Sažetak

Koncept simulacije kao imitiranja jednog procesa drugim, pokazao se veoma pogodnim za primenu na računarima. Kod računarskih simulacija se proces koji proučavamo zamenjuje odgovarajućim računarskim programom. Zbog postojanja velikog broja različitih sistema koje želimo da simuliramo, razvijen je i veliki broj različitih simulacija. Uprkos postojanja značajnih razlika među njima, simulacije imaju i dosta zajedničkih karakteristika i potreba. Zato se i pojavila ideja za kreiranjem *opšteg simulatora*, odnosno programa koji implementira zajedničke osobine različitih simulacija. *Opšti simulator* pri tome mora imati mogućnost proširenja prilikom implementiranja konkretnih simulacija. Njegovom upotrebom se smanjuje redundantnost koda prilikom kreiranja pojedinih simulacija, jer sve one koriste gotove funkcije osnovne aplikacije. Prilikom upotrebe *opšteg simulatora*, od kreatora modula za konretnu simulacija se zahteva niži nivo programerskog znanja. Prilikom kreiranja *opšteg simulatora* i njegovih modula, susrećemo se sa nizom problema. To su pitanje komunikacije osnovne aplikacije sa modulima u kojima su konkretne simulacije, problemi automatizacije pokretanja simulacija, zatim imenovanja, smeštanja i prikazivanja rezultata, kao i generisanja grafičkog korisničkog interfejsa (Graphic User Interface, GUI) i sistema za kontrolu pristupa i učitavanja proširenja, odnosno plaginova. U ovom radu je ilustrovano jedno proširenje (plagin) za *opšti simulator*, kojim se realizuje simulator za fizički problem nalaženje solitona u fotoničnim rešetkama. Na ovom primeru se pokazuje jedan način ispitivanja vremena potrebnog za izračunavanje problema na računaru i ranog prepoznavanja slučajeva sa dugim periodom računanja (u kojima se često rezultat i ne može dobiti), kao i potreba za postojanjem metode za analizu rezultata simulacije kroz rutinu za prepoznavanje oblika solitona.

Softverska implementacija opšteg simulatora

Raka Jovanović

26. decembar 2007

Sadržaj

1 Uvod **3**

2 Simulacije **6**

2.1 Potreba za simulacijama i modeliranjem 6

2.2 Računarske simulacije 7

2.3 Generisanje slučajnih brojeva kao česta potreba računarskih simulacija 9

2.4 Provera tačnosti i testiranje 12

3 Ideja ***opšteg simulatora*** **13**

3.1 *Opšti simulator* 13

3.2 Pojedinačne simulacije kao proširenja *opšteg simulatora* (Plaginovi) 15

4 Implementacija ***opšteg simulatora*** **17**

4.1 Razvojno okruženje i dodatni alati korišćeni pri razvoju *opšteg simulatora* 17

4.2 Opis klasa za komunikaciju simulacije i *opšteg simulatora* . . . 18

4.2.1 Parametari 19

4.2.2 Klase koje obezbeđuju komunikaciju 21

4.3 Klasa koja omogućuje puštanje većeg broja simulacija i nalaženja simulacija zahtevanih osobina 24

4.3.1 Oblik parametra za puštanje većeg broja simulacija . . 24

4.3.2 Pokretanje svih simulacija iz intervala 26

4.3.3 Traženje simulacija sa odgovarajućim osobinama 27

4.3.4 Opis klasa za imenovanje, smeštanje i pregled rezultata simulacija 33

5 Implementacija proširenja za *opšti simulator* **41**
5.1 Sistem modula (plaginova) 41
5.2 Smeštanje i upotreba plaginova 42
5.3 Kreiranje plaginova 45
5.4 Kosi hitac kao prost primer implementacije proširenja *opšteg simulatora* 48

6 Grafički korisnički interfejs (Graphic User Interface, GUI) **52**
6.1 Dva moguća pristupa kreiranju GUI-a 52
6.2 Opis GUI-a, odnosno dijaloga iz kojih se sastoji 53

7 Primer proširenja za *opšti simulator* **60**
7.1 Fizički model 60
7.1.1 Opis modela 60
7.1.2 Simulacija propagacije solitonskih rešenja 61
7.2 Petviashvilijev metod za traženje solitonskih rešenja u fotoničnim rešetkama 62
7.2.1 Rešavanje problema simulacijom 64
7.2.2 Softverska implementacija 67
7.3 Rezultati 70
7.3.1 Potreba za postojanjem metode za analizu rezultata . . . 72
7.3.2 Metoda za prepoznavanje oblika solitona 72
7.3.3 Rezultati optimizacija 76

8 Zaključak **77**

Glava 1

Uvod

Čovek ima potrebu za ispitivanjem sistema i procesa iz sveta koji ga okružuje. Simulacija kao proces imitiranja jednog procesa ili sistema drugim, pokazuje se kao efikasan metod za izvođenja zaključaka. Sa razvojem računara dolazi i do sve većeg nivoa njihovog korišćenja, u obliku računarskih simulacija. Kod računarskih simulacija, početni sistem se imitira određenim programom. One su pokazale višestruke prednosti u odnosu na klasični eksperiment i različite fizičke simulacije, kao što su smanjenje troškova i potencijalnih opasnosti. Računarske simulacije su omogućile i pručavanje nekih sistema za koje nijedan drugi način nije bio moguć. Njihova primena se razvila i u nekim neočekivanim sferama, poput obrazovanja i zabave.

Usled potrebe za simulacijama raznih tipova problema, došlo je i do razvoja velikog broja različitih simulacija. Iako su u velikoj meri različite među sobom, one imaju niz zajedničkih osobina i potreba. Sa druge strane, kreatori simulacija često nemaju veliko programersko znanje, već im je stručnost vezana za sam problem koji se simulira. Odatle i potiče ideja za kreiranje *opšteg simulatora*, programa koji će implementirati veći broj zajedničkih osobina i opštih potreba simulacija, a ostaviti otvorenom mogućnost njegovog proširenja za simulacije konkretnih problema. Na ovaj način se postižu dva cilja. Prvi je sprečavanje redundandantnosti koda, jer posebne simulacije koriste gotove funkcije *opšteg simulatora*, a drugi je to da je kreiranje simulacije koncentrisano na sam proces a ne na razvijanje grafičkog okruženja i sličnih,

prvenstveno programerskih, problema.

Prilikom kreiranja *opšteg simulatora* susrećemo se sa nizom različitih problema:

- Prvi problem je komunikacija između *opšteg simulatora* i modula koji implementiraju samu simulaciju. Najpre je potrebno definisati kakvi se podaci prosleđuju između njih, odnosno definisati oblik ulaznih i izlaznih parametara, a zatim i sam mehanizam njihove razmene.

- Drugi problem je automatizacija pokretanja niza eksperimenata. Postoje dva osnovna cilja ovakve automatizacije. Jedan je da se upoznamo sa samim sistemom kojeg simuliramo, što je najlakše učiniti analizom većeg broja eksperimenata. Drugim rečima, želimo da pokrenemo simulaciju za sve vrednosti ulaznih parametra iz nekog intervala vrednosti. Drugi cilj je da potvrdimo neke zaključke u vezi sistema koji simuliramo, najčešće odnose između ulaznih i izlaznih parametara. Znači, potrebna nam je automatizacija nalaženja odgovarajućih vrednosti ulaznih parametara za neke vrednosti izlaznih parametra i obrnuto, pri čemu je ovaj drugi slučaj trivijalan.

- Treći problem je pitanje imenovanja, smeštanja i prikaza velike količine podataka dobijenih kao rezultat vršenja eksperimenata i to na način pogodan i za čoveka i za računar.

- Četvrti problem je pitanje grafičkog korisničkog interfejsa (GUI), koji treba da bude pogodan za niz različitih simulacija, odnosno njegovog dinamičkog generisanja.

- Peti problem je kreiranje sistema za korišćenje proširenja, odnosno modula, za *opšti simulator*. To su pitanja njihovog smeštanja, učitavanja i kontrole upotrebe memorije. Takođe je potrebno i olakšati kreiranje modula razvojem niza gotovih klasa, koje će pri tome moći da se koriste.

Kreiranje proširenja za *opšti simulator* je u ovom radu ilustrovano simulacijom fizičkih procesa u fotoničnim rešetkama i nalaženja odgovarajućih solitonskih rešenja. Problem je prikazan kao fizički model, koji se zamenjuje odgovarajućim matematičkim modelom. Prilikom ovih izračunavanja vrši se i definisanje divergencije u računaru i daje više načina njenog ranog prepoznavanja prilikom izvođenja numeričkog eksperimenata. Na ovaj način se postiže značajno poboljšanje performansi. Kroz ovaj primer se pokazuje i potreba za postojanjem metode za analizu rezultata simulacija, kao i jedna konkretna implementacija takve metode na prepoznavanju oblika solitona.

Glava 2

Simulacije

2.1 Potreba za simulacijama i modeliranjem

Cilj ispitivanja sistema je da se vidi kako se on ponaša prilikom različitih uticaja, odnosno promena parametara sistema. Prvi pristup ovakvom ispitivanju je konstrukcija sistema i vršenje niza kontrolisanih eksperimenata nad njim. Ovakav postupak je često skup i opasan, a u nekim slučajevima čak i nemoguć. Važno je primetiti da nas prilikom ispitivanja složenog sistema obično ne interesuju sve njegove karakteristike, već samo određene. Sa druge strane, razumevanje ovakvog sistema je lakše ako se on predstavi na jednostavniji način, odnosno ako se napravi model koji će sadržati sve karakteristike koje su od interesa. Ovakvi modeli mogu biti fizički, poput modela aviona koji će nam služiti za njegovo testiranje na otpor vazduha u vazdušnom tunelu i koji će imati samo spoljašnji oblik aviona, jer je on jedini bitan prilikom ispitivanja otpora. Za drugi primer modela uzećemo atom. On se sastoji od jezgra sastavljenog od protona i neutrona i elektrona koji kruže oko jezgra. Model atoma je moguće napraviti uz pomoć kugli i žica ali na ovaj način se njegove osobine neće dobro prikazati, jer atom sadrži mnogo složenije odnose svojih konstituenata od onih koji se mogu obuhvatiti ovako prostim mehaničkim sistemom. Odgovarajući model je niz jednačina koje će opisati jezgro i elektrone. Ovakav model se naziva *matematički model*. Korisno je napraviti razliku između dinamičkih i stacionarnih modela. Sta-

cionarni modeli su oni koji prikazuju stanje sistema prilikom mirovanja, a dinamički su oni kojima se modeliraju sistemi koji se menjaju kroz vreme.

Simulacije su u tesnoj vezi sa dinamičkim modelima. One predstavljaju imitiranje jednog procesa nekim drugim procesom. Rezultat simulacije je rešenje sistema jednačina dinamičkog modela. Mogu se simulirati razni tipovi sistema kao što su fizički, hemijski, bioloski, sociološki... Simulacije možemo podeliti na sledeće tipove:

- Fizičke simulacije su one kod kojih objekte realnog sistema zamenjujemo fizičkim objekima, koji su manji i jefiniji od njih, a imaju slično ponašanje.
- Interaktivne simulacije su one kod kojih je uključena i interakcija sa čovekom. Fizičke i računarske simulacije mogu biti interaktivne.
- Računarske simulacije su one kod kojih računarski program oponaša sistem.

Za simulacije je nađen veliki broj primena, od istraživačkih, preko raznih vidova treninga na simulatorima, provera bezbednosti sistema, traženja optimalnih rešenja za razne inženjerske probleme i t d.

2.2 Računarske simulacije

Ako je proces implementrian na računaru, onda se takva simulacija naziva računarskom simulacijom. Računarsk simulacije imaju više značajnih funkcija:

- Ispitivanje detalja dinamičkog sistema. Simulacija nam kao tehnika omogućava da bolje ispitamo eksperimente. Veliki broj sistema nije moguće ispitivati eksperimentom zbog vremenskih perioda u kojima se oni dešavaju. Na primer, ponašanje galaksija, gde su periodi predugački, ili nuklearnih reakcija, gde su prekratki. Prilikom kreiranja računarskih simulacija, uz postojanje odgovarajućeg modela, ovakav problem vremena ne postoji.

- Razvoj novih hipoteza, modela i teorija. Posle izvođenja velikog broja simulacija, dobijamo veliki broj rezultata koji nam ukazuju na različite moguće teorije koje stoje iza sistema koji simuliramo.

- Simulacija kao zamena za eksperiment, kada ona zapravo predstavlja numerički eksperiment. Simulacija ima dve osnovne prednosti u odnosu na uobičajeni eksperiment. Prva je mogućnost izvođenja simulacija za koje je eksperiment nemoguće konstruisati, poput formiranja galaksija ili promena nekih prirodnih konstanti, koje su inače nepromenjive i posmatranja ponašanja sistema u takvim uslovima, na primer promena jačine gravitacije. Druga velika prednost je mogućnost lake promene vrednosti različitih parametara sistema, koja prilikom korišćenja klasičnog eksperimenta može zahtevati skupu dodatnu aparaturu. Važno je da simulacija ne služi samo kao zamena već i kao alat za kreiranja eksperimenta u smislu inspiracije, upoznavanja sa sistemom i sužavanja opsega parametara za koje će se vršiti eksperiment i kao uporedni alat prilikom analize rezultata eksperimenta.

- Simulacije se mogu koristiti i kao pedagoški alat, jer u mnogo čemu olakšavaju razumevanje nekog procesa. Za simulacije je lako konstruisati grafičke prezentacije. Ovakvom prezentacijom i "igranjemša parametrima sistema studenti dobijaju viši nivo razumevanja za ponašanje sistema. Ovakav postupak je moguć i klasičnim eksperimentom, ali je on često daleko skuplji.

Računarske simulacije su pokušaj modeliranja realnih sistema na računaru, tako da možemo analizirati kako se taj sistem ponaša. Tradicionalno, sistem se formalno modelira matematičkim modelom koji pokušavamo analitički da rešimo za razne vrednosti ulaznih parametara i na taj način predvidimo njegovo ponašanje. Računarske simulacije se najčešće zasnivaju na ovim modelima, ali su pogodne i za sisteme koje nije moguće na ovaj način reprezentovati.

Računarske simulacije se razvijaju naporedo sa opštim razvojem računara. Svoju moć su prikazale prvi put u vreme Drugog svetskog rata prilikom

simulacije eksplozije atomske bombe (Projekat Menheten). Ovim projektom su se pokazale velike prednosti ovakvih simulacija, kao što je mogućnost ponavljanja eksperimenta bez velikih troškova i uklanjanje opasnosti prilikom njih.

Računarske simulacije se mogu klasifikovati na osnovu nekoliko različitih kriterijuma.

- Stohastičke i determinističke. Determinističke simulacije su one kod kojih se za iste ulazne vrednosti parametara uvek dobija isto ponašanje simulacije. Stohalističke simulacije su one kod kojih se koriste pseudo slučajni brojevi za izazivanje nekih događaja u simulaciji, što za posledicu ima da ne mora biti isto ponašanje sistema za iste ulazne parametre.

- Ravnotežnog stanja i dinamičke. Simulacije ravnotežnog stanja traže, za određeno ulazno stanje sistema, odgovarajuće ravnotežno stanje, pa se one često koriste za modeliranje fizičkih sistema. Dinamičke simulacije modeliraju promene koje se dešavaju u sistemu na osnovu nekih impulsa koji se pojavljuju tokom vremena.

- Neprekidne i diskretne. Neprekidne su one simulacije kod kojih je sistem predstavljen nizom parcijalnih jednačina i simuliranje se svodi na njihovo rešavanje. Diskretne su one kod kojih je sistem predstavljen nizom entiteta i načinom na koji interaguju.

- Lokalne i distribuirane. Lokalne simulacije su simulacije koje se izvršavaju na jednom računaru a disribuirane su one kod kojih se simulacija izvršava na većem broju umreženih računara.

2.3 Generisanje slučajnih brojeva kao česta potreba računarskih simulacija

Postoji niz simulacija koje koriste slučajne brojeve. To su one kod kojih postoji potreba za nekim slučajnim događajima poput ruleta. Iz tog razloga i metoda za rešavanje ovog tipa simulacije dobija ime Monte-Karlo.

Postoje dva različita načina primene ove metode. Prvi, koji se primenjuje za stohastičke probleme (na primer problem slučajnog hoda), gde u samom sistemu koji simuliramo postoje neki slučajni događaji, kod kojih nas interesuju neke statističke vrednosti skupa rezultata. Drugi način primene je da neki deterministički problem zamenjujemo stohastičkim, odnosno takvim kod koga se prosečno rešenje za veliki niz realizacija poklapa sa rešenjem početnog determinističkog problema.

