



Modelování a simulace

IMS

Studijní opora

Petr Peringer

17. prosince 2012

Tento učební text vznikl za podpory projektu "Zvýšení konkurenceschopnosti IT odborníků – absolventů pro Evropský trh práce", reg. č. CZ.04.1.03/3.2.15.1/0003. Tento projekt je spolufinancován Evropským sociálním fondem a státním rozpočtem České republiky.

Obsah

1	Úvod	4
1.1	Metodické informace	4
1.2	Terminologie	5
1.3	Další zdroje informací	5
2	Modelování a simulace systémů	6
2.1	Základní pojmy	6
2.2	Princip modelování a simulace	7
2.3	Simulace a získávání znalostí	8
2.4	Oblasti použití simulace	8
2.5	Výhody simulačních metod	9
2.6	Problémy simulačních metod	9
2.7	Alternativní přístup – analytické řešení	10
2.8	Kdy je vhodné použít simulační metody	10
2.9	Základní etapy modelování a simulace	11
2.10	Vztahy mezi modely dynamických systémů	12
2.11	Klasifikace modelů	12
2.12	Simulační modely	13
2.13	Simulace	14
2.14	Verifikace a validace modelu	15
2.15	Simulační nástroje – přehled	16
2.16	Shrnutí	17
2.17	Otázky a úkoly	17
3	Modelování náhodných procesů	18
3.1	Generování pseudonáhodných čísel	18
3.2	Příklady generátorů	19
3.3	Transformace rozložení	21
3.3.1	Metoda inverzní transformace	23
3.3.2	Vylučovací metoda	24
3.3.3	Kompoziční metoda	24
3.4	Testování generátorů (pseudo)náhodných čísel	24
3.5	Shrnutí	25
3.6	Otázky, úkoly	25
4	Metoda Monte Carlo	26
4.1	Princip metody Monte Carlo	26
4.2	Přesnost metody Monte Carlo	27
4.3	Typická použití metody Monte Carlo	28
4.4	Implementace metody Monte Carlo	29
4.5	Shrnutí	29

4.6	Otázky a úkoly	29
5	Diskrétní simulace	31
5.1	Formy popisu diskretních systémů	31
5.1.1	Popis paralelismu	32
5.1.2	Události	32
5.1.3	Procesy	32
5.2	Systémy hromadné obsluhy	33
5.2.1	Vstupní tok požadavků	33
5.2.2	Fronty čekajících požadavků	34
5.2.3	Priority	34
5.2.4	Prioritní obsluha	35
5.2.5	Kendallova klasifikace SHO	35
5.3	Modelování systémů hromadné obsluhy	36
5.3.1	Typy obslužných linek	36
5.3.2	Popis chování transakcí	37
5.3.3	Příklad modelu SHO	38
5.4	Kalendář událostí, algoritmus řízení simulace	39
5.5	Přehled diskretní části SIMLIB/C++	40
5.5.1	Obecná struktura modelu	41
5.5.2	Popis simulačního experimentu	41
5.5.3	Modelový čas	41
5.5.4	Generátory pseudonáhodných čísel	42
5.5.5	Popis události	42
5.5.6	Příchod a odchod transakce	42
5.5.7	Popis procesu	43
5.5.8	Základní operace procesu	44
5.5.9	Priorita procesu	44
5.5.10	Fronty (třída Queue)	45
5.5.11	Zařízení (Facility)	46
5.5.12	Sklad (Store)	47
5.5.13	Příklady různých použití zařízení	48
5.5.14	Zpracování výsledků, statistiky	49
5.5.15	Příklad systému hromadné obsluhy – samoobsluha	53
5.6	Shrnutí	55
5.7	Otázky a úkoly	55
6	Spojité simulace	56
6.1	Aplikační oblasti spojité simulace	56
6.2	Formy popisu spojitých systémů	56
6.3	Soustavy obyčejných diferenciálních rovnic	57
6.4	Grafový popis s použitím bloků	58
6.4.1	Typy spojitých bloků	58
6.5	Převod rovnic vyššího řádu na soustavu rovnic 1. řádu	59
6.5.1	Metoda snižování řádu derivace	59
6.5.2	Metoda postupné integrace	60
6.6	Numerické metody	63
6.6.1	Metody pro řešení ODR prvního řádu	63
6.6.2	Princip a klasifikace numerických integračních metod	63
6.6.3	Jednokrokové metody	64
6.6.4	Eulerova metoda	64
6.6.5	Metody Runge-Kutta	66

6.6.6	Vícekrokové metody	67
6.6.7	Vlastnosti integračních metod	68
6.6.8	Tuhé systémy	69
6.6.9	Výběr integrační metody	70
6.7	Příklad: Systém dravec–kořist	70
6.8	Spojité simulační jazyky	73
6.8.1	Algoritmus řízení spojitě simulace	73
6.8.2	Pořadí vyhodnocování funkčních bloků	73
6.8.3	Algoritmus pro seřazení funkčních bloků	74
6.8.4	Rychlé smyčky	74
6.9	Parciální diferenciální rovnice	76
6.10	Použití SIMLIB/C++ pro spojitou simulaci	77
6.10.1	Blokové výrazy	77
6.10.2	Typy bloků v SIMLIB/C++	78
6.10.3	Popis experimentu	78
6.10.4	Příklad více experimentů v SIMLIB/C++	79
6.11	Shrnutí	81
6.12	Otázky a úkoly	82
7	Kombinovaná simulace	83
7.1	Stavové podmínky a stavové události	83
7.1.1	Problémy detekce změn stavových podmínek	84
7.1.2	Stavové podmínky v SIMLIB/C++	84
7.2	Algoritmus řízení kombinované simulace	85
7.3	Příklad modelu: skákající míček	85
7.4	Otázky a úkoly	86
8	Celulární automaty	88
8.1	Definice CA	88
8.2	Typy okolí	89
8.3	Pravidla CA	89
8.3.1	Reverzibilní automaty	90
8.4	Klasifikace CA	90
8.5	Příklad CA	90
8.6	Otázky a úkoly	91
9	Závěr	92



Kapitola 1

Úvod

Tento pomocný učební text je určen pro předmět "Modelování a simulace" (IMS) v rámci bakalářského studia na FIT VUT v Brně. Předmět IMS studenti absolvují ve třetím ročníku. Styl textu je stručný, podrobnosti je třeba hledat v literatuře, která je uvedena na konci textu. Text je zaměřen na následující oblasti:

- Principy modelování a simulace
- Generování a testování pseudonáhodných čísel
- Metoda Monte Carlo
- Diskrétní simulace, systémy hromadné obsluhy
- Spojitá simulace, numerické metody

Některé části textu jsou označeny piktogramy, které čtenáři signalizují důležitost jednotlivých partií textu a další údaje.

	Čas potřebný pro studium		Otázka, příklad k řešení
	Cíl		Počítačové cvičení, příklad
	Definice		Příklad
	Důležitá část		Reference
	Rozšiřující látka		Správné řešení
	Obtížná část		Souhrn
			Slovo tutora, komentář
			Zajímavé místo

Tabulka 1.1: Význam používaných piktogramů

1.1 Metodické informace

Předmět *Modelování a simulace* navazuje na celou řadu předmětů bakalářského studia. Vzhledem k zaměření tohoto textu jsou záměrně vynechány složitější matematické formulace. Přesnější matematický základ získá čtenář v případě potřeby studiem doplňkové literatury.

Nejdůležitější pro pochopení funkce simulačních systémů je absolvování programátorských předmětů (především jazyky C a C++, ale postačující je znalost programování v jakémkoli netriviálním programovacím jazyku). Dále předpokládáme základní znalosti numerické matematiky, pravděpodobnosti a statistiky. Pro řešení některých příkladů ve spojitě simulaci je vhodné mít základy fyziky a elektrotechniky. Simulace je interdisciplinární obor, proto lze uplatnit i znalosti z jiných oblastí vědy (biologie, chemie, astronomie, atd.).

Z uvedeného přehledu je patrné, že jeden z cílů předmětu je shrnout znalosti z několika oborů a naučit studenty vytvářet počítačové modely systémů a provádět s nimi simulační experimenty. Tento experimentální charakter simulace je významný pro pochopení celé řady souvislostí – zkoušet ”co se stane když” je významnou součástí výuky.

1.2 Terminologie

Velmi důležitou součástí výuky je správné používání odborné terminologie. V textu bude použita terminologie zavedená v rámci simulační komunity v České republice. Tato terminologie bude doplněna anglickou terminologií, protože internetové zdroje informací jsou obvykle v angličtině.

1.3 Další zdroje informací

Osnovu předmětu a aktuální informace včetně odkazů na další zdroje informací najdete na oficiální WWW stránce předmětu IMS

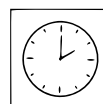
<http://www.fit.vutbr.cz/study/courses/IMS/>

speciálně v odkazech ”Informace veřejné” a ”Informace pro zapsané”.

Vybrané odkazy na doplňkové zdroje informací jsou uvedeny také na konci tohoto textu v sekci Literatura. Většina kvalitních knih a článků je v angličtině. Významným zdrojem informací mohou být také zdrojové texty některých volně dostupných programů, jejich dokumentace a příklady.

Kapitola 2

Modelování a simulace systémů



2:30

V této kapitole se zaměříme na stručný přehled základních principů, metod a nástrojů používaných v oboru modelování a simulace na číslicových počítačích. Obsah této kapitoly lze shrnout do následujících bodů: základní pojmy a princip simulace, souvislosti s ostatními obory, oblasti použití simulačních metod, výhody použití simulace, problémy vznikající při modelování a simulaci, alternativní postup – analytické řešení modelů a velmi stručný přehled použitelných simulačních nástrojů, převážně těch volně dostupných.

Tento přehled je nutný pro pochopení základů oboru modelování a simulace, jeho zvládnutí by mělo usnadnit další studium.

2.1 Základní pojmy

Pochopení principů modelování a simulace vyžaduje alespoň neformální definici několika základních pojmů se kterými budeme pracovat v dalším textu:

DEF

- **Systém** můžeme obecně definovat jako soubor elementárních částí (prvků systému), které mají mezi sebou určité vazby (propojení prvků).

Systémy je možné rozdělit do několika kategorií podle různých kritérií. Například podle existence můžeme systémy dělit na:

- reálné (existující) systémy
- nereálné (fiktivní, ještě neexistující) systémy – používají se například v počítačových hrách

nebo podle změn stavu na:

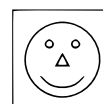
- statické systémy – nemění svůj stav v čase
- dynamické systémy – mění stav v čase

Pro simulaci jsou zajímavé především dynamické systémy. Příkladem jednoduchého dynamického systému může být supermarket se zákazníky, prodavači, zbožím, pokladnami, vozíky atd.

- **Model** je napodobenina systému jiným systémem — například (v našem případě výhradně) počítačovým programem. Model systému musí napodobovat všechny pro naše účely podstatné vlastnosti systému. Příkladem modelu může být soustava diferenciálních rovnic popisující let rakety nebo její ekvivalent ve tvaru blokového schématu.
- **Modelování** je proces vytváření modelů systémů na základě našich znalostí. Tento proces je obecně velmi náročný a často vyžaduje znalosti z více oborů. Kvalita vytvořeného modelu zásadním způsobem ovlivní výsledky získané experimentováním s modelem.

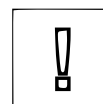
- **Simulace** je metoda získávání nových znalostí o systému experimentováním s jeho modelem. Pro účely simulace musí být model popsán odpovídajícím způsobem – ne každý model je pro simulaci vhodný. V tomto textu se omezíme pouze na simulaci na číslicových počítačích. Pro získání potřebných informací obvykle potřebujeme opakovat simulační experimenty vícekrát s různými parametry.

Podrobnější specifikace a souvislosti těchto pojmů naleznete v následujícím textu. Je také třeba si uvědomit, že některé tyto pojmy jsou využívány i v jiných souvislostech – například nelétající modely letadel používané jako dekorace nebo simulace ve smyslu "Das ganze tschechische Folk ist eine Simulantenbande" mají jiný cíl než získávání nových znalostí.



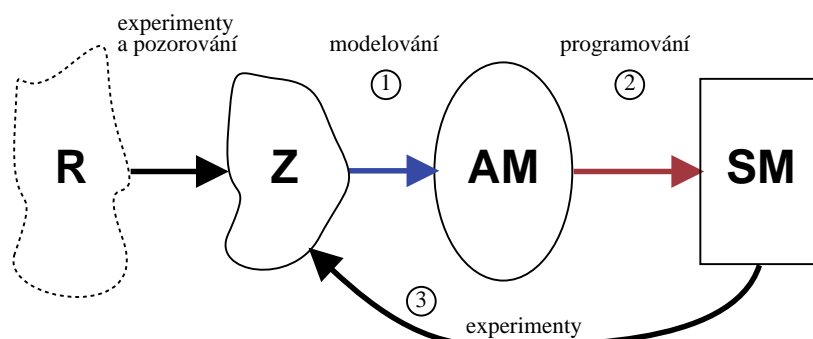
2.2 Princip modelování a simulace

Cílem simulace je získat nové znalosti o zkoumaném systému. Abychom mohli provádět simulaci, je nutné vytvořit vhodný model tohoto systému. Postupujeme tak, že:



1. Nejdříve vytvoříme tzv. *abstraktní model*, který nezahrnuje všechny naše znalosti o modelovaném systému, ale vybíráme jen ty vlastnosti, které jsou pro naše účely podstatné. Tím dosáhneme zjednodušení modelu na zvládnutelnou úroveň. Abstraktní model může mít například formu matematických rovnic.
2. Na základě abstraktního modelu pak vytváříme *simulační model*, který už dále nic nezjednodušuje a musí zahrnovat všechny vlastnosti abstraktního modelu. Rozdíl mezi abstraktním modelem a simulačním modelem je pouze možnost provádění experimentů – simulační model je spustitelný program, který počítá výsledky podle zadaného počátečního stavu, vstupů a parametrů modelu.
3. Se simulačním modelem provádíme simulační experimenty a jejich výsledky analyzujeme. Výsledkem jsou informace o chování systému ze kterých jejich zobecněním získáme nové znalosti.

Následující obrázek 2.1 znázorňuje celý proces získávání znalostí s využitím simulace.



Obrázek 2.1: Realita → Znalosti → Abstraktní Model → Simulační Model

2.3 Simulace a získávání znalostí

Modelování je velmi rozšířenou činností člověka již od počátků jeho existence — obecně lze říci, že naše představa světa je vlastně modelem reality (realita = okolní svět, který můžeme zkoumat). Naše znalosti jsou založeny na experimentálních pozorováních — jsou to zobecněné výsledky mnoha experimentů s reálnými systémy. Například přírodní zákony můžeme považovat za matematické modely chování reálného světa.

Při modelování (tj. vytváření modelu systému) vycházíme z informací o systému, které jsou dostupné (máme je v bázi znalostí). Model, který vznikne, reprezentuje formalizované znalosti o modelovaném systému z hlediska, které chceme zkoumat. Model obvykle pokrývá pouze tu část popisu celého systému, která je pro daný účel podstatná. Protože model vždy vychází z podmnožiny našich znalostí, které jsou neúplné, můžeme modelovat pouze to, co jsme schopni pochopit a popsat. Obecně lze říci, že veškeré naše znalosti světa představují model, který odráží úroveň našeho poznání. Simulační modelování tedy představuje proces transformace znalostí ze strojově neproveditelné reprezentace na reprezentaci proveditelnou na počítači.

Simulační model je tedy taková forma znalostí, která má jako základní charakteristickou vlastnost proveditelnost (tj. lze s ním na počítači provádět simulační experimenty). To jej odlišuje od jiných forem znalostí, které obvykle nejsou přímo použitelné k experimentování na počítačích.

Proces experimentování v reálném světě je vždy zatížen chybami měření a dalšími faktory, které mohou způsobit problémy při interpretaci výsledků. Navíc jsou experimenty s reálnými systémy někdy neekonomické, nebezpečné, nevhodné (například pokusy na zvířatech) nebo vůbec neproveditelné. Proto používáme metod počítačové simulace, která tyto nevýhody nemá.

Při simulačních experimentech se objevují jiné problémy, které znesnadňují jejich použití pro formulaci nových znalostí. V řadě případů je proto nejvhodnější kombinace reálných experimentů se simulačními.

Obecně platí, že musíme neustále konfrontovat znalosti získané oběma způsoby a kontrolovat tak platnost (validitu) modelu. Ověřování platnosti modelu je proces, v němž se snažíme dokázat, že náš model opravdu odpovídá modelovanému systému. Platnost modelu ale nelze dokázat absolutně.

2.4 Oblasti použití simulace

Zde si pro ilustraci uvedeme několik konkrétních příkladů použití simulace a simulačních metod v různých oblastech:

- Biologie a lékařství: model šíření epidemie AIDS, modely působení léků v organismu, modelování růstu bakterií
- Fyzika: model jaderného reaktoru, model šíření zvuku v místnosti
- Chemie: modely chemických reakcí, výpočty vlastností látek
- Astronomie: model srážky galaxií, simulace pohybu planet kolem Slunce
- Meteorologie: modely pro předpověď počasí (výsledky vidíte každý den v televizi)
- Geologie: model zemětřesení
- Technika obecně: simulované crash testy automobilů, model mikroprocesoru, simulace elektrických obvodů, nanotechnologie – chování atomů, ...

- Ekonomika: hromadná obsluha – model supermarketu, modely trhu s akcemi, ...
- Doprava: model dopravní situace ve městě a související model znečištění ovzduší
- Výuka: demonstrační modely, hry (např. simulátor letadla)
- Film, počítačové hry: modely pro různé vizuální efekty

Z uvedeného přehledu vidíme, že použití simulace je velmi rozsáhlé a zasahuje prakticky do všech oblastí. Proto je obvykle při vytváření složitých simulačních modelů nutná spolupráce odborníků z různých oborů. Odkazy na řadu modelů z praxe najdete na WWW stránkách předmětu IMS.

2.5 Výhody simulačních metod

Obecně můžeme za hlavní výhody simulace v porovnání s reálnými experimenty považovat nižší cenu, většinou i menší časovou náročnost experimentování a naprostou bezpečnost při provádění experimentů.

- Cena je důležité kritérium pro nasazení simulace. Experimenty s reálným systémem mohou být velmi drahé – například crash testy automobilů se dnes provádějí pouze pro validaci výsledků simulačních experimentů a proto je spotřeba automobilů při těchto testech výrazně menší.
- Rychlost experimentování je dalším důležitým faktorem pro simulaci. Simulaci můžeme zrychlovat a zpomalovat podle potřeby (zrychlování je samozřejmě omezeno výkonem našeho počítače). Například růst rostlin je velmi pomalý proces, který lze při simulaci výrazně urychlit a dosáhnout výsledků podstatně dříve. Naopak pro sledování velmi rychlých dějů v reálných systémech je třeba velmi drahé experimentální vybavení (nebo dokonce nelze tyto jevy sledovat vůbec), kdežto u simulačního modelu můžeme tyto děje libovolně zpomalit. Příkladem může být sledování pohybu atomů v nanotechnologických systémech.
- Bezpečnost je u některých reálných experimentů velký problém, často nelze tyto experimenty z bezpečnostních důvodů vůbec provádět. Příkladem mohou být některé jaderné reakce, šíření epidemií a podobně. Snad každý si pamatuje výsledek reálných experimentů v Černobylu. Simulační experimenty na počítačích jsou naprosto bezpečné a dovolují provádět cokoli.
- Simulace dovoluje experimentovat s modely velmi složitých systémů – jsme omezeni pouze výkonem našeho počítače. Reálné experimenty by u tak složitých systémů byly neproveditelné.
- Někdy je simulace jediný způsob jak experimentovat – například v astronomii můžeme simulovat srážky galaxií a podobné situace.

Protože výkon počítačů neustále roste, je ekonomicky výhodnější, rychlejší a často jediné možné experimentovat na modelech, než na originálech.

2.6 Problémy simulačních metod

Simulace na počítačích má vedle výhod také závažné nevýhody se kterými musíme počítat a snažit se je minimalizovat. Zde si uvedeme jejich přehled a s detaily se seznámíme až v dalším textu.

- *Problém validity modelu* je nejdůležitější, protože chybný model dává při simulaci chybné výsledky, které, pokud je aplikujeme mohou způsobit katastrofální následky. Ověřování validity modelu je velmi náročné a mělo by být provedeno ještě před prováděním experimentů, jejichž výsledky chceme použít.
- *Náročnost na výpočetní výkon počítačů* je obecný problém všech netriviálních simulačních modelů. Většina dnešních superpočítačů[19] je používána právě pro simulace. Vzhledem k neustálému růstu výkonu procesorů je dnes možné provádět relativně náročné simulace i na běžných osobních počítačích.
- Simulací získáváme *konkrétní numerické výsledky* pro daný experiment. Pokud potřebujeme například zjistit vliv změněných parametrů modelu, musíme celou simulaci opakovat, což může být časově náročné.
- Další nevýhodou je *nepřesnost a nestabilita numerického řešení* některých modelů. Tyto negativní vlastnosti numerických metod mohou zcela znehodnotit výsledky i v případě ověřeného a jinak bezchybného modelu. Proto je nutné znát vlastnosti použitých numerických metod a používat jen ty, které mají pro daný model vyhovující přesnost.
- Někdy je problémem i *vysoká náročnost vytváření modelů* — vytvořit kvalitní model mikroprocesoru může být stejně náročné jako navrhovat samotný procesor. Proto se používají modelovací jazyky, které jsou vhodné jak pro návrh, tak i pro simulaci logických obvodů. Obdobně se postupuje i v jiných oblastech — např. ve strojírenství, kde CAD¹ systémy jsou často doplněny i simulačními nástroji.

2.7 Alternativní přístup – analytické řešení

Analytické řešení modelů je jiný přístup než který používáme při simulaci. Chování systému popíšeme matematickými vztahy a tento abstraktní model řešíme *matematickými metodami*. Výsledky jsou potom ve formě funkčních vztahů, ve kterých se jako proměnné vyskytují parametry modelu — dosazením konkrétních hodnot získáme řešení. Toto je zásadní výhoda analytického řešení proti simulaci, protože je výsledek přesnější a jeho výpočet méně časově náročný.

Bohužel je tento způsob vhodný pouze pro jednoduché systémy nebo zjednodušené popisy složitých systémů. Pro složité matematické vztahy obvykle nemáme matematický aparát na jejich řešení.

Příkladem analytického řešení modelu může být klasická úloha z mechaniky: volný pád tělesa bez uvažování odporu vzduchu a nehomogenity gravitačního pole. Postup řešení a výsledek již znáte z fyziky.

2.8 Kdy je vhodné použít simulační metody

Simulaci používáme především v situacích, kdy:

- neexistuje úplná matematická formulace problému nebo nejsou známé analytické metody řešení matematického modelu;
- analytické metody vyžadují tak zjednodušující předpoklady, že je nelze pro daný model přijmout;

¹CAD = Computer Aided Design = Počítačem podporovaný návrh

- analytické metody jsou dostupné pouze teoreticky, jejich použití by bylo obtížné a simulační řešení je jednodušší;
- modelování na počítači je jedinou možností získání výsledků v důsledku obtížnosti provádění experimentů ve skutečném prostředí;
- potřebujeme měnit časové měřítko — simulace umožňuje téměř libovolné urychlování nebo zpomalování příslušných dějů.

2.9 Základní etapy modelování a simulace

1. *Vytvoření abstraktního modelu* — formování zjednodušeného popisu zkoumaného systému. Pro popis chování objektů reálného systému lze použít dva odlišné přístupy — spojitý a diskrétní. Některé vlastnosti reálného světa se nám jeví jako spojité, některé jiné jako diskrétní². Můžeme rozlišit dvě roviny pohledu na tento problém: hledisko modelu a hledisko implementace.

Z hlediska formulace modelu lze použít jak spojitý, tak diskrétní prostředky popisu. O tom, které prostředky budou vhodnější rozhoduje úroveň abstrakce, kterou při modelování použijeme. Například model osvětlení místnosti lze vytvořit jako diskrétní systém typu svítí/nesvítí, nebo jako spojitý model elektrooptického obvodu se všemi charakteristikami a přechodovými ději. Kterému způsobu popisu dáme přednost, závisí především na účelu modelu a na úrovni našich znalostí, ale je třeba si uvědomit, že implementace simulačního modelu na číslicovém počítači je vždy diskrétní.

2. *Vytvoření simulačního modelu* — zápis abstraktního modelu formou programu. Model má na číslicových počítačích vždy diskrétní charakter. Pouze dříve používané analogové počítače realizovaly spojitou simulaci spojitě na základě fyzikálních analogií. Implementace na číslicovém počítači musí vhodným způsobem napodobit (aproximovat) spojitý chování systému.

Vytvořený simulační model je třeba verifikovat – ověřit jeho správnost vzhledem k abstraktnímu modelu.

3. *Simulace* — experimentování s reprezentací simulačního modelu. Simulační experiment je modelem reálných experimentů. Popis experimentu lze rozdělit do několika fází:

- Příprava experimentu — vytvoření simulátoru, modelu, okolí modelu a jejich inicializace.
- Vlastní provedení experimentu — spuštění simulátoru, řízení běhu simulace a záznam výsledků.
- Ukončení experimentu — záznam výsledků a cílového stavu, zrušení modelu, okolí a simulátoru.

Každá fáze vyžaduje zvláštní část popisu experimentu. Experimenty mohou být libovolně opakovány a simulaci je obvykle možné interaktivně ovládat. Následující experimenty mohou vzít v úvahu výsledky předchozích experimentů.

4. *Analýza a interpretace výsledků* a ověřování jejich věrohodnosti. Při simulaci provádíme záznam průběhu simulace, nebo v případě interaktivní simulace přímo

²Například kvantová mechanika rozlišuje dvojí pohled na hmotu: vlna (spojitá) — částice (diskrétní).



jejich zobrazování. Vizualizace výsledků je prvním krokem k ověřování správnosti modelu, ale nikoliv posledním – je nutná důkladná analýza výsledků podle jejich typu:

- Statistické zpracování výsledků
- Porovnávání výsledků s reálně naměřenými daty
- Automatické vyhodnocení výsledků experimentu, například při provádění série optimalizačních experimentů

2.10 Vztahy mezi modely dynamických systémů

Model je speciální případ systému, který má homomorfní vztah k originálnímu vzorovému systému. Existuje zobrazení, které mapuje systém na model a přitom zachovává podstatné vlastnosti systému. Toto zobrazení je obecně nejednoznačné, to znamená, že jeden systém můžeme modelovat více modely podle účelu a úrovně abstrakce a jeden model může odpovídat více vzorům. To je možné, protože model zanedbává některé detaily systému.

Homomorfní zobrazení mezi systémem a modelem dovoluje zjednodušení struktury i chování modelu. Při zjednodušování jde o vztah N:1, kdy několik komponent systému nahradíme jednou komponentou modelu nebo složitější chování jednodušším. Zachování vztahu 1:1 mezi originálem a modelem je možné pouze ve speciálních případech, protože jinak by běžné modely nebyly realizovatelné pro svůj příliš velký rozsah. Například mechanické modely těles na atomární úrovni jsou možné pouze u velmi jednoduchých systémů. Vztahy systému a jeho modelů vyjadřuje následující tabulka:

systémy	vztah
reálný systém — abstraktní model	homomorfní
abstraktní model — simulační model	izomorfní

Izomorfní vztah znamená 1:1 vztah — nedochází již ke zjednodušování.

2.11 Klasifikace modelů

Přesné rozdělení modelů do kategorií je poměrně náročné, protože často neexistují jednoznačná klasifikační kritéria. Modely lze dělit například podle způsobu jejich matematického popisu, podle úrovně abstrakce, podle metod implementace (paralelní, distribuované) a podle celé řady dalších kritérií.

Tradičně dělíme modely na:

- Modely spojité – proměnné modelu mění svůj stav spojitě. Tyto modely jsou popsateľné například diferenciálními rovnicemi.
- Modely diskrétní – stav modelu se mění skokově v diskrétních časových okamžicích. Příkladem takového modelu může být například konečný automat.
- Modely kombinované – obsahují spojité i diskrétní prvky současně v jednom modelu.

