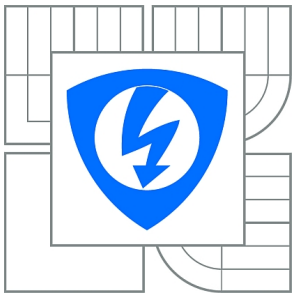
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

DEPARTMENT OF CONTROL AND INSTRUMENTATION

KNIHOVNA PRO MATEMATICKÉ VÝPOČTY V JAZYCE C++

ENHANCED MATHEMATICAL LIBRARY FOR C++

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

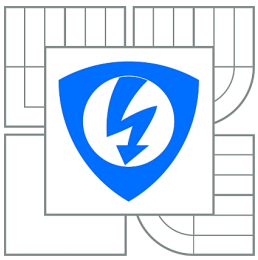
AUTOR PRÁCE — ALEŠ TEMEL

AUTHOR

VEDOUCÍ PRÁCE — Ing. PETR PETYOVSKÝ

SUPERVISOR

BRNO 2012



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky a komunikačních technologií

Ústav automatizace a měřicí techniky

Bakalářská práce

bakalářský studijní obor
Automatizační a měřicí technika

Student:	Aleš Temel	***ID:***	125670
Ročník:	3	***Akademický rok:***	2011/2012

NÁZEV TÉMATU:

Knihovna pro matematické výpočty v jazyce C++

POKYNY PRO VYPRACOVÁNÍ:

1. Seznamte se s principy uložení a zpracování tzv. "řídkých" matic v paměti počítače a algoritmy pro výpočty s těmito maticemi.
2. Navrhněte a implementujte knihovnu pro maticové výpočty s ohledem na úsporné uložení maticových dat v případě "řídkých" matic.
3. Seznamte se s principy uložení a zpracování čísel s plovoucí desetinnou čárkou v paměti počítače.
4. Navrhněte a implementujte základy knihovny pro přesný datový typ určený pro uložení čísel s plovoucí desetinnou čárkou.
5. Integrujte obě knihovny do společného rámce (framework), který umožní maticové výpočty s přesnými desetinnými čísly.
6. Demonstrujte funkčnost knihovny na vhodných příkladech.
7. Zhodnoťte dosažené výsledky (paměťovou náročnost, rychlost výpočtu, způsob implementace).
8. Srovnejte dosažené výsledky s existujícími řešeními pro ukládání řídkých matic řešení a navrhněte další možná rozšíření.

DOPORUČENÁ LITERATURA:

[1] Prata, S.: Mistrovství v C++, Computer press 2004, ISBN 80-251-0098-7.
[2] Virius, M.: Jazyky C a C++, Grada 2006, ISBN 80-247-1494-9.
[3] Peluch, T.: C++ matrix library, bakalářská práce, ÚAMT VUT 2007.

Termín zadání:	6.2.2012	***Termín odevzdání:***	28.5.2012

Vedoucí práce: Ing. Petr Petyovský
Konzultanti bakalářské práce:

doc. Ing. Václav Jirsík, CSc.
Předseda oborové rady

Abstrakt

Ve své bakalářské práci vytvářím knihovnu pro uchování matic a práci s maticemi. V tomto případě je problematika zaměřena hlavně na tzv. řídké matice. Jazyk C++ nenabízí mezi standardními knihovnami nástroje pro jednoduchou práci s řídkými maticemi. Nejčastější alternativou bývá použití dvourozměrného pole, tzv. 2D pole. 2D pole může být realizováno jako dvojitý ukazatel reprezentující řádky a sloupce matice. Základním problémem je fakt, že 2D pole se chová stejně k řídkému, tak i plnému nenulovému poli. Nezohledňuje se možnost uložit pole výhodněji. Mnou navržená knihovna pro uchování řídkých matic tento problém zohledňuje více různými způsoby. Nabízí nejen úsporný formát CSR (Compressed sparse rows), ale i alternativy pro uložení speciálně strukturovaných matic. Při tvorbě knihovny jsem kladl důraz především na velikost paměti, která bude potřeba na uložení objektu. Protože se jedná o matematickou knihovnu, tak obsahuje různé funkce vhodné pro práci s maticemi, jako výpočet determinantu, výpočet inverzní matice a podobně. Při výpočtech těchto funkcí se zohledňuje také velikost využité paměti, proto jsou i matice pro výpočty subdeterminantů ukládány do řídkých formátů a mezivýsledky jsou rovněž odstraňovány co nejdříve po jejich využití. Byla vytvořena také druhá knihovna, která se zabývá čísly uloženými s větší přesností než jsou standardní datové typy. Velikost potřebné paměti se zvyšuje se zvětšující se přesností čísla s pohyblivou řádovou čárkou.

