VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

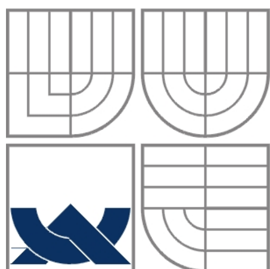
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

NUMERICKÉ METODY PRO SIMLIB/C++

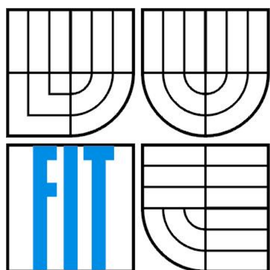
BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE ZBYŠEK NĚMEC
AUTHOR

BRNO 2008



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

NUMERICKÉ METODY PRO SIMLIB/C++

NUMERICAL METHODS FOR SIMLIB/C++

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE ZBYŠEK NĚMEC
AUTHOR

VEDOUCÍ PRÁCE Dr. Ing. PETR PERINGER
SUPERVISOR

BRNO 2008

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů　　　Akademický rok 2007/2008

Zadání bakalářské práce

Řešitel: **Němec Zbyšek**
Obor: Informační technologie
Téma: **Numerické metody pro SIMLIB/C++**
Kategorie: Modelování a simulace

Pokyny:

1. Prostudujte simulační knihovnu SIMLIB/C++ a metody implementace numerických metod.
2. Na základě existující implementace vytvořte nový návrh podsystému numerických integračních metod včetně sady testů.
3. Implementujte navržený podsystém v C++.
4. Implementovaný podsystém řádně otestujte a zhodnoťte přínos práce z hlediska efektivity a korektnosti kódu.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění prvních dvou bodů zadání

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Peringer Petr, Dr. Ing.**, UITS FIT VUT
Datum zadání: 1. listopadu 2007
Datum odevzdání: 14. května 2008

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

LICENČNÍ SMLOUVA
POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami

1. Pan

Jméno a příjmení: **Zbyšek Němec**
Id studenta: 79306
Bytem: U Včelína 1190, 696 02 Ratíškovice
Narozen: 23. 05. 1986, Hodonín
(dále jen "autor")

a

2. Vysoké učení technické v Brně

Fakulta informačních technologií
se sídlem Božetěchova 2/1, 612 66 Brno, IČO 00216305
jejímž jménem jedná na základě písemného pověření děkanem fakulty:

....
(dále jen "nabyvatel")

Článek 1
Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
   bakalářská práce

Název VŠKP: Numerické metody pro SIMLIB/C++
Vedoucí/školitel VŠKP: Peringer Petr, Dr. Ing.
Ústav: Ústav inteligentních systémů
Datum obhajoby VŠKP: ....

VŠKP odevzdal autor nabyvateli v:
tištěné formě počet exemplářů: 1
elektronické formě počet exemplářů: 2 (1 ve skladu dokumentů, 1 na CD)

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.
3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

Článek 2
Udělení licenčního oprávnění

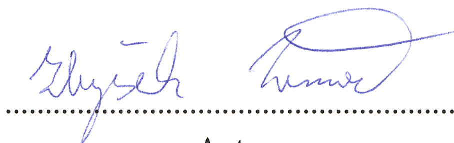
1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizování výpisů, opisů a rozmnoženin.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.
3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti:
   - ☐ ihned po uzavření této smlouvy
   - ☐ 1 rok po uzavření této smlouvy
   - ☐ 3 roky po uzavření této smlouvy
   - ☐ 5 let po uzavření této smlouvy
   - ☐ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací)
4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/ 1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

Článek 3
Závěrečná ustanovení

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: ....



....	....
Nabyvatel	Autor

Abstrakt

Ve své práci se zabývám problematikou užití numerických metod a jejich implementace v objektově orientované simulační knihovně SIMLIB/C++. Navrhnul a realizoval jsem úpravu rozhraní a podsystému numerických integračních metod knihovny SIMLIB s cílem umožnit její snadnější rozšiřitelnost o externí integrační metody. Díky tomu jsem mohl simulační knihovnu SIMLIB obohatit o sadu nových metod z knihovny GSL(GNU Scientific Library) a některé zajímavé metody v jazyce Fortran uvedené v databázi Netlib. Nové i existující metody jsem řádně otestoval a porovnal jejich vlastnosti z hlediska efektivity, stability a přesnosti.

Klíčová slova

modelování a simulace, SIMLIB/C++, numerické integrační metody, GSL, NetLib

Abstract

In my thesis I deal with desing and implementation of numerical integration methods in object-oriented simulation library SIMLIB/C++. I have proposed and implemented modification of application interface and subsystem of numerical integration methods in SIMLIB library to allow easier extension with external methods. I also added a set of new external methods from GSL(GNU Scientific Library) and some of the interesting methods written in Fortran language from Netlib repository into SIMLIB. I have tested the new and the existing methods and I have compared their properties from the viewpoint of efficiency, stability and accuracy.

Keywords

modelling and simulation, SIMLIB/C++, numerical integration methods, GSL, NetLib

Citace

Zbyšek Němec: Numerické metody pro SIMLIB/C++, bakalářská práce, Brno, FIT VUT v Brně, 2008

Numerické metody pro SIMLIB/C++

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Dr. Ing. Petra Peringera. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Zbyšek Němec
9.5.2008

Poděkování

Na tomto místě bych chtěl poděkovat vedoucímu mojí práce Dr. Ing. Petru Peringerovi za podporu a inspiraci při tvorbě této práce a velké množství času, které mi věnoval při pravidelných konzultacích a debatách.

© Zbyšek Němec, 2008.

Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Obsah....1
1 Úvod....3
2 Numerická integrace....4
2.1 Numerické řešení diferenciálních rovnic....4
2.1.1 Princip a klasifikace numerických metod....4
2.1.2 Jednokrokové metody....4
2.1.3 Vícekrokové metody....6
2.1.4 Tuhé systémy....7
2.1.5 Vlastnosti numerických metod a výběr optimální metody....7
3 Návrh podsystému num. integrace....10
3.1 Stávající podsystém num. integrace v SIMLIB....10
3.1.1 Třídy numerické integace....10
3.1.2 Uživatelské rozhraní numerické integace....12
3.2 Úprava stávajícího podsystému....14
3.2.1 Rozhraní numerické integrace....14
3.2.2 Třída IntegratorValues....15
3.3 Zařazení externích metod v novém podsystému....18
3.3.1 Třída ExternalMethod....19
3.3.2 Třída GslMethod a její potomci....19
3.3.3 Třída FortranMethod a její potomci....19
3.4 Výsledný podsystém numerické integrace....20
4 Implementace....22
4.1 Numerická integace v knihovně GSL....22
4.2 Externí metody v jazyce Fortran....23
4.2.1 Smíšené programování C++ a Fortran....23
4.2.2 Vlastnosti implementovaných method v jazyce Fortran....24
4.3 Zhodnocení výsledné implementace....26
5 Testování....27
5.1 Kruhový test....27
5.2 Van der Polův oscilátor....28
5.3 Kmitání struny....29
6 Závěr....31
Literatura....32

Seznam příloh...33
Příloha 1: Překlad a instalace SIMLIB...34

1 Úvod

Základním prostředkem pro získávání znalostí o chování reálného světa kolem nás je experimentování. Člověk provádí různé pokusy od počátku věků a většina znalostí získaných touto formou tvoří základy přírodních věd. Jsou ale také oblasti, kdy by vytvoření samotného modelu bylo příliš nákladné nebo nebylo vůbec možné. Představme si například praktickou realizaci srážky dvou hvězd ve vesmíru. V tento okamžik vstupují na scénu prostředky pro modelování a simulaci, která probíhá na počítači. Tento přístup má značnou výhodu v tom, že je mnohem levnější a rychlejší než reálné experimentování. Pouhou změnou parametrů simulace můžeme vyzkoušet tisíce variant v nesrovnatelně menším čase bez potřeby specializovaného vybavení.

Zástupcem jednoduchého simulačního softwaru je i knihovna SIMLIB/C++[1], které se bude týkat tato práce. Budu se orientovat především na simulaci fyzikálních a chemických dějů, jejichž vlastnosti se mění průběžně s časem. Takové systémy můžeme označit jako spojité. V jednotlivých kapitolách se pokusím vysvětlit, jakým způsobem jsou tyto systémy popsány a co je nezbytné k jejich simulaci.

Cílem této práce je upravit část knihovny SIMLIB tak, aby do ní bylo možné přidat externí integrační metody podílející se na výpočtu nového stavu modelu při spojité simulaci.

V kapitole 2 se seznámíme s teoretickými znalostmi z oblasti numerické integrace, která se používá při simulaci spojitých systémů popsaných diferenciálními rovnicemi. Dále získáme představu o důležitých vlastnostech těchto metod, se kterými musíme při jejich použití počítat.

Kapitola 3 představuje návrh nového podsystému numerické integrace knihovny SIMLIB. Obsahuje popis původního podsystému, prostředků, které poskytoval, a změny, které na něm bylo nutné provést. Je zde také popsáno začlenění externích metod do hierarchie tříd a výsledný podsystém, který vznikl sloučením všech dílčích úprav.

Kapitola 4 popisuje úskalí při implementaci nového podsystému numerických metod a pozoruhodné aspekty, se kterými jsem se v jednotlivých knihovnách setkal. Nakonec je popsán rozsah práce a provedena diskuse efektivity a korektnosti kódu.

Kapitola 5 obsahuje popis testovacích příkladů a výsledky porovnání interních metod, obsažených v knihovně SIMLIB, s nově přidanými externími metodami.

Závěrečná kapitola 6 popisuje ve zkratce moji práci a její přínos. Zmiňuji se zde také o cestách, kterými je možné se ubírat při dalším vývoji numerické integrace v knihovně SIMLIB.

2 Numerická integrace

Základním způsobem popisu složitých systémů při spojité simulaci jsou soustavy obyčejných diferenciálních rovnic a algebraických rovnic. Tato práce se bude zaobírat pouze obyčejnými diferenciálními rovnicemi(ODR, anglicky ODE z *ordinary differential euation*), protože právě pro jejich řešení se používá numerická integrace. K řešení algebraických rovnic se také užívá jistý typ numerických metod, nicméně tato tématika není náplní mojí práce.

