys, taip pat buvo padidinta iki $1 \cdot 10^{-6}$, bet rezultatų radimas išliko minimalus. Gauti rezultatai aprašyti 31 lentelėje.

31 lentelė. *Rezultatai sprendžiant penkiamatę Rosenbrock funkciją naudojant modeliuojamo atkaitinimo algoritmą*

Stebima metrika	**LCG**	**MT**	**PCG**
Sėkmingų sprendimų skaičius	0	1	1
Vidutinis iteracijų skaičius sprendimui rasti	-	12991	9452
Vidutinė sprendimo paklaida	-	$5.85 \cdot 10^{-6}$	$8.13 \cdot 10^{-6}$
Vidutinis iteracijų skaičius kai buvo pasiekta stagnacija	-	8718	8758
Vidutinė geriausia sprendimo paklaida, kai buvo pasiekta stagnacija	-	127.92	129.71

Apibendrinus modeliuojamo atkaitinimo (SA) algoritmo taikymo rezultatus penkių dimensijų atveju, galima pastebėti, kad generatoriaus pasirinkimas ir toliau turi didelės įtakos algoritmo efektyvumui. MT generatorius ir PCG pasiekė geresnius rezultatus nei LCG. MT rezultatai parodė šiek tiek geresnį sėkmingų sprendimų skaičių ir tikslumą. MT taip pat pasiekė sprendimus su mažesne vidutine sprendimo paklaida ir užtikrino greitesnį sprendimų radimą nei kiti generatoriai. PCG, nors ir rodė mažesnį sėkmingų sprendimų skaičių, pasiekė panašų sprendimų tikslumą, bet su mažesniu iteracijų skaičiumi.

LCG ir penkių dimensijų atveju vėl pasirodė prasčiausiai, pasiekdamas nedaug sėkmingų sprendimų ir didesnę sprendimų paklaidą. Tai patvirtina, kad LCG nėra tinkamas pasirinkimas sprendžiant sudėtingus optimizavimo uždavinius su SA algoritmu. Taigi, kaip ir dvimatės erdvės atveju, MT ir PCG užtikrina geresnį tikslumą ir efektyvumą, o MT išlieka geriausiu pasirinkimu tiek sėkmingų sprendimų skaičiaus, tiek tikslumo atžvilgiu,

Išvados

Rezultatai rodo, kad atsitiktinių skaičių generatoriaus pasirinkimas turi didelę įtaką optimizavimo algoritmų našumui.

1. Kolmogorov-Smirnov testo rezultatai::

    - Mersenne Twister generatorius iš visų bandymų dažniausiai generavo atsitiktinių skaičių seką pasiskirsčiusią pagal tolygųjį skirstinį. Iš visų bandymų tokių sekų buvo sugeneruota net 97% kartų. Šio generatoriaus vidutinė p-reikšmė iš visų generatorių buvo didžiausią, o vidutinė KS statistinė reikšmė sutapo su PCG generatoriaus reikšme. Tai reiškia, kad MT generatorius generuoja atsitiktines sekas labiausiai pasiskirsčiusias pagal tolygųjį skirstinį.
    - Permutacinis kongruentinis generatorius testo sąlygas atitiko 94% kartų ir pademonstravo geresnius atsitiktinių skaičių generavimo sugebėjimus nei LCG generatorius. Šis generatorius taip pat generuoja atsitiktinių skaičių sekas gerai atitinkančias tolygųjį pasiskirstymą.
    - Tiesinis kongruentinis generatorius KS testo rezultatuose pasirodė prasčiausiai ir atsitiktines sekas sugeneravo tik 87% kartų iš visų atliktų bandymų. Naudojant LCG gauta ir mažiausia p-reikšmė bei didžiausia KS statistika, t.y. LCG generuoja didžiausią neatitikimą tolygiajam skirstiniui.

2. Generatorių rezultatai dvimatėje ir penkiamatėje erdvėje:

    - MT generatorius nuosekliai demonstravo geresnius rezultatus daugelyje algoritmų ir testavimo funkcijų tiek dvimatėje, tiek penkiamatėje erdvėje. Jis pasiekė daugiau sėkmingų sprendimų, greitesnį sukonvergavimą į globalų minimumą, o sėkmingų sprendimų paklaida taip pat buvo daugelį atvėju mažesnė. Nors kai kuriose situacijose PCG parodė šiek tiek aukštesnį tikslumą, MT išliko lyderiu pagal sėkmingų sprendimų skaičių ir bendrą patikimumą. Aukštesnėje erdvėje MT ir PCG taip pat pasiekė geresnius rezultatus nei LCG. MT buvo šiek tiek geresnis pagal sprendimo tikslumą ir iteracijų skaičių. Mersenne Twister rezultatai parodė, kad jis yra patikimiausias ir universaliausias pasirinkimas – užtikrindamas stabilų sėkmingų sprendimų skaičių, greitą sukonvergavimą ir tikslius rezultatus.
    - PCG generatorius, nors ir konkurencingas tikslumo atžvilgiu, daugelyje atvejų reikalavo daugiau iteracijų, kad pasiektų tokius pat rezultatus kaip MT. Penkiamatės erdvės atveju PCG taip pat pasiekė aukštus sėkmingų sprendimų rodiklius, kartais net konverguodamas greičiau į globalų minimumą nei MT ar LCG. Taigi, PCG gali būti tinkama alternatyva tam tikras atvejais sprendžiant tam tikras funkcijas - pavyzdžiui Rosenbrock. Su šia funkcija PCG sugebėjo daugelyje atvejų rasti sprendimą greičiau nei MT.
    - LCG generatorius visais atvejais demonstravo prastesnius rezultatus nepriklausomai nuo funkcijos nei nuo naudojamo optimizacijos algoritmo. Nors ir kai kuriose situacijose, kaip PSO, algoritmas sugebėjo rasti pakankamai daug sprendinių, LCG vis vien nusileisdavo efektyvesniems ir geresniems atsitiktinių skaičių generatoriams, kaip PCG ir Mersenne Twister.

3. Optimizacijos algoritmų rezultatai dvimatėje ir penkiamatėje erdvėje:

    - Genetinis algoritmas (GA) buvo ypač efektyvus tiek dvimatėje, tiek penkiamatėje erdvėje naudojant MT, kuris parodė geriausią balansą tarp greito sprendimų paieškos ir sėkmingų sprendimų skaičiaus. PCG šiuo atveju pasižymėjo geriausiu tikslumu, tačiau dėl dažnesnės stagnacijos ir didesnio iteracijų skaičiaus MT išliko lyderiu. LCG našumas genetiniuose algoritmuose buvo prastas, dažnai reikėjo daugiau iteracijų ir sprendimai buvo mažiau tikslūs.

- Dalelių spiečiaus optimizavimo algoritmas (PSO) rodė stabiliai gerus rezultatus tiek abiejose erdvėse naudojant MT ir PCG generatorius – pasiekė aukštus sėkmingų sprendimų rodiklius bei tikslumus. Tačiau, kaip ir dviejų dimensijų taip ir penkių dimensijų atveju PCG dažniau reikėjo daugiau iteracijų, kad pasiektų tokius pat rezultatus kaip MT. MT buvo greitesnis ir pasiekė didesnius sėkmingų sprendimų rodiklius. LCG buvo žymiai prastesnis pasirinkimas, nes pasiekė mažiau sėkmingų sprendimų ir dažnai reikalavo daugiau iteracijų.
- Modeliuojamo atkaitinimo (SA) algoritmas neparodė skirtingų rezultatų perėjus prie aukštesnės erdvės. MT išliko patikimiausiu ir greičiausiu pasirinkimu, užtikrindamas didesnį tikslumą ir mažesnį iteracijų skaičių. PCG pateikė panašius, bet šiek tiek mažiau pastovius rezultatus, o LCG veikimas SA algoritme buvo prasčiausias – beveik visose funkcijose sėkmingų sprendimų skaičius buvo minimalus arba sprendimai nebuvo randami.
- Paprastosios atsitiktinės paieškos algoritmas (PRS), būdamas pats paprasčiausias, kurio principas paremtas paprasčiausiu skaičių generavimu, parodė, kad atsitiktinių skaičių generatoriai be papildomų modifikacijų ar žingsnių turi didelę įtaką algoritmo efektyvumui. MT generatorius pralenkė tiek PCG, tiek LCG visose funkcijose parodydamas geriausius rezultatus.