V literatuře [2] je možné nalézt toto rozdělení modelů:



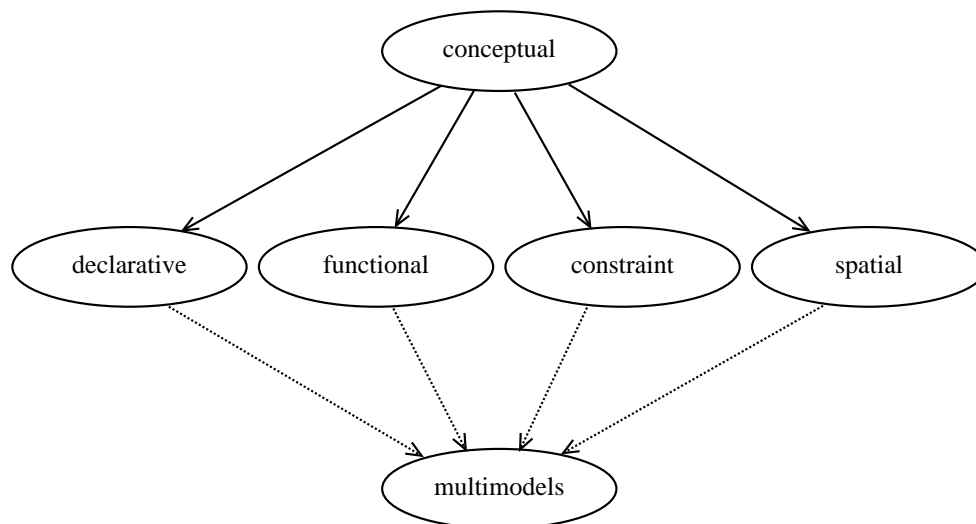
- Konceptuální (conceptual) modely jsou neformální popisy systému, které definují základní strukturu systému (objekty a jejich vztahy) a slovník pojmů, ne-definují tedy kategorie běžné v teorii systémů (stav, událost, funkce). Obvykle se používají v počáteční fázi modelování a jsou výchozím bodem pro návrh simulačního modelu. Jsou základníází znalostí o systému. Mají tvar náčrtků s popisem objektů a vztahů mezi nimi. Často se také používají speciální grafy (concept graphs) a schemata.
- Deklarativní (declarative) modely popisují, jak se mění stav systému ve stavovém prostoru jako důsledek vstupů do systému. Obsahují dvě základní komponenty: stavy a události. Existují dva duální přístupy, které považují za primární buď stavy nebo události. Do této kategorie modelů patří například automaty a Petriho sítě.
- Funkcionální (functional) modely jsou grafy obsahující dvě základní komponenty: funkce a proměnné. Existují dva přístupy podle toho zda graf má uzly reprezentující proměnné nebo funkce. Používají se pro modely fyzikálních objektů, které jsou přímo propojeny a pro modelování toku materiálů systémem.
Příklady: systémy hromadné obsluhy s frontami (tok entit), kompartmentové modely v biologii, blokové modely (spojité systémy, logické obvody) a podobně. Mají obvykle tvar signál–blok–signál nebo funkce–proměnná–funkce.
- Modely popsané rovnicemi a grafy (constraint models) jsou vhodné pro reprezentaci přírodních zákonů. Příkladem jsou diferenciální a diferenční rovnice, elektrická schemata a speciální grafy (bond graphs).
- Prostorové modely (spatial models) vyjadřují prostorovou dekompozici systému, jsou použitelné vždy, když rozdělíme svět na mnoho malých částí, které mají definováno chování a snažíme se pochopit chování celého systému. Za prostorové modely můžeme považovat například celulární automaty a parciální diferenciální rovnice.
- Multimodely (multimodels) vznikají propojením modelů, které mohou být různě popsány. Většina složitých modelů je tohoto typu, protože je třeba sledovat celou řadu aspektů modelovaného systému (předchozí typy modelů sledovaly pouze jeden směr). Využívají mnoho různých pohledů na různých úrovních abstrakce a proto pokrývají větší spektrum systémů.

Na obrázku 2.2 je naznačen vztah těchto tříd modelů.

2.12 Simulační modely

Simulační model je systém vzniklý zobrazením reálného systému do podoby proveditelného modelu (počítačového programu). Vlastnost proveditelnosti znamená, že existuje aparát (počítač), který je schopen interpretovat model tak, aby vykazoval potřebné chování. Simulačním modelem nemusí být jen běžným způsobem zapsaný program, ale může mít například formu Petriho sítě za předpokladu, že máme k dispozici simulátor Petriho sítí. Důležitá je schopnost provádění experimentů se simulačním modelem.

V následujícím textu budeme pojem *model* používat také ve smyslu *simulační model* tam, kde bude rozlišení zřejmé z kontextu.



Obrázek 2.2: Klasifikace modelů

2.13 Simulace

Simulace je proces experimentování s vhodnou reprezentací simulačního modelu. Cílem simulace je analýza chování systému v závislosti na počátečním stavu, vstupních veličinách a hodnotách parametrů. Při simulaci postupujeme tak, že obvykle opakovaně řešíme daný model podle následujícího postupu:

1. inicializace: nastavení hodnot parametrů modelu a počátečního stavu
2. běh simulace (simulation run): zadávání vstupních podnětů z okolí, záznam chování modelu
3. vyhodnocení výstupních dat: získání potřebných informací o chování systému a jeho cílovém stavu

Simulační běhy opakujeme tak dlouho, dokud nezískáme dostatek informací o chování systému nebo pokud nenalezneme takové hodnoty parametrů, pro něž má systém žádané chování.

Typy simulace

Podle použitých modelů a simulačních metod rozlišujeme různé typy simulace. Uvedeme si několik příkladů klasifikace simulačních přístupů:

- Spojitá/diskrétní/kombinovaná simulace – toto je tradiční dělení podle typu modelů se kterými experimentujeme
- Simulace na analogovém/číslicovém počítači, fyzikální simulace – dělení podle použitých simulačních prostředků. Například aerodynamické vlastnosti letadla můžeme zkoumat na modelu v počítači nebo také na fyzikálním modelu v aerodynamickém tunelu. V tomto textu se věnujeme výhradně simulaci na číslicových počítačích.

- Kvalitativní/kvantitativní simulace – rozlišení podle přesnosti zpracovávaných údajů. Kvalitativní simulace uvažuje pouze několik málo hodnot stavových proměnných a trendy jejich změny³. Kvalitativní simulace je vhodná při modelování systémů, které neumíme přesně popsat.
- Simulace v reálném čase (Real-Time simulation) – tento typ simulace vyžaduje synchronizaci běhu simulace s reálným časem. To je nutné například při simulacích typu "hardware in the loop", kdy součástí simulačního modelu jsou reálné podsystémy. Tento přístup se často používá například při návrhu a testování řídicích obvodů v průmyslu.
- Interaktivní simulace – někdy je zapotřebí při simulačním běhu interaktivně vstupovat do procesu simulace a vybírat různé varianty dalšího postupu, případně simulaci předčasně zastavit a zkoumat stav modelu.
- Paralelní a distribuovaná simulace – složité modely vyžadují zapojit do výpočtu mnoho procesorů/počítačů. Simulační systém musí zajistit rozložení výpočtu na jednotlivé procesory a jejich vzájemnou komunikaci. Tato oblast je velmi aktuální a vyžaduje studium speciálních algoritmů a technik.
- Vnořená simulace – někdy může být součástí modelu komponenta, která provádí rozhodování na základě výsledků simulačních experimentů s vnořeným modelem.
- Virtuální realita – tato vysoce interaktivní kombinace grafiky a simulace vyžaduje například také real-time a paralelní zpracování.

Při studiu literatury samozřejmě naleznete i řadu dalších pohledů na simulaci a související simulační nástroje. To je způsobeno širokým spektrem možných aplikací simulace v praxi.

2.14 Verifikace a validace modelu

Verifikace modelu předchází vlastní etapě simulace. Při verifikaci simulačních modelů ověřujeme korespondenci *simulačního a abstraktního* modelu, tj. izomorfní vztah mezi abstraktním modelem a simulačním modelem. Izomorfismus znamená, že simulační model již 1:1 odpovídá abstraktnímu modelu z hlediska struktury a chování. Podobně jako u programů v běžných programovacích jazycích představuje verifikace simulačního modelu jeho ladění.

Validace modelu (ověřování platnosti) je proces, v němž se snažíme dokázat, že skutečně pracujeme s modelem adekvátním modelovanému systému. Jde o jeden z nejobtížnějších problémů modelování protože nelze absolutně dokázat přesnost modelu. Validitu modelu chápeme jako míru použitelnosti a správnosti získaných výsledků.

Při validaci musíme provádět neustálou konfrontaci informací, které o modelovaném systému máme a které simulací získáváme. Pokud chování modelu neodpovídá předpokládanému chování originálu, musíme model vhodným způsobem modifikovat a pokračovat v jeho validaci.

Problematika validity simulačních modelů je značně náročná a silně závisí na konkrétním typu modelů. Zvláště propracovány jsou především různé statistické validační techniky pro systémy hromadné obsluhy a podobné stochastické systémy. Podrobnosti najdete v případech potřeby například v literatuře [3].

³Například hodnota je kladná/nulová/záporná a roste/nemění se/klesá.



2.15 Simulační nástroje – přehled

Simulační nástroje (anglicky: simulation tools) jsou programové prostředky (překladače, knihovny, integrovaná prostředí), které usnadňují popis simulačních modelů a experimentů. Typicky jsou použitelné pro všechny potřebné činnosti při modelování a simulaci:

- Práce s abstraktními systémy – některé simulační systémy umí automaticky převádět popis rovnicemi na simulační model (například prostředí jazyka Modelica [14]).
- Programování simulačních modelů – simulační jazyky a knihovny komponent modelů usnadňují popis modelů. Popis je jednodušší a knihovny komponent jsou již verifikované.
- Experimentování se simulačními modely – součástí simulačních nástrojů jsou prostředky pro popis experimentů a jejich řízení.
- Vizualizace a vyhodnocování výsledků – získávání informací a jejich analýza a případné použití pro další simulační experimenty. Vizualizační nástroje tvoří samostatnou skupinu, která je použitelná i v jiných aplikačních oblastech (statistické zpracování, dolování dat, atd.).

V rámci předmětu IMS můžeme použít řadu nástrojů. Pro použití ve výuce jsou vhodné jak systémy komerční, které jsou obvykle kvalitní a drahé, tak systémy volně dostupné. Zvláště výhodné jsou tzv. "open source" simulační systémy, které dovolují studovat a měnit zdrojový kód programů. V následujícím textu si stručně charakterizujeme několik vybraných simulačních nástrojů:

- Komerční systémy:
 - Dymola/Modelica [14] – velmi vhodný systém pro fyzikální simulace (elektrické obvody, mechanické systémy, vedení tepla). Modelica je objektově orientovaný jazyk vhodný především pro popis spojitých systémů rovnicemi.
 - Matlab/Simulink [11] – v praxi velmi často používaný systém, rozšiřitelný prostřednictvím modulů (toolkitů). Základem je jazyk Matlab speciálně vyvinutý pro efektivní numerické výpočty s maticemi. Spojité modely je možné popsat v maticovém tvaru a řešit vestavěnými algoritmy. Grafický editor Simulink umožňuje blokový opis modelů.
- Volně dostupné "open source" systémy:
 - SciLab [12] – obdoba systému Matlab, nemá tak propracované uživatelské rozhraní a je z části nekompatibilní s Matlabem.
 - Octave [13] – obdoba systému Matlab, nemá zabudované GUI, je téměř kompatibilní s Matlabem.
 - OpenModelica [15] a Modelica Standard Library jsou volně dostupné implementace prostředí jazyka Modelica.
 - SimPack [20] – jednoduchá knihovna podprogramů v C a C++ pro výukové použití.
 - SIMLIB/C++ [6] – simulační knihovna pro C++

Existuje podstatně více různých simulačních nástrojů, značná část z nich je volně dostupná. Často jde o specializované výzkumné nástroje, které nemají příjemné uživatelské rozhraní. Na Internetu jsou rozsáhlé přehledy dostupných simulačních nástrojů (komerčních i nekomerčních) s odkazy a charakteristikami.

2.16 Shrnutí

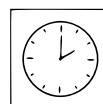
Tato kapitola měla seznámit čtenáře s metodou simulace, použitím simulace v různých oborech, naznačit její výhody a problémy. Důležité jsou především definice významu základních pojmů a souvislosti s alternativními metodami řešení problémů. Přehled nástrojů pro simulaci je pouze orientační, s některými nástroji se podrobněji seznámíte v dalším textu.

2.17 Otázky a úkoly

1. Co je cílem simulace?
2. Vyjmenujte alespoň 5 příkladů použití simulačních metod v praxi.
3. Vytvořte abstraktní model volného pádu tenisového míčku. Zařadte ho podle běžných klasifikačních kritérií do správné kategorie.
4. Mějme model dopravní situace ve městě. Zařadte ho podle běžných klasifikačních kritérií do správné kategorie.
5. Vyjmenujte jaké výhody má použití simulace při návrhu mikroprocesorů.

Kapitola 3

Modelování náhodných procesů



2:00

Modely složitých systémů často vyžadují popis neurčitě specifikovaných jevů. Jeden ze způsobů vyjádření neurčitosti je pravděpodobnostní popis. Vzhledem k tomu, že v předmětu "Numerické metody a statistika" je vysvětleno vše podstatné, zopakujeme v tomto textu pouze některé základní pojmy a souvislosti.

Modelem náhodných jevů jsou v simulačních systémech *generátory pseudonáhodných čísel* s požadovaným rozložením. Pro simulaci využíváme především generátory *pseudonáhodných čísel*, protože je umíme efektivně generovat s využitím speciálních algoritmů a jejich výsledky vykazují stejné statistické vlastnosti jako opravdu náhodná čísla. Generování opravdu náhodných čísel je obecně problematické, protože obvykle vyžaduje speciální technické vybavení¹ a navíc je velmi pomalé.

3.1 Generování pseudonáhodných čísel

Základem pro generování jakéhokoli rozložení je generátor rovnoměrně rozložených čísel v intervalu $\langle 0, 1 \rangle$. V této kapitole si ukážeme jak lze generovat taková pseudonáhodná čísla a jak je lze transformovat na různá rozložení.

Pro generování pseudonáhodných čísel na číslicových počítačích existuje celá řada různých algoritmů. Nejčastěji používané generátory využívají princip lineárního kongruentního generátoru (anglicky: Linear Congruential Generator, LCG):

$$x_{i+1} = (ax_i + b) \bmod m$$

kde

- operace *mod* znamená zbytek po celočíselném dělení,
- a , b a m jsou vhodně zvolené konstanty.

Tento generátor generuje celá čísla s rovnoměrným rozložením v rozsahu $0 \leq x_i < m$. Pro převod na požadovaný rozsah $\langle 0, 1 \rangle$ musíme výsledek x_{i+1} dělit modulem m .

Při počátečním nastavení x_0 (v anglické literatuře naleznete termín "seed") každé použití uvedeného výrazu generuje další číslo. Jelikož počet možných hodnot v intervalu $0 \leq x_i < m$ je omezen, začne se nejpozději po m generovaných číslech opakovat stejná posloupnost. Tato tzv. *perioda generátoru* může způsobit problémy u delších simulačních experimentů. Proto je třeba volit generátory s co možná největší periodou.

Pro dosažení potřebných statistických vlastností výsledné posloupnosti je nutná volba vhodných konstant a , b , m . Pro volbu konstant existují pravidla (viz. například [7]) — modul m je vhodné volit tak, aby měl hodnotu 2^n , kde n je počet bitů čísel typu unsigned integer. Potom nemusíme provádět operaci modulo, protože ta

¹Pro generování náhodných čísel lze někdy využít i vlastností běžného technického vybavení (časování diskových operací, čas příchodu paketů ze sítě, čas stisku kláves atd.) – viz například `/dev/random` a jeho implementace v Linuxu.



je implicitní vlastností celočíselných výpočtů na počítači. Tím se generování zrychlí, protože operace celočíselného dělení je časově náročná.

V literatuře obvykle najdeme vhodně zvolené konstanty, které byly řádně testovány. Některé možnosti jsou uvedeny v následující tabulce:

zdroj	a	b	m
RAND	69069	1	2^{32}
Numerical Recipes	1664525	1013904223	2^{32}
VAX ANSI C	1103515245	12345	2^{31}
Park & Miller	16807	0	$2^{31} - 1$

Při použití jednoduchých kongruentních generátorů musíme mít na paměti i jejich negativní vlastnosti:

- Při špatné volbě parametrů dojde k výraznému zhoršení statistických vlastností generátoru (viz následující příklady).

Velkým problémem kongruentních generátorů je závislost po sobě jdoucích generovaných čísel – to je zvláště nevhodné při generování n -tic náhodných čísel. Proto tyto jednoduché generátory nejsou vhodné pro náročná použití – například pro metodu Monte Carlo.

Existují generátory, které kombinují princip kongruentního generátoru s dalšími operacemi a dosahují tak výrazně lepších výsledků. Pro náročná použití lze doporučit například generátor "Mersenne Twister" [8], který je velmi rychlý a kvalitní.

- Kongruentní generátor generuje celá čísla v rozsahu $0 \leq x_i < m$ s rovnoměrným rozložením. Protože často potřebujeme tzv. normalizované rozložení s rozsahem $\langle 0, 1 \rangle$ musíme použít operaci dělení modulem m a výsledek uložit jako číslo v plovoucí čárce. Například v ISO C můžeme použít standardní funkci `rand`, která generuje pseudonáhodná čísla² v rozsahu $0..RAND_MAX$:

```
x = rand() / (RAND_MAX + 1.0)
```

- Pokud potřebujeme generovat náhodná celá čísla, je třeba postupovat opatrně, protože kongruentní generátory mají málo náhodné nejméně významné bity generovaných čísel.

Například:

```
i = 1 + ( rand() % 10 )
```

generuje celá čísla v intervalu $1..10$, ale se špatnými charakteristikami. Proto raději vždy používejte

```
i = 1 + 10 * (rand() / (RAND_MAX + 1.0))
```

3.2 Příklady generátorů

Pro ilustraci možných problémů si ukážeme příklady (ne)vhodnosti určitých parametrů kongruentního generátoru. Při každém použití je nutné ověřit kvalitu použitého generátoru — pro počítačové hry obvykle nepotřebujeme příliš kvalitní generátory, ale například pro metodu Monte Carlo jsou většinou nutné lepší statistické vlastnosti než které poskytují jednoduché kongruentní generátory.

²Je třeba poznamenat, že některé implementace standardní knihovny jazyka C používají nekvalitní generátory pseudonáhodných čísel.

Příklad: Jednoduchý kongruentní generátor v C

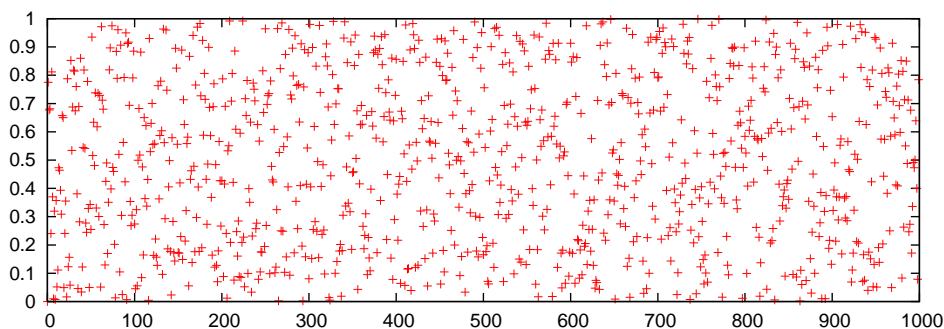
Implementace kongruentních generátorů je velmi jednoduchá. Potřebujeme pouze jednu globální proměnnou a jednu funkci:

```
static unsigned long ix = seed;    // počáteční_hodnota
double Random(void) {
    ix = ix * 69069L + 1;          // implicitní operace modulo
    return ix / ((double)ULONG_MAX + 1);
}
```

Předpokládáme, že výpočet probíhá s přesností 32 bitů – operace modulo je implicitní vlastností výpočtu s celými čísly bez znaménka. Tento generátor nepatří k nejlepším, ale je rychlý a má dlouhou periodu 2^{32} .

Příklad: Výstup generátoru $a = 69069$, $b = 1$, $m = 2^{32}$

Na následujících obrázcích vidíme výsledky generátoru s vhodně zvolenými parametry. Obrázek 3.1 znázorňuje prvních 1000 čísel, které generátor vytvořil při počáteční



Obrázek 3.1: 1000 čísel z generátoru $a = 69069$, $b = 1$, $m = 2^{32}$

hodnotě $ix = 7$. Další obrázek 3.2 zobrazuje 1000 dvojic po sobě jdoucích čísel, zobrazených v rovině tak, že vždy dva po sobě jdoucí vzorky použijeme jako souřadnice (x, y) – první číslo je souřadnice x a druhé je souřadnice y . Poslední obrázek 3.3 demonstruje typickou vlastnost všech jednoduchých kongruentních generátorů – závislost po sobě jdoucích čísel, která se projeví pravidelnou strukturou bodů (x, y) patrnou při velkém zvětšení (zobrazeno 1000 dvojic z celkem $2 \cdot 10^8$ generovaných čísel).

Příklad: Výstup generátoru s chybnými parametry $a = 7$, $b = 1$, $m = 2^{32}$

Pro porovnání si ukážeme vliv špatně zvolených parametrů na posloupnost generovaných čísel. Zobrazení prvních 1000 čísel (obr 3.4) nenaznačuje žádný problém, ale na obrázku 3.5 je vidět silnou závislost dvojic po sobě jdoucích čísel. Tento generátor je zcela nevhodný pro jakékoli použití v simulaci.

Upozornění:

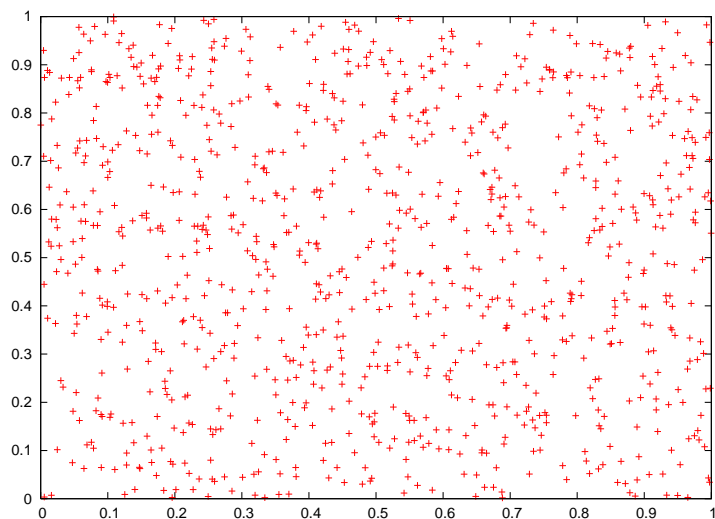
Pro ověření kvality generátoru je vždy nutné provést statistické testy – pouhý pohled na grafické znázornění nestačí.



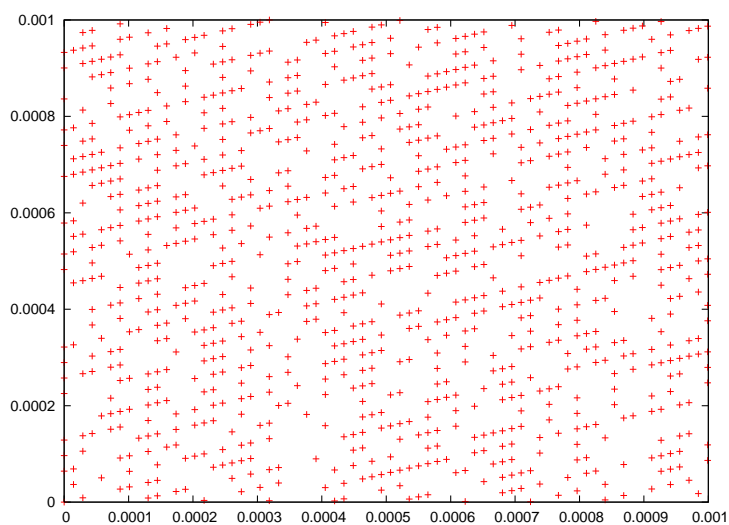
Mersenne twister

V roce 1997 byl vytvořen nový generátor nazvaný *Mersenne Twister* [8] (autoři Makoto Matsumoto a Takuji Nishimura). Tento generátor nemá většinu problémů předchozích





Obrázek 3.2: Dvojice čísel použité jako x,y souřadnice bodů

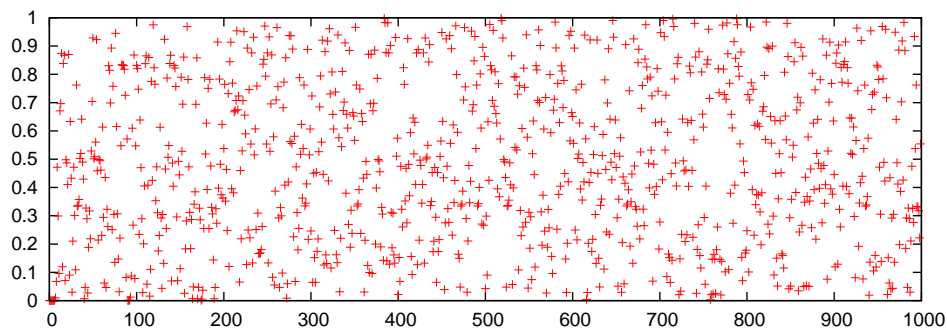


Obrázek 3.3: Dvojice při zvětšení 1000 krát

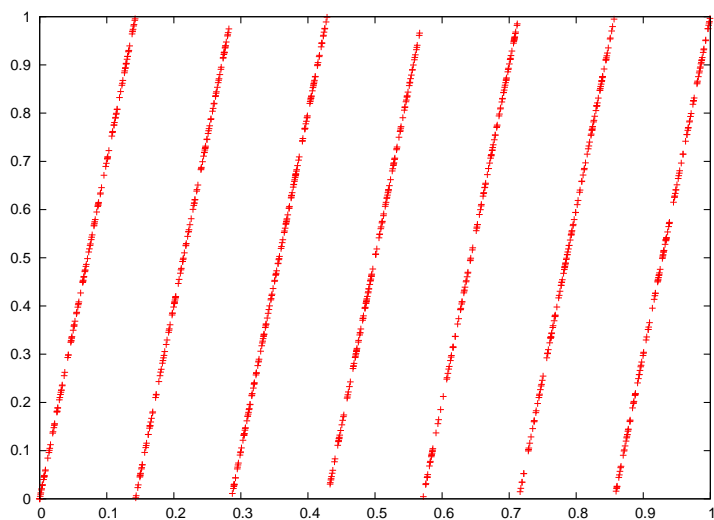
generátorů. Má periodu $2^{19937} - 1$ a má rovnoměrné rozložení až do 623 rozměrů. Navíc je rychlejší než většina ostatních generátorů. Proto je často používán pro náročnější simulace. Jeho nevýhodou je větší složitost kódu než u jednoduchých kongruentních generátorů.

3.3 Transformace rozložení

Protože při simulaci nevystačíme pouze s rovnoměrným rozložením, potřebujeme generátory dalších požadovaných rozložení. Tyto generátory obvykle vytváříme tak, že generujeme čísla s rovnoměrným rozložením a ta transformujeme na požadované roz-



Obrázek 3.4: 1000 čísel z generátoru s chybnými parametry $a = 7$, $b = 1$, $m = 2^{32}$



Obrázek 3.5: Dvojice čísel použité jako x,y souřadnice bodů

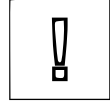
ložení některou z transformačních metod. Zde budeme prezentovat metody:

- inverzní transformace — vychází z inverzní podoby distribuční funkce cílového rozložení. U některých rozložení nelze použít (např. když distribuční funkci nelze vyjádřit elementárními funkcemi, nebo je její výpočet příliš náročný).
- vylučovací — sérií pokusů hledáme číslo, které vyhovuje funkci hustoty cílového rozložení.
- kompoziční — složitou funkci hustoty rozložíme na několik jednodušších, na každý interval pak můžeme použít jinou metodu.

Pro generování některých rozložení existují jiné metody specificky vytvořené pro dané rozložení. Například pro generování normálního (Gaussova) rozložení se využívá sčítání několika čísel s rovnoměrným rozložením.

3.3.1 Metoda inverzní transformace

Tato metoda využívá inverzní funkci k distribuční funkci požadovaného rozložení. Je vhodná pro taková rozložení, pro která je možné snadno vypočítat inverzní distribuční funkci (například exponenciální rozložení).