2.1 Numerické řešení diferenciálních rovnic

2.1.1 Princip a klasifikace numerických metod

Při numerickém řešení diferenciální rovnice s počátečními podmínkami(označováno také jako počáteční úloha, anglicky IVP z *initial value problem*) hledáme přibližné řešení y jako posloupnost hodnot $y(t_i) = y(t_0), y(t_1), y(t_2),..., y(t_n)$ v diskrétních bodech(uzlech) intervalu $I=(t_0,t_n), t_i=t_0+\Sigma^i_{k=0} h_k$, $i=0,1,...,n$, které dělí interval I na n dílů o velikosti h. Hodnota h se pak nazývá integrační krok. Ten se může v průběhu numerické integrace měnit a jeho aktuální hodnota je dána rozdílem:

$$h_i = t_{i+1} - t_i \tag{2.1}$$

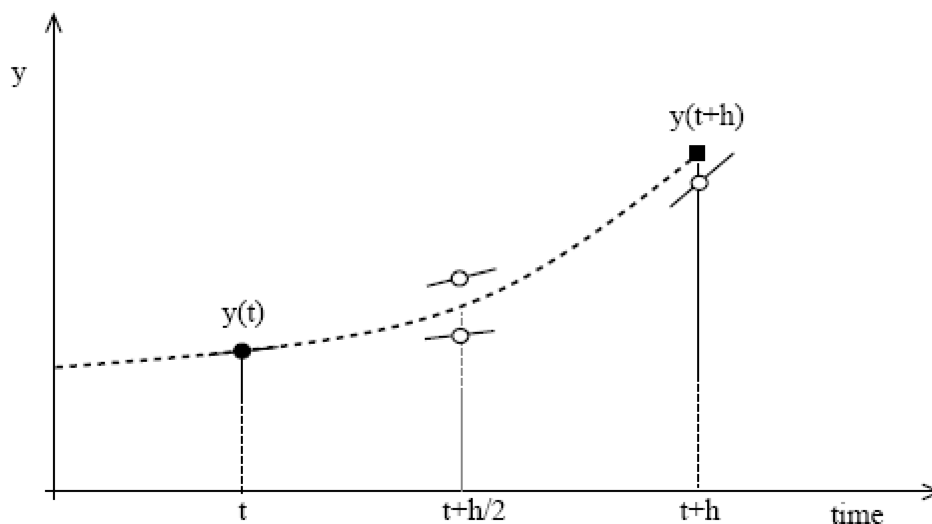
Princip výpočtu pak může být dvojího druhu. Buď se jedná o aproximaci $y(T)$ polynomem N-tého stupně(Taylorův rozvoj) na základě známé aktuální hodnoty $y(t)$, tj. počáteční stav nebo výsledek předchozího kroku, nebo o extrapolaci hodnoty $y(t+h)$. V případě aproximace pak stupeň N rozvoje určuje tzv. řád metody.

Jestliže k výpočtu nové hodnoty y_{i+1} postačuje znalost aktuálního stavu y_i , pak se jedná o metodu jednokrokovou. Oproti tomu metody vícekrokové využívají historii stavů nebo vstupů, takže je nutné znát řešení ve více předchozích uzlech, např. $y_{i-1}, y_{i-2},..., y_{i-s}$, $s \in N$. Pro $s=1$ se jedná o metodu dvoukrokovou, pro $s=2$ o metodu tříkrokovou atd. Tyto metody se potýkají s problémy při startu výpočtu, kdy historie stavů není k dispozici.

2.1.2 Jednokrokové metody

Princip jednokrokových metod je znázorněn na obrázku 2.1. Derivace v zadaném bodě odpovídá směrnici tečny k výslednému funkčnímu průběhu. Tyto směrnice lze vypočítat pro každý bod roviny podle zadané diferenciální rovnice $y'=f(t, y(t))$. Protože známe hodnotu řešení $y(t)$ na počátku kroku, můžeme využít hodnotu derivace v tomto bodě k výpočtu následující hodnoty $y(t+h)$. Nejjednodušší

jednokroková metoda(Eulerova metoda) počítá novou hodnotu přímo, metody vyšších řádů, jako jsou metody Runge-Kutta, počítají ještě s několika pomocnými body uvnitř kroku.



Obrázek 2.1: Princip jednokrokových metod

2.1.2.1 Eulerova metoda

Eulerova metoda, jinde v literatuře se můžeme setkat také s pojmenováním metoda Eulerových polygonů, je nejjednodušším představitelem jednokrokových metod, protože pro výpočet nového stavu používá pouze derivaci ve výchozím bodě. Vzorec pro výpočet nového stavu na konci kroku pak vypadá takto:

$$y_{i+1} = y_i + hf(t_i, y_i) \quad , \tag{2.1}$$

kde $f(t_i, y_i)$ je hodnota derivace na začátku i-tého kroku.

Jak již bylo zmíněno, Eulerova metoda je velmi jednoduchá, což s sebou nese určité negativní vlastnosti. Konvergence metody je pomalá a pro dosažení vyšší přesnosti je nutné volit malý krok. To vede k nárůstu počtu aritmetických operací(opakované vyčíslování pravých stran diferenciálních rovnic), a tedy k nízké efektivitě. Proto není v praxi příliš použitelná a využívá se metod vyšších řádů.

2.1.2.2 Metody Runge-Kutta

Rodina metod Runge-Kutta používá oproti Eulerově metodě mezivýpočty uvnitř kroku(při zachování stejné délky kroku), čímž dosahuje mnohem přesnějších výsledků. Základní variantou těchto metod jsou metody explicitní, kde se výpočet nového stavu na konci kroku řídí těmito vzorci:

$$y_{n+1} = y_n + h\,\Sigma_{i=1}^{s} b_i k_i \quad , \tag{2.2}$$

kde:

$$k_1 = f(t_n, y_n)$$

$$k_2 = f(t_n + c_{2h}, y_n + a_{21} h k_1) \tag{2.3}$$

$$\dots$$

$$k_s = f(t_n + c_{sh}, y_n + a_{s1} h k_1 + a_{s2} h k_2 + \dots + a_{ss-1} h k_{s-1})$$

Konstanta s pak určuje počet použitých pomocných bodů a koeficienty $a_{ij} (1 \leq j < i \leq s)$, b_i a c_i $(i \in (1,s))$ jsou u těchto metod vypočteny tak, aby metoda zvoleného řádu odpovídala Taylorovu rozvoji funkce y_t stejného řádu.

Nedílnou součástí simulace je výpočet odhadu chyby metody, který dovoluje pružně reagovat a měnit délku kroku metody tak, aby se chyba udržovala v zadaných mezích. Velká výhoda spočívá v tom, že si metoda podle aktuální chyby rozhodne, zda je přesnost dostačující a je možné krok prodloužit, nebo naopak zkrátit, aby byla dosažena požadovaná velikost chyby. U metod Runge-Kutta je možné tento výpočet doplnit např. použitím metody polovičního kroku, dochazí při ní však k radikálnímu nárůstu vyčíslování pravé strany rovnic a značně tak zpomaluje výpočet. Proto se častěji využívá tzv.párových metod.

Mějme metodu řádu p, tato metoda potom bude k odhadu chyby kroku používat metodu řádu $p-1$ podle vzorce:

$$y^*_{n+1} = y_n + h \Sigma_{i=1}^{s} b^*_i k_i \quad , \tag{2.4}$$

kde hodnoty k_i jsou stejné jako pro metodu vyššího řádu a výpočet chyby kroku je následovný:

$$e_{n+1} = y_{n+1} - y^*_{n+1} = h \Sigma_{i=1}^{s} (b_i - b^*_i) k_i \tag{2.5}$$

Tento způsob je podstatně efektivnější, protože recykluje již vypočtené hodnoty. Využívají je varianty Runge-Kuttových metod Fehlbergova, Vernerova a Dormand-Princeova.

2.1.3 Vícekrokové metody

Tyto metody používají k výpočtu nového stavu y_{i+1} hodnoty z předchozích k kroků, kde počet k je určen řádem metody. Obecně pro každou k krokovou metodu platí vzorec:

$$y_{n+1} = \Sigma_{i=0}^{r} \alpha_i y_{n-i} + h \Sigma_{j=-1}^{s} \beta_j f_{n-j} \tag{2.6}$$

Pokud je $\beta_{-1} = 0$, lze hodnotu y_{n+1} určit z $r+1$ předchozích hodnot y_n(respektive z $s+1$ předchozích hodnot f_n) a jedná se o metodu explicitní, v jiném případě se musí rovnice 2.6 řešit iteračně a jedná se o metodu implicitní. Typickým příkladem explicitní vícekrokové metody je metoda Adams-Bashford čtvrtého řádu:

$$y_{n+1} = y_n + \frac{h}{24} (55 f_n - 59 f_{n-1} + 37 f_{n-2} - 9 f_{n-3}) \tag{2.7}$$

Je na ní možné pozorovat, že kromě aktuální hodnoty používá také hodnoty derivací ze tří předcházejících kroků, což bývá zároveň problém těchto metod při jejich startu, kdy nejsou tyto hodnoty k dispozici. Častým řešením je použití jednokrokové metody pro výpočet prvních k kroků potřebných pro start metody.

Příkladem implicitní metody je pak metoda Adams-Moulton čvtrtého řádu, která oproti předcházejícímu příkladu využívá pouze dvou hodnot derivací z předcházejících kroků, ale navíc potřebuje k výpočtu hodnotu derivace f_{n+1}:

$$y_{n+1}=y_n+\frac{h}{24}(9f_{n+1}+19f_n-5f_{n-1}+f_{n-2}) \tag{2.8}$$

Prakticky jsou často používány metody prediktor-korektor, které kombinují přístup explicitních i implicitních metod. Prvním krokem je odhad nové hodnoty y_{n+1} explicitní metodou a následný výpočet derivace f_{n+1} v tomto bodě. Díky získaným hodnotám může být použita implicitní metoda ke zpřesnění aproximace y_{n+1}.

2.1.4 Tuhé systémy

Uvést přesnou definici pro tuhost systému je obtížné. Často se však v různých zdrojích(např. [2]) setkáme s neformální definicí tuhého systému(anglicky *stiff system*) podobné naší:

Tuhé systémy jsou speciálním případem, kdy se popis chování systému skládá z jevů s velmi odlišnými časovými konstantami, díky nimž během simulace dochází k rychlé změně výsledného řešení.

Vzniká tak problém s určením optimální délky kroku numerické integrace. Pokud zvolíme krok malý, který by vyhovoval rychlým dějům, výpočet bude značně neefektivní kvůli nárůstu kumulované chyby(viz vlastnosti numerických metod), naopak pokud zvolíme dlouhý krok, výsledky simulace nebudou dostatečně přesné.

V praxi se tento jev projevuje tak, že klasické metody pro numerickou integraci musí použít extrémně malou velikost kroku, jinak vedou k nestabilnímu řešení nebo jsou pro řešení daného systému zcela nevhodné. Tato skutečnost si vynutila vývoj speciálních metod, které se dokáží s tuhými systémy vypořádat lépe, nicméně ideální metoda doposud nebyla nalezena.

2.1.5 Vlastnosti numerických metod a výběr optimální metody

Nejdůležitějšími vlastnostmi numerických metod jsou přesnost, rychlost(efektivita) a stabilita. Tyto vlastnosti jasně určují použitelnost metody pro řešení daného problému a její chování při změně parametrů simulace.