Apibendrinant, optimizacijos algoritmų veikimas tiek dvimatėje, tiek aukštesnėje erdvėje, parodė, kad MT generatorius yra pats efektyviausias ir patikimiausias pasirinkimas tiek sėkmingų sprendimų, tiek tikslumo ir konvergavimo į globalų minimumą atžvilgiu. PCG, nors ir rodė gerus rezultatus, dažniau reikalavo daugiau iteracijų nei MT, tačiau buvo panašus, o kartais net geresnis tikslumo (vidutinės paklaidos) atžvilgiu. LCG generatorius nuolat demonstravo prastesnius rezultatus ir yra nerekomenduotinas sudėtingoms optimizavimo užduotims. Remiantis atliktais tyrimais, siūlomos šios rekomendacijos:

- Pagrindinis generatorius: MT generatorių rekomenduojama naudoti kaip pagrindinį pasirinkimą įvairiuose optimizavimo algoritmuose dėl jo patikimumo ir universalumo.
- Alternatyva tikslumui: PCG galima svarstyti tais atvejais, kai reikalingas didesnis tikslumas, tačiau priimtinas mažesnis sėkmingų sprendimų skaičius.
- LCG vengimas: Dėl prastų rezultatų visose nagrinėtose metrikose, LCG nerekomenduojamas naudoti optimizavimo užduotims, reikalaujančioms sudėtingo ieškojimo ir išnaudojimo balanso.
- Ateityje verta tirti hibridinius metodus, derinant MT ir PCG privalumus, taip pat analizuoti parametrų optimizavimo poveikį siekiant sumažinti specifinių generatorių ar algoritmų trūkumus.

Literatūra ir šaltiniai

[Abu+23] L. Abualigah ir kiti. „Hybrid Slime Mold and Arithmetic Optimization Algorithm with Random Center Learning and Restart Mutation“. Iš: *Biomimetics* 8.5 (2023), puslapis 396. https://doi.org/10.3390/biomimetics8050396.

[Ack87] D. H. Ackley. *A Connectionist Machine for Genetic Hillclimbing*. Kluwer Academic Publichers, 1987.

[Ado05] E. P. Adorio. „MVF - Multivariate Test Functions Library in C for Unconstrained Global Optimization“. Iš: *Computer Science, Engineering* (2005).

[BB12] J. Bergstra, Y. Bengio. „Random Search for Hyper-Parameter Optimization“. Iš: *Journal of Machine Learning Research* 13 (2012), puslapiai 281–305.

[BR18] C. Blum, A. Roli. „Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison“. Iš: *ACM Computing Surveys (CSUR)* 35.3 (2018), puslapiai 268–308.

[Cav13] M. Cavazzuti. „Deterministic Optimization“. Iš: *Optimization Methods: From Theory to Design Scientific and Technological Aspects in Mechanics*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, puslapiai 77–102. ISBN: 978-3-642-31187-1. https://doi.org/10.1007/978-3-642-31187-1_4. URL: https://doi.org/10.1007/978-3-642-31187-1_4.

[DDS+24] S. Dauzère-Pérès, J. Ding, L. Shen, K. Tamssaouet. „The flexible job shop scheduling problem: A review“. Iš: *European Journal of Operational Research* 314.2 (2024), puslapiai 409–432. ISSN: 0377-2217. https://doi.org/10.1016/j.ejor.2023.05.017. URL: https://www.sciencedirect.com/science/article/pii/S037722172300382X.

[Eas83] M. E. Easom. „An investigation of the sensitivity of minimum cycle time to various input parameters for flexible manufacturing systems using simulation“. Disertacija. Loughborough University, 1983.

[ES98] R. C. Eberhart, Y. Shi. „A comparison of particle swarm optimization and genetic algorithms“. Iš: *Proceedings of the 1998 international conference on evolutionary computation (IEEE World Congress on Computational Intelligence)* (1998), puslapiai 611–616.

[GDT+09] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, M. Booth, F. Rossi. *GNU Scientific Library: Reference Manual*. 3rd. Network Theory Ltd., 2009.

[Hol75] J. H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT Press, 1975.

[JAM+89] D. S. Johnson, C. R. Aragon, L. A. McGeoch, C. Schevon. „Optimization by simulated annealing: An experimental evaluation; part II, graph coloring and number partitioning“. Iš: *Operations Research* 39.3 (1989), puslapiai 378–406.

[Jia19] H. L. Jian Ma. „Research on Rosenbrock Function Optimization Problem Based on Improved Differential Evolution Algorithm“. Iš: *Scientific Research Publishing* (2019).

[Jr51] F. J. M. Jr. „The Kolmogorov-Smirnov Test for Goodness of Fit“. Iš: *Journal of the American Statistical Association* 46.253 (1951), puslapiai 68–78. https://doi.org/10.1080/01621459.1951.10500769. URL: https://www.tandfonline.com/doi/abs/10.1080/01621459.1951.10500769.

[KE95] J. Kennedy, R. Eberhart. „Particle Swarm Optimization“. Iš: *Proceedings of ICNN'95 - International Conference on Neural Networks*. IEEE. 1995, puslapiai 1942–1948.

[KGV83] S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi. „Optimization by Simulated Annealing“. Iš: *Science* 220.4598 (1983), puslapiai 671–680.

[Knu97] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. 3rd. Addison-Wesley, 1997.

[KW08] M. Kalos, P. Whitlock. *Monte Carlo Methods*. Wiley, 2008. ISBN: 9783527407606. URL: https://books.google.lt/books?id=b8Xb4rBkygUC.

[LEc17] P. L'Ecuyer. „Random Number Generation". Iš: *Handbook of Computational Statistics*. Springer, Berlin, Heidelberg, 2017, puslapiai 35–71.

[LEc88] P. L'Ecuyer. „Efficient and portable combined random number generators". Iš: *Communications of the ACM* 31.6 (1988), puslapiai 742–749.

[Lev98] A. Levy. „Convergence properties of the Nelder-Mead simplex method in low dimensions". Iš: *SIAM Journal on Optimization* 9.1 (1998), puslapiai 112–147.

[LJC19] Y. Li, L. Jiao, Z. Cai. „A Study on the Impact of Random Number Generation on Genetic Algorithm for Flexible Job Shop Scheduling Problem". Iš: (2019).

[LM02] M. Laguna, R. Marti. „Experimental Testing of Advanced Scatter Search Designs for Global Optimization of Multimodal Functions". Iš: *Journal of Global Optimization* 22.1-4 (2002), puslapiai 239–268.

[Mar05] C. S. Marcin Molga. „Test functions for optimization needs". Iš: (2005).

[Mat23] MathWorks. *Random Number Generators*. https://www.mathworks.com/help/matlab/random-number-generators.html. 2023.

[MN98] M. Matsumoto, T. Nishimura. „Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator". Iš: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 8.1 (1998), puslapiai 3–30.

[MU49] N. Metropolis, S. Ulam. *The Monte Carlo Method*. Tomas 44. Original work introducing the Monte Carlo method. Journal of the American Statistical Association, 1949, puslapiai 335–341.

[MV12] Z. Ma, G. A. E. Vandenbosch. „Impact of Random Number Generators on the performance of particle swarm optimization in antenna design". Iš: *2012 6th European Conference on Antennas and Propagation (EUCAP)*. 2012, puslapiai 925–929. https://doi.org/10.1109/EuCAP.2012.6205998.