Generisanje slučajnih brojeva

Poseban problem prilikom primene ovakvih metoda je generisanje slučajnih brojeva. Najjednostavnije rešenje za ovakav problem bi bilo, na primer, uzimanje neke funkcije trenutnog računarskog vremena. Ovakvi brojevi jesu slučajni (osim za neke jako kratke procese), međutim oni su neupotrebljivi za razvoj ovakvih simulacija. Najveći problem je nemogućnost reprodukcije istog niza slučajnih brojeva za proveru rada programa. Iz ovih razloga se koriste pseudo slučajni brojevi, odnosno niz brojeva koji su generisani nekim numeričkim algoritmom iz neke početne vrednosti, odnosno 'semena'. Ovakav niz slučajnih brojeva treba da ima nekoliko važnih osobina: da ima jako dugačak period pre nego što počne da se ponavlja, da se ravnomerano pojavljuju vrednosti iz traženog intervala, da postoji što manje korelacija. Loš izbog generatora ovakvih brojeva prouzrokuje netačne rezultate korišćenja Monte-Karlo metode.

Generatori slučajnih brojeva generišu celobrojne vrednosti iz intervala (0, N_{max}), a zatim se one transformišu na odgovarajući interval. Postoji nekoliko algoritama za generisanje slučajnih brojeva:

- **Kongruentna metoda** je jednostavna i popularna metoda za generisanje slučajnih brojeva. Kod nje se bira fiksirani množilac c i neki koren X_0, a ostale vrednosti iz niza pseudo slučajnih brojeva se dobija sledećom formulom:

$$X_{n+1} = c \times X_n (MOD) N_{max} \tag{2.1}$$

Bitno je da množilac c ima dobre osobine da bi se dobile duže niske slučajnih brojeva. Pogodno je i da je seme neparno. Pokazalo se da postoji značajna korelacija susednih trojki u nizu generisanih brojeva, ali se i dalje ovakav generator često koristi zbog njegove jednostavnosti i zadovoljavajuće šlučajnostiža veliki broj problema. Često se koristi i sledeći generator:

$$X_{n+1} = 16807 \times X_n (MOD) 2^{31} - 1 \quad (2.2)$$

- **Miksovane kongruentne metode** se zasnivaju na kombinovanju nekoliko kongruentnih generatora. Kao primer moěmo navesti: jedan generator koji generiše tabelu slučajnih brojeva, a drugi iz nje izvlači brojeve po slučajno generisanom indeksu. Za najbolje rezultate dva generatora treba da imaju različita semena i multiplikatore.

- **Šift registar algoritam** je brza metoda koja ima manje problema sa korelacijom od kongurentnih metoda. Prvo je potrebno generisati početnu tabelu slučajnih brojeva, a novi slučajni brojevi se generišu već postojećim, pomoću sledeće formule:

  $$X_n = X_{n-p} \cdot XOR \cdot X_{n-q} \quad (2.3)$$

  p i q moraju biti izabrani na odgovarajući način, odnosno par (p, q) treba da zadovoljava uslov

  $$X^p + X^q + 1 = ProstTrinom \quad (2.4)$$

  Često korišćen par je (250, 103). Slučajni brojevi većeg kvaliteta se dobijaju izborom većih vrednosti za p i q. Kvalitet slučajnih brojeva u velikoj meri zavisi od početno generisanih slučajnih brojeva.

2.4 Provera tačnosti i testiranje

Ideja prilikom kreiranja simulacije je da ona oponaša ponašanje nekog sistema iz prirode. Izlazne vrednosti koje se dobijaju izvršavanjem simulacije bi trebalo da se poklapaju sa onim koje odgovaraju simuliranom procesu, do neke razumne granice tačnosti. Pitanje je kako možemo biti sigurni u to ako odgovarajući eksperimentalni podaci ne postoje. Simulacija je upravo i pravljena da bi mogla da predvidi ponašanje sistema bez vršenja odgovarajućeg eksperimenta. Iz ovog razloga je tesitiranje i provera tačnosti od velike važnosti.

Treba razumeti da postoji nekoliko uzroka greške kod simulacija:

- Programske greške. To su one greške koje uzrokuju da se program ne ponaša na željeni način. Prve koje srećemo su sintaksne i one koje dovode do prekida rada programa, poput nelegalnog pristupa memoriji. One se najčešće lako ispravljaju. Postoje greške koje izazivaju čudno ponašanje sistama koje se takođe lako ispravljaju. Problem predstavljaju one koje prouzrokuju naizgled ispravno ponašanje, koje međutim nije tačno. Jedini način za nalažetje ovakvih grešaka je intenzivno testiranje i testiranje specijanih slučajeva za koje znamo rezultat.

- Numeričke greške. To su one koje nastaju usled numeričkog računa. Postoje dva tipa greške, one koje nastaju iz ograničenosti računarskog prostora, odnosno zaokruživanja usled njega. Drugi tip je onaj koji nastaje iz zamene analitičkih izraza numeričkim. One se nalaze tako što se rezultati izvršavanja ograničenih delova programa porede sa poznatim analitičkim rezultatima za izraze koje pretstavlja taj deo programa.

- Modelarske greške. Su one koje su uzrokovane nepotpunim modeliranjem sistema, odnosno odsustvom nekih osobina koje postoje u fizičkom sistemu. One se nalaze intenzivnim poređenjen rezultata dobijenih simulacijom sa rezultatima fizičkih eksperimenata.

Jako je važno ukloniti sve ove greške, da bi simulacija davala rezultate koji se poklapaju sa rezultatima eksperimenta.

Glava 3

Ideja *opšteg simulatora*

3.1 *Opšti simulator*

Postoji veliki broj sistema koji se mogu simulirati i to na niz različitih načina. Međutim, iako u velikoj meri različite, simulacije imaju niz zajedničkih osobina. Sa druge strane, prilikom korišćenja simulacija u istraživačke svrhe, način njihove upotrebe se ne razlikuje mnogo, bez obzira na to o kakvom se sistemu radi ili koja je metoda korišćena za njeno kreiranje. Iz ovih razloga se pojavljuje ideja kreiranja okruženja, odnosno programa koji će moći da pokreće niz različitih simulacija. Upravo to je namena *opšteg simulatora. Opšti simulator* zapravo predstavlja niz modula koji implementiraju standardne potrebe simulacija.

Bez obzira da li se radi o simuliranju kosog hitca, kretanja automobila u saobraćaju, bacanju novčića..., za svaku od ovih simulacija je karakteristično da postoje neki ulazni parametri od kojih će ona zavisiti. Tako će kod kosog hitca biti važan ugao hitca i početna brzina, kod simulacije saobraćaja broj automobila, i mapa ulica, a kod bacanja novčića broj bacanja. Cilj vršenja eksperimenta (pokretanja simulacije za određene vrednosti ulaznih parametara), je ispitivanje sistema odnosno njenogovog ponašanja koje će se videti iz vrednosti nekih izlaznih parametara. Na primer kod kosog hitca nas interesuje koliko daleko je otišao projektil, kod ispitivanja saobraćaja nas interesuje

gde su nastala zagušenja a kod bacanja novčića nas može interesovati broj pojavljivanja glave. Jedna od funkcija *opšteg simulatora* je da omogući unos ulaznih parametara za simulaciju i da od nje preuzima vrednosti izlaznih. Za unos vrednosti ulaznih parametara je neophodno kreirati odgovarajući grafički korisnički interfejs (GUI), koji će se generisati u zavisnosti od potreba simulacije. Isto tako je potrebno implementirati i pregled izlaznih podataka eksperimenta. Izlazni podaci mogu biti neki niz skalarnih vrednosti, kao pređeni put prilikom kosog hitca i maksimalna postignuta visina, koji će biti prikazani kao brojevi ili druge oznake. U drugim slučajevima ovakav prikaz nam nije od koristi, kao u slučaju prikaza nastanka zagušenja u saobraćaju, gde nam je potreban prikaz gusitina saobraćaja na različitim delovima mape. *Opšti simulator* zbog ovakvih potreba raznih simulacija implementira i razne vidove grafičkog prikaza rezultata.

Prilikom upotrebe različitih simulacija, najčešće nam nije glavni cilj ispitivanje pojedinačnih eksperimenata, već dolaženja do nekih opštih zaključaka vezanih za odnose ulaznih i izlaznih parametara. Na primer, formula zavisnosti ugla i pocetne brzine projektila i dužine leta kod kosog hitca, ili zavisnost razdaljine raskrsinica i ugla među ulicama i nastanka gužvi. Olakšavanje dolaska do ovih zaključaka je još jedna funkcija *opšteg simulatora*. Vršenje ove funkcija se sastoji iz sledećih delova:

- Omogućavanje automatizacije pokretanja većeg broja eksperimenata, jer je najlakše doći do zaključaka analizom većeg broja slučajeva.
- Olakšavanje analize rezultata niza eksperimenata. Na primer, za uviđanje Gausove raspodele broja pojavljivanja glave kod niza bacanja novčića, neophodan je grafik zavisnosti između broja pojavljivanja glave i broja bacanja. Grafici zavisnosti parametara u velikoj meri olakšavaju razumevanja neke pojave, pa zato *opšti simulator* omogućuje pojednostavljenje njihovog kreiranje.
- Potvrda zakljucaka i olakšavanje njihove ispravke u slučaju greške. Kada smo došli do nekih zaključaka u vezi ispitivanog sistema, želimo

da na brz i jednostavan način možemo i da ih potvrdimo. Provera da li nam je zaključak dobar je jednostavna, pošto smo došli do zavisnosti ulaznih i izlaznih parametara. Potrebno je samo da pokrenemo niz eksperimenata za ulazne parametre i da utvrdimoimo da li su dobijeni odgovarajući izlazni parametri. U slučaju da to nije tačno, odnosno da naš zaključak nije ispravan, velika nam je pomoć da znamo za koje ulazne parametre bi se dobili očekivani izlazni. Ovo je još jedna od funkcija *opšteg simulatora*.

Osnovni deo *opšteg simulatora* ne implementira ni jedanu konkretnu simulaciju, već se one priključuju u obliku posebnih dodataka, odnosno proširenja.

3.2 Pojedinačne simulacije kao proširenja *opšteg simulatora* (Plaginovi)

Prilikom kreiranja *opšteg simulatora* jedna od osnovnih ideja je da opšte funkcije, odnosno one koje se mogu primeniti nezavisno od konkretne simulacije, budu što je moguće nezavisnije od samih simulacija. Iz ovog razloga je program zasnovan na konceptu osnovne aplikacije sa njenim priključcima. Opšte funkcije su implemetirane u osnovnoj aplikaciji, a pojedinačne simulacije u odgovarajućim priključcima, bez kojih osnovna aplikacija nema nikakvu funkcionalnost.

Priključak, odnosno modul, će implementirati sve posebne karakteristike konkretne simulacije i metode kojom je ona kreirana. To će biti sâmo računanje simulacije. Kao, na primer, računanje kretanja projektila kod kosog hitca, kontrola vozila u zavisnosti od stanja na mapi kod simulacije saobraćaja, generisanje slučajnog broja i na osnovu njega odlučivanje da li je dobijena glava ili pismo prilikom simulacije bacanja nočića. Ovakav priključak, osim samog računanja simulacije, zadužen

je i za kreiranje i čuvanje svih dobijenih rezultata (na primer pozicije svih vozila na mapi). Još jedna važna funkcija, koja treba da omogući priključak, je mogućnost praćenja stanja izračunavanja (na primer redni broj bacanja novčića do kog se stiglo).

Međutim, ovakvi priključi nisu u potpunosti nezavisni od *opšteg simulatora*, već moraju da imaju mogućnost komunikacije sa njim, kao sto je razmena vrednosti ulaznih i izlaznih parametara. Ovo je postignuto postojanjem niza klasa čije korišćenje je neophodno u modulu. Radi olakšavanja programiranja modula za pojedine simulacija, pored ovih klasa kreirane su i pomoćne klase koje rešavaju neke od standardnih problema prilikom simulacija.

Na ovaj način se dobija niz prednosti prilikom kreiranja priključaka:

- Smanjena potreba za redundantnim kodom, jer je veliki deo zajedničkih funkcija sadržan u osnovnoj aplikaciji.
- Potreban je niži nivo znanja programiranja za kreatore proširenja.

Glava 4

Implementacija *opšteg simulatora*

4.1 Razvojno okruženje i dodatni alati korišćeni pri razvoju *opšteg simulatora*

Opšti simulator je pisan u programskom jeziku C++. Sam koncept *opšteg simulatora* zahteva visok nivo inkapsulacije (nezavisnosti od ostalih) komponenti i potrebe za izvlačenjem zajedničkih osobina za različite simulacije. Ovo su problemi koji nam neposredno nameću potrebu za objektno orjentisanim razvojem aplikacije, a samim tim i za takvim jezikom. Iz razloga što *opšti simulator* treba sukcesivno da izvršava veliki broj simulacija, javlja se potreba za brzim izvršnim kodom, kao i za velikom kontrolom nad upotrebom memorije. Ovo su upravo osobine koje C++ čine logičnim izborom. *Opšti simulator* je razvijan u *Microsoft Visual Studio .Net 2003*. Postoji više razloga za ovaj izbor:

- Standardni alati za razvoj C++ aplikacija.
- Olakšan razvoj GUI-a jer postoji dobro grafičko okruženje za dizajniranje potrebnih dialoga uz pomoć MFC (Microsoft Fundamental Classes) klasa.
- Visok nivo kontrole kreiranja izvršnog koda za aplikaciju. Ovo je omogućeno jednostavnim podešavanjem velikog broja direktiva prevodioca.

- Dobro organizovan *Help*, bez koga je praktično nemoguće razvijati veće aplikacije.
- Projekti mogu sadržati i potprojekte. Na ovaj način je rešen problem naporednog razvijanja dll-ova (dynamic link libraries, dinamički povezane biblioteke) i aplikacije koja će ih koristiti.
- Postojanje izuzetno kvalitetnog debagera.

Rezultat rada *opšteg simulatora* je često velika količina podataka, koju nije lako pregledati bez dodatne vizuelizacije. Za deo ovih podataka vizuelizacija je urađena od strane samog simulatora. Međutim, često se javlja potreba za većim brojem načina vizuelizovanja podataka. Kako *opštem simulatoru* nije glavna funkcija grafički prikaz njegovih rezultata, on kreira odgovarajuć e datoteke (fajlove) rezultata, koji se mogu gledati iz drugih programa. Programi koji su korišćeni u ovu svrhu su *PsiPlot 7.0*, *SigmaPlot 8.0*, *OriginPro 7.0*.

4.2 Opis klasa za komunikaciju simulacije i *opšteg simulatora*

Prvi problem na koji nailazimo prilikom implemetacije *opšteg simulatora* je problem komunikacije osnovne aplikacije i proširenja koje implementira jednu simulaciju. Postoji nekoliko delova komunikacije između *opšteg simulatora* i modula za simulaciju.

1. *Opšti simulator* treba da dobije informacije od modula za simulaciju o tome koje parametre koristi i kakve su njihove potencijalne vrednosti.
2. *Opšti simulator* treba da prosledi modulu za simulaciju vrednosti potrebnih parametara.
3. Modul za simulaciju, nakon nađenog ravnotežnog stanja (ili potvrde nemogućnosti nalaženja takvog stanja), treba da vrati simulatoru vrednosti od značaja za tu simulaciju.

4.2.1 Parametari

Parametri koji se koriste u simulacijama mogu biti raznovrsni. To mogu biti standardni tipovi podataka (kao što su celobrojni, logički i u pokretnom zarezu), a mogu biti i vrednosti iz određenog skupa (na primer imena modela aviona koji će biti testiran na otpor vazduha). Iz ovih razloga korisno je definisati zasebnu klasu koja će rešavati probleme za upotrebu parametara.

```
class RPodatak
        {
         CString          Name;
         CString          Value;
         int              Type;
         RTranslator     *mTranslator;

      public:
                          RPodatak (void);
                         ~RPodatak (void);
         CString          GetName (){return Name;}
         double           GetDoubleValue ();
         int              GetIntValue ();
         CString          GetStringValue ();
         void             SetValue (CString nValue);
         void             SetValue (double nValue);
         void             SetValue (int nValue);
         void             SetName (CString nName);
      };
```

Parametar je sa jedne strane određen svojim tipom, a sa druge imenom i vrednošću, koje su obe predstavljene niskom znakova. Vrednost parametra je najpogodnije predstaviti na ovaj način, jer se one mogu lako konvertovati (pisanjem odgovarajućih metoda) u bilo koji drugi tip podataka, pa je ovakav oblik pogodan za pregled vrednosti od strane čoveka.