Klíčová slova

Matice, šablona, knihovna, matematické funkce, pohyblivá řádová desetinná čárka, paměť, C++.

Abstract

I create a library for storing matrices and working with matrices in this bachelor's thesis. In this case, the issues concern mainly so-called sparse matrices. C++ do not provide among the standard libraries tools for easy working with sparse matrices. The most frequent alternative is the application of two-dimensional array (2D array). 2D array may be realized as a double pointer representing the rows and the columns of the matrix. The basic problem is that behavior of 2D array is identical both to the sparse, and to the full non-zero matrix. 2D array ignores the possibility to store matrices more favorable. The library for storing matrices designed by me takes into account the storing of sparse matrices by several different ways. It offers the sparse format CSR (Compressed sparse rows), as well as alternatives to save the specially structured matrices. In the course of creating library the main emphasis first of all I put on the amount of memory that is necessary for storing matrices. Because it is a mathematical library, it contains different functions suitable for working with matrices, such as determinant calculation, inverse matrix calculation and so. When calculate these functions it is necessary to take into account used memory size as well. Intermediate results are stored also in sparse format, and removed as soon as possible after their use. I created also the second library that deals with the floating point numbers stored with greater precision than standard data types like *float* and *double*. The size of occupied memory increases according to precision of floating point number.

Keywords

Matrix, template, library, mathematical functions, floating point, memory, C++.

TEMEL, A. Knihovna pro matematické výpočty v C++. Brno: Vysoké učení technické v Brně, Fakulta elektroniky a komunikačních technologií, 2012. 93s. Vedoucí Bakalářské práce Ing. Petr Petyovský

Prohlášení

„Prohlašuji, že svou bakalářskou práci na téma Knihovna pro matematické výpočty v jazyce C++ jsem vypracoval samostatně pod vedením vedoucího semestrální práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené bakalářské práce dále prohlašuji, že v souvislosti s vytvořením této bakalářské práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních, a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

V Brně dne: **28. května 2012** ……

podpis autora

Poděkování

Děkuji vedoucímu semestrální práce Ing. Petru Petyovskému za účinnou metodickou, pedagogickou a odbornou pomoc a další cenné rady při zpracování mé semestrální práce.

V Brně dne: **28. května 2012** ……

podpis autora

Obsah

Seznam tabulek

Seznam obrázků

Seznam zkratek

2D	dvourozměrné
B	byte
CSR	Compressed sparse rows – komprimované řídké řádky
CSC	Compressed sparse columbs – komprimované řídké sloupce

Seznam pojmů

Setter	Metoda umožňující nastavení hodnot objektu nebo třídy.
Getter	Metoda umožňující získání dat z objektu nebo třídy.
MantiseType	Typ datové proměnné pro uložení mantisy čísla.
Konverze	Změna interního formátu.

1 ÚVOD

Při řešení praktických úloh, např. při simulacích dynamických systémů, pracujeme pomocí maticové algebry. Jednotlivé matice často z technické podstaty úloh obsahují velké množství nulových prvků. Takovéto matice se nazývají řídké matice. Specifikaci pojmu „řídká matice“ bych definoval asi takto: Matice je řídká právě tehdy, když vzhledem k počtu obsažených nul je výhodné uložit ji v některém z formátů pro uložení řídkých matic. Základním problémem při práci s řídkými maticemi je nehospodárný způsob využití paměti, kterým je ukládání matic do 2D (dvourozměrného) pole. Při práci s maticí uloženou ve 2D poli se nerozlišuje, jestli se jedná o řídkou matici nebo matici naplněnou převážně nenulovými hodnotami. To je úlohou zadané knihovny. V běžném případě se ukládají všechny prvky, vč. nulových, a zbytečně je tak zabírána paměť. Proto řeším problém uchování řídkých matic ve výhodnějším způsobu uložení.

Problematika tvorby knihoven bude řešena v jazyce C++, což je objektově orientovaný programovací jazyk, který je vyvinutý na základě rozšíření jazyka C. C++ nabízí možnost tzv. generického programování, což znamená, že algoritmy programu jsou nezávislé na datových typech proměnných. Výhodou C++, na rozdíl od klasického C, je existence tříd, které budou využívány. Základní myšlenkou při práci bude fakt, že knihovna bude řešena jako třída, která v sobě bude obsahovat veškeré potřebné údaje pro existenci řídkých matic a veškeré metody a operátory umožňující práci a matematické operace s maticemi. Knihovna pro práci s řídkými maticemi bude použitelná pro různé datové typy, které budou obsahem matice. Jednotlivé algoritmy metod budou rozděleny do více souborů, podle jejich vlastností, pro větší přehlednost v kódu. Knihovna bude psána jako šablona, tedy nezávislá na datovém typu zapouzdřeného prvku, takovým způsobem, aby bylo možné zápis kódu chápat jako obecný. Šablona bude mít jeden parametr, který bude určovat, jakými datovými typy jsou prvky v matici. Datový typ parametru šablony by měl umět základní operace jako +, -, /, ==, aby bylo možné provozovat všechny operace v rámci řídkých matic. Uložení řídké matice bude nabízet 4 základní formáty, do kterých se matice může uložit. Aktivní formát bude vždy pouze jeden.