[Nis00] T. Nishimura. „Tables of 64-bit Mersenne Twisters". Iš: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 10.4 (2000), puslapiai 348–357.

[ONe14] M. E. O'Neill. „PCG: A family of simple fast space-efficient statistically good algorithms for random number generation". Iš: *arXiv preprint arXiv:1402.6246* (2014).

[Pyt23] Python Software Foundation. *random — Generate pseudo-random numbers*. https://docs.python.org/3/library/random.html. 2023.

[PM88] S. K. Park, K. W. Miller. „Random number generators: Good ones are hard to find". Iš: *Communications of the ACM* 31.10 (1988), puslapiai 1192–1201.

[PTV+07] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. 3rd. Cambridge University Press, 2007.

[RC98] F. J. Resende, B. V. Costa. „Using random number generators in Monte Carlo simulations". Iš: *Phys. Rev. E* 58.4 (1998), puslapiai 5183–5184. https://doi.org/10.1103/PhysRevE.58.5183. URL: https://link.aps.org/doi/10.1103/PhysRevE.58.5183.

[RTB12] S. Razavi, B. A. Tolson, D. H. Burn. „Review of Surrogate Modeling in Water Resources". Iš: *Water Resources Research* 48.7 (2012).

[SB13] S. Surjanovic, D. Bingham. *Easom Function*. `https://www.sfu.ca/~ssurjano/easom.html`. 2013.

[Sim15] P. A. Simionescu. „Random Search Optimization“. Iš: *Handbook of Research on Artificial Intelligence Techniques and Algorithms*. IGI Global, 2015, puslapiai 324–342.

[SS16] H. Schumann, T. Schürmann. „The Impact of Pseudo-Random Number Generators on Monte Carlo Simulations“. Iš: *Journal of Computational Physics* 123.2 (2016), puslapiai 465–489.

[SS89] W. Siedlecki, J. Sklansky. „A survey of genetic algorithms for feature selection“. Iš: *International journal of pattern recognition and artificial intelligence* 3.02 (1989), puslapiai 199–209.

[Tro23] P. Trojovský. „A new bio-inspired metaheuristic algorithm for solving optimization problems based on walruses behavior“. Iš: *Scientific Reports* (2023).