Kao što je već rečeno, simulacije najčešće imaju veći broj različitih parametara od kojih zavise. Sa druge strane, prilikom pokretanja simulacije, često se dešava da se pokaže da bi bilo korisno menjati na pogodan način izvesne vrednosti za koje je bilo očekivano da predstavljaju konstante sistema. Iz ovih razloga je korisno sve parametre simulacije grupisati u jednu strukturu radi lakšeg manipulisanja njima i olakšane mogućnosti izmena. Klasa koja rešava ovaj problem je

```
class RPodatakNiz {
  protected:
       RPodatak    mPodaci[RPODACINIZ_MAX_INDEX];
       int         MaxIndex;
  public:
                   RPodatakNiz (void);
                  ~RPodatakNiz (void);
       void        AddValue (CString nName,
                                [int, double, string]value);
       void        SetValue (CString nName,
                            [int, double, string] nValue);
       CString     GetValue (int Index);
       CString     GetName (int Index);
       bool        GetValue (CString nName, CString  &gValue);
       int         GetMaxIndex (){return MaxIndex;}
       bool        LoadFromFile (CString FileName);
       bool        SaveToFile (CString FileName);
       void        Reset (){MaxIndex = 0;}
       CString     GetParamList ();
 };
```

RPodatakNiz je niz RPodataka, koji je urađen na način koji omogućuje pristup pojedinim članovima na osnovu imena parametra. Ovde treba napomenuti važnost nekoliko metoda. To su *GetParamList*, koja nam daje nisku koja sadrži sva imena parametara koji se nalaze u nizu. Zatim, *LoadFromFile*,

SaveToFile su metode koje služe za učitavanje, odnosno upisivanje, svih parametara u datoteku. Metoda *GetParamList* je bitna za prenos spiska parametara simulacije u *opšti simulator*. *LoadFromFile*, *SaveToFile* su korisne za komunikaciju između ljudi koji koriste simulator, jer na ovaj način je omogućeno lako ponavljanje eksperimenata na drugim računarima.

4.2.2 Klase koje obezbeđuju komunikaciju

Da bi mogla da se standardizuje komunikacija sa simulacijom, moraju se izvući sve zajedničke osobine simulacija, odnosno tipovi radnji koje one vrše. Prvo treba prepoznati nekoliko osnovnih tipova parametara koje koristi simulacija, a to su:

- Parametri strukture. To su parametri koji predstavljaju neke nepromenjive osobine simulacije.
- Parametri simulacije. To su parametri na čijim intervalima će se vršiti masovna puštanja.
- Parametri rezultata. To su parametri koji su vezani za izlazno, odnosno ravnotežno stanje.

Klasa koja opisuje takve osobine simulacija je

```
class SimulatorAbstract {
protected:
    PathGenerator       *mPathGenerator;
public:
     RPodatakNiz         mParamStruct;
     RPodatakNiz         mParams;
     RPodatakNiz         mResult;

     SimulatorAbstract (void);
     ~SimulatorAbstract (void);
```

```
    virtual void       Calculate ();


    void   AddParam (double Value, string Name);
    void   AddParam (int Value, string Name);
    void   AddParam (string Value, string Name);

    void   AddParamStruct (double Value, string Name);
    void   AddParamStruct (int Value, string Name);
    void   AddParamStruct (string Value, string Name);


    virtual void       NapuniGenerator ();
    virtual void       CreateSimulationFile ();
    virtual bool       Converge ();
    virtual CString    GetParamList ();
    virtual CString    GetCalculatedValueList ();
    virtual CString    GetGrafikFileRow (CString NeededInfo[50],
                                         int NumberOfNeeded);
    virtual CString    GetParamValueString (CString ParamName);
    virtual bool       LoadFromFile ();
            void       Konvertuj ();

   };
```

Opšti postupak prilikom pokretanja simulacije je sledeći: najpre se vrši resetovanje parametara simulacije, zatim postavljanje vrednosti potrebnih parametara ručno ili iz neke datoteke (fajla) i na kraju izračunavanje same simulacije. Pretpostavimo da je A tipa *SimulatorAbstract*

```
A.Reset ();
A.LoadFromFile ("test.data");
A.Calculate ();
```

```
ili

A.Reset ();
A.AddParamInt ("2", "F");
A.AddParamInt ("2", "F");
...
A.AddParamDouble ("2.12", "M");
A.Calculate ();
```

Moduli za simulaciju treba da su izolovani od GUI-a *opšteg simulatora*, pa treba konstruisati klasu koja će biti most za ovu komunikaciju. Ista klasa će služiti i za pokretanje većeg broja simulacija, ali ovaj postupak će biti bliže objašnjen u daljim poglavljima.

```
class SimulationMultiPlayer {

        SimulatorPlayerParam    Params[MAXNUMBEROFPARAMS];
        int                     MaxParamIndex;
        SimulatorAbstract      *mSimulator;
        public:

                 SimulationMultiPlayer (void);
                ~SimulationMultiPlayer (void);

        bool     SetPlayerParam (double nParam, CString nName);
        bool     SetPlayerParams (double nOd,double nDo,
                                  double nKorak, CString nName);
        void     SetupSimulation ();
        void     SetupSimulationFromFile (CString FileName);
        void     Clear ();
        void     SetSimulator (SimulatorAbstract *nSimulator);
```

```
    void    PlayAllSimulations (int  ActiveIndexParam = 0);
    void    AddParam (SimulatorPlayerParam nParam);
    bool    FindValue (int MaxDepth,
                  CString ParamName,double ParamValue,
                  double SearchValueLower ,double SearchValueUpper,
                  CString SearchParamName, double SearchPrecision ,
                  int growDirection,int convergeSide);

    void    CreateGraphFile (CString NeededParam[100],
                            int NumberOfNeeded,
                            FILE *F,int  ActiveIndexParam =0);

};
```

Klasa *SimulationMultiPlayer* će takođe morati da ima vrednosti parametara simulacije, koje će proslediti pojedinačnim simulatorima. Postupak za pokretanje simulacije će biti sledeci. Prvo se bira simulator koji želimo, zatim se učitavaju parametri, pa se prosleđuju odgovarajućem modulu za simulaciju, a na kraju se pokreće simulator. Ovakva klasa naizgled deluje kao dupliranje posla oko pokretanja simulacije, ali se na ovaj način povećava inkapsulacija modula za simulaciju.

4.3 Klasa koja omogućuje puštanje većeg broja simulacija i nalaženja simulacija zahtevanih osobina

4.3.1 Oblik parametra za puštanje većeg broja simulacija

Kada želimo da pokrenemo veći broj simulacija, to će naj češće biti simulacije za vrednosti parametara iz nekog intervala. Na datom intervalu će,

u zavisnosti od osobina koje se ispituju, biti potrebno znati i koliko gusto treba birati vrednosti parametara. Dakle, za svaki parametar simulacije biće potrebno izvršiti ove radnje, pa je korisno definisati klasu sa tom funkcijom.

```
class SimulatorPlayerParam {
   protected:
    double          From;
    double          To;
    double          Step;
    double          CurrentValue;
    CString         Name;
   public:

                 SimulatorPlayerParam (void);
                ~SimulatorPlayerParam (void);

     void     SetParamSimple (double nParam);
     void     SetParams (double nOd, double nDo, double nKorak);
     void     SetParams (double nOd, double nDo,
                        double nKorak,double nCurrentValue);
     void     SetName (CString   nName){Name = nName;}
     bool     NextValue (double *Value = NULL);
     bool     PreValue (double *Value = NULL);
     void     Reset ();

     SimulatorPlayerParam   operator = (SimulatorPlayerParam n);

     };
```

Interval se definiše svojim početkom $From$, krajem To i gustinom, odnosno razmakom između čvorova na intervalu *Step*. Parametar u svakoj pojedinačnoj simulaciji ima jedinstvenu vrednost, pa je i nju potrebno zasebno izdvojiti $CurrentValue$. Pošto je potrebno preći ceo interval, mora posto-

jati način prelaska sa jednog čvora intervala (aktivne vrednosti parametra) na sledeći u nizu i dobiti obaveštenje o tome kada je ceo interval pređen. Zbog toga definišemo dve metode, $NextValue$ i $PreValue$, u zavisnosti od toga u kom pravcu želimo da se krećemo.

4.3.2 Pokretanje svih simulacija iz intervala

Pretpostavimo da naša simulacija zavisi od niza parametara, na primer $A^1, A^2, ..A^n$. Simuluaciju tipa stacionarno stanje (steady state) možemo sada posmatrati kao preslikavanje sa skupa $A^1 \times A^2 \times .. \times A^n$ na skup Rez. Rez predstavlja skup rešenja. Interval za koji puštamo simulaciju će biti

$$(a_0^1, a_k^1) \subseteq A^1, (a_0^2, a_m^2) \subseteq A^2, .., (a_0^n, a_j^n) \subseteq A^n \tag{4.1}$$

$$I = (a_0^2, a_k^1) \times (a_0^2, a_m^2) \times .. \times (a_0^n, a_j^n) \tag{4.2}$$

Ovo je neprekidni opis intervala, a pošto računari rade u diskretnom prostoru, potrebno je definisati odgovarajuće skupove čvorova. Pretpostavimo da su $s_{A^1}, s_{A^2}, .., s_{A^3}$ koraci na intervalima (a_0^1, a_k^1), (a_0^2, a_m^2) ,.., (a_0^n, a_j^n). Sada skupove čvorova možemo predstaviti kao

$$a_i^1 = a_0^1 + i * s_{A^1}, A_d^1 = \{a_i^1 | i = 0..k\} \tag{4.3}$$

$$a_i^2 = a_0^2 + i * s_{A^2}, A_d^2 = \{a_i^2 | i = 0..m\} \tag{4.4}$$

..

$$a_i^3 = a_0^3 + i * s_{A^n}, A_d^n = \{a_i^3 | i = 0..n\} \tag{4.5}$$

$$A_d^1 \times A_d^2 \times .. \times A_d^n \tag{4.6}$$

Znači, da bismo dobili ceo skup rezultata, za sve čvorove na intervalu, potrebno je izračunati simulacuju za sve moguće kombinacije parametara. Programski se ovo najlakše postiže tako što za jedan izabrani parametar prolazimo kroz svaku njegovu moguću vrednost i računamo simulaciju za sve kombinacije ostalih parametara. Rekurzivna metoda koja vrši ova izračunavanja je

```
void SimulationMultiPlayer::PlayAllSimulations (int ActiveIndexParam ) {
     if (ActiveIndexParam >= MaxParamIndex){
        if (ActiveIndexParam>0)
        {
            SetupSimulation ();
            mSimulator->Calculate ();
        }

            return;
    }
    Params[ActiveIndexParam].Reset ();
    do{
        PlayAllSimulations (ActiveIndexParam +1);
    }while (Params[ActiveIndexParam].NextValue ());
}
```

Ova metoda ima jedan parametar. To je redni broj parametra za koji prolazimo kroz sve vrednosti čvorova na intervalu. Prvo se proverava da li postoji parametar sa tim rednim brojem, ili smo već izabrali kombinaciju parametara. Ako je izabrana kombinacija, simulatoru prosledimo vrednosti svih parametara i izračunamo je. U suprotnom, idemo kroz jednu pojednu vrednost aktivnog parametra i za tu fiksiranu vrednost pozivamo metod za sledeći parametar po redu PlayAllSimulations (ActiveIndexParam + 1).

4.3.3 Traženje simulacija sa odgovarajućim osobinama

Kao što smo rekli, kod simulacija stacionarnog stanja, za odgovarajuće početno stanje tražimo odgovarajuće ravnotežno, odnosno izlazno stanje. Početno stanje je zapravo neka kombinacija vrednosti ulaznih parametara. Analogno tome, izlazno stanje je neka kombinacija vrednosti izlaznih parametara. Nekada je problem koji želimo da rešimo simulacijom sledeci: koji su ulazni parametri potrebni da bi se dobili odgovarajući izlazni. Primer

bi bio zadatak: koji je najveći teret koji može da izdrži most. Postupak nalaženja odgovarajućih ulaznih vrednosti parametara u opštem slučaju nije lako rešiv, pa problem svodimo na jednostavniji. Tražimo vrednost samo jednog ulaznog parametra, dok su ostali fiksirani za dobijanje odgovarajuće vrednosti jednog izlaznog parametra. Čak i ovakav prostiji problem nije jednostavno u potpunosti automatizovavi, već je potrebna i značajna pomoć korisnika.

Kao sto smo rekli, rezultat simulacije je elemenat skupa rešenja Rez, a $Rez = Rez^1 \times Rez^2 \times .. \times Rez^r$. Pretpostavimo da $(a^1, a^2, ..a^n) \in A$ vrednost ulaznih parametra a $(r^1, r^2, .., r^r) \in Rez^j$ vrednost izlaznih parametara za odgovarajuće ravnotežno stanje. Postoji još jedna mogućnost za rešenje koja nije elemenat skupa Rez. To je slučaj kada nije moguće naći ravnotežno stanje koje odgovara datom ulazu. Obeležimo taj slučaj sa $Div = (div^1, div^2, .., div^r)$ i uvedimo $Rez_p = Rez \cup \{Div\}$. Obeležimo preslikavanje određeno simulacijom sa S, tada je

$$S : A \longrightarrow Rez_p \tag{4.7}$$

Uvedimo funkciju S^j, gde je $j \in \{1..r\}$, takvu da je

$$a \in A \tag{4.8}$$

$$S(a) = (r^1, r^2, ..r^r) \tag{4.9}$$

$$S^j : A \longrightarrow Rez_j \tag{4.10}$$

$$S^j(a) = r^j \tag{4.11}$$

Znači, S^j predstavlja projekciju funkcije S na Rez^j. Napomenućemo još da u slučaju nenalavženja ravnotežnog stanja imamo $r^j = div^j$. Problem nalaženja svih kombinacija ulaznih parametra takvih da je $S^j(a) = S^j(a_1, a_2, .., a_r) = r^j$ nije lako rešiv, pa rešavamo sledeći jednostavniji problem: naći vrednost nekog parametra a_j za fiksirane vrednosti ostalih parametara $a_1, a_2, .., a_{j-1}, a_{j+1}, .., a_r$.

$$a_i = v_i \ , i \in \{1, 2, ..r\} \setminus \{k\} \tag{4.12}$$

$$S^j_{v_1,v_2,..,v_{k-1},v_{k+1},..,v_r} : A^k \longrightarrow Rez_j \tag{4.13}$$

$$S^j_{v_1,v_2,..,v_{k-1},v_{k+1},..,v_r}(a_k) = S^j(v_1, v_2, .., v_{k-1}, a_k, v_{k+1}, .., v_r) \tag{4.14}$$

Pretpostavimo da $A^k, Rez^j \subseteq R$ tada $S^j_{v_1,v_2,..,v_{k-1},v_{k+1},..,v_r} : R \longrightarrow R$. Za funkcije ovog tipa najjednostavnija metoda nalaženja rešenja je metoda polovljenja. Osnovna ideja ove metode je da za neki interval (a, b) proveravamo da li je $S^j_{v_1,v_2,..,v_{k-1},v_{k+1},..,v_r}((a + b)/2)$ jednaka traženoj vrednost. Ako jeste problem je rešen, a ako nije proveravamo u kojoj polovini intervala može da se nalazi rešenje, pa ponavljamo postupak za taj novi interval. Modifikovana metoda polovljenja za rešavanje ovog problema je implemantirana sledećom metodom:

```
bool SimulationMultiPlayer::FindValue (int MaxDepth,CString
                   ParamName,CString ResultName,double ResultValue,
                   double SearchValueLower ,double SearchValueUpper,
                   double SearchPrecision,
                   int growDirection,int convergeSide)
{
    double   tCalculate =   (SearchValueUpper + SearchValueLower)/2;
    double   calculatedParamValue;
    CString  tValue;
    bool     UpperHalf

    if (MaxDepth < 1)
            return false;

    SetPlayerParam (tCalculate,ParamName);
    SetupSimulation ();
    mSimulator->Calculate ();
    mSimulator->mResult.GetValue (ResultName,tValue);
```

```
    if (!mSimulator->Converge () )
    {
        if (convergeSide >0)
            UpperHalf   = true;
        else
            UpperHalf   = false;
    }else{
        if (abs (calculatedParamValue - ResultValue)< SearchPrecision)
                    return true;

        if ( (growDirection *  (calculatedParamValue- ResultValue))>0)
                    UpperHalf   = true;
                else
                    UpperHalf   = false;
    }

   if (UpperHalf)
               return FindValue (MaxDepth - 1,ParamName,ResultName,
                           ResultValue,tCalculate,SearchValueUpper,
                           SearchPrecision,growDirection,convergeSide);
   else
               return FindValue (MaxDepth - 1,ParamName,ResultName,
                            ResultValue,SearchValueLower,tCalculate,
                            SearchPrecision,growDirection,convergeSide);

}
```

Metoda ima niz parametara koji imaju sledeću svrhu :

1. $MaxDepth$. Pošto je metoda rekurzivna, potrebna je granica za maksimalnu dubinu rekurzije. Ovaj parametar se smanjuje za jedan prilikom svakog sledećeg rekurzivnog poziva.

