UNIVERZITET U NOVOM SADU
FAKULTET TEHNIČKIH NAUKA U NOVOM SADU

Dušan Stević

ANALIZA SERIJSKE I PARALELNE IMPLEMENTACIJE ALGORITAMA BAZIRANIH NA MONTE KARLO METODI

DIPLOMSKI RAD
- Osnovne akademske studije -

Novi Sad, 2020.

	УНИВЕРЗИТЕТ У НОВОМ САДУ ● **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА** 21000 НОВИ САД, Трг Доситеја Обрадовића 6	Датум:
	ЗАДАТАК ЗА ИЗРАДУ ДИПЛОМСКОГ (BACHELOR) РАДА	Лист/Листова: 1/1

(Податке уноси предметни наставник - ментор)

Врста студија:	☒ Основне академске студије
Студијски програм:	**Софтверско инжењерство и информационе технологије**
Руководилац студијског програма:	**Проф. др Мирослав Зарић**

Студент:	**Душан Стевић**	Број индекса:	**SW 10/2016**
Област:	**Примењене рачунарске науке и информатика**		
Ментор:	**др Игор Дејановић**		

НА ОСНОВУ ПОДНЕТЕ ПРИЈАВЕ, ПРИЛОЖЕНЕ ДОКУМЕНТАЦИЈЕ И ОДРЕДБИ СТАТУТА ФАКУЛТЕТА ИЗДАЈЕ СЕ ЗАДАТАК ЗА ДИПЛОМСКИ (Bachelor) РАД, СА СЛЕДЕЋИМ ЕЛЕМЕНТИМА:

- проблем – тема рада;
- начин решавања проблема и начин практичне провере резултата рада, ако је таква провера неопходна;
- литература

НАСЛОВ ДИПЛОМСКОГ (BACHELOR) РАДА:

АНАЛИЗА СЕРИЈСКЕ И ПАРАЛЕЛНЕ ИМПЛЕМЕНТАЦИЈЕ АЛГОРИТАМА БАЗИРАНИХ НА МОНТЕ КАРЛО МЕТОДИ

ТЕКСТ ЗАДАТКА:

Истражити Монте Карло методу, њене серијске и паралелне имплементације, и демонстрирати њену примену у решавању проблема у области софтверског инжењерства. Приказати употребу на проблемима апроксимације броја пи, предвиђање кретања вредности финансијске активе и израчунавање апроксимиране вредности одређеног интеграла.

Имплементирати серијска и паралелна решења у програмским језицима Go и Python и упоредити њихове перформансе. Обавити анализу јаког и слабог скалирања. Детаљано описати имплементацију и обавити анализу резултата. Податке визуализовати употребом програмског језика Pharo и библиотеке Roassal.

Руководилац студијског програма:	Ментор рада:

Примерак за: ☐ - Студента; ☐ - Ментора

Sadržaj

1. Uvod

Monte Karlo simulacija [1] spada u kategoriju probabilističkih aproksimativnih tehnika za simulaciju i predikciju. Fundus Monte Karlo simulacije čini stohastički pristup. Stohastički pristup, za razliku od determinističkog pristupa prilikom ponovnog pokretanja simulacija uvek daje nove rezultate koji se razlikuju od pređašnjih rezultata. U slučaju determinističkih algoritama rezultati simulacija su uvek isti. Na ovaj način formira se portfelj rešenja iz kojeg je potrebno izabrati najbolje za posmatrani problem. Stohastički pristup Monte Karlo simulacije ogleda se u činjenici da se za rešavanje problema iz određenog domena koriste pseudo-slučajni brojevi [2]. Pseudo-slučajni brojevi obezbeđuju stohastičnost modela. Monte Karlo simulacija nastoji da modeluje neizvesnost korišćenjem pseudo-slučajnih brojeva. Kao takva Monte Karlo simulacija ima širok dijapazon primene.

Osnovni motiv za pisanje ovog rada ogleda se u nastojanju da se na sistematičan način da pregled primene Monte Karlo simulacije u različitim domenima. Visoka aplikativnost modela omogućuje primenu u najrazličitijim sferama života. Struktura rada je koncipirana tako da prati tri dominantne oblasti primene Monte Karlo simulacije:

1. Izračunavanje aproksimirane vrednosti broja Pi.
2. Izračunavanje aproksimirane vrednosti finansijske aktive [3].
3. Izračunavanje aproksimirane vrednosti određenog integrala.

Tema ovog rada je razvoj serijske i paralelne verzije softverskog rešenja koje će omogućiti simulaciju izračunavanje aproksimirane vrednosti broja Pi, simulaciju izračunavanja aproksimirane vrednosti finansijske aktive i simulaciju izračunavanje aproksimirane vrednosti određenog integrala. Serijske i paralelne verzije programa ostvarene su u Python [4] i Golang [5] programskim jezicima. Pored razvoja serijske i paralelne verzije programa posebna pažnja je posvećena i vizuelizaciji dobijenih rešenja. Vizuelizacija rešenja zasnovana je na programskom jeziku Pharo [6] uz korišćenje graphic engine-a Roassal [7]. Takođe veoma važnu stavku u radu čine i eksperimenti skaliranja [8]. Kako je reč o radu koji obuhvata paralelno programiranje i računarstvo visokih performansi [9] bilo je potrebno sprovesti detaljne eksperimente kako bi se utvrdio uticaj broja procesnih jedinica i veličine posla na ubrzanje paralelnog programa u odnosu na serijski program. Eksperimenti skaliranja obuhvataju eksperimente jakog [10] i slabog skaliranja [11]. Cilj eksperimenata jakog i slabog skaliranja je da pokažemo kako se ovi algoritmi ponašanja na stvanom hardveru.

Cilj rada je da se pokaže kako Monte Karlo simulacija može da se primenjuje u različitim oblastima uz upotrebu naprednih tehnika programiranja. Pored razvoja samog programskog rešnja (serijska verzija programa), potrebno je razviti i efikasniju verziju programskog rešenja (paralelena verzija programa) koja će u potpunosti iskoristiti hardverski potencijal mašne na kojoj se pokreće. Benefiti paralelnog pristupa rešavanju problema možemo da posmatramo iz dva ugla:

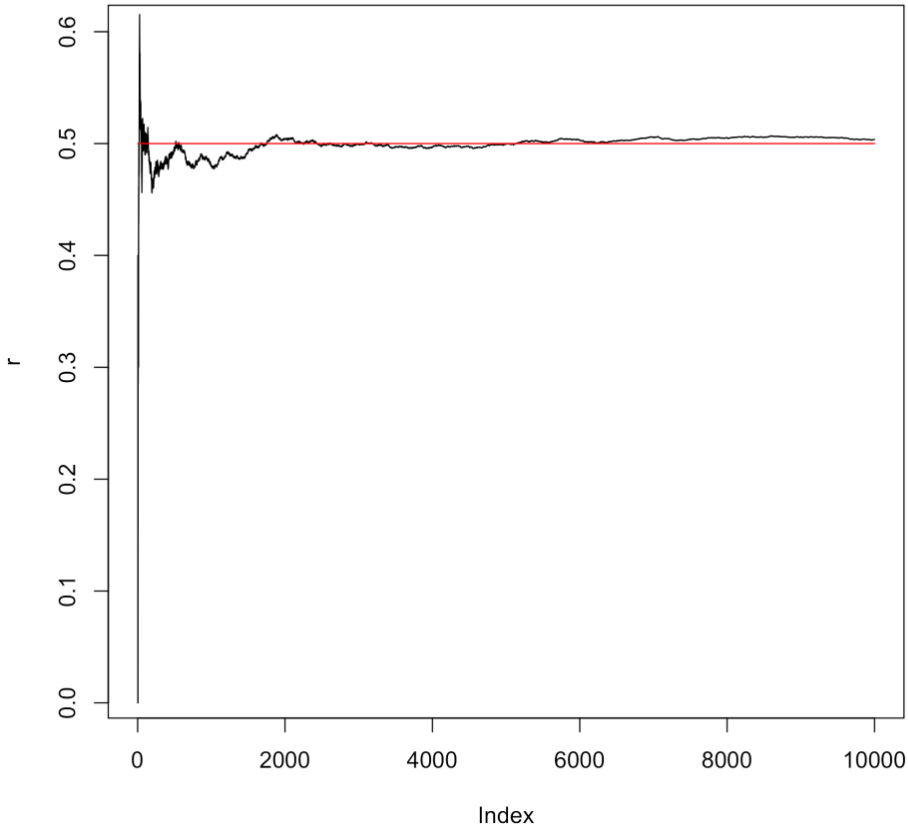
1. Porastom broja simulacija raste kvalitet rešenja. Osnovna motivacija za paralelni pristup je činjenica da kvalitet rešenja raste sa porastom broja simulacija. Porast broja simulacija koje se izvršavaju u paraleli dovode do približnije vrednost broja Pi, približnije vrednosti određenog integrala i većeg spektara predikcija kretanja cena finansijske aktive što omogućva donošenje bolje investicione odluke.
2. Brži dolazak do rešenja. Svaka simulacija zahteva određeno vreme izvršavanja. U sistemima u kojima vreme igra krucijalnu ulogu (berzansko i finansijsko poslovanje) odluke moraju da se donose u veoma kratkom vremenu. Ukoliko bi se koristio samo serijski program donošenje odluka bi jako dugo trajalo, a odluke na berzi moraju da se donesu u deliću sekunde pa je paralelno izvršavanje pogodno iz razloga što za kraće vreme dobijamo kontingent investicionih odluka od kojih treba da izaberemo najbolju.

Ovako implementirano rešenje ima više dimenzija. Prva dimenzija rešnja bi predstavljala oblast primene Monte Karlo simulacije (Pi, finansije, integrali). Druga dimenzija rešenja bi predstavljala tehnologija realizacije programskog rešenja (Python, Golang, Pharo). Dok bi treća dimenzija programskog rešenja predstavljala način izvršavanja programskog rešenja (serijsko, paralelno).

U nastavku rada biće analizirani osnovni teorijski koncepti Monte Karlo simulacije, gde će se nastojati da se pored opštih teorijskih koncepata detaljno prikažu teorijski koncepti Monte Karlo simulacije po oblastima primene. Biće opisane korišćene tehnike i tehnologije za implementaciju sistema. Predstaviće se razvojni put softverskog rešenja od specifikacije softverskog rešnja preko implementacije do vizuelizacije i verifikacije istog.

2. Teorijski koncepti primene Monte Karlo simulacije

Monte Karlo simulacija spada u kategoriju probabilističkih aproksimativnih tehnika za simulaciju i predikciju. Nastala je 50-ih godina prošlog veka tokom rada na Menhetn projekatu (engl. The Manhattan Project) [12]. Monte Karlo simulacija se razvila za potrebe simuliranja razornog dejstva hidrogenske bombe. Na razvoju Monte Karlo simulacije radili su čuveni naučnici: Enrico Fermi [13], John von Neumann [14], Nicholas Metropolis [15] i Stanislaw Ulam [16]. Naglu ekspanziju Monte Karlo simulacija doživljava sa ubrzanim razvojem računara. Bolje hardverske performanse savremenih računara doprinele su da se veći broj simulacija može odvijati u paraleli za kraće vreme. Na ovaj način upotrebom Monte Karlo simulacije se dolazi brže do preciznijih rezultata. Monte Karlo simulacija je zasnovana na teoriji velikih brojeva [17] i teoriji slučajnog uzorkovanja [18]. Osnovna ideja koja se krije iza Monte Karlo simulacije je da sa porastom broja simulacija eksperimentalna verovatnoća se približava (konvergira) teorijskoj verovatnoći (npr. kada bacamo novčić ako imamo jako puno pokušaja bacanja onda će odnos pismo glava biti pibližno 50% : 50%).



Slika 1 Teorija velikih brojeva na primeru bacanja novčića [19]

Na Slika 1 vidimo teoriju velikih brojeva na primeru bacanja novčića. Na x-osi nalazi se broj bacanja novčića, dok se na y-osi nalazi verovatnoća ishoda bacanja novčića. Crna krivudava linija predstavlja eksperimentalnu verovatnoću, a crvena prava linija predstavlja teorijsku verovatnoću. Prilikom bacanja novčića teorijska verovatnoća iznosi 50%, odnosno 0.5. Jednake su šanse prilikom svakog bacanja da padne ili pismo ili glava stoga je teorijska verovatnoća 0.5. Tokom sprovođenja eksperimenta bacanja novčića može se desiti da jedna strana novčića bude učestalija u odnosu na drugu što se ogleda u volatilnoj crnoj liniji. Ako je broj eksperimenta tj. bacanja novčića suviše mali odstupanje eksperimentalne verovatnoće od teorijske verovatnoće je najveće. Kako povećavamo broj bacanja crna linija se uprosečava (postaje sve ravnija i ravnija) tj.eksperimentalna verovatnoća konvergira teorijskoj verovatnoći. Ako je broj bacanja dovoljno velik odstupanje eksperimentalne verovatnoće od teorijske je minimalno. Zakon velikih brojava obezbeđuje da sa porastom broja simulacija raste kvalitet rešenja odnosno preciznost rešenja. Mali broj simulacija nije dovoljno reprezentativan da bi se mogli doneti neki opši zaključci stoga je potrebno povećati broj simulacija. Broj simulacija ograničen je sa donje strane veličinom reprezentativnog uzorka. Veličina reprezentativnog uzorka varira od slučaja do slučaja i najčešće se utvrđuje na empirijski način. Sa druge strane broj simulacija je ograničen sa gornje strane hardverskim performansama računara na kojem se program izvršava. Jako velik broj simulacija dovešće da predugog izvršavanja programa i preopterećenja celokupnog sistema. Stoga se mora utvrditi balans između nedovoljno reprezentativnog broja simulacija i suviše velikog broja simulacija. Kao potencijalno rešenje ovog

problema uvodi se paralelno programiranje. Paralelno programiranje omogućava da se za isto vreme dobije veći broj simulacija i da se na taj način formira reprezentativna baza simulacija koja nije suviše opterećujuća za celokupni sistem. Paralelnim programiranjem postigla bi se dva veoma važna cilja prilikom sprovođenja simulacija: veća preciznost i brži dolazak do rešenja.

2.1.Sistematičan pregled najvažnijih osobina Monte Karlo simulacije

Radi boljeg razumevanja sledećih poglavlja u radu potrebno je na sistematičan način dati pregled najvažnijih osobina Monte Karlo simulacije:

2.1.1. **Monte Karlo simulacija je aproksimativna tehnika:** Monte Karlo simulacija daje prednost brzini izračunavanja nad preciznošću izračunavanja. Nedovoljna preciznost pojedinačnih simulacija se kompenzuje velikim brojem simulacija koje dovode do porasta preciznosti ukupnog rešenja. Pojedinačne simulacije su brze ali neprecizne zbog toga se sprovodi veći broj simulacija koje obezbeđuju da se uprosečvanjem dobije aproksimativnija vrednost rešnja.

2.1.2. **Monte Karlo simulacija je probabilistička tehnika:** Monte Karlo simulacija zasnovana je na teoriji verovatnoće. Sa porastom broja simulacija eksperimentalna verovatnoća konvergira ka teorijskoj verovatnoći.

2.1.3. **Monte Karlo simulacija koristi stohastički pristup:** Monte Karlo simulacija koristi pseudo-slučajne brojeve [2] za rešavanje različitih problema. Stohastički pristup nasuprot determinističkom pristupu prilikom svakog ponovnog pokretanja algoritama daje nove rezultate. Generisanje uvek različitih rezultata doprinosi da u bazi simulacija budu disperzovane vrednosti. Disperzovane vrednosti obezbeđuju veću preciznost konačnih rezultata simulacija. Ukoliko bi algoritmi uvek vraćali istu vrednost baza simulacije bi se sastojala od samo jedne vrednosti ponovljene nekoliko puta. Agregirani rezultat simulacije se dobija uprosečavanjem parcijalnih rezultata simulacije. Prilikom uprosečavanja parcijalnih rezultata simulacija agregirani rezultat se ne bi razlikovao od parcijalnih rezultata. Parcijalni i agregirani rezultati bi bili isti čime bi se smanjila sposobnost Monte Karlo modela da oslika neizvesnosti i slučajnost.

2.1.4. **Monte Karlo simulacija koristi pseudo-slučajne brojeve:** Monte Karlo simulacija koristi pseudo-slučajne brojeve, a ne slučajne brojeve [20]. U simulacijama se koriste ugrađeni generatori pseudo-slučajnih brojeva. U programskim jezicima koji se koriste za implementaciju simulacija postoje ugrađeni generatori pseudo-slučajnih brojeva. Osnovna razlika između slučajnih i pseudo-slučajnih brojeva je način njihovog generisanja. Slučni brojevi nastaju korišćenjem nekog fizičkog sredstva (npr. kockice, novčić, rulet, točak sreće, bubanj, itd.). Za slučajne brojeve ne postoji algoritam koji ih generiše, dok kod pseudo-slučajnih brojeva postoji algoritam. Pošto možemo imati uvid u algoritam koji generiše pseudo-slučajne brojeve njihov nastanak nije u potpunosti slučajan pa ih iz tog razloga zovemo pseudo-slučajnim.

2.1.5. **Monte Karlo simulacija koristi teoriju velikih brojeva:** Monte Karlo simulacija je zasnovana na teoriji velikih brojeva. U matematičkoj teoriji sa porastom broja simulacija eksperimentalna verovatnoća konvergira teorijskoj verovatnoći pa imamo osnov da broj simulacija povećavamo do beskonačnosti. Međutim u stvarnom svetu, gde postoje hardverska ograničenja, broj simulacija je ograničen hardverskim performansama. Velik broj simulacija dovodi do preopterećenja sistema i predugog vremena izvršavanja te je iz tog razloga potrebno eksperimentalno utvrditi gornju granicu broja simulacija.

2.1.6. **Monte Karlo simulacija koristi teoriju slučajnog uzorkovanja:** Monte Karlo simulacija je zasnovana na teoriji slučajnog uzorkovanja. Algoritmi Monte Karlo simulacije zahtevaju formiranje slučajnog uzorka. Za potrebe ovog rada uzorak se formira na slučajan način i obuhvata tačke iz nekog zadatog intervala. Veličina uzorka jednaka je broju simulacija. Uzorak mora da bude reprezentativan i adekvatan [21]. Reprezentativnost uzorka ogleda se u činjenici da tačke u uzorku moraju da budu ravnomerno distribuirane nad čitavim intervalom posmatranja. Ne sme da se dogodi da dođe do nagomilavanja tačaka unutar zadatog intervala. Ukoliko dođe do nagomilavanja tačaka unutar zadatog intervala uzorak neće biti reprezentativan biće pristrasan zato što će forsirati određene tačke iz intervala. Problem pristrasnosti uzorka direktno se povezuje sa lošim performansama generator pseudo-slučajnih brojeva. Pored reprezentativnosti uzorka važna osobina uzorka je i njegova adekvatnost. Adekvatnost uzorka povezana je sa veličinom uzorka. Uzorak je adekvatan ako je dovoljno velik. Veličina uzorka tj. broj simulacija ne sme da bude suviše mali. Ako je broj simulacija suviše mali preciznost rešenja je mala. Potrebno je eksperimentalno utvrditi donju granicu broja simulacija. Monte Karlo simulacija zahteva da uzorak poseduje tri osobine: slučajnost, reprezentativnost i adekvatnost.

2.1.7. **Aplikativnost Monte Karlo simulacije:** Monte Karlo simulacija se koristi za rešavanje širokog spektra problema. Numerička matematika, fizička hemija, operaciona istraživanja,

veštačka inteligencija, finansije samo su neka od područja primene Monte Karlo simulacije [1]. U ovom radu Monte Karlo simulacija će se koristiti za: izračunavanje aproksimirane vrednosti broja Pi, izračunavanje aproksimirane vrednosti finansijske aktive i izračunavanje aproksimirane vrednosti određenog integral.

2.2.Teorijski koncepti primene Monte Karlo simulacije za izračunavanje aproksimirane vrednosti broja Pi

Monte Karlo simulacija se koristi za izračunavanje aproksimirane vrednosti broja Pi. Pođimo od pretpostavke da imamo kvadrat stranice a = 2 sa centrom u tački (0,0). Površina kvadrata se računa po sledećoj formuli:

$$P_{\text{kvadrata}} = a^2 = 2^2 = 4$$

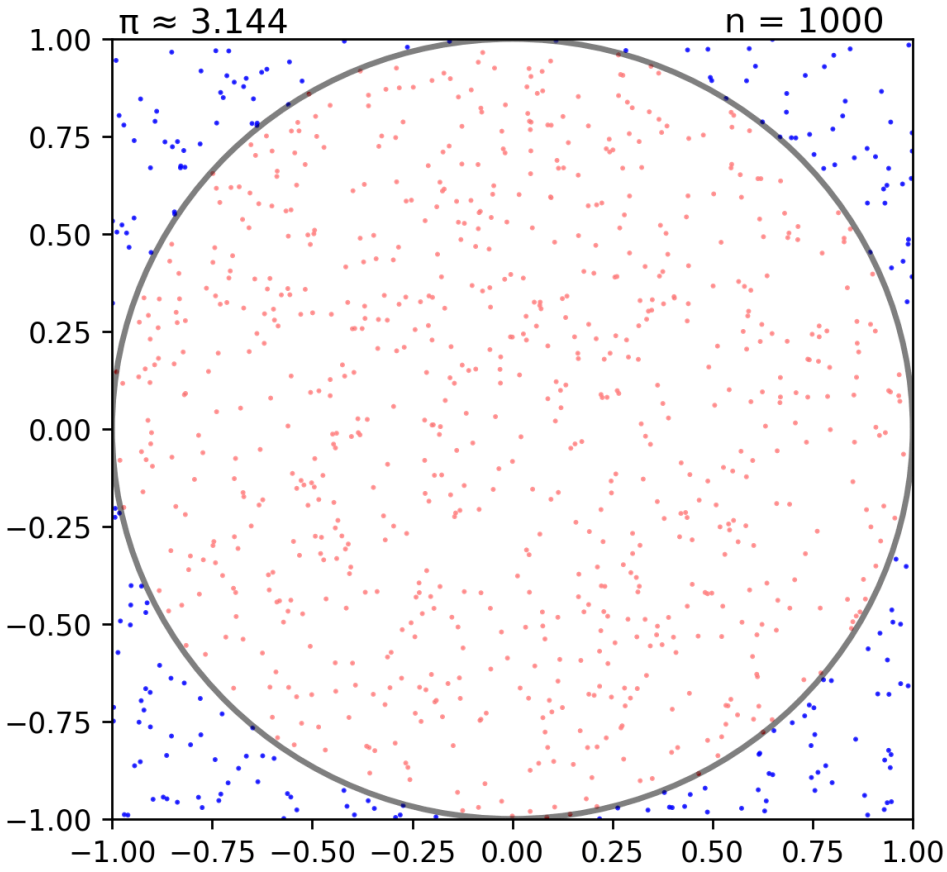
Formula 1 Površina kvadrata

U kvadrat upisujemo krug sa poluprečnikom r = 1 i centrom u tački (0,0). Površina kruga se računa po sledećoj formuli:

$$P_{\text{kruga}} = r^2\pi = 1^2\pi = \pi$$

Formula 2 Površina kruga

U kvadrat se upisuje jedinična kružnica. Centar kvadrata se poklapa sa centrom jedinične kružnice i on se nalazi u tački (0,0).



Slika 2 Monte Karlo simulacija za izračunavanje aproksimirane vrednosti broja Pi

Na Slika 2 vidimo kvadrat u koji je upisana jedinična kružnica. Crvene tačke predstavljaju tačke koje su upale u krug, dok plave tačke predstavljaju tačke koje su izvan površine kruga ali su upale na površinu kvadrata. Zbir crvenih i plavih tačaka jednak je ukupnom broju tačaka tj. ukupnom broju simulacija.