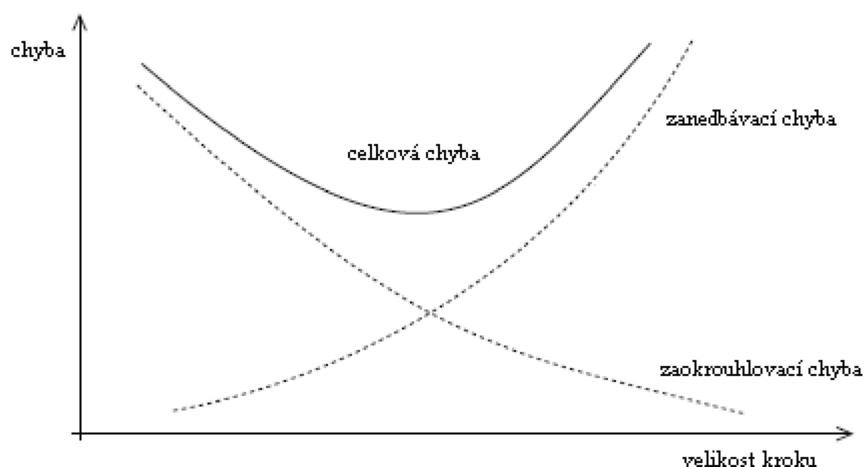
2.1.5.1 Přesnost metody

Obecně platí, že metody vyšších řádů jsou přesnější než metody řádů nižších. Pokud máme k dispozici srovnání, přesnost metody je potom určena rozdílem přesného řešení problému od vypočtené přibližné hodnoty, což můžeme označit jako tzv. chybu metody. Chyba jednoho kroku výpočtu se nazývá lokální chyba a je součtem dvou složek:

$$\varepsilon_l = \varepsilon_t + \varepsilon_r \quad (2.9)$$

První z nich je chyba numerické aproximace metody ε_t, se kterou se často setkáme také pod pojmenováním chyba zanedbávací(anglicky *truncation error*). Chyba ε_r je chybou zaokrouhlovací (anglicky *round-off error*), která je dána konečným počtem bitů pro uchování reálného čísla v počítači. Celková chyba n-tého kroku metody je pak ovlivněna kumulací lokálních chyb kroků předcházejících, proto je také označována jako chyba kumulovaná.

Na první pohled by se tedy mohlo zdát, že pokud zvolíme dostatečně nízkou velikost kroku, dosáhneme přesných výsledků, protože omezíme velikost zanedbávací chyby. Musíme ovšem vzít v úvahu, že zmenšení kroku metody způsobí zvýšení objemu výpočtů, a tedy i nárůst chyby zaokrouhlovací. Pokud bychom uvažovali opačně a zvětšili velikost kroku, poroste chyba zanedbávací a chyba zaokrouhlovací bude klesat. Tato závislost je jasně patrná ze vzorce 2.9 pro výpočet lokální chyby a graficky ji popisuje obrázek 2.2.



Obrázek 2.2: Závislost velikosti chyby výpočtu na velikosti kroku

2.1.5.2 Rychlost metody

Podobná situace jako u přesnosti nastává při srovnání rychosti metod. Teorie[3] tvrdí, že metody vyšších řádů mohou být rychlejší než metody nižších řádů, protože jsou zatíženy menší lokální chybou a mohou tak pracovat s větší délkou kroku. Stejný zdroj také doplňuje, že pravdivost tohoto výroku je podmíněna požadavky na přesnost. Pokud použijeme metodu vyššího řádu s požadavkem na malou přesnost výpočtu, dochází k degradaci rychlosti a přesnosti výsledku kvůli provádění

zbytečně velkého množství výpočtů. Podobně se metody vyšších řádů chovají, když jim omezíme maximální velikost kroku, např. vlivem vzorkování výsledných hodnot.

2.1.5.3 Stabilita metody

Stabilita metody vyjadřuje chování metody vzhledem k velikosti integračího kroku. Nestabilita řešení je jev provázený skokovou změnou chování metody, kdy metoda produkuje naprosto chybné výsledky. S rostoucím krokem se pravděpodobnost nestability zvyšuje. Zatímco u jednokrokových metod roste stabilita s řádem, u metod vícekrokových je situace opačná. Pro každou diferenciální rovnici existuje krok, při kterém nestabilita začíná - hovoříme o tzv. mezním kroku stability řešení. Dokonce mohou existovat problémy, u kterých s použitím určité metody není možné dosáhnout stabilního řešení.

2.1.5.4 Volba optimální metody

Při výběru použité integrační metody je třeba zamyslet se nad tím, co vlastně od metody potřebujeme. Je důležité držet se starého moudra, že „méně je někdy více“ a nepoužívat vždy tu nejsložitější metodu. V případech, kdy neznáme přesné počáteční hodnoty a jde nám pouze o informativní obraz průběhu řešení, bohatě postačí některá z metod nižšího řádu, protože stejně dosáhneme přesného výsledku pouze vzhledem k našemu nepřesnému vstupu. Pro obecná a přesná řešení většinou vyhovuje některá z variant metody Runge-Kutta řádu 4 nebo 5, které bychom mohli považovat za metody univerzální. Také je třeba mít na paměti, že tuhé systémy potřebují speciální metody.

3 Návrh podsystému num. integrace

Změny nutné pro rozšíření knihovny SIMLIB o externí metody si vyžádaly nezbytnou úpravu na stávajícím podsystému numerické integrace. Také jsem se pokusil o „vyčištění“ uživatelského rozhraní SIMLIBu pro numerickou integraci, kde stále přetrvávalo množství tříd, které jsou důležité z hlediska simulace, nikoliv z pohledu uživatele. Nakonec jsem se zabýval připojením externích metod do knihovny SIMLIB a patřičnou úpravou hierarchie tříd, která je s tím spojena.

3.1 Stávající podsystém numerické integrace

V této kapitole se seznámíme s prostředky spojenými s numerickou integrací, které simulační knihovna SIMLIB poskytuje. Dozvíme se něco o výhodách použitého řešení a hierarchií tříd, která podsystém numerické integrace realizuje. Bude se jednat o zevrubný popis; detailní vysvětlení všech veřejných metod jendotlivých tříd bylo již popsáno dříve[3].

3.1.1 Třídy numerické integace

3.1.1.1 Třída Integrator

Pro popis modelů v knihovně SIMLIB se využívá koncepce bloků, která je dobře známá z časů minulých. Třída Integrator je pak speciálním případem stavového bloku, který poskytuje základní stavební kámen pro numerickou integraci. Stejně jako u blokového zápisu má v SIMLIBu integrátor vstup, na nějž je přivedena integrovaná funkce $x(t)$(zadává se jako parametr konstruktoru nebo metodou `SetInput`, např. v případě použití polí integrátorů; vysvětleno v [3]), počáteční stav vyhovující počáteční podmínce a výstup, kterým je $\int x(t)dt$ a je přístupný voláním metody `Value`.

Zajímavostí spojenou nejen s integrátory, ale i s ostatními bloky, je tzv. automatická konstrukce výrazových stromů. Tato technika využívá přetěžování operátorů, kdy jejich vhodnou definicí u bloků zajistí, že se při jejich volání v inicializaci modelu dynamicky vytvoří speciální objekty a propojí je podle struktury výrazů do stromů[2]. Vyhodnocení vstupu integrátoru se pak provádí tzv. na žádost, kdy integrátor rozešle zprávy s požadavky na vyhodnocení všem vstupům, ty se vyhodnotí(v případě, že se jedná o listy výrazového stromu) nebo obdobně rekurentně požadují vyhodnocení svých vlastních vstupů, nakonec jsou získané hodnoty zpracovány integrátorem a je vrácen výsledek.

Kromě uživatelského rozhraní nabízí třída Integrátor také rozhraní pro numerické metody. Prostřednictvím sady metod poskytuje přístup ke čtení a změně aktuálních hodnot svého vstupu

a výstupu a také k jejich hodnotě z předcházejícího kroku numerické integrace, kterou si integrátor drží ve svém vnitřním stavu.

3.1.1.2 Třída IntegratorContainer

Tato třída spravuje kontejner integrátorů, které jsou použity v modelu simulace. Každy integrátor je při vytvoření svým konstruktorem do tohoto kontejneru zařazen a při zániku je z něj svým destruktorem vyjmut. Přitom musí být kontejner viditelný na globální úrovni, aby bylo umožněno ostatním entitám podílejícím se na spojité simulaci přístupovat k jeho položkám. Tento problém řeší návrhový vzor jedináček(anglicky *singleton*) blíže popsaný v literatuře[4].

Tato konstrukce definuje kontejner jako statickou položku třídy a zajišťuje, že bude existovat právě jedna instance třídy IntegrationContainer, která bude při prvním použití inicializována. Ostatní členské metody pak ke kontejneru přistupují pomocí soukromé metody Instance, která vrací ukazatel na seznam integrátorů. Tím je vyřešen problém s pořadím provádění konstruktorů globálních objektů v C++, který by mohl nastat v případě použití globálního objektu.

Ve svém rozhraní poskytuje třída IntegrationContainer prostředky pro zjištění počtu integrátorů či hromadnou inicializaci a vyhodnocení vstupů celého seznamu integrátorů. Důležitá je také metoda pro uschování stavu integrátorů do vnitřních pomocných proměnných a k ní metoda opačná pro obnovení uschovaných hodnot, které najdou uplatnění především při přípravě systému na krok numerické integrace.

3.1.1.3 Třída IntegrationMethod a její potomci

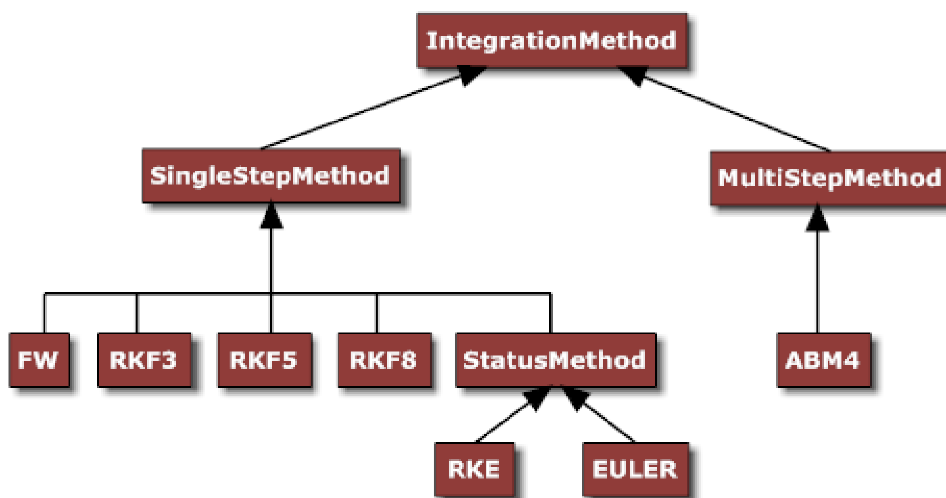
Jedná se o abstraktní bázovou třídu, která zapouzdřuje metody nezbytně nutné pro přípravu systému na provedení numerické integrace. Jak je patrné z obrázku 3.1, veškeré numerické metody nebo skupiny metod obsažené v knihovně SIMLIB jsou potomky této třídy. Jednotlivé numerické metody jsou jednoznačně určeny svým názvem a jejich seznam je privátním atributem bázové třídy IntegrationMethod. Z pohledu SIMLIBu jsou metody kvůli značnému rozdílu v implementaci rozděleny na jednokrokové, reprezentované třídou SingleStepMethod, a vícekrokové, zastoupené třídou MultiStepMethod.