2. $ParamName$,$ResultName$ predstavljaju imena ulaznog parametara po kome se pretražuje, kao i izlaznog čija se vrednost traži (u opisu problema to su indeksi k,j).
3. $ResultValue$ predstavlja vrednost traženog izlaznog parametra.
4. $SearchValueLower$, $SearchValueUpper$ predstavljaju donju i gornju granicu intervala na kome se traži rešenje.
5. $SearchPrecision$ predstavlja maksimalnu razliku po apsolutnoj vrednosti između $ResultValue$ i vrednosti traženog izlaznog parametra simulacije izračunate u sredini intervala.
6. $growDirection$. Ovaj parametar predstavlja monotonost funkcije koju računamo na intervalu $(SearchValueLower, SearchValueUpper)$. Vrednost 1 znači da je funkcija rastuća, a -1 da je opadajuća.
7. $convergeSide$ je parametar koji nam govori sa koje strane će naša funkcija divergirati, odnosno ako je došlo do divergencije na kojoj polovini intervala treba nastaviti potragu za rešenjem. Vrednost 1 znači u gornjoj, a -1 u donjoj.

Algoritam metode je sledeći. Pre poziva metode postave se vrednost svih fiksiranih parametara na željene vrednosti nizom poziva metode $PostaviParamInt$. Prvi korak je provera da li se došlo do maksimalne dozvoljene dubine rekurzije. Ako jeste, prekida se rad i vraća se vrednost $false$ koja označava da nije nađena odgovarajuća vrednost. U suprotnom se zatim izračunava simulacija za sredinu intervala ulaznog parametra. Proverava se da li je doslo do konvergencije i ako nije, postavlja se promenjiva $UpperHalf$ na $true$ ili $false$, u zavisnosti da li je u gornjoj ili donjoj strani deo intervala u kome se očekuje da funkcija konvergira. U suprotnom se proverava da li je nađeno rešenje sa zadovoljavajućom tačnošću

$$(abs(calculatedParamValue - ResultValue) < SearchPrecision). \tag{4.15}$$

Ako jeste, vraća se vrednost $true$, koja znači da je nađeno tačno rešenje. Ako ovo nije slučaj, promenjiva $UpperHalf$ dobija vrednost izraza

$$growDirection * (calculatedParamValue - ResultValue) > 0 \quad (4.16)$$

Pošto je leva strana nejednakosti proizvod razlike tražene vrednosti i dobijene vrednosti pomnožena sa parametrom koji određu je monotonost na već definisan način, jasno je da će biti dobro određena polovina intervala u kojoj se očekuje rešenje. Na kraju se na osnovu $UpperHalf$ poziva opet ista metoda za gornju ili donju polovinu intervala.

Važno je napomenuti da osnovna matematička metoda polovljenja ima tu manu da, ako na intervalu postoji više rešenja, ona nalazi samo jedno. Sama implementacija ove metode nam donosi još niz drugih slučajeva u kojima metoda neće nalaziti rešenja.

(a) Ako početni interval nije dobro izabran, t. j. ako u njemu ne postoji ni jedno rešenje, tada ga ni ova metoda naravno ne može naći.

(b) U slučaju da interval nije monoton, metoda može u jednom koraku rekurzije da ode u interval u kome ne postoji rešenje.

(c) U slučaju pogrešne procene vrednosti parametara $growDirection$ i $convergeSide$ metoda neće naći rešenje.

Za upotrebu ove metode potrebno je znati neke osobine simulacije koje testiramo. To su interval u kome očekujemo rešenje, monotonost na intervalu, kao i to sa koje strane očekujemo da će funkcija divergirati. Ovakvi zahtevi za poznavanjem osobina funkcije na prvi pogled deluju kao preveliki, ali nakon objašnjenja uobičajenog načina upotrebe *opšteg simulatora* u narednim poglavljima će se pokazati da oni to zapravo nisu.

4.3.4 Opis klasa za imenovanje, smeštanje i pregled rezultata simulacija

Kod puštanja svih simulacija iz oblasti tako i kod traženja simulacije sa željenim osobina nailazimo na isti problem: dobijen je veliki broj rezultata i njih je potrebno zapisati na način pogodan za kasnije korišćenje. Postoje dva tipa informacija za koje želimo da postoji mogućnost pristupa. To su rezultati rada jedne simulacije i pregled odnosa nekih parametara za veći broj simulacija.

Problem imenovanja i smeštanja rezultata

Simulacija ima potrebu za nizom podataka koji je opisuju. To su vrednosti ulaznih podataka, izlaznih podataka, pregled toka izračunavanja, a neki aspekti rezultata su često vidljivi samo u grafičkim prikazima. Znači, vrlo je verovatno da će za čuvanje rezultata simulacije biti potreban niz datoteka. Iz ovog razloga je pogodno zapisivati svaku simulaciju u zaseban direktorijum. Datoteke za zapis vrednosti ulaznih i izlaznih parametara je potrebno standardizovati na neki način koji će biti pregledan za čoveka, a jednostavan za korišćenje od strane računara. Najednostavnije način zapisa bi bio sledeći :

```
ImeParametra1
VrednostParametra1
ImeParametra2
VrednostParametra2
.
.
.
ImeParametraN
VrednostParametraN
```

za svaki parametar ide red sa imenom, zatim red sa vrednošću. Datoteka za opis toka izvršavanja nema potreba za standardizacijom jer ona služi samo za

pregled rada od strane čoveka, a sam kreator modula simulacije će najbolje definisati sta je bitno za tok rada. Što se tiče grafičkog prikaza, mogu se sačuvati kao slike (bmp, jpg,...), ali je često pogodnije zapisati ih u obliku matrica sa dve dimezije.

U slučaju problema sa jednom dimenzijom

```
X1   VrednostNaPozicijiX1
...
Xk   VrednostnaNaPozicijiXk
```

u slučaju problema sa dve dimenzijom

```
X1 Y1  VrednostNaPozicijiX1Y1
...
Xk Yk  VrednostnaNaPozicijiXkYl
```

u slučaju problema sa tri dimenzijom

```
X1 Y1 Z1 VrednostNaPozicijiX1
...
Xk Yl Zm VrednostNaPozicijiXkYlZm
```

Ovako dobijene datoteke su daleko veći od slika, ali imaju više prednosti. Prva je što se rezultat može prikazati na nekoliko različitih načina izborom funkcije preslikavanja vrednosti u boju u samom simulatoru. Druga, koja je važnija, je što je to standardni način zapisa datoteka za programe za grafički prikaz rezultata kao što su *SigmaPlot*, *PsiPlot* i *OriginPro* koji mogu praviti i slike.

Ostaje još pitanje imena direktorijuma u koji će biti smeštene datoteke simulacije. Način imenovanja ovog direktorijuma moraće biti sastavni deo modula simulacije, jer će samo kreator simulacije znati najpogodniji izbor

imena. Sa druge strane, *opšti simulator* mora imati mogućnost pristupa podacima simulacija za određene vrednosti ulaznih parametara, odnosno da žna"ime odgovarajućeg direktorijuma. Ovaj problem se najlakše rešava apstraktnom klasom za imenovanje direktorijuma simulacije koja će omogućiti komunikaciju između *opšteg simulatora* i modula za simulacije. To je sledeća klasa :

```
class PathGeneratorAbstract {
 protected:
    string          Root;
    char            DelimeterStruct;
    char            DelimeterParam;
    RPodatakNiz     *mParams;
    RPodatakNiz     *mParamStruct;


public:

    PathGeneratorAbstract (void);
    ~PathGeneratorAbstract (void);

    virtual     void  GetFullPath (char *FullPath, int length)=0;
    string      GetRootString ();
    void        SetRootString (string nRoot);
    void        SetParams (RPodatakNiz  *nParams);
    void        SetParamStruct (RPodatakNiz  *nParamStruct);

};
```

Ova klasa koja ima nekoliko podataka:

- *Root* je string koji predstavlja ime početnog direktorijuma u koji će se smeštati svi direktorijumi sa rezultatima simulacije.

- $DelimeterStruct$, $DelimeterParam$ su karakteri koji će nam služiti za razdvajanje imena parametara strukture/simulacije prilikom generisanja imena, na primer $DelimeterStruct$ = "/"bi značilo da za različite vrednosti parametara strukture pravimo posebne poddirektorijume.

- $mParams$, $mParamStruct$ su nizovi koji će sadržati vrednosti parametara od kojih će se generisati imena.

Nasleđene klase će implementirati metodu $GetFullPath$ koja će nam davati punu putanju direktorijuma za odgovarajuće vrednosti parametra.

Problemi pregleda rezultata

Pregled rezultata ima dva dela: pregled rezultata jedne simulacije i pregled nekih grupnih osobina za veći broj simulacija sa različitim vrednostima parametara.

Pregled rezultata jedne simulacije je u osnovi jednostavan, potrebno je samo prikazati vrednosti ulaznih i odgovarajućih izlaznih parametara. Nešto kompleksniji deo pregleda ovih rezultata je grafički prikaz. Grafički prikaz je preslikavanje neke matrice, opisane u poglavlju o imenovanju i smeštanju rezultata, u sliku. Ovaj problem je rešavan samo za probleme sa dve dimenzije, a u tom slučaju slika je zapravo dvodimenziona matrica kod koje vrednost na odgovarajućoj poziciji predstavlja boju. Zbog jednostavnosti, a pri tome je preglednost dobijenih rezultata dovoljna, radiće se sa paletom od 256 boja, odnosno slike će imati najviše 256 različitih boja. Znači, matrica slike ce biti definisana nad skupom $B = \{0, 1, .., 255\}$. Definišimo $Bojenje$, odnosno preslikavanje sa skupa vrednosti $V \subseteq R$ na skup B na sledeći način.

$$Bojenje : V \longrightarrow B \tag{4.17}$$

$$max = Max(V) \tag{4.18}$$

$$v \in V \tag{4.19}$$

$$Bojenje(v) = round[(v/max) * 255] \tag{4.20}$$

Funkcija koja implementira ovakav način bojenja ima sledeći oblik

```
function PrikaziSliku (CString FileName, int Type,
                      float Param1,float Param2,
                      CDC *tDC)
float                  Vrednosti[DimX][DimY];
unsigned char          Slika[DimX][DimY];

UcitajSliku (FileName,Type,Param1,Param2,Vrednosti,Max);

for (int i=0; i<DimX; i++)
    for (int j=0; j<DimX; j++){
        Slika[i][j] = (unsigned char) ( (Vrednosti[i][j]/Max)*255););
        tDC->SetPixel (i,j,Paleta[int (Slika[i][j])]);
    }
```

Sam algoritam ima sledeći oblik. Prvo se učitaju vrednosti koje treba grafički predstaviti i odredi maksimalna vrednost među njima. Zatim se za svaku od vrednosti odredi odgovarajući indeks boje. Konačno se na odgovarujućoj poziciji postavi piksel sa bojom koja u paleti ima taj indeks.

Različite osobine rezultata se mogu uvideti postavljem različitih filtera na matricu vrednosti. Filteri su neka preslikavanja oblika: $Filter : V \longrightarrow R$. Oni su sastavni deo funkcije $UcitajSliku$ i zato postoje dodatni parametri $Type$ koji određuje o kom filteru se radi, kao i $Param1$,$Param2$ koji su parametri potrebni filteru. Definisaćemo sledeću klasu koja će rešavati problem filtriranja

```
class RFilter {
    int            Type;
    float          Param1;
    float          Param2;
```

```
public:
    RFilter (void);
    ~RFilter (void);

    void SetParams (float Param1, float Param2);
    void SetType (int Type);

    float       Linear (float Value);
    float       Sqrt (float Value);
    float       Log (float Value);
    float       Pow (float Value);
    float       Convert (float Value);

};
```

Način upotrebe je sledeći. Prvo se izabere odgovarajući filter i postave vrednosti potrebnih parametara.

```
        Filter.SetType (Type);
        Filter.SetParams (Param1,Param2);
```

Filter ćemo koristiti nad poljima matrice vrednosti tako da će postojati petlja koja će svaku vrednost učitati i konvertovati na način određen filterom.

```
        fscanf (F,"%f %f %f\n",&X,&Y,&temp);
        Vrednosti[X][Y]     = Filter.Convert (temp);
```

Postoji veliki broj mogućih konverzija. U *opštem simulatoru* su implementirane one određene funkcijama:

- $Linear(Value) = Param1 * Value + Param2$. Pogodna je ako želimo da vidimo sve vrednosti iznad neke granice.

- $Sqrt(Value) = \sqrt{Value}$, $Pow(Value) = Param1^{Value}$ Filter Pow ima dva tipa efekta u zavisnosti od parametra $Param1$. Ako je $Param < 1$, upotrebom ovog filtera, polja sa manjim vrednostima doći će više do izražaja i suprotno tome, ako je $Param > 1$ ova polja se gube i počinju da liče na minimalnu vrednost skale. Pošto se filter Pow sa parametom od 0.5 često upotrebljava, $Sqrt$ je njegov skraćeni zapis.

- $Log(Value) = log_{Param1}Value$ Ovim filterom se postiže da kasnije bojenje koristi logaritamsku skalu.

Drugi problem kod pregleda rezultata je otkrivanje zavisnosti koje postoje između ulaznih i izlaznih parametara. Slično tome, moguće je da se pojavi potreba za uviđanjem odnosa između nekih izlaznih parametara. Ovakve odnose je najlakše prepoznati i okarakterisati iz raznih tipova grafika, grafikona, tabela sa stubićima i sl. Iz razloga postojanja velikog broja softverskih paketa koji su specijalizovani upravo za ovakve grafičke prikaze, *opšti simulator* ih ne kreira već samo generiše datoteke koje će moci da se koriste iz ovih paketa. Standardni oblik datoteke koju koriste ovi paketi je sledeći:

```
Elem1Vrednost1 Elem1Vrednost2 ... Elem1VrednostN
Elem2Vrednost1 Elem2Vrednost2 ... Elem2VrednostN
...
ElemMVrednost1 ElemMVrednost2 ... ElemMVrednostN
```

To je zapravo tekstualna datoteka u kojoj postoji po jedan red za svaki element, a u tom redu su vrednosti od interesa odvojeni znakovima beline.

Za kreiranje ovakvih datoteka nam je potrebna pomoćna klasa u kojoj će se definisati izlazna datoteka, odnosno podaci koji će se u nju upisivati. Ova klasa ima sledeći oblik:

```
class RGraphicMaker {
    CString      NeededParams[50];
    int          NumberOfNeeded;
```

```
public:
    RGraphicMaker (void);
    ~RGraphicMaker (void);

    CString*            GetNeededParams (){return NeededParams;}
    int                 GetNumberOfNeeded (){return NumberOfNeeded;}
    void                AddParam (CString nParam);
    void                Clear (){NumberOfNeeded = 0;}
    RGraphicMaker       operator = (RGraphicMaker nGraphicMaker);

};
```

Sâmo generisanje datoteka će se vršiti metodom *CreateGraphFile* klase *SimulationMultiPlayer*. Ova metoda je skoro analogna metodi *PlayAllSimulations*, osim što će se umesto metode *Calculate* klase *SimulatorAbstract* pozivati *GetGrafikFileRow*. Ova metoda neće računati simulacije, već će predhodno izračunate rezultate učitavati iz odgovarajućih datoteka, a kao rezultat će vraćati string sa odgovarajućim podacima formatiranim na traženi način.

Glava 5

Implementacija proširenja za *opšti simulator*

5.1 Sistem modula (plaginova)

Sistem plaginova je način proširivanja funkcionalnosti gotovih aplikacija. On se svodi na dodavanje novih dll-ova koji sadrže tu novu funkcionalnost. Prednost ovakvog načina rada je u tome što je unapređenje osnovne aplikacije omogućeno bez aktivnog učešeća njenog kreatora. Ne postoji ni potreba za posedovanjem koda osnovne aplikacije, jer prilikom korišćenja dll-ova nema potrebe za njenim dodatnim kompajliranjem, već samo za kodom koji nam služi za definisanje dll-a. Sistem plaginova je dodatno unapređenje u odnosu na standardnu upotrebu dll-ova. Prilikom standardne upotrebe dll-a, poznato je koje funkcije (ili/klase) stoje u njemu i kako su definisane, pa se unapređenje vrši zamenom postojećeg dll-a novom unapređenom verzijom. Na ovaj način se zapravo jedna datoteka (npr. *test.dll*) menja novom verzijom sa istim imenom. Ovako se lako unapređuje i optimizuje funkcionalnost postojeće aplikacije, ali se ona teško proširuje. Kod sistema plaginova postoji nadgradnja upotrebe dll-ova. Precizno se opisuje kako izgledaju funkcije (i/ili klase) koje se nalaze u dll-u (funkcije se mogu nalaziti i u većem broj dll-ova, ali to nije uobičajeno) koji predstavlja plagin, ali postoji i klasa u samoj aplikaciji koja nam omogućuje učitavanja većeg broja ovakvih dll-ova. Tako

će se sada za svako proširenje aplikacije dodavati datoteka sa novim imenom (npr. test1.dll), a popravke funkcionalnosti vršiće se zamenom istoimenih datoteka novim verzijama.

