1. Įžanga.

Šiuo darbu buvo bandoma patobulinti bei pataisyti klaidas esančias Šitkovskio metodo realizacijoje Java programavimo kalboje. Kadangi buvo žinoma, kad Fortran kalba parašyta realizacija veikia teisingai, nemažai laiko buvo skirta jos analizei. Taip pat buvo bandyta atlikti jos konvertavimą į kitas programavimo kalbas. Buvo pastebėta, kad geriausi konvertavimo įrankiai yra tarp Fortran ir C programavimo kalbų. Plačiau šie įrankiai bus aptariami 2 skyrelyje. Atlikus šių įrankių analizę buvo prieita išvados, kad rezultatai gauti naudojant juos yra blogos kokybės, reikalauja daug pakeitimų, kuriuos reikia atlikti rankiniu būdu.

Dėl šių priežasčių toliau darbe buvo bandoma patobulinti jau egzistuojančią Šitkovskio metodo realizaciją Java programavimo kalboje. Daugiausiai laiko čia buvo sugaišta analizuojant GMJ optimizavimo sistemą. Tai atliekant buvo pasigesta išsamesnio aprašymo ir komentarų programinėse realizacijose. Būtent šią spragą ir bandoma užpildyti 3 skyrelyje. Vėliau bus aptartos papildomai realizuotos klasės ir metodai, kurie palengvina Šitkovskio metodo testavimą ir analizę. Tai yra atlikta 4 skyrelyje. Penktame ir šeštame skyreliuose yra aprašyti autoriaus pastebėjimai, kur yra apdorojami duomenys apie apribojimus ir kur galėtų būti klaidos. Čia taip pat palygintos realizacijos C ir Java kalbomis.

2. Programos konvertavimas iš vienos programavimo kalbos į kitą.

Kaip buvo minėta anksčiau, darbo eigoje buvo išbandyti įvairūs automatinio programų konvertavimo iš vienos kalbos į kitą įrankiai. Be jau minėtų Fortran->C tipo konvertavimo įrankių taip pat buvo aptiktos priemonės, atliekančios Fortran programavimo kalba parašytų programų vertimą į Java išeities tekstą. Tačiau jos pasirodė nepatikimos ir nepasiekusios reikiamo lygio. Viena iš jų buvo f2j [1]. Šiek geresnė situacija yra tarp Fortran->C tipo konvertavimo įrankių. Tačiau čia iškilo problemų, kadangi įvairios Fortran programavimo kalbos versijos pasižymi skirtingomis standartinėmis funkcijomis (ypač duomenų įvedimo, išvedimo į ekraną). Todėl išeities tekstas gautas atlikus automatinį konvertavimą tiek C tiek Java kalbose nesikompiliuoja. Kaip parodė patirtis – pataisyti gautas klaidas reikalauja daug laiko sanaudų. Po konvertavimo gauta programa yra dažniausiai labai sunkiai suprantama, nes skirtingose programavimo kalbose yra leidžiamos skirtingos konstrukcijos, kurias pakeitus programa gali virsti labai griozdiška. Daug problemų taip pat sukelia tai, kad C programavimo kalboje galima naudoti tiek tiesioginius kintamuosius, tiek nuorodas į juos, o Java – tik pastaruosius. Dėl šios priežasties dauguma C->Java tipo konverterių visus paprastus kintamuosius verčia į masyvo tipo kintamuosius. Tai taip pat apsunkina gautų programų suvokimą. Galiausiai net ir gavus teisingai veikiančią programą ją dar tektų adaptuoti GMJ sistemai. Ši užduotis taip pat yra sudėtingesnė nei gali pasirodyti iš pirmo žvilgsnio. Visų pirma dėl to, kad reikia būti gerai susipažinusiam su GMJ sistema (3 skyrelis).

Darbe buvo naudotas f2c [2] Fortran->C tipo konverteris. Kadangi galutinis tikslas buvo gauti realizaciją Java programavimo kalboje, papildomai buvo naudotas C->Java konvertavimo įrankis c2java [3]. Buvo pastebėta, kad pastarojo darbas dažniausiai apsiriboja “goto” tipo konstrukcijų panaikinimu. Tai padaroma paprastų išsišakojimo struktūrų “if-else” bei taip vadinamų “exception” pagalba. Šie pakeitimai taip pat apsunkina programos suvokimą.

Dėl visų aukščiau išvardintų problemų, kurių neišsprendžiamumu buvo įsitikinta darbo metu, buvo nuspręsta pasirinkti kitą kelią, kuris aprašomas toliau.