Formula na osnovu koje se utvrđuje da li neka pseudo-slučjno generisana tačka sa koordinatama x i y pripada površini kruga se računa na sledeći način:

$$(x - p)^2 + (y - q)^2 = r^2$$

Formula 3 Jednačina kružnice sa centrom u tački C(p,q) i poluprečnikom r

Kako je reč o jediničnoj kružnici sa poluprečnikom r = 1 i centrom u tački (p=0,q=0) prethodna formula može da se koriguje:

$$x^2 + y^2 = 1$$

Formula 4 Jednačina jedinične kružnice

Pomoću jednačine jedinične kružnice izvode se sledeći zaključci:

1. $x^2 + y^2 < 1$, tačka se nalazi unutar jedinične kružnice
2. $x^2 + y^2 = 1$, tačka se nalazi na jediničnoj kružnici
3. $x^2 + y^2 > 1$, tačka se nalazi van jedinične kružnice

Odnos površine kruga i površine kvadrata iznosi:

$$\frac{P_{kruga}}{P_{kvadrata}} = \frac{\pi}{4}$$

Formula 5 Odnos površine kruga i površine kvadrata

Ovaj odnos površina približno možemo da dobijemo ako površinu kvadrata "izbombardujemo" pseudo-slučajno generisanim tačkama i onda napravimo odnos tačke koje su upale u krug prema ukupnom broju tačaka.

$$\frac{P_{kruga}}{P_{kvadrata}} \approx \frac{\text{broj tačaka u krugu}}{\text{ukupan broj tačaka}} \approx \frac{\text{broj tačaka u krugu}}{\text{ukupan broj simulacija}} \approx \frac{\pi}{4}$$

Formula 6 Odnos broja tačaka u krugu i ukupnog broja tačaka

Množenjem prethodnog odnosa sa 4 dobija se približna vrednost broja π koja je sve tačnija i tačnija kako povećavamo broj simulacija.

$$\pi \approx 4\frac{\text{broj tačaka u krugu}}{\text{ukupan broj tačaka}} \approx 4\frac{\text{broj tačaka u krugu}}{\text{ukupan broj simulacija}}$$

Formula 7 Izračunavanje aproksimirane vrednosti broja Pi

2.3.Teorijski koncepti primene Monte Karlo simulacije za izračunavanje aproksimirane vrednosti finansijske aktive

Monte Karlo simulacija se koristi za izračunavanje aproksimirane vrednosti finansijske aktive [3]. Finansijska aktiva obuhvata: akcije [22], obveznice [23], finansijske derivate [24], kriptovalute [25], itd. Monte Karlo simulacija omogućava da se na osnovu pređašnjeg kretanja cene finansijske aktive predvidi buduća cena finansijske aktive. Monte Karlo simulacija se u finansijama koristi kao prediktivna tehnika. Analiza i predviđanje finansijskih vremenskih serija [26] može se vršiti primenom Monte Karlo simulacije. Kako bi racionalni donosioci odluka (engl. Rational decision makers) [27] doneli ispravnu investicionu odluku u pogledu kupovine ili prodaje finansijske aktive neophodno je da izvrše korektnu predikciju cene finansijske aktive. Monte Carlo simulacija može da pomogne u predikciji cene finansijske aktive. Povlačenjem podatke o kretanju cena finansijske aktive sa berze dolazimo do istorijskih podataka. Buduća cena finansijske aktive može da se izračuna preko formule za eksponencijalni rast [28] koja uključuje volatilnost [29] i kontinuelnu stopu prinosa [30]. Volatilnost i kontinuelna stopa prinosa se računaju na osnovu istorijskih podataka.

$$\text{Periodic Daily Return} = \ln\left(\frac{\text{Day's Price}}{\text{Previous Day's Price}}\right)$$

Formula 8 Kontinualna stopa prinosa [31]

Formula za kontinualnu stopu prinosa se koristi kako bi se izračunao dnevni prinos finansijske aktive. Kontinualna stopa prinosa predstavlja signal investitorima koji treba da ukaže da li je isplativo ulagati u neku finansijsku aktivu. Pozitivna kontinualna stopa prinosa signalizira investitorima da ukoliko poseduju finansijsku aktivu u svom investicionom portfoliju da će ostvariti profit. Negativna kontinualna stopa prinosa ukazuje da ukoliko je finansijska aktiva prisutna u portfoliju investitor će ostvariti gubitak. Pored kontinualne stope prinosa na investitora prilikom kupovine ili prodaje finansijske aktive utiče i finansijski rizik koji se modeluje po sledećoj formuli:

$$\text{Drift} = \text{Average Daily Return} - \frac{\text{Variance}}{2}$$

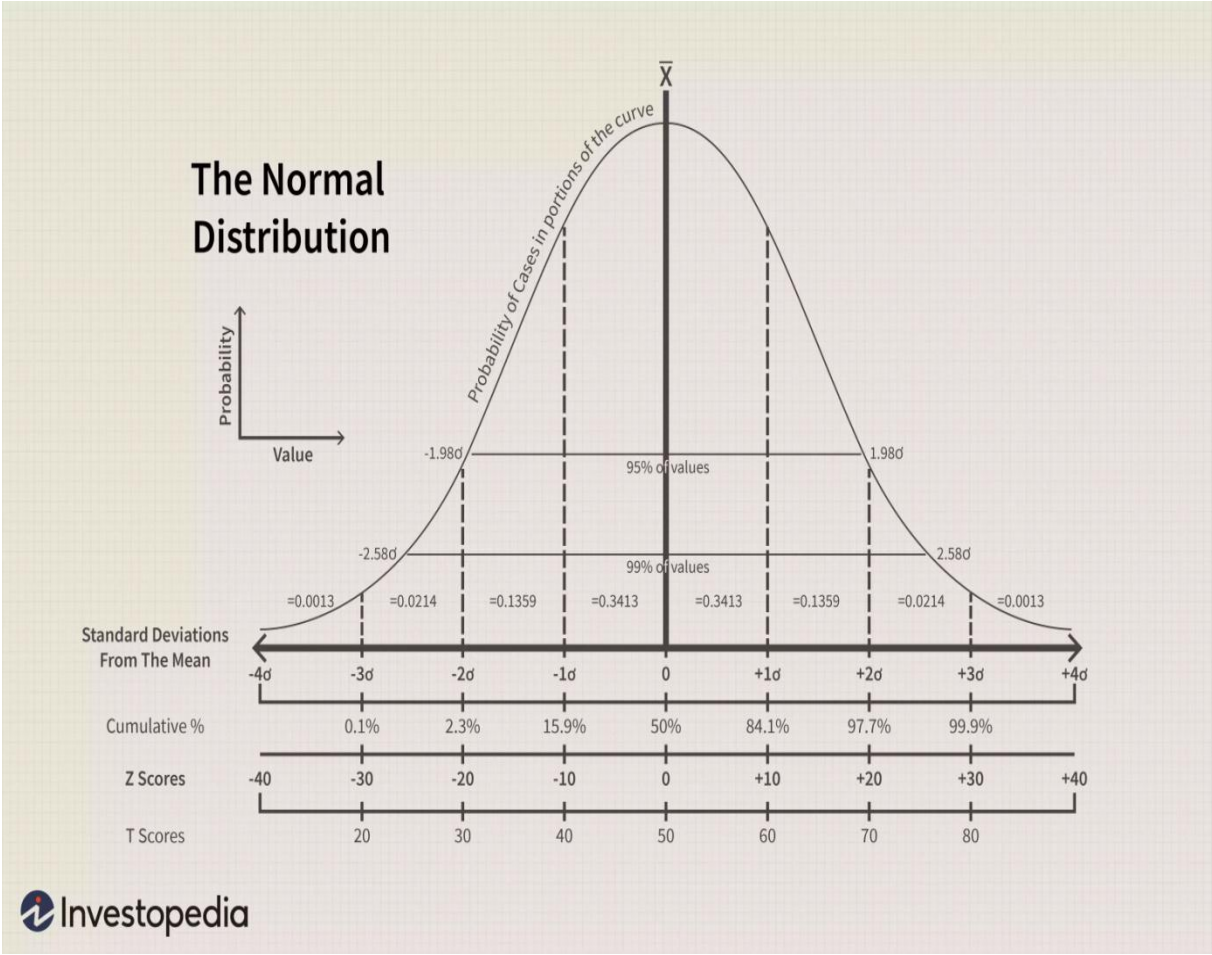
Formula 9 Finansijski drift kao mera rizika [31]

Formula za finansijski drift predstavlja meru rizika koju sa sobom nosi svako investiciono ulaganje. Racionalni investitor mora da napravi balans (engl. Trade-off) između prinosa i rizika. Više rizična finansijska aktiva ispraćena je višim stopama prinosa, dok manje rizična finansijska aktiva donosi niži prinos. Razlog ove zakonitosti leži u činjenici da investitori koji su skloni riziku moraju biti nagrađeni dok riziko-averzni investitori ne mogu da ostvaruju visoke stope prinosa.

$$\text{Random Value} = \sigma \times Z_{\text{score}}(\text{Rand}())$$

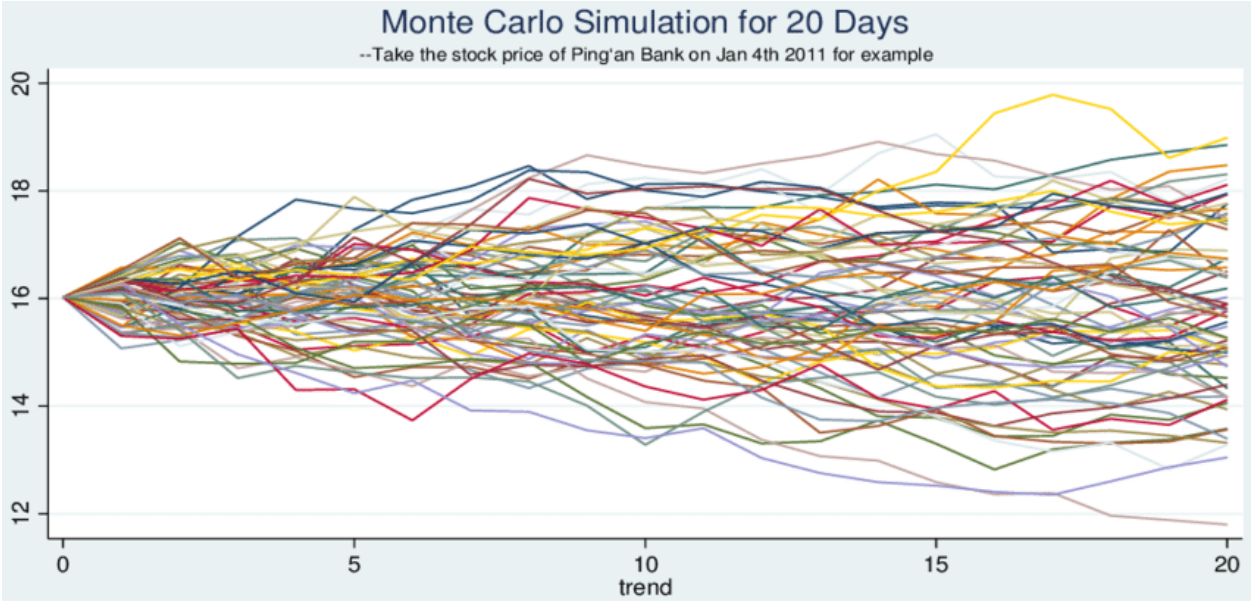
Formula 10 Slučajna vrednost [31]

Prethodnom formulom se modeluje neizvesnost u kretanju vremenskih serija. Tržišne perturbacije i volatilnost finansijskih vremenskih serija predstavljene su formulom za slučajnu vrednost. Standardna devijacija σ nam govori koliko vremenska serija osciluje tj. koliko je vremenska serija volatilna u odnosu na prosek vremenske serije. Standardna devijacija je mera disperzije vremenske serije [32]. Disperzija je pokazatelj raspršenosti vremenske serije. Varijansa predstavlja kvadrat standardne devijacije [33]. Z_{score} govori koliko smo standardnih devijacija udaljeni od proseka [34]. Ova mera je od izuzetne važnosti zato što direktno utiče na odabir one simulacije koja će se koristiti za finansijsko modelovanje. Pomoću Z_{score} mere formiraju se koridori standardnih devijacija. Sa što većim brojem simulacija dobijamo familiju krivih koje formiraju zvonastu (normalnu) distibuciju. Svaka kriva predstavlja jednu simulaciju tj. predikciju buduće cene finansijske aktive. Sa velikim brojem simulacija imamo aparat sa kojim možemo sa određenom verovatnoćom da tvrdimo kako će se cena finansijske aktive kretati u budućnosti.



Slika 3 Normalna distribucija i koridori standardnih devijacija [35]

Na Slika 3 vidimo zvonastu krivu normalne distribucije. Na x-osi se nalaze vrednosti Z_{score}, a na y-osi se nalazi verovatnoća. Simulacije formiraju fen dijagram [36] gde vidimo koja od simulacija je najverovatnija da se desi, a koje su manje verovatne da se dese. Najverovatnije predikcije su one koje se nalaze u koridoru +/- jedna standardna devijacija (σ), dok su manje verovatne predikcije koje se nalaze u koridoru +/- dve standardne devijacije (2σ) i +/- tri standardne devijacije (3σ).



Slika 4 Fen dijagram simulacija kretanja cene finansijske aktive [31]

Fen dijagram simulacija treba da posluži kao sredstvo investitorima da donesu investicionu odluku. Investitori se opredeljuju na osnovu fen dijagrama za najverovatnije kretanje cene finansijske aktive i pomoću tih predikcija formiraju svoje investicione strategije. Fen dijagram obuhvata predikcije koje se generišu na osnovu formule za buduću cenu finansijske aktive.

$$\text{Next Day's Price} = \text{Today's Price} \times e^{(\text{Drift}+\text{Random Value})}$$

Formula 11 Buduća cena finansijske aktive

2.4. Teorijski koncepti primene Monte Karlo simulacije za izračunavanje aproksimirane vrednosti određenog integrala

Monte Karlo simulacija se koristi za izračunavanje aproksimirane vrednosti određenog integrala. Pođimo od pretpostavke da imamo integrand:

$$f(x) = y = 2x$$

Formula 12 Integrand

Integrand je funkcija čiji određeni integral tražmo na nekom intervalu [37]. Uzmimo da želimo da nađemo određeni integral integranda na zatvorenom intervalu [1,2].

$$\int_a^b f(x)\,dx = \int_1^2 2x\,dx$$

Formula 13 Traženi određeni integral

Analitičko rešenje određenog integrala dobijamo primenom pravila za neodređeni i određeni integral [38]. Prilikom analitičkog rešavanja prvo je potrebno naći neodređen integral koristeći formulu i tablične vrednosti [39]:

$$F(x) = \int f(x)\,dx = \int 2x\,dx = x^2 + C$$

Formula 14 Neodređeni integral [38]

Kada smo izračunali neodređeni integral, pristupamo računanju neodređenog integrala u gornjoj i donjoj tački intervala:

$$F(b) = b^2 + C = 2^2 + C = 4 + C$$

Formula 15 Neodređeni integral u gornjoj tački intervala

$$F(a) = a^2 + C = 1^2 + C = 1 + C$$

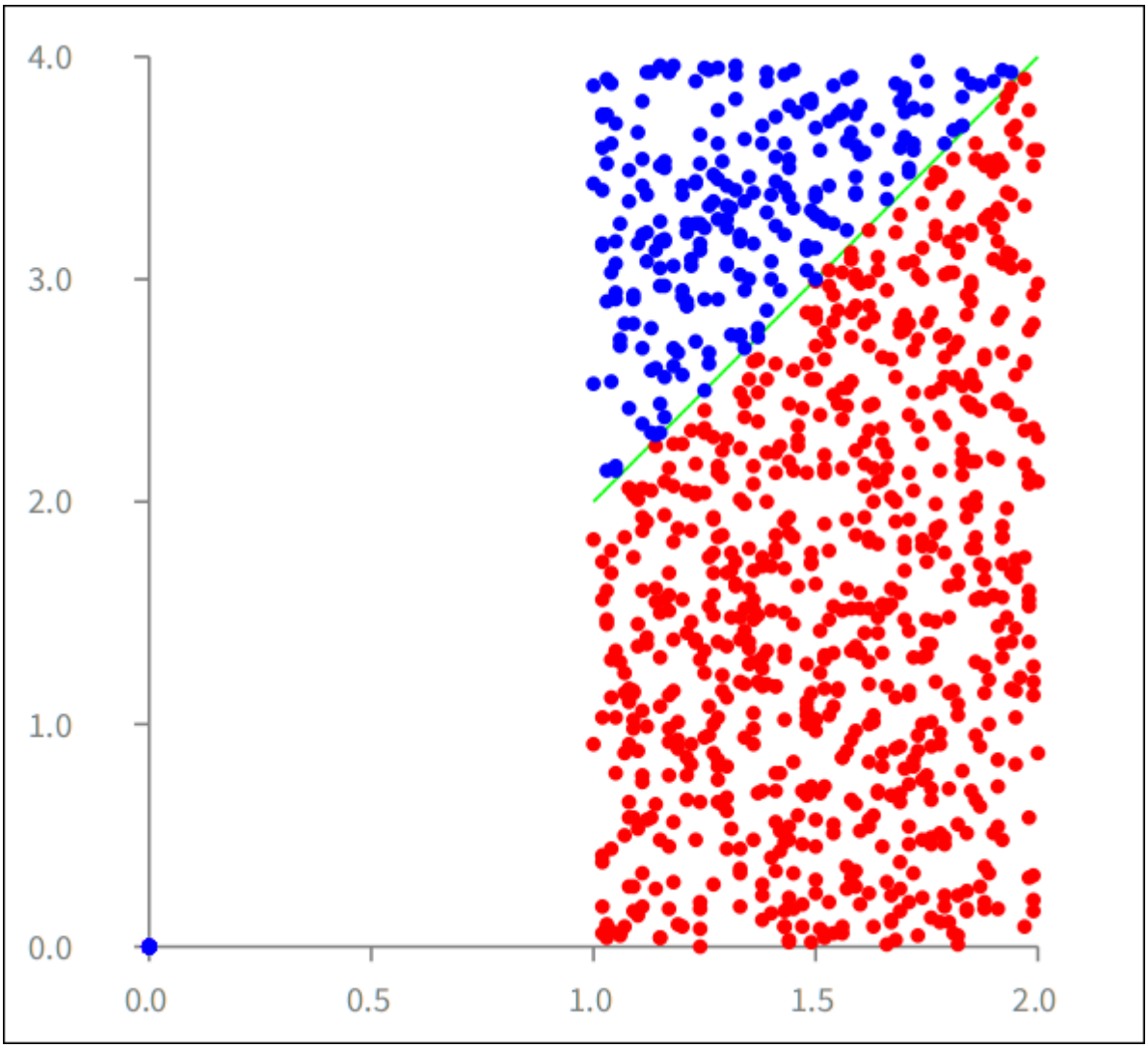
Formula 16 Neodređeni integral u donjoj tački intervala

Određeni integral se računa po formuli:

$$\int_a^b f(x)\,dx = [F(x)]_a^b = F(b) - F(a) = 3$$

Formula 17 Analitičko rešenje traženog određenog integrala

Na ovaj način dobijamo precizno analitičko rešenje. Prednost analitičkog rešenja je njegova preciznost, dok je mana dužina trajanja izračunavanja. Monte Karlo simulacija daje prednost brzini izračunavanja nad preciznošću izračunavanja. Monte Karlo simulacija spada u numeričke integracione tehnike. Prednost numeričkog rešenja je brzina izračunavanja dok je mana preciznost. Monte Karlo simulacija je aproksimativna tehnika koja se koristi u situacijama kada preciznost rešenja nije krucijalna, ali je brzina dolaska do rešenja od izuzetne važnosti. Preduslov primene Monte Karlo simulacije za izračunavanje aproksimirane vrednosti određenog integrala je da funkcija koju integralimo tj. integrand mora biti nenegativna neprekidna funkcija na traženom intervalu. Nenegativna funkcija je ona funkcija čije vrednosti su nenegativne. Grafik nenegativnih funkcija se nalazi na i/ili iznad x-ose. Neprekidna funkcija je ona funkcija koja nema skokove, prekide ili vertikalne asimptote.



Slika 5 Numeričko rešenje traženog određenog integrala

Na Slika 5 zelenom bojom je predstavljen integrand, crvenom bojom tačke čija je vrednost y koordinate manja od vrednosti funkcije u x koordinati tačke. Plavom bojom su prikazane tačke čija je vrednost y koordinate veća od vrednosti funkcije u x-koordinati tačke. Da bismo izračunali površinu ispod prave na određenom intervalu tj. integral neophodno je da pravu ograničimo po x-osi i po y-osi. Prava je ograničena po x-osi traženim intervalom [a,b] u konktetnom slučaju [1,2]. Da bismo ograničili funkciju po y-osi neophodno je da izračunamo maksimum funkcije y_{max} na posmatranom intervalu. Kada dobijemo maksimum funkcije na posmatranom intervalu tada je prava ograničena po y-osi intervalom koji se kreće od x-ose do maksimuma funkcije odnosno $[0,y_{max}]$ u konkretnom slučaju [0,4]. Sa ograničenjem po x-osi i ograničenjem po y-osi formira se pravougaonik koji omeđava pravu. Površina integrala računa se na osnovu površine omeđavajućeg

pravougaonika i odnosa broja tačaka koje se nalaze ispod prave i ukupnog broja tačaka koje se koriste za simulaciju.

$$\text{Širina}_{\text{pravougaonika}} = \text{b} - \text{a} = 2 - 1 = 1$$

Formula 18 Širina pravougaonika

$$\text{Dužina}_{\text{pravougaonika}} = \text{y}_{\text{max}} - 0 = 4 - 0 = 4$$

Formula 19 Dužina pravougaonika

$$\text{P}_{\text{pravougaonika}} = \text{Širina}_{\text{pravougaonika}} \times \text{Dužina}_{\text{pravougaonika}} = 1 \times 4 = 4$$

Formula 20 Površina pravougaonika

Kada imamo površinu pravougaonika potrebno je istu "izbombardovati" pseudo-slučajno generisanim tačkama i prebrojati one koje se nalaze ispod prave.Na osnovu sledeće proporcije dolazimo do formule za numeričko rešenje određenog integrala:

$$\frac{\text{Integral}}{\text{P}_{\text{pravougaonika}}} \approx \frac{\text{broja tačaka ispod prave}}{\text{ukupan broj tačaka}} \approx \frac{\text{broja tačaka ispod prave}}{\text{ukupan broj simulacija}}$$

Formula 21 Odnos broja tačaka ispod prave i ukupnog broja tačaka

U konkretnom slučaju površinu pravougaonika smo "izbombardovali" sa ukupno 1000 tačaka od toga je 749 završilo ispod prave. Povećanjem broja simulacija tj. broja tačaka kojima se površina pravougaonika "bombarduje" preciznost rešenja raste. Veći broj simulacija bolje aproksimira vrednost određenog integrala.

$$\text{Integral} \approx \text{P}_{\text{pravougaonika}} \frac{\text{broja tačaka ispod prave}}{\text{ukupan broj simulacija}} \approx 4\frac{749}{1000} \approx 2.99694$$

Formula 22 Izračunavanje aproksimirane vrednosti određenog integrala

Monte Karlo simulacijom koja spada u red numeričkih tehnika za aproksimiranje vrednosti određenog integrala dobili smo rezultate koji su približno jednaki rezultatima dobijenim na analitički način.

2.5.Input-Output analiza Monte Karlo simulacije za izračunavanje aproksimirane vrednosti broja Pi

Važno je da znamo šta je ulaz, a šta izlaz iz Monte Karlo simulacije za izračunavanje aproksimirane vrednosti broja Pi kako bismo bolje razumeli čtavu simulaciju.

1. Input: Pseudo-slučajno generisane koordinate tačaka (x koordinata i y koordinata) kojima se "bombarduje" površina kvadrata
2. Output: Aproksimirana vrednost broja Pi

2.6.Input-Output analiza Monte Karlo simulacije za izračunavanje aproksimirane vrednosti finansijske aktive

Važno je da znamo šta je ulaz, a šta izlaz iz Monte Karlo simulacije za izračunavanje aproksimirane vrednosti finansijske aktive kako bismo bolje razumeli čtavu simulaciju.