Protože každá integrační metoda potřebuje ke svým výpočtům uchovávat pomocné hodnoty, byla pro tyto účely v knihovně SIMLIB vytvořena vnořená třída Memory. Pro každou numerickou metodu registruje soukromý seznam pomocných pamětí, jejichž velikost se dynamicky mění podle počtu integrátorů systému.

Třída SingleStepMethod obsahuje abstraktní základ pro jednokrokové samostartující metody. Potomci této třídy mohou být použiti buď samostatně, nebo pro start metody vícekrokové, v tomto případě jsou dokonce schopni upravit své chování tak, aby nedocházelo k prudkému zvětšování délky kroku a vícekroková metoda tak byla lépe odstartována.

Druhou skupinou metod jsou metody vícekrokové, které jsou představovány třídou MultiStepMethod. Je pro ně typické, že musí být odstartovány některou z jednokrokových metod. Potomci této třídy mají k dispozici rozhraní pro určení startovací metody a následnou komunikaci s ní.

Zajímavým elementem hierarchie numerických metod je třída StatusMethod, která je potomkem třídy SingleStepMethod. Tato třída seskupuje speciální jednokrokové metody, které mají krok rozdělený na dvě poloviny, přičemž po provedení každé z nich se systém nachází v konzistentním stavu. Této jejich vlastnosti se využívá při přítomnosti stavových podmínek(speciální prvky sloužící k detekci změny stavu systému a vyvolání reakce na tuto změnu). Tyto metody potřebují kromě paměti pro stav integrátorů také paměť pro uchování hodnot stavových bloků. Kvůli tomu vznikla třída StatusMemory odvozená od třídy Memory, která se od svého předka liší v tom, že velikost paměti mění dynamicky podle počtu stavových bloků.



Obrázek 3.1: Hierarchie tříd numerické integrace v SIMLIB

3.1.2 Rozhraní numerické integace

Pro použití podsystému numerické integrace platí stejná pravidla jako pro jakoukoliv jinou část knihovny SIMLIB. Je třeba direktivou #include “simlib.h“ vložit rozhraní knihovny a při překladu k simulovanému modelu přilinkovat knihovnu SIMLIB.

Samotné rozhraní je pro jednoduchost tvořeno sadou funkcí, které bychom mohli rozdělit do dvou skupin. První skupinu tvoří obecné funkce pro nastavení parametrů spojité simulace. Mezi ně patří[1]:

1 U všech funkcí, které nemají uvedeny návratový typ, budeme uvažovat jako návratovou hodnotu void

- SetOutput(const char* name) – parametr name určuje název souboru, do kterého budou přesměrovány výstupy simulace. Pokud není tato funkce použita, pro výpisy bude použit standardní výstup.
- Init(double t0, double t1) – určuje počáteční a koncový čas simulace. Tato funkce musí být v modelu uvedena právě jednou, a to před voláním funkce Run. Pokud je parametr t1 zadán, musí být větší než t0, jinak provedení funkce vyvolá chybu a konec simulace. V případě, že zadán není, simulace běží do maximálního času 10^{30}. Tento jev se používá v případě omezení délky simulace pomocí stavových událostí.
- Run() – spouští simulaci. Pro provedení simulace je nezbytné tuto funkci uvést v modelu, jinak se simulace vůbec nespustí. Její nepřítomnost není nijak detekována a je plně pod kontrolou uživatele.
- Stop() – způsobí okamžité ukončení běhu simulace. Slouží k ukončení simulace před dosažením jejího koncového času.

Druhou kategorií jsou funkce ovlivňující průběh numerické integrace. Jejich součástí jsou:

- SetStep(double Min_Step, double Max_Step) – slouží k omezení velikosti kroku numerické integrace v rozsahu od Min_Step do Max_Step.
- SetAccuracy(double Abs_Err, double Rel_Err) – nastavuje velikost absolutní a relativní chyby. Výchozí hodnoty jsou 0 pro absolutní chybu a 10^{-3} pro chybu relativní.
- SetAccuracy(double Rel_Err) – varianta s implicitně určenou velikostí absolutní chyby na nulovou hodnotu.
- SetMethod(const char* name) – nastaví integrační metodu podle parametru name, pokud metoda se zadaným jménem neexistuje, je oznámena chyba a simulace je ukončena.
- char* const GetMethod() – vrátí jméno aktuálně nastavené metody.
- SetStarter(const char* name, const char* slave_name) – nastaví startovací jednokrokovou metodu metodě vícekrokové. Parametr name určuje název vícekrokové metody, parametr slave_name název jednokrokové startovací metody. Obě metody musí existovat a musí být uvedeny ve správném pořadí.
- SetStarter(const char* slave_name) – aktuálně používané metodě nastaví startovací metodu. Pokud není aktuálně používaná metoda vícekroková nebo metoda určena parametrem slave_name není existující jednokrokovou metodou, je vrácena chyba.
- char* const GetStarter(const char* name) – vrací jméno metody, která je nastavená jako startovací pro metodu určenou parametrem name. Metoda určená parametrem name musí být vícekroková a musí existovat.

- char* const GetStarter() – vrací jméno metody, která je nastavená jako startovací pro aktuálně používanou metodu. Přitom musí být aktuální metoda vícekroková.

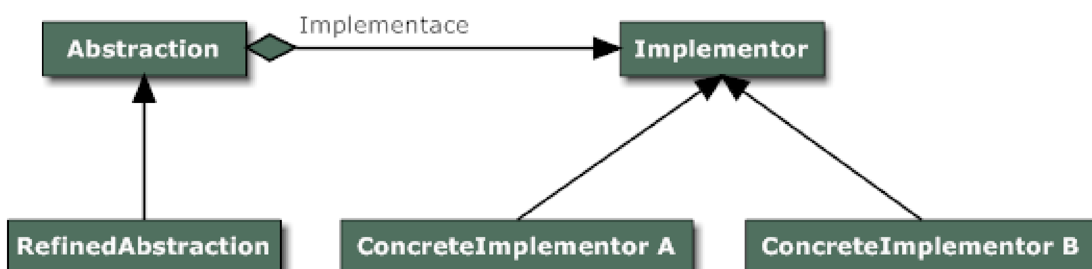
Z numerických metod nabízí knihovna SIMLIB pětici standardních jednokrokových metod, kterými jsou metoda Eulerova, Englandova varianta Runge-Kuttovi metody, která je zároveň přednastavena jako základní metoda, pokud uživatel neurčí jinak, a trojice metod Runge-Kutta Fehlberg řádu 3, 5 a 8. Navíc obsahuje také jednu vícekrokovou metodu(Adams-Bashforth-Moulton čtvrtého řádu) a Fowler-Wartonovu metodu pro řešení tuhých systémů. Přesnější specifikace těchto metod je popsána v literatuře[3].

3.2 Úprava stávajícího podsystému

3.2.1 Rozhraní numerické integrace

Deklarace třídy IntegrationMethod a jejich potomků se v SIMLIBu nachází spolu s ostatními třídami v rozhraní knihovny simlib.h, což z návrhového hlediska není příliš vhodné. Cílem by mělo být „čisté" uživatelské rozhraní, které bude obsahovat pouze třídy a funkce důležité z hlediska modelu, nikoliv simulace. Proto jsem se rozhodl oddělit vlastní implementaci numerických metod od jejich uživatelského rozhraní. Možné řešení nabízí návrhový vzor most(anglicky *bridge*) popsaný v literatuře[4], jehož struktura je zobrazena na obrázku 3.2. V našem případě se jedná o jeho degenerovanou variantu, protože existuje pouze jedna implementace. Třída Implementor tím ztrácí na významu, přesto je jeho použití vyhodné z hlediska snazší udržovatelnosti kódu. Změna provedená v implementační části nemá totiž na uživatele vliv, protože jeho rozhraní zůstává neměnné.

Konstrukce vzoru most dosáhneme vyčleněním hierarchie numerických integračních metod z uživatelského rozhraní knihovny symlib.h do neveřejného hlavičkového souboru, ke kterému bude přistupovat pouze simulační část knihovny. Přístup k implementaci tak máme zcela pod kontrolou a před samotným uživatelem je skryta.



Obrázek 3.2: Struktura návrhového vzoru bridge

S dalším rozšiřováním knihovny o nové numerické metody roste také potřeba jednoduchým způsobem získat seznam všech použitelných metod. Nabízí se několik kvalitativně srovnatelných řešení, např. nadefinovat dvě funkce, přičemž první bude vracet počet dostupných metod a druhá vrátí název metody zadaného čísla. Rozhodl jsem se naplno využít možností, které poskytuje C++, a zpřístupnit seznam metod uložených v kontejneru pomocí iterátorů. Bylo tedy nutné přidat dvě funkce vracející konstantní iterátor na začátek a konec vektoru dostupných metod, aby byl zajištěn přístup jen pro čtení:

- const std::vector<const char*>::iterator FirstMethod() – vrací konstantní iterátor na začátek vektoru metod.
- const std::vector<const char*>::iterator LastMethod() – vrací konstantní iterátor na konec vektoru metod.

Výpis seznamu dostupných metod pak lze zapsat velmi jednoduše:

```
std::vector<const char *>::iterator FirstIt;
for (FirstIt=FirstMethod();FirstIt<LastMethod();FirstIt++)
{ Print("%s \n", *FirstIt);}
```

3.2.2 Třída IntegratorValues

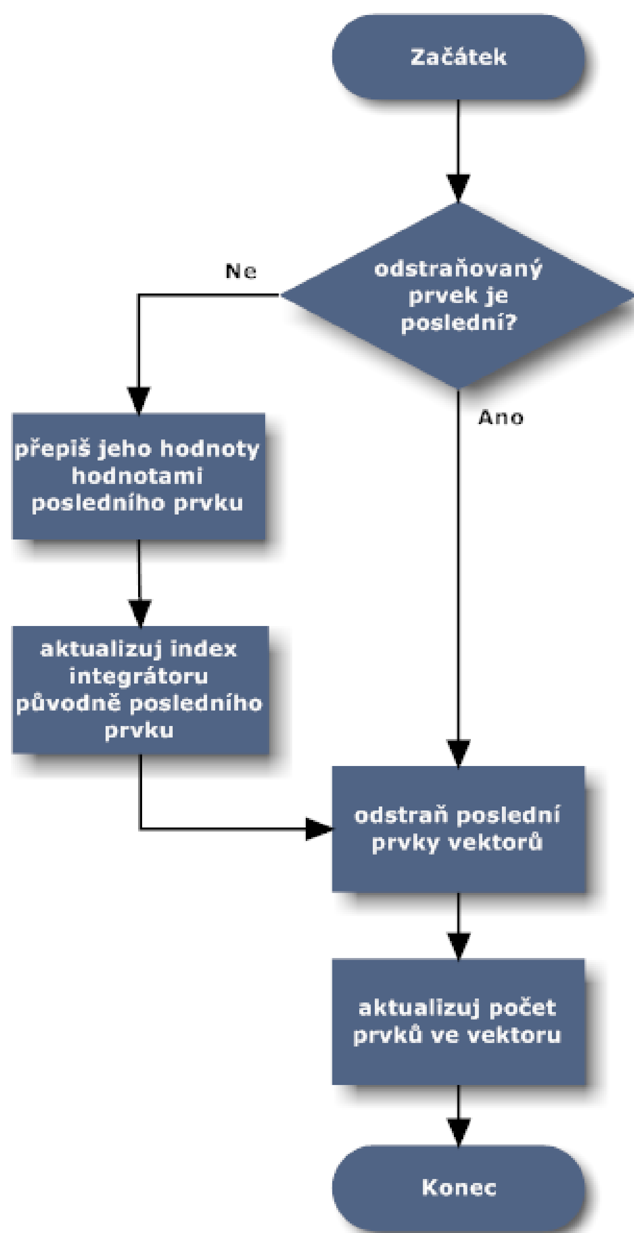
Při studiu externích metod, kterými chci obohatit knihovnu SIMLIB, jsem zjistil, že externí metody používají pole integrátorů, kdežto v SIMLIBu plní tento účel seznam integrátorů reprezentovaný třídou IntegratorContainer. Hodnoty vstupu a výstupu jsou pak vázány na konkrétní integrátor a jsou uchovány v jeho stavu. Musel jsem se potýkat s takovým řešením, které by s nejmenším úsilím zkombinovalo oba přístupy tak, aby bylo možné použít externí metody a v samotné knihovně došlo k minimálnímu počtu úprav.