3. Optimizavimo sistemos (GMJ) bendrosios savybės.

Be pagrindinių GMJ sistemos klasių, kurios nusako kaip atrodys vartotojo sąsaja, parametrų įvedimo langas šiam darbui didžiausios reikšmės turi šios klasės: *AbstractTask, Task, Domain, DomainWithConstraint, AbstractMethod, Method,* MethodBase. Dabar kiekviena iš jų bus aptariama atskirai ir detaliau. *AbstractTask* ir *Task* niekuo nesiskiria išskyrus tai, kad pirmoji yra abstrakti klasė, o antrasis – interfeisas, kurį realizuoja ta klasė. Norint atitinkamai paveldėti arba realizuoti vieną iš jų reikia atkreipti dėmesį į du metodus:

Domain domain();

double f(Point pt);

Pirmasis turi grąžinti darbinę sritį apie kurią plačiau bus kalbama šiek tiek vėliau. Antrajame turi būti realizuota funkcija, kurią reikia minimizuoti. Jei pvz. mums reikia aprašyti tokią funkciją:

$F(X) = X_0^2 + 4(X_1 - 2)(X_1 - 4) + 3;$

Jos realizacija atrodys taip:

```
public double f (Point pt){
        return pt.x[0]*pt.x[0] + (pt.x[1]-2)*(pt.x[1]-4)*4 +3;
}
```

Šis metodas yra dažniausiai kviečiamas daug kartų programos eigoje su skirtingomis *Point pt* kintamojo reikšmėmis. Pastarąjį kintamąjį galima laikyti paprasčiausiu masyvu ar reikšmių vektoriumi. Pagal metodo *f* grąžinamas reikšmes naudojami optimizavimo metodai nustato prie kurių parametro *Point pt* reikšmių yra pasiekiamas funkcijos *f* minimumas.

Kaip minėta anksčiau, *Domain* klasė nusako metodo darbo sritį. *DomainWithConstraint* klasė yra jos specializacija darbui su apribojimais. Pagrindiniai kintamieji abiejose klasėse yra šie:

public double min[];

public double max[];

public Point defaultPoint;

Kintamieji min ir max atitinkamai nusako kintamųjų kitimo minimalias ir maksimalias ribas. Šių masyvų dydis yra lygūs funkcijos *f* kintamųjų skaičiui. Todėl pvz. min[1] = -5, max[1] = 5; reiškia, kad X_1 kintamasis gali kisti diapazone [-5 .. 5]. Kintamasis *defaultPoint* nurodo vektoriaus X reikšmes pradiniu laiko momentu. Iš *Domain* ir *DomainWithConstraint* klasių dar reiktų išskirti šiuos metodus:

public String[] dimensions ();

public double constraintAt (int c, Point pt);

Pirmasis grąžina kintamuosius atitinkančius eilutės tipo pavadinimus. Jie naudojami GMJ sistemoje, tačiau net ir nagrinėjant metodus be šios sistemos buvo pastebėta, kad šis metodas yra vis tiek naudojamas nustatant kintamųjų skaičių.

Antrasis metodas yra labai svarbus nagrinėjant Šitkovskio metodą, kadangi jo pagalba yra nusakomi apribojimai. Pirmasis metodo parametras nurodo, kurį apribojimą reikia apdoroti, o antrasis – reikšmių vektorius, kurio teisingumą duoto apribojimo atžvilgiu reikės negrinėti. Apskritai kalbant apie apribojimus reikia pasakyti, kad jie yra aprašyti neišreikština forma. Jie gali būti nusakomi lygybių ir nelygybių forma. Aukščiau minėtas metodas "neskiria" šių dviejų apribojimų tipų – jis paprasčiausiai grąžina apribojimo reikšmę duotame taške. Šį atskyrimą turi atlikti pats uždavinio sprendimui naudojamas metodas. Paprastai tam yra naudojami du parametrai – m (arba *MParameter*) ir me (arba *MEParameter*). Pirmasis reiškia visų

apribojimų skaičių, o antrasis – lygybės formos apribojimų skaičių. Norint apribojimus realizuoti minėtu *constraintAt* metodu reikia visus apribojimus apsirašyti f(x) = 0 arba f(x) >= 0 forma priklausomai nuo to ar tai lygybių ar nelygybių apribojimas. Dėl pastarosios formos galima padaryti vieną išvadą – konkretus apribojimas *i* yra tenkinamas, jei jį atitinkanti *constraintAt(i,…)* reikšmė yra lygi nuliui (lygybinių apribojimų atveju) arba daugiau/lygi nuliui (nelygybinių apribojimų atveju). Aukščiau minėti parametrai *MParameter* ir *MEParameter* dažnai yra perduodami per vidines klases, pvz. *MethodMProvider* ir *MethodMEProvider*. Tačiau tai turi prasmės tik naudojantis GMJ sistema. Testuojant kurį nors metodą atskirai šiuos parametrus galima priskirti tiesiogiai. Dar keletas logiškų išvadų būtų tokios:

MParameter >= MEParameter;

MParameter – MEParameter = mc;

Čia mc – nelygibinių apribojimų skaičius.

Eksperimentais buvo nustatyta, kad neįvykdžius pirmojo apribojimo metodai paprasčiausiai ignoruoja *MEParameter* blogą reikšmę – ją priskiria *MParameter* parametro reikšmei.

Dabar aptarkime klasių *AbstractMethod*, *MethodBase* bei interfeiso *Method* savybes. Visų pirma reikia pasakyti, kad nagrinėjant Šitkovskio metodo veikimą nereikia išplėsti (extend) ar realizuoti nei vienos iš minėtų klasių ar interfeiso, kadangi pati pagrindinė Šitkovskio metodo klasė netiesiogiai realizuoja *Method* interfeisą. Visos aukščiau minėtos klasės bei interfeisas turi kelis svarbius jau aptartus parametrus (*MParameter* ir *MEParameter*) bei du metodus:

int iterations ();

Result run (ResultLogger l, Task t);

Pirmasis metodas grąžina metode naudojamų iteracijų skaičių. Šitkovskio metode šis skaičius yra lygus 40. Reikšmė pagal nutylėjimą yra nurodoma klasėje *MethodBase* ir yra lygi 1000. Tačiau dažniausiai optimizavimo metodams užtenka žymiai mažiau iteracijų norint gauti atsakymą su pageidaujamu tikslumu. Šis skaičius paprasčiausiai reiškia maksimaliai leistiną iteracijų skaičių. Dažniausiai jis užtikrina metodo darbo pabaigą, jei jis diverguoja.

Antroji funkcija pradeda pačio optimizavimui naudojamo metodo (šiuo atveju – Šitkovskio) darbą. Kaip galima spėti iš pavadinimo ši funkcija iškviečiama kaip atskiras procesas, kuris dirba lygiagrečiai su pagrindine programa. *run* funkcijai yra paduodami du kintamieji:

ResultLogger l – rezultatų registravimo kintamasis. Jo dėka galima fiksuoti metodo rezultatus kiekvienoje iteracijoje;

Task t – sprendžiamo uždavinio kintamasis. Ši klasė jau buvo aptarta anksčiau.

4. Sukurtos pagalbinės klasės.

Testuojant ir analizuojant Šitkovskio metodo versiją buvo realizuotos šios pagalbinės klasės: *MyDomain*, *TestTask* ir *TestShitkowky*. Visų jų realizacija yra pateikta priede A. Testuojant buvo pasirinkta dviejų kintamųjų funkcija. Todėl *MyDomain* klasėje reikia nurodyti tris kintamųjų kitimo rėžius (nes vienas yra taip vadinamas pseudo kintamasis arba laisvasis narys). Tai galima pastebėti realizacijoje, kuri yra pateikta priede A. Toje pat klasėje realizuotas ir apribojimus nusakantis metodas *constraintAt*. Jei šis metodas turi grąžinti reikšmę apribojimui *i*, kuris nėra apibrėžtas, metodas parasčiausiai turi grąžinti bet kurį teigiamą skaičių – šioje realizacijoje tai yra 1. Kiti apribojimai yra nustatomi "*switch – case*" arba vieno "*if*" tipo struktūromis.

Kita svarbi papildomai realizuota klasė yra *TestTask*. Joje svarbiausi yra du metodai:

public Domain domain ()

public double f (Point pt)

Pirmasis grąžina jau aptartos klasės *Domain* tipo kintamąjį. Antroji funkcija realizuoja pačią užduotį, kurią reikia optimizuoti. Ji grąžina tos užduoties reikšmę naudojant duotą reikšmių vektorių *Point pt*. Viso optimizavimo tikslas rasti tokį reikšmių vektorių, kuris kaip įmanoma geriau tenkintų keliamus apribojimus ir prie kurio optimizuojama funkcija *f* įgytų minimalią reikšmę.