5.2 Smeštanje i upotreba plaginova

Prvi problem sa kojim se srećemo je kako prepoznati koji dll-ovi su zapravo plaginovi za *opšti simulator*. Prvi i najjednostavniji pristup je smeštanje dll-ova u zaseban direktorijum. Kod *opšteg simulatora* to će biti *INSTALL-DIR/Plugins*. Sve dll-ove (odnosno datoteke sa ekstenzijom .dll) iz ovog direktorijuma pokušavamo da učitamo kao plaginove.

Drugi problema prilikom upotrebe dll-ova je to što svu memoriju koju smo počeli da koristimo u dll/Osnovnoj aplikaciji moramo da oslobodimo u istom dll/Osnovnoj aplikaciji. Kao što je već rečeno, komunikacija sa simulacijama će se vršiti preko apstraktne klase *SimulationAbstract*, odnosno u svakom dll-u koji je plagin, biće definisana nasleđena klasa koja implementira funkcionalnost. Da bi polimorfizam klasa funkcionisao u C++, potrebno je raditi sa pokazivačima na objekte, a ne sa samim objektima. Zbog toga ćemo biti u obavezi da kreiramo objekte u dll-u, a da u glavnoj aplikaciji koristimo pokazivače na njih. Za rešavanje ovih problema biće nam potrebne dve klase, prva čiji će objekti biti u glavnoj aplikaciji koja će služiti za učitavanje plaginova u glavnu aplikaciju, i druga koja će biti zadužena za isporučivanje odgovarajućih pokazivača i oslobađanje odgovarajuće memorije.

Klasa u glavnoj aplikaciji, koja učitava i čuva sve plaginove, ima sledeći oblik:

```
class RPluginLoader
 { protected:
    SimulatorAbstract*  Simulators[MAX_PLUGINS];
    HINSTANCE           Dlls[MAX_PLUGINS];

    int                 MaxIndex;
```

```
public:
    RPluginLoader (void);
   ~RPluginLoader (void);

    void                    LoadPlugin (string   DllName);
    void                    LoadPluginsFromDirectory (string   DirectoryName);
    int                     GetMaxIndex (){return MaxIndex;}
    string                  GetName (int Index);
    SimulatorAbstract*      GetSimulator (int Index);
    void                    FreeDLLs ();
};
```

Ova klasa ima tri podatka, a to su:

- *Simulators* je niz pokazivača na učitane simulatore.
- *Dlls* je niz identifikatora dll-ova koje smo učitali.
- MaxIndex je brojač koliko smo dll-ova učitali.

LoadPluginsFromDirectory je metoda koja učitava sve plaginove iz odgovarajućeg direktorijuma. Njeno izvršavanje se sastoji iz niza poziva metode *LoadPlugin*. Metoda *LoadPlugin* sadrži nekoliko važnih koraka, koje ćemo posebno izdvojiti:

```
DllInstance = ::LoadLibrary (DllName.c_str ());
```

je prvi korak, koji zapravo učitava dll u memoriju, i daje nam njen identifikator koji će služiti za njeno korišćenje.

```
typedef void* ( *REGFUNCADDR) (); Func = (REGFUNCADDR)
GetProcAddress (DllInstance, "GetSimulation");
```

Nakon toga, iz dll učitavamo funkciju *GetSimulation* koja nam vraća pokazivač na simulator koji je sadržan u datom plaginu. Ako su predhodna dva koraka uspešno izvršena, ostaje nam još da učitamo iz dll-a pokazivač na odgovarajući simulator i da njega i identifikator dll-a sačuvamo u odgovarajućim nizovima.

```
thisSimulator = (SimulatorAbstract *) regFunc (); if
(thisSimulator){
                Simulators[MaxIndex]  = thisSimulator;
                Dlls[MaxIndex]        = DllInstance;
                MaxIndex++;
}
```

FreeDLLs je metoda koja se poziva na kraju rada aplikacije, zadužena za oslobađanje memorije. Oslobađaju se simulatori

```
FuncFreeSimulation= (REGFUNCADDR)GetProcAddress
(Dlls[i],"FreeSimulations"); FuncFreeSimulation (),
```

tako što se iz dll-a učita funkcija *FreeSimulations*, koja oslobađa odgovarajuće pokazivače. Drugi deo je oslobađanje dll-ova

```
FreeLibrary (Dlls[i]);
```

Funkcije *GetSimulator*, *FreeSimulations*, služe za kontrolu memorije u samom dll-u iz osnovne aplikacije. Za njihovu implementaciju je neophodna sledeća klasa:

```
    class SimulatorFactory
{
    int                          MaxIndex;
    SimulatorAbstract            *Simulators[SIMULATOR_FACTORY_MAX_INDEX];
```

```
public:
    SimulatorFactory (void);
    ~SimulatorFactory (void);
    void        AddSimulator (SimulatorAbstract *N);
    void        FreeSimulators ();
};
```

5.3 Kreiranje plaginova

Upotreba plaginova u *opštem simulatoru* ima dva cilja. Prvi je lako proširivanje funkcionalnosti osnovne aplikacije novim tipovima simulacija. Drugi je da se što više olakša razvoj novih simulacija, t. j. da se njihova implementacija što je moguće više svede na sâmo programiranje numerike, koja je potrebna za njeno izračunavanje, jer većina potencijalnih korisnika nije iskusna u programiranju. Pojednostavljivanje programiranja plaginova se zasniva na kreiranju biblioteke sa klasama koje će implementirati svu osnovnu i uobičajenu funkcionalnost. Sa druge strane, razvoj novih plaginova se pojednostavljuje isporukom šablon projekta za njihov razvoj.

RCommon

RCommon je ime biblioteke koja sadrži nekoliko pomoćnih funkcija, kao i sledeće klase:

- RPodatak.
- RPodatakNiz.
- SimulatorAbstract.
- PathGeneratorAbstract.
- SimulatorFactory.

- RTranslator, RTranslatorPair.

Sve klase osim *RTranslator*, *RTranslatorPair* su opisane u prethodnim poglavljima. *RTranslator* je pomoć na klasa, koja služi za olakšavanje implemetacije podataka čija vrednost pripada određenom skupu podataka. Za takve podatke je često pogodno da se mogu predstaviti na dva načina, jedan koji bi se prikazivao korisniku aplikacije iz GUI-a, a drugi koji će se koristiti u samom programu (na primer, prikaz imena u obliku niza flagova). *RTranslator* je zapravo klasa koja implementira ovakav prevodilac. Ona koristi pomoćnu klasu:

```
class RTranslationPair{
        public:
   string      Original;
   string      Translation;
   string           GetTranslation (){return Translation;}
   string           GetOriginal (){return Original;}
};
```

koja predstavlja par (original, prevod). Sama klasa ima sledeći oblik:

```
class RTranslator {
    RTranslationPair    mPairs[MAX_PAIR_INDEX];
    int                 MaxIndex;
public:
    RTranslator (void);
    void        AddPair (string  Original, string Translation);
    void        AddPair (int  Original, string Translation);
    bool        GetOriginal (string Translation, string &Original);
    bool        GetTranslation (string Original, string &Translation);
    ~RTranslator (void);
```

Upotreba prevodioca je sledeća. Prvo se prevodiocu doda niz parova (original, prevod):

```
TranslatorInputRay.AddPair ( 1,"GAUS");
TranslatorInputRay.AddPair (  2,"DIPOL");
TranslatorInputRay.AddPair (  3 ,"KVAD");
TranslatorInputRay.AddPair (  4,"VORTEX");
```

a kasnije se na osnovu originala dobija prevod i obrnuto:

```
GetTranslation ("1", Translation);
GetOriginal ("GAUS", Original);
```

Šablon projekat za plagin

Šablon projekat za plagin se sastoji iz sledećih datoteka:

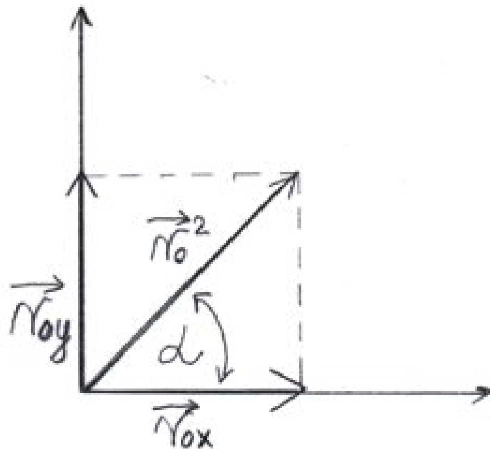
- SimulatorTest.h,SimulatorTest.cpp u kojima je definisana klasa koja je nasleđena od *SimulatorAbstract*.
- PathGeneratorTest.h, PathGeneratorTest.cpp u kojima je definisana klasa nasleđena od *PathGeneratorAbstract* sa prostom implementacijom metode *GetFullPath*.
- Test.cpp je datoteka u kojoj su definisane funkcije koje se eksportuju iz dll-a.

```
#include ".\h\SimulatorPetrasvili.h"

#include "..\..\Rcommon\h\SimulatorFactory.h"

SimulatorFactory mFactory;

extern "C" __declspec (dllexport) void*  GetSimulation () {

    SimulatorAbstract *Temp;
    Temp    = new SimulatorTest ();
```

```
        mFactory.AddSimulator (Temp);
        return Temp;
    }

    void extern "C" __declspec (dllexport) void  FreeSimulations () {
        mFactory.FreeSimulators ();
    }
```

5.4 Kosi hitac kao prost primer implementacije proširenja *opšteg simulatora*

Simulacija kosog hitaca zapravo predstavlja simulaciju kretanja tela koje ima pocetnu brzinu v, čiji je intenzitet v_0 a ugao u odnosu na horizontalnu ravan α, pod uticajem gravitacije čije je ubrzanje g.



Za ovakvo kretanje imamo nekoliko vrednosti od interesa:

- Zavisnost vertikalne (y) i horizontalne (x) udaljenosti od pocetne tačke

kretanja. Ova zavisnost se može izraziti sledećom formulom:

$$y = x \tan \alpha - \frac{g^2 x}{2 v_0 \cos^2 \alpha} \tag{5.1}$$

- Najveću postignutu visinu H koja se izračunava formulom :

$$H = \frac{v_0^2 \sin^2 \alpha}{2g} \tag{5.2}$$

- Najveću postignutu udaljenost D koja se izracunava sledećom formulom

$$D = \frac{v_0^2 \sin 2\alpha}{g} \tag{5.3}$$

Prilikom kreiranja proširenja, prvi korak je da se kreiraju odgovarajuće klase za izvršavanje simulacije i za smeštanje dobijenih rezultata.

```
class SimulatorKosiHitac :   public SimulatorAbstract

class PathGeneratorKosiHitac : public PathGeneratorAbstract
```

Pošto je u pitanje jednostavan primer, ne postoji potreba za proširenjem funkcionalnosti klasa za smeštanje rezultata u odnosu na osnovnu klasu koja rešava taj problem.

Postoji niz koraka, odnosno metoda klase SimulatorKosiHitac koje treba implementirati da bi proširenje dobilo svoju funkcionalnost.

- Potrebno je da se obezbedi identifikovanje simulacije, odnosno implementacija sledeće metode.

  ```
  void    GetName (char *gName){strcpy_s (gName,1000, "KosiHitac");}
  ```

- Potrebno je izvršiti niz inicijalizacija u konstruktoru ove klase:

- Odrediti direktorijum u kome će biti smešteni poddirektorijumi koji će sadržati rezultate simulacija.

  ```
  mPathGenerator->SetRootString ("d:\\primeri\\KosiHitac\\");
  ```

- Potrebno je definisati dva tipa parametara. Prvo parametre simulacije odnosno parametre koji su nam od najvećeg interesa za istraživanje i čije će se vrednosti često menjati.

  ```
      AddParam ( (double)0.795,"Ugao");
      AddParam ( (double)2,"Brzina");
  ```

  a zatim parametre strukture, odnosno parametre koji će se ređe menjati. Oni najčešće služe za fizičke konstante i za definisanje izlaza (NXY predstavlja dimenzije slike izlaza, a MaxGrafik određuje skalu koja se koristi prilikom njenog kreiranja)

  ```
      AddParamStruct ( (double)9.81, "Gravitacija");
      AddParamStruct ( (int)256, "NXY");
      AddParamStruct ( (double)100, "MaxGrafik");
  ```

• Potrebno je da se obezbedi sâmo izračunavanje simulacije i smeštanje rezultata odnosno impementacija metode Calculate (). Prilikom implementacije ove metode imamo nekoliko važnih koraka. Prvi je da se obezbedi da se rezultati nalaze na odgovarajućem mestu, odnosno da se postavi odgovarajući aktivni direktorijum:

  ```
  mPathGenerator->GetFullPath (temp, 5000);
  createAllDirectories ( (const char *)temp);
  SetCurrentDirectoryA (temp);
  ```

  , zatim pristup parametrima simulacije

```
mParams.GetValueDouble ("Brzina",V);
mParams.GetValueDouble ("Ugao",Alfa);

mParamStruct.GetValueDouble ("Gravitacija",temp1);
mParamStruct.GetValueDouble ("MaxGrafik", MaxGrafik);
mParamStruct.GetValueDouble ("NXY", NXY);
```

Sledeći korak je izracunavanje vrednosti od interesa za simulaciju i njihov zapis u datoteku na ranije opisani način i eventulano kreiranje odgovarajućih slika

```
MaxVisina   =  (pow (V,2) * pow (sin (2*Alfa),2))/ (2 *G);
MaxDaljina  = (pow (V,2) * sin (2*Alfa))/G;

fopen_s (&F, "Rezultat.txt","w");

fprintf (F,"MaxVisina\n");
fprintf (F,"%f\n", MaxVisina);

fprintf (F,"MaxDaljina\n");
fprintf (F,"%f",MaxDaljina);

fclose (F);
```

Poslednji korak je da se dobijeni rezultati učitaju u simulator:

```
mResult.LoadFromFile ("Rezultat.txt");
```

Glava 6

Grafički korisnički interfejs (Graphic User Interface, GUI)

Do sada su bili opisivani problemi komunikacije *opšteg simulatora* sa modulima u kojima su implementirane simulacije nekih problema, a u ovom poglavlju će biti bliže opisana komunikacija između čoveka i *opšteg simulatora*, odnosno GUI i njegovo generisanje.

6.1 Dva moguća pristupa kreiranju GUI-a

Ova komunikacija se vrši preko niza dijaloga u koje mogu da se unose različiti tipovi podataka. Postoje dva moguća pristupa implementaciji ovih dialoga:

- Prvi pristup je da programer modula kreira i ove dialoge. Na ovaj način, komunikacija između *opšteg simulatora* i modula sa implementacijom određene simulacije postaje naizgled prostija za dva koraka. Prvi je da modul prosledi *opštem simulatoru* koji parametri su mu potrebni, a drugi da *opšti simulator* pošalje vrednosti parametara modulu. Međutim, uprošćenje postoji samo u slučaju kada puštamo samo po jednu simulaciju, ali u slučaju puštanja svih simulacija iz nekog intervala ili traženja simulacija sa određenim osobinama, opet su nam potrebna oba koraka. Još jedna mana ovog pristupa je što se od kreatora mod-

ula traži znanje kreiranje ovakvih dialoga i sužava mu se izbor alata, u kojima može da piše svoj modul, na one u kojima je moguće i pisati neku vrstu GUI-a.

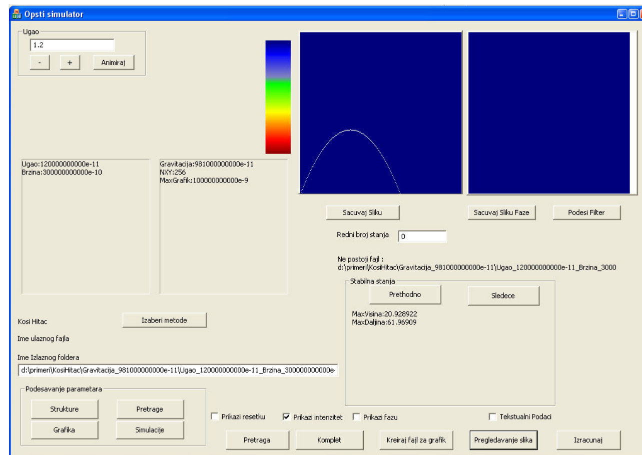
- Drugi pristup je da *opšti simulator* kreira ove dijaloge. Kao što je već rečeno, sva potrebna komunikacija između modula i osnovne aplikacije je već implementirana, tako da je potrebno samo na osnovu tih informacija kreirati dialoge. Na ovaj način se postiže i to da se od kreatora modula jedino zahteva da ispoštuje specifikaciju komunikacije i da implementira svoj modul u nekom okruzenju koje omogućuje kreiranje dll-a, koji se može koristiti iz C++ implementiranog u "Microsoft Visual Studio-u. U *opštem simulatoru* je iskorišćen ovaj poslednji pristup.