Po důkladné analýze jsem se rozhodl vytvořit pro správu hodnot integrátorů speciální třídu IntegratorValues. Musí pro ni platit, že bude mít právě jednu instanci, která bude poskytovat integračním metodám a řídící části simulace přístup k hodnotám integrátorů. S podobným problémem jsme se setkali u třídy IntegrationContainer a i tady nám východisko poskytne návrhový vzor jedináček[4]. Logickou volbou pro uložení hodnot vstupu a výstupu integrátorů z aktuálního a minulého kroku jsou datové kontejnery vector, které budou statickými položkami třídy. Výhodou šablony vector je její kompatibilita s jednorozměrným polem, navíc nabízí automatickou úpravu velikosti po vložení nebo vyjmutí prvku, kterou bych musel u dynamicky alokovaného pole řešit sám.

Jednotlivé metody třídy IntegratorValues:

- IntegratorValues& Instance() – vrací referenci na instanci třídy IntegratorValues, jejímž prostřednictvím přistupují ostatní metody ke kontejnerům s hodnotami integrátorů.
- int AddIntegrator() – metoda rozšíří vektory hodnot integátorů o jeden prvek s výchozí hodnotou 0. Vrací index pro přístup k těmto hodnotám.
- RemoveIntegrator(int Index) – odstraní z vektorů hodnoty integrátoru umístěné na daném indexu vektorů. Pro zachování sousvislého bloku paměti postupuje podle diagramu uvedeném na obrázku 3.3.
- SaveState(int Index) – uloží stav integrátoru.
- RestoreState(int Index) – obnoví stav integrátoru z minulého kroku.
- SetState(double Value,int Index) – nastaví aktuální hodnotu výstupu integrátoru.
- SetOldState(double Value,int Index) – obnoví hodnotu výstupu integrátoru hodnotou z minulého kroku.
- double GetState(int Index) – vrací hodnotu výstupu integrátoru z aktuálního kroku.
- double GetOldState(int Index) – vrací hodnotu výstupu integrátoru z minulého kroku.
- SetDiff(double Value,int Index) – nastaví hodnotu vstupu integrátoru aktuálního kroku na hodnotu Value.
- SetOldDiff(double Value,int Index) – nastaví hodnotu vstupu integrátoru minulého kroku na hodnotu Value.
- double GetDiff(int Index) – vrací hodnotu vstupu integrátoru z aktuálního kroku.
- double GetOldDiff(int Index) – vrací hodnotu vstupu integrátoru z minulého kroku.
- double* GetStateVector() – vrací ukazatel na pole výstupních hodnot integrátorů.
- double* GetDiffVector() – vrací ukazatel na pole vstupních hodnot integrátorů.

Jak je patrné, názvy metod pro práci s hodnotami integrátorů zůstaly zachovány ve stejném tvaru, jako tomu bylo v původním popisu třídy Integrator[3], takže zbylé změny byly pouze mechanickým nahrazením třídy Integrator třídou IntegratorValues a svůj úkol minimalizovat změny v knihovně SIMLIB jsem splnil.



Obrázek 3.2: Vývojový diagram metody RemoveIntegrator

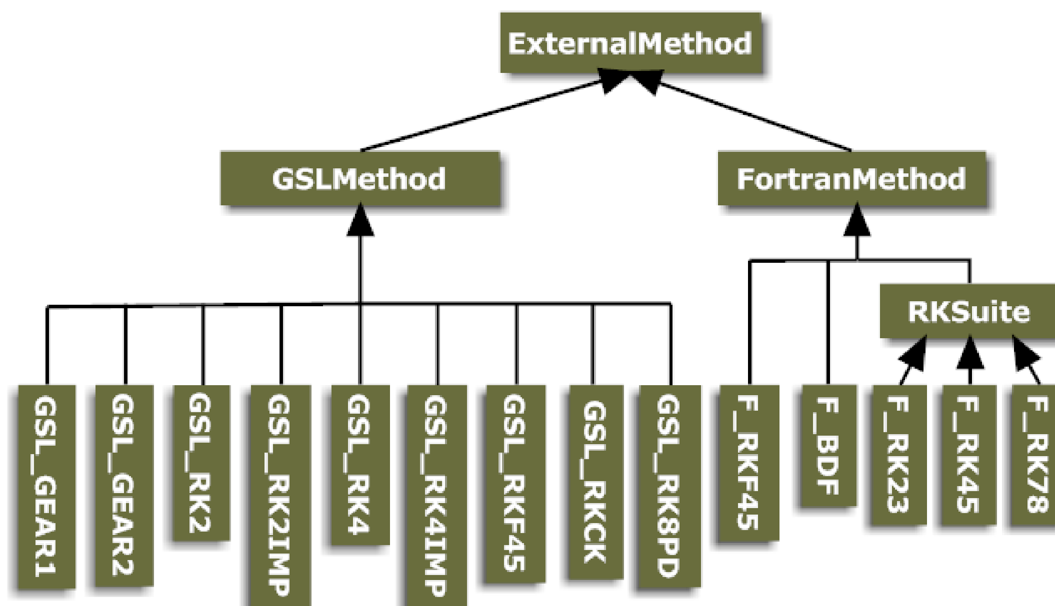
3.3 Zařazení externích metod v novém podsystému

Jedná se o metody, které nejsou součástí knihovny SIMLIB a jejich chování se výrazně liší od metod obsažených v této knihovně, proto je není možné zařadit do klasické hierarchie numerických metod jako potomky tříd SingleStepMethod nebo MultiStepMethod. Bylo tedy nutné pro ně vytvořit speciální třídu ExternalMethod.

Jedním ze zdrojů pro externí metody, které jsem použil v mé práci, je knihovna GSL[5]. Obsahuje širokou škálu metod Runge-Kutta a jiné, díky čemuž je schopna pokrýt většinu problémů, na které by mohl uživatel při řešení obyčejných diferenciálních rovnic narazit. Vybral jsem si ji také proto, že je napsána v jazyce C, takže její propojení s podsystémem numerické integrace knihovny SIMLIB nebude představovat žádný problém.

Jako zdroj numerických metod v jazyce Fortran mi posloužila databáze NetLib[6]. Rozdíl oproti GSL je především v tom, že GSL je jediná knihovna, která má jednotné rozhraní pro všechny metody, zatímco NetLib obsahuje sadu knihoven, z nichž každá má své vlastní rozhraní. Proto vytvořit obecnou třídu, která by spojovala požadavky všech fortranovských knihoven není z hlediska návrhu korektní.

Na základě předcházejících faktů a analýzy jednotlivých knihoven s externími metodami jsem vytvořil návrh třídní hierarchie externích integračních metod(obrázek 3.4), který bude součástí podsystému numerické integrace knihovny SIMLIB. V následujících kapitolách budou popsány jednotlivé třídy tvořící podskupinu externích integračních metod.



Obrázek 3.4: Hierarchie externích metod

3.3.1 Třída ExternalMethod

Třída ExternalMethod je základním stavebním kamenem, od kterého se odvíjí ostatní externí metody. Je přímým potomkem třídy IntegrationMethod. Jedná se o abstraktní třídu obecnou pro všechny externí metody. Oproti třídě IntegrationMethod přidává metody pro zjištění a aktualizaci počtu integrátorů, který si každá externí metoda udržuje, aby mohla pružně reagovat na změnu počtu integrátorů v systému změnou velikosti pomocné paměti.

Každá externí metoda provádějící numerickou integraci musí být jejím potomkem a musí definovat metody, které jsou volány při provedení kroku simulace. Jedná se o metody PrepareStep pro přípravu kroku metody, Integrate pro provedení kroku numerické integrace a metodu TurnOff, která je volána při změně integrační metody nebo na konci simulace, aby po metodě „uklidila".

Popis metod třídy ExternalMethod:

- ActualizeINum() – aktualizuje počet integrátorů v aktuálním kroku.
- int GetINum() – vrací počet integrátorů z minulého kroku.
- SetINum(int Number) – nastaví počet integrátorů na hodnotu danou parametrem Number.

3.3.2 Třída GslMethod a její potomci

Rozhraní numerické integrace v knihovně GSL je tvořeno sadou funkcí řešících výběr metody, alokaci pomocných pamětí atd. Oproti tomu SIMLIB používá odlišnou koncepci s vyšší úrovní abstrakce. Zásadní problém tedy spočívá v převedení rozhraní numerických metod z GSL na rozhraní, které očekává knihovna SIMLIB. I zde využijeme toho, že tento typ problému není nikterak neobvyklý a návod k jeho řešení poskytuje návrhový vzor adaptér[4].

Úlohu adaptéru v tomto případě plní abstraktní třída GslMethod, od které budou odvozeny jednotlivé numerické metody obsažené v knihovně GSL. Poskytuje rozhraní pro přístup k atributům třídy, které určují obecné nastavení integračních metod knihovny GSL. Ty mohou být statickými prvky třídy, protože jsou jednotné pro všechny obsažené metody. Jednotliví potomci této třídy se pak liší pouze použitou integrační metodou.