Postup při vytváření generátoru můžeme shrnout do 3 bodů:

1. Inverze distribuční funkce: F^{-1}
2. Generování náhodného čísla s rovnoměrným rozložením: $x = R(0, 1)$
3. Výsledek: $y = F^{-1}(x)$

Tato metoda je použitelná vždy, když lze jednoduše vypočítat inverzní funkci k distribuční funkci zadaného rozložení.

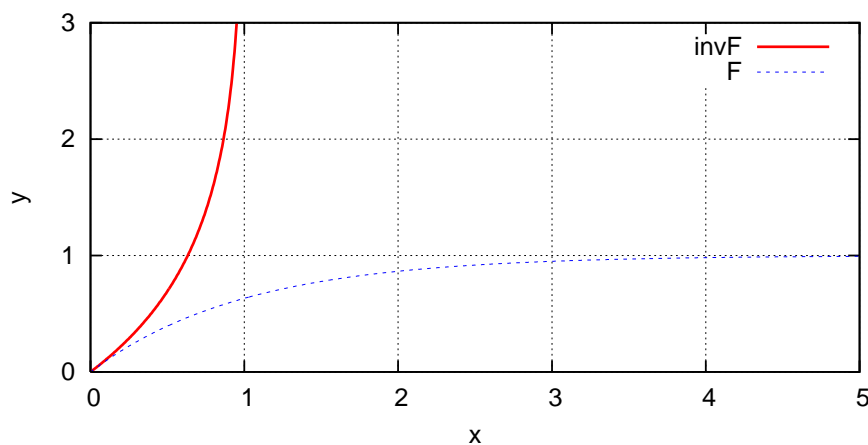
Příklad: Generování exponenciálního rozložení
Exponenciální rozložení má distribuční funkci

$$F(x) = 1 - e^{-\frac{x-A}{b}}$$

funkce inverzní k F je

$$F^{-1}(x) = A - b * \ln(1 - x)$$

Obrázek 3.6 znázorňuje obě funkce graficky pro $A = 0$ a $b = 1$.



Obrázek 3.6: Inverzní distribuční funkce exponenciálního rozložení

Generátor exponenciálního rozložení vypočte výsledek vyhodnocením výrazu

$$y_i = A - b * \ln(1 - x_i)$$

kde rovnoměrně rozložené x_i získáme například vhodným kongruentním generátorem.

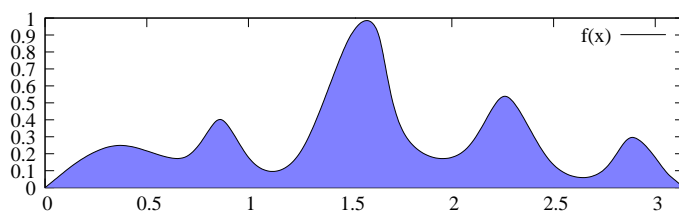
3.3.2 Vylučovací metoda

Tato metoda pracuje na principu náhodné volby bodu v ploše obdélníku ohraničujícího graf funkce hustoty daného rozložení. Pokud máme generátor rovnoměrného rozložení, můžeme generovat náhodně (s rovnoměrným rozložením) body uvnitř obdélníku o souřadnicích (x, y) . Pokud náhodným bodem "trefíme" plochu pod funkcí hustoty f , vrátíme souřadnici x jako nově vygenerované náhodné číslo. Pokud "netrefíme", generujeme další bod a opakujeme tento postup.

Náhodnou veličinu ξ s funkcí hustoty $f(x), x \in \langle x_1, x_2 \rangle, f(x) \in \langle 0, M \rangle$ (M je max. hodnota $f(x)$) generujeme takto:

1. Generování souřadnice $x = R(x_1, x_2)$
2. Generování souřadnice $y = R(0, M)$
3. Je-li $y \leq f(x)$, pak x prohlásíme za hodnotu náhodné veličiny ξ jinak jdeme znovu na bod 1.

Intuitivně lze snadno pochopit, že tam, kde jsou větší hodnoty funkce f je také vyšší pravděpodobnost že zasáhneme plochu a vrátíme x , což přesně koresponduje s větší hustotou pravděpodobnosti výskytu této hodnoty x . Nejlépe princip metody pochopíte na příkladu – viz obrázek 3.7.



Obrázek 3.7: Příklad použití vylučovací metody

Tato metoda je nepoužitelná pro rozložení s neomezeným rozsahem, případně s malým poměrem plochy pod funkcí f k celkové ploše obdélníku ve kterém generujeme náhodné body – příliš často se potom "netrefíme" a metoda není efektivní³.

Na rozdíl od metody inverzní transformace je tato metoda použitelná i pro rozložení, pro která je obtížné vypočítat distribuční funkci a její inverzi.

3.3.3 Kompoziční metoda

Pro některá rozložení je možné použít několik různých metod pro různé oblasti funkce hustoty rozložení a výsledek poskládat z těchto komponent. Tato metoda je vhodná pro komplikovanější průběhy funkce hustoty pravděpodobnosti f .

3.4 Testování generátorů (pseudo)náhodných čísel

Pokud používáme jakýkoli generátor náhodných čísel, musíme se vždy přesvědčit, zda vyhovuje pro naši oblast použití. Někdy může špatně fungovat i převzatý a v praxi ověřený generátor, protože implementace generátoru (například na speciálním technickém vybavení) může mít specifické chování, které podstatně zhorší kvalitu generátoru.

³Efektivitu této metody lze v takových případech zlepšit, pokud negenerujeme body v obdélníkové oblasti, ale v oblasti zadané grafem funkce, která je ve všech bodech zadaného rozsahu větší než naše funkce hustoty.

Existuje mnoho programů použitelných pro provádění různých statistických testů číselných posloupností. Celá řada z nich je volně dostupná, některé jsou součástí různých statistických nástrojů (například GNU R [10]). Vhodná je například sada testů *dieharder* [9], která je implementována v jazyku C a je volně dostupná i se zdrojovými texty.

3.5 Shrnutí

V této kapitole jsme se seznámili se základními principy generování, transformace a testování pseudonáhodných čísel pro použití v simulaci. Při vytváření simulačních modelů obvykle využíváme ověřené generátory zabudované do simulačních nástrojů, ale přesto musíme vždy zkontrolovat vhodnost použitého generátoru pro daný model.

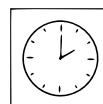
3.6 Otázky, úkoly

- Napište obecný vzorec pro kongruentní generátor. Jaké musí mít generátor parametry, aby byl použitelný pro simulaci?
- Jaká je maximální možná délka periody kongruentního generátoru při vhodné zvolené parametrech?
- Naprogramujte vámi vhodně vybraný generátor pseudonáhodných čísel s modulem $\max 2^{32}$. Experimentálně ověřte délku periody generátoru. Zamyslete se nad tím, jak zjistit že se perioda opakuje.
- Naprogramujte kongruentní generátor rovnoměrného rozložení a proveďte základní statistické testy. Vyzkoušejte i nevhodné parametry generátoru a porovnejte výsledky testů.
- Nakreslete průběh distribuční funkce exponenciálního rozložení se zvolenými parametry, nakreslete průběh funkce inverzní a v grafu naznačte princip metody inverzní transformace.
- Naprogramujte generátor exponenciálního rozložení a proveďte jeho základní testování.



Kapitola 4

Metoda Monte Carlo



1:00

Cílem této kapitoly je seznámení s metodou Monte Carlo. Tento název označuje celou třídu algoritmů pro simulaci systémů, které jsou založeny na experimentování se stochastickými metodami s využitím generátorů náhodných čísel.

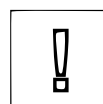
Metoda Monte Carlo je používána v celé řadě oblastí pro výpočet určitých integrálů, zvláště vícerozměrných. Metody Monte Carlo jsou velmi důležité pro obor "Počítačová fyzika" (computational physics) a příbuzné obory. Používají se také pro řešení integro-diferenciálních rovnic popisujících například osvětlení v 3D modelech.

Metoda Monte Carlo je založena na provádění náhodných experimentů s modelem systému a jejich vyhodnocení. Tato metoda nevyžaduje skutečně náhodná čísla, postačující jsou kvalitní generátory pseudonáhodných čísel. Jednoduché kongruentní generátory nejsou vhodné pro náročnější simulace. Výsledkem provedení velkého počtu experimentů je obvykle pravděpodobnost určitého jevu nebo průměrná hodnota veličiny. Na základě experimentálně získané pravděpodobnosti nebo průměru a známých vztahů obsahujících uvedenou hodnotu pak počítáme potřebné výsledky.

Existují různé varianty metody Monte Carlo (například Metropolis), které dosahují vyšší efektivity nerovnoměrným rozložením náhodných experimentů.

4.1 Princip metody Monte Carlo

Základem metody Monte Carlo jsou náhodné experimenty a jejich vyhodnocení. Pro názornost budeme demonstrovat princip metody na velmi jednoduchém příkladu¹, pro který známe analytické řešení – jde o výpočet určitého integrálu



$$\int_0^{\pi} \sin(x) dx$$

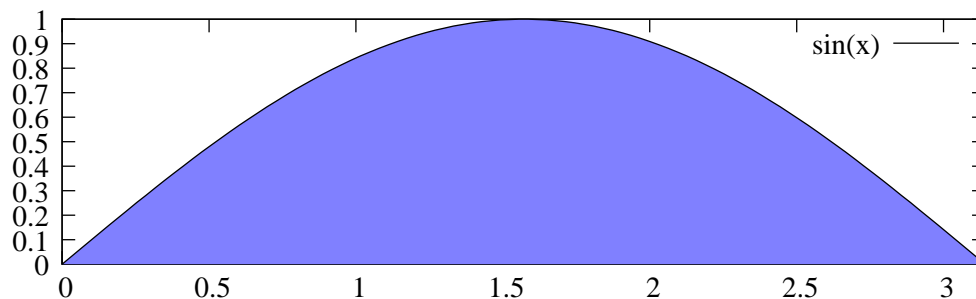
kteřý má hodnotu 2. Postup řešení metodou Monte Carlo je následující:

1. Generujeme N náhodných bodů (x_i, y_i) s rovnoměrným rozložením v zadané ploše.
2. Vypočteme pravděpodobnost P jevu $y_i < \sin(x_i)$, tj. pravděpodobnost že bod leží v ploše pod integrovanou funkcí – viz obrázek 4.1.

3. Vypočítáme přibližný výsledek: $\int_0^{\pi} \sin(x) dx \approx \pi P$
kde π je plocha celého obdélníka

Tato varianta metody Monte Carlo je jednoduchá a snadno pochopitelná, ale má jednu zásadní nevýhodu – musíme znát rozsah hodnot funkce, abychom mohli generovat y_i s rovnoměrným rozložením. Proto se v praxi více používá rychlejší a korektnější ale poněkud méně názorný postup:

¹Upozornění: Pro jednorozměrné integrály se metoda Monte Carlo nepoužívá, protože běžné numerické metody jsou efektivnější, ale pro vysvětlení principu metody nejsou vícerozměrné integrály dostatečně názorné.



Obrázek 4.1: Výpočet integrálu metodou Monte Carlo

1. Generujeme N náhodných bodů $x_i \in (0, \pi)$ s rovnoměrným rozložením.
2. Vypočteme průměr hodnot funkce pro tyto body

$$E = \frac{1}{N} \sum_{i=1}^N \sin(x_i)$$

3. Výsledek získáme vynásobením rozsahu přes který integrujeme (v našem případě $\pi - 0$) experimentálně získaným průměrem hodnot funkce v náhodných bodech:

$$\int_0^{\pi} \sin(x) dx \approx \pi E$$

Lze dokázat, že pro $N \rightarrow \infty$ platí rovnost:

$$\int_0^{\pi} \sin(x) dx = \pi E$$

Výhodou tohoto přístupu je, že potřebuje méně generovaných náhodných čísel a nemusíme znát extrémy funkce na zadaném intervalu.

Poznámka: Použití metody Monte Carlo pro jednoduché integrály není typické a je zde uvedeno pouze pro ilustraci principu metody.

4.2 Přesnost metody Monte Carlo

Přesnost metody je úměrná odmocnině z počtu experimentů – metoda tedy není příliš přesná (pokud zvýšíme počet experimentů 100 krát, přesnost vzroste pouze 10 krát). Lze dokázat, že pro odhad relativní chyby platí:

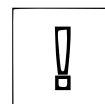
$$err = \frac{1}{\sqrt{N}}$$

kde N je počet provedených experimentů.

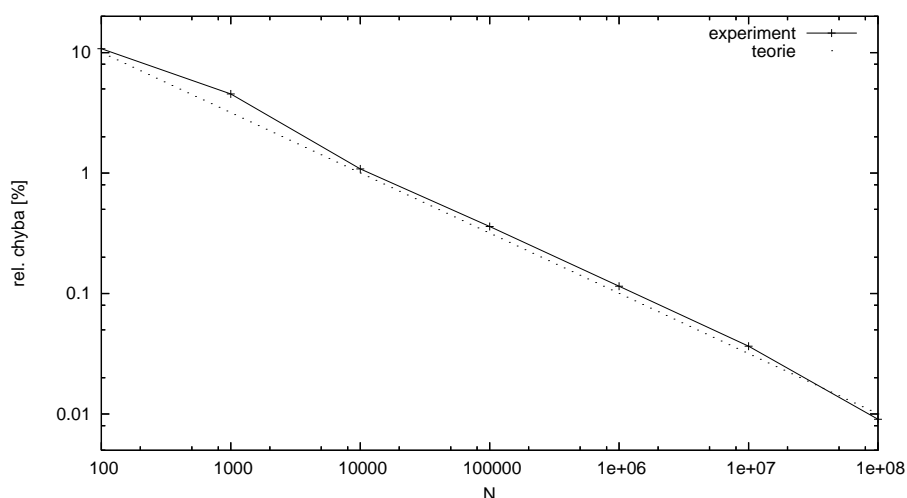
Příklad: Experimentální ověření závislosti chyby na počtu pokusů

Obrázek 4.2 demonstruje závislost chyby metody Monte Carlo na počtu provedených pokusů². Konkrétně byl použit předchozí příklad, pro který známe přesné

²Pozor – osa y má stupnici v procentech!



analytické řešení a můžeme snadno vypočítat chybu metody. Uvedený průběh horního odhadu chyby je získán jako maximum z 10 výpočtů integrálu metodou Monte Carlo pro počet náhodných experimentů N od 100 do 10^8 .



Obrázek 4.2: Experimentálně zjištěná závislost chyby na počtu pokusů

Efektivita metody Monte Carlo ve srovnání s klasickými numerickými metodami roste s počtem rozměrů (stupňů volnosti) řešeného problému. Zatímco složitost výpočtu běžných numerických metod roste exponenciálně s počtem rozměrů, u metody Monte Carlo roste pouze lineárně.

4.3 Typická použití metody Monte Carlo

Metoda Monte Carlo je jedna z nejstarších metod používaných na číslicových počítačích, ale její princip byl znám ještě před vznikem počítačů.

- Princip metody Monte Carlo byl poprvé použit pravděpodobně již v 18. století. Takzvaná Buffonova úloha byla řešena házením jehly na linkovaný papír. Jejím cílem je experimentální zjištění pravděpodobnosti jevu "jehla leží na lince" a výpočet hodnoty π dosazením do vzorce pro výpočet uvedené pravděpodobnosti.
- Metoda byla použita v rámci projektu "Manhattan" (vývoj první atomové bomby za druhé světové války) – bylo to první použití metody na počítači.
- V současné době je tato metoda často využívána pro výpočty N -rozměrných integrálů (efektivní je především pro velká N)
 - V oblasti molekulárních simulací je třeba počítat $3N$ -rozměrné integrály, kde N je počet částic v systému.
 - Netypické je například použití principu metody pro kontrolu složení salámu – přiložíme pravidelnou³ mřížku bodů, spočítáme kolik bodů leží na které složce salámu, tento experiment několikrát zopakujeme s jinak posunutou/otočenou mřížkou a získáme poměr jednotlivých složek.

³Zamyslete se nad tím proč může být mřížka pravidelná když metoda Monte Carlo vyžaduje náhodné experimenty.

- Metodu Monte Carlo lze použít i pro řešení parciálních diferenciálních rovnic. Náhodným experimentem je v těchto případech tzv. náhodná procházka z výchozího bodu, ve kterém počítáme hodnotu řešení.

Podobný princip jako používá metoda Monte Carlo používají i další příbuzné metody. Například optimalizační metoda simulované žíhání (Simulated Annealing) používá náhodně generované body pro hledání extrému funkce.

4.4 Implementace metody Monte Carlo

Metoda Monte Carlo je velmi jednoduchá – výpočet určitého K -rozměrného integrálu funkce f na rozsahu `range` můžeme zapsat v C++ například takto:

```
double mc_integral(unsigned nexp, range_t range, fun_ptr_t f)
{
    double sum=0;
    for (unsigned i = 0; i < nexp; i++) {
        vector<double> x = RandomVector(range);
        sum += f(x);
    }
    double average = sum/nexp;
    return average * Product(range);
}
```

kde

- `RandomVector(range)` vygeneruje souřadnice náhodného bodu v definičním oboru integrované funkce f . Souřadnice tvoří vektor K rovnoměrně rozložených pseudonáhodných čísel se zadanými rozsahy `range` pro všechny rozměry $x_i \in (x_{i \min}, x_{i \max})$.
- `product(range)` je součin velikosti rozsahů přes které integrujeme pro všechny rozměry $\prod_{i=1}^N x_{i \max} - x_{i \min}$

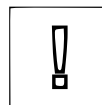
Poznámka: Se změnou rozměru integrálu K se mění pouze počet vygenerovaných náhodných čísel uložených ve vektoru x a počet násobení ve funkci `Product(range)`, ostatní kód zůstává stejný.

4.5 Shrnutí

Metoda Monte Carlo je simulační metoda založená na stochastických experimentech s modelem. Je výhodná především pro vícerozměrné integrály, kdy běžné metody nejsou efektivní. Výhodou metody je velmi jednoduchá implementace. Nevýhodou je relativně malá přesnost metody.

4.6 Otázky a úkoly

- Vyjmenujte oblasti použití metody Monte Carlo a její výhody/nevýhody.
- Demonstrujte použití metody Monte Carlo na výpočtu objemu koule o poloměru r . Výsledek porovnejte s analytickým řešením. Vysvětlete, proč je v tomto případě použití metody Monte Carlo neefektivní.

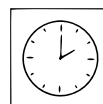


- Napište vzorec pro odhad chyby metody Monte Carlo. Kolik experimentů je zapotřebí pro získání přesnosti kolem 1 promile?
- Proč je metoda Monte Carlo vhodná především pro výpočet N-rozměrných integrálů pro velká N?
- Napište 2 programy pro výpočet obecně N-rozměrného určitého integrálu libovolné N-rozměrné funkce definované v programu. Použijte lichoběžníkovou metodu a metodou Monte Carlo. Porovnejte dobu řešení a dosaženou přesnost pro různá N.



Kapitola 5

Diskrétní simulace



5:30

Cílem této kapitoly je seznámení s formami popisu diskrétních systémů a principy implementace diskrétních simulačních experimentů. Budou definovány základní pojmy se zaměřením na systémy hromadné obsluhy. Po absolvování této kapitoly budete schopni popsat jednoduché diskrétní systémy s využitím simulační knihovny SIMLIB/C++ nebo přímo v jakémkoli programovacím nebo simulačním jazyku. Předpokládáme alespoň základní znalosti programování a datových struktur.

Obsah kapitoly:

- Způsoby popisu diskrétních systémů
- Systémy hromadné obsluhy (queuing systems)
 - Klasifikace systémů
 - Aktivní entity: procesy, události.
 - Pasivní entity: fronty, zařízení, sklady
- Princip "discrete-event" simulace – základní algoritmus řízení diskrétní simulace s kalendářem událostí.
- Implementace diskrétního simulačního systému – přehled použití SIMLIB/C++.
- Nástroje pro sběr statistik
- Příklad s ukázkou řešení

5.1 Formy popisu diskrétních systémů

Diskrétní systémy můžeme popsat celou řadou různých způsobů. Všechny umožňují popis struktury systému a jeho chování v čase. V tomto textu vybereme jen ty základní formy popisu:

- Popis programem v simulačním nebo obecném programovacím jazyce, obvykle formou procesů nebo událostí.
- Konečné automaty jsou významné pro základní popis chování prvků složitých systémů (používáme je například pro popis sekvenčních logických obvodů).
- Celulární automaty popisují prostorové systémy tvořené mřížkou buněk. Chování celého složeného systému je popsáno formou pravidel popisujících chování jednotlivých buněk v závislosti na stavu jejich okolí.
- Grafový popis formou Petriho sítě (C. A. Petri, 1962, existují různé varianty Petriho sítě – barvené, stochastické, atd.). Petriho síť je graf popisující stavy a přechody mezi stavy, umožňuje vyjádření paralelismu a nedeterminismu.

- DEVS (Discrete EVent Specified system) je popis vycházející z teorie systémů [5]. DEVS existuje v několika modifikacích, dovoluje popis hierarchických modelů a distribuovanou simulaci. Je využitelný i pro popis spojitých systémů.

Existují další formalismy pro popis diskretních systémů: sítě automatů, procesní algebry (CCS, CSP, ...), π -calculus, CHAM (CHemical Abstract Machine), PRAM a řada dalších.

Každý z uvedených způsobů popisu je použitelný pro popis diskretních modelů, ale pro určité oblasti jsou některé výhodnější než ostatní – vždy je třeba zvolit vhodný způsob podle aktuální situace.

5.1.1 Popis paralelismu

Jednotlivé v reálných systémech souběžně probíhající činnosti (procesy) můžeme popsat sekvencí kroků – například příkazy programovacího jazyka. Každá taková posloupnost příkazů vyjadřuje obvykle chování celé třídy procesů. Pokud máme více procesů (paralelní procesy), musíme zajistit jejich vzájemnou komunikaci. Popis komunikace procesů je možný například prostřednictvím zpráv. Součástí popisu chování procesů musí být také zajištění jejich synchronizace při používání sdílených prostředků.

Pro simulaci často využíváme pouze jeden procesor, proto musíme paralelní procesy implementovat kvaziparalelně – přepínáme jednotlivé procesy podle situace tak, že vždy běží pouze jeden z nich a ostatní jsou pozastaveny. Tento přístup výrazně zjednodušuje implementaci simulátoru – není nutné žádné zamykání při přístupu do sdílených datových struktur. Některé podrobnosti budou uvedeny v následujících podkapitolách.

5.1.2 Události

Událost popisuje změnu stavu diskretního systému. Forma popisu se liší podle použitého formalismu – například v Petriho síti je událost popsána přechodem, v konečném automatu hranou, v programovacím jazyce například několika příkazy.

Událost je z hlediska doby trvání atomická operace – proběhne celá v jednom okamžiku modelového času (má nulovou dobu trvání). V simulačních systémech je obvykle implementace události triviální – jde o obyčejnou funkci (podprogram).

Typickým příkladem událostí může být příchod zákazníka do obchodu nebo zahájení obsluhy zákazníka u pultu. Událost může také popsat činnost s nenulovou dobou trvání, pokud tento čas zanedbáme (abstrahujeme) – například vyzvednutí vozíku u samoobsluhy.

Jakýkoli diskretní systém je možné popsat pomocí událostí, ale ne vždy je výsledný model srozumitelný. Proto častěji používáme popis pomocí procesů, který je čitelnější a srozumitelnější.

5.1.3 Procesy

V diskretním modelování často používáme popis chování formou procesů. Proces definujeme jako posloupnost událostí, které jsou postupně prováděny. Součástí popisu procesu jsou prostředky vyjadřující čekání po zadanou dobu v modelovém čase, prostředky popisující čekání ve frontě a podobně. Příkazy vyjadřující čekání oddělují jednotlivé události v popisu procesu.

V reálných systémech probíhají procesy paralelně – jejich činnost probíhá současně. Pro implementaci paralelismu v simulačním systému vystačíme s tzv. *kvaziparalelním* zpracováním procesů.

Kvaziparalelismus je forma implementace paralelních procesů – jde o provádění ”paralelních” procesů na jednoprocessorovém počítači postupně. Pokud má dojít k současnému provedení událostí několika procesy, provedou se v daném okamžiku modelového času, ale jedna po druhé (tj. nikoliv současně v reálném čase). Toto postupné provádění na jedné straně zjednodušuje implementaci simulačního systému, ale musíme počítat s tím, že pokud výsledek simulace závisí na pořadí těchto událostí, může vzniknout problém. Simulační systémy obvykle definují pojem *priorita procesu*, který jednoznačně určuje pořadí provedení současných událostí. Pokud potřebujeme modelovat nedeterminismus musíme obvykle na základě pravděpodobností vhodně měnit priority procesů.

V modelovaných systémech často existuje mnoho paralelně probíhajících a vzájemně komunikujících procesů. Abychom zpřehlednili popis složitých systémů používáme objekty, které mohou zapouzdřovat několik procesů. Speciálním případem takových objektů jsou tzv. *agenti* – objekty modelující (například inteligentní) prvky reálných systémů, které samostatně provádějí zadanou činnost.

5.2 Systémy hromadné obsluhy

Systémy hromadné obsluhy (zkratka SHO, anglicky: Queueing Systems) jsou systémy obsahující zařízení, která poskytují obsluhu transakcím. Typický SHO obsahuje:

- transakce (procesy) a popis jejich příchodů do systému
- obslužné linky (zařízení, sklady) a popis obsluhy v těchto linkách
- fronty (anglicky: Queue) ve kterých transakce čekají

Při simulaci SHO sledujeme:

- informace o časovém průběhu transakce procházející systémem,
- doby čekání ve frontách,
- zatížení obslužných linek,

obvykle ve formě statistik. Simulací zjišťujeme různá zdržení transakcí a můžeme optimalizovat výkon systému (například změnou jeho parametrů nebo struktury).

Příkladem typického systému hromadné obsluhy je například benzinová pumpa s několika stojany a pokladnou. Simulace dovoluje studovat nejen běžný provoz systému, ale i různé mimořádné události – například poruchy.

5.2.1 Vstupní tok požadavků

Vstupem systému hromadné obsluhy jsou požadavky (zákazníci, objednávky, výrobky) přicházející z okolí systému. Obvyklý je stochastický popis procesu příchodů při jehož modelování zadáváme typ požadovaného rozložení a parametry. Musíme specifikovat:

- střední dobu mezi příchody, která může mít například exponenciální rozložení, nebo
- počet příchodů za jednotku času (tzv. intenzita příchodů vstupních požadavků), který může mít například Poissonovo rozložení.

Protože exponenciální a Poissonovo rozložení spolu souvisí, lze oba tyto parametry mezi sebou snadno převádět.

5.2.2 Fronty čekajících požadavků

Fronty známe z běžných situací v reálném světě. Fronta se vytvoří vždy, když požadavek chce být obslužen ale musí čekat protože zařízení je již obsazeno jiným požadavkem. Pro fronty je charakteristické:

- Řazení požadavků ve frontě. Nejčastěji se používají fronty typu:
 - FIFO (First In First Out = první přijde první odejde),
 - LIFO¹ (Last In First Out = poslední příchozí první odchází),
 - SIRO (Service in Random Order = zařazení do fronty v náhodném pořadí)
 - Prioritní fronta (řadí podle priorit příchozích)
- Způsob výběru požadavků z fronty. Typicky vybíráme první prvek z fronty, ale existují speciální případy, kdy vyjmeme jiný prvek. Například *fronta s netrpělivými požadavky* dovoluje, aby netrpělivý požadavek opustil frontu, překročí-li doba čekání ve frontě určitou mez.
- Omezení největší možné délky fronty:
 - Běžně používáme neomezené fronty i když v reálných systémech je počet požadavků (například počet zákazníků) vždy omezen.
 - Fronta konečná s omezením kapacity na zadanou hodnotu je nutná v případě modelování omezeného prostoru pro čekající požadavky.
 - Nulová fronta je speciální případ, kdy požadavek vůbec nemůže vstoupit do fronty a je zrušen (systém se ztrátami).

Implementace prioritních front je jedna z významných oblastí algoritmizace a datových struktur. Fronty mohou být implementovány s využitím lineárního seznamu, stromových struktur (heap) nebo dalšími speciálními metodami (například tzv. "calendar queue").