6.2 Opis GUI-a, odnosno dijaloga iz kojih se sastoji

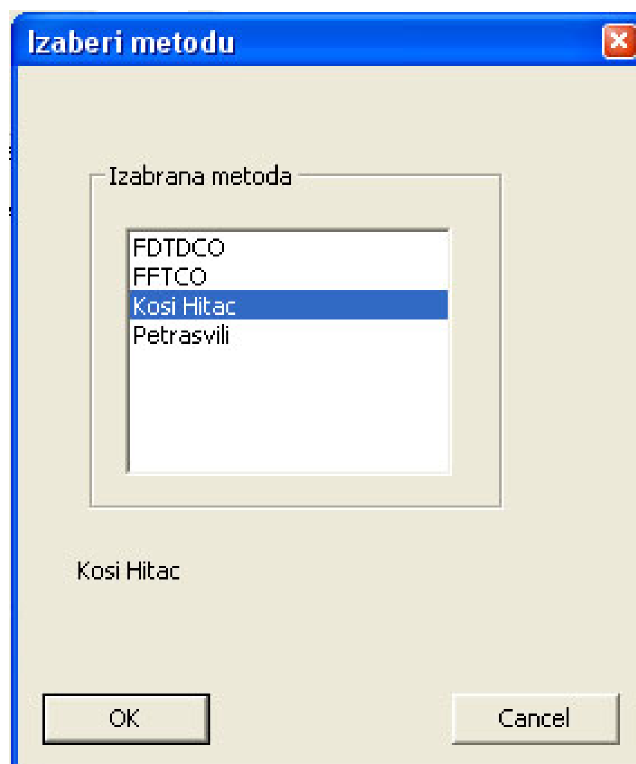
Važno je razumeti da je GUI *opšteg simulatora* u neraskidivoj vezi sa modulima za pojedine simulacije. Bez ovakvih modula, osnovna aplikacija zapravo i nema funkcionalnost, pa samim tim ni operativni korisnički interfejs. GUI i njegovo generisanje će biti opisani kroz primer plagina za kosi hitac.

GUI *opšti simulator* ima niz prozora (dialoga) i to su :

- Glavni prozor. On nam omogućuje pristup ostalim dijalozima i prikazuje sve informacije vezane za trenutno aktivni simulator.

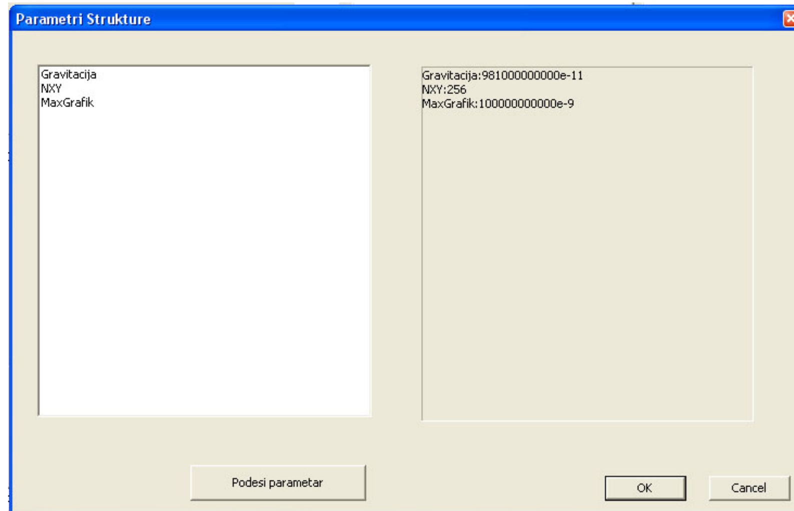


- Dijalog za izbor proširenja. Izborom novog proširenja će se izmeniti izgled ostalih dijaloga:

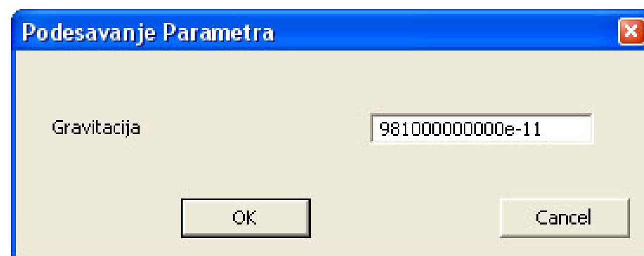


- Dialog za izbor parametra strukture koji želimo da promenimo. Lista ovog dijaloga se automatski puni na osnovu proširenja koje je trenutno

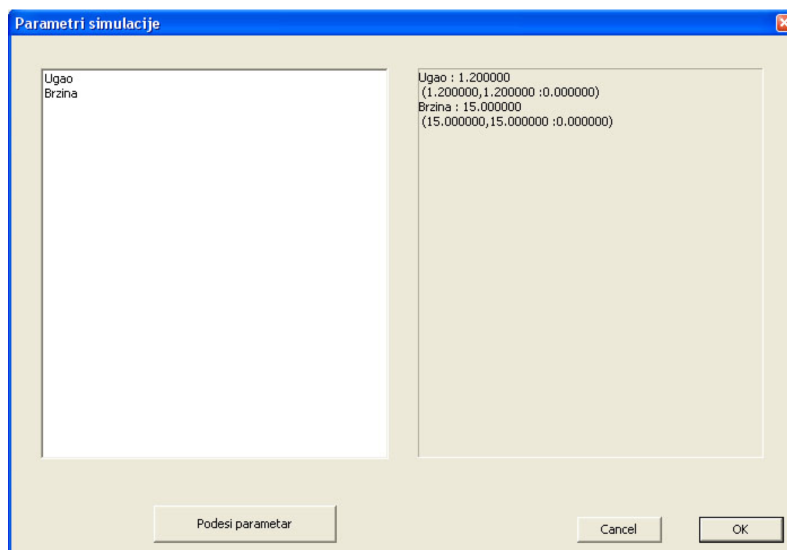
aktivno. Parametri strukture su prosti parametri, odnosno to je parametar koji ima samo jednu vrednost kojom je u potpunosti opisan.



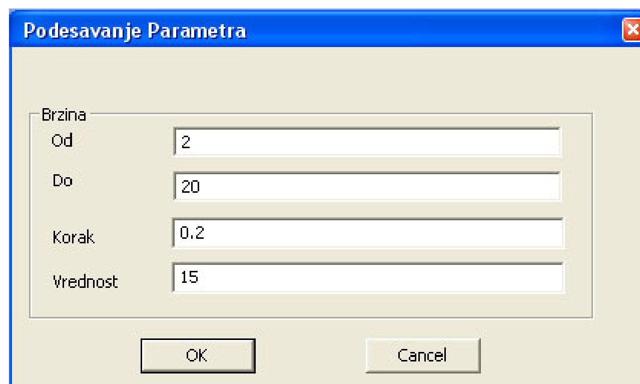
- Dijalog za podešavanje prostih parametara. Ovakav dijalog mora imati mogućnost prikazivanja imena parametra i postavljanja njegove vrednosti.



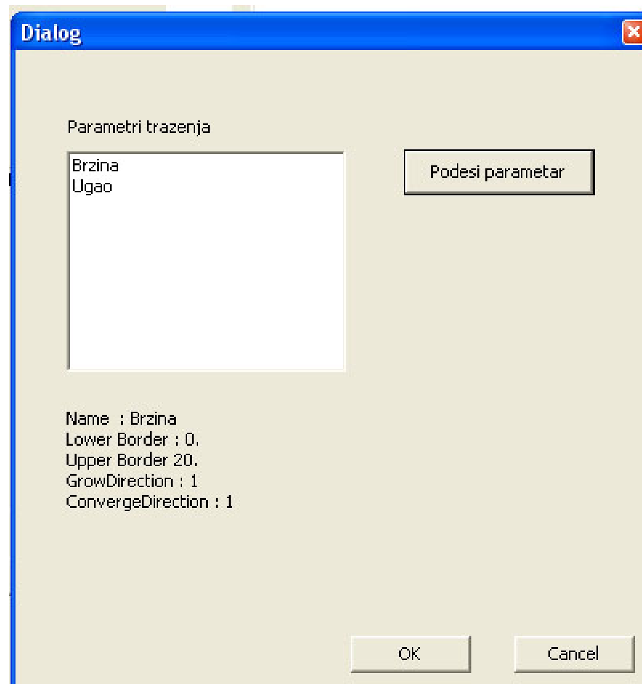
- Izbor parametra simulacije koji želimo da promenimo. I ovaj dijalog se automatski kreira na osnovu aktivnog plagina.



- Dijalog za podešavanje parametara simulacije. On takođe mora da ima mogućnost prikazivanja imena parametra i postavljanja niza bitnih vrednosti parametara, kao što su interval u kome se njegove vrednosti mogu nalaziti (Od, Do), aktivna vrednost i korak (gustina za puštanje svih simulacija iz intervala).



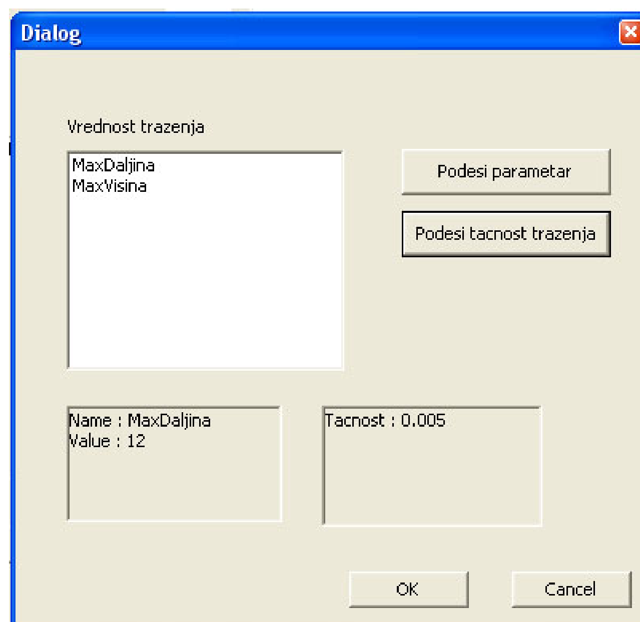
- Podešavanje pretrage. Sastoji se iz nekoliko dialoga. Prvi dijalog služi za izbor parametra po kome će se pretraživanje vršiti. Ovaj prozor se puni parametrima trenutno aktivne simulacije, a on ujedno prikazuje i karakteristike pretrage koja će se vršiti:



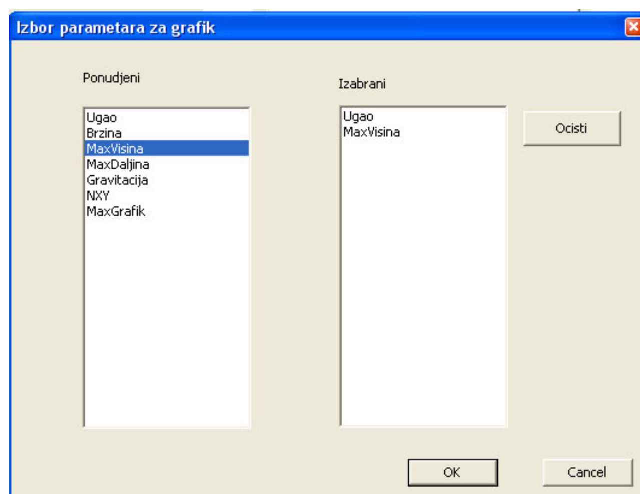
Sledeći prozor je za podešavanje karakteristika pretrage za izabrani parametar iz predhodnog prozora:

, treći prozor služi za podešavanje tražene izlazne vrednosti. Prozor se puni listom izlaznih parametara koji odgovaraju aktivnoj simulaciji.

On otvara dijaloge za podešavanje prostih parametara i to za tačnost pretrage, kao i same vrednosti koja se traži. Prozor još prikazuje izabrane vrednosti ovih parametara.



- Izbor parametara za kreiranje grafika. Sastoji se iz dve liste. Jedna se puni sa svim parametrima koje koristi simulacija (ulaznim, izlaznim i strukturnim), a u drugu se upisuju oni parametri koje želimo da imamo u datoteci za generisanje grafika.



Kao što je već rečeno, GUI je implementiran korišćenjem MFC-a, odnosno njegovih određenih klasa.

Glava 7

Primer proširenja za *opšti simulator*

7.1 Fizički model

Upotreba *opšteg simulatora* i pisanje odgovarajućeg proširenja za njega će biti prikazano praveći simulaciju za problem nalaženja solitonskih rešenja u fotoničnim rešetkama.

7.1.1 Opis modela

Solitoni

U linearnoj optičkoj sredini, usled difrakcije, talasni snopovi se šire prilikom propagacije. Međutim, u nelinearnoj sredini, u nekim slučajevima talasi mogu propagirati tako da se ne šire i ne rasipaju, već čuvaju svoj oblik i veličinu. U slučaju takve propagacije, formiraju se dinamički i strukturno stabilni objekti, tzv. optički solitoni. Nastanak i stabilnost solitona potiče iz ravnoteže između disperzije ili difrakcije (koje teže da razruše talas) i nelinearnih fenomena (koji teže da ga lokalizuju). Na taj način širenje talasa može biti potpuno kompenzovano nelinearnim fenomenima. Solitoni se mogu smatrati kao talasi koji su ograničeni na određeni interval vremena i određeni region prostora.

Prostiranje laserskih zrakova u fotorefraktivnim sredinama

Fotorefraktivne sredine su sredine u kojima može doći do fotorefraktivnog efekta, to jest svetlošću izazvane lokalne promene indeksa prelamanja. Interakcija laserskih zrakova u fotorefraktivnim kristalima može se opisati talasnom jednačinom, koja je bazirana na Kukhtarevljevom modelu. U ovom magistarskom radu, talasna jednačina je rešavana u aproksimaciji sporo menjajućih obvojnica električnog polja i u paraksijalnoj aproksimaciji (gde se smatra da se svetlost prostire pod malim uglom u odnosu na optičke ose). Razmatrane su različite propagacione geometrije laserskih zrakova (kopropagirajuće i kontrapropagirajuće), kao i različite forme ulaznog zraka (Gausijani, vorteksi, dipoli, kvadrupoli...).

Fotonične rešetke

Fotonične rešetke su jedna od realizacija koncepta fotoničnih kristala. To su materijali koji poseduju periodičnu prostornu strukturu, zahvaljujući kojoj je moguće kontrolisati prostiranje svetlosti kroz njih. Fotonični kristali se mogu smatrati optičkim analogonom poluprovodnika, u smislu da menjaju propagacione karakteristike svetlosti na isti način kao što atomske rešetke menjaju osobine elektrona zbog svoje zonske strukture. To znači da možemo dizajnirati i konstruisati fotonične kristale sa fotoničnim zabranjenim zonama, koje sprečavaju prostiranje svetlosti date frekvencije u nekim (ili svim) pravcima.

7.1.2 Simulacija propagacije solitonskih rešenja

Da bi se objasnilo ponašanje kontrapropagirajućih solitona, korišćen je vremenski nezavisan model zasnovan na teoriji fotorefraktivnog efekta. Model se sastoji iz dve talasne jednačine u paraksijalnoj aproksimaciji za propagaciju zraka u izotropnoj sredini. Modelne jednačine u računarskom prostoru glase:

$$i\partial_z F = -\Delta F - \Gamma F \frac{I + I_g}{1 + I + I_g}, \tag{7.1}$$

$$-i\partial_z B = -\Delta B - \Gamma B \frac{I + I_g}{1 + I + I_g}, \tag{7.2}$$

gde su F i B obvojnice zrakova koji propagiraju s leve i desne strane, Δ je transverzni laplasijan, a Γ je bezdimenziona konstanta sprezanja. Veličina $I = |F|^2 + |B|^2$ je intenzitet laserskog zračenja. Skaliranje prostornih koordinata na način $x/x_0 \to x$, $y/x_0 \to y$ i $z/L_D \to z$ je urađeno da bi se dobile bezdimenzione propagacione jednačine; x_0 predstavlja tipičnu širinu zraka a L_D difrakcionu dužinu.

Optički indukovana rešetka I_g je formirana postavljanjem gausijanskih zrakova na čvorove rešetke. Posmatrani su razni tipovi rešetki: heksagonalna, kvadratna i cilindrična.

Ove jednačine se rešavaju metodom propagirajućih zrakova. Koristi se brzi Furijeov transform (FFT) za transformaciju jednačina u inverzni prostor, u kome se i rešavaju. Kao i pri rešavanju svih parcijalnih diferencijalnih jednačina i ovde su granični i početni uslovi veoma bitni i moraju se uzimati u obzir.

7.2 Petviashvilijev metod za traženje solitonskih rešenja u fotoničnim rešetkama

Za traženje solitonskih rešenja u fotoničnim rešetkama korišćen je dobro poznati i provereni Petviashvilijev metod, koji je ovde malo modifikovan da bi se popravila konvergencija rešenja.