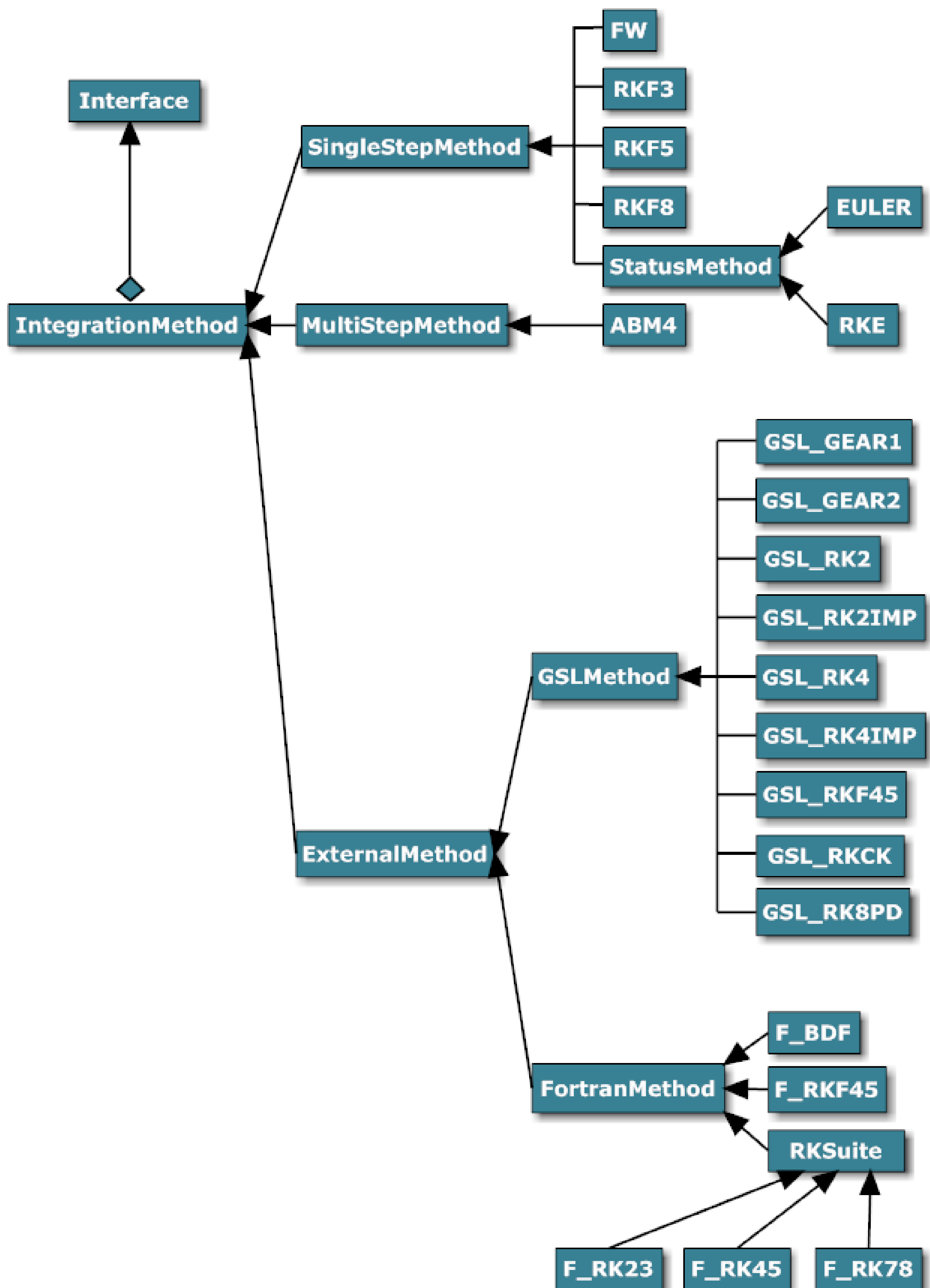
3.3.3 Třída FortranMethod a její potomci

Třída FortranMethod je abstraktní bázovou třídou pro externí fortranovské metody. Sdružuje podmnožinu externích metod, které sice mají rozdílné rozhraní, ale stejně je spojují určité podobné vlastnosti. Oproti třídě ExternalMethod si její instance navíc uchovávají čas posledního kroku numerické integrace, díky čemuž se v případě opakování simulace, kdy se čas vrací zpět do počátečního stavu, dokáží resetovat a bez problémů opakují výpočet znovu.

Jednotliví potomci této třídy potom představují adaptéry na rozhraní fortranovských knihoven provádějících numerickou integraci. Každý z nich je svojí povahou unikátní, protože se nastavení pro jednotlivé knihovny značně liší. Jedinou výjimku tvoří třída RKSuite, která je adaptérem pro skupinu integračních metod stejnojmenné knihovny.

3.4 Výsledný podsystém numerické integrace

Výsledkem všech předcházejících dílčích úprav je nový podsystém numerické integrace. Z velké části se opírá o původní hierarchii tříd, kterou dále rozšiřuje o možnost přidání externích metod. Nový návrh a použití návrhových vzorů také umožnilo oddělení rozhraní numerické integrace knihovny SIMLIB od její implementace. To vede k přehlednějšímu, lépe strukturovanému a snáze udržovatelnému kódu. Výsledná hierarchie tříd nového podsystému je pak zobrazena na obrázku 3.5.



Obrázek 3.5: Hierarchie tříd nového podsystému numerické integrace

4 Implementace

Na základě podrobné analýzy jsem provedl implementaci nového návrhu podsystému numerické integrace knihovny SIMLIB v jazyce C++. Dbal jsem při tom především na efektivitu a korektnost výsledného kódu. Následující kapitoly budou popisovat některé detaily a komplikace, se kterými jsem se při implementaci setkal.

4.1 Numerická integace v knihovně GSL

Je nutné poznamenat, že koncept numerické integrace je v GSL velmi propracovaný a skládá se z několika oddělených částí. Nejdříve je nutné nadefinovat vlastnosti ODE systému, se kterým budeme pracovat. V GSL je reprezentovaný datovým typem *gsl_odeiv_system*, který zahrnuje počet rovnic v systému a funkci pro vyčíslení jejich pravých stran.

Nízkoúrovňovým prvkem, jenž pracuje nad tímto systémem, je tzv. *krokovací funkce*, která zároveň vyjadřuje, jaká metoda bude k numerické integraci použita. Jejím úkolem je výpočet výstupních hodnot integrátorů a chyby metody po jednom kroce numerické integrace. K dispozici jsou metody Runge-Kutta řádu 2, 4 a jejich implicitní varianty, Runge-Kutta-Fehlberg řádu 5, Runge-Kutta Cash-Karp řádu 5 a Runge-Kutta Dormand-Prince řádu 8. Dále jsou přítomné také Gearovy metody, které se vyznačují především svojí stabilitou.

Současným trendem v oblasti numerických integrací je proměnlivá délka kroku, kterou samozřejmě umožňuje i GSL. Na základě změny výstupu integrátoru vypočítané krokovací funkcí se *kontrolní funkce* snaží určit optimální velikost kroku podle zvolené přesnosti. Algoritmus pro úpravu velikosti kroku pak začíná výpočtem požadované úrovně chyby D_i pro každý integrátor podle vozrce:

$$D_i = \epsilon_{\text{abs}} + \epsilon_{rel} |y_i| \quad , \tag{4.1}$$

kde ε_{abs} je absolutní chyba, ε_{rel} je relativní chyba a y_i je vypočtený výstup integrátoru. Ta je porovnána s chybou E_i vypočítanou krokovací funkcí. Pokud chyba E_i kteréhokoliv integrátoru překročí požadovanou úroveň o více jak 10%, pak musí metoda vhodným způsobem zredukovat velikost kroku. Toho dosáhne pomocí vzorce

$$h_{new} = h_{old} S (E/D)^{(-1/q)} \quad , \tag{4.2}$$

kde q odpovídá řádu použité metody(např. $q=2$ pro metodu Runge-Kutta druhého řádu), S je bezpečnostní faktor a poměr E/D je maximem z jednotlivých poměrů E_i/D_i. Opačná situace nastává v případě, že chyba E je pro všechny integrátory nejméně poloviční ve srovnání s požadovanou úrovní D. Velikost kroku je potom vhodné zvětšit tak, aby výsledná chyba odpovídala našim požadavkům.

Pro výpočet nového kroku se použije vztah:

$$h_{new}=h_{old} S(E/D)^{(-1/(q+1))} \tag{4.3}$$

Poslední částí systému GSL pro řešení ODE je *evoluční algoritmus*. Slouží jako řídící entita nad výše zmíněnými složkami systému a kombinuje jejich výsledky tak, aby z počátečního času *t0* po jednotlivých krocích postupovala až ke koncovému času *t1*. Pokud kontrolní funkce signalizuje, že krok výpočtu je příliš velký na to, aby byla dosažena požadovaná přesnost, evoluční algoritmus se vrátí zpět na začátek aktuálního kroku integrace a spustí krokovací funkci s doporučenou velikostí zmenšeného kroku. Tento proces se cyklicky opakuje, dokud není nalezena přijatelná velikost kroku.

Co se týče vlastností jednotlivých krokových metod, odpovídají teoretickému popisu uvedenému v kapitole 2. Rychlost metod je srovnatelná s metodami implemenotovanými v SIMLIBu a většina metod si po dlouhou dobu(vzhledem ke svému řádu) drží solidní přesnost. Jediný problém jsem zaznamenal u explicitní metody Runge-Kutta čtvrtého řádu, která v poměrně krátkém časovém intervalu začala produkovat nepřesné výsledky. Naopak mě mile překvapila metoda Runge-Kutta druhého řádu, která si velice dobře poradila i s řešením rovnic s větší přesností, než je její primární účel.

4.2 Externí metody v jazyce Fortran

Fortran je imperativní programovací jazyk určený především pro vědecké výpočty a numerické aplikace. Je známý tím, že má velmi výkonné překladače, takže je jeho výsledný kód rychlý a efektivní, což byla také myšlenka, která mě přivedla k použití metod napsaných v tomto jazyce. Užití metod v jiném jazyce než je C nebo C++ však přináší celou řadu nových problémů, ať už se jedná o kompatibilitu typů nebo správu dynamické paměti. Naštěstí není kombinace Fortranu s C nebo C++ nijak neobvyklým jevem a je dostatek publikací[6,7], které o ní pojednávají.

4.2.1 Smíšené programování C++ a Fortran

Všechny poznatky uvedené v této kapitole jsou vázány na použití volně dostupného kompilátoru g77 a u některých jiných kompilátorů se mohou lišit podle jejich nastavení(např. konvence pro pojmenovávání podprogramů u kompilátoru f77). Aby bylo možné použít prostředky poskytované jednotlivými knihovnami v jazyce Fortran, je nutné se seznámit alespoň se základy smíšeného programování.

Rozhraní fortranovských knihoven je tvořeno podprogramy pro nastavení metody, provedení kroku metody a případný reset metody. Ostatní části, jako je např. volba velikosti kroku, si řeší samotná metoda. Proto abychom mohli tyto podprogramy používat, je třeba importovat jejich deklaraci ve formě srozumitelné jazyku C++. Toho dosáhneme pomocí konstrukce:

extern “C“

{deklarace funkcí reprezentujících fortranovské podprogramy...},

kde “C“ může být libovolný literál popisující jazyk, který obsahuje požadovaný zdroj a bude použit při linkování. Pro jazyk Fortran se standardně používá “C“, ale není vyloučeno ani použití literálu “Fortran“.