5.2.3 Priority

Přicházející požadavky nemusí být rovnocenné. Každý požadavek může mít tzv. *prioritu*, což je číslo² udávající který požadavek bude přednostně obslužen nebo zařazen do fronty na lepší místo. Rozlišujeme několik různých typů priorit:

- Priorita při čekání na obsluhu.
U jedné obslužné linky lze vytvářet i několik FIFO front požadavků – každá obsahuje požadavky se zadanou stejnou prioritou. Obvyklejší ale je jedna *prioritní fronta*, ve které vyšší priority automaticky "přebíhají" požadavky s nižší prioritou. Požadavky se stejnou prioritou jsou řazeny stylem FIFO.
- Priorita obsazení zařízení s přerušením právě probíhající obsluhy.
Požadavek na obsluhu může také mít zvláštní *prioritu obsluhy* – viz následující sekce "Prioritní obsluha".

Rozlišujeme tedy dva typy priority: *prioritu procesu*, která je významná pro řazení ve frontách a pro pořadí provádění událostí plánovaných na stejný čas; a *prioritu obsluhy*, která umožní přerušování již probíhající obsluhy v zařízení.

¹Frontu typu LIFO znáte pod názvem zásobník.

²Existují dva přístupy: vyšší číslo = vyšší priorita a například u operačních systémů často používané vyšší číslo = nižší priorita.

5.2.4 Prioritní obsluha

Vstupem požadavku s vyšší prioritou nastane jedna ze čtyř základních možností pro právě probíhající obsluhu požadavku s nižší prioritou:

1. Započatá obsluha se normálně ukončí (tzv. slabá priorita).
2. Prioritní obsluha – obsluha se přeruší a začne obsluha příchozího požadavku s vyšší prioritou (tzv. silná priorita). Požadavek, jehož obsluha byla přerušena:
 - odchází ze systému neobsloužen
 - nebo se vrací znovu do fronty a když je později znovu obsluhován, tak:
 - obsluha pokračuje od přerušného místa,
 - nebo začíná znovu od počátku.
3. Jsou-li všechny linky obsazeny a u každé je fronta, požadavek se sám rozhodne, do které fronty se zařadí.
4. Vytvářejí-li požadavky jednu společnou frontu, požadavek z fronty vstupuje do té obslužné linky, která se nejdříve uvolní.

Obslužná síť

Pojem *obslužná síť* označuje systém, který vznikne spojením několika obslužných linek. Rozlišujeme několik typů obslužných sítí podle jejich komunikace s okolím:

Otevřená obslužná síť – probíhá výměna požadavků mezi sítí a okolím.

Uzavřená obslužná síť – nedochází k výměně požadavků mezi sítí a okolím.

Smíšená obslužná síť – kombinuje předchozí dva způsoby: pro některé typy požadavků je síť otevřená, pro jiné uzavřená.

Statické vlastnosti sítě jsou definovány počtem a charakteristikou obslužných linek, a topologií obslužné sítě. Dynamické vlastnosti obslužné sítě jsou definovány charakteristikou procesů příchodu a obsluhy požadavků, charakteristikou procesu přechodu požadavků mezi obslužnými linkami a strategií obsluhy požadavků v obslužných linkách sítě.

5.2.5 Kendallova klasifikace SHO

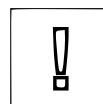
Běžně používaný standard stručného a přehledného vyjádření typu systémů hromadné obsluhy zavedl D. G. Kendall. Klasifikace používá tři hlavní hlediska:

- **X** – typ stochastického procesu popisujícího příchod požadavků k obsluze,
- **Y** – zákon rozložení délky obsluhy,
- **c** – počet dostupných obslužných linek.

Kendallova klasifikace je obecně zapisována ve tvaru **X/Y/c**, kde:

- **X, Y** jsou velká písmena *M, D, G, E_n, K_n, GI* – viz následující tabulka,
- **c** je přirozené číslo, včetně nekonečna ∞ .

Následující tabulka ukazuje několik často používaných případů:



Symbol	X	Y
M	Poissonův proces příchodů tj. exponenciální rozložení vzájemně nezávislých intervalů mezi příchody ³	exponenciální rozložení doby obsluhy
E_k	Erlangovo rozložení intervalů mezi příchody s parametrem λ a k	Erlangovo rozložení doby obsluhy s parametry λ a k
K_n	rozložení χ^2 intervalů mezi příchody, n stupňů volnosti	rozložení χ^2 doby obsluhy
D	pravidelné deterministické příchody	konstantní doba obsluhy
G	žádné předpoklady o procesu příchodu	jakékoliv rozložení doby obsluhy
GI	rekurentní proces příchodů	

Příklad: Systém typu M/M/1

Tento systém má exponenciální rozložení intervalů mezi příchody, exponenciální rozložení dob trvání obsluhy a jednu obslužnou linku.

5.3 Modelování systémů hromadné obsluhy

Systém hromadné obsluhy modelujeme jako typický diskrétní systém:

- Procesy (transakce) modelují paralelní chování aktivních objektů v systému (příchod zákazníka do obchodu, jeho činnost při nákupu, odchod).
- Obslužné linky a jejich stav modelujeme speciálními objekty (například objekt "pokladna" zapouzdřuje prodavače i s potřebným vybavením).
- Fronty jsou speciální seznamy nebo jiné datové struktury, ve kterých mohou být uloženy objekty čekající až je někdo z fronty vyzvedne (také mohou z fronty odejít samy).

Jednoduché systémy lze implementovat velmi snadno i bez použití speciálních simulačních nástrojů. Simulační nástroje ale velmi usnadňují nejen popis modelu, ale především řízení simulace a sběr statistik. Jednu konkrétní implementaci si ukážeme v následujících podkapitolách.

5.3.1 Typy obslužných linek

Základní dělení obslužných linek provádíme podle jejich kapacity:

- $kapacita = 1$, zařízení (anglicky: Facility) může najednou obsluhovat pouze jeden požadavek – jde o zařízení s výlučným přístupem.
- $kapacita > 1$, sklad (Store) může poskytovat svoji kapacitu více transakcím současně.

Pokud modelujeme N zařízení stejného typu, pak záleží na přístupu:

- Každé zařízení má vlastní frontu – použijeme pole N zařízení.

- K zařízením vede jediná fronta – použijeme sklad s kapacitou N nebo pole N zařízení se sdílenou frontou (toto řešení je složitější).

Příklad: Systém s obslužnými linkami (samoobsluha)

Systém s vozíky, speciálním oddělením lahůdek a pokladnami můžeme modelovat například takto:

- Vozíky modelujeme jako sklad x vozíků (jedna fronta), každý proces-zákazník zabere jedno místo v tomto skladu. Až dojde místo, čeká ve frontě až někdo vozík vrátí. Zabrání jedné pozice ve skladu znamená vyzvednutí vozíku.
- Např. dva prodavače lahůdek modelujeme jako dvě zařízení se sdílenou frontou. Jakmile je zařízení-prodavač volné, obsadí jej první požadavek z fronty.
- Pět pokladen můžeme modelovat jako pět samostatných zařízení (ke každému je zvláštní fronta).

5.3.2 Popis chování transakcí

Tato podkapitola naznačuje způsob popisu nejčastěji používaných situací, ke kterým dochází při modelování systémů hromadné obsluhy.

- Příchod a odchod transakce

Generování transakcí (procesů) obvykle provádíme periodicky se opakující událostí – událost vytvoří nový proces a naplánuje svoje opakování za určitý čas. Nově vytvořený proces je obvykle okamžitě aktivován, tj. je naplánováno jeho spuštění.

Transakce opouští systém tím, že skončí její popis (konec funkce), nebo provede speciální příkaz (obvykle nazvaný Cancel nebo Terminate). Simulační systém zařídí likvidaci transakce.

- Obsazení zařízení

- Obslužná linka s kapacitou 1 (Zařízení, Facility) je buď volná nebo obsazená. Obsazení zařízení provádíme operací Seize.
- Obslužná linka s kapacitou $N > 1$ (Sklad, Store) má obsazeno 0 až N míst. Obsazení požadované kapacity skladu provádíme operací Enter s parametrem udávajícím jakou kapacitu požadujeme.

Pokud nelze provést obsazení, musí transakce čekat ve frontě do které je automaticky zařazena. Až se dostane na první místo ve frontě (u skladu je možné i dříve) a skončí aktuálně probíhající obsluha je transakce reaktivována, opustí frontu, provede obsazení a pokračuje prováděním své obsluhy.

- Neblokující obsazení zařízení

Pokud transakce přistupuje k zařízení, ale nechce čekat ve frontě, musí před pokusem o obsazení dotazem zjistit stav zařízení (to je možné díky kvaziparalelnímu zpracování) a zařízení obsadí jen pokud je volné, jinak provedeme "plán B".

- Náhodný výběr zařízení

Transakce si náhodně vybírá jedno z N zařízení. Tento výběr řešíme generováním náhodného čísla a podle výsledku například podmíněným příkazem nebo příkazem `switch` vybereme příslušné zařízení a další provádíme akce.

- Výběr s prioritou zařízení

Transakce přistupuje prioritně k jednomu ze zařízení. Toto snadno vyřešíme správným pořadím testování obsazenosti zařízení – obsadíme první volné zařízení v pořadí od zařízení s nejvyšší prioritou. Pokud jsou všechna obsazena řadíme se do společné fronty.

- Modelování obsluhy

Vlastní obsluhu modelujeme obvykle pouhým čekáním po stanovenou dobu, protože co se děje v rámci obsluhy nás obvykle detailně nezajímá.

- Ukončení obsluhy

Obsluhu zařízení ukončujeme speciálním příkazem (Release). V rámci operace uvolňování je obvykle proveden test neprázdnosti vstupní fronty a případně aktivace první transakce ve frontě. Zařízení musí opustit stejná transakce která jej obsadila.

Uvolnění zadané kapacity skladu může provést libovolná transakce příkazem (Leave), který nakonec opět testuje vstupní frontu a aktivuje první nebo případně i další transakce, jejichž požadavek lze uspokojit. Zde záleží na zvolené strategii skladu.

5.3.3 Příklad modelu SHO

Pro ilustraci způsobu popisu systému si uvedeme jednoduchý příklad SHO:

Do opravářské dílny přichází zákazníci v intervalech daných exponenciálním rozložením pravděpodobnosti se střední hodnotou 20 minut.

V dílně jsou dva opraváři: jeden zpracovává normální zakázky a druhý náročné zakázky. Každá třetí zakázka je náročná. Vyřízení normální zakázky trvá 15 minut s exponenciálním rozložením pravděpodobnosti, vyřízení náročné zakázky zabere 45 minut s exponenciálním rozložením. Zákazník čeká na vyřízení své zakázky a pak systém opouští.

Model popíšeme pseudokódem a zjednodušíme – neuvažujeme sběr statistik:

```
Zařízení narocne("Náročné zakázky"); // obsahuje i frontu
Zařízení normalni("Normální zakázky");
```

```
Proces Zakaznik:                // transakce
// ... možná lokální data pro každého zákazníka
Popis_chování:
  if (Random() <= 0.33)          // s pravděpodobností 0.33
    Obsazení(narocne);           // pokus o obsazení zařízení
                                // může znamenat čekání ve frontě
    Čekání(Exponential(45));
    Uvolnění(narocne);           // ukončení obsluhy
  else
    ... téměř totéž pro druhou variantu
// konec popisu objektů typu Zakaznik
```

```
Událost Generator:
  Popis_chování:
    nový Zakaznik, Aktivace_ihned();
    Aktivace_v_čase (Time + Exponential(20));
// konec popisu generátoru zákazníků
```

$x + y$

```

Popis_experimentu:
    Inicializace_simulátoru(0, 1000); // doba simulace
    // inicializace modelu:
    (nový Generator) -> Aktivace_ihned();
    Běh_simulace();
// konec popisu experimentu

```

V pseudokódu je patrná objektová struktura modelu a popis experimentu. Každý simulační systém má podobné možnosti popisu modelu, někdy jsou části modelu zabudovány do simulačního systému. Například generátor zákazníků je často používán a je v podstatě vždy stejný – proto může být snadno nahrazen jedním příkazem. Je také možné popsat uvedený systém graficky (blokovým schematem nebo Petriho sítí) a použít tomu odpovídající simulační nástroje.

5.4 Kalendář událostí, algoritmus řízení simulace

Nyní si naznačíme princip implementace diskrétního simulačního systému. Uvedenému způsobu implementace s kalendářem událostí odpovídá anglický termín "next-event"⁴.

Kalendář (future event list, Pending Event Set) je uspořádaná datová struktura (typu prioritní fronta) uchovávající aktivační záznamy událostí/procesů. Každá naplánovaná budoucí událost ("next event") má v kalendáři aktivační záznam ve tvaru:

$(atime, priority, event)$

kde:

- *atime* je čas příští aktivace události,
- *priority* je priorita procesu/události definující relativní uspořádání událostí plánovaných na stejný čas,
- *event* je odkaz na danou událost nebo proces.

Kalendář definuje operace umožňující:

- Test je-li kalendář prázdný
- Vkládání nových záznamů s jejich zařazením na správné místo.
- Výběr prvního záznamu s nejmenším aktivačním časem.
- Výběr požadovaného konkrétního záznamu – toto je nutné v případě, že potřebujeme již naplánovanou událost z kalendáře zrušit.
- Inicializace a destrukce kalendáře

První tři operace musí být velmi efektivní, aby nebrzdily simulaci. Diskrétní simulátor

s kalendářem událostí se řídí následujícím algoritmem:

⁴Existuje ještě další způsob implementace "activity scanning", který zde nebudeme prezentovat.




```

čas = T_START
Inicializace kalendáře událostí a modelu
while( Kalendář je neprázdný ) {
    Vyjmi první aktivační záznam (AZ) z kalendáře
    if ( čas atime v AZ > T_END )
        break; Ukončení cyklu
    Nastav čas na aktivační čas atime v AZ
    Proveď popis chování události event v AZ
}
čas = T_END; Konec simulace.

```

Provádění událostí obvykle zahrnuje i vytváření, plánování a rušení dalších událostí. V případě popisu formou procesů je provedení události implementováno jako pokračování v popisu procesu od místa, kde byl přerušen (proces je posloupnost událostí a jedna z nich je provedena). Pokud není požadována žádná další budoucí událost je kalendář prázdný a simulace může skončit.

Při simulaci uvedeným algoritmem na jednoprocessorovém počítači vždy běží jen jeden proces nebo událost (tzv. kvaziparalelní zpracování). Ostatní procesy čekají ve frontách nebo jsou registrovány v kalendáři, případně jsou pasivní. Proto nemůže být aktivní proces při simulaci přerušen a v době svého běhu má přístup ke všem zdrojům modelu. Provádění procesu je přerušeno až na jeho vlastní žádost například při čekání příkazem `Wait` (obdoba kooperativního/nepreemptivního multitaskingu). Pokud je proces popsán jednou funkcí, musí být zajištěno, že její provádění je možné zastavit a později znovu obnovit na stejném místě. Toto přepínání procesů je například v SIMLIB/C++ řešeno (bohužel nepřenositelně) úschovou/obnovou určité oblasti zásobníku a použitím funkcí `setjmp()` a `longjmp()`. Podrobnosti zájemci naleznou ve zdrojovém kódu.

5.5 Přehled diskretní části SIMLIB/C++

Tato kapitola uvádí stručný přehled prostředků pro diskretní simulaci tak jak jsou implementovány v SIMLIB/C++. Cílem je seznámení s jednou z možných forem popisu diskretních systémů.

SIMLIB/C++ je jednoduchá simulační knihovna pro C++, která poskytuje základní prostředky pro diskretní modelování:

Process – bazová třída pro modelování procesů

Event – bazová třída pro modely událostí

Facility – obslužná linka s výlučným přístupem

Store – obslužná linka se zadanou kapacitou

Queue – prioritní fronta

Statistiky – sada tříd pro sběr a uchování statistických informací

V následujícím textu si ukážeme jejich použití při modelování různých částí diskretních systémů. Uvedené prostředky lze libovolně doplňovat a také modifikovat například s využitím dědičnosti. Některé třídy jsou určeny pouze jako bazové třídy pro dědění – jsou to tzv. abstraktní třídy.

Poznámka: Podrobnější dokumentaci, zdrojové texty knihovny a příkladů naleznete na WWW stránce [6].

5.5.1 Obecná struktura modelu

Zdrojový soubor s kódem modelu a popisem experimentu má obecně strukturu:

```
#include "simlib.h"    // vložení rozhraní knihovny

// deklarace obslužných linek

// deklarace tříd popisujících procesy a události

// popis simulačního experimentu ve funkci main()
```

Simulační model v SIMLIB/C++ je běžný program v jazyce C++. Všechny konstrukce a knihovny jazyka C++ jsou použitelné – můžete například model doplnit grafickým uživatelským rozhraním. Větší modely je možné rozložit do více zdrojových souborů (modulů). Pro ladění modelu lze využít běžné ladicí nástroje.

5.5.2 Popis simulačního experimentu

Popis experimentu je obvykle celý uveden ve funkci `main()` a má strukturu:

```
int main() {
    <příkazy1>    // základní inicializace
                //   například SetOutput("soubor");
    Init(<počáteční čas>, <koncový čas>);
                // inicializace simulátoru
    <příkazy2>    // inicializace modelu
                //   například vytvoření objektů
    Run();       // běh simulace
    <příkazy3>    // zpracování výsledků
                //   například tisk statistik
}
```

Sekvenci `Init(t0,t1); ...; Run(); ...;` lze libovolně opakovat. Toho využijeme například při optimalizačních experimentech, kdy hledáme nejlepší parametry modelu pro dosažení zadaných kritérií.

5.5.3 Modelový čas

Modelový čas je proměnná, kterou nastavuje simulační systém:

```
double Time;
```

Příkaz `Init(t0,t1);` v popisu experimentu vždy nastaví počáteční čas na `t0`, další změny provádí algoritmus řízení simulace. Po dosažení času `t1` nebo po zadání speciálního příkazu `Stop();` je simulační běh ukončen. Do proměnné `Time` nelze zapisovat přiřazovacím příkazem, proto zápis:

```
Time = 10;
```

vyvolá chybu při překladu. Posun času řídí výhradně jádro simulátoru.

Jednotka času použitá v modelu může být libovolná a musí být stejná v celém modelu. Model může mít základní časové údaje například v minutách a všechny ostatní časové údaje se musí přepočítat na minuty.

5.5.4 Generátory pseudonáhodných čísel

SIMLIB obsahuje celou řadu generátorů, pro naše účely vystačíme se základními generátory:

```
Random();          -- rovnoměrné rozložení, R(0,1)
Uniform(L, H);     -- rovnoměrné rozložení, R(L,H)
Exponential(E);    -- exponenciální rozložení se středem E
Normal(M, S);      -- normální rozložení se středem M a rozptylem S
```

Všechny generátory vrací hodnotu typu `double`. Základní generátor `Random()`; je velmi jednoduchý kongruentní generátor (viz kapitola 3). SIMLIB/C++ umožňuje nahradit tento generátor jiným generátorem. Způsob jak to udělat najdete v dokumentaci.

5.5.5 Popis události

Událost je jednorázová (nepřerušitelná) akce vyvolaná v určitém modelovém čase zadaném při plánování její aktivace příkazem `Activate`. V SIMLIB musí být vždy odvozena od abstraktní báze třídy `Event`. Každá událost musí definovat metodu `Behavior` s příkazy, které se provedou při výskytu události. Žádný příkaz nemůže vyvolat čekání v modelovém čase. Často se používají periodické události, které se samy znovu aktivují za určitý časový interval.

Uvedeme si příklad popisu periodické události:

```
class Udalost : public Event {
    void Behavior() {
        // příkazy popisující akce události
        Activate(Time + e); // bude se periodicky aktivovat
    }
};
```

Vytvoření a plánování aktivace události na zadaný čas obvykle zapisujeme v rámci jednoho příkazu takto:

```
(new Udalost)->Activate();          // na čas Time
(new Udalost)->Activate(Time + e); // na čas Time + e
(new Udalost)->Activate(t);         // pozor na t<=Time
```

Pokud potřebujeme uchovat odkaz na nově vzniklé události, musíme si vytvořit pomocnou proměnnou s tímto odkazem například takto:

```
Event *e = new Udalost;
e->Activate(10); // aktivace v čase 10
e->Cancel();     // likvidace události
```

5.5.6 Příchod a odchod transakce

Generování transakcí (procesů, výjimečně událostí) provádíme dynamicky operací `new` obvykle se současnou aktivací procesu:

```
class Generator : public Event {
    void Behavior() {
        (new Proc)->Activate(); // nový proces, aktivován ihned
        Activate(Time+Exponential(2)); // periodické generování
    }
};
```

Transakce opouští systém implicitně po ukončení metody popisující její chování (**Behavior**), případně explicitním použitím příkazu **Cancel()** :

```
class Proc : public Process {
    void Behavior() {
        //... příkazy - viz dále
        if (podmínka)
            Cancel(); // konec procesu
        jiný_proces -> Cancel();
        //...
    } // proces implicitně končí (opouští systém)
};
```

5.5.7 Popis procesu

Procesy (transakce) musí být odvozeny z abstraktní třídy **Process**.

```
class Transakce : public Process {
    // případná uložená data transakce
public:
    Transakce( parametry ) { // konstruktor - má stejné jméno
                            // a může mít parametry
        // konstruktor je nepovinný a popisuje inicializaci procesu
    }
    void Behavior() {
        // ...
        // příkazy popisující chování procesu
        // ...
    }
};
```

Po aktivaci procesu se volá metoda **Behavior** (chování). Vykonává se normálně jako běžná funkce, je však přerušitelná při provádění některých operací:

- Explicitní čekání příkazem **Wait(dt)** po zadanou dobu. Čekání modeluje nějakou činnost procesu trvající **dt** časových jednotek, která nás blíže nezajímá.
- Čekání ve frontě u zařízení může nastat například při obsazování zařízení nebo skladu v rámci příkazů **Seize** a **Enter**.
- Při zadání příkazu **Passivate()** se provádění procesu pozastaví na neurčito až do jeho příští aktivace například příkazem **Activate(čas)**.

Plánování aktivace procesu

Proces se chová jako posloupnost událostí popsaných v rámci jedné metody:

```
void Behavior() {
    // ... akce - událost a
    Wait(3); // čekání
    // ... pokračování procesu - událost b
}
```

Proces při provádění příkazu **Wait(3)** čeká 3 časové jednotky. Při diskrétní simulaci to znamená, že se naplánuje další pokračování tohoto procesu na čas $Time + 3$

(příkazem `Activate(Time+3)`). Aktivační záznam této události je při tom uložen do kalendáře událostí, proces je přerušen a pokračuje se prováděním první plánovaná akce v kalendáři.

5.5.8 Základní operace procesu

- Vytvoření instance procesu provedeme dynamicky:

```
Transakce *t = new Transakce;
```

Ukazatel `t` není nutný – pokud jej nepotřebujeme v rámci popisu modelu, nemusíme výsledek operace `new` ukládat. Simulační systém aktivní procesy registruje v kalendáři a po ukončení popisu chování je automaticky zruší.

- Plánování (re)aktivace procesu do kalendáře:

```
t->Activate(tm);
```

Aktivuje se v čase tm (implicitně je to $tm = Time$ tj okamžitě). Aktivace ještě neznamená spuštění procesu – k tomu dojde až skončí právě prováděná událost a aktivační záznam tohoto procesu bude na prvním místě v kalendáři událostí.

- Zrušení procesu i jeho registrace ve všech strukturách (fronty, kalendář):

```
t->Cancel(); // nebo delete t;
```

- Suspendování procesu znamená pozastavení jeho provádění až do jeho další aktivace jiným procesem. Suspendovaný proces nemá záznam v kalendáři.

```
Passivate();
```

Poznámka: Pro události nelze z vyjmenovaných operací použít `Passivate`, protože událost proběhne vždy bez přerušení.

5.5.9 Priorita procesu

Každý proces má atribut `Priority`, který ovlivňuje jeho řazení do front. Je to celočíselná proměnná s implicitní hodnotou 0 která odpovídá nejnižší prioritě. Nastavení priority procesu je možné při jeho vzniku parametrem konstruktoru nebo později přiřazením do `Priority`. Pro ilustraci si uvedeme příklad různých možností nastavení priority:

```
class MProc : public Process {
public:
    MProc() : Process( PRIORITA1 ) { };
    void Behavior() {
        Priority = PRIORITA2;
        Seize(F);
        Priority = 0; // změna priority na 0
        // ...
    }
};
```

Čekání procesu

Explicitní čekání příkazem `Wait(expr)` způsobí, že proces je přerušen (podobně jako při provedení příkazu `Passivate`) a do kalendáře je naplánovaná událost aktivace procesu na čas `Time+expr`.

Čekání může nastat také pokud proces přistupuje k zařízením u kterých je možné čekání ve vstupní frontě (`Facility`, `Store`). Například při zápisu:

```
Facility F("F");
// v popisu chování procesu:
Seize(F); // zde může čekat
...;
Release(F); // zde nikdy nečeká
```

je možné, že pokus o obsazení zařízení příkazem `Seize` povede k čekání ve frontě u zařízení.

Příklad: Timeout — přerušení čekání

V některých modelech je třeba provést přerušení čekání po určité době (například při modelování netrpělivých požadavků). Následující příklad naznačuje řešení tohoto problému:

$x + y$

```
class Timeout : public Event {
    Process *Id; // o který proces jde
public:
    Timeout(Process *id, double dt) :
        Id(id) // odkaz na proces
    {
        Activate(Time+dt); // kdy vyprší čas
    }
    void Behavior() {
        Id->Cancel(); // zrušení procesu = odchod požadavku
        Cancel();    // a zrušení této události
    }
};

// Použití třídy Timeout:
class MProc : public Process {
    void Behavior() {
        Timeout *t = new Timeout(this, 10);
        Seize(F); // možné čekání ve frontě
        delete t; // jen pokud nevypršel čas
        ...
    }
};
```

5.5.10 Fronty (třída Queue)

Třída `Queue` definuje prioritní frontu řazenou podle priority procesu a v případě stejných priorit procesů používá jako další úroveň řazení FIFO. Pokud potřebujeme jiné řazení, můžeme definovat novou třídu a použít ji stejným způsobem. Nejlepší způsob jak postupovat je dědit nebo převzít zdrojový kód třídy `Queue` a modifikovat ho podle požadavků.

Fronty obvykle nepotřebujeme explicitně vytvářet, protože jsou součástí zařízení. Jen ve speciálních případech, když potřebujeme vlastní frontu můžeme definovat například:

```
Queue q;    // fronta pro speciální použití
```

Následující příklady demonstrují použití vlastní fronty. Například vložení procesu do fronty zapíšeme:

```
void Behavior() { // popis chování procesu
    ...
    q.Insert(this); // proces se vloží do fronty
    Passivate();    // pozastaví se
    ...
}
```

Jiný proces (nebo událost) může vložený proces z fronty vybrat a znovu aktivovat:

```
...
if (!q.Empty()) { // pokud je fronta neprázdná
    Process *tmp = q.getFirst(); // výběr prvního
    tmp->Activate(); // aktivace
}
```

5.5.11 Zařízení (Facility)

Zařízení je obsaditelné procesem pro výlučný přístup (tj. maximálně jeden proces je obsluhován). Každé zařízení obsahuje dvě fronty požadavků:

- fronta čekajících požadavků (vnější)
- fronta přerušených požadavků (vnitřní) – v této frontě čekají požadavky kterým byla obsluha přerušena příchodem požadavku s vyšší prioritou obsluhy.

Při vytvoření zařízení máme možnost zadat parametry:

```
Facility F1("Jméno zařízení");
Facility F2("Jméno zařízení", Queue *fronta);
Facility F3[10];
```

1. Jméno zařízení se používá pouze pro zřehlednění výstupů (tiskne se ve výstupu a statistikách zařízení).
2. K zařízení můžeme připojit jinou frontu (např. společnou s jiným řízením) – viz zařízení F2.
3. Konstrukce pole zařízení nemůže být v SIMLIB/C++ parametrizována, ale je možné kdykoli (a nejen u polí) nastavit potřebné hodnoty explicitně:
 - jméno zařízení příkazem `Fac5[i].SetName("Jméno2")`
 - frontu u zařízení příkazem `Fac5[i].SetQueue(fronta2)`

Příklad: Obsazení zařízení

Obsazení linky, vykonání obsluhy a uvolnění linky popíšeme takto:

```

Facility F("Fac");
...
class P : public Process {
    void Behavior() {
        ...
        Seize(F);           // obsazení, může čekat ve frontě
        Wait(Exponential(10)); // probíhá obsluha
        Release(F);         // uvolnění
        ...
    }
};

```

Priorita obsluhy v zařízení

U zařízení rozlišujeme dva typy priorit:

- Priorita procesu určuje řazení ve vstupní frontě
- Priorita obsluhy v zařízení je parametrem operace obsazení **Seize**. Pokud má příchozí požadavek na obsazení vyšší prioritu obsluhy než aktuálně probíhající obsluha, dojde k přerušení obsluhy a začne být obsluhován nově příchozí požadavek. Priorita obsluhy se používá především pro modelování poruch zařízení.