Paskutinė papildomai sukurta klasė yra *TestShitkowsky*. Jos realizacija yra labai paprasta ir turi vienintelį *main* metodą, kuris yra visų testavimui sukurtų klasių vykdymo "pradžios taškas". Būtent šioje funkcijoje yra nurodoma, kad sprendžiamas uždavinys bus iš *TestTask* klasės, jis bus sprendžiamas naudojant Šitkovskio metodą. Čia taip pat pradedamas vykdyti ir pats Šitkovskio metodas išsaugant gautus rezultatus. Šios kaip ir kitų dviejų papildomai sukurtų klasių realizacija yra pateikte priede A.

5. Šitkovskio metodo Java kalboje ypatumai.

Šis metodas priskiriamas lokalaus optimizavimo metodų grupei. Viena iš sudėtinių jo dalių yra gradientinio mažėjimo algoritmas. Šitkovskio metodo išskirtinis bruožas yra tai, kad jis naudojant Lagranžo daugiklį sugeba rasti lokalų minimumą atsižvelgiant į apibrėžtus apribojimus. Būtent jiems ir bus toliau skiriamas didžiausias dėmesis.

Apribojimai gali būti nusakomi lygybių arba nelygybių forma. Jiems apibrėžti yra naudojamas tas pats *DomainWithConstraint* (naudojant papildomai sukurtas klases – *MyDomain*) klasės metodas *constraintAt*. Plačiau apie tai yra aprašyta 3 skyrelyje. Paanalizavus Šitkovskio metodo realizaciją Java kalba buvo pastebėta, kad šis apribojimų metodas iškviečiamas tik *func_* funkcijoje. Tai reiškia, kad ši funkcija yra labai svarbi tiriant apribojimų veikimą. *func_* yra iškviečiama keturiose Šitkovskio metodo vietose. Buvo pastebėta, kad jau po pirmo iškvietimo gautas reikšmių vektorius (kuris pasibaigus algoritmo darbui yra ir galutinis rezultatas) yra artimas galutiniam rezultatai. Vėlesnėje algoritmo darbo eigoje šios reikšmės kinta labai nedaug, todėl didžiausią dėmesį reiktų kreipti į algoritmo darbo pradžią. Reikia pastebėti, kad Šitkovskio metodo realizacija Java kalboje faktiškai ignoruoja apribojimus ir tai išryškėja taip pat algoritmo darbo pradžioje.

Toliau nagrinėjant apribojimus taip pat reiktų paminėti keletą svarbių globalių kintamųjų: *g* ir a*ctive*. Pirmasis yra slankaus kablelio tipo masyvas (double[]), antrasis – loginio tipo kintamųjų masyvas (boolean[]). *g* kintamasis yra apribojimų funkcijos reikšmės esant konkrečiam reikšmių vektoriui. Jei visi apribojimai yra tenkinami, šio masyvo visos reikšmės turi būti neneigiamos. Nagrinėjant apribojimus Šitkovskio metode, kuris realizuotas Java kalba visas šio kintamojo panaudojimo vietas reikia gerai išanalizuoti.

6. GMC sistemos savybės bei panaudojimas.

Optimizavimo metodai C kalba yra realizuoti GMC sistemoje. Egzistuoja dvi šios sistemos versijos. Viena iš jų yra pritaikyta LINUX operacinei sistemai, kita – UNIX-SOLARIS. Autoriui pavyko testuoti tik viena iš jų – GMC-LINUX. Taip pat buvo pastebėta, kad veiksmų seka yra labai svarbi atliekant užduotį. Todėl ji bus pateikta detaliau:

1. Susikurti direktoriją xxx jau egzistuojančios gmc.linux direktorijos viduje.

2. Nukopijuoti į ją instaliacinį failą gmclinux.tgz.
3. Išarchyvuoti šį failą toje pat direktorijoje naudojant komdą “'tar -zxf gmclinux.tgz”.
4. Atlikti norimus pakeitimus shit.c ar kitame faile (pvz. įdėti tarpinių spausdinimų).
5. Sukurti vykdomąjį failą komandinėje eilutėje parašius “make” komandą.
6. GMC sistemą paleisti komanda “test”.
7. Išsirinkti lokalaus tipo metodą “Nlp”, kuris užtikrina Šitkovskio metodo veikimą.
8. “Parameters” meniu punkte pasirinkti norimus kintamųjų kitimo rėžius bei apribojimų skaičių bei tipą (parametrai *M* ir *ME*).
9. Pasirinkti rezultatų išvedimo formą meniu punkte “Results”.
10. Pasirinkti meniu punktą Operation->Run.
11. Rezultatus stebėti specialiame rezultatų lange ir ekrane, jei yra atliekami tarpiniai spausdinimai.