1. Input: povučeni istorijski podaci o kretanju cena finansijske aktive za određeni vremenski interval sa specijalizovanog sajta za praćenje finansijske aktive Yahoo finance [40]
2. Output: Predikcije budućih cena finansijske aktive za određeni vremenski interval

2.7.Input-Output analiza Monte Karlo simulacije za izračunavanje aproksimirane vrednosti određenog integrala

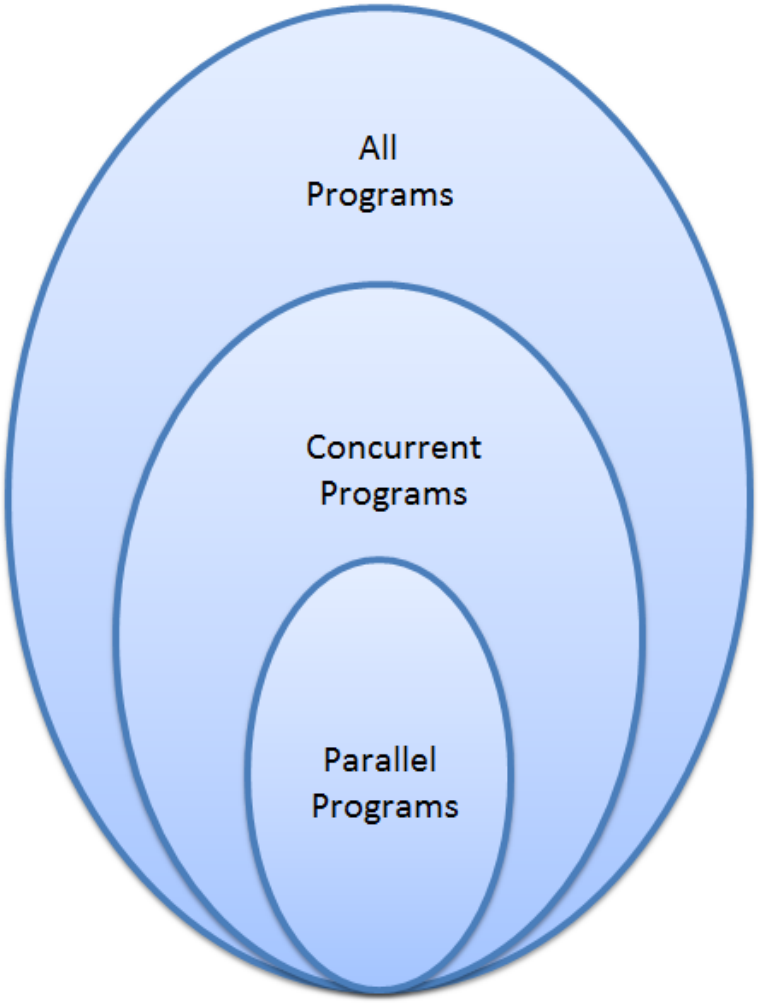
Važno je da znamo šta je ulaz, a šta izlaz iz Monte Karlo simulacije za izračunavanje aproksimirane vrednosti određenog integrala kako bismo bolje razumeli čtavu simulaciju.

1. Input: Pseudo-slučajno generisane koordinate tačaka (x koordinata i y koordinata) kojima se "bombarduje" površina pravougaonika
2. Output: Aproksimirana vrednost određenog integrala

3. Korišćene tehnike

3.1.Paralelno programiranje

Za potrebe razvoja projektnog rešenja korišćena je tehnika paralelnog programiranja. Radi boljeg razumevanja kada je paralelno programiranje moguće primeniti, potrebno je napraviti distinkciju između konkurentnog i paralelnog programiranja. Konkurentno i paralelno programiranje nisu sinonimi. Konkurentno programiranje predstavlja pisanje takvog koda da se njegovi delovi mogu nezavisno izvršavati [41]. Konkurentnost podrazumeva postojanje i kompozicija nezavisnih izvršivih jedinica, gde u zavisnosti od nivoa apstrakcije, izvršne jedinice mogu biti: delovi koda, niti, procesi. Sa druge strane paralelizam predstavalja simultano izvršavanje računanja (engl. computation), tj. izvršnih jedinica. Paralelno programiranje je podskup konkurentnog programiranja [41].

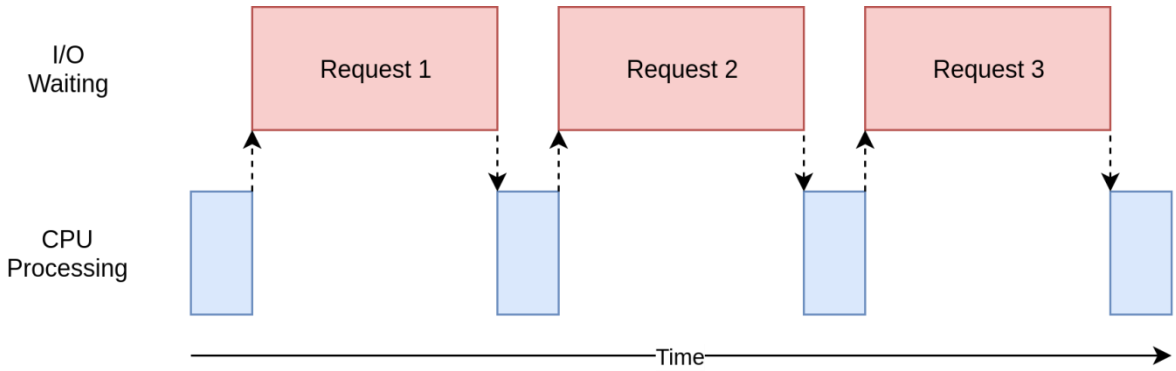


Slika 6 Konkurentno vs. Paralelno programiranje [41]

Paralelno programiranje omogućava bolju iskorišćenost hardverskog potencijala. Paralelnim programiranjem ukupan posao moze da se razdeli na više procesorskih jedinica čime se postiže veća iskorišćenost procesora. Na odluku da li koristiti konkurentno programiranje ili paralelno programiranje za rešavanje nekog problema utiče priroda rešavanog problema.

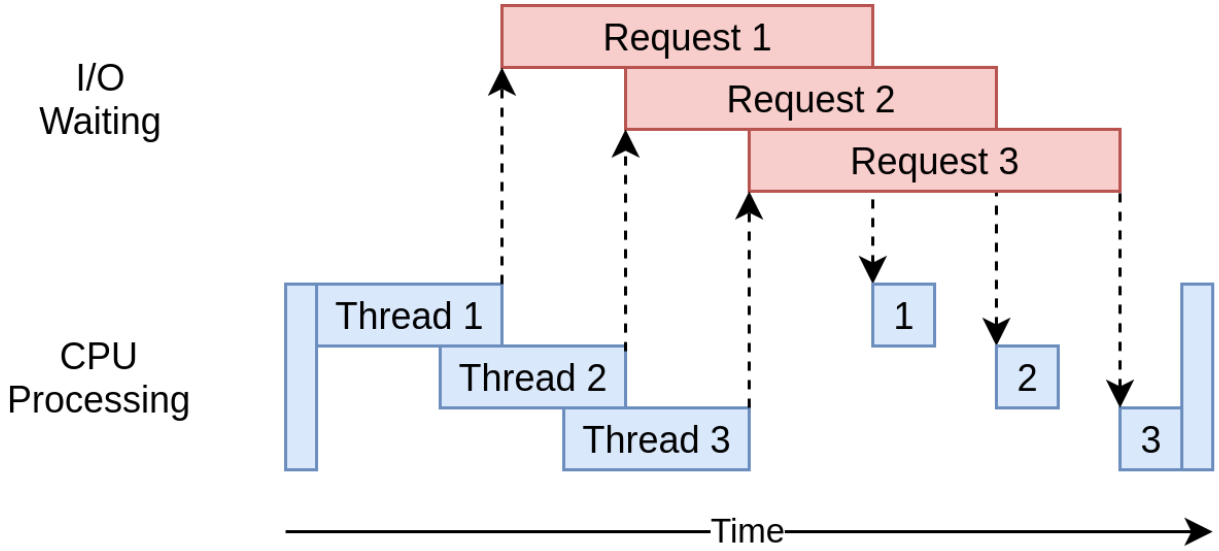
3.2.I/O Bound programi

I/O Bound programi najviše vreme provedu blokirani I/O operacijama (mreža, disk, itd.). Mogu se ubrzati preplitanjem vremena čekanja [41].



Slika 7 Primer I/O Bound programa [41]

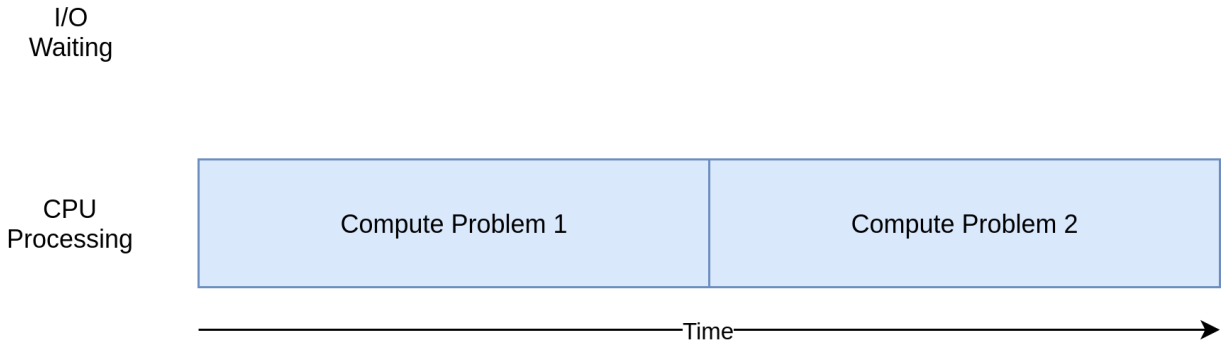
Niti se prepliću, čime se prepliće i njihovo čekanje, te otuda dolazi do ubrzanja rada programa [41]. Niti se međusobom prepliću i konkurišu za ograničeni resurs te odatle imamo ime konkurentno programiranje. Konkurentnost predstavlja borbu za ograničen resurs.



Slika 8 Primer preplitanja niti, konkurentno izvršavanje I/O Bound programa [41]

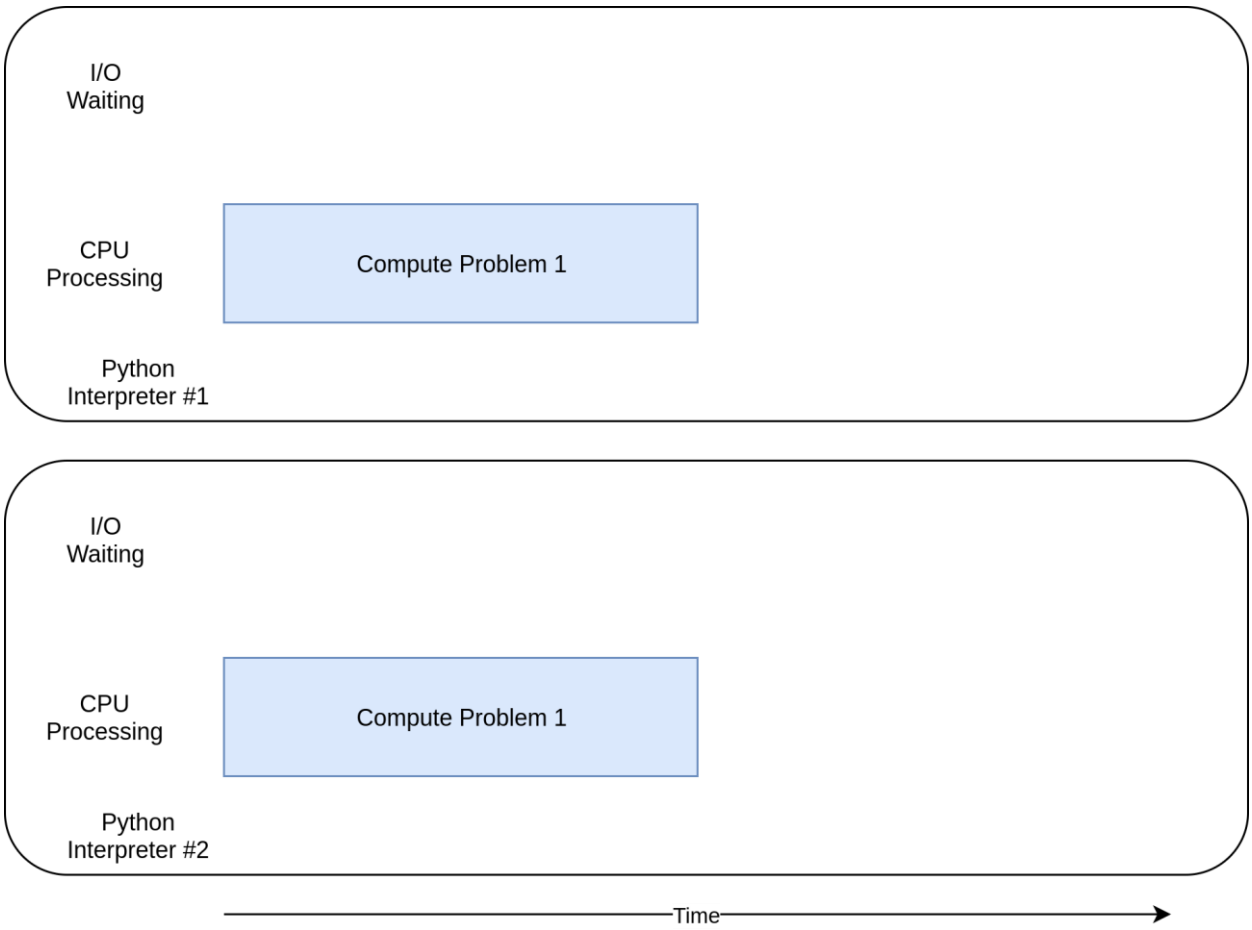
3.3.CPU Bound programi

CPU Bound programi najviše vremena provedu računajući. CPU je prezauzet i obično ga nema dovoljno. Ovakvi programi se mogu ubrzati povećanjem procesorske moći i njenim pravilnim iskorišćavanjem [41].



Slika 9 Primer CPU Bound programa [41]

Svaki proces pokreće poseban interpretera. Ova operacija je dosta skuplja od pokretanja niti. Svaki proces ima svoju memoriju. Razmena podataka se odvija putem posebnih protokola za međuprocesnu komunikaciju [41].



Slika 10 Paralelno izvršavanje CPU Bound programa [41]

Obično se dobri rezultati postižu kada se svakom CPU-u dodeli po jedan proces. Više procesa od CPU-ova dovodi do nepotrebne komutaciju procesa. Nekada se bolji rezultati dobijaju kada se smanji broj procesa, jer nema dovoljno posla. U zavisnosti od problema koji se rešava, povećanje broja procesa ne donosi uvek bolje rezultate. Kada se bira broj procesa i/ili broj CPU-ova, dobro je teorijski se osloniti na jako i slabo skaliranje, a zatim empirijski utvrditi najbolju konfiguraciju [41].

3.4.Preporuka za izbor Konkurentno vs. Paralelno programiranje

Za optimizaciju CPU Bound programa koristiti paralelno programiranje. Za optimizaciju I/O Bound programa koristiti konkurentno programiranje [41].

4. Korišćene tehnologije

Za potrebe razvoja projektnog rešenja korišćen je sledeći tehnološki stek (engl. Tech stack) :

1. Python [4]
2. Golang [5]
3. Pharo [6]
4. Roassal [7]
5. Yahoo Finance API [40]

U nastavku rada sledi detaljan opis svake od navedene tehnologije.

4.1.Python

Python programski jezik jedan je od najpopularnijih programskih jezika danas. Razvio ga je Guido van Rossum [42] 1991. godine [43]. Python je interpretiran dinamički jezik visokog nivoa [44]. Podržava više paradigmi programiranja: imperativno programiranje, proceduralno programiranje, objektno programiranje, funkcionalno programiranje, itd. Svoju popularnost stekao je zahvaljujući činjenici da je multi-platformski (engl. Cross-platform) programski jezik [44].

4.1.1. Python random modul

Python random modul [45] se koristi za generisanje pseudo-slučjnih brojeva. Ovaj modul implementira generatore pseudo-slučajnih brojeva koji se koriste za generisanje x i y koordinate tačaka koje se koriste za Monte Karlo simulaciju.

4.1.2. Python time modul

Python time modul [46] se koristi za generisanje dekoratora za merenje vremena izvršavanja simulacija. Dekoratori predstavljaju funkcije koje prihvataju kao parametar funkciju (ili uopšte callable) i vraćaju izmenjenu verziju [47].

4.1.3. Python i/o modul

Python i/o modul [48] se koristi za generisanje izlaznih fajlova iz simulacija koji se koriste kao ulaz za vizuelizaciju. Eksportuju se rezultati simulacija i eksperimenata skaliranja. Format podataka koji se eksportuju je tekstualan.

4.1.4. Python copy modul

Python copy modul [49] se koristi za operacije plitke i duboke kopije.

4.1.5. Python biblioteka NumPy

Python biblioteka NumPy [50] se koristi za numeričke operacije sa N-dimenzionim nizovima. U radu se koristi za izračunavanje osnovnih statističkih obeležja vremenske serije kao što su prosek, standardna devijacija i varijansa vremenske serije. Pored ove primene koristi se i za diferencijaciju vremenske serije. Diferencijacija vremenske serije (engl. Shifting/lagging time series) je postupak utvrđivanja razlike između dve uzastopne instance vremenske serije.

4.1.6. Python biblioteka Pandas

Python biblioteka Pandas [51] služi za prikupljanje, analizu i manipulaciju podacima. Osnovna jedinica rada u Pandas biblioteci je pandas DataFrame [52]. Pandas datareader [53] se koristi za preuzimanje podataka sa Yahoo Finance API-a u pandas DataFrame formatu.

4.1.7. Python biblioteka SciPy

Python biblioteka SciPy [54] se upotrebljava za numerička i statistička izračunavanja. Konkretno u radu se koristi statistički modul SciPy biblioteke. Stats modul [55] se koristi za izranavanje Z_{score}-a.

4.1.8. Python biblioteka Multiprocessing

Python biblioteka Multiprocessing [56] se koristi za paralelno izvršavanje Monte Karlo simulacija. Paralelno programiranje je moguće upotrebom biblioteke multiprocessing [41]. Korišćenjem Pool objekta [57] obezbeđuje se paralelno izvršavanje više Monte Karlo simulacija simultano. Pool objekat zapravo predstavlja Pool procesa gde se svakom procesu iz Pool-a dodeljuje deo podataka da obrađuje. Tako se postiže paralelizam podataka (engl. Data parallelism) [58]. Više procesa u paraleli obrađuju parcijalne podatke. Parcijalni podaci predstavljaju deo polaznih podataka. Svaki deo polaznih podataka dodeljuje se

jednom procesu iz Pool-a. Nakon što su parcijalni podaci obrađeni parcijalni rezultati se slivaju u Pool. Pool predstavlja agregator parcijalnih rezultata. Veći broj procesa koji u paraleli obrađuju manju količinu podataka otvara prostor da se ukupna količina podataka poveća tj. da se broj simulacija poveća. Ulaz u Pool predstavljaju parcijalni delovi ukupne količine podataka, a izlaz iz Pool-a predstavlja agregirani rezultat simulacije. Unutar Pool-a nalaze se procesi koji paralelno obrađuju podatke.

4.2.Golang

Golang je programski jezik čiji je razvoj započet 2007. godine, a javno je postao dostupan od 2009. godine. Razvili su ga u kompaniji Google [59] Robert Griesemer [60], Rob Pike [61], i Ken Thompson [62]. Golang je statički tipiziran, kompajliran programski jezik [63]. Sličan C-u ali sa memory safety, garbage collection i direktnom podrškom za konkurentno programiranje (engl. Communicating Sequential Processes (CSP)) [63]. Svoju popularnost je doživeo zahvaljujući činjenici da je vrlo efikasan, jednostavan i čitak za korišćenje [63].

4.2.1. Golang paket rand

Golang paket rand [64] se koristi za generisanje pseudo-slučjnih brojeva. Ovaj paket implementira generatore pseudo-slučajnih brojeva koji se koriste za generisanje x i y koordinate tačaka koje se koriste za Monte Karlo simulaciju.

4.2.2. Golang paket time

Golang paket time [65] se koristi za merenje vremena izvršavanja simulacija.

4.2.3. Golang paket os

Golang paket os [66] se koristi za generisanje izlaznih fajlova iz simulacija koji se koriste kao ulaz za vizuelizaciju. Eksportuju se rezultati simulacija i eksperimenata skaliranja. Format podataka koji se eksportuju je tekstualan.

4.2.4. Golang paket strings

Golang paket strings [67] se koristi za operacije sa stringovima.

4.2.5. Golang paket gonum/stat

Golang paket gonum/stat [68] se koristi za izračunavanje osnovnih statističkih obeležja vremenske serije kao što su prosek, standardna devijacija i varijansa vremenske serije.

4.2.6. Golang paket gaussian

Golang paket gaussian [69] se koristi za izračunavanje Z_{score}-a.

4.2.7. Golang paket quote

Golang paket quote [70] se koristi za preuzimanje podataka sa Yahoo Finance API-a.

4.2.8. Go rutine

Go rutina (engl. Goroutines) je nit (engl. thread) kreirana i upravljana od strane Go izvršnog okruženja (engl. runtime). Ključna reč go startuje novu Go rutinu koja izvršava funkciju [63]. Koristeći Go rutine funkcija se izvršava konkurentno [71]. Kada ispred funkcije napišemo ključnu reč go ne moramo da čekamo da funkcija završi svoje izvršavanje da bismo izvršili ostatak koda [71]. Evaluacija funkcije i argumenata funkcije se dešava u tekućoj Go rutini dok se izvršavanje funkcije odvija u novoj Go rutini [63]. Izvršavanje svih Go rutina odvija se u istom adresnom prostoru tako da pristup deljenoj memoriji mora biti sinhronizovan. Paket sync definiše korisne primitive za sinhronizaciju iako u Go-u ovo često nije neophodno [63].

4.2.9. Go kanali

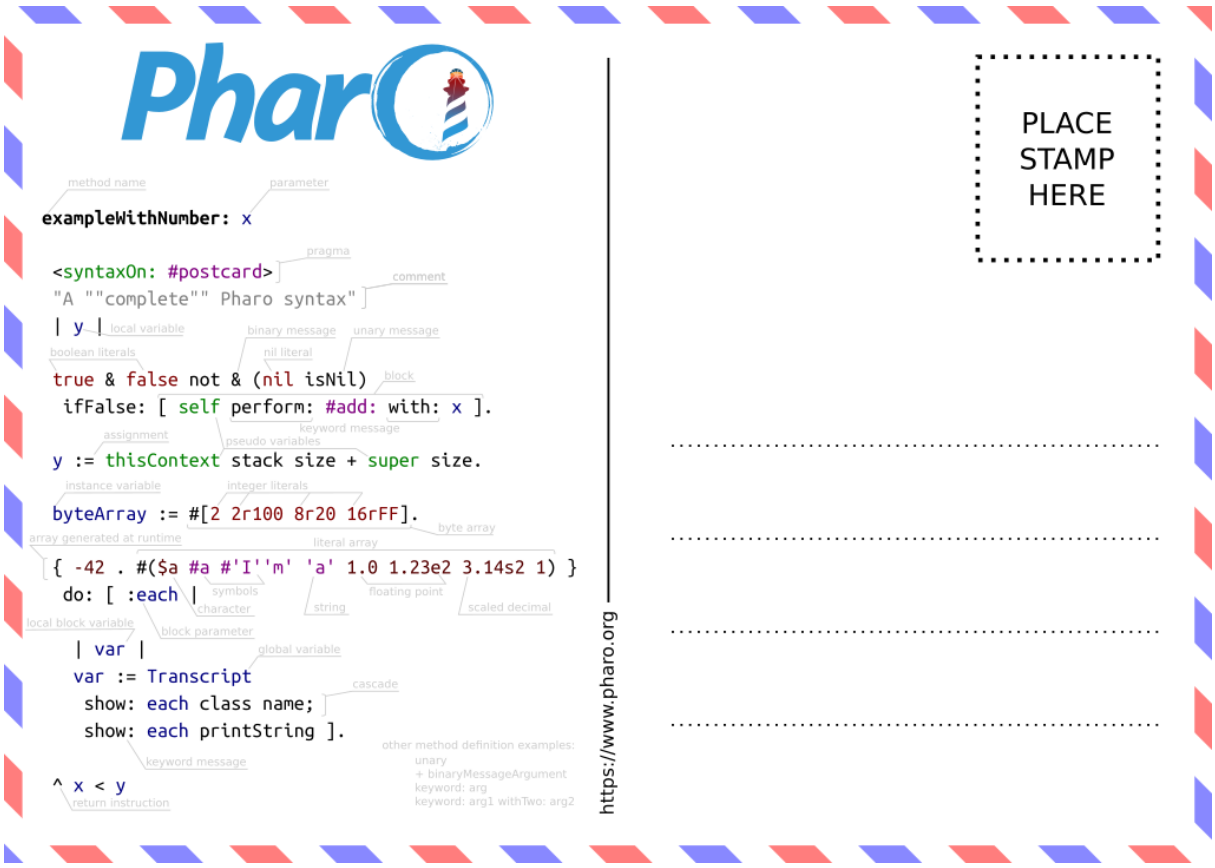
Go kanali (engl. Channels) predstavljaju tipizirane veze preko kojih se mogu slati vrednosti upotrebom operatora kanala (engl. channel operator) [63]. Operator kanala je predstavljen strelicom gde je tok podataka određen smerom strelice. Kanali se kreiraju dinamički upotrebom make funkcije. Podrazumevano čitanje i pisanje u kanal je blokirajuće dok druga strana ne obavi suprotnu operaciju. Ovo omogućava implicitnu sinhronizaciju Go rutina [63].