Zbog rotacione simetrije problema, prethodne jednačine sugerišu postojanje solitona u obliku $u(x, y)$:

$$F = u(x, y) \cos\theta \exp(i\mu z), \tag{7.3}$$

$$B = u(x, y) \sin\theta \exp(-i\mu z), \tag{7.4}$$

gde je θ proizvoljni ugao projekcije a μ je propagaciona konstanta. Jednačine (7.1) i (7.2) su transformisane u jednu (istu) degenerisanu jednačinu:

$$-\mu u + \Delta u + \Gamma u \frac{|u|^2 + I_g}{1 + |u|^2 + I_g} = 0. \tag{7.5}$$

Rešenja jednačine (7.5) u formi osnovnih, dipolnih ili vorteksnih solitona mogu se dobiti iteracionom procedurom u Furijeovom inverznom prostoru. Prvo se definiše linearni i nelinearni član:

$$P = \frac{\Gamma I_g}{1 + |u|^2 + I_g}, \tag{7.6}$$

$$Q = -\frac{\Gamma |u|^2 u}{1 + |u|^2 + I_g}. \tag{7.7}$$

Zatim se izvrši Furijeova transformacija jednačine (7.5) i dobija se:

$$\hat{u} = \frac{1}{|\vec{k}|^2 + \mu}[\widehat{Pu} - \hat{Q}]. \tag{7.8}$$

Ovde "$\hat{}$ "označava dvodimenzionalni (2D) Furije transform. Pravolinijska (naivna) iteracija jednačine (7.8) ne daje solitonska rešenja niti konvergira. Sledeći ideju koju je dao Petviashvili, uvodimo stabilizacione faktore (projektore) u jednačini (7.8):

$$\alpha = \int [(|\vec{k}|^2 + \mu)\hat{u} - \widehat{Pu}]\hat{u^*}\overrightarrow{dk}, \tag{7.9}$$

$$\beta = -\int \hat{Q}\hat{u^*}\overrightarrow{dk}, \tag{7.10}$$

i onda se konstruiše sledeća iteraciona šema:

$$\hat{u}_{m+1} = \frac{1}{|\vec{k}|^2 + \mu}\left[\left(\frac{\alpha_m}{\beta_m}\right)^{\frac{1}{2}} \widehat{P_m u_m} - \left(\frac{\alpha_m}{\beta_m}\right)^{\frac{3}{2}} \hat{Q}_m\right]. \tag{7.11}$$

Kad je konvergencija dostignuta, važi $\alpha_m = \beta_m$ i solitonska rešenja su pronađena. Koristeći ovu iteracionu proceduru, dobijaju se različita solitonska rešenja koja će biti predstavljena u ovom radu.

7.2.1 Rešavanje problema simulacijom

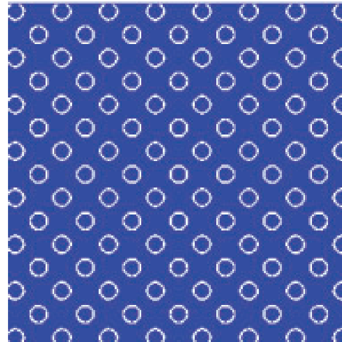
Rešavanje parcijelnih diferencijalnih jednačina analitičkim putem je moguće samo za mali broj grupa ovakvih problema, a jednačina (7.5) nije u jednoj od tih grupa. Posebno je teško naći solitonska rešenja analitičkim putem. Mi ćemo simulirati ovaj problem na konačnom delu prostora $((-xmax, xmax) \times (-ymax, ymax))$.

Rešetke

Problem je simuliran za nekoliko tipova rešetki intenziteta I_g, sa određenim defektima. Ove rešetke su definisane sledecim formulama i slikama koje prikazuju njihov oblik:

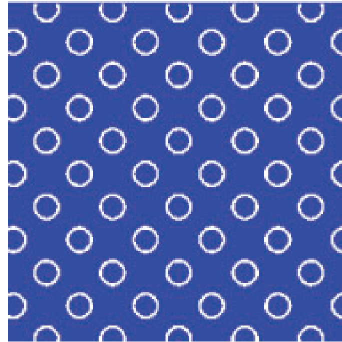
- Kosa kvadratna kosinusna

$$I_g(x, y) = I_0(cos(\frac{\pi(x+y)}{d}))^2(cos(\frac{\pi(x-y)}{d}))^2 \tag{7.12}$$



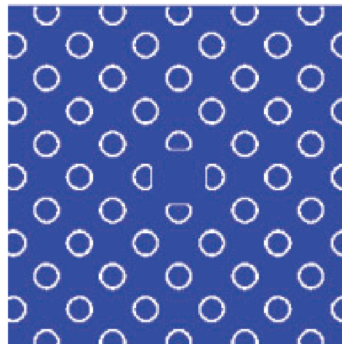
- Kosa kvadratna sinusna bez defekta

$$I_g(x, y) = I_{sin} = I_0(sin(\frac{\pi(x+y)}{d * \sqrt{2}}))^2(sin(\frac{\pi(x-y)}{d\sqrt{2}}))^2 \tag{7.13}$$



sa defektom

$$I_g(x,y) = I_{sin}(1 - e^{-\frac{1}{8}(\frac{(x^2+y^2)\pi^2}{2d^2})^4}) \quad (7.14)$$

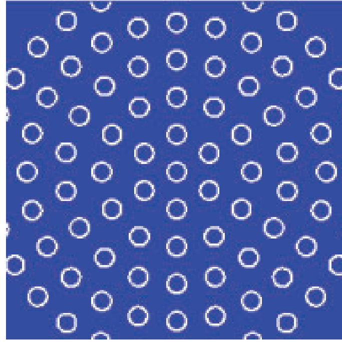


- Cilindrična, heksagonalna. Ove dve rešetke je teško zapisati u obliku jedne formule već je lakše opisati algoritam kojim se generišu:

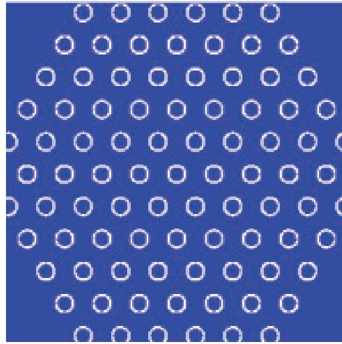
  Prvo za sve vrednosti x, y:
  $I_g(x,y) = 0$
  zatim za svaki kružic (centar je (x_c, y_c)) i za sve vrednosti x, y sumirajmo:

$$I_g(x,y) = I_g(x,y) + I_0 e^{-2\frac{(x-x_c)^2-(y-y_c)^2}{2\sigma^2}}) \quad (7.15)$$

  a pozicije kružica su opisane slikama, za cilindricnu :



, a za heksagonalnu :



Za ova dva tipa rešetki vršeni su i numerički eksperimenti i za rešetke sa različitim defektima. Kod cilindrične, ne pojavljuju se neki od prstenova a kod heksagonalne neki od šestouglova

Ulazni oblici

Za ovu simulaciju interesantno je više oblika ulaza (vrednosti funkcije za početnu iteraciju) i to:

- Gausian.

$$u(x,y) = Ae^{-\frac{x^2-y^2}{2\sigma^2}} \tag{7.16}$$

- Dipol.

$$u(x,y) = Ae^{-\frac{x^2-(y+d)^2}{2\sigma^2}} - Ae^{-\frac{x^2-(y+d)^2}{2\sigma^2}} \tag{7.17}$$

- Kvadrupol.

$$u(x,y) = Ae^{-\frac{x^2-y^2}{2\sigma^2}} + \tag{7.18}$$

$$Ae^{-\frac{x+d^2-y^2}{2\sigma^2}} + Ae^{-\frac{(x-d)^2-y^2}{2\sigma^2}} +$$
$$Ae^{-\frac{x^2-(y+d)^2}{2\sigma^2}} + Ae^{-\frac{x^2-(y-d)^2}{2\sigma^2}} \tag{7.19}$$

- Vorteks. U formuli koja ga definiše r, φ su sferne koordinate koje odgovaraju Dekartovim x,y.

$$u(x, y) = A(\frac{r}{\sigma})e^{-\frac{r^2}{2\sigma^2}}e^{i\varphi T_c} \tag{7.20}$$

7.2.2 Softverska implementacija

Za najveći broj simulacija interesuje nas efekat promene ulaznih parametara. U našem problemu, ovi parametri su μ, Γ, I_g i u za prvu iteraciju. Softverska implementacija treba da omogući promenu ulaza za ove parametre. Sa druge strane, potrebno je definisati kriterijum za konvergenciju. Prvo ćemo definisati:

$$S1(n) = Sum1 = \int_{-xmax}^{xmax}\int_{-ymax}^{ymax} |u_n(x, y) - u_{n-1}(x, y)|dxdy \tag{7.21}$$
$$S2(n) = Sum2 = \int_{-xmax}^{xmax}\int_{-ymax}^{ymax} |u_n(x, y)|dxdy \tag{7.22}$$

Sada je kriterijum konvergencije sledeći:

$$Sum1 < (Sum2 * \varepsilon) \tag{7.23}$$

gde je ε preciznost potrebna da solitoni budu stabilni. U našim eksperimentima je i ovo bila ulazna vrednost, ali se pokazalo da se stabilnost uspostavlja kada je $\varepsilon = 10^{-15}$.

Drugačiji problem je odlučiti šta će biti smatrano kao divergencija i kako prepoznati takve slučajeve u što je moguće nižem broju iteracija. Kao osnovni kriterijum, definisaćemo maksimalni broj iteracija koji će biti računat (n_{max}), pa ako kriterijum konvergencije (jednakost (7.23)) nije ispunjen to-

likom, ili manjem broju iteracija, taj slučaj će biti smatran divergentnim. U našim eksperimentima koristili smo $n_{max} = 30000$. Važno je razumeti da ovo nije isto što i divergencija u standardnom matematičkom smislu. Mi želimo da pokrenemo simulaciju za veliki broj razlčitih vrednosti ulaznih parametara, a u najvećem broju slučajeva solitoni neće postojati. Drugim rečima, u najvećem broju slučajeva neće nastupiti konvergencija. Ako uspemo da prepoznamo ovakve slučajeve pri malom broju iteracija, ukupno vreme računanja će biti značajno manje. Kriterijumi koji se koriste za prepoznavanje ovih slučajeva su sledeći:

- $(Sum1/Sum2 > 1) \wedge (BrojIteracija > MaxNestabilnihIteracija)$ Ovaj metod za nalaženje solitona mora nakon određenog broja iteracija da se stabilizuje i da počne da se kreće ka rešenju jednačine (7.5). Ako $(Sum1/Sum2 > 1)$ proglasimo za kriterijum nestabilnosti, treba odrediti $MaxNestabilnihIteracija$. Eksperimentalnom metodom je pokazano da je optimalna vrednost za $MaxNestabilnihIteracija = 150$.

- Drugi kriterijum računarske divergencije je prepoznavanje jako sporo konvergirajućih simulacija. Simulacije koje ispunjavaju ovaj kriterijum mogu da budu konvergentne i u klasičnom matematičkom smislu, ali one ne ispunjavaju gore definisani osnovni kriterijum računarske konvergencije.

$$mod(BrojIteracija, TestFrekvencija) = 0 \tag{7.24}$$

$$TestTacnost_i = S1(BrojIteracija)/S2(BrojIteracija) \tag{7.25}$$

$$\frac{TestTacnost_{i-1}}{TestTacnost_i} < MinBrzinaKonvergencije \tag{7.26}$$

$$i = i + 1 \tag{7.27}$$

  Ako su obe jednakosti (7.24,7.26) zadovoljene za neku iteraciju, smatraćemo tu simulaciju jako sporo konvergirajućom. Ideja je proveriti svakih $TestFrekvencija$ (u našoj simulaciji $TestFrekvencija = 600$) iteracija da li se vrednost

$S1(IterationNumber)/S2(IterationNumber)$ promenila za neki minimalni relativni korak $MinBrzinaKonvergencije$ (u našoj simulaciji 1.05).

- Visok nivo nestabilnosti funkcije u, na velikom broju uzastopnih iteracija u nizu će takođe ukazati da je ta simulacija divergentna. Kriterijum za ispitivanje ovog slučaja, koji se proverava prilikom svake iteracije, najlakše se objašnjava sledećim pseudo kodom:

```
if (Sum1/Sum2 > MaxStabilnosti) then
       begin
         IndikatorDiv =  IndikatorDiv +1
          if (IndikatorDiv >MaxPeriodNestabilnosti)
                      divergencija  =true
        end
else
    IndikatorDiv = 0
```

U našoj simulaciji $MaxStabilnosti = 0.05$ a $MaxPeriodNestabilnosti = 650$.

- Kriterijum 'klackalice' se zasniva na analizi ponašanja sledecih funkcija:

$$PravacTacnosti(i) = \frac{S1(i-1)}{S2(i-1)} - \frac{S1(i)}{S2(i)} \quad (7.28)$$

$$Okret(i) = PravacTacnosti(i) * PravacTacnosti(i-1) \quad (7.29)$$

$PravacTacnosti(i) > 0 (< 0)$ nam govori da su u_i, u_{i-1} međusobno bliži (dalji) nego u_{i-1}, u_{i-2} u sledećoj normi :

$$norm(f,g) = \int_{-xmax}^{xmax} \int_{-ymax}^{ymax} |f(x,y) - g(x,y)| dxdy. \quad (7.30)$$

$Okret(i) < 0$ nam govori da je $PravacTacnosti$ promenio znak pri i-toj iteraciji. Ako $PravacTacnosti$ često menja znak (za manje od 35

iteracija) i to se dešava $MinimalanRed$ puta za redom, taj red ćemo nazvati redom klackalice. Neka je $PocetakKlackalice/KrajKlackalice$ budu prva/poslednja iteracija u tom redu, kod koje važi $Switch(i) < 0$.

$$TacnostPocetka = \frac{S1(PocetakKlackalice)}{S2(PocetakKlackalice)} \tag{7.31}$$

$$TacnostKraja = \frac{S1(KrajKlackalice)}{S2(KrajKlackalice)} \tag{7.32}$$

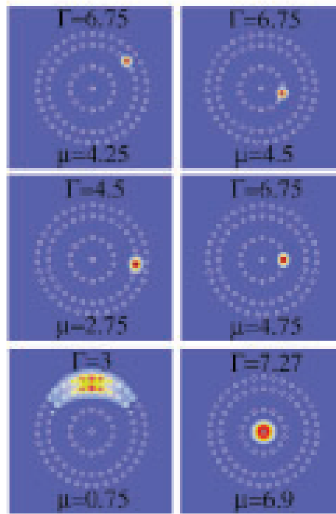
$$\frac{|TacnostPocetka - TacnostKraja|}{TacnostKraja} < BrzinaKlackalice \tag{7.33}$$

Ako je jednakost (7.33) tačna, kažemo da je nastupila divergencija.

Važno je reći da svi ovi kriterijumi mogu biti prestrogi, usled lošeg izbora parametara kojima su definisani, odnosno da mogu davati pogrešne rezultate (t. j. da se neki konvergentni slučajevi smatraju divergentnim). Međutim, ovo nije veliki problem, jer krajnju kontrolu rezultata vrši čovek koji može da ponovi simulaciju u sumnjivim oblastima koristeći slabiji kriterijum.

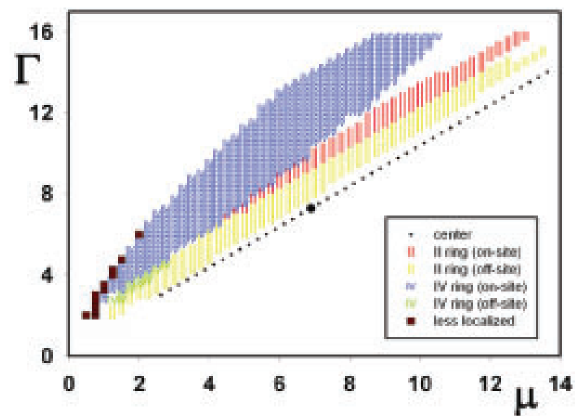
7.3 Rezultati

Simulacije ovog problema su puštane za razne vrednosti ulaznih parametara, a najinteresantniji rezultati su dobijani puštanjem vorteksa (jednacina (7.20)) u cilindričnoj rešetki sa defektima na prvom i trećem prstenu. Za ove fiksirane parametre vršeni su eksperimenti za širok raspon vrednosti μ i Γ. μ je bilo iz intervala $\mu \in (0, 14)$, a $\Gamma \in (0, 16)$ i to sa čvorovima na razmaku od 0.25 i sa manjim razmacima u interesantnijim regionima. Pregledano je približno 5000 tačaka na Pentium4 CPU na 3 Ghz sa 2GB RAM-a. Broj iteracija po jednoj tački je varirao od 150 - 90000 iteracija po tački, odnosno od 50 - 9000 sekundi. Ovde treba skrenuti pažnju na to da se na kritičnim tačkama išlo do 90000 umesto do 30000 iteracija (kako je definisana konvergencijama). Praktičnom upotrebom je pokazano da se snižavanjem kriterijuma divergencije u kritičnim oblastima dobijeni tačni rezultati za te oblasti bez velikog porasta vremena izračunavanja celokupnog ispitivanja.