Příklady demonstrující použití priority obsluhy:

```
Seize(Fac);    // proces A obsazuje zařízení
```

V obsluze je proces A se standardní prioritou obsluhy (0).

```
Seize(Fac, 1); // proces B přerušuje obsluhu procesu A
```

Jiný proces B žádá o obsluhu s vyšší prioritou obsluhy. Proces A je odstaven do vnitřní fronty a do obsluhy se dostává B.

```
Release(Fac); // proces B končí obsluhu, pokračuje obsluha procesu A
```

Při uvolnění zařízení procesem B zařízení dokončí rozpracovanou obsluhu procesem z vnitřní fronty s nejvyšší prioritou obsluhy (v našem případě A).

5.5.12 Sklad (Store)

Sklad umožňuje simultánní přístup ke zdroji s určitou kapacitou (parkoviště, paměť počítače, místa v autobuse). Proces požaduje ve skladu obsazení části jeho kapacity c . Je-li dostupná kapacita volná, přidělí se a zmenší se množství dostupné volné kapacity. Není-li dostatek místa čeká proces ve frontě až se uvolní požadovaný prostor.

Příklad: Základní použití skladu

```

Store Voziky("Voziky", 50);
...
// Proces typicky provádí následující operace:
Enter(Voziky, 1);
// ... činnost
Leave(Voziky, 1);

```


Obsazená kapacita skladu nijak nesouvisí s procesem – uvolnit ji může libovolný jiný proces. Po uvolnění kapacity příkaz `Leave` postupně prochází frontu čekajících od první položky a všechny požadavky s uspokojitelným požadavkem na obsazení kapacity skladu jsou obslouženy. Tento přístup je použitelný, ale může vést k vyhladovění (starvation) procesů s velkými požadavky. Připravovaná nová verze SIMLIB bude mít možnost měnit toto chování podle požadavků.

Poznámka: Sklad nemá prioritu obsluhy.

5.5.13 Příklady různých použití zařízení

V této podkapitole si ukážeme použití zařízení v různých typických situacích. Především jsou prezentovány některé strategie výběru z N zařízení.

$$x + y$$

Neblokuující obsazení linky

Transakce by ráda obsadila zařízení, ale nechce čekat ve frontě:

```
Facility F("Fac");
class Proc : public Process {
    void Behavior() {
        ...
        if (!F.Busy())    // nejdříve zjistíme situaci
            Seize(F);     // pokud je volno, obsadíme
        else
            ...           // jinak provedeme něco jiného
    }
}
```

Vzhledem ke způsobu implementace simulátoru se nemusíme obávat přepnutí procesů mezi testem obsazenosti zařízení metodou `Busy` a jeho obsazením metodou `Seize`.

Náhodný výběr z N zařízení

Transakce přistupuje (staví se do fronty) k jednomu z N zařízení která náhodně vybírá. Nedeterministický výběr zařízení modelujeme rovnoměrným rozložením.

```
const int N = 3;
Facility F[N]; // pole zařízení, každé má vlastní frontu

class Proc : Process {
    void Behavior() {
        ...
        int idx = int( N * Random() ); // vybere náhodné zařízení
        Seize(F[idx]);                // obsazení
        ...
        Release(F[idx]);              // uvolnění
        ...
    }
};
```

Výběr zařízení s prioritou

Transakce přistupuje k jednomu z N zařízení, priorita je daná pořadím v jakém procházíme pole, pokud jsou všechna zařízení obsazena, jde do fronty u posledního zařízení.

```
const int N = 3;
Facility F[N];
...
int idx;
for(idx=0; idx < N-1; idx++) // skončíme na posledním zařízení
    if(!F[idx]->Busy())
        break; // první neobsazené zařízení nalezeno
Seize(F[idx]);
...
Release(F[idx]);
...
```

Když mají všechna zařízení jednu společnou frontu, musíme vyřešit problém uvolňování zařízení. Při sdílené frontě dojde při obsazování z této fronty k výběru toho zařízení, které se jako první uvolnilo a jeho index neodpovídá hodnotě proměnné `idx`. Proto musíme prohledat pole zařízení a zjistit, které z nich uvedený proces obsluhuje:

```
...
for(idx=0; idx < N; idx++) // prohledáme všechna zařízení
    if( F[idx].in == this ) // tento proces je obsluhován zařízením
        break;
Release(F[idx]);
...
```

Výběr zařízení podle délky fronty

Transakce jde do fronty u zařízení s nejkratší frontou. V případě stejné délky front vybere první takové zařízení.

```
const int N = 30; // počet zařízení
Facility F[N];
...
int idx=0;
for (int a=0; a < N; a++)
    if (F[a].QueueLen() < F[idx].QueueLen())
        idx=a; // hledáme minimální frontu
Seize(F[idx]);
...
```

5.5.14 Zpracování výsledků, statistiky

Při simulaci musíme průběžně sledovat stav modelu a zaznamenávat důležité údaje pro pozdější zpracování. Při simulaci systémů hromadné obsluhy obvykle zaznamenáváme statistické údaje. Získané statistiky poskytují následující informace o objektech modelu:

- Celkové vytížení zařízení – jak dlouho z celkové doby simulace bylo zařízení obsazeno.
- Využití kapacity skladů

- Délky front u zařízení a skladů.
- Doby čekání transakcí ve frontách.
- Celková doba, kterou transakce existuje v systému (a poměr doby užitečné činnosti/čekání ve frontě).

Statistiky pro všechny výše uvedené údaje poskytují:

- maximum – nejvyšší zaznamenanou hodnotu
- minimum – nejmenší zaznamenanou hodnotu
- průměrnou hodnotu
- směrodatnou odchylku

Statistiky v SIMLIB

SIMLIB definuje několik tříd pro sběr statistických dat:

- Stat – běžná statistika
- TStat – časově závislá statistika
- Histogram – histogram se statistikou

Pro všechny uvedené třídy jsou definovány základní operace:

- `s.Clear()` – inicializace statistiky
- `s(x)` – záznam další hodnoty `x`
- `s.Output()` – tisk získaných hodnot na výstup

Třída Stat

Objekty třídy Stat uchovávají tyto hodnoty:

- součet vstupních hodnot: $\sum_{i=1}^n x_i$
- součet čtverců vstupních hodnot: $\sum_{i=1}^n x_i^2$
- minimální vstupní hodnotu
- maximální vstupní hodnotu
- počet zaznamenaných hodnot: n

Metoda `Output` tiskne tyto hodnoty a navíc průměrnou hodnotu

$$\mu = \frac{\sum_{i=1}^n x_i}{n}$$

a směrodatnou odchylku

$$s = \sqrt{\frac{\sum_{i=1}^n x_i^2 - n\mu^2}{n-1}}$$

Příklad: Ukázka použití třídy Stat

Pro ilustraci zaznamenáme 1000 čísel vygenerovaných generátorem rovnoměrného rozložení s rozsahem čísel $\langle 0, 100 \rangle$ do statistiky.

```
int main() {
    Stat p;
    for (int a=0; a<1000; a++)
        p(Uniform(0, 100)); // záznam hodnoty
    p.Output();
}
```

Výsledek získaný metodou `Output` je textový:

```
+-----+
| STAT                                     |
+-----+
| Min = 0.403416                        Max = 99.9598 |
| Number of records = 1000              |
| Average value = 49.8424                |
| Standard deviation = 28.8042           |
+-----+
```

Třída TStat

Objekty třídy TStat sledují časový průběh vstupní veličiny. Používají se k výpočtu průměrné hodnoty vstupu za určitý časový interval, například průměrné délky fronty. Objekty třídy TStat uchovávají tyto hodnoty:

- sumu součinu vstupní hodnoty a časového intervalu
- sumu součinu čtverce vstupní hodnoty a časového intervalu
- minimální vstupní hodnotu
- maximální vstupní hodnotu
- počet vstupních hodnot
- počáteční čas

Metoda `Output` tiskne kromě uložených hodnot také průměrnou hodnotu vstupu za čas od inicializace statistiky metodou `Clear` do okamžiku volání metody `Output`. Při vícenásobných experimentech je nutné explicitně volat metodu `Clear` v inicializační části experimentu.

Třída Histogram

V SIMLIB je histogram automaticky doplněn o statistiku. To je výhodné v případě, kdy neodhadneme správně rozsah histogramu a potřebujeme informace pro jeho úpravu. Následující příklad ukazuje záznam 10000 vzorků s exponenciálním rozložením do histogramu:

```
//Histogram("Jméno", OdHodnoty, Krok, PocetTrid);
Histogram expo("Expo", 0, 1, 15);

for (int a=0; a<10000; a++)
    expo(Exponential(3));
```

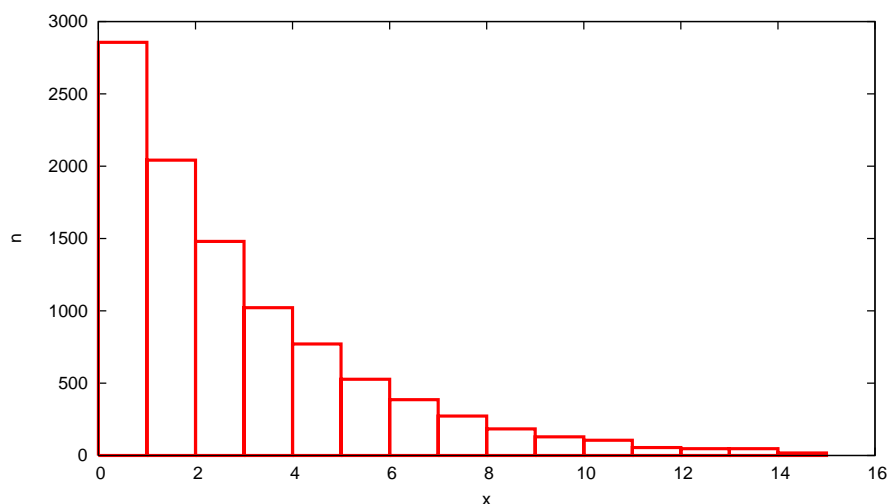
Výsledek:

```
+-----+
| HISTOGRAM Expo                                     |
+-----+
+-----+
| STATISTIC Expo                                     |
+-----+
| Min = 0.00037629                                Max = 24.8161 |
| Number of records = 10000                        |
| Average value = 2.94477                          |
| Standard deviation = 2.91307                     |
+-----+
+-----+-----+-----+-----+-----+
| from | to | n | rel | sum |
+-----+-----+-----+-----+-----+
| 0.000 | 1.000 | 2856 | 0.285600 | 0.285600 |
| 1.000 | 2.000 | 2042 | 0.204200 | 0.489800 |
| 2.000 | 3.000 | 1480 | 0.148000 | 0.637800 |
| 3.000 | 4.000 | 1022 | 0.102200 | 0.740000 |
| 4.000 | 5.000 | 771 | 0.077100 | 0.817100 |
| 5.000 | 6.000 | 527 | 0.052700 | 0.869800 |
| 6.000 | 7.000 | 386 | 0.038600 | 0.908400 |
| 7.000 | 8.000 | 273 | 0.027300 | 0.935700 |
| 8.000 | 9.000 | 184 | 0.018400 | 0.954100 |
| 9.000 | 10.000 | 129 | 0.012900 | 0.967000 |
| 10.000 | 11.000 | 105 | 0.010500 | 0.977500 |
| 11.000 | 12.000 | 55 | 0.005500 | 0.983000 |
| 12.000 | 13.000 | 47 | 0.004700 | 0.987700 |
| 13.000 | 14.000 | 47 | 0.004700 | 0.992400 |
| 14.000 | 15.000 | 17 | 0.001700 | 0.994100 |
+-----+-----+-----+-----+-----+
```

Histogram zaznamenává pro každý interval `<from,to>` počet vstupních hodnot `n`, které padly do tohoto intervalu. Sloupec `sum` označuje součet relativních četností výskytů hodnot `rel` ve všech předchozích intervalech, včetně aktuálního. Obvykle by tento sloupec měl začínat hodnotou nula a končit hodnotou 1.0, aby histogram zachytil všechny zaznamenané údaje. Textové zobrazení histogramu je možné převést do grafické podoby a zobrazit názorněji – viz obrázek 5.1.

Poznámka: Histogram by neměl mít příliš mnoho položek. Jsou známy případy studentů, kteří při odevzdání projektu vytiskli histogram delší než 30 stran A4 nebo





Obrázek 5.1: Grafická podoba histogramu

měli v histogramu jen asi 10 zaznamenaných hodnot. Takové výsledky jsou samozřejmě nevyhovující a podstatně zjednodušují a zrychlují hodnocení...

5.5.15 Příklad systému hromadné obsluhy – samoobsluha

Do samoobsluhy přicházejí zákazníci v intervalech daných exponenciálním rozložením se střední hodnotou 8 minut. Každý zákazník si nejprve vezme vozík ze seřadiště kde je jich celkem 50. Zákazník vstoupí do prodejny a s pravděpodobností 30% jde okamžitě k pultu s lahůdkami, kde obsluhují dvě prodavačky. Obsloužení zákazníka zde trvá průměrně 2 minuty (exponenciální rozložení) a pak zákazník pokračuje běžným nákupem. Běžný nákup trvá 10-15 minut s rovnoměrným rozložením. Nakonec zákazník platí u jedné z pěti pokladen, kterou si vybere podle nejkratší fronty. Doba obsluhy u pokladny se řídí exponenciálním rozložením se střední hodnotou 3 minuty. Při odchodu ze systému zákazník vrací vozík.

$$x+y$$

Analýza systému

- Vozíky: sklad 50 kusů, jedna fronta
- Lahůdky: sklad s kapacitou 2 (prodavačky), jedna fronta
- Pokladny: 5 zařízení (pokladen), každé má zvláštní frontu
- Transakce provádí následující činnost:
 1. příchody: exponenciální rozložení, střední hodnota 8 minut
 2. zabrat vozík,
 3. 30% k lahůdkám, obsluha: exponenciální rozložení, střední hodnota 2 minuty
 4. nákup: rovnoměrné rozložení 10-15 minut
 5. výběr pokladny podle délky fronty
 6. placení: exponenciální rozložení, střední hodnota 3 minuty
 7. vrácení vozíku

Implementace modelu v SIMLIB/C++

```
#include "simlib.h"

const int POC_POKLADEN = 5;

// zařízení:
Facility Pokladny[POC_POKLADEN];
Store Lahudky("Oddělení lahůdek", 2);
Store Voziky("Seřadiště vozíků", 50);

Histogram celk("Celková doba pobytu v systému", 0, 10, 30);
```

První část modelu obsahuje definice konstant a objektů modelu včetně histogramu celkové doby pobytu zákazníků (transakcí) v systému.

```
class Zakaznik : public Process {
    void Behavior() {
        double prichod = Time;        // čas příchodu
        Enter(Voziky, 1);
        if ( Random() <= 0.30 ) {      // pravděpodobnost 30%
            Enter(Lahudky, 1);
            Wait(Exponential(2));
            Leave(Lahudky, 1);
        }
        Wait(Uniform(10, 15));         // nákup
        // výběr pokladny podle délky fronty
        int idx = 0;
        for (int a=0; a < POC_POKLADEN; a++)
            if (Pokladny[a].QueueLen() < Pokladny[idx].QueueLen())
                idx = a;
        Seize(Pokladny[idx]);          // vstup do fronty u pokladny
        Wait( Exponential(3) );        // placení
        Release(Pokladny[idx]);
        Leave(Voziky, 1);
        celk(Time-prichod);            // záznam doby strávené v systému
    }
};
```

Chování zákazníků je popsáno formou procesu. Zákazník si při příchodu poznačí čas, aby bylo možné při odchodu vypočítat dobu po kterou byl v systému a zaznamenat ji do histogramu.

```
class Prichody : public Event {
    void Behavior() {
        (new Zakaznik)->Activate();
        Activate( Time + Exponential(8) );
    }
};
```

Generování příchodů zákazníků je běžná periodicky se opakující událost. Vytvořený zákazník je ihned aktivován.

```
int main() // popis experimentu
{
    SetOutput("Samoo.dat");
    Init(0, 1000);
    (new Prichody)->Activate(); // start generátoru
    Run(); // běh simulace

    // tisk statistik:
    celk.Output();
    Lahudky.Output();
    Voziky.Output();
    for (int a=0; a < POC_POKLADEN; a++)
        Pokladny[a].Output();
}
```

Popis experimentu přesměruje výstup do souboru "Samoo.dat" a po ukončení simulace tiskneme statistická data do tohoto souboru voláním metod `Output`.

5.6 Shrnutí

Tato kapitola je orientována především na modely systémů hromadné obsluhy. Obsahuje klasifikaci SHO, popis jejich typické struktury a příklady jednotlivých částí modelů. Pro pochopení funkce diskrétních simulačních systémů je důležitá také znalost základních algoritmů řízení diskrétní simulace. V této kapitole byl prezentován algoritmus založený na kalendáři událostí, který je typicky využíván při implementaci simulátorů diskrétních simulačních modelů. Část kapitoly je věnována popisu jednoduchého simulačního nástroje – knihovny SIMLIB/C++ včetně příkladů jejího praktického použití. Získané znalosti je možné použít pro efektivní návrh a implementaci simulačních modelů a jednoduchých simulačních systémů.

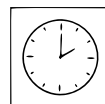
5.7 Otázky a úkoly

1. Co je systém hromadné obsluhy? Definujte jeho komponenty.
2. Co je Kendallova klasifikace?
3. Co znamená M/M/2 ?
4. Vyjmenujte typy obslužných linek.
5. K čemu používáme priority v SHO?
6. Jaké typy výsledků získáme při simulaci SHO?
7. Vytvořte model vhodně zvoleného SHO v SIMLIB/C++, proveďte několik experimentů a analyzujte výsledky.
8. Naprogramujte jednoduchý diskrétní simulační systém s kalendářem událostí. Použijte popis chování formou událostí. Vytvořte v něm model jednoduché obslužné linky s frontou.



Kapitola 6

Spojitá simulace



4:30

Cílem této kapitoly je seznámení se spojitou simulací. Jsou uvedeny metody pro popis spojitých modelů, ukázka implementace simulátoru a příklady modelů. Obsahem této kapitoly jsou základní informace o tom, jaké jsou aplikace spojité simulace, formy popisu spojitých modelů, metody převodu diferenciálních rovnic vyšších řádů na soustavu rovnic prvního řádu, přehled numerických metod, spojité simulační jazyky a příklady.

6.1 Aplikační oblasti spojité simulace

Spojitá simulace má dlouhou tradici v celé řadě oborů. Spojité modely používáme například pro aplikace v následujících oblastech:

- Modely řízení systémů v průmyslu (strojírenství, chemický průmysl, ...)
- Modely elektrických obvodů.
- Modelování přírodních dějů ve fyzice, astronomii – modely pohybu planet, modely nanotechnologických systémů
- Simulační modely v oblasti biologie, ekologie – modely růstu mikroorganismů, modely spalování odpadu, ...
- Počítačové hry – například simulátory letadel.

6.2 Formy popisu spojitých systémů

V tomto textu si ukážeme použití:

- soustav obyčejných diferenciálních rovnic
- soustav algebraických rovnic
- blokových schemat
- parciálních diferenciálních rovnic (pouze orientačně)

a možnosti kombinací uvedených forem popisu. Většina složitých simulací (počasí, zemětřesení, deformace materiálů) vyžaduje použití parciálních diferenciálních rovnic, kterým se bohužel nemůžeme věnovat, protože vyžadují použití náročných matematických postupů.

Spojité systémy je možné popisovat i jinými formalismy – existují například Bond grafy (anglicky: Bond graphs) nebo spojité varianty Petriho sítí.

6.3 Soustavy obyčejných diferenciálních rovnic

Obyčejné diferenciální rovnice popisují závislosti změny veličin – vyskytují se v nich derivace proměnných (obvykle podle času). Například napětí na cívce můžeme popsat diferenciální rovnicí

$$U_L = L \frac{di}{dt}$$

kde L je indukčnost cívky a i je proud tekoucí cívkou. Typicky jsou složité systémy popsány soustavou¹ diferenciálních a algebraických rovnic, pro kterou můžeme použít například maticový popis:

$$\begin{aligned} \frac{d}{dt} \vec{w}(t) &= \mathbf{A}(t) \vec{w}(t) + \mathbf{B}(t) \vec{x}(t) \\ \vec{y}(t) &= \mathbf{C}(t) \vec{w}(t) + \mathbf{D}(t) \vec{x}(t) \end{aligned}$$

kde:

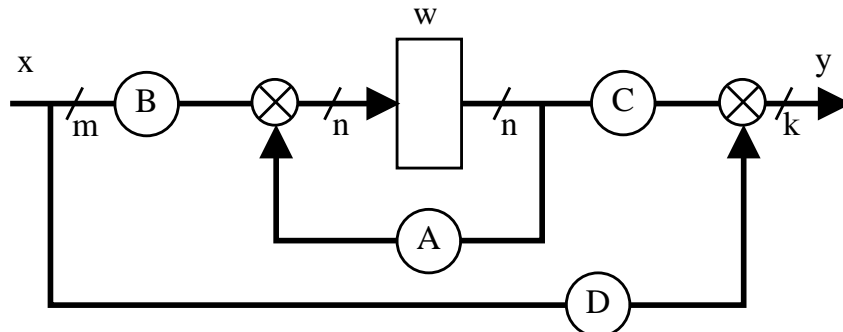
\vec{x} je vektor m vstupů,

\vec{w} vektor n stavových proměnných (výstupy integrátorů),

\vec{y} vektor k výstupů a

$\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}$ jsou matice koeficientů.

Tuto soustavu rovnic můžeme vyjádřit i formou blokového schématu:



Obě tyto formy popisu jsou ekvivalentní a navzájem převoditelné. Maticový popis systémů je snadno použitelný při modelování v systémech typu Matlab [11] (Scilab [12], Octave [13]). Některé nadstavby (Simulink, Scicos) nad těmito systémy umožňují zadávat modely i grafickou formou.

Typy diferenciálních rovnic

Obyčejné diferenciální rovnice (ODR, anglicky: ODE = Ordinary Differential Equations) klasifikujeme podle koeficientů (prvků matic $\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}$):

- Konstantní koeficienty popisují lineární systémy. Příklad rovnice: $y' + 5y = 0$, kde y' znamená první derivaci y podle času.
- Obecné nelineární funkce popisují nelineární systémy. Pro tyto systémy obvykle nemáme analytické řešení. Příklad rovnice: $y'' + \sin(y) = 0$.

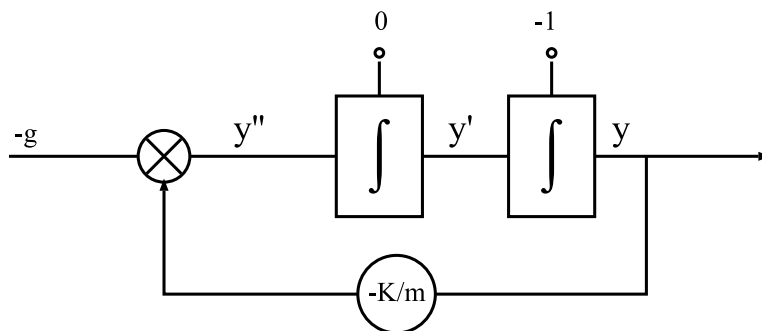
¹V náročných průmyslových aplikacích je běžné, že celkový počet rovnic v popisu systému dosahuje desítek až stovek tisíc.

Další klasifikace bere v úvahu závislost koeficientů na čase:

- nezávislé na čase (stacionární systémy) – koeficienty nesmí být funkcí času, ale mohou být funkcí stavu nebo vstupu.
- časově proměnné – v maticích se vyskytují koeficienty popsané funkcí času. Typickým příkladem jsou Besselovy diferenciální rovnice.

6.4 Grafový popis s použitím bloků

Tento popis vychází z postupů používaných při simulaci na analogovém počítači. Následující příklad demonstruje použití blokového popisu při popisu mechanického systému (závaží na pružině):



Tento systém je popsateľný diferenciální rovnicí 2. řádu:

$$y'' + \frac{K}{m}y + g = 0$$

s počátečními podmínkami $y_0 = -1$ a $y'_0 = 0$

Grafový popis je velmi názorný, ale pro velké soustavy rovnic musí být hierarchicky strukturován aby bylo možné jeho přehledné zobrazení na monitoru počítače.

6.4.1 Typy spojených bloků

Bloky rozdělujeme do několika základních kategorií:

- Funkční bloky (nazýváme je též *bezpečkové*, protože nemají vnitřní stav – jejich výstup je pouze funkcí vstupů):
 - konstanty (nemají žádný vstup)
 - násobení konstantou je obvykle zvláštní blok (jeden vstup, jeden parametr)
 - běžné aritmetické operace $+$, $-$, $*$, $/$ mají dva a více vstupů
 - standardní funkce: \sin , \cos , ...
 - uživatelem definované funkce (obvykle složené ze standardních funkcí a výrazů nebo definované tabulkou, obecně mohou mít n vstupů)
- Stavové bloky (paměťové, mají vnitřní stav, který musíme nastavit před spuštěním simulace – říkáme, že nastavujeme *počáteční podmínky*) produkují výstup, který kromě vstupu závisí i na stavu bloku:

- integrátory – počítají integrál vstupu, musíme zadat počáteční hodnotu výstupu.
- stavové proměnné – například stav relé, hystereze a podobné bloky, které mají paměť.
- zpoždění – posouvá vstupní signál v čase, musí si pamatovat celý vstup po dobu danou parametrem udávajícím velikost zpoždění.

Bloky jsou obvykle součástí knihoven modelů v simulačních systémech, můžeme je skládat do hierarchických struktur a vytvářet tak složité modely. Například Modelica Standard Library [14] obsahuje řadu komponent, které je možné skládat do větších celků.

6.5 Převod rovnic vyššího řádu na soustavu rovnic 1. řádu

Rovnice vyšších řádů musíme převést na soustavu rovnic prvního řádu, protože máme vhodné numerické metody pouze pro řešení rovnic prvního řádu². Každou rovnici vyššího řádu je možné transformovat na soustavu rovnic prvního řádu a z těchto rovnic můžeme snadno vytvořit blokové schema. V tomto textu se zaměříme na dvě základní metody převodu:

- *Metoda snižování řádu derivace* je jednodušší, ale vyžaduje, aby na pravé straně rovnice nebyly derivace vstupu.
- *Metoda postupné integrace* zvládne i rovnice s derivacemi vstupu na pravé straně.

Obě metody využívají úprav rovnice a zavádění pomocných proměnných. Jsou použitelné i na soustavy rovnic vyššího řádu – stačí postup opakovat pro každou zadanou rovnici.

6.5.1 Metoda snižování řádu derivace

Pro převod obecné rovnice tvaru:

$$ay^{(n)} + by^{(n-1)} + \dots + ky' + zy = f(x, y, y', \dots)$$

postupujeme ve dvou krocích:

1. Upravíme rovnici tak, aby na levé straně byl pouze nejvyšší řád derivace.

$$y^{(n)} = \dots$$

2. Protože máme n -tou derivaci, můžeme sekvencí integrátorů získávat nižší derivace, dokud nezískáme y .

$$\begin{aligned} y^{(n-1)} &= \int y^{(n)} dt \\ y^{(n-2)} &= \int y^{(n-1)} dt \\ &\dots \\ y &= \int y' dt \end{aligned}$$

²Existují například metody pro řešení rovnic druhého řádu ale ty se používají pouze ve speciálních případech.



Pravé strany těchto rovnic můžeme nahradit pomocnými proměnnými, které reprezentují výstupy integrátorů (stavové proměnné), ale není to nutné.

Poznámka: Podmínkou použitelnosti metody je, že na pravé straně nesmí být žádné derivace vstupů (x' , x'' , ...).