4.2.10. Go Baferovani kanali

Kanali mogu biti baferovani (engl. Buffered Channels) i u tom slučaju pisanje se blokira samo ukoliko je kanal pun. Čitanje se blokira samo ukoliko je kanal prazan. Kreiranje bafera omogućeno je drugim parametrom make funkcije [63].

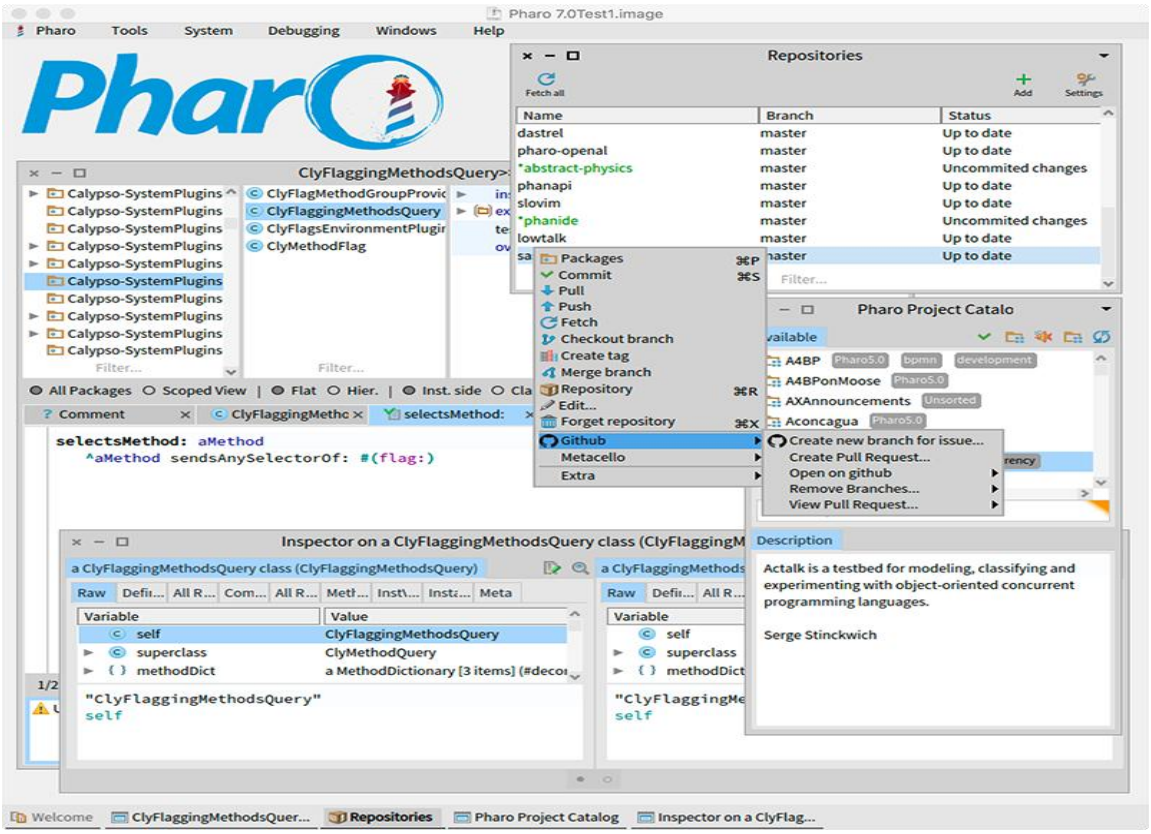
4.3.Pharo

Pharo je pravi objektno orijentisani programski jezik, sve je objekat uključujući i razvojno okruženje (engl. Integrated Development Environment (IDE)) [72]. Razvili su ga Stéphane Ducasse [73] i Marcus Denker [74] 2008. godine [75]. Na razvoj Pharo programskog jezika veliki uticaj je imao Smalltalk programski jezik [76]. Pharo je objektno-orijentisani, dinamički tipiziran, reflektivni programski jezik [77]. Pharo poseduje neke od naprednijih koncepata softverskog inženjerstva: pristup da je sve objekat, razmena poruka, "živ" sistem, virtualna mašina [72]. Pharo ima jednostavan i moćan objektni model. Sve je objekat tj. instanca klase. Sve metode su javne i virtualne. Svi atributi su zaštićeni. Podržava jednostruko nasleđivanje (engl. Single inheritance) [72]. Činjenica da Pharo predstavlja "ziv" sistem odnosi se na ideju da sve što vidite su objekti sa kojima možete stupiti u interakciju i menjati ih "naživo". Program koji se razvija je nerazdvojni deo razvojnog okruženja [72]. Poruke (engl. messages) predstavljaju nameru (šta treba uraditi). Metode opisuju kako treba nešto uraditi. Objekat koji prima poruku zovemo prijemnikom (engl. receiver) [72]. Kako je sve objekat tj. instanca klase odatle sledi da su klase i poruke takođe objekti. Celokupno procesiranje se obavlja razmenom poruka (engl. message passing) između objekata [72]. Pharo programski jezik je napisan u samom sebi. Svi elementi programskog jezika su napisani u Pharo programskom jeziku [72].Svoju popularnost je doživeo zahvaljujući izuzetno konciznoj sintaksi.



Slika 11 Sintaksa Pharo programskog jezika [75]

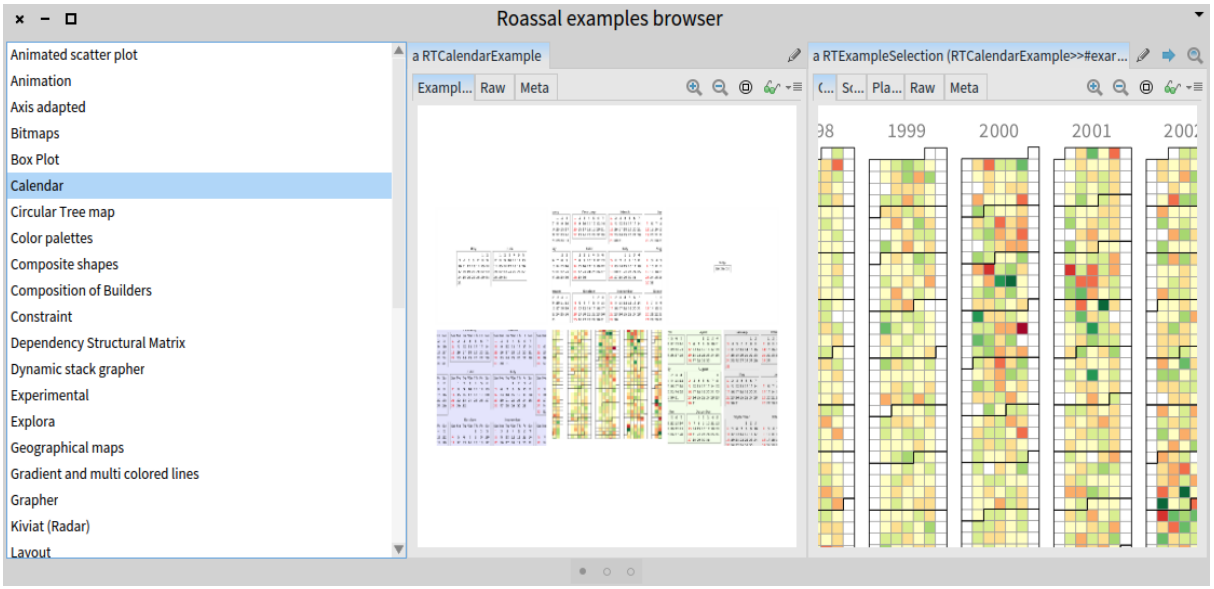
Koliko je sintaksa Pharo programskog jezika koncizna najbolje svedoči činjenica da se celokupna sintaksa jezika može smestiti na poleđinu jedne razglednice. Pharo nije "crna kutija" [72]. Introspekcija obezbeđuje korisniku interaktivnost sa Pharo razvojnim okruženjem. Celokupno razvojno okruženje su objekti sa kojima korisnike može da ostvari interakciju.



Slika 12 Primer Pharo razvojnog okruženja [78]

4.4.Roassal

Roassal je graphic engine, napisan u Pharo programskom jeziku [7]. Razvio ga je Alexandre Bergel [79] 2013. godine. Koristi se za interaktivnu vizuelizaciju podataka. Stiže sa proširivom bazom predefinisanih primera koje korisniku olakšavaju vizuelizaciju podataka.

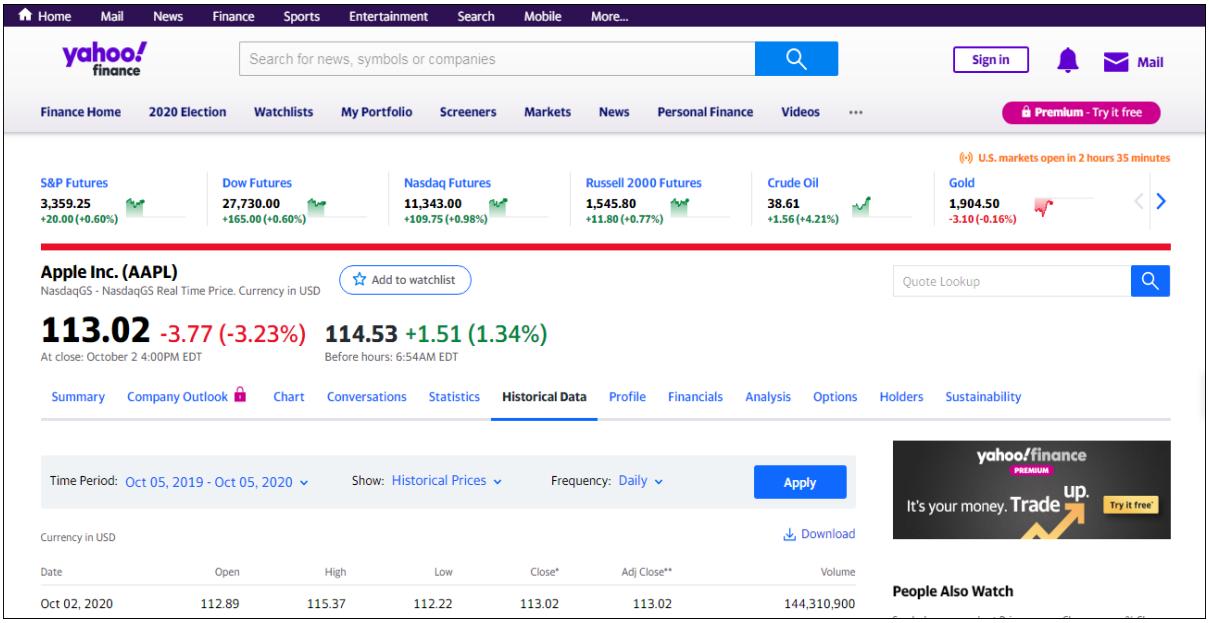


Slika 13 Roassal baza predefinisanih primera [7]

Roassal baza predefinisanih primera obuhvata vizuelizaciju: struktura u obliku stabla, struktura u obliku mreže, teorije grafova, fajl sistema, vremenskih serija, geografskih mapa, heat mapa, različitih animacija, itd [80].

4.5.Yahoo Finance API

Yahoo Finance API obezbeđuje povlačenje finansijskih podataka sa specijalizovanog sajta za praćenje i analizu kretanja cena finansijske aktive Yahoo Finance [40]. Pre nego što se podaci povuku neophodno je uneti nekoliko kriterijuma pretrage.



Slika 14 Yahoo Finance API

Potrebno je uneti ime finansijske aktive čije istorijske podatke želimo da preuzmemo kao i vremenski intervala za koji će se podaci preuzeti. Na osvnovu imena finansijske aktive i donje i gornje granice vremenskog intervala vrši se preuzimanje istorijskih podataka o kretanju cene finansijske aktive.

5. Specifikacija i arhitektura sistema

5.1.Specifikacija sistema

Za potrebe razvoja softverskog rešenja posmatrani sistem možemo trojako raslojiti. Prva dimenzija rešnja bi predstavljala oblast primene Monte Karlo simulacije (Pi, finansije, integrali). Druga dimenzija rešenja bi predstavljala tehnologija realizacije programskog rešenja (Python, Golang, Pharo). Dok bi treća dimenzija programskog rešenja predstavljala način izvršavanja programskog rešenja (serijsko, paralelno). Raslojavanjem sistema po dimenzijama dobijamo smernice kako je potrebno razviti programsko rešnje. U nastavku rada radi lakše implementacije sistema biće dati pseudo-kodovi serijske i paralelne verzije programa po oblastima primene.

Pseudo-kod serijske verzije Monte Karlo simulacije za izračunavanje aproksimirane vrednosti broja Pi:

1. Na pseudo slučajan način odredimo koordinatu x na intervalu [0,1]
2. Na pseudo slučajan način odredimo koordinatu y na intervalu [0,1]
3. Dobili smo tačku sa koordinatama (x,y)
4. Radimo sa jediničnom kružnicom čiji je centar u koordinatnom početku C(0,0), a radijus r=1
5. Proveravamo da li je udaljenost novo formirane tačke od centra manja od 1
6. Uslov na osnovu kojeg se proverava da li se tačka se nalazi unutar kružnice $x^2 + y^2 < r^2$ odnosno u slučaju jedinične kružnice $x^2 + y^2 < 1$
7. Ukoliko se tačka nalazi unutar kružnice uvećavamo brojač tačaka unutar kružnice
8. Ukupan broj simulacija predstavlja ukupan broj tačaka kojima se "bombarduje" površina kvadrata opisanog oko jedinične kružnice
9. Zbog odnosa površine kruga i površine kvadrata koja iznosi π / 4 odnos broja tačaka unutar kružnice i ukupnog broja tačaka množimo sa 4 kako bismo dobili aproksimiranu vrednost broja π
10. $\frac{P_{kruga}}{P_{kvadrata}} \approx \frac{\text{Broj tačaka u krugu}}{\text{Ukupan broj tačaka}}$
11. Izračunali smo približno vrednost broja π koja je sve tačnija i tačnija kako povećavamo broj simulacija odnosno broj tačaka sa kojima "bombardujemo" površinu kvadrata
12. $\pi \approx 4\frac{\text{Broj tačaka u krugu}}{\text{Ukupan broj tačaka}}$

Pseudo-kod paralelne verzije Monte Karlo simulacije za izračunavanje aproksimirane vrednosti broja Pi:

1. Podelimo ukupan broj simulacija tj. ukupan broj tačaka sa kojima "bombardujemo" površinu kvadrata sa brojem procesa iz Pool-a
2. Pretpostavka da imamo i procesa u Pool-u
3. Svaki proces obrađuje jedan deo ukupnog posla tj. svaki proces obrađuje deo ukupnog broja tačaka. Svaki proces obrađuje istu količinu posla. Ukupna količina posla u ovom slučaju ukupan broj tačaka se razdeli na jednake delove tako da svaki proces dobije jednaku količinu tačaka da obradi
4. Pod obradom tačaka se podrazumeva provera da li je udaljenost novo formirane tačke od centra manja od 1
5. Svi procesi svoj posao obavljaju simultano tj. u paraleli
6. Nakon što završe sa obradom parcijalni rezultati obrade se slivaju u Pool
7. $\frac{P_{kruga}}{P_{kvadrata}} \approx \frac{Rezultat_{proces\_1}+\cdots+Rezultat_{proces\_i}}{\text{Ukupan broj tačaka}}$
8. Parcijalni rezultati koje procesi vraćaju agregiraju se u Pool-u
9. Zbog odnosa površine kruga i površine kvadrata koja iznosi π/4 odnos broja tačaka unutar kružnice i ukupnog broja tačaka množimo sa 4 kako bismo dobili aproksimiranu vrednost broja π
10. Izračunali smo približno vrednost broja π koja je sve tačnija i tačnija kako povećavamo broj simulacija odnosno broj tačaka sa kojima "bombardujemo" površinu kvadrata
11. $\pi \approx 4\frac{Rezultat_{proces\_1}+\cdots+Rezultat_{proces\_i}}{\text{Ukupan broj tačaka}}$

Pseudo-kod serijske verzije Monte Karlo simulacije za izračunavanje aproksimirane vrednosti finansijske aktive:

1. Povlačimo finansijske podatke preko Yahoo Finance API-a
2. Računamo prinos finansijske aktive
3. Računamo standardnu devijaciju finansijske aktive odnosno volatilnost
4. Računamo buduću cenu finansijske aktive
5. Formiramo fen dijagram krivih
6. Odabiramo onu predikciju koja ima najveću verovatnoću da se desi. Najveću verovatnoću da se desi ima ona predikcija koja se nalazi u koridoru +/- jedna standardna devijacija

7. Ostale predikcije odbacujemo

Pseudo-kod paralelne verzije Monte Karlo simulacije za izračunavanje aproksimirane vrednosti finansijske aktive:

1. Umesto da vršimo predikciju po predikciju sekvencijalno. Funkcija za izračunavanje buduće cene finansijske aktive se poziva simultano
2. Svaki proces obrađuje jedan deo ukupnog posla tj. svaki proces vrši izračunavanje buduće cene finansijske aktive. Svaki proces obrađuje istu količinu posla. Ukupna količina posla u ovom slučaju ukupan broj predikcija se razdeli na jednake delove tako da svaki proces dobije jednaku količinu predikcija da simulira
3. Svi procesi svoj posao obavljaju simultano tj. u paraleli
4. Više procesa istovremeno vrši predikcije čime se za kraće vreme formira fen dijagram krivih
5. Svaki proces računa buduće cene finansijske aktive sa različitim ulaznim parametrima čme se oslikava tržišna nestabilnost i slučajnost
6. Nakon što završe sa predikcijama procesi rezultate predikcija prepuštaju Pool-u
7. Pool vrši agregiranje parcijalnih rezultata predikcija i na taj način formira bazu predikcija na osnovu koje se formira fen dijagram
8. Formiramo fen dijagram krivih
9. Odabiramo onu predikciju koja ima najveću verovatnoću da se desi. Najveću verovatnoću da se desi ima ona predikcija koja se nalazi u koridoru +/- jedna standardna devijacija
10. Ostale predikcije odbacujemo

Pseudo-kod serijske verzije Monte Karlo simulacije za izračunavanje aproksimirane vrednosti određenog integrala:

1. Imamo funkciju f(x) za koju tražmo određeni integral na nekom intervalu [a,b]
2. Na osnovu intervala [a,b] određenog integrala formira se horizontalno ograničenje integranda. Horizontalno ograničenje integranda predstavlja širinu omeđavajućeg pravougaonika. Širina pravougaonika iznosi b - a
3. Računamo maksimum funkcije ymax na intervalu [a,b]
4. Na osnovu maksimuma funkcije na intervalu formira se vertikalno ograničenje integranda. Vertikalno ograničenje integranda predstavlja visinu omeđavajućeg pravougaonika. Visina pravougaonika iznosi ymax – 0 = ymax
5. Horizontalno i vertikalno ograničenje doprinose da se posmatrana funkcija oiviči kako po horizontali tako i po vertikali
6. Nakon što je funkcija oivičena pravougaonikom, čije stranice su horizontalno i vertikalno ograničenje, pristupamo Monte Karlo simulaciji
7. Random() funkcija uzima vrednosti iz intervala [0,1]
8. Na pseudo slučajan način odredimo koordinatu x na intervalu [a,b]
9. $x = a + (b - a)\text{Random}()$
10. Na pseudo slučajan način odredimo koordinatu y na intervalu [0,ymax]
11. $y = 0 + (y_{max} - 0)\text{Random}() = y_{max}\text{Random}()$
12. Dobili smo tačku sa koordinatama (x,y)
13. Proveravamo da li je vrednost y koordinate novo formirane tačke manja od vrednosti funkcije u x koordinati
14. $y_{koordinata} < f(x_{koordinata})$
15. Ako je prethodni uslov zadovoljen uvećavamo vrednost brojača tačaka ispod grafika funkcije
16. Ukupan broj simulacija predstavlja ukupan broj tačaka kojima se "bombarduje" površina pravougaonika. Površina pravougaonika ima zadatak da oiviči graf funkcije za koju tražimo odrađeni integral
17. $\frac{\text{Integral}}{P_{pravougaonika}} = \frac{\text{broja tačaka ispod grafika funkcije}}{\text{ukupan broj tačaka}}$
18. Kako znamo površinu pravougaonika, integral računamo na osnovu prethodnog odnosa
19. $\text{Integral} = \frac{\text{broja tačaka ispod grafika funkcije}}{\text{ukupan broj tačaka}} P_{pravougaonika}$

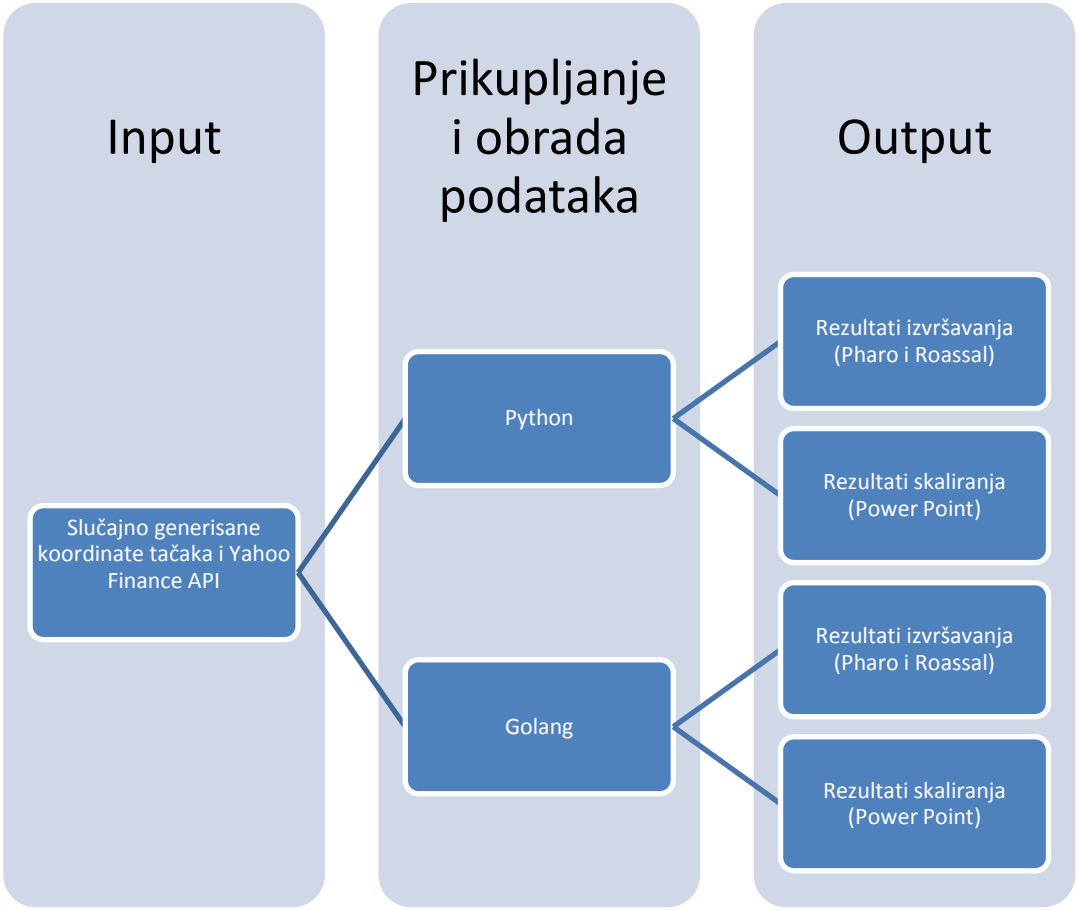
Pseudo-kod paralelne verzije Monte Karlo simulacije za izračunavanje aproksimirane vrednosti određenog integrala:

1. Podelimo ukupan broj simulacija tj. ukupan broj tačaka sa kojima "bombardujemo" površinu pravougaonika sa brojem procesa iz Pool-a
2. Pretpostavka da imamo i procesa u Pool-u

3. Svaki proces obrađuje jedan deo ukupnog posla tj. svaki proces obrađuje deo ukupnog broja tačaka. Svaki proces obrađuje istu količinu posla. Ukupna količina posla u ovom slučaju ukupan broj tačaka se razdeli na jednake delove tako da svaki proces dobije jednaku količinu tačaka da obradi
4. Pod obradom tačaka se podrazumeva provera da li je vrednost y koordinate novo formirane tačke manja od vrednosti funkcije u x koordinati
5. Svi procesi svoj posao obavljaju simultano tj. u paraleli
6. Nakon što završe sa obradom parcijalni rezultati obrade se slivaju u Pool
7. Parcijalni rezultat obrade predstavlja integral koji je proces izračunao na parcijalnom skupu podataka tj. na skupu podataka koji je manji od polaznog skupa ali dovoljno reprezentativan
8. $\text{Integral} = \frac{\text{Integral}_{\text{proces\_1}}+\cdots+\text{Integral}_{\text{proces\_i}}}{\text{i}}$
9. Parcijalni rezultati koje procesi vraćaju agregiraju se u Pool-u
10. Izračunali smo približno vrednost traženog integrala koja je sve tačnija i tačnija kako povećavamo broj simulacija odnosno broj tačaka sa kojima "bombardujemo" površinu pravougaonika koji ograničava grafik funkcije

5.2.Arhitektura sistema

Kada govorimo o arhitekturi sistema ono što je zajedničko za sve tri primene Monte Karlo simulacije je tok podataka (engl. Data flow). Od izuzetne je važnosti da se razume kako su nastajanje i vizuelizacija podataka međusobno povezani.