Priedas

```
# Kolmogorov-Smirnov testas
class LCG:
    def __init__(self, seed=None, a=5, c=1, m=2**32):
        if seed is None:
            seed = int((time.time() * 1000) % m)
        self.seed = seed % m
        self.a = a
        self.c = c
        self.m = m

    def random(self):
        self.seed = (self.a * self.seed + self.c) % self.m
        return self.seed / self.m

def lcg_random_numbers(n, seed=None):
    lcg = LCG(seed)
    return [lcg.random() for _ in range(n)]

def perform_ks_tests(generator, n_iterations, n_numbers,
                     significance_level, ks_crit):
    p_values = []
    ks_stats = []
    accept_pvalue = 0
    accept_kscrit = 0
    unique_counts = []

    for _ in range(n_iterations):
        if generator == 'lcg':
            random_numbers = lcg_random_numbers(n_numbers)
        elif generator == 'mt':
            random.seed()
            random_numbers = [random.random() for _ in range(n_numbers)]
        elif generator == 'pcg':
            rng_pcg = np.random.default_rng()
            random_numbers = rng_pcg.random(n_numbers)

        ks_stat, p_value = kstest(random_numbers, 'uniform')
        p_values.append(p_value)
        ks_stats.append(ks_stat)

        if p_value > significance_level:
            accept_pvalue += 1;
        if ks_stat < ks_crit:
            accept_kscrit += 1;
        unique_count = len(set(random_numbers))
        unique_counts.append(unique_count)
```

```
    avg_p_value = np.mean(p_values)
    avg_ks_stat = np.mean(ks_stats)
    avg_unique_count = np.mean(unique_counts)
    return avg_p_value, avg_ks_stat, accept_pvalue,
            accept_kscrit, avg_unique_count

n_iterations = 1000
n_numbers = 100000
ks_crit = 1.36/math.sqrt(n_numbers)
significance_level=0.05

avg_p_value_lcg, avg_ks_stat_lcg, accept_pvalue_lcg, accept_kscrit_lcg,
 avg_unique_count_lcg = \
 perform_ks_tests('lcg', n_iterations, n_numbers,significance_level,ks_crit)
avg_p_value_mt, avg_ks_stat_mt, accept_pvalue_mt, accept_kscrit_mt,
 avg_unique_count_mt = \
 perform_ks_tests('mt', n_iterations, n_numbers,significance_level,ks_crit)
avg_p_value_pcg, avg_ks_stat_pcg, accept_pvalue_pcg, accept_kscrit_pcg,
 avg_unique_count_pcg = \
 perform_ks_tests('pcg', n_iterations, n_numbers,significance_level,ks_crit)

# Rezultatai
print("LCG:")
print(f"Vidutinė KS-reikšmė: {avg_ks_stat_lcg}")
print(f"Vidutinė p-reikšmė: {avg_p_value_lcg}")
print(f"Hypotezės p-value priemimu skaičius: \
{accept_pvalue_lcg}, {(accept_pvalue_lcg/n_iterations)*100}%")
print(f"Hypotezės ks_stat priemimu skaičius: \
{accept_kscrit_lcg}, {accept_kscrit_lcg/n_iterations*100}%")
print(f"Skirtingi skaičiai: {avg_unique_count_lcg}")

print("\nMersenne Twister:")
print(f"Vidutinė KS-reikšmė: {avg_ks_stat_mt}")
print(f"Vidutinė p-reikšmė: {avg_p_value_mt}")
print(f"Hypotezės p-value priemimu skaičius: \
{accept_pvalue_mt}, {accept_pvalue_mt/n_iterations*100}%")
print(f"Hypotezės ks_stat priemimu skaičius: \
{accept_kscrit_mt}, {accept_kscrit_mt/n_iterations*100}%")
print(f"Skirtingi skaičiai: {avg_unique_count_mt}")

print("\nPCG:")
print(f"Vidutinė KS-reikšmė: {avg_ks_stat_pcg}")
print(f"Vidutinė p-reikšmė: {avg_p_value_pcg}")
print(f"Hypotezės p-value priemimu skaičius: \
{accept_pvalue_pcg}, {accept_pvalue_pcg/n_iterations*100}%")
print(f"Hypotezės ks_stat priemimu skaičius: \
{accept_kscrit_pcg}, {accept_kscrit_pcg/n_iterations*100}%")
print(f"Skirtingi skaičiai: {avg_unique_count_pcg}")
```

```
## Generatoriai:
# PCG realizavimas:
rng = np.random.default_rng(np.random.PCG64())

# MT generatoriaus realizavimas
rng = random.Random()


# Papildomi parametrai naudojami su LCG
MODULUS = 2**32
MULTIPLIER = 5
INCREMENT = 1
# LCG realizavimas
class LCG:
    def __init__(self, seed=None):
        if seed is None:
            seed = self._initial_seed()
        self.seed = seed % MODULUS
        self.a = MULTIPLIER
        self.c = INCREMENT
        self.m = MODULUS

    def _initial_seed(self):
        import time
        return int((time.time() * 1000) % MODULUS)

    def random(self):
        self.seed = (self.a * self.seed + self.c) % self.m
        return self.seed / self.m

    def uniform(self, lower, upper):
        return lower + (upper - lower) * self.random()

rng = LCG()


## Funkcijų pritaikymas:
# Parametrai naudojami Levy funkcijai su visais generatoriais
LOWER_BOUND = -10
UPPER_BOUND = 10
ERROR_THRESHOLD = 1e-8 # Parametras gali būti keičiamas pagal poreikį
GLOBAL_MINIMUM = 0  # Levy funkcijos globalus minimumas

# Levy funkcija
def function(solution):
    w = 1 + (np.array(xx) - 1) / 4
    term1 = (np.sin(np.pi * w[0]))**2
    term3 = (w[-1] - 1)**2 * (1 + (np.sin(2 * np.pi * w[-1]))**2)
```

```
    sum_terms = sum((wi - 1)**2 * (1 + 10 * (np.sin(np.pi *
                                                wi))**2) for wi in w[:-1])
    return term1 + sum_terms + term3


# Parametrai naudojami Easom funkcijai su visais generatoriais
LOWER_BOUND = -10
UPPER_BOUND = 14
ERROR_THRESHOLD = 1e-8 # Parametras gali būti keičiamas pagal poreikį
GLOBAL_MINIMUM = -1  # Easom funkcijos globalus minimumas

# Easom funkcija
def function(solution):
    x, y = solution
    return -np.cos(x) * np.cos(y) * np.exp(-((x - np.pi) ** 2 +
     (y - np.pi) ** 2))


# Parametrai naudojami Ackley funkcijai su visais generatoriais
LOWER_BOUND = -5
UPPER_BOUND = 5
ERROR_THRESHOLD = 1e-8 # Parametras gali būti keičiamas pagal poreikį
a = 20
b = 0.2
c = 2 * np.pi
GLOBAL_MINIMUM = 0  # Ackley funkcijos globalus minimumas

# Ackley funkcija
def function(solution):
    sum1 = sum(x ** 2 for x in solution)
    sum2 = sum(np.cos(c * x) for x in solution)
    term1 = -a * np.exp(-b * np.sqrt(sum1 / DIMENSIONS))
    term2 = -np.exp(sum2 / DIMENSIONS)
    return term1 + term2 + a + np.exp(1)


# Parametrai naudojami Rosenbrock funkcijai su visais generatoriais
LOWER_BOUND = -5
UPPER_BOUND = 10
ERROR_THRESHOLD = 1e-8 # Parametras gali būti keičiamas pagal poreikį
GLOBAL_MINIMUM = 0  # Rosenbrock funkcijos globalus minimumas

# Rosenbrock funkcija
def function(solution):
    return sum(100 * (solution[i+1] - solution[i]**2)**2 + (1 - solution[i])**2
               for i in range(len(solution) - 1))

# Parametrai naudojami Six-hump Camel funkcijai su visais generatoriais
LOWER_BOUND_X1 = -3
UPPER_BOUND_X1 = 3
```

```
LOWER_BOUND_X2 = -2
UPPER_BOUND_X2 = 2
ERROR_THRESHOLD = 1e-8
GLOBAL_MINIMUM = -1.0316 # Six-hump Camel funkcijos globalus minimumas

# Six-Hump Camel funkcija
def function(solution):
    x1, x2 = solution
    return 4*x1**2 - 2.1*x1**4 + (x1**6) / 3 + x1*x2 - 4*x2**2 + 4*x2**4

### Paprastoji atsitiktinė paieška
### naudojant skirtingas funkcijas ir skirtingus generatorius

# Geriausio sprendinio paieška
def search_best_solution(max_iterations=1000000):

    # Naudojama taikant Levy, Easom, Ackley ir Rosenbrock funkciją
    search_space = [[LOWER_BOUND, UPPER_BOUND] for _ in range(2)]

    # Naudojama taikant Six-Hump Camel funkciją
    # search_space = [
    #     [LOWER_BOUND_X1, UPPER_BOUND_X1],
    #     [LOWER_BOUND_X2, UPPER_BOUND_X2],
    # ]

    best_solution = None
    best_fitness = float('inf')
    best_iteration = -1
    success_count_in_run = 0
    success_iterations = []
    success_errors_in_run = []
    run_errors = []
    error_at_iterations = []

    for iteration in range(max_iterations):
        xx = [rng.uniform(search_space[i][0], search_space[i][1])
                          for i in range(2)]
        fitness = easom_function(xx)
        error = abs(fitness - GLOBAL_MINIMUM)
        error_at_iterations.append((xx, fitness, error))

        if error <= 1e-5:
            success_count_in_run += 1
            success_errors_in_run.append(error)
            success_iterations.append(iteration + 1)
            print(f"Success Found: Iteration {iteration + 1}, \
            Error {error}, Coordinates {xx}, Fitness {fitness}")

        if fitness < best_fitness:
            best_solution = xx
```

```
            best_fitness = fitness
            best_iteration = iteration + 1

        if error > 1e-5:
            run_errors.append((iteration, error))

    return (
        success_count_in_run,
        best_iteration,
        best_fitness,
        best_solution,
        error_at_iterations,
        success_errors_in_run,
        run_errors,
        success_iterations,
    )

# Atliekama paieška kelis kartus ir surandami rezultatai
def run_multiple_searches(num_runs, max_iterations):
    total_success_count = 0
    total_iterations_to_solution = 0
    total_time = 0
    all_errors = []
    success_errors = []
    unsuccessful_errors = []
    first_solution_iterations = []
    min_fitness_after_success_all_runs = []

    for run in range(num_runs):
        start_time = time.time()

        (
            success_count_in_run,
            best_iteration,
            best_fitness,
            best_solution,
            error_at_iterations,
            success_errors_in_run,
            run_errors,
            success_iterations,
        ) = search_best_solution(max_iterations)