Každý podprogram pak musíme vyjádřit pomocí funkce, jejíž název bude shodný s názvem ve Fortranu, ale malými písmeny a s podtržítkem na konci. Tyto funkce budou předávat všechny svoje parametry pomocí ukazatelů(respektive operátorem &), čímž dosáhneme fortranovské obdoby předávání odkazem, protože C++ standardně předává všechny svoje parametry hodnotou(odlišná situace je u polí a struktur). Pro převod jednotlivých parametrů funkce je pak nutné znát kompatibilitu základních datových typů v obou jazycích, kterou ukazuje tabulka 4.1. Při předávání řetězců musí být navíc na konci seznamu parametrů funkce uvedena jejich délka v pořadí, v jakém jsou uvedeny v deklaraci funkce.

Fortran	C++
INTEGER	int
REAL	float
DOUBLE PRECISION	double
LOGICAL	bool
CHARACTER*N	char[N]

Tabulka 4.1: Kompatibilita základních datových typů v jazycích Fortran a C++

Rozdíl je také v indexování polí. Zatímco C(potažmo C++) indexuje od nuly, Fortran indexuje svá pole od jedničky. Proto musíme uvažovat, že hodnoty nacházející se ve Fortranu na určitém indexu pole, budou v C++ na o jedničku nižším indexu. Aby nebylo rozdílů málo, tak u vícerozměrných polí Fortran přistupuje k prvkům v opačném pořadí než C++. Pro přístup k prvku, který by se u dvojrozměrného pole *y* ve Fortranu nacházel na pozici *y(5,7)*, musíme v C++ použít *y[8][6]*.

4.2.2 Vlastnosti implementovaných method v jazyce Fortran

Metoda `f_rkf45` představuje Fehlbergovu variantu Runge-Kuttovy metody pátého řádu. Je vhodná především pro řešení soustavy netuhých a středně tuhých rovnic. Dokáže dlouho držet svoji přesnost, což je ovšem vykoupeno její menší rychlostí. Jedná se o ideální metodu pro řešení obecných problémů. Bez problémů si poradí s časovým posunem a skokovou změnou hodnot integrátorů.

Metody obsažené v balíku RKSuite jsou, jak už napovídá název, různými variantami klasických Runge-Kuttových metod, které slouží k řešení systému obyčejných diferenciálních rovnic prvního řádu. Pro jejich použití nabízí knihovna dva typy úloh reprezentované zkratkami UT a CT.

UT je odvozeno od anglického *usual task* a slouží k získání přibližných hodnot výstupů integrátorů v sadě bodů. Bude to také úloha, jejíž použití budeme od knihovny vyžadovat. Naopak CT(z anglického *complicated task*) slouží k řešení složitějších úloh, které vyžadují přesnou kontrolu průběhu integrace. Pro dosažení vyšší přesnosti knihovna nabízí výpočet globální chyby, jeho použití ale sníží rychlost vybrané metody přibližně na polovinu, takže jsem ji nevyužil. Co se týče uplatnění jednotlivých metod, tak rozsah přesnosti pro jejich efektivní použití je uveden v tabulce 4.2. Pro řešení tuhých systémů není vhodné metody používat, ale v nezbytném případě autoři doporučují použít metodu `f_rk23`, která si dokáže z vybraných metod poradit nejlépe.

Velký problém jsem zaznamenal v případě, kdy metody této knihovny musely vlivem stavové události zkrátit velikost kroku a provést znovu krok integrace. Tyto metody si totiž udržují čas posledního úspěšně provedeného kroku ve svém vnitřním stavu a opakovaný výpočet kroku považují za chybový stav. Situaci jsem vyřešil resetováním metody, čímž se smazal její vnitřní stav, který bránil opětovnému vyčíslení pro daný časový interval.

Přesnost	Nejefektivnější metoda
10^{-2} -- 10^{-4}	f_rk23
10^{-3} -- 10^{-6}	f_rk45
10^{-5} a více	f_rk78

Tabulka 4.2: Volba metody knihovny RKSuite vzhledem k požadované přesnosti

Z poslední použité knihovny VODE jsem si vybral metodu pro řešení tuhých systémů `f_bdf`, založenou na BDF(z anglického *backward differentiation formulae*), což je vícekroková metoda využívající zpětné diference. Tato metoda potřebuje k výpočtu matici jakobiánů, kterou musí dodat buď uživatel, nebo si ji vypočítá systém sám. Protože jsem si nebyl jistý rychlostí jednotlivých přístupů, tak jsem pro zjednodušení zvolil druhou možnost. Přesto metoda ukázala při řešení tuhých systému neobvyklou rychlost, takže se tento faktor neukázal být klíčovým.

I přes svou specializaci se metoda velmi dobře uplatňovala při řešení netuhých systémů a ukázala tak svou univerzálnost. Nicméně ani při její implementaci jsem se nevyhnul problémům. Jako jediná si tato metoda při prvním volání uloží do pomocných pamětí vstupní hodnoty integrátorů, se kterými pracuje po celý zbytek numerické integrace. To se úkázalo jako problematické v případě skokové změny hodnoty integrátorů při stavové události. Jako jedno z možných řešení se nabízel reset metody po každé diskrétní události, to by ale vedlo ke ztrátě její efektivity(např. vlivem častého

vzorkování hodnot). Proto jsem zavedl do SIMLIBu příznak SIMLIB_ChangeState, který se nastaví v případě provedení stavové události, která by mohla vést ke změně hodnot integátorů. Pokud je tento příznak nastaven, metoda f_bdf se resetuje a načte si do vnitřního stavu aktuální hodnoty integrátorů.

4.3 Zhodnocení výsledné implementace

Zamýšlená úprava podsystému numerických metod si vyžádala přidání asi 1700 řádek kódu. Také bylo nutné provést úpravy na původním podsystému, ale jejich počet je v porovnání s množstvím nového kódu zanedbatelný.

Při implementaci jsem se zaměřil především na efektivitu výsledného kódu. Dosažení větší efektivity zhatila nemožnost externích metod používat přímý přístup k hodnotám integrátorů. V každém kroce se tak kopírují aktuální hodnoty vstupů integrátorů do pomocného pole, které metody používají. Obdobně se na konci kroku kopírují získané výstupní hodnoty integrátorů zpět do vektorů, které uchovávají jejich hodnoty. V tomto směru externí metody oproti interním ztrácí. Tuto nevýhodu ale nahrazují tím, že během jednoho jejich volání provedou několik kroků numerické integrace za sebou až po další diskrétní událost. Jak je vidět z výsledků testování, tak jejich výkon je srovnatelný s metodami v SIMLIBu, v některých případech jsou dokonce výkonnější.

Veškerý implementovaný kód také odpovídá normě C++, takže je bez problémů přenositelný mezi jednotlivými architekturami. K jeho snazší upravitelnosti také přispívá kvalitní návrh podporovaný použitím návrhových vzorů.

5 Testování

Pro testování jsem použil všechny příklady spojité a kombinované simulace uvedené v knihovně SIMLIB. Navíc jsem se rozhodl popsat detailněji několik zástupců typických problémů, se kterými se při spojité simulaci setkáme. Všechny testovací příklady mají svou vlastní složku na přiloženém CD v adresáři **test**. Měření jsem prováděl na počítači Pentium M 1,6 GHz, 512MB RAM na systému LINUX.

5.1 Kruhový test

Jedná se o problém, který řeší jednoduchou diferenciální rovnici:

$$y''(t)+ky(t)=0 \qquad y(0)=0, y'(0)=|k| \tag{5.1}$$

Rovnici budeme řešit s parametrem *k=1*, pro který má rovnice analytické řešení:

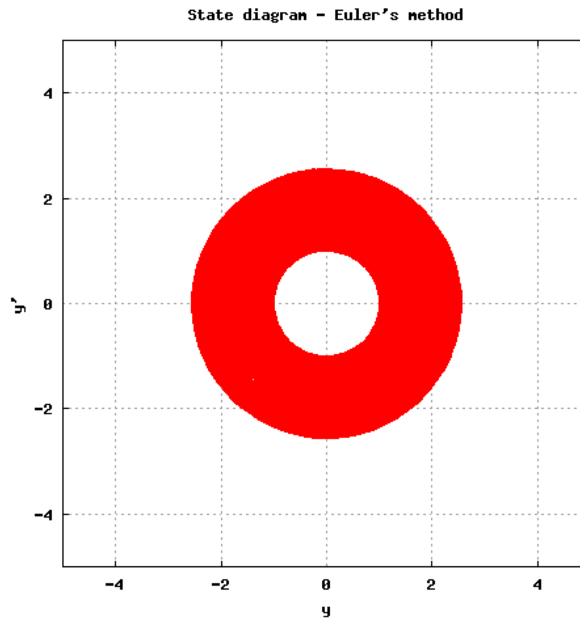
$$y=\sin(t) \tag{5.2}$$

Pro fázový diagram řešení a jeho derivace platí, že se jedná o kružnici. Vzhledem k dostatečné délce simulace(od 0 do 10000π) je na fázovém diagramu patrná přesnost metody v podobě vznikajících prstenců u méně přesných metod, viz obrázek 5.1. Rychlost jednotlivých metod pak ukazuje tabulka 5.1.

metoda	čas[s]	metoda	čas[s]	metoda	čas[s]
rkf8	0.60	gsl_rk8pd	0.60	f_rkf45	1.25
rkf5	1.02	gsl_rkck	1.08	f_rk78	1.64
rke	1.36	gsl_rkf45	1.52	f_rk45	2.31
abm4	2.19	gsl_rk4	2.62	f_bdf	2.78
rkf3	4.56	gsl_rk4imp	3.95	f_rk23	14.56
fw	17.47	gsl_gear2	4.95		
euler	157.32	gsl_rk2	11.21		
		gsl_rk2imp	20.00		
		gsl_gear1	214.08		

Tabulka 5.1: Rychlost metod při řešení kruhového testu

Tento test ukázal, jak si jednotlivé metody stojí při řešení jednoduchých diferenciálních rovnic. Velkou rychlost ukázaly především metody vyšších a středních řádů, které využily možnost protáhnout si krok.