Slika 15 Dijagram toka podataka

Tok podataka je povezan sa input-output analizom. Ulazi u Monte Karlo simulaciju su ili slučajno generisane koordinate tačaka ili finansijski podaci koji su povučeni preko Yahoo Finance API-a. Za prikupljanje i obradu podataka zaduženi su Python i Golang programski jezici. Vizuelizacija rešenja vrši se upotrebom Pharo programskog jezika i Roassal graphic engine-a. Izlaz iz Monte Karlo simulacija su sami rezultati simulacija, ali i rezultati eksperimentalnog skaliranja. Poglavlje implementacija sistema bavi se prikupljanjem i obradom podataka realizovanom u Python i Golang programskim jezicima. Poglavlje vizuelizacija implementiranog sistema ima zadatak da prikaže rezultate Monte Karlo simulacija po oblastima primene koristeći Pharo i Roassal. Posebno poglavlje je posvećeno rezultatima eksperimentalnog skaliranja

6. Implementacija sistema

Cilj ovog poglavlja je da se prikaže kako je programski izvršena implementacija Monte Karlo simulacija po oblastima primene. U ovom delu biće prikazana implementacija u Python i Golang programskim jezicima.

6.1.Implementacija u Python programskom jeziku

6.1.1. Python implementacija Monte Karlo simulacije za izračunavanje aproksimirane vrednosti broja Pi

Dekorator se koristi za merenje vremena izvršavanja serijske i paralelne verzije programa. Dekorator za merenje vremena se koristi tako što se funkcija čije nas vreme izvršavanja interesuje anotira dekoratorom.

```
def calculate_execution_time(function):
    def calculate_duration(*args, **kwargs):
        start_time = time.time()
        executing_function = function(*args, **kwargs)
        end_time = time.time()
        execution_time = round(end_time - start_time, 7)
        return executing_function, execution_time
    return calculate_duration
```

Listing 1 Dekorator za merenje vremena izvršavanja serijske i paralelne verzije programa

Klasa MonteCarloSimulationPi sadrži atribute: number_of_processes kojim se definiše broj procesa u Pool-u, parallel_flag određuje da li je izvršvanje programa serijsko ili paralelno i experiment_flag koji govori da li smo u režimu eksperimentalnog skaliranja.

```
class MonteCarloSimulationPi:
    def __init__(self, number_of_processes):
        self.number_of_processes = number_of_processes
        self.parallel_flag = False
        self.experiment_flag = False
```

Listing 2 MonteCarloSimulationPi klasa

Klasa MonteCarloSimulationPi sadrži metode: simulation_pi, mcs_pi_serial i mcs_pi_parallel. Metoda simulation_pi predstavlja okosnicu serijskog i paralelnog izvršavanja programa. U ovoj metodi je izvršena implementacija Monte Karlo simulacije.

```
def simulation_pi(self, number_of_simulations):
    if self.experiment_flag == True:
        inside = 0
        for _ in range(number_of_simulations):
            x = random.random()
            y = random.random()
            # The unit circle is the circle of radius 1 centered at the origin(0, 0)
            # in the Cartesia coordinate system in the Euclidean plane.
            if x * x + y * y < 1:
                inside = inside + 1
        return inside
```

Listing 3 Metoda simulation_pi

Metoda mcs_pi_serial predstavlja serijsku implementaciju izračunavanja aproksimirane vrednosti broja Pi. Iznad definicije funkcije se nalazi anotacija dekoratora za merenje vremena izvršavanja.

```
@calculate_execution_time
def mcs_pi_serial(self, number_of_simulations):
    self.parallel_flag = False
    pi = 4 * self.simulation_pi(number_of_simulations) / number_of_simulations
    return pi
```

Listing 4 Metoda mcs_pi_serial

Metoda mcs_pi_parallel predstavlja paralelnu implementaciju izračunavanja aproksimirane vrednosti broja Pi. Iznad definicije funkcije se nalazi anotacija dekoratora za merenje vremena izvršavanja.

```
@calculate_execution_time
def mcs_pi_parallel(self, number_of_simulations):
    self.parallel_flag = True
    pool = Pool(processes=self.number_of_processes)
    number_of_simulations_per_process = int(number_of_simulations / self.number_of_processes)
    simulations_per_process = []
    # Append the same value multiple times to a list
    # To add v, n times, to l:
    # l += n * [v]
    simulations_per_process += self.number_of_processes * [number_of_simulations_per_process]
    inside_sum = pool.map(self.simulation_pi, simulations_per_process)
    pi = 4 * sum(inside_sum) / number_of_simulations
    return pi
```

Listing 5 Metoda mcs_pi_parallel

6.1.2. Python implementacija Monte Karlo simulacije za izračunavanje aproksimirane vrednosti finansijske aktive

Dekorator se koristi za merenje vremena izvršavanja serijske i paralelne verzije programa. Dekorator za merenje vremena se koristi tako što se funkcija čije nas vreme izvršavanja interesuje anotira dekoratorom.

```
def calculate_execution_time(function):
    def calculate_duration(*args, **kwargs):
        start_time = time.time()
        executing_function = function(*args, **kwargs)
        end_time = time.time()
        execution_time = round(end_time - start_time, 7)
        return executing_function, execution_time


    return calculate_duration
```

Listing 6 Dekorator za merenje vremena izvršavanja serijske i paralelne verzije programa

Klasa MonteCarloSimulationFinance sadrži atribute: start_date, end_date, ticker_symbol, data, time_series, number_of_processes i parallel_flag. Atributi start_date i end_date se koriste kako bi se definisao vremenski interval za koji će se prikupljati finansijski podaci preko Yahoo Finance API-a. Atribut ticker_symbol predstavlja ime finansijske aktive za koju se podaci prikupljaju. Ova tri atributa su neophodna za korišćenje Yahoo Finance API-a. Prikupljena vremenska serija predstavljena je atributom time_series, koja predstavlja sirove podatke povučene preko berzanskog API-a. Rafinirani podaci predstavljeni su atributom data. Atribut number_of_processes definiše broj procesa u Pool-u, dok parallel_flag određuje da li je izvršvanje programa serijsko ili paralelno.

```
class MonteCarloSimulationFinance:
    def __init__(self, start_date, end_date, ticker_symbol, number_of_processes):
        self.start_date = start_date
        self.end_date = end_date
        self.ticker_symbol = ticker_symbol
        self.data = None
        self.time_series = None
        self.number_of_processes = number_of_processes
        self.parallel_flag = False
```

Listing 7 MonteCarloSimulationFinance klasa

Klasa MonteCarloSimulationFinance sadrži više pomoćnih metoda i 3 glavne metode. Pomoćna metoda data_acquisition za prikupljanje podataka preko Yahoo Finance API-a.

```
def data_acquisition(self):
    stock = data.DataReader(self.ticker_symbol, 'yahoo', self.start_date, self.end_date)
    stock = stock.dropna()
    self.time_series = stock['Close']
```

Listing 8 Metoda data_acquisition

Pomoćna metoda calculate_periodic_daily_return za obradu i diferencijaciju vremenskih serija. Ova metoda računa kontinualnu stopu prinosa.

```
# Differencing time series = Shifting and lagging time series
def calculate_periodic_daily_return(self):
    # Differencing time series
    # The diff() function calculates the first differences of the time series.
    self.data = np.log(self.time_series).diff().dropna()
    # Shifting and lagging time series
    # self.data=np.log(self.time_series / self.time_series.shift(1)).dropna()
```

Listing 9 Metoda calculate_periodic_daily_return

Pomoćna metoda calculate_z_score koja se koristi za izračunavanje Z_{score}-a.

```
def calculate_z_score(self):
    return norm.ppf(random.random())
```

Listing 10 Metoda calculate_z_score

Pomoćna metoda calculate_average_daily_return za izračunavanje proseka vremenske serije.

```
def calculate_average_daily_return(self):
    return np.mean(self.data)
```

Listing 11 Metoda calculate_average_daily_return

Pomoćna metoda calculate_variance za izračunavanje varijanse vremenske serije.

```
def calculate_variance(self):
    return np.var(self.data)
```

Listing 12 Metoda calculate_variance

Pomoćna metoda calculate_standard_deviation za izračunavanje standardne devijacije vremenske serije.

```
def calculate_standard_deviation(self):
    return np.std(self.data)
```

Listing 13 Metoda calculate_standard_deviation

Pomoćna metoda calculate_drift za izračunavanje finansijskog drifta.

```
def calculate_drift(self):
    return self.calculate_average_daily_return() - self.calculate_variance() / 2
```

Listing 14 Metoda calculate_drift

Pomoćna metoda calculate_random_value za izračunavanje slučajne vrednosti.

```
def calculate_random_value(self):
    return self.calculate_standard_deviation() * self.calculate_z_score()
```

Listing 15 Metoda calculate_random_value

Metoda simulation_finance predstavlja okosnicu serijskog i paralelnog izvršavanja programa. U ovoj metodi je izvršena implementacija Monte Karlo simulacije.

```
# prediction window size: number of prediction days per simulation
def simulation_finance(self, number_of_simulations, prediction_window_size):
    predictions = []
    prediction = []
    for i in range(number_of_simulations):
        # today's price
        prediction.append(self.time_series.iloc[-1])
        for j in range(prediction_window_size):
            # Next Day's Price=Today's Price × e^(Drift+Random Value)
            prediction.append(
                prediction[-1] * pow(math.e, (self.calculate_drift() + self.calculate_random_value())))

        predictions.append(copy.deepcopy(prediction))
        prediction.clear()
    return predictions
```

Listing 16 Metoda simulation_finance

Metoda mcs_finance_serial predstavlja serijsku implementaciju izračunavanja aproksimirane vrednosti finansijske aktive. Iznad definicije funkcije se nalazi anotacija dekoratora za merenje vremena izvršavanja.

```
@calculate_execution_time
def mcs_finance_serial(self, number_of_simulations, prediction_window_size):
    self.parallel_flag = False
    return self.simulation_finance(number_of_simulations, prediction_window_size)
```

Listing 17 Metoda mcs_finance_serial

Metoda mcs_finance_parallel predstavlja paralelnu implementaciju izračunavanja aproksimirane vrednosti finansijske aktive. Iznad definicije funkcije se nalazi anotacija dekoratora za merenje vremena izvršavanja.

```
@calculate_execution_time
def mcs_finance_parallel(self, number_of_simulations, prediction_window_size):
    self.parallel_flag = True
    pool = Pool(processes=self.number_of_processes)
    number_of_simulations_per_process = int(number_of_simulations / self.number_of_processes)
    simulations_per_process = []
    # Append the same value multiple times to a list
    # To add v, n times, to l:
    # l += n * [v]
    # Mapping a function with multiple arguments to a multiprocessing pool will distribute
    # the input data across processes to be run with the referenced function.
    simulations_per_process += self.number_of_processes * [
        (number_of_simulations_per_process, prediction_window_size)]
    predictions = pool.starmap(self.simulation_finance, simulations_per_process)
    return predictions
```

Listing 18 Metoda mcs_finance_parallel

6.1.3. Python implementacija Monte Karlo simulacije za izračunavanje aproksimirane vrednosti određenog integrala

Dekorator se koristi za merenje vremena izvršavanja serijske i paralelne verzije programa. Dekorator za merenje vremena se koristi tako što se funkcija čije nas vreme izvršavanja interesuje anotira dekoratorom.

```python
def calculate_execution_time(function):
    def calculate_duration(*args, **kwargs):
        start_time = time.time()
        executing_function = function(*args, **kwargs)
        end_time = time.time()
        execution_time = round(end_time - start_time, 7)
        return executing_function, execution_time

    return calculate_duration
```

Listing 19 Dekorator za merenje vremena izvršavanja serijske i paralelne verzije programa

Klasa MonteCarloSimulationIntegration sadrži atribute: number_of_processes kojim se definiše broj procesa u Pool-u, parallel_flag određuje da li je izvršvanje programa serijsko ili paralelno i experiment_flag koji govori da li smo u režimu eksperimentalnog skaliranja. Pored atributa imamo i tri konstante: donja i gornja granica intervala i veličina podeoka na koji se interval deli.

```python
class MonteCarloSimulationIntegration:
    def __init__(self, number_of_processes):
        self.number_of_processes = number_of_processes
        self.parallel_flag = False
        self.experiment_flag = False
        # Upper and Lower Bounds of Integral.
        self.LOWER_BOUND = 1
        self.UPPER_BOUND = 2
        # The area under the graph of a function can be found by adding slices that approach zero in width.
        self.SLICE_SIZE = 0.01
```

Listing 20 MonteCarloSimulationIntegration klasa

MonteCarloSimulationIntegration sadrži metode: function, simulation_integration, mcs_integration_serial i mcs_integration_parallel. Metoda function je integrand tj. funkcija za koju tražimo određeni integrala na nekom intervalu.

```python
def function(self, x):
    return 2*x
```

Listing 21 Metoda function

Metoda simulation_integration predstavlja okosnicu serijskog i paralelnog izvršavanja programa. U ovoj metodi je izvršena implementacija Monte Karlo simulacije.

```
def simulation_integration(self, number_of_simulations):
    if self.experiment_flag == True:
        # Points under the graph of a function.
        below = 0
        lower_bound_interval = self.LOWER_BOUND
        upper_bound_interval = self.UPPER_BOUND
        # Define the interval between the lower and upper bound.
        x = []
        # Function Values
        y = []
        # Maximum of the function f(x) on the interval[lower_bound, upper_bound]
        f_max = self.function(self.LOWER_BOUND)

        while lower_bound_interval < upper_bound_interval:
            x.append(lower_bound_interval)
            t = self.function(lower_bound_interval)
            y.append(t)
            if t > f_max:
                f_max = t
            lower_bound_interval += self.SLICE_SIZE

        for _ in range(number_of_simulations):
            x_rand = self.LOWER_BOUND + (self.UPPER_BOUND - self.LOWER_BOUND) * random.random()
            y_rand = 0 + f_max * random.random()
            if y_rand < self.function(x_rand):
                below = below + 1
        # Rectangle area that surrounds the area under the graph of a function.
        a = self.UPPER_BOUND - self.LOWER_BOUND
        b = f_max - 0
        rectangle_area = a * b
        # bellow = Points under the graph of a function.
        # number_of_simulations = Total number of points = Points inside rectangle
        proportion = below / number_of_simulations
        integral = proportion * rectangle_area
        return integral
```

Listing 22 Metoda simulation_integration

Metoda mcs_integration_serial predstavlja serijsku implementaciju izračunavanja aproksimirane vrednosti integrala. Iznad definicije funkcije se nalazi anotacija dekoratora za merenje vremena izvršavanja.

```
@calculate_execution_time
def mcs_integration_serial(self, number_of_simulations):
    self.parallel_flag = False
    integral = self.simulation_integration(number_of_simulations)
    return integral
```

Listing 23 Metoda mcs_integration_serial

Metoda mcs_integration_parallel predstavlja paralelnu implementaciju izračunavanja aproksimirane vrednosti integrala. Iznad definicije funkcije se nalazi anotacija dekoratora za merenje vremena izvršavanja.

```
@calculate_execution_time
def mcs_integration_parallel(self, number_of_simulations):
    self.parallel_flag = True
    pool = Pool(processes=self.number_of_processes)
    number_of_simulations_per_process = int(number_of_simulations / self.number_of_processes)
    simulations_per_process = []
    # Append the same value multiple times to a list
    # To add v, n times, to l:
    # l += n * [v]
    # Mapping a function with multiple arguments to a multiprocessing pool will distribute
    # the input data across processes to be run with the referenced function.
    simulations_per_process += self.number_of_processes * [number_of_simulations_per_process]
    # list of partial result per process
    list_of_integral_per_process = pool.map(self.simulation_integration, simulations_per_process)
    # cumulative result, aggregating partial results
    integral_per_processes = sum(list_of_integral_per_process)
    integral = integral_per_processes / self.number_of_processes
    return integral
```

Listing 24 Metoda mcs_integration_parallel

6.2.Implementacija u Golang programskom jeziku

6.2.1. Golang implementacija Monte Karlo simulacije za izračunavanje aproksimirane vrednosti broja Pi

Struktura MonteCarloSimulationPi sadrži polja: numberOfProcesses kojim se definiše veličina bafera u baferovanom kanalu, parallelFlag određuje da li je izvršvanje programa serijsko ili paralelno i experimentFlag koji govori da li smo u režimu eksperimentalnog skaliranja.

```
type MonteCarloSimulationPi struct {
        numberOfProcesses int
        parallelFlag      bool
        experimentFlag    bool
}
```

Listing 25 MonteCarloSimulationPi struktura

Go nema klase ali se mogu definisati metode nad tipovima. Metoda je funkcija koja ima specijalni receiver parametar. Metode se u osnovi ponašaju kao obične funkcije [63]. Sledeće metode su definisane nad strukturom MonteCarloSimulationPi: simulationPi, mcsPiSerial, mcsPiParallel.

Metoda simulationPi predstavlja okosnicu serijskog i paralelnog izvršavanja programa. U ovoj metodi je izvršena implementacija Monte Karlo simulacije.

```
func (monteCarloSimulationPi *MonteCarloSimulationPi) simulationPi(numberOfSimulations int, channel chan int) {
        if monteCarloSimulationPi.experimentFlag == true {
                inside := 0
                s := rand.NewSource(time.Now().UnixNano())
                r := rand.New(s)
                for i := 0; i < numberOfSimulations; i++ {
                        x := r.Float64()
                        y := r.Float64()
                        if (x*x + y*y) < 1 {
                                inside++
                        }
                }
                channel <- inside
```

Listing 26 Metoda simulationPi

Metoda mcsPiSerial predstavlja serijsku implementaciju izračunavanja aproksimirane vrednosti broja Pi.

```
func (monteCarloSimulationPi *MonteCarloSimulationPi) mcsPiSerial(numberOfSimulations int) (float64, float64) {
        startTime := time.Now()
        monteCarloSimulationPi.parallelFlag = false
        channel := make(chan int)
        go monteCarloSimulationPi.simulationPi(numberOfSimulations, channel)
        inside := <-channel
        pi := 4 * float64(inside) / float64(numberOfSimulations)
        executionTime := time.Since(startTime).Seconds()
        return pi, executionTime
}
```

Listing 27 Metoda mcsPiSerial

Metoda mcsPiParallel predstavlja paralelnu implementaciju izračunavanja aproksimirane vrednosti broja Pi.

```go
func (monteCarloSimulationPi *MonteCarloSimulationPi) mcsPiParallel(numberOfSimulations int) (float64, float64) {
	startTime := time.Now()
	monteCarloSimulationPi.parallelFlag = true
	numberOfSimulationsPerProcess := numberOfSimulations / monteCarloSimulationPi.numberOfProcesses
	/*	Buffered channels are useful when you know how many goroutines you have launched,
		want to limit the number of goroutines you will launch, or want to limit
		the amount of work that is queued up. */
	channel := make(chan int, monteCarloSimulationPi.numberOfProcesses)

	for i := 0; i < monteCarloSimulationPi.numberOfProcesses; i++ {
		go monteCarloSimulationPi.simulationPi(numberOfSimulationsPerProcess, channel)
	}

	var inside int
	for i := 0; i < monteCarloSimulationPi.numberOfProcesses; i++ {
		inside += <-channel
	}
	pi := 4 * float64(inside) / float64(numberOfSimulations)
	executionTime := time.Since(startTime).Seconds()
	return pi, executionTime

}
```

Listing 28 Metoda mcsPiParallel

6.2.2. Golang implementacija Monte Karlo simulacije za izračunavanje aproksimirane vrednosti finansijske aktive

Struktura MonteCarloSimulationFinance sadrži polja: startDate, endDate, tickerSymbol, data, timeSeries, numberOfProcesses. Polja startDate i endDate se koriste kako bi se definisao vremenski interval za koji će se prikupljati finansijski podaci preko Yahoo Finance API-a. Polje tickerSymbol predstavlja ime finansijske aktive za koju se podaci prikupljaju. Ova tri polja su neophodna za korišćenje Yahoo Finance API-a. Prikupljena vremenska serija predstavljena je poljem timeSeries, koja predstavlja sirove podatke povučene preko berzanskog API-a. Rafinirani podaci predstavljeni su poljem data. Polje numberOfProcesses definiše veličinu bafera u baferovanom kanalu.

```go
type MonteCarloSimulationFinance struct {
	numberOfProcesses int
	timeSeries        []float64
	startDate         string
	endDate           string
	tickerSymbol      string
	data              []float64
}
```

Listing 29 Struktura MonteCarloSimulationFinance

Go nema klase ali se mogu definisati metode nad tipovima. Metoda je funkcija koja ima specijalni receiver parametar. Metode se u osnovi ponašaju kao obične funkcije [63]. Pomoćna metoda dataAcquisition za prikupljanje podataka preko Yahoo Finance API-a.

```go
func (monteCarloSimulationFinance *MonteCarloSimulationFinance) dataAcquisition() {
	stock, _ := quote.NewQuoteFromYahoo(monteCarloSimulationFinance.tickerSymbol,
		monteCarloSimulationFinance.startDate, monteCarloSimulationFinance.endDate, quote.Daily, true)
	monteCarloSimulationFinance.timeSeries = stock.Close
}
```