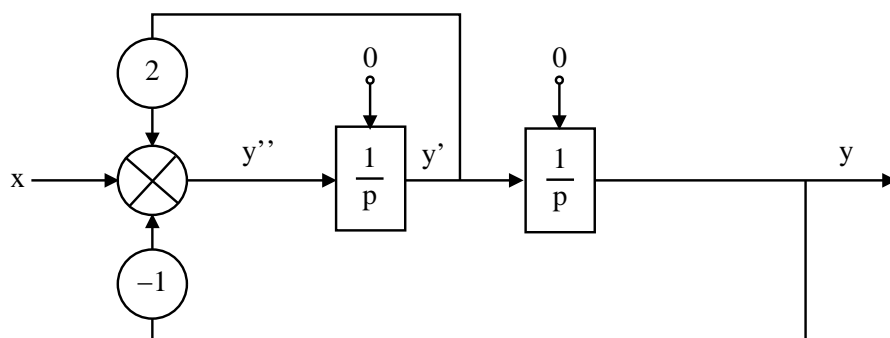
Příklad: Převod rovnice druhého řádu

Máme rovnici $y'' - 2y' + y = x$, počáteční podmínky jsou nulové. Po úpravě metodou snižování řádu derivace získáme tuto soustavu rovnic:

$$\begin{aligned} y'' &= 2y' - y + x \\ y' &= \int y'' \\ y &= \int y' \end{aligned}$$

$x + y$

Blokové schéma odpovídající zadané rovnici je uvedeno na obrázku 6.1. Symbol $\frac{1}{p}$ označuje blok typu integrátor, který provádí numerickou integraci vstupu a má vnitřní stav, který musí být nastaven před začátkem simulace.



Obrázek 6.1: Výsledné blokové schéma

Poznámky:

- Blokové schéma při použití této metody má vždy typickou strukturu – řadu integrátorů propojených za sebou, na vstupu prvního je blokový výraz počítající nejvyšší derivaci, na výstupu posledního je y . Počet integrátorů je dán nejvyšším řádem derivace v původní rovnici.
- Musíme dávat pozor na nastavení počátečních podmínek pro každý integrátor. Pokud počáteční podmínky nejsou uvedeny, předpokládáme, že jsou nulové.

6.5.2 Metoda postupné integrace

Tato metoda je vhodná i pro rovnice s derivacemi vstupu x na pravé straně. Pro převod obecné rovnice tvaru:

$$ay^{(n)} + by^{(n-1)} + \dots + ky' + zy = f(x, x', x'', \dots, y, y', \dots)$$

postupujeme takto:

1. Upravíme rovnici tak, aby na levé straně byl pouze nejvyšší řád derivace.

$$y^{(n)} = \dots$$

2. Integrujeme obě strany rovnice

$$y^{(n-1)} = \text{vyraz1} + \int \text{vyraz2} \, dt$$

a průběžně zavádíme nové stavové proměnné w_i pro ty podvýrazy, kde zůstane integrál (může zůstat jen na pravé straně):

$$w_i = \int \text{vyraz2} \, dt$$

Nahrazením podvýrazu touto proměnnou dostaneme rovnici

$$y^{(n-1)} = \text{vyraz1} + w_i$$

a postupně opakujeme integraci a zavádění stavových proměnných dokud nemáme rovnici pro výpočet y .

3. Nakonec provedeme výpočet nových počátečních podmínek (pro čas t_0) pro všechny pomocné stavové proměnné, které jsme zavedli.

$$w_i(t_0) = y^{(n-1)}(t_0) - \text{vyraz1}(t_0)$$

Hodnotu $y^{(n-1)}(t_0)$ známe – jsou to původní počáteční podmínky.

Metoda postupné integrace je použitelná za následujících podmínek:

- V rovnici musí být (u derivací) pouze konstantní koeficienty (jinak nelze provádět další úpravy rovnice).
- Nejvyšší řád derivace vstupu x musí být menší než nejvyšší řád derivace y , protože jinak by nám zůstaly některé derivace vstupu ve výsledných rovnicích. Tyto derivace by vyžadovaly implementaci derivačního bloku v simulačním systému. Numerický výpočet derivace je sice možný, ale způsobuje problémy, protože výrazně zhoršuje přesnost výpočtu.

Příklad: Převod rovnice druhého řádu

Pro tento příklad použijeme rovnici zapsanou v operátorovém tvaru³:

$$p^2 y + 2py + y = p^2 x + 3px + 2x$$

Postup:

1. Osamostatníme nejvyšší řád derivace:

$$p^2 y = p^2 x + p(3x - 2y) + (2x - y)$$

2. Provedeme integraci obou stran (v operátorovém tvaru jde o dělení p):

$$py = px + (3x - 2y) + \frac{1}{p}(2x - y)$$

³Viz Laplaceova transformace, kterou znáte z matematiky.

Zavedeme pomocnou proměnnou

$$w_1 = \frac{1}{p}(2x - y)$$

Ještě jednou opakujeme integraci:

$$y = x + \frac{1}{p}(3x - 2y + w_1)$$

Další pomocná stavová proměnná

$$w_2 = \frac{1}{p}(3x - 2y + w_1)$$

Nyní už máme výslednou soustavu rovnic:

$$\begin{aligned} w_1 &= \frac{1}{p}(2x - y) \\ w_2 &= \frac{1}{p}(3x - 2y + w_1) \\ y &= x + w_2 \end{aligned}$$

3. Zbývá stanovit počáteční podmínky integrátorů. Máme zadány počáteční podmínky pro y :

$$y'(0) = y'_0$$

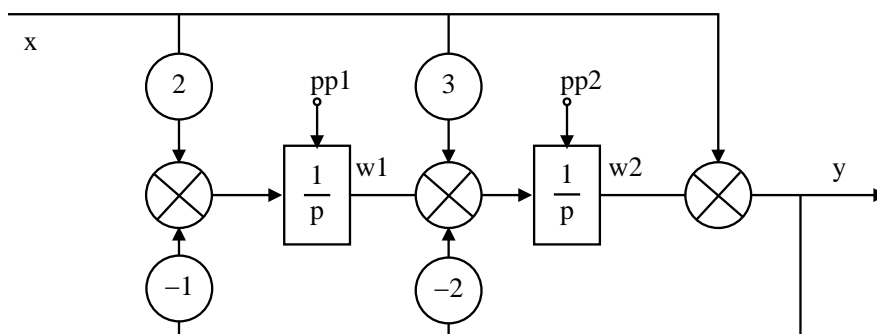
$$y(0) = y_0$$

Z toho vypočítáme počáteční hodnoty pouhým dosazením a úpravou výchozích rovnic z kroku 2 (předpokládáme nulovou počáteční hodnotu vstupu $x(0) = 0$ a nulové všechny derivace x):

$$pp2 = w_2(0) = y_0$$

$$pp1 = w_1(0) = y'_0 + 2y_0$$

Výsledné blokové schema je na obrázku 6.2. Tato metoda vede na jiný typický tvar blokového schematu než metoda snižování řádu derivace.



Obrázek 6.2: Blokové schema získané metodou postupné integrace

6.6 Numerické metody

Při spojitě simulaci potřebujeme metody pro

- řešení obyčejných diferenciálních rovnic (ODR) prvního řádu (řešíme úlohu s počátečními podmínkami⁴, anglicky: *Initial value problem*),
- řešení algebraických rovnic,
- řešení parciálních diferenciálních rovnic různých typů – tuto rozsáhlou oblast záměrně vynecháme a uvedeme pouze jeden ilustrační příklad.

Často je třeba použít více typů metod současně. Typické jsou například metody pro řešení algebraicko-diferenciálních rovnic.

Numerické metody jsou popsány v rozsáhlé literatuře. Existuje řada volně dostupných implementací různých metod (viz např. Netlib[16]). V tomto textu uvedeme pouze stručné shrnutí principů a základních vlastností vybraných numerických integračních metod. V praxi je nutné dobře znát vlastnosti používaných metod, protože tak se vyhneme nepříjemným překvapením způsobeným chybnými výsledky, i když máme validní model.

6.6.1 Metody pro řešení ODR prvního řádu

Hledáme řešení obyčejné diferenciální rovnice

$$y' = f(t, y)$$

které má tvar:

$$y(T) = y_0 + \int_0^T f(t, y) dt$$

Na číslicovém počítači je řešení aproximováno v bodech $t_0, t_1, t_2, \dots, t_n$. Interval mezi těmito body nazýváme *integrační krok*: $h_i = t_{i+1} - t_i$. Většina existujících integračních metod dovoluje měnit velikost integračního kroku v průběhu simulace, některé automaticky upravují krok tak, aby dosáhly požadované přesnosti.

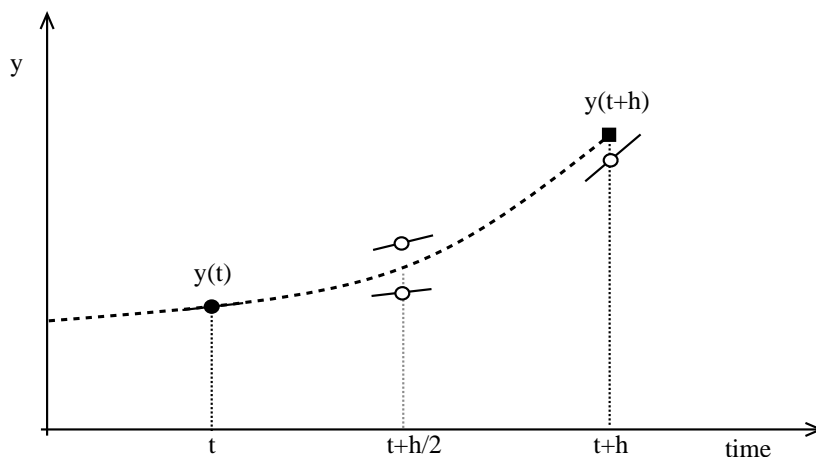
6.6.2 Princip a klasifikace numerických integračních metod

Obecný princip metody N -tého řádu:

1. Aproximace $y(T)$ polynomem N -tého stupně (Taylorův rozvoj) na základě aktuální hodnoty $y(t)$, kterou známe (buď jde o počáteční stav, nebo výsledek předchozích kroků). Stupeň polynomu N definuje tzv. *řád metody*.
2. Extrapolace – výpočet nové hodnoty $y(t + h)$.

Integrační metody můžeme rozdělit na:

- *jednokrokové* – vychází jen z aktuálního stavu $y(t)$
- *vícekrokové* – používají historii stavů $y(t), y(t - h), y(t - 2h) \dots$ nebo vstupů. Tyto metody mají problém při startu, kdy nemáme k dispozici historii.



Obrázek 6.3: Princip jednokrokových metod

6.6.3 Jednokrokové metody

Princip jednokrokových metod je znázorněn na obrázku 6.3. 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 počítají ještě s několika pomocnými body uvnitř kroku.

6.6.4 Eulerova metoda

Tato metoda je nejjednodušší protože počítá výsledek pouze s využitím derivace ve výchozím bodu – viz obrázek 6.4. Vzorec pro výpočet hodnoty na konci kroku:

$$y(t+h) = y(t) + hf(t, y(t))$$

kde $f(t, y(t))$ je hodnota derivace na začátku kroku.

Eulerova metoda je velmi jednoduchá, ale v praxi ji téměř nepoužíváme, protože není dostatečně efektivní.

Ukázka implementace Eulerovy metody

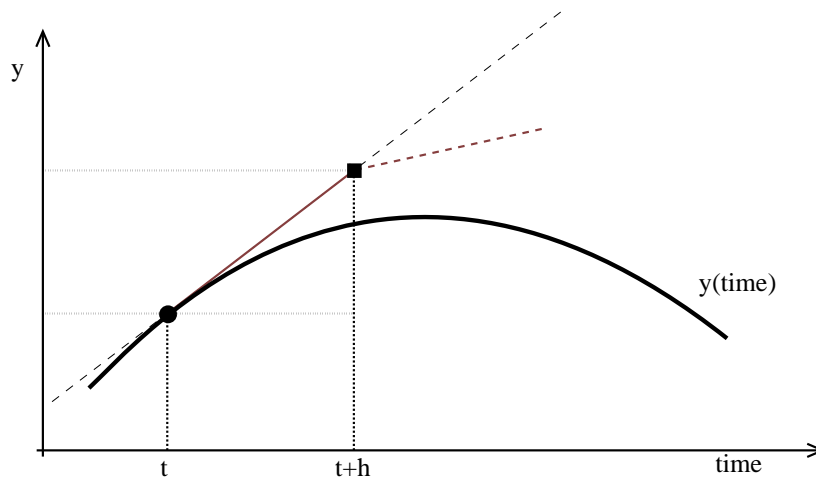
Následující kód ukazuje použití Eulerovy metody pro simulaci jednoduchého systému – jde o tzv. *kruhový test* popsáný rovnicí $y'' + y = 0$ (počáteční podmínka $y(0) = 1$), jehož analytickým řešením je $y(t) = \cos(t)$.

Kód je napsán v jazyku ISO C99, stavy integrátorů jsou v poli `y`, vstupy integrátorů (pravé strany rovnic) musíme při výpočtu uložit do pole `yin`.

```
double yin[2];           // vstupy integrátorů
double y[2] = { 1.0, 0.0 }; // počáteční podmínky
double time = 0;         // modelový čas
```

⁴Upozornění: Je třeba rozlišovat numerické integrační metody pro řešení ODR od numerických metod pro výpočet určitých integrálů. Tyto dvě třídy metod řeší různě formulované problémy.





Obrázek 6.4: Princip Eulerovy metody

```
double h = 0.01;           // krok

void update() {
    // Popis systému: výpočet VSTUPŮ integrátorů
    yin[0] = y[1];          // y'
    yin[1] = -y[0];         // y'' = -y
}

void integrate_euler() { // krok integrace:
    update();              // výpočet aktuálních vstupů
    for (int i = 0; i < 2; i++) // postupně pro všechny integrátory
        y[i] += h * yin[i];    // výpočet nového stavu
    time += h;              // posun času
}

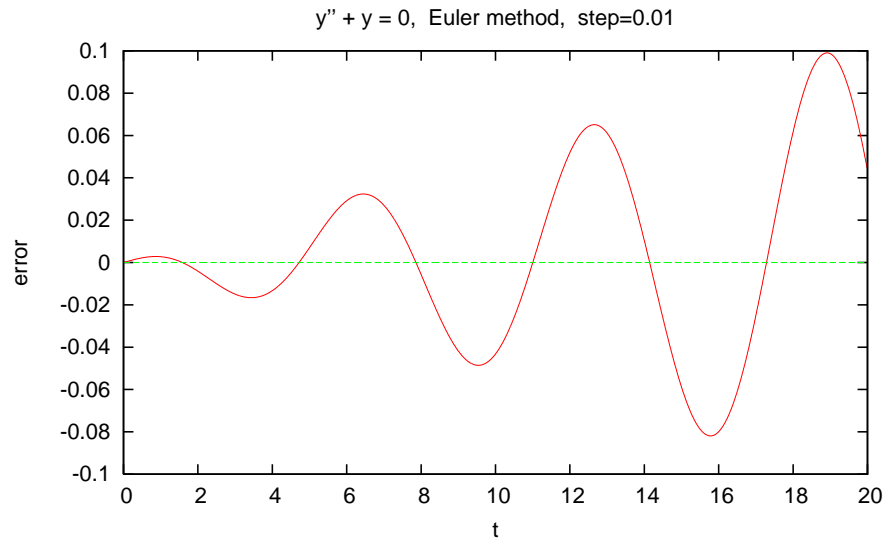
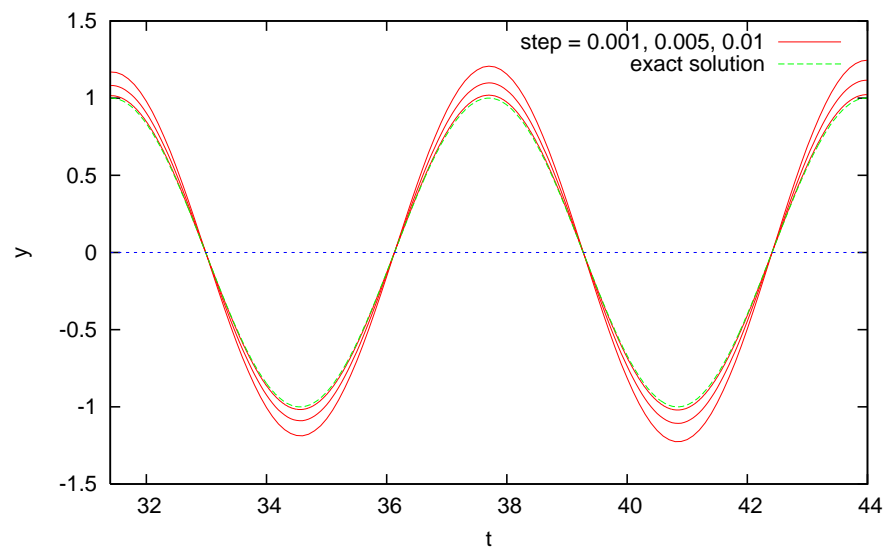
int main() { // popis experimentu
    while (time < 20) {
        printf("%10f %10f\n", time, y[0]);
        integrate_euler();
    }
    return 0;
}
```

Příklad: chyba Eulerovy metody

Protože známe přesné analytické řešení našeho příkladu, můžeme sledovat absolutní chybu Eulerovy metody na obrázku 6.5. Je zřejmé, že chyba narůstá s počtem kroků.

Příklad: závislost chyby na kroku

Výsledky stejného příkladu pro tři různé velikosti kroku ukazují jak roste odchylka od přesného řešení s rostoucí délkou kroku. Obrázek 6.6 zobrazuje pouze část řešení – osa x nezačíná nulou.

Obrázek 6.5: Rovnice $y'' + y = 0$, Eulerova metoda, chyba pro krok 0.01Obrázek 6.6: Řešení rovnice $y'' + y = 0$, Eulerova metoda, různý krok

6.6.5 Metody Runge-Kutta

Tyto metody provádí další výpočty uvnitř kroku a tím dosahují při stejné délce kroku podstatně přesnějších výsledků než Eulerova metoda. Existuje celá řada variant metod Runge-Kutta, zde si ukážeme dvě základní:

- RK2 je metoda druhého řádu a pro výpočet používá jeden pomocný bod uprostřed kroku:

$$k_1 = hf(t, y(t))$$

$$k_2 = hf(t + \frac{h}{2}, y(t) + \frac{k_1}{2})$$

$$y(t+h) = y(t) + k_2$$

- RK4 je metoda čtvrtého řádu:

$$k_1 = hf(t, y(t))$$

$$k_2 = hf(t + \frac{h}{2}, y(t) + \frac{k_1}{2})$$

$$k_3 = hf(t + \frac{h}{2}, y(t) + \frac{k_2}{2})$$

$$k_4 = hf(t + h, y(t) + k_3)$$

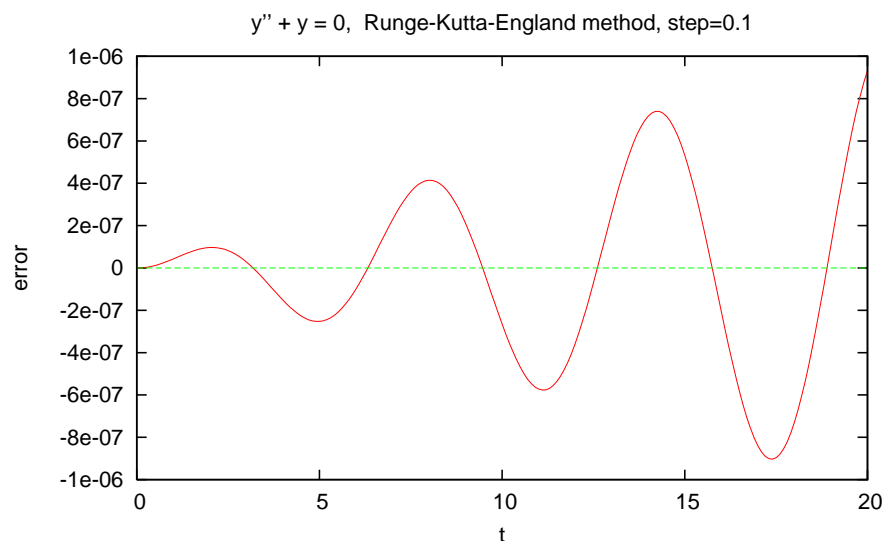
$$y(t+h) = y(t) + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6}$$

Různé další varianty naleznete v literatuře. Tyto metody jsou velmi často používány – prakticky každý spojitý simulační systém obsahuje alespoň jednu RK metodu.

Metody Runge-Kutta je možné doplnit o výpočet odhadu chyby metody, což dovoluje průběžně měnit délku kroku, aby se chyba udržovala v zadaných mezích. Proměnná délka kroku vede k efektivnější simulaci, protože lze použít větší krok a metoda si sama rozhodne o jeho zkrácení pokud to je v některých částech výpočtu nutné.

Přesnost metody Runge-Kutta

Metody Runge-Kutta jsou při stejném kroku přesnější než Eulerova metoda. Na obrázku 6.7 zobrazujícím použití jedné z variant metody Runge-Kutta 4. řádu pro náš testovací příklad je chyba o několik řádů menší i při desetkrát delším kroku metody.



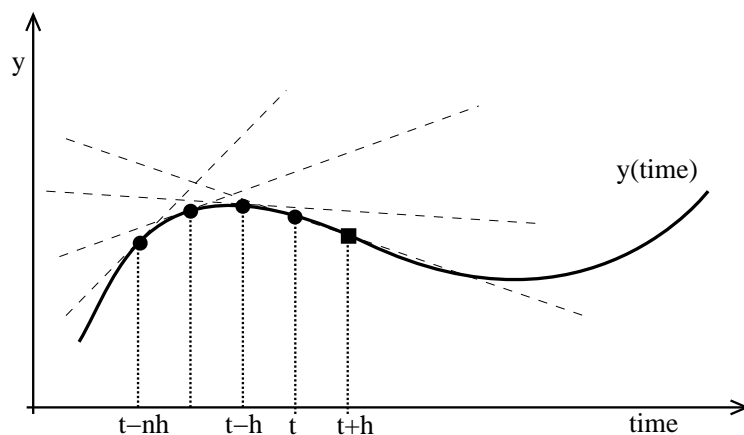
Obrázek 6.7: Rovnice $y'' + y = 0$, metoda RKE, chyba pro krok 0.1

6.6.6 Vícekrokové metody

Tyto metody používají hodnoty z předchozích kroků pro výpočet hodnoty následující – viz obrázek 6.8. Počet použitých předchozích hodnot závisí na řádu metody. Obecně platí že přesnost metody roste s řádem metody, ale roste také výpočetní náročnost.

Tato třída metod musí řešit problém startu metody – prvních N kroků obvykle počítáme použitím některé jednokrokové metody⁵. Tento problém znesnadňuje použití

⁵Existují také samostartující vícekrokové metody.



Obrázek 6.8: Princip vícekrokových metod

těchto metod při změnách délky kroku, což je významné například při kombinované simulaci.

Příkladem typické vícekrokové metody je metoda Adams-Bashforth která používá kromě aktuální hodnoty derivace ještě 3 hodnoty z minulých kroků.

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

y_n je hodnota na začátku kroku, y_{n+1} je výsledná hodnota na konci kroku.

Existuje řada různých variant vícekrokových metod, například metody typu prediktor/korektor zpřesňují výsledek dalšími dodatečnými výpočty se zahrnutím právě vypočítané hodnoty y_{n+1} . Například metoda Adams-Bashforth-Moulton přidává k základu danému výše uvedenou metodou Adams-Bashforth tento korektor:

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

6.6.7 Vlastnosti integračních metod

Nejdůležitější vlastností numerických integračních metod je jejich přesnost. Další důležitou vlastností je stabilita numerického řešení při použití dané metody. Nezanedbatelná je také složitost implementace metod.



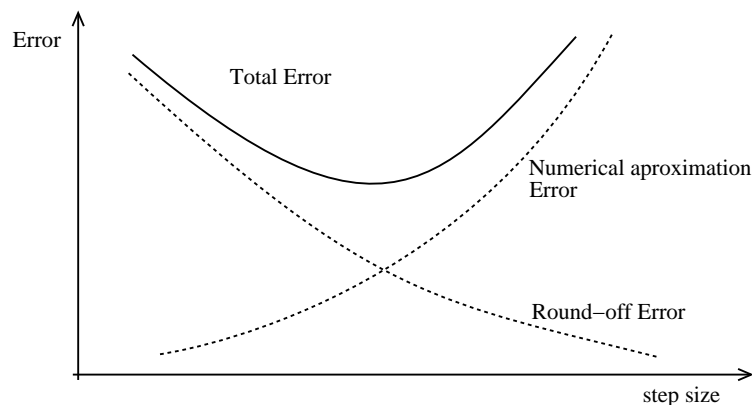
Chyba numerické metody

Výsledná chyba metody je dána součtem několika vlivů:

- Lokální chyby (v jednom kroku)
 - Chyba zaokrouhlovací (round-off error) je závislá na přesnosti aritmetiky počítače. Například počítání s přesností float (24 bitů) je poznamenáno touto chybou podstatně více než počítání s přesností double (53 bitů).
 - Chyba metody numerické aproximace (truncation error) je dána především řádem metody, tj. počtem členů Taylorova rozvoje který použijeme. Metody vyšších řádů mají obecně vyšší přesnost při stejné délce kroku.

- Akumulovaná chyba roste v průběhu výpočtu, protože se průběžně sčítají vlivy lokálních chyb.

Vliv lokálních chyb na přesnost metody v závislosti na délce kroku znázorňuje obrázek 6.9. Je zřejmé, že existuje určitý optimální rozsah délky kroku, ve kterém je celková



Obrázek 6.9: Závislost chyby numerických metod na délce kroku

chyba minimální. Nemá smysl volit příliš malý krok, protože je výpočet méně efektivní a při extrémně malém kroku i méně přesný.

Stabilita numerické metody

Další důležitou vlastností je stabilita metody, která vyjadřuje chování metody při řešení některých problémů v závislosti na délce integračního kroku. Chování metody se může při zvětšování kroku skokově změnit – řešení se stane nestabilní a dostáváme zcela chybné výsledky. Pro některé metody a typy problémů nemusí stabilní řešení dokonce vůbec existovat.

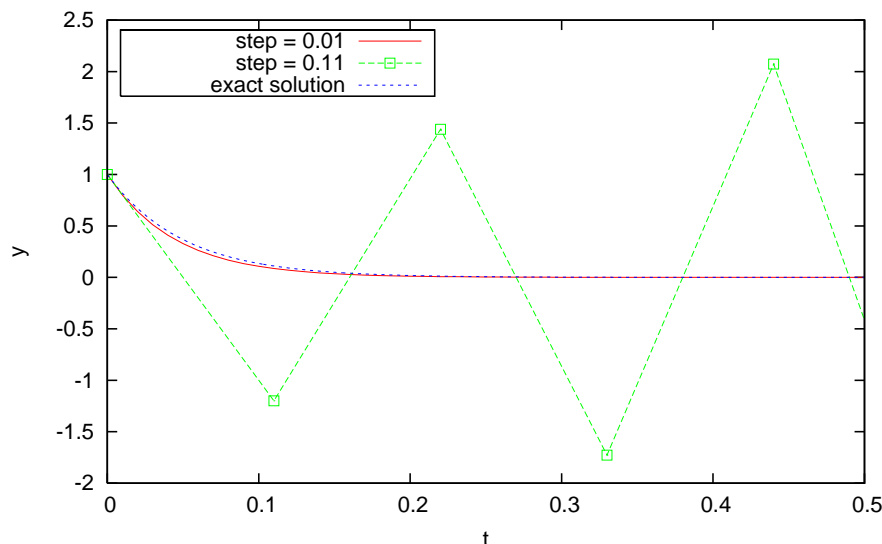
Příklad: Ukázka nestabilního chování numerické metody

Řešení rovnice $y' = -20y$ s počáteční podmínkou $y(0) = 1$ Eulerovou metodou vede na nestabilní řešení – viz obrázek 6.10.