Buvo pastebėta, kad nesilaikant šių nurodymų galima susidurti su tam tikromis problemomis:

- Nukopijavus failą(us) ne į tam pačiam vartotojui priklausančią direktoriją “make” komanda automatiškai neranda kompiliavimo įrankio.
- Pabandžius naudoti egziztuojančius *Makefile* failus buvo gautos įvairios klaidos (kompiliatorius “nerado” Client.c faile naudojamų kintamųjų).
- Naudojant kitą Linux sistemą nei esančią “diedas.soften.ktu.lt” iškildavo problemų ne tik kompiliavimo bet ir jau sukurto vykdomojo failo “test” paleidimo metu dėl įvairių bibliotekų nesuderinamumo.

Visų šių problemų galima išvengti prisilaikant aukščiau išvardintos veiksmų sekos.

Kalbant apie GMC reikia pabrėžti, kad tiek optimizuojamas uždavinys, tiek apribojimai yra atitinkamai realizuojami šiuose fi.c failo metoduose:

*double fi (const double *x, int n);*

*void constr (const double *x, int n, double *g, int ng);*

Jų sintaksė ir realizacijos idėjos labai panašios į jau aprašytų ir naudojamų metodų GMJ sistemoje.

7. Šitkovskio metodo C kalboje ypatumai.

Šitkovskio metodas GMC sistemoje yra realizuotas failuose *shit.c, Shit.c, Shit.h, shit.h*. Pagrindinis iš jų yra pirmasis. Lyginant jį ir realizaciją Java kalba, skirtumus sunku įžiūrėti dėl didelio šių kalbų ir ypač naudojamo interfeiso skirtumų. Nepaisant to, abiejose realizacijose galima aptikti svarbiausius metodus nagrinėjant apribojimus: *func_(), grad_()* ir kitus. Jų pačių realizacijose ypatingų skirtumų nepastebėta, tačiau daug daugiau informacijos buvo gauta bandant spręsti tą patį uždavinį su abiejomis realizacijomis bei stebint tarpinių spausdinių rezultatus. Buvo pastebėti tokie dėsningumai:

- Tiek C, tiek Java versijose *func_()* metodas iškviečiamas keturis kartus, du iš jų – metode *grad_()*.
- Šitkovskio metodas C kalba ribojimus tikrina daug daugiau kartų nei realizacija Java kalba.
- Turbūt dėl aukščiau paminėto pastebėjimo versija C kalba ribojimus traktuoja geriau nei realizacija Java kalba.

- C realizacijoje metodas *func_()* iškviečiamas daug kartų (apie 50 ir daugiau). Daugiausiai iš šių iškvietimų yra antroje metodo *func_()* panaudojimo vietoje.
- C realizacijoje reikšmės kinta didesniame diapazone ir daugiau nei Java realizacijoje.

Metodo *func_()* panaudojimo atvejai C ir Java kalbomis yra pateikti priede B.

8. Šitkovskio metodo realizacijų C ir Java kalbomis veikimo palyginimas.

Šio darbo metu buvi įsitikinta, kad daugiausiai informacijos apie šių realizacijų skirtumus galima gauti ne atliekant išsamų išeities teksto lyginimą, o stebint tarpinius rezultatus su įvairiais testiniais duomenimis. Žemiau pateiktose lentelėse yra surašyti stebėjimo duomenys:

Lentelė 1. Realizacijos C kalba rezultatai, kai naudojama pirma funkcija ir pirmos grupės ribojimai.

Apribojimų parametrai	X_0	X_1	Y	Iteracijų skaičius	Pirmasis *func_()* kvietimas	Antrasis *func_()* kvietimas	Trečiasis *func_()* kvietimas	Ketvirtasis *func_()* kvietimas
M=2; ME=1;	-0,99	1	2,49	6	1	107	12	24
M=2; ME=0;	1	1	0,5	7	1	109	14	28
M=1; ME=0;	1	1	0,5	2	1	1	4	4

Lentelė 2. Realizacijos C kalba rezultatai, kai naudojama antra funkcija ir pirmos grupės ribojimai.