Listing 30 Metoda dataAcquisition

Pomoćna metoda calculatePeriodicDailyReturn za obradu i diferencijaciju vremenskih serija. Ova metoda računa kontinualnu stopu prinosa.

```
// Differencing time series = Shifting and lagging time series
func (monteCarloSimulationFinance *MonteCarloSimulationFinance) calculatePeriodicDailyReturn() {
	for i := 1; i < len(monteCarloSimulationFinance.timeSeries); i++ {
		monteCarloSimulationFinance.data = append(monteCarloSimulationFinance.data,
			math.Log(monteCarloSimulationFinance.timeSeries[i]/monteCarloSimulationFinance.timeSeries[i-1]))
	}
}
```

Listing 31 Metoda calculatePeriodicDailyReturn

Pomoćna metoda calculateZScore koja se koristi za izračunavanje Z_{score}-a.

```
func (monteCarloSimulationFinance *MonteCarloSimulationFinance) calculateZScore() float64 {
	normalDistribution := gaussian.NewGaussian(0, 1)
	/*	Source code for random number generator https://play.golang.org/p/ZdFpbahgC1
		The default number generator is deterministic, so it'll
		produce the same sequence of numbers each time by default.
		To produce varying sequences, give it a seed that changes.
		Note that this is not safe to use for random numbers you
		intend to be secret, use `crypto/rand` for those.
		Seeding - Go provides a method, Seed(seed int64), that allows you
		to initialize this default sequence. Implementation is slow
		to make it faster rand.Seed(time.Now().UnixNano()) is added.
		Seed is the current time, converted to int64 by UnixNano.
		Gives constantly changing numbers. */

	// Seed
	s := rand.NewSource(time.Now().UnixNano())
	// Randomly changing numbers.
	r := rand.New(s)
	/*	Call the resulting `rand.Rand` just like the
		functions on the `rand` package. */
	return normalDistribution.Ppf(r.Float64())

}
```

Listing 32 Metoda calculateZScore

Pomoćna metoda calculateAverageDailyReturn za izračunavanje proseka vremenske serije.

```
func (monteCarloSimulationFinance *MonteCarloSimulationFinance) calculateAverageDailyReturn() float64 {
	/*	computes the weighted mean of the dataset.
		we don't have any weights (ie: all weights are 1)
		so we just pass a nil slice. */
	return stat.Mean(monteCarloSimulationFinance.data, nil)
}
```

Listing 33 Metoda calculateAverageDailyReturn

Pomoćna metoda calculateVariance za izračunavanje varijanse vremenske serije.

```
func (monteCarloSimulationFinance *MonteCarloSimulationFinance) calculateVariance() float64 {
	/*	computes the weighted variance of the dataset.
		we don't have any weights (ie: all weights are 1)
		so we just pass a nil slice. */
	return stat.Variance(monteCarloSimulationFinance.data, nil)

}
```

Listing 34 Metoda calculateVariance

Pomoćna metoda calculateStandardDeviation za izračunavanje standardne devijacije vremenske serije.

```
func (monteCarloSimulationFinance *MonteCarloSimulationFinance) calculateStandardDeviation() float64 {
	return math.Sqrt(monteCarloSimulationFinance.calculateVariance())

}
```

Listing 35 Metoda calculateStandardDeviation

Pomoćna metoda calculateDrift za izračunavanje finansijskog drifta.

```
func (monteCarloSimulationFinance *MonteCarloSimulationFinance) calculateDrift() float64 {
        return monteCarloSimulationFinance.calculateAverageDailyReturn() - monteCarloSimulationFinance.calculateVariance()/2
}
```

Listing 36 Metoda calculateDrift

Pomoćna metoda calculateRandomValue za izračunavanje slučajne vrednosti.

```
func (monteCarloSimulationFinance *MonteCarloSimulationFinance) calculateRandomValue() float64 {
        return monteCarloSimulationFinance.calculateStandardDeviation() * monteCarloSimulationFinance.calculateZScore()
}
```

Listing 37 Metoda calculateRandomValue

Metoda simulationFinance predstavlja okosnicu serijskog i paralelnog izvršavanja programa. U ovoj metodi je izvršena implementacija Monte Karlo simulacije.

```
func (monteCarloSimulationFinance *MonteCarloSimulationFinance) simulationFinance(numberOfSimulations int, predictionWindowSize int, channel chan [][]float64) {
        var prediction []float64
        var predictions [][]float64
        for i := 0; i < numberOfSimulations; i++ {
                // today's price
                prediction = append(prediction, monteCarloSimulationFinance.timeSeries[len(monteCarloSimulationFinance.timeSeries)-1])
                for j := 0; j < predictionWindowSize; j++ {
                        // Next Day's Price=Today's Price × e^(Drift+Random Value)
                        prediction = append(prediction,
                                prediction[len(prediction)-1]*math.Pow(math.E, (monteCarloSimulationFinance.calculateDrift()+monteCarloSimulationFinance.calculateRandomValue()
                }
                predictions = append(predictions, prediction)
                /*      Setting the slice to nil is the best way to clear a slice.
                        nil slices in go are perfectly well behaved and setting the slice to nil
                        will release the underlying memory to the garbage collector. */
                prediction = nil
        }

        channel <- predictions

}
```

Listing 38 Metoda simulationFinance

Metoda mcsFinanceSerial predstavlja serijsku implementaciju izračunavanja aproksimirane vrednosti finansijske aktive.

```
func (monteCarloSimulationFinance *MonteCarloSimulationFinance) mcsFinanceSerial(numberOfSimulations int, predictionWindowSize int) ([][]float64, float64) {
        startTime := time.Now()
        channel := make(chan [][]float64)
        go monteCarloSimulationFinance.simulationFinance(numberOfSimulations, predictionWindowSize, channel)
        predictions := <-channel
        executionTime := time.Since(startTime).Seconds()
        return predictions, executionTime
}
```

Listing 39 Metoda mcsFinanceSerial

Metoda mcsFinanceParallel predstavlja paralelnu implementaciju izračunavanja aproksimirane vrednosti finansijske aktive.

```
func (monteCarloSimulationFinance *MonteCarloSimulationFinance) mcsFinanceParallel(numberOfSimulations int, predictionWindowSize int) ([][][]float64, float64) {
        startTime := time.Now()
        numberOfSimulationsPerProcess := numberOfSimulations / monteCarloSimulationFinance.numberOfProcesses
        /*      Buffered channels are useful when you know how many goroutines you have launched,
                want to limit the number of goroutines you will launch, or want to limit
                the amount of work that is queued up. */
        channel := make(chan [][]float64, monteCarloSimulationFinance.numberOfProcesses)
        for i := 0; i < monteCarloSimulationFinance.numberOfProcesses; i++ {
                go monteCarloSimulationFinance.simulationFinance(numberOfSimulationsPerProcess, predictionWindowSize, channel)
        }

        var predictions [][][]float64
        for i := 0; i < monteCarloSimulationFinance.numberOfProcesses; i++ {
                prediction := <-channel
                predictions = append(predictions, prediction)

        }
        executionTime := time.Since(startTime).Seconds()
        return predictions, executionTime
}
```

Listing 40 Metoda mcsFinanceParallel

6.2.3. Golang implementacija Monte Karlo simulacije za izračunavanje aproksimirane vrednosti određenog integrala

Struktura MonteCarloSimulationIntegration sadrži polja: numberOfProcesses kojim se definiše veličina bafera u baferovanom kanalu, parallelFlag određuje da li je izvršvanje programa serijsko ili paralelno i experimentFlag koji govori da li smo u režimu eksperimentalnog skaliranja.

```
type MonteCarloSimulationIntegration struct {
        numberOfProcesses int
        parallelFlag      bool
        experimentFlag    bool
}
```

Listing 41 MonteCarloSimulationIntegration struktura

Imamo tri konstante: donja i gornja granica intervala i veličina podeoka na koji se interval deli.

```
const (
        // Lower bound of Integral.
        lowerBound = 1.0
        // Upper bound of Integral.
        upperBound = 2.0
        // The area under the graph of a function can be found by adding slices that approach zero in width.
        sliceSize = 0.01
)
```

Listing 42 MonteCarloSimulationIntegration konstante

Go nema klase ali se mogu definisati metode nad tipovima. Metoda je funkcija koja ima specijalni receiver parametar. Metode se u osnovi ponašaju kao obične funkcije [63]. Sledeće metode su definisane nad strukturom MonteCarloSimulationIntegration. Metoda function je integrand tj. funkcija za koju tražimo određeni integrala na nekom intervalu.

```
func (MonteCarloSimulationIntegration *MonteCarloSimulationIntegration) function(x float64) float64 {
        return 2 * x
}
```

Listing 43 Metoda function

Metoda simulationIntegration predstavlja okosnicu serijskog i paralelnog izvršavanja programa. U ovoj metodi je izvršena implementacija Monte Karlo simulacije.

```
func (MonteCarloSimulationIntegration *MonteCarloSimulationIntegration) simulationIntegration(numberOfSimulations int, channel chan float64) {
        if MonteCarloSimulationIntegration.experimentFlag == true {
                // Points under the graph of a function.
                below := 0
                lowerBoundInterval := lowerBound
                upperBoundInterval := upperBound
                // Define the interval between the lower and upper bound.
                var x []float64
                // Function Values
                var y []float64
                // Maximum of the function f(x) on the interval[lower_bound, upper_bound]
                fMax := MonteCarloSimulationIntegration.function(lowerBound)
                for lowerBoundInterval < upperBoundInterval {
                        x = append(x, lowerBoundInterval)
                        t := MonteCarloSimulationIntegration.function(lowerBoundInterval)
                        y = append(y, t)
                        if t > fMax {
                                fMax = t
                        }
                        lowerBoundInterval += sliceSize

                }

                s := rand.NewSource(time.Now().UnixNano())
                r := rand.New(s)
                for i := 0; i < numberOfSimulations; i++ {
                        xRand := lowerBound + (upperBound-lowerBound)*r.Float64()
                        yRand := 0 + fMax*r.Float64()
                        if yRand < MonteCarloSimulationIntegration.function(xRand) {
                                below++
                        }
                }
                // Rectangle area that surrounds the area under the graph of a function.
                a := upperBound - lowerBound
                b := fMax - 0
                rectangleArea := a * b
                // bellow = Points under the graph of a function.
                // number_of_simulations = Total number of points = Points inside rectangle
                proportion := float64(below) / float64(numberOfSimulations)
                integral := float64(proportion) * float64(rectangleArea)
                channel <- integral
```

Listing 44 Metoda simulationIntegration

Metoda mcsIntegrationSerial predstavlja serijsku implementaciju izračunavanja aproksimirane vrednosti integrala.

```
func (MonteCarloSimulationIntegration *MonteCarloSimulationIntegration) mcsIntegrationSerial(numberOfSimulations int) (float64, float64) {
	startTime := time.Now()
	MonteCarloSimulationIntegration.parallelFlag = false
	channel := make(chan float64)
	go MonteCarloSimulationIntegration.simulationIntegration(numberOfSimulations, channel)
	integral := <-channel
	executionTime := time.Since(startTime).Seconds()
	return integral, executionTime
}
```

Listing 45 Metoda mcsIntegrationSerial

Metoda mcsIntegrationParallel predstavlja paralelnu implementaciju izračunavanja aproksimirane vrednosti integrala.

```
func (MonteCarloSimulationIntegration *MonteCarloSimulationIntegration) mcsIntegrationParallel(numberOfSimulations int) (float64, float64) {
	startTime := time.Now()
	MonteCarloSimulationIntegration.parallelFlag = true
	numberOfSimulationsPerProcess := numberOfSimulations / MonteCarloSimulationIntegration.numberOfProcesses
	/*	Buffered channels are useful when you know how many goroutines you have launched,
		want to limit the number of goroutines you will launch, or want to limit
		the amount of work that is queued up. */
	channel := make(chan float64, MonteCarloSimulationIntegration.numberOfProcesses)
	// partial result per process
	for i := 0; i < MonteCarloSimulationIntegration.numberOfProcesses; i++ {
		go MonteCarloSimulationIntegration.simulationIntegration(numberOfSimulationsPerProcess, channel)
	}

	var integralPerProcesses float64
	// cumulative result, aggregating partial results
	for i := 0; i < MonteCarloSimulationIntegration.numberOfProcesses; i++ {
		integralPerProcesses += <-channel
	}
	integral := float64(integralPerProcesses) / float64(MonteCarloSimulationIntegration.numberOfProcesses)
	executionTime := time.Since(startTime).Seconds()
	return integral, executionTime

}
```

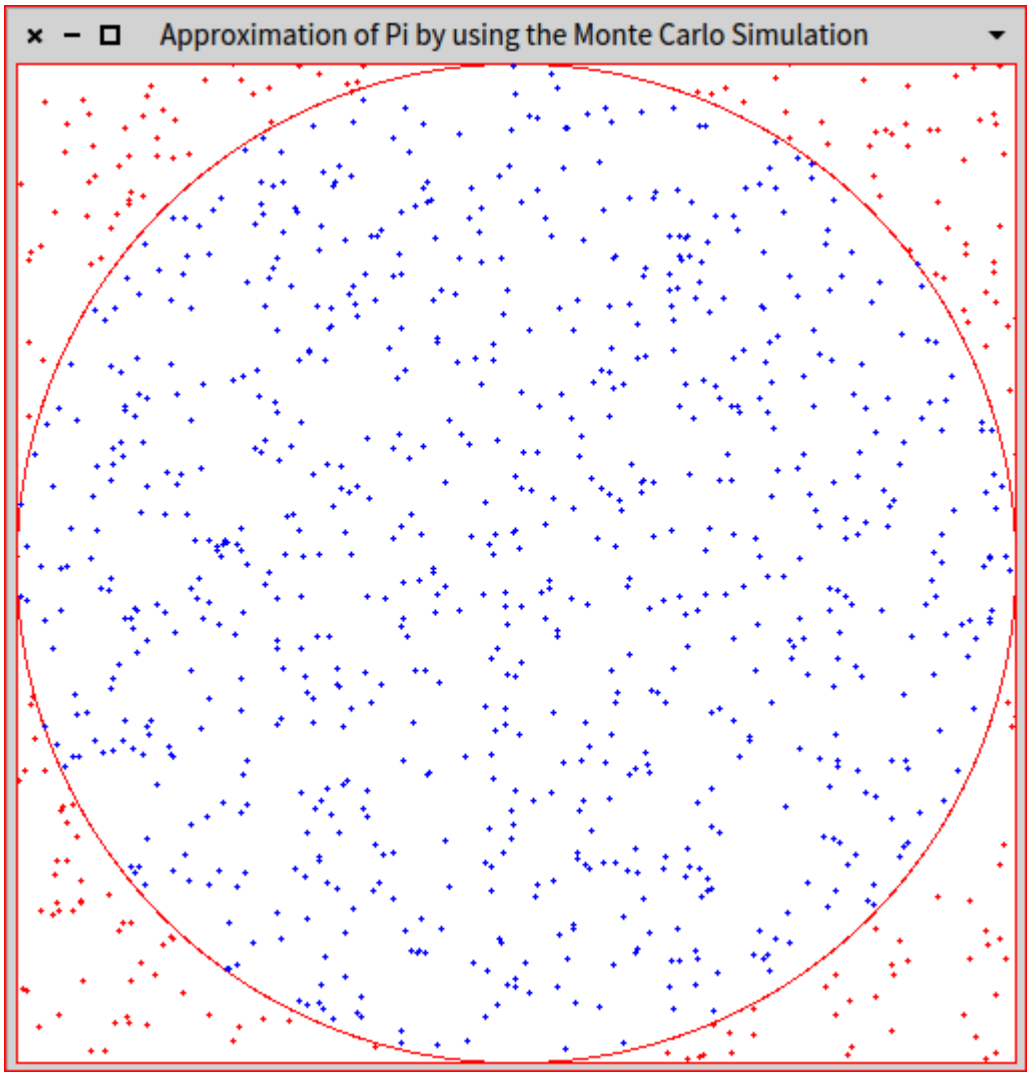
Listing 46 Metoda mcsIntegrationParallel

7. Vizuelizacija implementiranog sistema

Nakon što su izvršene Monte Karlo simulacije i eksportovani dobijeni rezultati potrebno je izvršiti vizuelizaciju dobijenih rezultata. Cilj ovog poglavlja je da se prikaže kako je programski izvršena vizuelizacija Monte Karlo simulacija po oblastima primene. U ovom delu biće prikazana vizuelizacija u Pharo programskom jeziku i Roassal graphic engine-u.

7.1. Vizuelizacija Monte Karlo simulacije za izračunavanje aproksimirane vrednosti broja Pi

Vizuelizacija Monte Karlo simulacije za izračunavanje aproksimirane vrednosti broja Pi izvršena je pomoću Pharo programskog jezika. Dat je prikaz vizuelizacije Monte Karlo simulacije za izračunavanje aproksimirane vrednosti broja Pi kada je broj simulacija n = 1000. Kako broj simulacija raste tako raste gustina plavih i crvenih tačaka na slici. Ukupan broj tačaka jednak je ukupnom broju simulacija. Vizuelizacija se pokreće kopiranjem sadražaja fajla MonteCarloSimulationPiPlayground.txt u Pharo Playground.

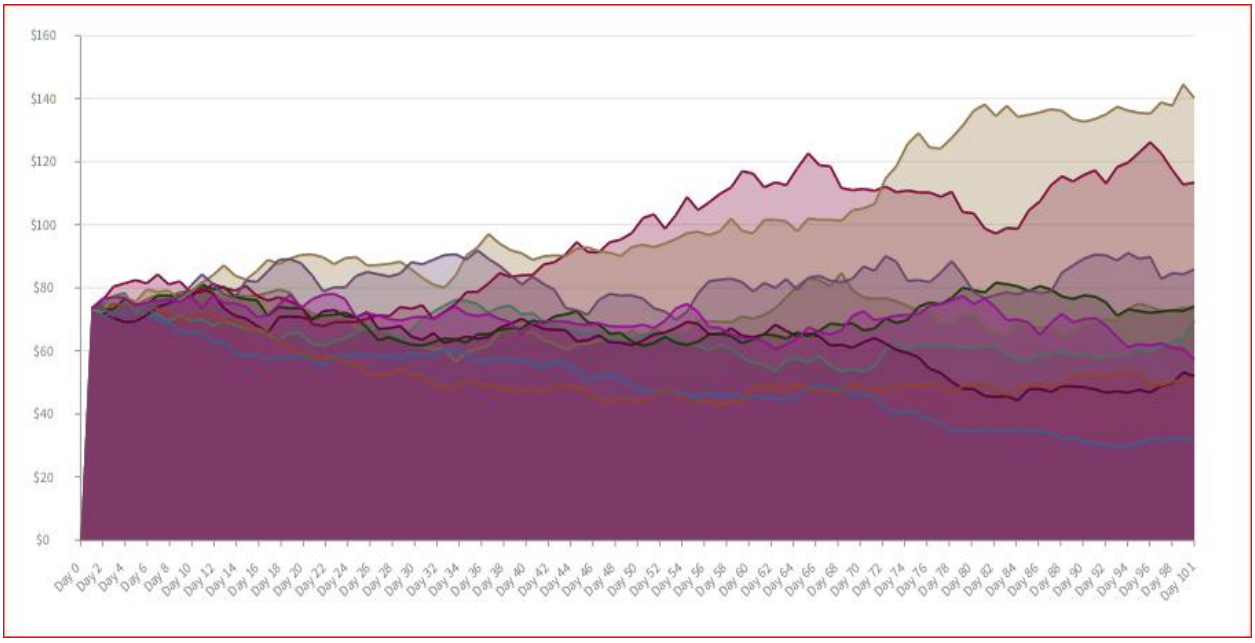


Slika 16 Vizuelizacija Monte Karlo simulacije za izračunavanje aproksimirane vrednosti broja Pi u Pharo programskom jeziku

7.2. Vizuelizacija Monte Karlo simulacije za izračunavanje aproksimirane vrednosti finansijske aktive

Vizuelizacija Monte Karlo simulacije za izračunavanje aproksimirane vrednosti finansijske aktive izvršena je pomoću Roassal graphic engine-a. Dat je prikaz vizuelizacije Monte Karlo simulacije za izračunavanje aproksimirane vrednosti finansijske aktive kada je broj simulacija n = 10, a veličina prediktivnog prozora tj. broja dana za koje se vrši predikcija w = 100. Svaka simulacija predstavlja jednu krivu na fen dijagramu. Veličina prediktivnog prozora je zapravo broj dana u budućnosti za koje se vrši predikcija. Na x-osi se nalazi veličina prediktivnog prozora tj.vreme dok se na y-osi nalazi cena. Kako broj simulacija raste tako raste gustina

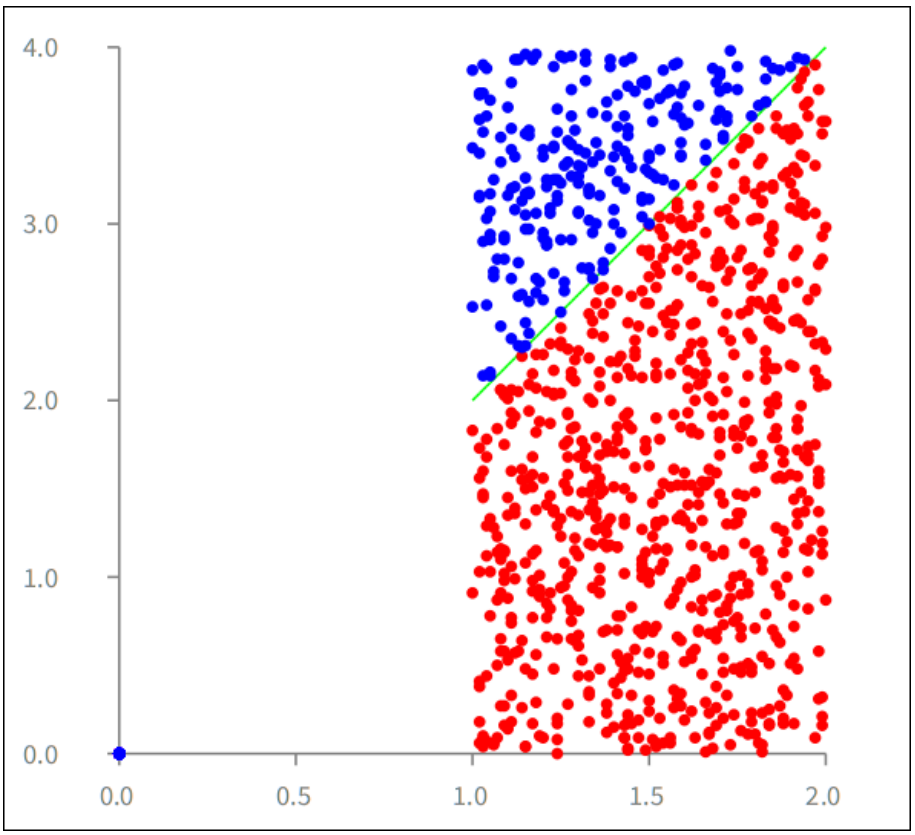
fen dijagrama na slici. Na osnovu fen dijagrama investitori se opredeljuju za one predikcije koje se nalaze u koridoru +/- jedna standardna devijacija zato što su to najverovatnije predikcije. Vizuelizacija se pokreće kopiranjem sadražaja fajla MonteCarloSimulationFinancePlayground.txt u Pharo Playground.



Slika 17 Vizuelizacija Monte Karlo simulacije za izračunavanje aproksimirane vrednosti finansijske aktive u Roassal graphic engine-u

7.3.Vizuelizacija Monte Karlo simulacije za izračunavanje aproksimirane vrednosti određenog integrala

Vizuelizacija Monte Karlo simulacije za izračunavanje aproksimirane vrednosti određenog integrala izvršena je pomoću Pharo programskog jezika. Dat je prikaz vizuelizacije Monte Karlo simulacije za izračunavanje aproksimirane vrednosti određenog integrala kada je broj simulacija n = 1000. Kako broj simulacija raste tako raste gustina plavih i crvenih tačaka na slici. Ukupan broj tačaka jednak je ukupnom broju simulacija. Vizuelizacija se pokreće kopiranjem sadražaja fajla MonteCarloSimulationIntegrationPlayground.txt u Pharo Playground.