6.6.8 Tuhé systémy

Při modelování některých systémů může dojít k tomu, že jejich popis chování zahrnuje jevy s velmi rozdílnými časovými konstantami. Při jejich numerickém řešení vzniká problém s volbou délky integračního kroku. Když zvolíme krok podle rychlých dějů, je výpočet neefektivní a narůstá akumulovaná chyba. Naopak pro dlouhý krok nedostaneme při použití běžných metod dobré výsledky. Proto existuje celá řada speciálních metod, které dovolují výpočet s dlouhým krokem i pro tyto tzv. "tuhé systémy" (anglicky: stiff systems).

Tuhost soustavy rovnic vyjadřujeme koeficientem tuhosti, který můžeme neformálně definovat jako poměr časových konstant nejrychlejších a nejpomalejších jevů. S tuhými systémy se v praxi často setkáváme například v oblasti modelů chemických reakcí, řídicích systémů, elektrických obvodů atd. Jednoduchým příkladem popisu tuhého systému může být rovnice $y'' + 101y' + 100y = 0$



Obrázek 6.10: Ukázka nestabilního řešení Eulerovou metodou

6.6.9 Výběr integrační metody

Při výběru integrační metody pro řešení problému si musíme uvědomit, že neexistuje univerzální (nejlepší) metoda. Často je nejvhodnější vyzkoušet několik metod, porovnat jejich výsledky a vybrat tu nejefektivnější. Je třeba respektovat několik základních doporučení:

- Obvykle vyhovuje některá varianta metody Runge-Kutta (4. řádu).
- Nespojité ve funkci $f(t, y)$ znemožňují nebo ztěžují použití více krokových metod.
- Tuhé systémy vyžadují speciální metody.
- Je třeba vyzkoušet různé integrační metody a různé velikosti kroku.
- Existuje horní limit velikosti kroku daný například požadovanou přesností nebo intervalem výstupu výsledků.

6.7 Příklad: Systém dravec–kořist

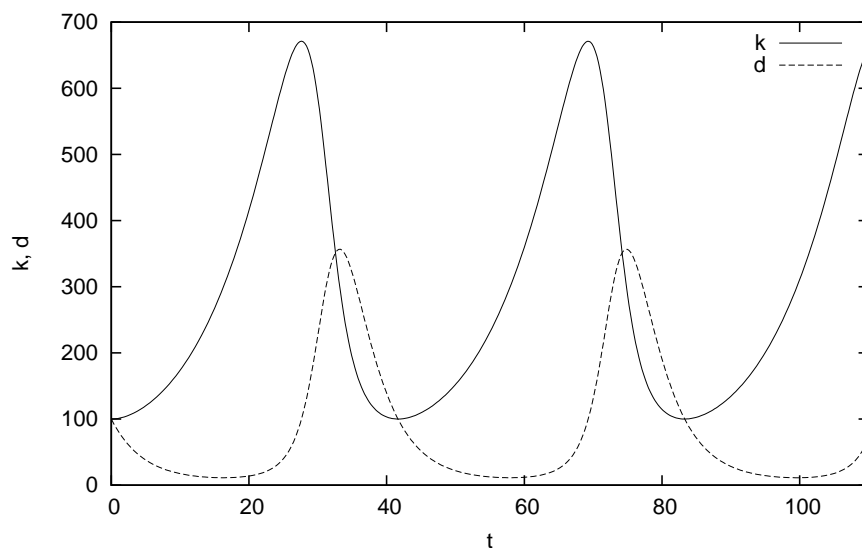
Model systému dravec–kořist⁶ vytvoříme jako spojitý dynamický systém, ve kterém dvě stavové proměnné reprezentují množství dravců a množství kořisti. Diferenciální rovnice

$$\begin{aligned}\frac{dx_k}{dt} &= ax_k - bx_kx_d \\ \frac{dx_d}{dt} &= cx_kx_d - dx_d\end{aligned}$$

popisují vliv počtu dravců x_d a kořisti x_k na změny počtu dravců a kořisti.

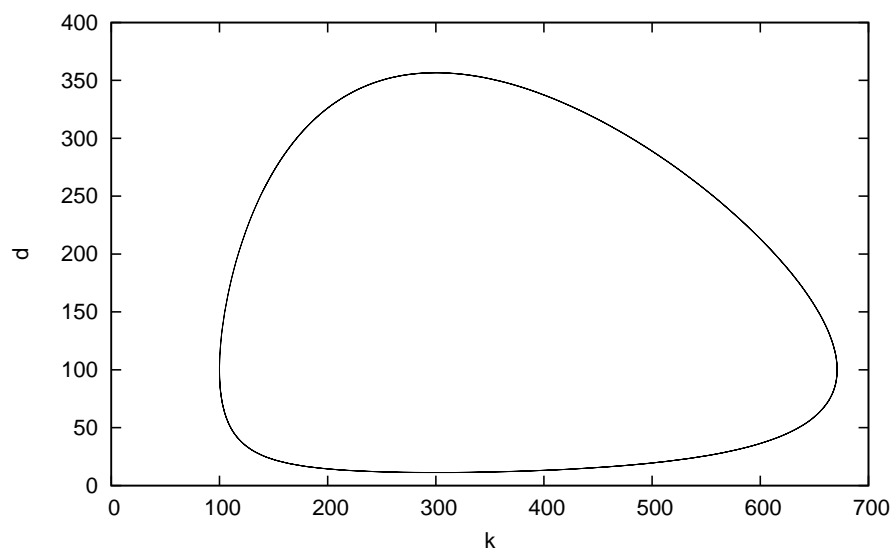
Výsledky pro vhodně zvolené koeficienty a počáteční stav můžeme zobrazit jako graf závislosti stavových proměnných (počtu dravců a kořisti) na čase – viz obrázek

x+y



Obrázek 6.11: Časový průběh řešení

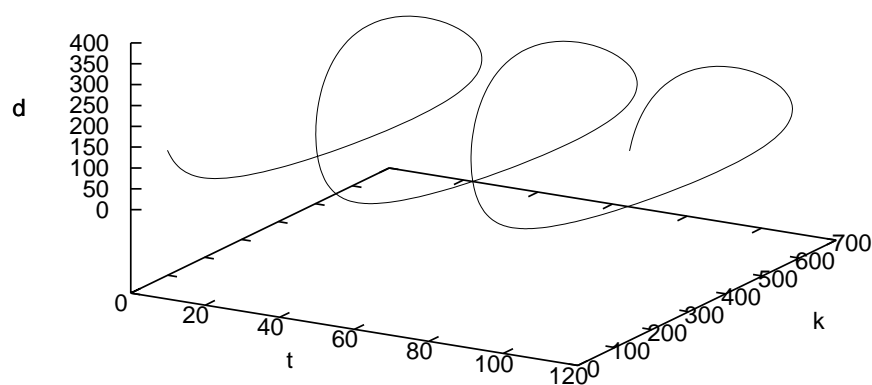
6.11. Také můžeme použít zobrazení ve fázové rovině, kdy na osách jsou hodnoty stavových proměnných systému – viz obrázek 6.12. Čas při tomto zobrazení není ex-



Obrázek 6.12: Zobrazení ve fázové rovině

plicitně uveden, ale každý bod na křivce udává stav systému v určitém čase – celá křivka zobrazuje historii stavů systému. Zobrazení ve fázové rovině může lépe vystihovat chování systému než časové průběhy. To platí zvláště u chaotických systémů. Pokud zobrazíme obě stavové proměnné v závislosti na čase ve 3D zobrazení, získáme obrázek 6.13.

⁶Podle tvůrců jsou modely dravec–kořist někdy nazývány Lotka-Voltera modely.



Obrázek 6.13: Časový průběh řešení zobrazený ve 3D

Pro porovnání jsou na obrázku 6.14 uvedeny skutečné počty rysů (*lynx*) a zajíců (*hare*) získané v rozmezí let 1900–1920 v severní části Kanady.



Obrázek 6.14: Reálný příklad vývoje počtu dravců a kořisti

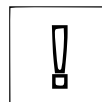
6.8 Spojité simulační jazyky

V této části se zaměříme na přehled základních vlastností a požadavků na nástroje pro popis modelu a experimentů. Ukážeme si také příklady praktického použití některých simulačních systémů.

6.8.1 Algoritmus řízení spojitě simulace

Spojité simulační systém pracuje podle následujícího základního algoritmu:

1. Inicializace — nastavuje počáteční stav stavových proměnných a simulátoru
2. Cyklus dokud není konec simulace:
 - (a) Pokud je vhodný čas — výstup sledovaných hodnot
 - (b) Krok numerické integrace:
 - Vyhodnocení derivací (vstupů integrátorů). Některé integrační metody vyhodnocují derivace několikrát v průběhu jednoho kroku.
 - Výpočet nového stavu integrační metodou.
 - Posun modelového času, který některé integrační metody provádí několikrát v průběhu jednoho kroku.
3. Výstup sledovaných hodnot po skončení simulace



6.8.2 Pořadí vyhodnocování funkčních bloků

Při vyhodnocování derivací dochází k postupnému výpočtu jednotlivých vstupů integrátorů. V rámci těchto výpočtů vyhodnocujeme také výrazy, u kterých záleží na pořadí jejich vyhodnocování.

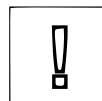
Například při vyhodnocování následujících funkcí a ukládání mezivýsledků do proměnných dojde k problému:

```
a = fa(1,b)    # b ještě není vypočítáno
b = fb(a)
c = fc(a,b)
...
```

Řešením je zajistit vyhodnocování v potřebném pořadí použitím vhodné metody:

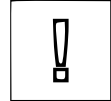
- *řazení funkčních bloků* je metoda používaná v překladačích simulačních jazyků – příkazy jsou přeuspořádány tak, aby při jejich postupném vyhodnocování bylo vyhodnoceno vše potřebné ještě před použitím.
- *vyhodnocování na žádost* (viz SIMLIB) je metoda vhodná pro použití v objektově orientovaných knihovnách pro simulaci. Vyhodnocovaný objekt rozešle zprávy s požadavky na vyhodnocení všem vstupům, ty se vyhodnotí (stejným způsobem mohou požadovat vyhodnocení vlastních vstupů) a vrátí požadované hodnoty, které objekt zpracuje a pošle na výstup.

Poznámka: Paměťové bloky (integrátory) mají oddělený vstup a jejich výstup se mění až po dokončení kroku proto jsou nezávislé na pořadí vyhodnocování.

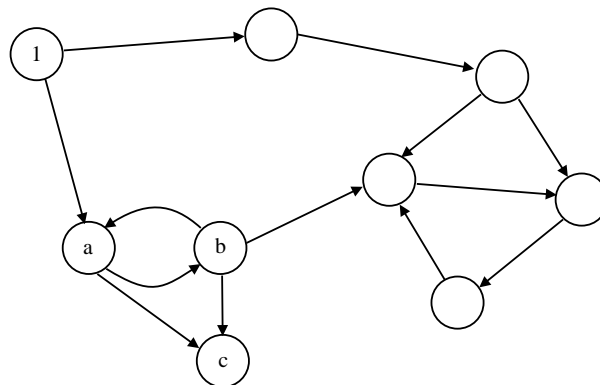


6.8.3 Algoritmus pro seřazení funkčních bloků

Nyní si ukážeme princip základního algoritmu (hledání silných komponent grafu), který je použitelný pro seřazení funkčních bloků tak, aby všechny vstupní hodnoty byly spočítány ještě před vyhodnocením bloku.



1. Vybudujeme graf závislostí bloků. Příklad jak může takový graf vypadat je na obrázku 6.15.



Obrázek 6.15: Příklad grafu závislostí funkčních bloků

2. Připravíme si prázdný seznam S pro ukládání výsledného pořadí a provedeme algoritmus řazení bloků:

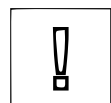
```
WHILE(graf je neprázdný):
  FOR(všechny uzly):
    IF(do uzlu nevstupují hrany tj. jsou známy všechny jeho vstupy)
      - operaci uzlu zařadíme na konec seznamu  $S$ 
      - zrušíme uzel (a všechny hrany z něj vystupující) z grafu
  // tím jsme z grafu odstranili vše co už lze spočítat,
  // zbývají jen uzly, které ještě nešlo vyhodnotit,
  // protože neměly vypočteny některé vstupy
  IF(nic nebylo odstraněno)
    // graf ještě obsahuje uzly, ale nelze je zpracovat
    KONEC2 - je přítomna tzv. rychlá smyčka
  KONEC - seznam  $S$  obsahuje všechny bloky v potřebném pořadí
```

3. Tento algoritmus pouze detekuje rychlé smyčky a skončí. Pokud je v grafu přítomna rychlá smyčka (naš příklad obsahuje dvě) musíme ji vyřešit a po jejím odstranění z grafu pokračovat v řazení stejným algoritmem.

Poznámka: Uvedený algoritmus má kvadratickou časovou složitost⁷

6.8.4 Rychlé smyčky

Rychlé smyčky (anglicky: "algebraic loops") tvoří cyklus v orientovaném grafu závislostí funkčních bloků. Například pro vyhodnocení bloku A potřebujeme hodnoty jiných bloků a tyto jiné bloky mohou zpětně požadovat hodnotu A . Přítomnost rychlých



⁷Pro zájemce o algoritmy: Existuje lepší Tarjanův algoritmus pro tento způsob řazení (anglicky: topological sorting) s lineární složitostí.

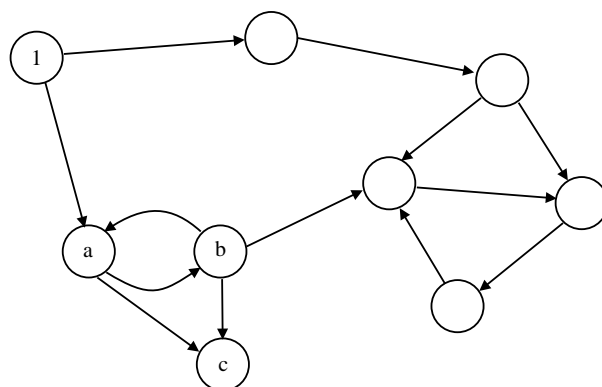
smyček je někdy způsobena příliš vysokou úrovní abstrakce, například zanedbáním některých integrátorů.

Při řešení rychlých smyček můžeme postupovat dvěma způsoby:

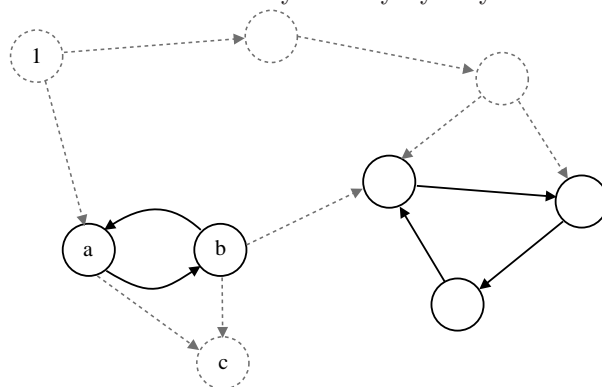
- Rozpojení cyklu speciálním blokem, který (například iteračně) řeší algebraické rovnice. To zpomaluje simulaci, protože v každém kroku vyžaduje řešení těchto rovnic⁸.
- Přeprogramování modelu tak, aby neobsahoval smyčky (například vložení integrátoru). Toto je možné, protože smyčky často vzniknou přílišným zjednodušováním popisu systému.

Příklad rychlé smyčky

Graf na obrázku 6.16 vyjadřuje závislosti bloků na jiných hodnotách bloků. Algoritmus řazení funkčních bloků v grafu nalezne dvě rychlé smyčky – viz obrázek 6.17. Možné řešení smyčky vložением speciálního bloku mezi bloky a–b naznačuje obrázek 6.18.

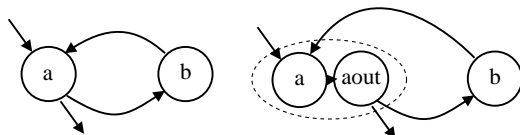


Obrázek 6.16: Příklad rychlé smyčky – výchozí stav



Obrázek 6.17: Rychlé smyčky po odstranění ostatních uzlů

⁸Pro řešení algebraických rovnic lze použít například metodu půlení intervalu (bisection), metodu Regula-falsi, případně další.



Obrázek 6.18: Řešení rychlé smyčky mezi bloky a, b

6.9 Parciální diferenciální rovnice

Tyto diferenciální rovnice obsahují derivace podle více proměnných – obvykle podle prostorových souřadnic a podle času. Používají se pro popis prostorových systémů, například pro popis šíření vln v tělesech. Oblast parciálních diferenciálních rovnic je velmi rozsáhlá (většina náročných simulací spadá do této oblasti), a přesahuje rámec bakalářského studia, proto je zde pouze pro informaci.

Numerické řešení těchto rovnic vyžaduje diskretizaci v prostorových souřadnicích – z parciálních derivací vzniknou diference. Princip řešení naznačuje následující příklad.

Příklad: Kmitající struna

Struna je popsána parciální diferenciální rovnicí definující dynamiku systému:

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

Počáteční podmínky definují počáteční polohu struny:

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

a rychlost na koncích

$$y'(x, 0) = 0$$

Okrajové podmínky definují upevnění obou konců struny:

$$y(0, t) = y(l, t) = 0$$

Pro řešení této rovnice musíme použít vhodnou metodu řešení parciálních diferenciálních rovnic. Zde aplikujeme metodu přímek, která spočívá v tom, že strunu rozdělíme na dostatečně velký počet úseků o velikosti Δx a všechny prostorové derivace nahradíme v každém bodě struny x_i diferencemi

$$\left. \frac{\partial^2 y}{\partial x^2} \right|_{x_i} = \frac{y_{i+1} - 2y_i + y_{i-1}}{\Delta x^2}$$

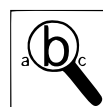
Touto diskretizací dostaneme soustavu obyčejných diferenciálních rovnic, protože nám zůstanou pouze derivace podle času. Pro každý úsek struny na pozici x_i máme rovnici popisující výchylku y_i :

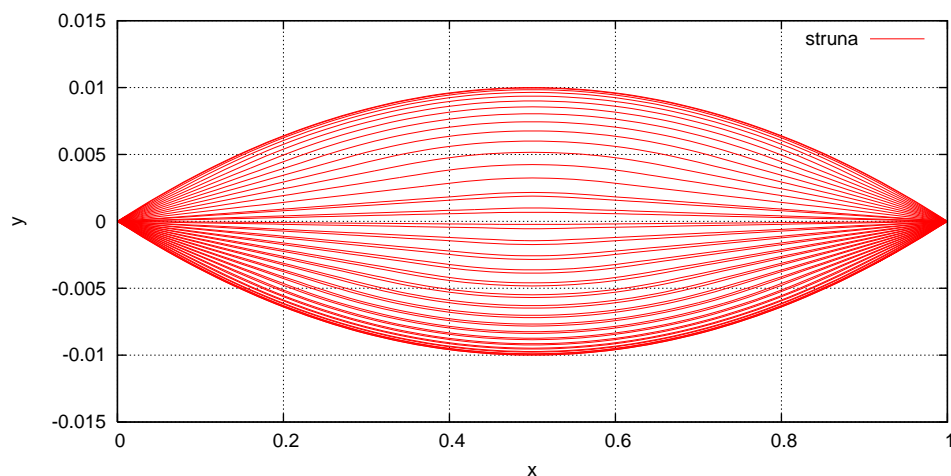
$$\frac{d^2 y_i}{dt^2} = a \frac{y_{i+1} - 2y_i + y_{i-1}}{\Delta x^2}$$

kterou již umíme řešit.

Na obrázku 6.19 je pro ilustraci uveden výsledek simulace touto metodou pro strunu rozdělenou na 100 úseků⁹.

⁹Zdrojový text tohoto příkladu najdete na WWW stránce s příklady SIMLIB/C++ [6].





Obrázek 6.19: Řešení modelu kmitání struny

6.10 Použití SIMLIB/C++ pro spojitou simulaci

SIMLIB/C++ je knihovna pro spojitou i diskrétní simulaci. V této podkapitole si ukážeme její nástroje pro spojitou simulaci a jejich použití pro popis spojitých modelů.

6.10.1 Blokové výrazy

Modely v SIMLIB/C++ využívají speciální postup pro blokový popis modelu:

- Automatická konstrukce výrazových stromů využívá možnost přetěžovat operátory v C++. Vhodně definované operátory pro bloky zajistí, že se při jejich volání v inicializaci modelu dynamicky vytvoří speciální objekty a propojí je podle struktury výrazů do stromů.
- Při simulaci jsou bloky vyhodnocovány voláním speciální metody `Value()`, která vrací výslednou hodnotu typu `double`. Tato metoda nejdříve vyhodnotí všechny vstupy bloku voláním jejich metody `Value`, potom se získanými hodnotami provede operaci danou typem bloku a nakonec vrátí výsledek této operace. Tím se automaticky řeší problém uspořádání funkčních bloků – viz podkapitola 6.8.2.

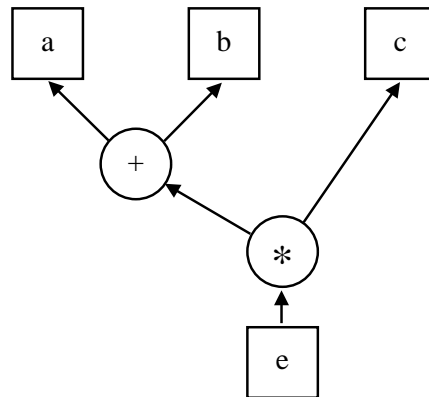
Příklad: Blokový výraz

```
Expression e = (a + b) * c
```

Inicializace objektu `e` automaticky vytvoří stromovou strukturu – viz obrázek 6.20. Proměnná `e` potom obsahuje odkaz na tuto dynamicky vytvořenou strukturu. Volání `e.Value()` vrátí výsledek výrazu pro aktuální hodnoty bloků `a`, `b`, `c`.

Poznámka: Při používání hodnoty modelového času v blokových výrazech nelze použít proměnnou `Time` jako v diskrétní simulaci, protože přetěžování operátorů v C++ nerozliší proměnnou od konstanty nebo literálu. Místo toho musíme použít speciální blok `T`, který zapouzdřuje referenci na `Time` a každé volání jeho metody `Value` vrací aktuální hodnotu modelového času.





Obrázek 6.20: Příklad blokové struktury výrazu

6.10.2 Typy bloků v SIMLIB/C++

Pro definici základních bloků jsou předdefinovány následující třídy a bloky:

- Blok reprezentující modelový čas **T** je speciální bloková varianta proměnné **Time**, kterou v blokových výrazech nelze použít (vyhodnotí se jako konstanta).
- Odkazy na blokové výrazy: odkaz na blok **Input** používáme jen pro parametry funkcí, které mají pracovat s bloky, speciální blok **Expression** je použitelný jako kterýkoli jiný blok.
- Třídy **Constant**, **Parameter**, **Variable** definují bloky použitelné jako konstanty (nelze je měnit), parametry (lze měnit jen mezi simulačními běhy), a proměnné (lze měnit kdykoli).
- Funkční bloky pro obecnou funkci **Function** a pro běžné funkce **Sin**, **Exp**, **Max**, **Sqrt**, **Abs**, atd.
Nelinearity typu omezení **Lim**, tření **Frict**, ...
Speciální interní bloky používané pro blokové výrazy reprezentují základní aritmetické operace: **_Add**, **_Mul**, ...
- Stavové bloky: **Integrator** zapouzdřuje numerickou integrační metodu. Další bloky popisují nelinearity s vnitřním stavem: hysterezní křivku **Hysteresis**, relé¹⁰ **Relay**, vůle v převodech **Blash** a podobně.

Výše uvedený přehled není úplný, podrobnější informace naleznete v dokumentaci [6] a ve zdrojových textech.

6.10.3 Popis experimentu

Sledování stavu modelu:

- třída **Sampler** – periodické volání funkce
- funkce **SetOutput(filename)** – přesměrování výstupu do zadaného souboru

¹⁰Implementace této třídy přesně detekuje okamžik přepnutí relé.

- funkce `Print(fmt,...)` – tisk typu `printf` na výstup

Nastavení parametrů simulace:

- Nastavení kroku funkcí `SetStep(minstep,maxstep)` nastaví povolené limity pro automatickou změnu integračního kroku. Globální proměnné `StepSize`, `MinStep`, `MaxStep` (dostupné pouze pro čtení) obsahují informace o aktuální hodnotě a nastaveném rozsahu kroku.
- Nastavení požadované přesnosti dosáhneme funkcí `SetAccuracy(abs,rel)`. Integrační metoda porovnává odhad chyby a v případě, že nevyhovuje uvedeným hodnotám, zkracuje automaticky krok.
- Nastavení integrační metody zajišťuje funkce `SetMethod(name)`. Hodnota řetězce se jménem metody může v aktuální verzi SIMLIB nabývat hodnot: "abm4", "euler", "fw", "rke" (tato metoda je implicitní), "rkf3", "rkf5" a "rkf8".

Řízení simulace je stejné jako u diskrétní simulace:

- `Init(t0,t1), Run()`
- `Stop()` – ukončení jednoho experimentu
- `Abort()` – ukončení programu

6.10.4 Příklad více experimentů v SIMLIB/C++

Uvedený příklad ukazuje, jak lze provést sérii experimentů s různými hodnotami parametrů systému. Model popisuje kmitání kola zavěšeného na pružině s tlumičem. Komentáře popisují jednotlivé sekce.

Popis systému provedeme diferenciální rovnicí druhého řádu:

$$y'' = (F - D * y' - k * y) / M$$

kde

- D je koeficient tlumení,
- k je tuhost pružiny a
- M je hmotnost kola.

Uvažujeme nulové počáteční podmínky a vstup ve formě skokové změny síly F .

```
// model kmitání kola (verze 2 - několik experimentů)
```

```
#include "simlib.h"
```

```
struct Kolo {                                // popis systému
    Parameter M, D, k;
    Integrator v, y;
    Kolo(Input F, double _M, double _D, double _k):
        M(_M), D(_D), k(_k),                // parametry systému
        v( (F - D*v - k*y) / M ),            // rychlost
        y( v ) {}                             // výchylka
    void PrintParameters() {                  // tisk poznámky
```

$x + y$


```

        Print("# hmotnost = %g kg ", M.Value());
        Print("  tlumeni = %g ",      D.Value());
        Print("  tuhost = %g \n",      k.Value());
    }
};

```

Popis systému kola je proveden blokovými výrazy na vstupech integrátorů v a y. Tyto výrazy získáme z výchozí rovnice metodou snižování řádu derivace.

```

// define objektů modelu ...
double _m=5, _d=500, _k=5e4;    // implicitní hodnoty parametrů
Constant F(100);                // síla působící na kolo k
Kolo k(F, _m, _d, _k);          // instance modelu kola

```

Zde vytvoříme a inicializujeme model kola k. Zvolené hodnoty parametrů nejsou reálné, byly nastaveny tak, aby výsledky pěkně vypadaly.

```

// sledování stavu modelu ...
void Sample() {
    // sledujeme čas, výchylku, rychlost
    Print("%f %g %g\n", T.Value(), k.y.Value(), k.v.Value());
}
Sampler S(Sample, 0.001); // vzorkování se zadanou periodou

```

Pro průběžné sledování stavu použijeme speciální periodickou událost, která volá funkci `Sample`, ve které zaznamenáváme stav. Třída `Sampler` byla vytvořena především pro zjednodušení popisu vzorkování stavu spojitých modelů s ohledem na to, aby se uživatelé pracující pouze se spojitými modely nemuseli učit popisovat diskrétní události.