Apribojimų parametrai	X_0	X_1	Y	Iteracijų skaičius	Pirmasis *func_()* kvietimas	Antrasis *func_()* kvietimas	Trečiasis *func_()* kvietimas	Ketvirtasis *func_()* kvietimas
M=2; ME=1;	0,18	1,02	14,68	7	1	111	14	28
M=2; ME=0;	-0,2	3,38	-0,38	8	1	257	11	26
M=1; ME=0;	2,04	3,59	4,52	8	1	170	16	16

Lentelė 3. Realizacijos C kalba rezultatai, kai naudojama antra funkcija ir antros grupės ribojimai.

Apribojimų parametrai	X_0	X_1	Y	Iteracijų skaičius	Pirmasis *func_()* kvietimas	Antrasis *func_()* kvietimas	Trečiasis *func_()* kvietimas	Ketvirtasis *func_()* kvietimas
M=2; ME=1;	-0,5	1,36	10,06	4	1	7	8	16
M=2; ME=0;	0,65	3,07	-0,56	5	1	108	10	12
M=1; ME=0;	0	3	-1	5	1	5	10	2

Lentelė 4. Realizacijos Java kalba rezultatai, kai naudojama pirma funkcija ir pirmos grupės ribojimai.

Apribojimų parametrai	X_0	X_1	Y	Pirmasis *func_()* kvietimas	Antrasis *func_()* kvietimas	Trečiasis *func_()* kvietimas	Ketvirtasis *func_()* kvietimas
M=2; ME=1;	0	0	0,5	1	0	3	6
M=2; ME=0;	0	0	0,5	1	0	3	6
M=1; ME=0;	0	0	0,5	1	0	3	3

Lentelė 5. Realizacijos Java kalba rezultatai, kai naudojama antra funkcija ir pirmos grupės ribojimai.

Apribojimų parametrai	X_0	X_1	Y	Pirmasis *func_()* kvietimas	Antrasis *func_()* kvietimas	Trečiasis *func_()* kvietimas	Ketvirtasis *func_()* kvietimas
M=2; ME=1;	0	0	35	1	0	3	6
M=2; ME=0;	0	0	35	1	0	3	6
M=1; ME=0;	0	0	35	1	0	3	3

Lentelė 6. Realizacijos Java kalba rezultatai, kai naudojama antra funkcija ir antros grupės ribojimai.

Apribojimų parametrai	X_0	X_1	Y	Pirmasis *func_()* kvietimas	Antrasis *func_()* kvietimas	Trečiasis *func_()* kvietimas	Ketvirtasis *func_()* kvietimas
M=2; ME=1;	0	0	35	1	0	3	6
M=2; ME=0;	0	0	35	1	0	3	6
M=1; ME=0;	-2,4	-2,4	6,2	1	2	6	6

Lentelėse yra pateikti duomenys, kai pirmoji funkcija aprašoma taip:
$Y = (X_0 - 0.5)^2 + (X_1 - 0.5)^2$;
Antroji funkcija yra pateikta trečio skyrelio pradžioje. Pirmos grupės ribojimai aprašomi tokiomis lygtimis:
$F_0(X) = X_0^2 + X_1^2 - 1$;
$F_1(X) = X_0^2 + X_1^2 - 2$;
$F_2(X) = X_0^2 + X_1^2 - 4$;
Antrosios grupės ribojimai atrodo taip:
$F_0(X) = X_0 + X_1$;
$F_1(X) = X_0 X_1 - 2$;
$F_2(X) = 2X_0 + X_1 - 1$;

Parametrai M ir ME atitinkamai reiškia visų ribojimų ir ribojimų lygybių forma skaičių. Kaip matome šiuo atveju tretieji ribojimai yra faktiškai ignoruojami. Funkcijos *func_()* kvietimų skaičius yra sudedamas kiekvienam jo pasitaikymo atvejui atskirai. Visi jie yra pateikti priede B. Paanalizavus šias lenteles galima padaryti keletą svarbių išvadų:

1. C realizacijoje aukščiau minėtas metodas yra kviečiamas daug daugiau kartų nei Java realizacijoje.
2. Tiek C, tiek Java realizacijoje *func_()* metodas pirmoje pozicijoje visada yra kviečiamas vieną kartą.
3. Rezultatai gauti Java realizacijoje dažniausiai visiškai nepriklauso nuo ribojimų, kai tuo tarpu C realizacijoje – skirtumas yra sąlyginai nemažas.
4. Mažiausiai iteracijų ir tuo pačiu metodo *func_()* kvietimo atvejų pasitaiko tada, kai yra mažiau ribojimų (šiuo atveju vienas nelygybinis ribojimas).
5. Didžiausi skirtumai tarp C ir Java realizacijų išryškėja vykdant antrąjį *func_()* metodo kvietimą. Čia ir reikėtų pradėti ieškoti galimų klaidų.
6. Analizuojant tarpinius duomenis po kiekvienos iteracijos abiejose realizacijose paaiškėjo, kad C versijoje atsakymo reikšmė daug kartų ir ženkliai kinta. Tuo tarpu Java realizacijoje atsakymas susiformuoja jau pačioje darbo pradžioje ir beveik nekinta.