Slika 18 Vizuelizacija Monte Karlo simulacije za izračunavanje aproksimirane vrednosti određenog integrala u Pharo programskom jeziku

8. Verifikacija rešenja (eksperimenti skaliranja)

8.1.Teorijski koncepti eksperimenata skaliranja

Eksperimenti skaliranja predstavljaju važan deo projektnog rešenja. U eksperimentima skaliranja bilo je neophodno ispitati kako ubrzanje koje se dobija paralelizacijom koda zavisi od broja procesnih jedinica u slučaju jakog skaliranja odnosno kako ubrzanje zavisi od broja procesnih jedinica i obima posla koji se izvršava u slučaju slabog skaliranja. U Python programskoj realizaciji za procesne jedinice koristi se Pool procesa iz standardne multiprocessing biblioteke, dok se u Golang implementaciji koriste Go rutine. Tokom eksperimentisanja primećeno je da serijska i paralelna verzija simulacija u Golang-u imaju kraće vreme izvršavanja u odnosu na simulacije u Python-u. Brže izvršavanje Golang programa u odnosu na Python programe prepisuje se činjenici da procesi u Python-u ne dele istu memoriju i da gube vreme na preključivanje [81]. Go rutine dele istu memoriju i ne gube vreme na preključivanje i zato je izvršavanje Golang programa nekoliko desetina puta brže u odnosu na izvršavanje programa u Python-u.

$$\text{speedup} = \frac{t_1}{t_N}$$

Formula 23 Ubrzanje paralelnih programa [8]

Ubrzanje se računa kao odnos vremena izvršavanja programa na jednoj procesnoj jedinici (serijski program) i vremena izvršavanja programa na N procesnih jedinica za istu količinu posla. U slučaju da je problem savršeno idealan za paralelizaciju tj. da se kompletan program može paralelizovati tada je ubrzanje koje se postiže proporcionalno broj procesnih jedinica. Kako u eksperimentima jakog tako i u eksperimentima slabog skaliranja tri veličine igraju ključnu ulogu na ostvarene performanse: s – procenat ukupnog vremena izvršenja serijskog programa koje se ne može paralelizovati, p – procenat ukupnog vremena izvršenja serijskog programa koje se može paralelizovati i N – broj procesnih jedinica.

$$s + p = 1$$

Formula 24 Proporcija ukupnog vremena izvršavanja programa [8]

Kako je celokupan serijski program (važi za sve tri primene Monte Karlo simulacije) moguće paralelizovati odatle sledi da je s = 0 i p = 1 odnosno teorijski maksimum ubrzanja jednak je broj procesnih jedinica N.

8.2.Arhitektura sistema nad kojom su vršeni eksperimenti skaliranja

Cilj eksperimenata jakog i slabog skaliranja je da pokažemo kako se implementirani algoritmi ponašaju na stvanom hardveru. Stoga je arhitektura sistema nad kojom su vršeni eksperimenti skaliranja od izuzetne važnosti zato što direktno utiče na rezultate skaliranja.

Model: Dell Inspiron 15 3000

Procesor: Intel(R) Core(TM) i5-4210U CPU @ 1.70GHz, 1701 Mhz, 2 Core(s), 4 Logical Processor(s)

RAM memorija: 4 GB, 1600MHz, DDR3L

Grafička kartica: NVIDIA GeForce 920M

Cache memorija: 3 MB

Hard drive: 500 GB HDD

Operativni sistem: Microsoft Windows 7 Professional Version 6.1.7601 Service Pack 1 Build 7601

8.3.Teorijski koncepti eksperimenata jakog skaliranja (Amdalov zakon)

U eksperimentima jakog skaliranja ubrzanje koje se postiže paralelizacijom posmatra se u odnosu na promenu broja procesnih jedinica dok se količina posla drži fiksnom. U stručnoj literaturi ovaj tip eksperimenta poznat je kao Amdalov zakon [10]. Ubrzanje koje se postiže prema Amdalovom zakonu računa se prema sledećoj formuli:

$$\text{Amdahl's law speedup} = \frac{1}{(s + \frac{p}{N})}$$

Formula 25 Ubrzanje prema Amdalovom zakonu [8]

Kako je u konkretnom slučaju s = 0 i p = 1 formulu za ubrzanje prema Amdalovom zakonu možemo da korigujemo:

$$\text{Amdahl's law speedup} = \frac{1}{(s + \frac{p}{N})} = \frac{1}{(0 + \frac{1}{N})} = N$$

Formula 26 Teorijski maksimum ubrzanja po Amdalovom zakonu

Zaključujemo da je teorijski maksimum ubrzanja po Amdalovom zakonu jednak broju procesnih jedinica N.

8.4.Teorijski koncepti eksperimenata slabog skaliranja (Gustafsonov zakon)

U eksperimentima slabog skaliranja ubrzanje koje se postiže paralelizacijom posmatra se u odnosu na promenu broja procesnih jedinica i proporcionalnog povećanja količina posla. Rast broja procesorskih jezgara isparaćen je proporcionalnim rastom količine posla te se na taj način postiže konstantan posao po procesorskom jezgru. U stručnoj literaturi ovaj tip eksperimenta poznat je kao Gustafsonov zakon [11]. Ubrzanje koje se postiže prema Gustafsonov zakonu računa se prema sledećoj formuli:

$$\text{Gustafson's law speedup} = s + p \times N$$

Formula 27 Ubrzanje prema Gustafsonovom zakonu [8]

Kako je u konkretnom slučaju s = 0 i p = 1 formulu za ubrzanje prema Gustafsonovom zakonu možemo da korigujemo:

$$\text{Gustafson's law speedup} = s + p \times N = 0 + 1 \times N = N$$

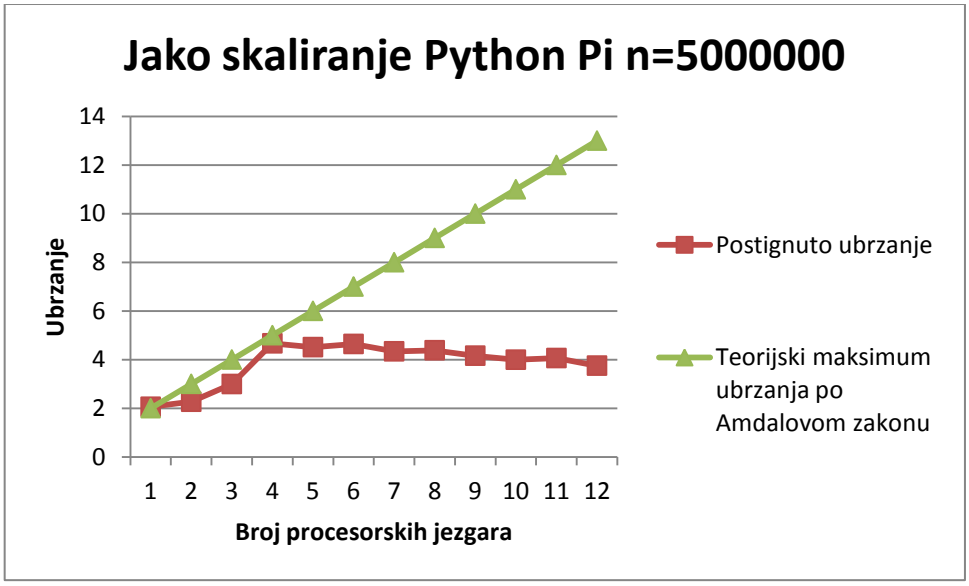
Formula 28 Teorijski maksimum ubrzanja po Gustafsonovom zakonu

Zaključujemo da je teorijski maksimum ubrzanja po Gustafsonovom zakonu jednak broju procesnih jedinica N.

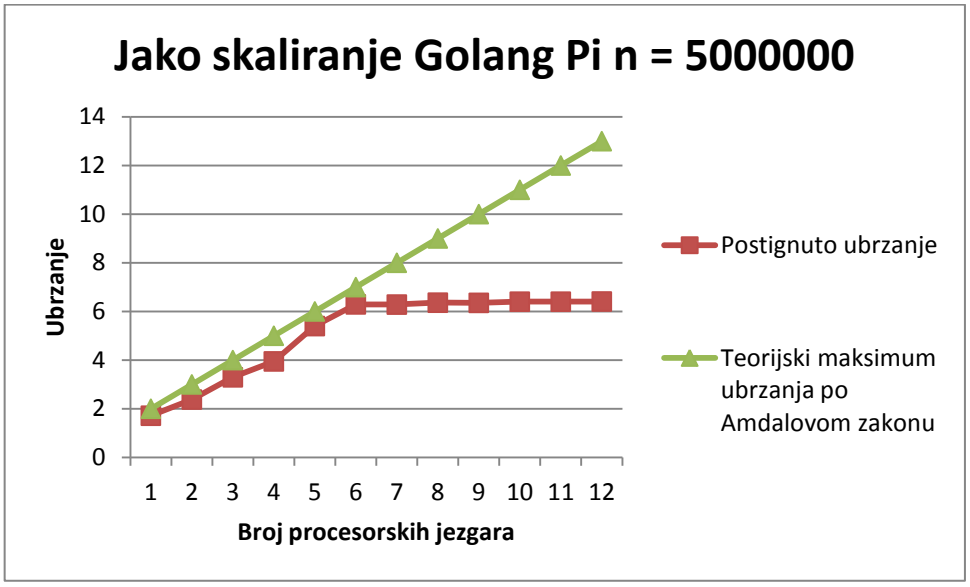
8.5.Verifikacija rešenja Monte Karlo simulacije za izračunavanje aproksimirane vrednosti broja Pi

8.5.1. Eksperimenti jakog skaliranja Monte Karlo simulacije za izračunavanje aproksimirane vrednosti broja Pi

Na x-osi se nalazi broj procesnih jedinica (broj procesorskih jezgara) dok se na y-osi nalazi ubrzanje. Zelena linija predstavlja teorijski maksimum ubrzanja po Amdalovom zakonu (idealno skaliranja) dok crvena linja predstavlja ostvareno ubrzanje. Go implementacija postiže bolje rezultate zato što go rutine dele istu memoriju i ne gube vreme na preključivanje.



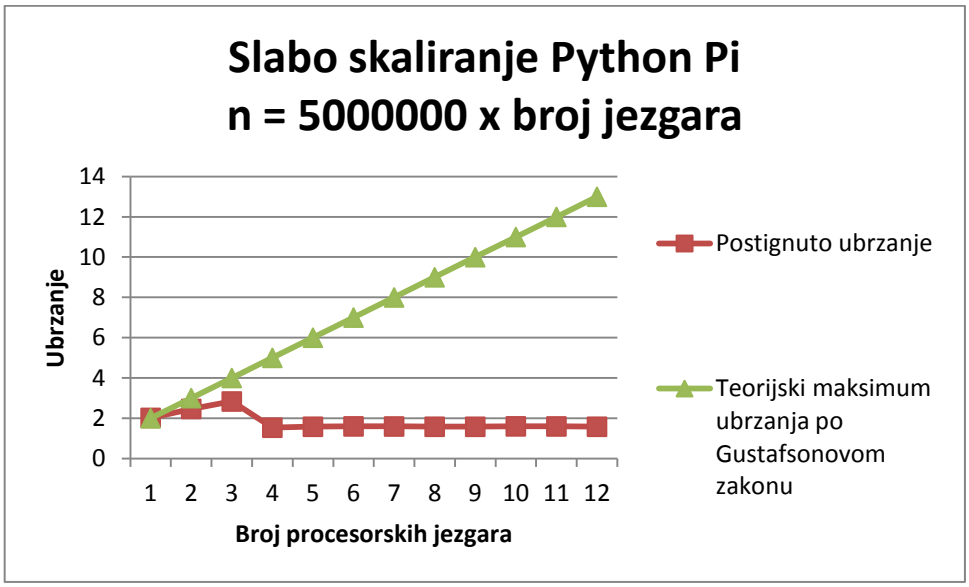
Slika 19 Jako skaliranje Python Pi n = 5000000 simulacija



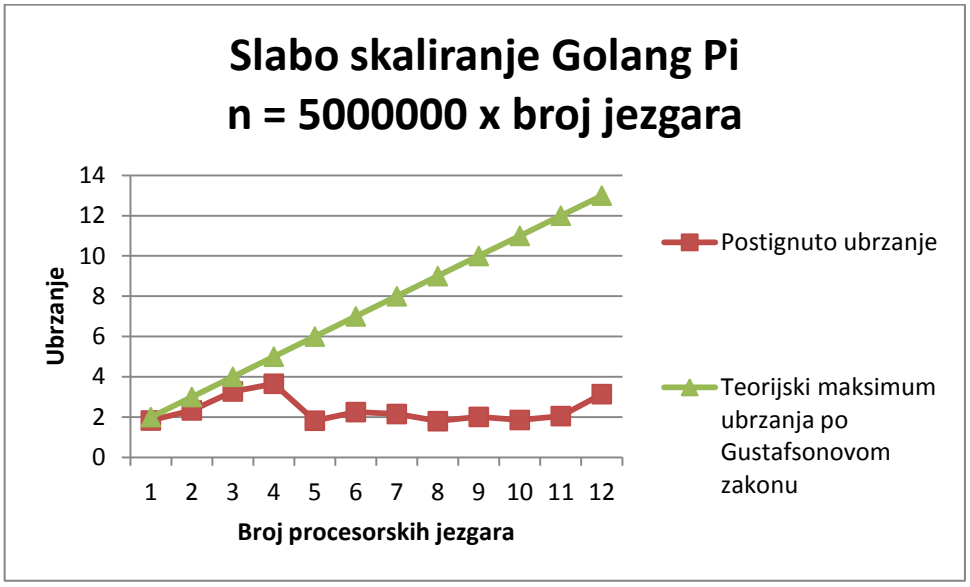
Slika 20 Jako skaliranje Golang Pi n = 5000000 simulacija

8.5.2. Eksperimenti slabog skaliranja Monte Karlo simulacije za izračunavanje aproksimirane vrednosti broja Pi

Na x-osi se nalazi broj procesnih jedinica (broj procesorskih jezgara) dok se na y-osi nalazi ubrzanje. Zelena linija predstavlja teorijski maksimum ubrzanja po Gustafsonovom zakonu (idealno skaliranja) dok crvena linja predstavlja ostvareno ubrzanje. Go implementacija postiže bolje rezultate zato što go rutine dele istu memoriju i ne gube vreme na preključivanje.



Slika 21 Slabo skaliranje Python Pi n = (5000000 x broj jezgara) simulacija

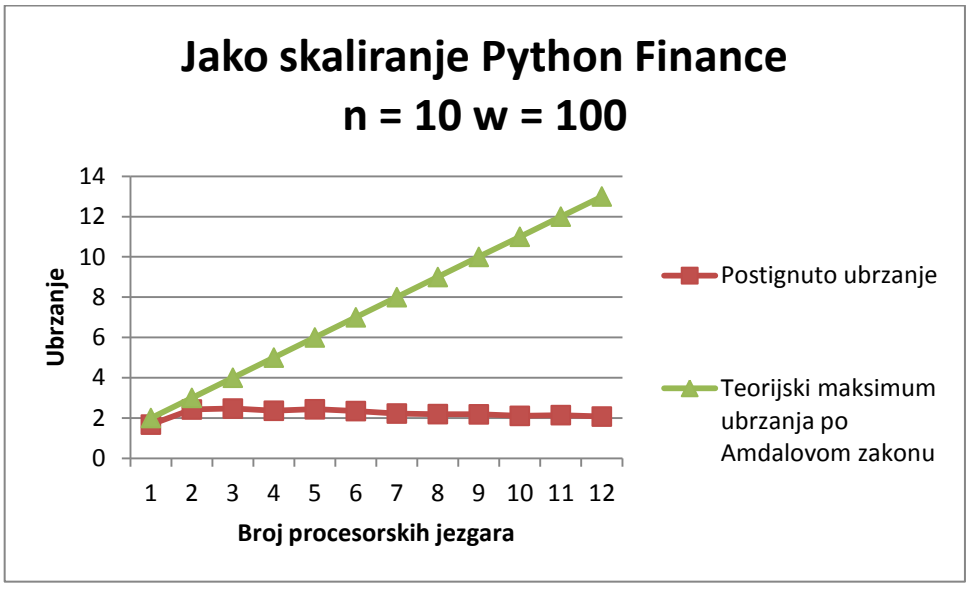


Slika 22 Slabo skaliranje Golang Pi n = (5000000 x broj jezgara) simulacija

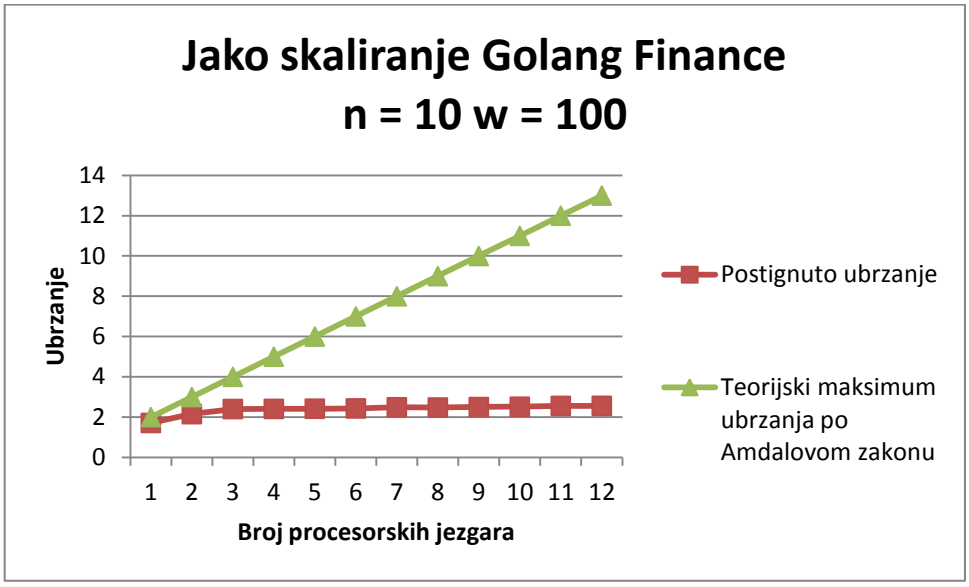
8.6.Verifikacija rešenja Monte Karlo simulacije za izračunavanje aproksimirane vrednosti finansijske aktive

8.6.1. Eksperimenti jakog skaliranja Monte Karlo simulacije za izračunavanje aproksimirane vrednosti finansijske aktive

Na x-osi se nalazi broj procesnih jedinica (broj procesorskih jezgara) dok se na y-osi nalazi ubrzanje. Zelena linija predstavlja teorijski maksimum ubrzanja po Amdalovom zakonu (idealno skaliranja) dok crvena linja predstavlja ostvareno ubrzanje. Go implementacija postiže bolje rezultate zato što go rutine dele istu memoriju i ne gube vreme na preključivanje. Simulacija cene finansijske aktive određena je brojem ponavljanja n i veličinom prediktivnog prozora w.



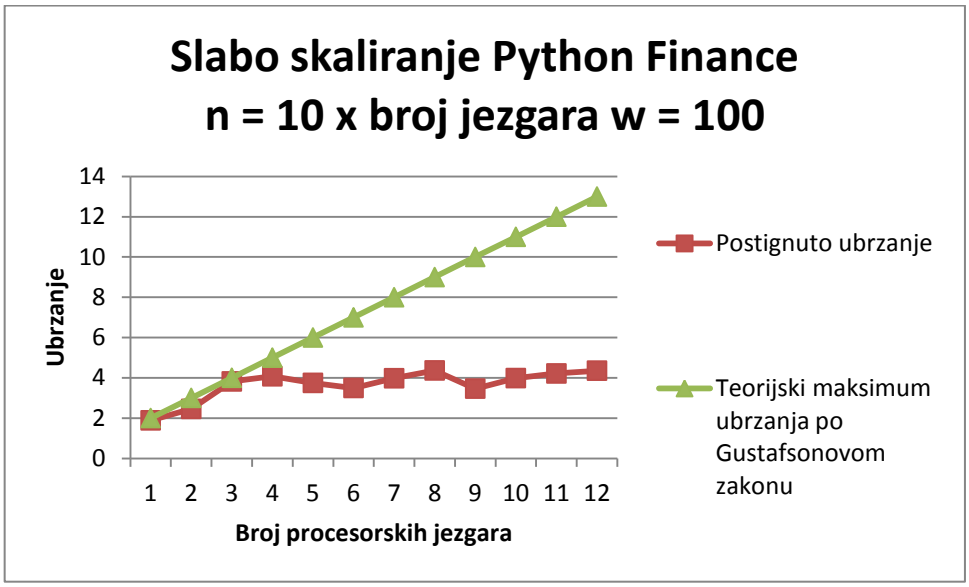
Slika 23 Jako skaliranje Python Finance n = 10 simulacija i w = 100 dana u budućnosti



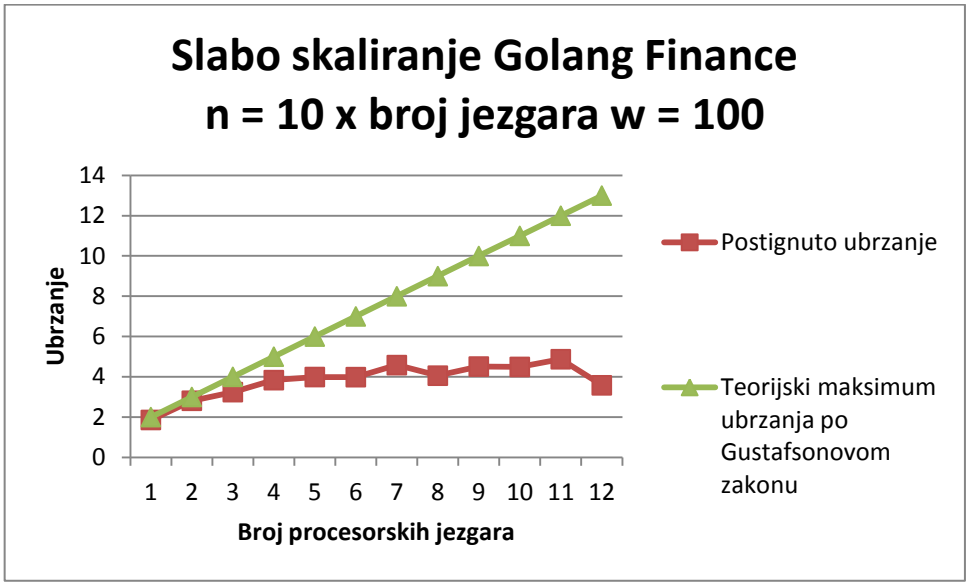
Slika 24Jako skaliranje Golang Finance n = 10 simulacija i w = 100 dana u budućnosti

8.6.2. Eksperimenti slabog skaliranja Monte Karlo simulacije za izračunavanje aproksimirane vrednosti finansijske aktive

Na x-osi se nalazi broj procesnih jedinica (broj procesorskih jezgara) dok se na y-osi nalazi ubrzanje. Zelena linija predstavlja teorijski maksimum ubrzanja po Gustafsonovom zakonu (idealno skaliranja) dok crvena linja predstavlja ostvareno ubrzanje. Go implementacija postiže bolje rezultate zato što go rutine dele istu memoriju i ne gube vreme na preključivanje. Simulacija cene finansijske aktive određena je brojem ponavljanja n i veličinom prediktivnog prozora w.