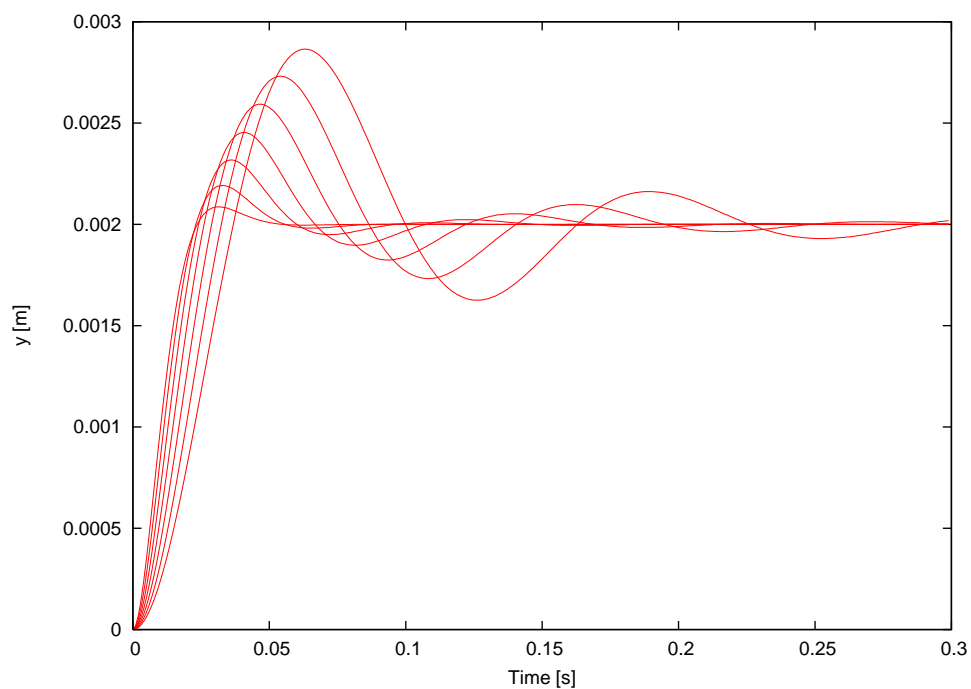
```

// popis více experimentů ...
int main() {
    SetOutput("kolo2.dat");
    _Print("# KOL02 - model tlumení kola (více experimentů)\n");
    for(double m=_m/2; m<=_m*5; m*=1.4) {
        k.M = m;    // nastaví parametr M
        k.D = _d;   // nastaví parametr D
        k.k = _k;   // nastaví parametr k
        k.PrintParameters();
        Print("# Time  y  v \n");
        Init(0, 0.3);    // inicializace experimentu
        SetAccuracy(1e-6, 0.001); // max. povolená chyba integrace
        Run();           // simulace
        Print("\n");     // oddělí výstupy
    }
}

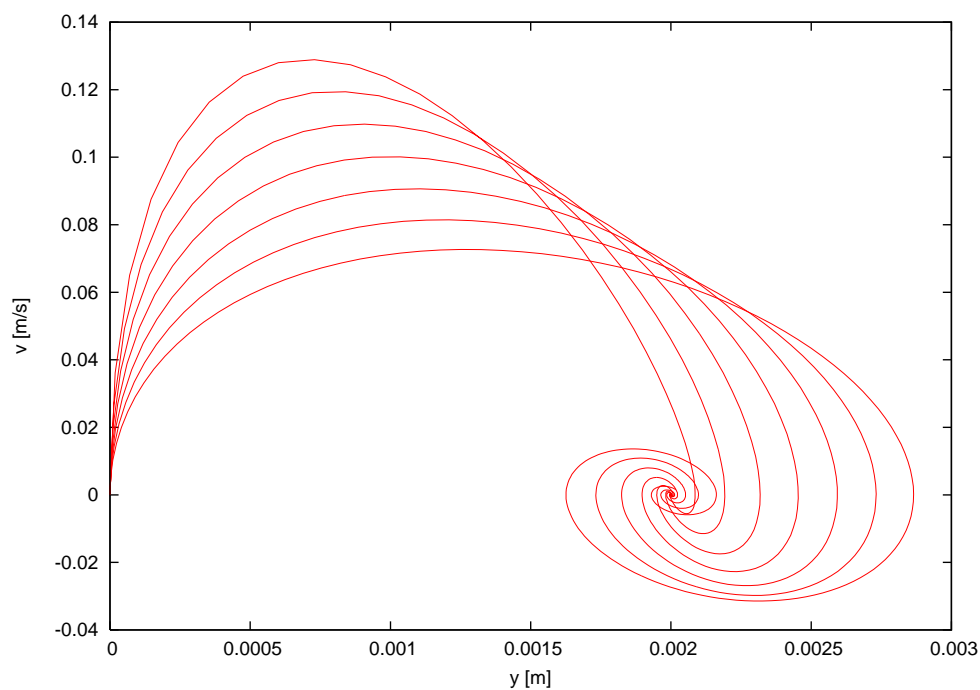
```

Řízení simulace popsané ve funkci `main` provádí více experimentů s různými hodnotami parametrů modelu (zde měníme hmotnost kola). Textový výstup je ve formátu vhodném ke zobrazení programem Gnuplot[17].

Výsledné časové průběhy pro 7 experimentů s různou hmotností kola jsou uvedeny na obrázku 6.21. Zobrazení ve fázové rovině na obrázku 6.22 dává do vztahu rychlost a výchylku kola.



Obrázek 6.21: Výsledky



Obrázek 6.22: Výsledky zobrazené ve fázové rovině

6.11 Shrnutí

Tato kapitola čtenáře seznámila se základním přehledem v oblasti spojité simulace. Byly prezentovány oblasti použití spojité simulace, způsoby popisu spojitého modelu,

nejpoužívanější numerické metody a jejich vlastnosti, spojitý simulační nástroj a základní principy jejich implementace. Tato oblast simulací má velmi dlouhou tradici a je k dispozici rozsáhlá literatura k dalšímu studiu. Pro zájemce je k dispozici také řada zajímavých příkladů.

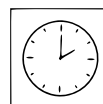
6.12 Otázky a úkoly

1. Vyjmenujte základní typy bloků pro popis spojitých systémů.
2. Vysvětlete princip Eulerovy integrační metody.
3. Napište rovnici volného pádu hmotného bodu v homogenním gravitačním poli, převedte ji na soustavu rovnic prvního řádu metodou snižování řádu derivace a naprogramujte s využitím Eulerovy metody. Výsledky simulace zobrazte například programem GnuPlot.
4. Mějme rovnici: $2y'' + 5y' - y = 0$. Převedte metodou snižování řádu derivace, naprogramujte a simulujte po dobu 20 časových jednotek. Zvolte vhodný krok. Výsledek zobrazte například programem GnuPlot.
5. Upravte kód kruhového testu s Eulerovou metodou: Zaměňte kód pro Eulerovu metodu za Runge-Kuta druhého řádu. Porovnejte výsledky při stejné délce kroku.
6. Upravte kód kruhového testu s Eulerovou metodou: Zaměňte kód pro Eulerovu metodu za některou vícekrokovou metodu. Porovnejte výsledky při stejné délce kroku.
7. Na WWW naleznete vhodný netriviální spojitý příklad s výsledky simulace, naprogramujte ho v SIMLIB/C++ a porovnejte výsledky.



Kapitola 7

Kombinovaná simulace



1:00

Kombinovaná (hybridní) simulace spojuje spojitou a diskrétní simulaci do jednoho celku. Spojením spojitých a diskrétních modelů vznikne kombinovaný model, ve kterém je třeba řešit následující problémy:

- **Problém kombinace událostí a numerické integrace.** Je nutné zajistit, aby událost měla možnost pracovat s aktuálním stavem spojitě části modelu. Toho lze dosáhnout tzv. "dokročením" numerické integrační metody přesně na čas události. V případě plánovaných událostí je dokročení jednoduché — délka posledního kroku numerické integrace před provedením události se zkrátí tak, aby událost nastala na konci kroku.

Dalším problémem je možnost skokových změn spojitěho stavu modelu. Například je nutné restartovat víceukrokové integrační metody, pokud událost změní stav integrátoru.

- **Stavové podmínky a detekce jejich změn.** V případě, že je třeba provádět události podmíněné spojitým stavem kombinovaného modelu (např. sepnutí relé při dosažení určitého napětí), musíme mít v modelu speciální ("stavové") podmínky, které je nutné při spojitě simulaci vyhodnocovat a v případě jejich změny provádět speciální ("stavové") události.

Pozor — pouze *změna* stavové podmínky způsobí stavovou událost.

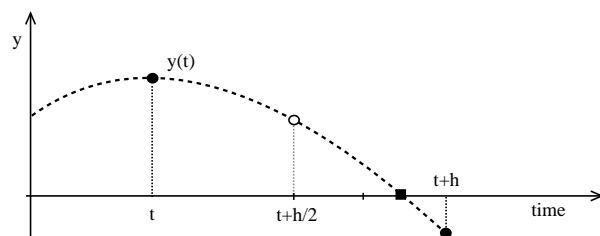
Protože by provádění stavových událostí až po dokončení integračního kroku běžné délky způsobovalo příliš velké chyby výpočtu, musí kombinovaný simulátor provádět "dokročení" na čas stavové události, který (na rozdíl od plánovaných událostí) neznáme předem. Nalezení přesného času provedení stavové události odpovídá problému numerického řešení algebraických rovnic typu $f(t) = 0$, kde t je hledaný čas.

7.1 Stavové podmínky a stavové události

Stavová podmínka (State Condition) je booleovský výraz obsahující relační operace nad zadanými hodnotami spojitých veličin v modelu, například $y(t) > x(t)$. *Stavová událost (State Event)* je událost, která nastane pouze při *změně* hodnoty stavové podmínky — a protože se spojitě hodnoty, na kterých stavová podmínka závisí průběžně počítají, nelze takové události naplánovat do kalendáře. Pro nalezení přesného času kdy má dojít ke stavové události lze použít iterační metody, např. půlení intervalu (*bisection*), Regula-falsi nebo Newtonovu metodu. Volba vhodné metody závisí především na řešeném modelu a formulaci stavových podmínek.

Příklad: Detekce dopadu míčku na zem

Hledáme řešení algebraické rovnice $y(t) = 0$, kde $y(t)$ je výška míčku nad podložkou — viz Obr 7.1.

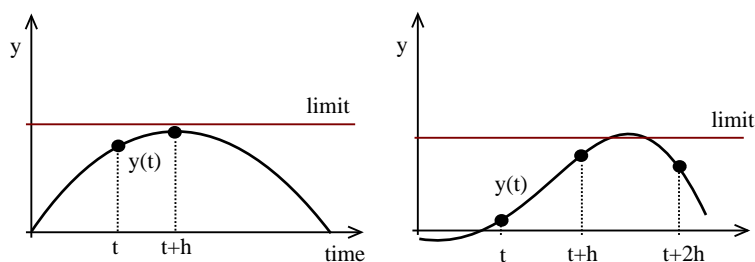


Obrázek 7.1: Hledání času dopadu míčku

7.1.1 Problémy detekce změn stavových podmínek

Protože numerické řešení není zcela přesné a navíc je získáváme jen v určitých časových okamžicích, můžou nastat problémy — viz Obr 7.1.1. Nedetekování změny stavové podmínky může být způsobené například:

- numerickou chybou (řešení nedosáhne zadané hodnoty),
- příliš dlouhým krokem (překročení dvou nebo i více změn).



Obrázek 7.2: Chyby detekce změn stavových podmínek

Tyto problémy je nutné vyřešit například vhodnou změnou stavových podmínek v modelu.

Pokud je v modelu více stavových podmínek a provedení jejich stavových událostí ovlivňuje stav jiných podmínek, mohou tyto závislosti způsobit problémy (např. při nevhodném pořadí zpracování stavových podmínek a událostí). Proto je nutná opatrnost při psaní takových modelů.

7.1.2 Stavové podmínky v SIMLIB/C++

SIMLIB definuje speciální bázev abstraktní třídy jako základ pro bloky typu detektor změn stavových podmínek:

- `Condition` — detekce jakékoli změny,
- `ConditionUp` — změna `false` → `true`,
- `ConditionDown` — změna `true` → `false`

Podmínka je vždy ve tvaru (`vstup >= 0`). Odvozené třídy musí definovat metodu `void Action()` s popisem stavové události.

SIMLIB používá metodu půlení intervalu při které zkracuje krok až k hodnotě `MinStep`. Výhodou této metody je zaručená konvergence.

7.2 Algoritmus řízení kombinované simulace

Pseudokód algoritmu řízení kombinované simulace spojuje algoritmus řízení diskrétní simulace typu "next-event" s algoritmem řízení spojitě simulace a navíc doplňuje detekci změn stavových podmínek.

```

Inicializace modelového času
Inicializace kalendáře událostí
Inicializace modelu
Inicializace stavových podmínek
while ( není konec simulace) {
    while ( čas < čas první položky v kalendáři) {
        Uložení stavu a času          ***
        Krok numerické integrace a posun času
        Vyhodnocení stavových podmínek
        if ( podmínka změněna )
            if ( krok <= minimální_krok)
                Potvrzení nového stavu podmínek
                Provedení stavové události ===
                krok = běžná_velikost_kroku
            else
                Obnova stavu a času          ***
                krok = krok/2
                if (krok < minimální_krok)
                    krok = minimální_krok
    }
    čas = čas první položky v kalendáři    @@@
    Odebrání první položky z kalendáře
    Provedení plánované události
}

```

Stavová událost může plánovat jinou událost (a tím změnit čas první položky v kalendáři).

Krok numerické integrace nesmí překročit čas první plánované události. Délka posledního kroku před tímto časem proto musí být vhodně upravena (`koncový_čas - Time`) – jde o tzv. "dokročení".¹

7.3 Příklad modelu: skákající míček

Typickým příkladem kombinovaného modelu je model skákajícího míčku, ve kterém je odraz od podložky popsán formou stavové události. V SIMLIB/C++ je možné použít třídu `ConditionDown`, která detekuje změnu podmínky $y(t) \geq 0$ z `true` na `false`. Následující kód popisuje model volného pádu $y'' = -g$, stavovou podmínku pro dopad $y \geq 0$ a stavovou událost modelující odraz míčku se ztrátou rychlosti.

```

struct Micek : ConditionDown { // skákající míček
    Integrator v,y;             // stavové proměnné
    void Action() {              // popis stavové události
        v = -0.8 * v.Value();    // ztráta energie
        y = 0;                   // eliminace nepřesnosti
    }
}

```

¹Pokud by přitom nastal problém s minimální délkou kroku, je možné ho řešit prodloužením předchozího kroku.

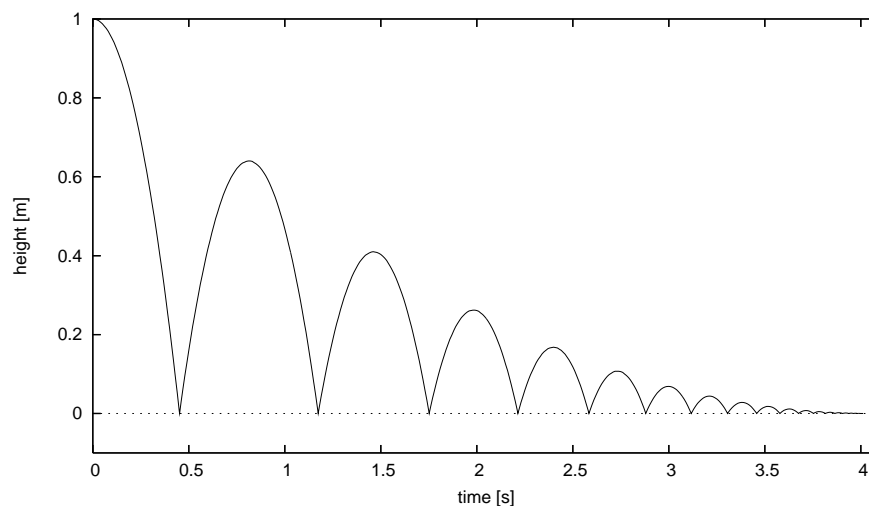
```

Micek(double initialposition) :
    ConditionDown(y), // (y>=0) detekce změny true-->false
    v(-g),             // y' = INTG( -g )
    y(v, initialposition) // y = INTG( y' )
    {}
};

Micek m1(1.0);          // instance modelu

```

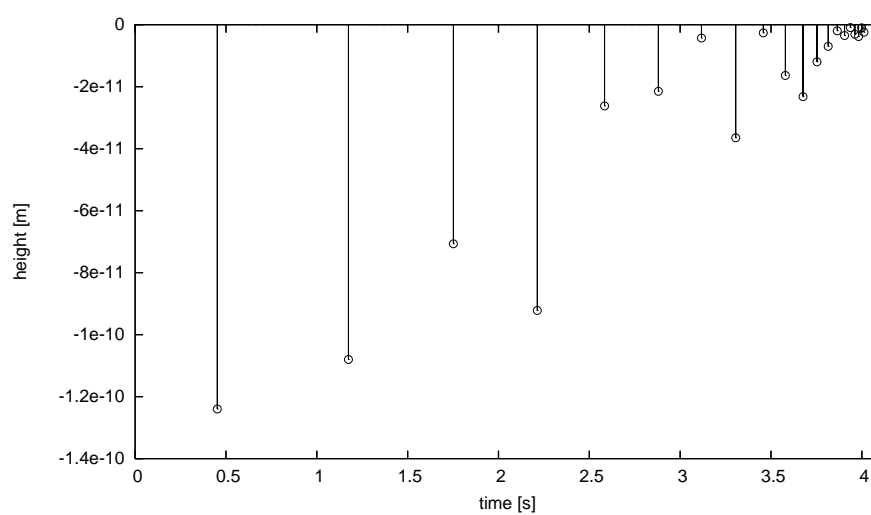
Úplný příklad (včetně popisu experimentu) je součástí zdrojových textů knihovny SIMLIB. Na obrázku 7.3 je časový průběh výšky míčku. SIMLIB používá metodu půlení kroku a pro zadaný minimální krok 10^{-9} je chyba detekce dopadu zanedbatelná — viz obr. 7.4.



Obrázek 7.3: Skákající míček — výsledek

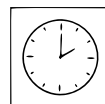
7.4 Otázky a úkoly

1. Vysvětlíte kdy a proč je nutné restartovat víceukrokové metody numerické integrace při kombinované simulaci.
2. Na jaké problémy můžeme narazit při zjišťování přesného času stavových událostí?
3. Proč se stavové události provádí pouze při změnách stavových podmínek a ne vždy když podmínka platí?
4. Upravte kód algoritmu řízení kombinované simulace tak, aby zahrnoval kód pro dokročení na plánovanou událost.

Obrázek 7.4: Chyba detekce dopadu pro minimální krok 10^{-9}

Kapitola 8

Celulární automaty



0:30

Pojem *Celulární automat* (CA, anglicky *Cellular Automaton*) označuje diskretní model prostorového systému založený na struktuře základních komponent — buněk. Stav CA je možné počítat v jednotlivých krocích (používáme diskretní časovou množinu) aplikací tzv. pravidel, která počítají následující stav ze stavu buňky a jejího okolí.

Existují různé varianty CA lišící se definicí okolí, pravidel atd. Použití CA sahá od jednoduchých her (Life) přes simulace dopravy, šíření epidemií, chemických reakcí, růstu krystalů, generování textur a fraktálních útvarů až po modely umělého života.

Tato kapitola obsahuje pouze velmi stručný úvod do problematiky CA.

8.1 Definice CA

CA je diskretní systém složený z následujících komponent:

- Pole buněk (*Lattice*): obecně n -rozměrné, obvykle 1D nebo 2D, může být konečné nebo nekonečné, všechny buňky jsou stejné (rovnoměrné dělení prostoru).
- Konečná množina S stavů buňky: například $S = \{0, 1\}$.
- Okolí N (*Neighbourhood*) definuje počet a pozici sousedních buněk se kterými daná buňka pracuje (používá je jako vstup pro pravidla).
- Pravidla (*Rules*) popisují chování buňky — jde o funkci stavu buňky a jejího okolí definující nový stav buňky v čase:

$$s(t+1) = f(s(t), N_s(t))$$

kde $s \in S$ a N_s je stav všech buněk v okolí.

Typické vlastnosti CA

- Konfigurace CA je definována jako stav všech buněk a popisuje stav systému.
- Stav CA se vyvíjí v čase a prostoru podle zadaných pravidel
- Čas i prostor jsou diskretizovány
- Počet stavů buňky je konečný (ale existují varianty CA, u kterých může být stav popsán reálným číslem).
- Buňky jsou identické, ale někdy mohou existovat speciální buňky, které mají např. neměnný stav (modelují pevné hranice).
- Následující stav buňky závisí pouze na aktuálním stavu.

Implementace CA

Základní implementace CA je jednoduchá, pokud nevyžadujeme speciální vlastnosti simulátoru, jako například akceleraci výpočtu¹.

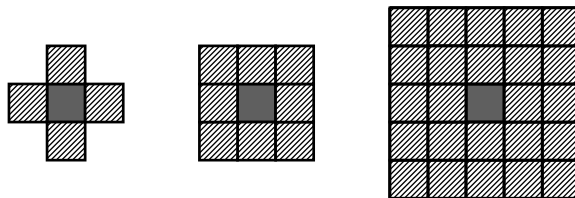
Vyhodnocování pravidel musí probíhat tak, aby výsledky vypočítané pro nový čas $t + 1$ neovlivnily vstup pravidel pro ještě nevyhodnocené buňky. Typicky lze použít dva stavy (aktuální a nový) a po výpočtu všech nových stavů se toto označení zamění (z nového stavu se stane aktuální).

Při implementaci "nekonečného" pole buněk je možné využívat podobné principy jako u řídkých matic — ukládají se např. pouze nenulové prvky do dynamických datových struktur. Existuje řada volně dostupných implementací CA ze kterých lze čerpat informace — viz WWW.

8.2 Typy okolí

Volba okolí závisí na rozměru prostoru a tvaru buněk, například pro 2D a čtvercové buňky můžeme (mimo jiné) použít okolí typu:

- Von-Neumann
- Moore
- Extended Moore



Obrázek 8.1: 2D okolí typu Von-Neumann, Moore a Extended Moore

Existuje mnoho dalších typů okolí — viz WWW.

8.3 Pravidla CA

Pravidla musí popisovat změnu stavu pro všechny možnosti vstupu. Jde o funkce stavu buňky $s(t)$ a jejího okolí $N_s(t)$:

$$s(t + 1) = f(s(t), N_s(t))$$

Tyto funkce mohou mít různé vlastnosti, podle kterých rozlišujeme různé typy pravidel, např.:

”legal”: u tohoto typu pravidel z nulového vstupu nesmí vzniknout nenulový.

”totalistic”: rozhoduje *součet* hodnot ze vstupních buněk (jako např. u hry Life)

¹Jako příklad netriviální implementace doporučuji ”HashLife”.

Celkový počet všech možných pravidel závisí na počtu stavů buňky a velikosti okolí. Například pro jednorozměrné okolí 1D, kdy máme na vstupu 3 buňky (jednu aktuální a dvě sousední) se stavy $\{0, 1\}$ (tzv. elementární CA) existuje celkem $2^3 = 8$ možností na vstupu (každou hodnotu lze vyjádřit jako jedno 3-bitové číslo) a tedy celkem $2^8 = 256$ všech možných funkcí (pravidel).

000	-->	X0
001	-->	X1
010	-->	X2
011	-->	X3
100	-->	X4
101	-->	X5
110	-->	X6
111	-->	X7

Každá taková funkce je jednoznačně definována konkrétními hodnotami 8 bitů v pořadí X7 X6 . . . X1 X0. Pokud vyjádříme hodnotu těchto bitů číselně, můžeme všechna pravidla jednoznačně očíslovat (hodnoty mají rozsah 0 – 255) a použít toto označení k jejich identifikaci (např. "pravidlo 30").

8.3.1 Reverzibilní automaty

Reverzibilní automat je systém, který neztrácí informaci při svém vývoji v čase. Proto je v každém okamžiku možno otočit běh času nazpátek a vracet se k předchozím stavům. Například pokud bychom definovali nový stav buňky takto:

$$s(t+1) = f(s(t), N_s(t)) - s(t-1)$$

je možné pro libovolné f vypočítat předchozí stav:

$$s(t-1) = f(s(t), N_s(t)) - s(t+1)$$

8.4 Klasifikace CA

Celulární automaty můžeme rozdělit² podle jejich dynamického chování do 4 základních kategorií:

třída 1: Automat po konečném počtu kroků dosáhne jednoho konkrétního ustáleného stavu.

třída 2: Chování automatu skončí periodicky se opakující posloupností stavů (s krátkou periodou) nebo automat nakonec zůstane stabilně v některém ze stavů.

třída 3: Automat vykazuje chaotické chování (deterministický chaos).

třída 4: Kombinace běžného a chaotického chování (sem patří např. Life), CA z této kategorie nejsou reverzibilní.

8.5 Příklad CA

Hra "Life" je CA, který nastavíme na počáteční stav a spustíme. Automat pro hru Life je definován takto:

²Zdroj: kniha Wolfram S.: *New Kind of Science* <http://www.wolframscience.com/>

- Buňka může být ve stavu '0' (mrtvá) nebo '1' (živá).
- Pole buněk je dvourozměrné (2D), buňky mají čtvercový tvar.
- Okolí je typu Moore složené z 8 okolních buněk.
- Pravidla: výsledný stav buňky závisí na počtu "živých" buněk v okolí:
 - buňka '1' zůstane ve stavu '1', když má 2 nebo 3 sousedy '1'
 - buňka '0' se změní na '1', když má právě 3 sousedy '1'
 - jinak bude nový stav buňky '0'

I takto jednoduchý CA vykazuje velmi zajímavé chování – viz příklady na WWW.

8.6 Otázky a úkoly

1. Definujte pojem CA.
2. Jaké různé typy okolí CA znáte?
3. Napište pseudokód algoritmu řízení simulace modelu popsaného CA.
4. Implementujte jednoduchý elementární CA (např pravidlo 30), proveďte 50 kroků s vhodně zvoleným počátečním stavem a výsledný časový průběh vhodným způsobem zobrazte.
5. Na jaké problémy můžeme narazit při implementaci neomezeně velikého pole buněk a jaké jsou možnosti jejich řešení?

Kapitola 9

Závěr

Tato příručka nezahrnuje celou řadu zajímavých partií modelování a simulace. Zájemci o podrobnější informace a příklady najdou použitelné odkazy v seznamu literatury. Obecně není vhodné studovat pouze z jednoho zdroje, proto doporučuji prostudovat alespoň některé volně dostupné zdroje na Internetu. Poměrně rozsáhlý seznam vybraných odkazů najdete na WWW [18] a také na stránkách předmětu IMS.

Tento text neprošel žádnou jazykovou úpravou, proto je možné, že ještě obsahuje chyby. Uvítám veškeré konstruktivní připomínky a náměty na doplnění textu. Pište na e-mail: peringer@fit.vutbr.cz

Seznam změn v textu

2006-09: originální verze

2008-11: drobné úpravy algoritmu řízení diskretní simulace

2009-01: drobné úpravy interpunkce

2012-12: doplnění kapitol: Kombinovaná simulace, Celulární automaty

2012-12-17: oprava $X_9 \ X_8 \rightarrow X_7 \ X_6$

Literatura

- [1] Rábová Z. a kol: *Modelování a simulace*, skriptum VUT, 1992
- [2] Fishwick P. A.: *Simulation Model Design and Execution – Building Digital Worlds*, Prentice Hall, 1995
- [3] Ross S.: *Simulation*, 3rd Edition, Academic Press, 2002
- [4] Law A., Kelton D.: *Simulation Modelling and Analysis*, McGraw-Hill, 1991
- [5] Zeigler B., Praehofer H., Kim T.: *Theory of Modelling and Simulation*, 2nd edition, Academic Press, 2000
- [6] SIMLIB/C++ home page
<http://www.fee.vutbr.cz/~peringer/SIMLIB/> (prosinec 2012)
- [7] Wikipedia: http://en.wikipedia.org/wiki/Linear_congruential_generator
- [8] Mersenne Twister Home Page <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html>
(září 2006)
- [9] Brown R.: *DieHarder: A Random Number Test Suite*, 2006,
<http://www.phy.duke.edu/~rgb/General/dieharder.php>
- [10] The R Project for Statistical Computing, <http://www.r-project.org/>
- [11] MathWorks home page, <http://www.mathworks.com/>
- [12] Scilab – A Free Scientific Software Package, <http://www.scilab.org/>
- [13] GNU Octave, <http://www.gnu.org/software/octave/>
- [14] Modelica and the Modelica Association, <http://www.modelica.org/> (září 2006)
- [15] The OpenModelica Project, <http://www.ida.liu.se/~pelab/modelica/OpenModelica.html>
(září 2006)
- [16] NETLIB home page, <http://www.netlib.org/>
- [17] Gnuplot home page, <http://www.gnuplot.info/>
- [18] Seznam odkazů modelování a simulace,
<http://www.fit.vutbr.cz/~peringer/UNOFFICIAL/simulation/>
- [19] TOP500 Supercomputer Sites, <http://www.top500.org/>
- [20] SimPack Toolkit, <http://www.cise.ufl.edu/~fishwick/simpack.html> (září 2006)
- [21] Wolfram S.: *New Kind of Science*, Wolfram Media, 2002