7. Dėl pastebėjimų praeitame punkte bei 6 lentelės rezultatų galima daryti išvadą, kad Java versijoje neteisingai dirba ne tik apribojimų apdorojimas bet kartais ir pats metodas.

9. Sukurtų papildomų klasių panaudojimas.

Norint panaudoti papildomai sukurtas Java klases reikia atlikti tokius veiksmus:

1. Nusikopijuoti į direktoriją xxx GMJ sistemos išeities tekstus ir juos sukompiliuoti su komanda *javac* arba į specialą *xxx/jre/lib/ext* direktoriją nukopijuoti GMJ sistemą atitinkančius Java archyvus *jar*. Antruoju atveju direktorija xxx yra standartinė direktorija, kur yra suinstaliuoti Java programavimo kalbos įrankiai.
2. Tris pagalbines klases *MyDomain, TestTask* ir *TestShitkowky* nukopijuoti į pasirinktą direktoriją yyy.
3. Jei buvo nukopijuoti šių klasių išeities tekstai, juos reikia sukompiliuoti naudojant standartinę komandą *javac*.
4. Komandinėje eilutėje parašyti "*java TestShitkowsky*" ir rezultatus stebėti ekrane arba faile, jei išvedimas nukreiptas į jį.
5. Atlikti norimus pakeitimus vienoj iš šių (tikėtina, kad *MyDomain arba TestTask) arba pačioje Shitk* klasėje.

Visus šiuos veiksmus kartoti tol, kol gaunamas norimas rezultatas.

10. Papildoma informacija

Daugiau informacijos, taip pat ataskaitoje paminėtus failus galima rasti adresu [4].

Priedas A.

TestTask realizacija:

```
// Task.java
import lt.ktu.gmj.propertySheet.*;
import lt.ktu.gmj.*;

public class TestTask extends AbstractTask {
MyDomain domain=new MyDomain();

 public double multiplier=1; //Specialus daugiklis

// Metodas skirtas GMJ sistemai
 public void customize (PropertyManager manager)
 {
   String []functions={
    "(sum_i(x[i]-0.5)**2)/ n"
   };

   manager.add ( new DoubleProperty("Multiplier",
                new FieldProvider(this, "multiplier"), 0.1, 1000));
   manager.add ( new ChoiceProperty("Function",
                new FieldProvider(this, "function"),
                functions ));
 }

// Duomenu sriti grazinantis metodas
 public Domain domain ()
 { return domain; }
```

```
// Optimizuojama funkcija f
 public double f (Point pt)
 {
            double y = 0;
       y = pt.x[0]*pt.x[0]*multiplier + (pt.x[1]-2)*(pt.x[1]-4)*4*multiplier+3;
       return y;
 }

};
```

MyDomain realizacija:

```
// Domain.java

import lt.ktu.gmj.propertySheet.*;
import lt.ktu.gmj.*;
import lt.monarch.function.*;
import lt.monarch.function.parser.*;


public class MyDomain extends DomainWithConstraint{

public MyDomain (){
try{
// Duomenu vektoriaus reiksmiu kitimo ribos
min[0]=-10;
max[0]=10;
min[1]=-10;
max[1]=10;
min[2]=-10;
max[2]=10;
// Duomenu vektoriaus pradines reiksmes
defaultPoint.x[0]=0;
defaultPoint.x[1]=0;
defaultPoint.x[2]=0;
// Apribojimu masyvas
constraint=new Function[3];
}
catch(Exception e){e.printStackTrace();}
}
// Duomenu pavadinimu masyvas - jo ilgis naudojamas nustatant kintamuju ir
// duomenu srities apribojimu skaiciu
 static final String []dimensions={
  "X Quadratic Argument",
  "Y Quadratic Argument",
  "Dummy"
 };


// Grazina auksciau aprasyta masyva
 public String[] dimensions ()
 { return dimensions; }

// Apribojimu realizacija
 public double constraintAt (int c, Point pt)
 {
 // Pirmas apribojimas
   if(c==0)
    return pt.x[0]+pt.x[1];
 // Antrasis apribojimas
   if(c==1)
```

```
    return pt.x[0]*pt.x[1]-2;
 // Treciasis apribojimas
   if(c==2)
    return 2*pt.x[0]+pt.x[1]-1;
 // Grazinama reiksme jei tokio apribojimo nera
   return 1;
 }
};
```