Obrázek 5.1: Prstenec u nepřesného řešení Eulerovou metodou

5.2 Van der Polův oscilátor

Tento příklad jsem převzal ze zdroje[9]. Jedná se o představitele tuhého systému popsáného rovnicí:

$$y'' - u(1-y^2)y' + y = 0 \qquad y(0)=0, y'(0)=1 \quad , \tag{5.3}$$

kde parametr u určuje koeficient tuhosti systému. Pro tento konkrétní test jsem použil koeficient 40000, který by měl donutit klasické metody k tomu, aby rapidně snížily velikost kroku. Naopak metody určené pro tuhé systémy, jako je Fowler-Wartenova metoda, by si díky svojí specializaci měly poradit hravě.

metoda	čas[s]	metoda	čas[s]	metoda	čas[s]
rkf3	0.95	gsl_rk2	1.39	f_bdf	0.05
fw	1.07	gsl_rkck	1.44	f_rkf45	1.25
rkf5	1.21	gsl_rkf45	1.46	f_rk23	2.90
rke	1.24	gsl_rk4	1.86	f_rk45	3.26
abm4	5.08	gsl_rk8pd	2.17	f_rk78	3.34
rkf8	12.7	gsl_rk4imp	3.87		
euler	14.8	gsl_gear2	3.92		
		gsl_gear1	4.18		
		gsl_rk2imp	4.32		

Tabulka 5.2: Efektivita jednotlivých metod pro Van der Polův oscilátor s koeficientem tuhosti 40000

Skvělé výsledky prokázala metoda `f_bdf`, která ukázala svou silnou stránku, když byla až 18x rychlejší než další metoda v pořadí. Naopak velké zklamání přišlo u metody Fowler-Warten, která byla dokonce pomalejší než Fehlbergova varianta Ruge-Kuttovy metody třetího řádu. Obecně lze říci, že z explicitních Runge-Kuttových metod si vedly lépe metody nižších řádů, které se dokáží vypořádat s menší délkou kroku lépe než metody vyššího řádu.

5.3 Kmitání struny

Tento příklad jsem převzal ze zdroje[2]. Jedná se o test výkonnosti numerické integrace modelu s velkým počtem integrátorů. Kmitání struny popisuje parciální diferenciální rovnice definující dynamiku systému:

$$\frac{\partial^2 y}{\partial t^2} = a \frac{\partial^2 y}{\partial x^2} \tag{5.4}$$

s počátečními podmínkami určujícími počáteční polohu struny a rychlost na jejich koncích:

$$y(x,0) = -\frac{4}{l^2} x^2 + \frac{4}{l} x \tag{5.5}$$

$$y'(x,0) = 0 \tag{5.6}$$

Okrajové podmíky pak definují upevnění obou konců struny:

$$y(0,t) = y(l,t) = 0 \tag{5.7}$$

Protože knihovna SIMLIB neposkytuje přímo prostředky pro řešení parciálních diferenciálních rovnic, bude nutné ji užitím vhodné metody převést na systém obyčejných diferenciálních rovnic. Metodou přímek jsme strunu rozdělili na dostatečně velký počet úseků o velikosti Δx a všechny prostorové derivace v každém bodě struny jsme nahradili x_i diferencemi:

$$\frac{\partial^2 y}{\partial x^2} | x_i = \frac{y_{i+1} - 2y_i + y_{i-1}}{\Delta x^2} \tag{5.8}$$

Touto diskretizací dostaneme soustavu obyčejných diferenciálních rovnic, kterou již umíme řešit. Pro každý úsek struny na pozici x_i pak platí rovnice popisující výchylku y_i:

$$\frac{\partial^2 y_i}{\partial t^2} = a \frac{y_{i+1} - 2y_i + y_{i-1}}{\Delta x^2} \tag{5.9}$$

Z následujícího testu jsem vyloučil metodu Eulerovou, Fowler-Wartenovu a první variantu Gearovy metody, které se svojí povahou k řešení tohoto problému nehodí. Strunu jsem rozdělil na 1000 dílků, což odpovídá počtu 2002 integrátorů. Metody jsem tak při relativně krátké době simulace donutil k velkému množství výpočtů, aby bylo možné lépe porovnat jejich výkon při řešení této úlohy.

metoda	čas[s]	metoda	čas[s]	metoda	čas[s]
rkf3	12.13	gsl_rk8pd	9.17	f_rk23	14.04
rkf5	36.57	gsl_rkck	15.40	f_rk78	26.16
rke	51.82	gsl_rkf45	19.40	f_bdf	26.80
abm4	78.21	gsl_rk4	23.72	f_rk45	33.85
rkf8	138.20	gsl_rk2	28.43	f_rkf45	38.45
		gsl_rk4imp	52.28		
		gsl_gear2	82.21		
		gsl_rk2imp	120.58		

Tabulka 5.3: Výkonnost metod při simulaci modelu kmitání struny

V tomto případě se nedalo přesně rozhodnout, jaký typ metod je vhodnější pro řešení systému s velkým počtem integrátorů. Zatímco u interních SIMLIBovských metod na tom byly lépe metody nižších řádů, z externích metod knihovny GSL dosahovaly rychleji výsledku metody kvalitnější. Nakonec u metod v jazyce Fortran úplně „vyhořely“ univerzální metody středního řádu, které v porovnání s ostatními fortranovskými metodami vykazovaly horší výsledky.

6 Závěr

Při návrhu a implementaci úprav podsystému numerické integrace v SIMLIBu jsem dbal na to, abych minimálním počtem změn dosáhl možnosti připojit a používat externí integrační metody. Zároveň jsem se pokusil o to, aby došlo k oddělení kódu důležitého z hlediska uživatele(respektive modelu) od výkonného kódu samotné knihovny. Nutným vyústěním této snahy bylo oddělení rozhraní podsystému numerické integrace od její implementace. Tato úprava by měla v budoucnosti vést k větší přehlednosti a snazší upravitelnosti.

Knihovnu SIMLIB jsem rozšířil o 14 externích metod, které jsem rozdělil do dvou nezávislých modulů, o jejichž instalaci se rozhoduje uživatel při překladu knihovny. Toto rozšíření s sebou přineslo spoustu nových a zajimavých integračních metod a větší volnost pro uživatele, který má nyní k dispozici širší škálu metod pro jeho model. Za obzvášťě velký přínos považuji přidání Dormand-Princeovy varianty Runge-Kuttovy metody, která vykazovala skvělé výsledky ve všech prováděných testech. Po výpadku Fowler-Wartenovy metody, která při testech tuhých systémů nevykazovala žádnou výhodu oproti normálním metodám, se mi podařilo najít vhodnou náhradu, která se objevila v podobě metody zpětných diferencí, jež se v této oblasti osvědčila na výbornou.

Knihovna SIMLIB nabízí v oblasti numerické integrace ještě dostatek prostoru pro další rozvoj. O pomyslný krůček dál se posunula například v oblasti řešení tuhých systémů, ale stále je v řešení této problematiky teprve na začátku cesty. Dalším možným směrem vývoje by bylo vytvoření ucelené sady testů s jejich podrobným popisem, účelem a očekávanými hodnotami, která by jasně vymezovala oblast použití jednotlivých integračních metod. Cesta se otvírá také pro implementaci metod řešících parciální derivace.

Literatura

[1] *SIMLIB/C++ homepage*[online]. Dokument dostupný na URL http://www.fit.vutbr.cz/~peringer/SIMLIB/ (duben 2008)

[2] Peringer, P. *Modelování a simulace,* studijní opora, VUT Brno, 2006

[3] Leška, D. *Objektově orientovaný přístup k numerickým metodám,* diplomová práce, VUT Brno, 1997

[4] Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Návrh programů pomocí vzorů,* Praha, Grada 2003

[5] *GSL homepage*[online]. Dokument dostupný na URL http://www.gnu.org/software/gsl/ (duben 2008)

[6] *NETLIB ODE solvers*[online]. Knihovna dostupná na URL http://www.netlib.org/ode/ (duben 2008)

[7] Kochhar, R. *Fortran and C/C++ Mixed Programming*[online], Department of Physiology: Univeristy of Wisconsin - Madison, Feb. 9.1999. Dokument dostupný na URL http://www.physiology.wisc.edu/comp/docs/ (duben 2008)

[8] Nelson H. F. Beebe *Using C and C++ with Fortran*[online], Department of Mathematics – University of Utah, 2001. Dokument dostupný na URL http://www.math.utah.edu/software/c-with-fortran.html (duben 2008)

[9] Cartwright, J. *The Dynamics of Runge-Kutta Methods – Stiff Problems*[online], 1995. Dokument dostupný na URL http://lec.ugr.es/~julyan/papers/rkpaper/node6.html (duben 2008)

[10] *Runge–Kutta methods - Wikipedia, the free encyclopedia*[online]. Dokument dostupný na URL http://en.wikipedia.org/wiki/Runge-Kutta_method (duben 2008)

[11] *Linear multistep method - Wikipedia, the free encyclopedia*[online]. Dokument dostupný na URL http://en.wikipedia.org/wiki/Linear_multistep_method (duben 2008)

Seznam příloh

Příloha 1. Překlad a instalace SIMLIB

Příloha 2. CD se zdrojovými soubory bakalářské práce

Příloha 1: Překlad a instalace SIMLIB

Jak již bylo zmíněno, externí metody, které jsem do knihovny SIMLIB přidal, jsem rozdělil do dvou samostatných modulů - **fortran** a **gsl**. O jejich připojení do knihovny se rozhoduje na základě parametrů překladu. Oba moduly vyžadují pro úspěšný překlad přítomnost některých podpůrných knihoven. Modul **gsl** je závislý na přítomnosti stejnojmenné knihovny GSL v systému. Modul **fortran** vyžaduje přítomnost překladače jazyka Fortran(v ideálním případě g77) a instalaci podpůrné knihovny, kterou nabízí SIMLIB.

Systém pro překlad a instalaci knihovny SIMLIB je tvořen sadou souborů Makefile, které na základě parametrů vytvoří uživateli knihovnu přesně podle jeho požadavků. Pro spuštění překladu se používá příkaz make, jehož chování je ovlivněno parametry, které jsou mu dodány. Bez přídavných parametrů přeloží knihovnu SIMLIB v základní verzi, tj. bez přídavných modulů. Seznam parametrů ovlivňujících jeho činnost je pak následující:

- ***all*** [1] - přeloží knihovnu se všemi přídavnými moduly.
- ***test*** [1,2] - provede základní test funkcionality knihovny.
- ***add-support-lib*** - přidá do systému pomocnou fortranovskou knihovnu lfsimlib. Tato knihovna je nezbytná pro správný běh metod modulu fortran.
- ***add-support-lib64*** - přidá do systému 64-bitovou verzi pomocné fortranovské knihovny.
- ***rem-support-lib*** - odstraní pomocnou fortranovskou knihovnu ze systému.
- ***clean*** - smaže veškeré soubory vytvořené při překladu.
- ***install*** - nainstaluje přeloženou knihovnu do systému.
- ***uninstall*** - odinstaluje knihovnu včetně pomocných knihoven ze systému.
- ***MODULES=" gsl fortran "*** - přeloží knihovnu s moduly obsaženými v uvozovkách.
- ***32*** - vynucený 32-bitový překlad.
- ***64*** - vynucený 64-bitový překlad.

Pro použití externích metod z modulů **fortran** a **gsl** je nutné při překladu modelu přilinkovat kromě samotné knihovny SIMLIB ještě podpůrné knihovny pro používané moduly. Modul **gsl** vyžaduje připojení knihoven *lgsl* a *lgslcblas*. Modul fortran potom knihovny *lg2c* a *lfsimlib*.

1 Parametry pro překlad a testování je možné rozšířit parametry 32 nebo 64 pro X-bitový překlad

2 Pro otestování knihovny s přídavnými moduly je nutné je uvést jako další parametr v podobě MODULES=" seznam modulů "