        total_success_count += success_count_in_run
        first_solution_iterations.extend(success_iterations)
        all_errors.extend([error for _, _, error in error_at_iterations])
        success_errors.extend(success_errors_in_run)
        unsuccessful_errors.extend([error for _, error in run_errors])

        if success_iterations:
            last_success_iteration = success_iterations[-1]
```

```
            errors_after_last_success = [
                error for iteration, error in run_errors
                if iteration > last_success_iteration]
            if errors_after_last_success:
                min_fitness_after_success_all_runs.append(
                    min(errors_after_last_success))
        else:
            if run_errors:
                min_fitness_after_success_all_runs.append(min(
                    error for _, error in run_errors))

        end_time = time.time()
        total_time += (end_time - start_time)

    # Skaičiuojami rezultatai
    avg_time_per_run = total_time / num_runs
    avg_error_all_runs = np.mean(all_errors)
                         if all_errors else None
    avg_error_successful = np.mean(success_errors)
                         if success_errors else None
    avg_error_unsuccessful = np.mean(unsuccessful_errors)
                         if unsuccessful_errors else None
    avg_min_fitness_after_success = (
        np.mean(min_fitness_after_success_all_runs)
                         if min_fitness_after_success_all_runs else None)
    avg_iterations_to_solution = (
        np.mean(first_solution_iterations)
                         if first_solution_iterations else None)

    # Atspausdinami rezultatai
    print(f"\nTotal Runs: {num_runs}")
    print(f"Total Success Count (fitness <= 1e-5): {total_success_count}")
    print(f"Average Time per Run: {avg_time_per_run:.4f} seconds")

    if total_success_count > 0:
        print(f"Average Iteration to Find Solution:\
         {avg_iterations_to_solution}")
        print(f"Average Error for Successful Runs:\
         {avg_error_successful}")
    else:
        print("No successes found within the defined error threshold.")

    if avg_error_all_runs is not None:
        print(f"Average Error Across All Runs: {avg_error_all_runs}")
    if avg_error_unsuccessful is not None:
        print(f"Average Error for Unsuccessful Runs: \
        {avg_error_unsuccessful}")
    if avg_min_fitness_after_success is not None:
        print(f"Average Minimum Fitness/Error per Run (After Successes):\
         {avg_min_fitness_after_success}")
```

```
    else:
        print("Average Minimum Fitness/Error per Run (After Successes): N/A")

run_multiple_searches(num_runs=1, max_iterations=1000000)



### Genetinis Algoritmas
### naudojant skirtingas funkcijas ir skirtingus generatorius

# Algoritmo parametrai
POPULATION_SIZE = 100
error_threshold = 1e-8 # keičiamas pagal funkciją
DIMENSIONS = 2 # 2 arba 5
stagnation_limit = 20
reset_limit = 30
total_runs = 1000

class Individual:
    def __init__(self, chromosome):
        self.chromosome = chromosome
        self.fitness = self.calculate_fitness()
        self.error = abs(self.fitness - GLOBAL_MINIMUM)

    @classmethod
    def create_gene(cls):
        return rng.uniform(LOWER_BOUND, UPPER_BOUND)

    @classmethod
    def create_chromosome(cls):
        return [cls.create_gene() for _ in range(DIMENSIONS)]
    # #Naudoti šitą metodą jei naudojama Six-Hump Camel funkcja
    # @classmethod
    # def create_chromosome(cls):
    #     x1 = cls.create_gene(LOWER_BOUND_X1, UPPER_BOUND_X1)
    #     x2 = cls.create_gene(LOWER_BOUND_X2, UPPER_BOUND_X2)
    #     return [x1, x2]

    def calculate_fitness(self):
        return function(self.chromosome)

    def mate(self, partner):
        child_chromosome = [
            gp1 if rng.random() < 0.5 else gp2
            for gp1, gp2 in zip(self.chromosome, partner.chromosome)
        ]
        mutated_gene = self.mutate_gene()
        child_chromosome[int(rng.uniform(0, DIMENSIONS))] = mutated_gene
        return Individual(child_chromosome)
```

```
    def mutate_gene(self):
        mutation_strength = 0.5
        # mutation = rng.normal(0, mutation_strength) # jei naudojamas PCG
        # mutation = rng.random() * mutation_strength # jei naudojamas LCG
        mutation = random.gauss(0, mutation_strength) # jei naudojamas MT
        # Naudojamas su Six-Hump Camel funkcija
        # gene_index = int(rng.uniform(0, DIMENSIONS))
        # Naudojamas su Six-Hump Camel funkcija
        # mutated_gene = self.chromosome[gene_index] + mutation
        # # Naudojamas su Six-Hump Camel funkcija
        # if gene_index == 0:  # x1 within [-3, 3]
        #     return np.clip(mutated_gene, LOWER_BOUND_X1, UPPER_BOUND_X1)
        # else:  # x2 within [-2, 2]
        #     return np.clip(mutated_gene, LOWER_BOUND_X2, UPPER_BOUND_X2)
        mutated_gene = self.chromosome[int(rng.uniform(0, DIMENSIONS))] + mutation
        return np.clip(mutated_gene, LOWER_BOUND, UPPER_BOUND)

def genetic_algorithm():
    generation = 1
    found = False
    population = []
    error_history = []

    # Initialize the population
    for _ in range(POPULATION_SIZE):
        chromosome = Individual.create_chromosome()
        population.append(Individual(chromosome))

    stagnation_count = 0
    reset_count = 0
    best_error = float('inf')
    best_generation = 0
    best_chromosome = []

    # Metrics for runs
    success_count = 0
    total_successful_generations = 0
    total_best_solution_error = 0.0
    total_stopping_generations = 0
    stopped_due_to_reset_count = 0
    generations_at_reset_count = 0
    best_solution_error_at_reset = 0.0

    while not found:
        population = sorted(population, key=lambda ind: ind.error)

        # Check if error is within threshold
        if population[0].error <= error_threshold:
            found = True
            best_error = population[0].error
```

```
        best_generation = generation
        best_chromosome = population[0].chromosome
        error_history.append(best_error)
        print(f"Solution found at generation {generation}:
            Minimum Error: {best_error:.11f}, Chromosome: {best_chromosome}")
        success_count += 1
        total_successful_generations += generation
        total_best_solution_error += best_error
        break

    current_best_error = population[0].error
    error_history.append(current_best_error)

    # Update best error and chromosome
    if current_best_error < best_error:
        best_error = current_best_error
        best_generation = generation
        best_chromosome = population[0].chromosome
        stagnation_count = 0
    else:
        stagnation_count += 1

    if stagnation_count > stagnation_limit:
        for _ in range(int(0.2 * POPULATION_SIZE)):
            chromosome = Individual.create_chromosome()
            population[int(rng.uniform(0, POPULATION_SIZE))] =\
                Individual(chromosome)
        reset_count += 1
        stagnation_count = 0

        if reset_count >= reset_limit:
            print(f"Stopped due to 30 resets at generation {generation}.")
            print(f"Minimum Error encountered: {best_error:.11f},
                at Generation: {best_generation}, Chromosome: {best_chromosome}")
            total_stopping_generations += generation
            stopped_due_to_reset_count += 1
            generations_at_reset_count += generation
            best_solution_error_at_reset = best_error
            error_history.append(best_error)
            break

    new_generation = []

    # Elitism
    s = int(0.10 * POPULATION_SIZE)
    new_generation.extend(population[:s])

    s = int(0.90 * POPULATION_SIZE)
    for _ in range(s):
        parent1 = tournament_selection(population)
```

```
            parent2 = tournament_selection(population)
            child = parent1.mate(parent2)
            new_generation.append(child)

        population = new_generation
        generation += 1

    return success_count, total_successful_generations, total_best_solution_error,\
        total_stopping_generations, stopped_due_to_reset_count,\
            generations_at_reset_count, best_solution_error_at_reset

def tournament_selection(population):
    tournament_size = 5
    selected = [population[int(rng.uniform(0, POPULATION_SIZE))]
                for _ in range(tournament_size)]
    selected = sorted(selected, key=lambda ind: ind.error)
    return selected[0]

success_count_total = 0
total_successful_generations = 0
total_best_solution_error = 0.0
total_stopping_generations = 0
stopped_due_to_reset_count_total = 0
total_generations_at_reset_count = 0
total_best_solution_error_for_resets = 0.0

for _ in range(total_runs):
    results = genetic_algorithm()
    success_count, successful_generations,\
        best_solution_error, stopping_generations,\
            stopped_due_to_reset_count, generations_at_reset_count,\
                best_solution_error_at_reset = results