TestShitkowsky realizacija:

```
import java.applet.Applet.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.io.PrintStream;
import java.util.Hashtable;
import javax.swing.*;
import lt.ktu.gmj.*;
import lt.ktu.gmj.utils.*;
import lt.ktu.gmj.propertySheet.Header;
import lt.ktu.gmj.propertySheet.PropertySheet;
import java.awt.event.*;
import lt.ktu.gmj.methods.*;

public class TestShitkowsky {
// Visu testavimu "pradzios taskas"
    public static void main(String args[])
    {
// Sprendiama uzduotis
    TestTask tt = new TestTask();
// Rezultatu registravimo kintamasis
    MyResultLogger rs = new MyResultLogger();
// Sitkovskio metodo inicializavimas
    Shitk sh = new Shitk();
// Visu apribojimu skaicius
    sh.MParameter=2;
// Lygibemis isreikstu apribojimu skaicius
    sh.MEParameter=1;
// Sitkovskio metodo veikimo pradzia
    Result res = sh.run(rs,tt);
//Rezultato isvedimas i ekrana
    System.out.println(" Point " + res.point.x[0]+", "+res.point.x[1]+", "+res.point.x[2]);
    }

}
```

Priedas B.

Metodo *func_()* panaudojimas Šitkovskio metodo realizacijoje C kalba

```
…
/* Pirmasis metodo func_() panaudojimo atvejis */
L51: // 479 eilutė
    func_(m, &max_, n, f, &g[1], &x[1], function, constraints);
    grad_(m, mmax, n, f, &g[1], &df[1], &dg[dg_offset], &x[1], &active[1],
      function, constraints);
…
/* Antrasis metodo func_() panaudojimo atvejis */
L197: // 1096 eilute
```

```
/*   NEW FUNCTION AND GRADIENT VALUES */

   func_(m, mmax, n, f, &g[1], &x[1], function, constraints);
…
/* Trečiasis metodo func_() panaudojimo atvejis */
/* Computing MAX */ // 1567eilute
          d__2 = on, d__3 = (d__1 = x[i], fabs (d__1));
          xeps = eps * MAX (d__2,d__3);
          xepsi = on / xeps;
          x[i] += xeps;
          func_(&c__0, mmax, n, &feps, &g[1], &x[1], function, constraints);
…
/* Ketvirtasis metodo func_() panaudojimo atvejis */
          for (j = 1; j <= i__2; ++j) { // 1578 eilute
             if (! active[j]) {
                   goto L2;
             }
             i__3 = -j;
             func_(&i__3, mmax, n, &gjeps, &g[1], &x[1], function, constraints);
             dg[j + i * dg_dim1] = (gjeps - g[j]) * xepsi;
L2:
             ;
          }
…
```

Metodo *func_()* panaudojimas Šitkovskio metodo realizacijoje Java kalba

```
…
/*L51: CR_MM*/ // 291 eilute
                    func_( m, max_, g, result, task );
                    grad_( m, mmax, result, g, df, dg, active, task );
…
/*L197: CR_MM*/          L197 = TRUE;  //732 eilute
          /* NEW FUNCTION AND GRADIENT VALUES */
                                   func_( m, mmax, g, result, task );
…
                              temp = result.value; func_( c__0, mmax, g, result, task );  // 1033 eilute
                              feps = result.value; result.value = temp;
…
// 1043 eilute
                                                  temp  = result.value; func_( i__3, mmax, g, result, task );
                                                  gjeps = result.value; result.value = temp;
…
```

Literatūra

[1] Fortran to Java converter. http://www.npac.syr.edu/projects/pcrc/f2j.html.
[2] Fortran to C convertyer. http://netlib.bell-labs.com/netlib/f2c/index.html
[3] C to Java converter. http://www.soften.ktu.lt/~stonis/c2java/ index.html
[4] Šitkovskio metodo analizės darbo aprašymas. http://www.lksoft.lt/~giedrius