Slika 25 Slabo skaliranje Python Finance n = (10 x broj jezgara) simulacija i w = 100 dana u budućnosti

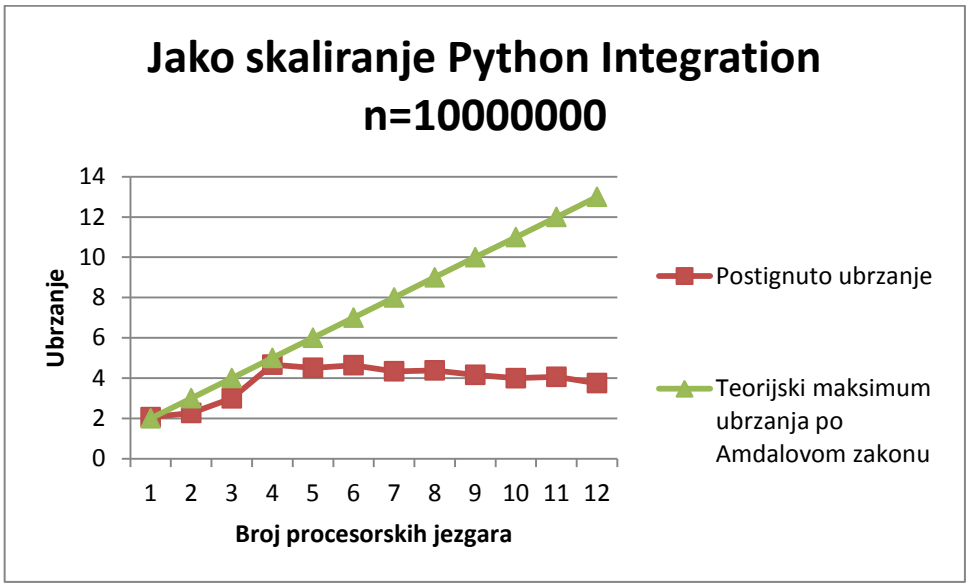


Slika 26 Slabo skaliranje Golang Finance n = (10 x broj jezgara) simulacija i w = 100 dana u budućnosti

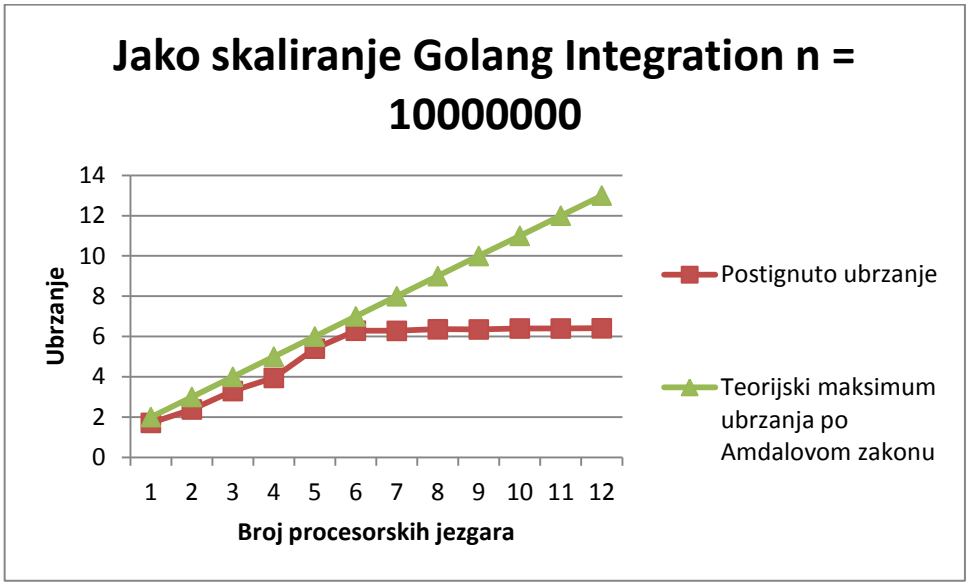
8.7.Verifikacija rešenja Monte Karlo simulacije za izračunavanje aproksimirane vrednosti određenog integrala

8.7.1. Eksperimenti jakog skaliranja Monte Karlo simulacije za izračunavanje aproksimirane vrednosti određenog integrala

Na x-osi se nalazi broj procesnih jedinica (broj procesorskih jezgara) dok se na y-osi nalazi ubrzanje. Zelena linija predstavlja teorijski maksimum ubrzanja po Amdalovom zakonu (idealno skaliranja) dok crvena linja predstavlja ostvareno ubrzanje. Go implementacija postiže bolje rezultate zato što go rutine dele istu memoriju i ne gube vreme na preključivanje.



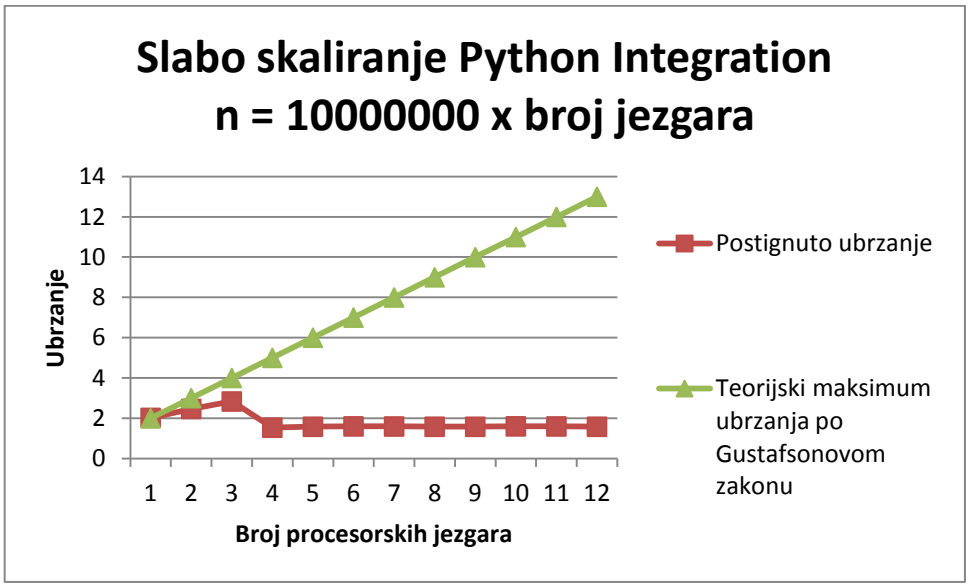
Slika 27 Jako skaliranje Python Integration n = 10000000 simulacija



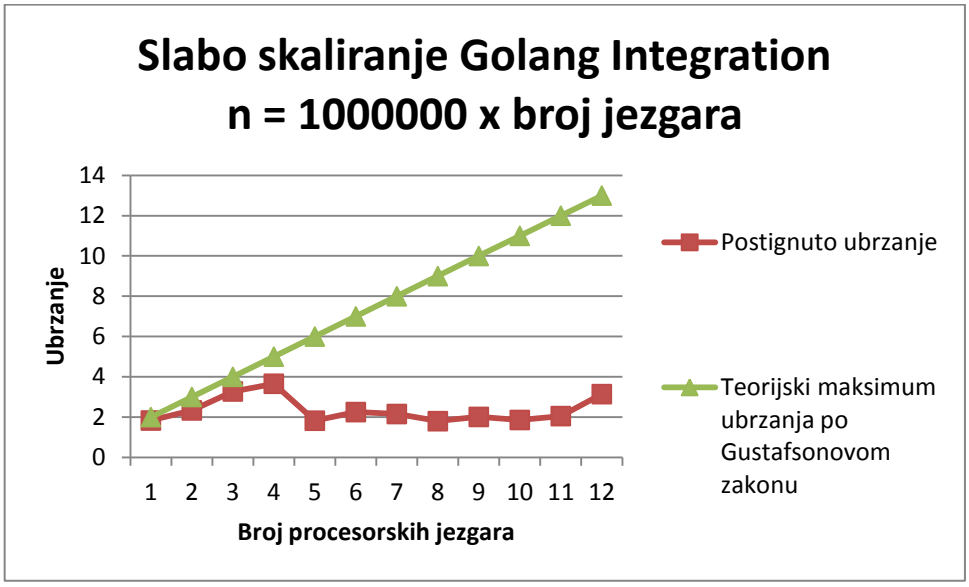
Slika 28 Jako skaliranje Golang Integration n = 10000000 simulacija

8.7.2. Eksperimenti slabog skaliranja Monte Karlo simulacije za izračunavanje aproksimirane vrednosti određenog integrala

Na x-osi se nalazi broj procesnih jedinica (broj procesorskih jezgara) dok se na y-osi nalazi ubrzanje. Zelena linija predstavlja teorijski maksimum ubrzanja po Gustafsonovom zakonu (idealno skaliranja) dok crvena linja predstavlja ostvareno ubrzanje. Go implementacija postiže bolje rezultate zato što go rutine dele istu memoriju i ne gube vreme na preključivanje.



Slika 29 Slabo skaliranje Python Integration n = (10000000 x broj jezgara) simulacija



Slika 30 Slabo skaliranje Golang Integration n = (10000000 x broj jezgara) simulacija

9. Zaključak

Cilj ovog istraživanja bio je da se pokaže različita primena Monte Karlo simulacije. U ovom radu Monte Karlo simulacija je korišćena za: izračunavanje aproksimirane vrednosti broja Pi, izračunavanje aproksimirane vrednosti finansijske aktive i izračunavanje aproksimirane vrednosti određenog integrala.

Za realizaciju softverskog rešenja korićena je tehnika paralelnog programiranja uz upotrebu sledećih programskih jezika: Python, Golang i Pharo (Roassal).

Kompletan rad može da se podeli u tri velike tematske celine. Prvoj tematskoj celini pripada pikupljanje i obrada ulaznih podataka. Podaci koji se koriste u simulacijama su pseudo-slučajno generisani ili se povlače preko Yahoo Finance API-a. Obrađeni podaci predstavljaju ulaz u algoritme Monte Karlo simulacija. Pored prikupljanja i obrade ulaznih podataka ovoj tematskoj celini pripada i razvoja algoritama Monte Karlo simulacija. Prva tematska celina je realizovana u Python i Golang programski jezicima. Druga tematska celina posvećena je vizuelizaciji dobijenih rezultata iz simulacija. Vizuelizacija podataka je od izuzetnog značaja za bolje razumevanje implementiranih algoritama. Na ovaj način formira se tok podataka. Sirovi podaci ulaze u algoritme simulacija, bivaju obrađeni i kao takvi se vraćaju na vizuelizaciju. Druga tematska celina realizovana je u Pharo programskom jeziku i Roassal graphic engine-u. Treće tematska celina bavi se problematikom eksperimentalnog skaliranja. Eksperimenti skaliranja obuhvataju eksperimente jakog i slabog skaliranja. Cilj eksperimenata jakog i slabog skaliranja je da pokažemo kako se ovi algoritmi ponašanja na stvanom hardveru. Treća tematska celina je realizovana u Python i Golang programskim jezicima.

Ideja je bila da se razvije jedinstven sistem koji će omogućiti korisniku da se bolje upozana sa mogućnostima Monte Karlo simulacija. Pored teorijskih koncepata poseban akcenat u radu je bio na praktičnoj primeni Monte Karlo simulacije.

Zbog široke aplikativnosti Monte Karlo simulacije ideja je da se postojeće istraživanje u budućnosti proširi novim oblastima primene, ali i da se eksperimentalno skaliranje vrši na hardverskoj opremi boljih performansi kako bi se postigli još bolji rezultati.

10. Literatura

[1] https://en.wikipedia.org/wiki/Monte_Carlo_method, Datum pristupanja 23.07.2020.

[2] https://en.wikipedia.org/wiki/Pseudorandom_number_generator#Potential_problems_with_deterministic_generators, Datum pristupanja 15.09.2020.

[3] https://www.investopedia.com/terms/f/financialasset.asp, Datum pristupanja 11.8.2020.

[4] https://www.python.org, Datum pristupanja 10.09.2020.

[5] https://golang.org, Datum pristupanja 17.09.2020.

[6] https://pharo.org, Datum pristupanja 26.09.2020.

[7] http://agilevisualization.com, Datum pristupanja 30.07.2020.

[8] https://www.kth.se/blogs/pdc/2018/11/scalability-strong-and-weak-scaling/, Datum pristupanja 01.09.2020.

[9] https://en.wikipedia.org/wiki/High-performance_technical_computing, Datum pristupanja 28.08.2020.

[10] https://en.wikipedia.org/wiki/Amdahl%27s_law, Datum pristupanja 03.09.2020.

[11] https://en.wikipedia.org/wiki/Gustafson%27s_law, Datum pristupanja 03.09.2020.

[12] https://en.wikipedia.org/wiki/Manhattan_Project, Datum pristupanja 12.09.2020.

[13] https://en.wikipedia.org/wiki/Enrico_Fermi, Datum pristupanja 13.09.2020.

[14] https://en.wikipedia.org/wiki/John_von_Neumann, Datum pristupanja 13.09.2020.

[15] https://en.wikipedia.org/wiki/Nicholas_Metropolis, Datum pristupanja 13.09.2020.

[16] https://en.wikipedia.org/wiki/Stanislaw_Ulam, Datum pristupanja 13.09.2020.

[17] https://en.wikipedia.org/wiki/Law_of_large_numbers#:~:text=In%20probability%20theory%2C%20the%20law,a%20large%20number%20of%20times.&text=For%20example%2C%20while%20a%20casino,a%20large%20number%20of%20spins., Datum pristupanja 05.09.2020.

[18] https://en.wikipedia.org/wiki/Simple_random_sample, Datum pristupanja 16.08.2020.

[19] https://confounding.net/2012/03/12/thats-not-how-the-law-of-large-numbers-works/, Datum pristupanja 09.08.2020.

[20] https://en.wikipedia.org/wiki/Random_number, Datum pristupanja 15.09.2020.

[21] http://metodologijamentus.weebly.com/uploads/5/1/9/7/51973983/14._uzorkovanje.pdf, Datum pristupanja 10.09.2020.

[22] https://www.investopedia.com/terms/s/stock.asp#:~:text=Bonds-,What%20Is%20a%20Stock%3F,stock%20are%20called%20"shares.", Datum pristupanja 08.09.2020.

[23] https://www.investopedia.com/terms/b/bond.asp, Datum pristupanja 08.09.2020.

[24] https://www.investopedia.com/terms/d/derivative.asp, Datum pristupanja 08.09.2020.

[25] https://www.investopedia.com/terms/c/cryptocurrency.asp, Datum pristupanja 08.09.2020.

[26] https://en.wikipedia.org/wiki/Time_series#:~:text=A%20time%20series%20is%20a,sequence%20of%20discrete-time%20data., Datum pristupanja 20.07.2020.

[27] https://www.investopedia.com/terms/r/rational-behavior.asp#:~:text=Rational%20behavior%20refers%20to%20a,or%20utility%20for%20an%20individual.&text=Most%20classical%20economic%20theories%20are,an%20activity%20are%20behaving%20rationally., Datum pristupanja 10.09.2020.

[28] https://en.wikipedia.org/wiki/Exponential_growth, Datum pristupanja 26.08.2020.

[29] https://www.investopedia.com/terms/v/volatility.asp, Datum pristupanja 14.08.2020.

[30] https://www.investopedia.com/terms/r/rateofreturn.asp, Datum pristupanja 13.08.2020.

[31] https://www.investopedia.com/terms/m/montecarlosimulation.asp, Datum pristupanja 20.07.2020.

[32] https://en.wikipedia.org/wiki/Standard_deviation, Datum pristupanja 11.08.2020.

[33] https://en.wikipedia.org/wiki/Variance, Datum pristupanja 11.08.2020.

[34] https://www.simplypsychology.org/z-table.html#:~:text=A%20z-table%2C%20also%20called,standard%20normal%20distribution%20(SND)., Datum pristupanja 11.08.2020.

[35] https://www.investopedia.com/terms/b/bell-curve.asp, Datum pristupanja 11.08.2020.

[36] https://en.wikipedia.org/wiki/Fan_chart_(time_series), Datum pristupanja 31.07.2020.

[37] https://www.mathsisfun.com/calculus/integration-definite.html#:~:text=A%20Definite%20Integral%20has%20start,Indefinite%20Integral, Datum pristupanja 10.09.2020.

[38] https://en.wikipedia.org/wiki/Integral, Datum pristupanja 05.09.2020.

[39] https://www.mathsisfun.com/calculus/integration-rules.html, Datum pristupanja 12.09.2020.

[40] https://finance.yahoo.com, datum pristupanja 12.08.2020.

[41] https://github.com/vladaindjic/ntp-2020/blob/master/napredni-python/code/konkurentno_programiranje/pregled.md, Datum pristupanja 02.08.2020.

[42] https://en.wikipedia.org/wiki/Guido_van_Rossum, Datum pristupanja 13.09.2020.

[43] https://en.wikipedia.org/wiki/Python_(programming_language), 29.09.2020.

[44] http://www.igordejanovic.net/courses/tech/Python/, Datum pristupanja 31.07.2020.

[45] https://docs.python.org/3/library/random.html, Datum pristupanja 03.10.2020.

[46] https://docs.python.org/3/library/time.html, Datum pristupanja 03.10.2020.

[47] http://www.igordejanovic.net/courses/ntp/napredni-python/, Datum pristupanja 31.07.2020.

[48] https://docs.python.org/3/library/io.html#, Datum pristupanja 03.10.2020.

[49] https://docs.python.org/3/library/copy.html, Datum pristupanja 04.10.2020.

[50] https://numpy.org, Datum pristupanja 03.10.2020.

[51] https://pandas.pydata.org, Datum pristupanja 03.10.2020.

[52] https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html, Datum pristupanja 03.10.2020.

[53] https://pandas-datareader.readthedocs.io/en/latest/remote_data.html, Datum pristupanja 03.10.2020.

[54] https://www.scipy.org, Datum pristupanja 03.10.2020.

[55] https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.norm.html, Datum pristupanja 03.10.2020.

[56] https://docs.python.org/3/library/multiprocessing.html, Datum pristupanja 03.10.2020.

[57] https://docs.python.org/3/library/multiprocessing.html#multiprocessing.pool.Pool, Datum pristupanja 04.10.2020.

[58] https://en.wikipedia.org/wiki/Data_parallelism#:~:text=Data%20parallelism%20is%20parallelization%20across,on%20each%20element%20in%20parallel., Datum pristupanja 04.10.2020.

[59] https://en.wikipedia.org/wiki/Google, Datum pristupanja 04.10.2020.

[60] https://en.wikipedia.org/wiki/Robert_Griesemer_(computer_programmer), Datum pristupanja 04.10.2020.

[61] https://en.wikipedia.org/wiki/Rob_Pike, Datum pristupanja 04.10.2020.

[62] https://en.wikipedia.org/wiki/Ken_Thompson, Datum pristupanja 04.10.2020.

[63] http://www.igordejanovic.net/courses/tech/GoLang/index.html, Datum pristupanja 01.08.2020.

[64] https://golang.org/pkg/math/rand/, Datum pristupanja 04.10.2020.

[65] https://golang.org/pkg/time/, Datum pristupanja 04.10.2020.

[66] https://golang.org/pkg/os/, Datum pristupanja 04.10.2020.

[67] https://golang.org/pkg/strings/, Datum pristupanja 04.10.2020.

[68] https://godoc.org/gonum.org/v1/gonum/stat, Datum pristupanja 04.10.2020.

[69] https://github.com/chobie/go-gaussian, Datum pristupanja 04.10.2020.

[70] https://godoc.org/github.com/markcheno/go-quote#NewQuoteFromYahoo, Datum pristupanja 04.10.2020.

[71] https://levelup.gitconnected.com/goroutines-and-channels-concurrent-programming-in-go-9f9f8495c34d, Datum pristupanja 04.10.2020.

[72] http://www.igordejanovic.net/courses/tech/Pharo/index.html, Datum pristupanja 03.08.2020.

[73] http://stephane.ducasse.free.fr, Datum pristupanja 05.10.2020.

[74] http://marcusdenker.de, Datum pristupanja 05.10.2020.

[75] https://en.wikipedia.org/wiki/Pharo, Datum pristupanja 05.10.2020.

[76] https://sr.wikipedia.org/wiki/Smalltalk, Datum pristupanja 05.10.2020.

[77] A. Bergel, D. Cassou, S. Ducasse and J. Laval, Deep into Pharo, Square Bracket Associates, 2013.

[78] S. Ducasse, D. Chloupis, N. Hess and D. Zagidulin, Pharo By Example 5, Lulu.com & Square Bracket Associates, 2018.

[79] A. Bergel, Roassal @ Pharo TechTalk, Pharo TechTalk, 2017.

УНИВЕРЗИТЕТ У НОВОМ САДУ ● **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**
21000 НОВИ САД, Трг Доситеја Обрадовића 6

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Редни број, **РБР**:	
Идентификациони број, **ИБР**:	
Тип документације, **ТД**:	Монографска публикација
Тип записа, **ТЗ**:	Текстуални штампани документ
Врста рада, **ВР**:	Дипломски рад
Аутор, **АУ**:	Душан Стевић
Ментор, **МН**:	др Игор Дејановић, ванредни професор
Наслов рада, **НР**:	Анализа серијске и паралелне имплементације алгоритама базираних на Монте Карло методи
Језик публикације, **ЈП**:	српски / латиница
Језик извода, **ЈИ**:	српски
Земља публиковања, **ЗП**:	Србија
Уже географско подручје, **УГП**:	Војводина
Година, **ГО**:	2020.
Издавач, **ИЗ**:	Ауторски репринт
Место и адреса, **МА**:	Факултет техничких наука, Нови Сад, Трг Доситеја Обрадовића 6
Физички опис рада, **ФО**: (поглавља/страна/ цитата/табела/слика/графика/прилога)	10/48/79/0/18/12/0
Научна област, **НО**:	Електротехника и рачунарство
Научна дисциплина, **НД**:	Примењене рачунарске науке и информатика
Предметна одредница/Кључне речи, **ПО**:	Напредне технике програмирања
УДК	
Чува се, **ЧУ**:	Библиотека Факултета техничких наука, Нови Сад, Трг Доситеја Обрадовића 6
Важна напомена, **ВН**:	
Извод, **ИЗ**:	У овом раду је детаљно извршена анализа серијске и паралелне имплементације алгоритама базираних на Монте Карло методи. Приказана је употреба на проблемима апроксимације броја пи, предвиђања кретања вредности финансијске активе и израчунавања апроксимиране вредности одређеног интеграла. Имплементација алгоритама реализована је у програмским језицима Python и Go, док је визуелизација података одрађена употребом програмског језика Pharo и библиотеке Roassal.
Датум прихватања теме, **ДП**:	
Датум одбране, **ДО**:	

Чланови комисије, **КО**:

Председник:	др Милан Видаковић, редовни проф.
Члан:	др Жељко Вуковић, доцент
Члан, ментор:	др Игор Дејановић, ванредни проф.

Потпис ментора

UNIVERSITY OF NOVI SAD ● **FACULTY OF TECHNICAL SCIENCES**
21000 NOVI SAD, Trg Dositeja Obradovića 6

KEY WORDS DOCUMENTATION

Accession number, **ANO**:	
Identification number, **INO**:	
Document type, **DT**:	Monographic publication
Type of record, **TR**:	Textual material
Contents code, **CC**:	Bachelor Thesis
Author, **AU**:	Dušan Stević
Mentor, **MN**:	Igor Dejanović, Associate Professor, Ph.D
Title, **TI**:	A Monte Carlo method based serial and parallel algorithm implementation analysis
Language of text, **LT**:	Serbian
Language of abstract, **LA**:	Serbian
Country of publication, **CP**:	Serbia
Locality of publication, **LP**:	Vojvodina
Publication year, **PY**:	2020.
Publisher, **PB**:	Author's reprint
Publication place, **PP**:	Faculty of Technical Sciences, Novi Sad, Trg Dositeja Obradovića 6
Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes)	10/48/79/0/18/12/0
Scientific field, **SF**:	Electrical and computer engineering
Scientific discipline, **SD**:	Applied computer science and informatics
Subject/Key words, **S/KW**:	Advanced Programming Techniques
UC	
Holding data, **HD**:	Library of the Faculty of Technical Sciences, Novi Sad, Trg Dositeja Obradovića 6
Note, **N**:	
Abstract, **AB**:	This paper describes a detailed analysis of the serial and parallel implementation of algorithms based on the Monte Carlo method. Several problems were analyzed including problems of approximation of the number pi, prediction of the movement of the value of financial assets, and calculation of the approximate value of a definite integral is presented. The implementation of the algorithms was realized in the programming languages Python and Go, while the visualization of the data was done using the programming language Pharo and the Roassal library.

Accepted by the Scientific Board on, **ASB**:

Defended on, **DE**:

Defended Board, **DB**:	President:	Milan Vidaković, Full Professor, Ph.D	
	Member:	Željko Vuković, Assistant Professor, Ph.D	Menthor's sign
	Member, Mentor:	Igor Dejanović, Associate Professor, Ph.D	