    success_count_total += success_count
    total_successful_generations += successful_generations
    total_best_solution_error += best_solution_error
    total_stopping_generations += stopping_generations
    stopped_due_to_reset_count_total += stopped_due_to_reset_count
    total_generations_at_reset_count += generations_at_reset_count

    if stopped_due_to_reset_count > 0:
        total_best_solution_error_for_resets += best_solution_error_at_reset

# Vidurkiai
average_successful_generations = total_successful_generations /\
    success_count_total if success_count_total > 0 else 0
average_best_solution_error = total_best_solution_error /\
    success_count_total if success_count_total > 0 else 0
average_stopping_generations = total_generations_at_reset_count /\
    stopped_due_to_reset_count_total if stopped_due_to_reset_count_total > 0 else 0
average_best_solution_error_for_resets = total_best_solution_error_for_resets /\
```

```
    stopped_due_to_reset_count_total if stopped_due_to_reset_count_total > 0 else 0

# Rezultatai
print(f"\nResults after {total_runs} runs:")
print(f"Success Count: {success_count_total}")
print(f"Average Successful Generations: {average_successful_generations}")
print(f"Average Minimum Error: {average_best_solution_error}")
print(f"Average Stopping Generations (due to reset count):
    {average_stopping_generations}")
print(f"Stopped due to Reset Count: {stopped_due_to_reset_count_total}")
print(f"Average Minimum Error for Runs Ending with Reset Count:
    {average_best_solution_error_for_resets}")

### Genetinis algoritmas
## Pagerinta Ackley funkcija ir Ackley funkcija penkiamatėje erdvėje:
import numpy as np
import random

# Parametrai
POPULATION_SIZE = 100
DIMENSIONS = 2 # 2 arba 5
LOWER_BOUND = -32.768
UPPER_BOUND = 32.768
MAX_GENERATIONS = 5000 # 2000 dviejų dimensijų
STAGNATION_LIMIT = 50
error_threshold = 1e-5 #  1e-5 arba  1e-6
total_runs = 1000

# Ackley funkcija
def ackley_function(solution):
    a = 20
    b = 0.2
    c = 2 * np.pi
    solution = np.array(solution)
    d = len(solution)
    sum1 = np.sum(np.square(solution))
    sum2 = np.sum(np.cos(c * solution))
    return -a * np.exp(-b * np.sqrt(sum1 / d)) - np.exp(sum2 / d) + a + np.e

# Generatoriai
# LCG parametrai
MODULUS = 2**32
MULTIPLIER = 5
INCREMENT = 1
class LCG:
    def __init__(self, seed=None):
        if seed is None:
            seed = self._initial_seed()
        self.seed = seed % MODULUS
        self.a = MULTIPLIER
```

```
        self.c = INCREMENT
        self.m = MODULUS

    def _initial_seed(self):
        import time
        return int((time.time() * 1000) % MODULUS)

    def random(self):
        self.seed = (self.a * self.seed + self.c) % self.m
        return self.seed / self.m

    def uniform(self, lower, upper):
        return lower + (upper - lower) * self.random()

# rng = LCG()
rng = random.Random()
# rng = np.random.default_rng(np.random.PCG64())

class Individual:
    def __init__(self, chromosome):
        self.chromosome = chromosome
        self.fitness = self.calculate_fitness()
        self.error = abs(self.fitness)

    @classmethod
    def create_gene(cls):
        return rng.uniform(LOWER_BOUND, UPPER_BOUND)

    @classmethod
    def create_chromosome(cls):
        return [cls.create_gene() for _ in range(DIMENSIONS)]

    def calculate_fitness(self):
        return ackley_function(self.chromosome)

    def mate(self, partner, generation, max_generations):
        child_chromosome = [
            gp1 if rng.random() < 0.5 else gp2
            for gp1, gp2 in zip(self.chromosome, partner.chromosome)
        ]
        mutated_gene = self.mutate_gene(generation, max_generations)
        child_chromosome[int(rng.uniform(0, DIMENSIONS))] = mutated_gene
        return Individual(child_chromosome)

    def mutate_gene(self, generation, max_generations):
        mutation_strength = max(0.01, 0.1 * (1 - generation / max_generations))
        # mutation = rng.normal(0, mutation_strength) # jei naudojamas PCG
        # mutation = rng.random() * mutation_strength # jei naudojamas LCG
        mutation = random.gauss(0, mutation_strength) # jei naudojamas MT
        mutated_gene = self.chromosome[int(rng.uniform(0, DIMENSIONS))] + mutation
```

```
        return np.clip(mutated_gene, LOWER_BOUND, UPPER_BOUND)


def tournament_selection(population):
    tournament_size = 5
    selected = [population[int(rng.uniform(0, len(population)))]
                for _ in range(tournament_size)]
    selected = sorted(selected, key=lambda ind: ind.error)
    return selected[0]

def random_refinement(best_solution, best_error):
    refined_solution = [
        np.clip(coord - rng.uniform(1e-8, 1e-5), LOWER_BOUND, UPPER_BOUND)
        for coord in best_solution
    ]
    refined_fitness = ackley_function(refined_solution)
    if refined_fitness < best_error - 1e-8:
        return refined_solution, refined_fitness
    return best_solution, best_error


def genetic_algorithm():
    generation = 1
    stagnation_count = 0
    reset_count = 0
    population = [Individual(Individual.create_chromosome())
        for _ in range(POPULATION_SIZE)]
    best_error = float('inf')
    best_chromosome = []
    best_generation = 0

    while generation <= MAX_GENERATIONS:
        population = sorted(population, key=lambda ind: ind.error)

        if population[0].error < best_error:
            best_error = population[0].error
            best_chromosome = population[0].chromosome
            best_generation = generation
            stagnation_count = 0
        else:
            stagnation_count += 1

        if best_error <= error_threshold:
            print(f"Solution found at generation {generation}:
                Minimum Error: {best_error}, Chromosome: {best_chromosome}")
            return 1, generation, best_error, reset_count

        if stagnation_count > STAGNATION_LIMIT:
            for _ in range(int(0.2 * POPULATION_SIZE)):
                population[int(rng.uniform(0, POPULATION_SIZE))] =\
```

```
                    Individual(Individual.create_chromosome())
            stagnation_count = 0
            reset_count += 1

        if generation % 100 == 0:
            best_chromosome, best_error =\
                random_refinement(best_chromosome, best_error)

        new_generation = []
        elitism_count = int(0.1 * POPULATION_SIZE)
        new_generation.extend(population[:elitism_count])

        for _ in range(POPULATION_SIZE - elitism_count):
            parent1 = tournament_selection(population)
            parent2 = tournament_selection(population)
            child = parent1.mate(parent2, generation, MAX_GENERATIONS)
            new_generation.append(child)

        population = new_generation
        generation += 1
    print(f"Reached max generations. Best solution: Error: {best_error},
        Chromosome: {best_chromosome}")
    return 0, generation, best_error, reset_count

success_count_total = 0
total_successful_generations = 0
total_best_solution_error = 0.0
total_stopping_generations = 0
total_resets = 0
total_unsuccessful_errors = 0.0

for run in range(total_runs):
    print(f"Run {run + 1} of {total_runs}:")
    success, generations, best_solution_error, reset_count = genetic_algorithm()

    success_count_total += success
    if success > 0:
        total_successful_generations += generations
        total_best_solution_error += best_solution_error
    else:
        total_stopping_generations += generations
        total_unsuccessful_errors += best_solution_error
        total_resets += reset_count

# Vidurkiai
if success_count_total > 0:
    avg_successful_generations = total_successful_generations / success_count_total
    avg_best_solution_error = total_best_solution_error / success_count_total
else:
    avg_successful_generations = 0
```

```
    avg_best_solution_error = 0

unsuccessful_runs = total_runs - success_count_total
if unsuccessful_runs > 0:
    avg_stopping_generations = total_stopping_generations / unsuccessful_runs
    avg_minimum_error_unsuccessful = total_unsuccessful_errors / unsuccessful_runs
else:
    avg_stopping_generations = 0
    avg_minimum_error_unsuccessful = 0

# Rezultatai
print(f"\nResults after {total_runs} runs:")
print(f"Success Count: {success_count_total}")
print(f"Average Successful Generations: {avg_successful_generations}")
print(f"Average Best Solution Error (Successes): {avg_best_solution_error}")
print(f"Average Stopping Generations (Unsuccessful Runs):
    {avg_stopping_generations}")
print(f"Average Minimum Error for Unsuccessful Runs:
    {avg_minimum_error_unsuccessful}")


### Dalelių spiečiaus optimizacijos algoritmas
### naudojant skirtingas funkcijas ir skirtingus generatorius

# Pritaikoma 2 dimensijų skaičiavimams su Levy, Easom, Ackley,
# Rosenbrock, Six-Hump Camel funkcijomis
DIMENSIONS = 2

# Pritaikoma 5 dimensijų skaičiavimams su Levy, Ackley,
# Rosenbrock funkcijomis
# DIMENSIONS = 5

class Particle:
    def __init__(self):
        # Naudojama taikant Levy, Easom, Ackley ir Rosenbrock funkciją