Slika 7.1: opisi idu sleva na desno od gore na dole IV - ring onsite, II- ring on site , IV - ring off-site, II- ring off-site , less localized, centar, bele oblasti su divergentne tačke

Rezultat ovih izračunavanja je uočavanje raznih oblika dobijenih solitona, kao i oblasti u kojim se oni ne mogu dobiti i zavisnosti osobina solitona od vrednosti parametara μ i Γ. Ova zavisnost se najbolje ilustruje sledećim grafikom i slikom:



7.3.1 Potreba za postojanjem metode za analizu rezultata

Simulacija je puštena za veliki broj različitih vrednosti parametara μ i Γ, pa bi ručno pregledanje rezultata tih simulacija i njihovo razvrstavanje po obliku zahtevalo veliki napor i puno vremena. Sa druge strane, postoji mogućnost da će se potreba za sličnim razvrstavanjem pojaviti i u eksperimentima u kojima se koriste druge vrednosti parametara koji su u ovom slučaju bili fiksirani (kao što su I_g i početna vrednost za u).

Problem prepoznavanja oblika dobijenih solitona nam ukazuje na još jednu potrebnu funkcionalnost *opšteg simulatora*. Svaki modul za simulaciju, osim mogućnosti izračunavanja same simulacije, treba da ima i mogućnost određene analize tih rezultata. Ona može biti i neki kompleksni proračun nad rezultatom, kao u slučaju prepoznavanja oblika. Ovakva analiza rezultata je karakteristična za svaku simulaciju, pa nije moguće definisati neki opšti metod u okviru *opšteg simulatora*, već treba da bude implementirana od strane kreatora plagina. Komunikacija sa ovom metodom će biti ista kao i sa metodom za izračunavanje, pa neće biti bliže opisivana.

7.3.2 Metoda za prepoznavanje oblika solitona

Pregledom nekih rezultata dobijenih puštanjem simulacije u poglavlju Rezultati, prepoznali smo oblike opisane na slici (7.3). Sada treba definisati algoritam uz pomoć kog bi bilo moguće automatizovati proces prepoznavanja tih oblika.

Dobijeni solitoni, odnosno njihovi oblici, mogu imati sledeće karakteristike po kojima ih prepoznajemo:

- (USLOV1) Suma intenziteta svih tačaka solitona (u) je približno jednaka sumi tačaka solitona koje se poklapaju sa rešetkom.
- (USLOV2) Najveći razmak između tačaka sa intenzitetom iznad neke

minimalne vrednosti, koja je definisana u odnosu na maksimalnu vrednost funkcije u, je veći od neke granice.

- (USLOV3) Suma intenziteta svih tačaka solitona približno je jednaka sumi tačaka solitona koje se nalaze na prstenu oko 0., 2. ili 4. kruga.

Na osnovu sledeće tabele možemo odrediti o kojem se solitonu radi:

Tip	USLOV1	USLOV2	USLOV3
II ring on-site	$\top$	$\perp$	2
II ring off-site	$\perp$	$\perp$	2
IV ring on-site	$\top$	$\perp$	4
IV ring off-site	$\perp$	$\perp$	4
center	$\top$	$\perp$	0
less localized	nije bitan	$\top$	nije bitan

Prilikom implementacije ispitivanja oblika solitona na računaru nije nam potrebno direktno ispitivanje USLOVA1 i USLOVA3, već samo odnos odgovarajućih suma. Iz tog razloga definišemo funkcije koja će izračunati sumu svih tačaka koje ispunjavaju neku kombinaciju uslova (opcija za USLOV1, USLOV3).

```
float SumaUslova (float U[DIMX][DIMY],float IG[DIMX][DIMX],
                 float MinIntenziteResetke,bool UnutarResetke,
                 int minRnaKvadrat, int maxRnaKvadrat){

    float       Sum = 0;
    int         i,j;
    float       HelpKrug ;

    for (i=0; i < DIMX; i++)
        for (j=0; j < DIMY; j++)

            if ( ( (Resetka[i][j]>MinIntenziteResetke) == UnutarResetke)){
```

```
                HelpKrug =  pow ( (i*koef + offset),2)   +
                            pow ( (j*koef + offset),2);

                if ( (HelpKrug>minRnaKvadrat) && (HelpKrug<maxRnaKvadrat) )
                        Sum += U[i][j];

            }
    return Sum;
}
```

- U, IG su dvodimenzioni nizovi koji sadrže vrednosti funkcije u, odnosno rešetke I_g u odgovarajućim tačkama.

- $MinIntenziteResetke$. Rešetka I_g nije prosto niz tačaka koje mogu biti u rešetki ili izvan nje, već je to funkcija koja ima svoju vrednost za svaku tačku. Zaključi o karakteristikama oblika solitona su učinjeni na osnovu vizuelizacije rešetke I_g, a ona je urađena tako što su tačke rešetke čije je intenzitet veći od $MinIntenziteResetke$ prikazani kao rešetka.

- $UnutarResetke$ je parametar koji nam govori da li sumiramo tačke koje su unutar ili izvan rešetke.

- $minR$,$maxR$ su mali i veliki poluprečnik prstena po kom sumiramo.

Sam algoritam je sledeći: prolazi se kroz svaku tačku u i proverava se da li je ispunjen uslov pripadanja (nepripadanja) rešetki

$(Resetka[i][j] > MinIntenziteResetke) == UnutarResetke)$

Ako je on ispunjen, proverava se da li se tačka nalazi u odgovarajućem prstenu. Prvo je potrebno indekse dvodimenzionog niza prevesti na koordinate prostora gde se nalazi prsten

$$x = (i * koef + offset)$$

$$y = (j * koef + offset) \tag{7.34}$$

a zatim se na uobičajen način proverava da li je ispunjen uslov da tačka pripada prstenu, odnosno da se nalazi unutar većeg kruga i izvan manjeg, kojima je prsten definisan:

$minRnaKvadrat < x^2 + y^2 < maxRnaKvadrat$

i ako jeste, dodaje se vrednost funkcije u na ukupnu sumu.

Za ispitivanje USLOVA2 potrebno je naći minimalnu ($MinI = min(I)$) i maksimalnu ($MaxI = max(I)$) vrednost za I, takvu da važi uslov:

$$U[I][J] > \frac{Max(U)}{koef}, \quad J \in \{0, .., DIMY - 1\} \tag{7.35}$$

zatim minimalnu ($MinJ = min(J)$) i maksimalnu ($MaxJ = max(J)$) vrednost za J, takvu da važi uslov:

$$U[I][J] > \frac{Max(U)}{koef}, \quad I \in \{0, .., DIMX - 1\} \tag{7.36}$$

Sada se USLOV2 može izraziti kao :

$$(MaxI - MinI)^2 + (MaxJ - MinJ)^2 > MinimalniRaspon^2 \tag{7.37}$$

Funkcija koja ispituje oblik je trivijalna. Ona nizom jednostavnih upita proverava ispunjenost uslova i na osnovu tabele određuje oblik ili pak vraća 'neodređen' ako nije ispunjena neka kombinacija uslova iz tabele. Ovaj neodređen oblik je bitan, jer se u njemu nalaze novi oblici, čije karakteristike prepoznajemo i na osnovu njih dodajemo nove uslove. Sama tabela oblika je postupno konstruisana upravo ovakvim postupkom.

7.3.3 Rezultati optimizacija

Iz ovog grafika se jasno vidi da je između ispitanih tačaka samo $18 - 23\%$ konvergentno, a sa druge strane prosečan broj iteraciaja za ove tačke je oko 5000. Za većinu divergentnih tačaka, osim onih blizu graniče između oblasti konvergencije i divergencije, njihova divergencija je prepoznata za 150 (prvim kriterijumom), ili 600 (drugim kriterijumom) iteracija. Iz ovih podataka se vidi da je upotrebom kriterijuma za rano prepoznavanje divergencija, u poređenju sa korišćenjem isključivo osnovnog kriterijuma konvergencije, ukupno vreme računanja smanjeno približno 20 puta.

Glava 8

Zaključak

Sa porastom moći računara, računarske simulacije su dobile daleko značajniju ulogu u istraživačkim poslovima. Iz istog razloga, došlo je i do relativnog porasta vrednosti ljudskog vremena u odnosu na računarsko. Da bi se ljudsko vreme što efikasnije koristilo, teži se sve većem stepenu automatizacije izvršavanja raznih jednostavnih i repetetivnih poslova. Na prvi pogled, istraživanje raznih sistema uz pomoć simulacija ne deluje kao proces koji je moguće automatizovati, zato što je neophodan čovek da izvede zaključke o ispitivanom sistemu. Ovakvo viđenje je tačno, ali ne u potpunosti. Proces zaključivanja je svojstven čoveku i računar ga tu ne može zameniti, ali u segmentu prikupljanja podataka o pojavi koja nas interesuje, putem pokretanja simulacije, postoji niz operacija koje je moguće automatizovati.

Opšti simulator je upravo to i postigao, tako što je postupak prikupljanja početnih podataka ručnim pokretanjem niza simulacija, gde se zapravo najveći deo vremena provodi u čekanju da računar završi eksperiment, zamenio automatskim pokretanjem eksperimenata tokom noći, čije rezultate čovek može da analizira sledećeg dana. Zahvaljujući pogodno definisanom sistemu automatizacije, čovek može na osnovu zaključaka dobijenih iz početnih rezultata da ponovi isti postupak za celu novu zonu od interesa, umesto ponovnog ručnog i iscrpljujućeg pokretanja pojedinačnih simulacija.

Opšti simulator omogućuje automatizaciju i nekih procesa za koje na izgled ona nije moguća, kao što je nalaženje simulacije sa određenim izlaznim karakteristikama. Ovo je postignuto jednostavnim raščlanjivanjem postupaka koji čovek vrši da bi takvu simulaciju našao. Čovek prvo izvrši neku analizu sistema koji simulira, a na osnovu dobijenih zaključaka nalazi rešenje određenim iterativnim postupkom, najčešće metodom polovljenja. Računar teško može uspešno da izvrši analizu rezultata, ali je moguće isprogramirati određenu iterativnu metodu koja će, pošto covek unese odgovarajuće parametre, doći do rešenja. Upavo na ovakav način je rešen ovaj problem u okviru *opšteg simulatora*.

U savremenom računarskom svetu, prilikom razvoja aplikacija, vodeće tendencije su povećanje modularnosti, odnosno nezavisnosti različitih delova programa među sobom i pojednostavljivanja proširenja. *Opšti simulator* je upravo vođen ovakvim idejama. Automatizacija i vizuelizacija rezultata su kreirani što je moguće nezavisnije od samih simulacija. Posledica ovakvog postupka je da se proširivanje aplikacije, odnosno dodavanje novih simulacija, svodi na kreiranje dll-ova koji ispunjavaju određenu specifikaciju. Kreiranje dll-ova sa novim simulacijama je olakšano postojanjem biblioteka *RCommon.Lib* i *RImage.lib*, koje sadrže niz klasa i funkcija sa gotovom funkcionalnošću, a koje se mogu koristiti po potrebi.

Kreiranje plaginova za *opšti simulator* je ilustrovano na jednostavnom primeru kosog hitca. Pokazano je da je sa vrlo malo dodatnog programiranja omogućena puna analiza problema. Drugi primer proširenja za simulaciju propagacije solitonskih rešenja u fotoničnim rešetkama nam ukazuje na mogućnosti rada i sa mnogo kompleksnijim sistemima. Na ovom primeru je prikazana i pogodnost nezavisnosti *opšteg simulatora* od samih proširenja, što omogućuje kreatoru implementacije niz optimizacija za datu simulaciju.

Korišćenjem *opšteg simulatora* na konkretnim problemu potvrđene su očekivane prednosti ovakvog programa za korišćenje simulacija:

- Jednostavna i efikasna vizuelizacija rezultata, kako u samom simulatoru tako i u pomoćnim programima, bez dodatnog programiranja.
- Razvoj modula za simulaciju je ubrzan, jer se rezultati brzo analiziraju.
- Lako se proširuje rezultat novim informacijama, koje se automatski vizuelizuju.
- Modul za simulaciju je izolovan od ostatka aplikacije, tako da se u njemu mogu primenjivati razni trikovi za povećavanje performansi.
- Neki procesi koji su ranije bili analizirani od strane čoveka su efikasno automatizovani, tako da je upotreba ljudskog vremena postala daleko efikasnija.
- Greške koje su prouzrokovane automatizacijom su lako prepoznate i otklonjene promenom parametara ili promenom delova koda modula za simulaciju.

Rad zasnovan na delovima magistarskog pod nazivom 'Software implementation of Petviashvili's method for finding solutions in photonic lattices', je predstavljen na konferenciji u Mađarskoj. Sam program je trenutno u upotrebi u Institutu za fiziku u Zemunu i za njega su implemetirani dodatni moduli za simulaciju kretanja laserskih zraka kroz fotonične rešetke pomoću brze Furieove transformacije, zatim metodom konačnih razlika (sa fiksnom i promenjivom mrežom). Njegovom upotrebom je omogućeno i objavljivanja niza radova u fizičkim časopisima, čime pokazuje i svoju praktičnu i realnu upotrebnu vrednost.

Bibliografija

[1] An intruduction to computer simulation, M.M. Woolfson and G.J Pert, 1999

[2] The World is a Process, Simulations in Natural and Social Siences, Stephan Hartmann, 1996

[3] Computer Simulation Methods in Theoretical Physics, Dieter W Heermann, 1989

[4] A Guide to Monte-Carlo Simulations in Statistical Physics, David P. Landau and Kurt Binder, 2000

[5] OOP Demystified, James Keogh and Mario Giannini , 2004

[6] Object-Oriented Programming in C++ (4th Edition), Robert Lafore , 2001

[7] Understanding C++ for MFC, Richard F. Raposa, 2001

[8] Objects, Abstraction, Data Structures and Design: Using C++ , Elliot B. Koffman and Paul A. T. Wolfgang , 2005

[9] Introduction to C++ Programming and Graphics, Constantine Pozrikidis, 2007

[10] Mathematical and Computer Programming Techniques for Computer Graphics, Peter Comninos, 2005

[11] Random Number Generation and Monte Carlo Methods (Statistics and Computing), James E. Gentle, 2004

[12] Number Theory for Computing, Song Y. Yan and M.E. Hellmann , 2002

[13] The Essential Guide to User Interface Design: An Introduction to GUI Design Principles and Techniques, Wilbert O. Galitz , 2007

[14] GUI Bloopers 2.0: Common User Interface Design Don'ts and Dos (Morgan Kaufmann Series in Interactive Technologies) , Jeff Johnson , 2007

[15] Working With Microsoft Visual Studio 2005 , Craig Skibo, Marc Young, and Brian Johnson , 2006

[16] Numerical Recipes in C++: The Art of Scientific Computing, William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery ,2002

[17] Numerical Recipes Example Book (C++), William T. Vetterling, William H. Press, Saul A. Teukolsky, and Brian P. Flannery, 2003

[18] Optical Solitons (Academic Press, San Diego, 2003),Y. S. Kivshar and G. P. Agrawal .

[19] Photonic Crystals: Molding the flow of light (Princeton U. Press, 1995),J.D. Joannopoulos, R.D. Meade, and J.N. Winn.

[20] Optimization Theory and Methods: Nonlinear Programming (Springer Optimization and Its Applications) by Wenyu Sun and Ya-xiang Yuan , 2006

[21] Practical Optimization: Algorithms and Engineering Applications by Andreas Antoniou and Wu-Sheng Lu , 2007

[22] 2D Object Detection and Recognition: Models, Algorithms, and Networks by Yali Amit (Hardcover - Nov 1, 2002)

[23] Computer Simulation: APractical Perspective, Roger McHaney, 1991

[24] Object-Oriented Computer Simulation of Discrete-Event Systems, Jerzy Tyszer, 1999

[25] Handbook of Simulation: Principles, Methodology, Advances, Applications, and Practice, Jerry Banks, 1998

[26] C++ for Mathematicians: An Introduction for Students and Professionals,Edward R. Scheinerman, 2006

[27] Simulation, By Shane G. Henderson, Barry L. Nelson, 2006

[28] Modelling, Simulation And Optimization Of Complex Processes: Proceedings Of Complex Processes, Hans Georg Bock, 2005

[29] Computational Physics: Problem Solving with Computers ,Rubin H. Landau, Cristian C. Bordeianu, 2007

[30] Monte Carlo and Quasi-Monte Carlo Methods,By Harald Niederreiter, D Talay 2004

[31] Modelling and Simulation: Exploring Dynamic System Behaviour, Louis G. Birta, Gilbert Arbez, 2007

[32] Microscopic traffic simulations of road networks using high-performance computers, A. Bachem, 1996