        self.position = [random.uniform(LOWER_BOUND, UPPER_BOUND)
        for _ in range(DIMENSIONS)]
        # Naudojama taikant Six-Hump Camel funkciją
        # self.position = np.array(
        # [rng.uniform(LOWER_BOUND_X1, UPPER_BOUND_X1),
        # rng.uniform(LOWER_BOUND_X2, UPPER_BOUND_X2)])
        self.velocity = [random.uniform(-1, 1) for _ in range(DIMENSIONS)]
        self.best_position = self.position.copy()
        self.best_fitness = function(self.position)
        self.position_history = [self.position.copy()]

    def update_velocity(self, global_best_position, inertia_weight,
                        cognitive_weight=1.5, social_weight=1.5):
        cognitive_component = [cognitive_weight * random.random() * \
         (self.best_position[i] - self.position[i]) for i in range(DIMENSIONS)]
```

```
        social_component = [social_weight * random.random() * \
         (global_best_position[i] - self.position[i]) for i in range(DIMENSIONS)]
        self.velocity = [inertia_weight * self.velocity[i] + \
                         cognitive_component[i] + social_component[i]
                         for i in range(DIMENSIONS)]
        # Naudojama taikant Levy, Easom, Ackley ir Rosenbrock funkciją
        V_max = 0.2 * (UPPER_BOUND - LOWER_BOUND)
        # Naudojama taikant Six-Hump Camel funkciją
        # V_max = 0.2 * (UPPER_BOUND_X1 - LOWER_BOUND_X1)
        self.velocity = np.clip(self.velocity, -V_max, V_max)

    def update_position(self):
        self.position = [self.position[i] + self.velocity[i] \
                         for i in range(DIMENSIONS)]
        # Naudojama taikant Levy, Easom, Ackley ir Rosenbrock funkciją
        self.position = np.clip(self.position, LOWER_BOUND, UPPER_BOUND)
        # Naudojama taikant Six-Hump Camel funkciją
        # self.position = np.clip(
        # self.position, [LOWER_BOUND_X1, LOWER_BOUND_X2],
        # [UPPER_BOUND_X1, UPPER_BOUND_X2])
        self.position_history.append(self.position.copy())

    def evaluate(self):
        fitness = function(self.position)
        if fitness < self.best_fitness:
            self.best_fitness = fitness
            self.best_position = self.position.copy()

# PSO algoritmas
def pso(plot_movement=False):
    particles = [Particle() for _ in range(POPULATION_SIZE)]
    global_best_position = particles[0].best_position
    global_best_fitness = particles[0].best_fitness
    fitness_history = []

    initial_inertia_weight = 0.9
    final_inertia_weight = 0.4
    stagnation_counter = 0

    lowest_error_during_stagnation = float('inf')
    iteration_of_best_error = -1
    best_error_overall = float('inf')
    iteration_of_best_overall = -1
    iteration_of_best_fitness = -1

    for iteration in range(MAX_ITERATIONS):
        inertia_weight = initial_inertia_weight - ((initial_inertia_weight -
                                                    final_inertia_weight) *
                                                     (iteration / MAX_ITERATIONS))
```

```
        fitness_improved = False

        for particle in particles:
            particle.evaluate()
            if particle.best_fitness < global_best_fitness:
                global_best_fitness = particle.best_fitness
                global_best_position = particle.best_position
                fitness_improved = True

        current_error = abs(global_best_fitness)
        if current_error < best_error_overall:
            best_error_overall = current_error
            iteration_of_best_overall = iteration

        if current_error < lowest_error_during_stagnation:
            lowest_error_during_stagnation = current_error
            iteration_of_best_error = iteration

        for particle in particles:
            particle.update_velocity(global_best_position,
                                     inertia_weight=inertia_weight)
            particle.update_position()

        fitness_history.append(global_best_fitness)

        if fitness_improved:
            stagnation_counter = 0
        else:
            stagnation_counter += 1

        if stagnation_counter >= STAGNATION_LIMIT:
            print(f"Stagnated after {iteration} iterations with lowest error \
             from global minimum achieved: {lowest_error_during_stagnation:.12f}\
              (achieved at iteration {iteration_of_best_error })")
            break

        if abs(global_best_fitness) <= ERROR_THRESHOLD:
            break

    if plot_movement:
        plot_particle_movement(particles)

    return global_best_fitness, global_best_position, fitness_history, iteration,
     stagnation_counter, [p.position for p in particles],
     best_error_overall, iteration_of_best_error

# Atliekama paieška kelis kartus ir surandami rezultatai
def run_pso_multiple_times(num_runs=1000):
    success_count = 0
    stagnation_count = 0
```

```
total_iterations_success = 0
total_iterations_stagnation = 0
total_position_variance_success = 0
total_position_variance_stagnation = 0
total_best_fitness_success = 0
total_best_fitness_stagnation = 0
total_best_error_success = 0
total_best_error_stagnation = 0
total_best_fitness_iteration_success = 0
total_best_fitness_iteration_stagnation = 0

for _ in range(num_runs):
    global_best_fitness, global_best_position, fitness_history, iterations,
     stagnation_counter, final_positions, best_error_overall,
     iteration_of_best_error = pso(plot_movement=False)

    if abs(global_best_fitness) <= ERROR_THRESHOLD:
        success_count += 1
        total_iterations_success += iterations
        total_best_fitness_success += global_best_fitness
        total_best_error_success += best_error_overall
        total_best_fitness_iteration_success += iteration_of_best_error
        position_variance = np.var(final_positions)
        total_position_variance_success += position_variance
    elif stagnation_counter >= STAGNATION_LIMIT:
        stagnation_count += 1
        total_iterations_stagnation += iterations
        total_best_fitness_stagnation += global_best_fitness
        total_best_error_stagnation += best_error_overall
        total_best_fitness_iteration_stagnation += iteration_of_best_error
        position_variance = np.var(final_positions)
        total_position_variance_stagnation += position_variance

success_rate = (success_count / num_runs) * 100
stagnation_rate = (stagnation_count / num_runs) * 100
average_convergence_speed_success = total_iterations_success/success_count
 if success_count > 0 else 0
average_convergence_speed_stagnation = \
total_iterations_stagnation/stagnation_count
 if stagnation_count > 0 else 0
average_position_variance_success =
total_position_variance_success/success_count if success_count > 0 else 0
average_position_variance_stagnation =
total_position_variance_stagnation/stagnation_count
  if stagnation_count > 0 else 0
average_best_error_success =
total_best_error_success/success_count if success_count > 0 else 0
average_best_error_stagnation =
total_best_error_stagnation/stagnation_count if stagnation_count > 0 else 0
average_best_fitness_iteration_success =
```

```
    total_best_fitness_iteration_success/success_count if success_count > 0 else 0
    average_best_fitness_iteration_stagnation =
    total_best_fitness_iteration_stagnation/stagnation_count
     if stagnation_count > 0 else 0

    print(f"Success Rate: {success_rate:.2f}%")
    print(f"Stagnation Rate: {stagnation_rate:.2f}%")
    print(f"Average Convergence Speed (Successful Runs): \
    {average_convergence_speed_success}")
    print(f"Average Convergence Speed (Stagnation Runs): \
     {average_convergence_speed_stagnation}")
    print(f"Average Position Variance (Successful Runs): \
     {average_position_variance_success}")
    print(f"Average Position Variance (Stagnation Runs): \
     {average_position_variance_stagnation}")
    print(f"Average Best Error (Successful Runs): \
    {average_best_error_success:.12f}")
    print(f"Average Best Error (Stagnation Runs):\
     {average_best_error_stagnation:.12f}")
    print(f"Average Iteration of Best Fitness (Stagnation Runs):\
     {average_best_fitness_iteration_stagnation}")
    print(f"Total Runs: {num_runs}")
    print(f"Success Runs: {success_count}")
    print(f"Stagnation Runs: {stagnation_count}")

if __name__ == "__main__":
    run_pso_multiple_times(num_runs=1000)


### Modeliuojamas atkaitinimas
def simulated_annealing(func, bounds, global_min, trials, threshold, vector_input):
    lower_bound, upper_bound = bounds
    #Naudojamas Six-Hump Camel funkcijai
    # lower_bound_x1, upper_bound_x1 = bounds[0]
    #Naudojamas Six-Hump Camel funkcijai
    # lower_bound_x2, upper_bound_x2 = bounds[1]
    success_count = 0
    unsuccessful_count = 0
    total_successful_error = 0
    total_unsuccessful_error = 0
    total_iterations_success = 0
    total_iterations_failure = 0

    for trial in range(trials):
        solution_size = 2 if not vector_input else 5
        current_solution = [rng.uniform(lower_bound, upper_bound)
                            for _ in range(solution_size)]

        # Naudojamas six hump camel funkcijai
        # current_solution = [
```

```
#     rng.uniform(lower_bound_x1, upper_bound_x1),
#     rng.uniform(lower_bound_x2, upper_bound_x2)
# ]

current_fitness = func(current_solution)
best_solution = current_solution
best_fitness = current_fitness

temperature = 100000
cooling_rate = 0.995
perturbation_scale = 5.0
stagnation_limit = 5000
stagnation_count = 0
multi_step_perturbation = 3

iterations = 0
success_found = False

best_solution_at_threshold = None
best_fitness_at_threshold = None
best_iterations_at_threshold = 0

while temperature > 1e-7:
    for _ in range(multi_step_perturbation):
        new_solution = [
            max(min(val + rng.uniform(-perturbation_scale,
                                      perturbation_scale),
                    upper_bound), lower_bound)
            for val in current_solution
        ]
        # # Naudojamas six hump camel funkcijai
        # new_solution = [
        #     max(min(val + rng.uniform(-perturbation_scale,
        #                               perturbation_scale),
        #             upper_bound_x1 if i == 0 else upper_bound_x2),
        #         lower_bound_x1 if i == 0 else lower_bound_x2)
        #     for i, val in enumerate(current_solution)
        # ]
        new_fitness = func(new_solution)

        max_fitness_diff = 500
        acceptance_probability = math.exp(min((current_fitness -\
                                               new_fitness) /\
                                              temperature, max_fitness_diff))

        if new_fitness < best_fitness:
            best_solution = new_solution
            best_fitness = new_fitness
            stagnation_count = 0
        else:
```

```
                stagnation_count += 1

            if new_fitness < current_fitness or rng.random() <\
                  acceptance_probability:
                current_solution = new_solution
                current_fitness = new_fitness

            if abs(best_fitness - global_min) <= threshold:
                success_found = True
                best_solution_at_threshold = best_solution
                best_fitness_at_threshold = best_fitness
                best_iterations_at_threshold = iterations
                break

            if stagnation_count >= stagnation_limit:
                break

            iterations += 1

        perturbation_scale = max(perturbation_scale * 0.995, 0.01)
        temperature = max(temperature * cooling_rate, 1e-7)

    if success_found and best_solution_at_threshold is not None:
        error = abs(best_fitness_at_threshold - global_min)
        success_count += 1
        total_successful_error += error
        total_iterations_success += best_iterations_at_threshold
        print(f"\nTrial {trial + 1}: Success, Coordinates:
            {best_solution_at_threshold},
            Error: {error}, Iterations: {best_iterations_at_threshold}")
    else:
        unsuccessful_count += 1
        total_unsuccessful_error += abs(best_fitness - global_min)
        total_iterations_failure += iterations
        print(f"\nTrial {trial + 1}: Failure, Coordinates: {best_solution},
              Error: {abs(best_fitness - global_min)}, Iterations: {iterations}")

avg_iterations_success = total_iterations_success /\
      success_count if success_count > 0 else 0
avg_iterations_failure = total_iterations_failure /\
      unsuccessful_count if unsuccessful_count > 0 else 0
avg_success_error = total_successful_error /\
      success_count if success_count > 0 else 0
avg_failure_error = total_unsuccessful_error /\
      unsuccessful_count if unsuccessful_count > 0 else 0

print(f"\nResults after {trials} trial(s):")
print(f"Success Count: {success_count}")
print(f"Failure Count: {unsuccessful_count}")
print(f"Average Success Error: {avg_success_error}")
```

```
    print(f"Average Failure Error: {avg_failure_error}")
    print(f"Average Iterations for Success: {avg_iterations_success}")
    print(f"Average Iterations for Failure: {avg_iterations_failure}")

simulated_annealing(
    func=ackley_function,
    bounds=(-32.768, 32.768),
    global_min=0,
    threshold=1e-8,
    trials=1000,
    vector_input=False # False jei 2D, True jei 5D
)



### Modeliuojamas ataitinimas su Ackley funkcija

def simulated_annealing_pcg(func, bounds, global_min, trials, threshold, dimensions):
    (lower_bound, upper_bound) = bounds
    success_count = 0

    rng = np.random.default_rng(np.random.PCG64())
    unsuccessful_count = 0
    total_successful_error = 0
    total_unsuccessful_error = 0
    total_iterations_success = 0
    total_iterations_failure = 0

    for trial in range(trials):
        current_solution = [rng.uniform(lower_bound, upper_bound)
                            for _ in range(dimensions)]
        current_fitness = func(current_solution)
        best_solution = current_solution
        best_fitness = current_fitness

        temperature = 100000
        cooling_rate = 0.995
        perturbation_scale = 5.0
        stagnation_limit = 5000
        stagnation_count = 0
        best_solution_at_threshold = None
        best_fitness_at_threshold = None
        best_iterations_at_threshold = 0
        success_found = False
        iterations = 0

        while temperature > 1e-7:
            new_solution = [
                max(min(current_solution[i] + rng.uniform(-perturbation_scale,
                                                          perturbation_scale),
```

```
                                                upper_bound),
                                                lower_bound)
            for i in range(dimensions)
        ]
        new_fitness = func(new_solution)

        delta = current_fitness - new_fitness
        acceptance_probability = math.exp(delta / (temperature * 1e-2)
                                          )if delta < 0 else 1.0

        if new_fitness < current_fitness or rng.random() < acceptance_probability:
            current_solution = new_solution
            current_fitness = new_fitness
            stagnation_count = 0
        else:
            stagnation_count += 1

        if new_fitness < best_fitness:
            best_solution = new_solution
            best_fitness = new_fitness

        if abs(best_fitness - global_min) <= threshold:
            best_solution_at_threshold = best_solution
            best_fitness_at_threshold = best_fitness
            best_iterations_at_threshold = iterations
            success_found = True
            break

        # Reduce perturbation scale and cool the temperature
        perturbation_scale = max(1e-6, temperature * 0.01)
        temperature *= cooling_rate

        # Stagnation reset
        if stagnation_count >= stagnation_limit:
            current_solution = [rng.uniform(lower_bound, upper_bound)
                                for _ in range(dimensions)]
            current_fitness = func(current_solution)
            temperature = 100000
            perturbation_scale = 5.0
            stagnation_count = 0

        iterations += 1

    if success_found and best_solution_at_threshold is not None:
        error = abs(best_fitness_at_threshold - global_min)
        success_count += 1
        total_successful_error += error
        total_iterations_success += best_iterations_at_threshold
        print(f"\nTrial {trial + 1}: Success, Coordinates:
            {best_solution_at_threshold},
```

```
                    Error: {error}, Iterations: {best_iterations_at_threshold}")
            else:
                unsuccessful_count += 1
                total_unsuccessful_error += abs(best_fitness - global_min)
                total_iterations_failure += iterations
                print(f"\nTrial {trial + 1}: Failure, Coordinates:
                    {best_solution},
                    Error: {abs(best_fitness - global_min)}, Iterations: {iterations}")

    # Final results
    avg_iterations_success = total_iterations_success /\
          success_count if success_count > 0 else 0
    avg_iterations_failure = total_iterations_failure /\
          unsuccessful_count if unsuccessful_count > 0 else 0
    avg_success_error = total_successful_error /\
          success_count if success_count > 0 else 0
    avg_failure_error = total_unsuccessful_error /\
          unsuccessful_count if unsuccessful_count > 0 else 0

    print(f"\nResults after {trials} trial(s):")
    print(f"Success Count: {success_count}")
    print(f"Failure Count: {unsuccessful_count}")
    print(f"Average Success Error: {avg_success_error}")
    print(f"Average Failure Error: {avg_failure_error}")
    print(f"Average Iterations for Success: {avg_iterations_success}")
    print(f"Average Iterations for Failure: {avg_iterations_failure}")

bounds = (-32.768, 32.768)
global_min = 0

simulated_annealing_pcg(
    func=ackley_function,
    bounds=bounds,
    global_min=global_min,
    dimensions = 5, #5 arba 2
    trials=1000
)
```