Cílem práce je vytvořit knihovnu pro uchování řídké matice tak, aby uživatel mohl provádět s datovým typem řídké matice operace jako s klasickým dvourozměrným polem a přitom měl neustále matici uloženou v co nejvýhodnějším formátu šetřícím jeho paměťový prostor. Dalším cílem je co nejlépe zvládnout přepočetní algoritmy a konverze interních formátů řídkých matic. Jedná se hlavně o problém změny formátů, při kterém je nutné vymazat starý formát a vytvořit formát nový. Není možné nejprve smazat starý formát, protože by se ztratila data. V případě, že bych nejprve vytvořil formát nový, měl bych v jednom okamžiku zbytečně vytvořené dva formáty. Bude tedy nutné mazat a vytvářet formáty postupně. Dále pak vytvořit spektrum operátorů a metod, které bude uživatel moci používat při práci s maticemi a jejich matematickými operacemi. Knihovna není psána pro konkrétního uživatele, metody tedy nebudou specifické. Zvolené metody umožňující matematické operace budou obecně vhodné pro matematické počty s maticemi. Podle konkrétních požadavků se pak mohou dopsat metody další, jako například výpisy jednotlivých sloupců, řádků apod.

Další vlastností programovacích jazyků včetně C++ je způsob, jakým je implementována číselná hodnota v paměti. V případě proměnných platí, že se ukládají do různých datových typů jako *int, long int, double* atd… Každý z těchto datových typů je omezen svým vymezeným datovým prostorem a tím pádem i počtem cifer nebo maximální velikostí čísla. Proto vytvářím knihovnu, která předpokládá i uložení čísla na velké množství (řádově 10^9) platných cifer. V tomto případě bude přesnost čísla omezena pamětí, do které jsem schopen ukládat data, a proměnnými, do kterých budou ukládány pomocné proměnné pro správnou funkčnost algoritmů. Velikost zabrané paměti čísly bude vymezena podle potřebné přesnosti čísla. To znamená, že pro čísla malých přesností nebude formát vymezovat zbytečně paměť navíc. Cílem této knihovny je realizace třídy pro uchování čísla na velký počet platných cifer při dosažení obdobných vlastností jako mají datové typy *float* a *double*. Dalším cílem je zohlednit paměťovou náročnost knihovny, i když se bude jednat o číslo s malým počtem platných cifer a dále dosáhnout co nejlepší přesnosti a minimální paměťové náročnosti s ohledem na časovou náročnost.

Obě knihovny budou vzájemně kompatibilní. To znamená, že po přidání obou knihoven k projektu bude možné provádět maticové výpočty, kde obsahem matic budou přesná čísla.

Správnost dosažených výsledků bude porovnána s nástroji zabývajícími se obdobnou problematikou. V případě řídkých matic budou výsledné matice porovnány se stejným zadáním v MATLABu. Výsledné operace přesných desetinných čísel budou porovnávány se stejným zadáním v MS EXCEL.

Výsledky budou porovnány s existujícími řešeními pro ukládání řídkých matic. Budou navržena další rozšíření, která nebudou naimplementována a budou užitečná pro efektivnější použití knihoven.

2 ÚVOD DO ŘÍDKÝCH MATIC

2.1 Řídké matice

Řídké matice (sparse matrices) jsou speciálním případem klasických dvourozměrných matic, jejichž struktura je stejná jako u klasických plných dvourozměrných matic, obsahují však velké množství nulových prvků. Jinak je možné s řídkými maticemi provádět klasické maticové výpočetní operace. Matici lze rovněž uložit do paměti počítače jako hustou matici, tedy matici, která má každou jednotlivou hodnotu uloženu v paměti, a to včetně nul. Toto řešení je však nevýhodné z hlediska nehospodárného uložení matice, kdy jsou ukládány všechny prvky včetně nulových.

To, že matice obsahuje spoustu nulových prvků, nám sice při práci s maticí nijak nevadí, ale je důležité se zamyslet nad tím, jak je vlastně tato matice v paměti uložena. Jde o to, že přesáhne-li počet nulových prvků určitou úroveň, je výhodnější uložit do paměti pouze prvky nenulové, bohužel však na úkor toho, že každý uložený prvek potřebuje většinou ke svému dohledání více paměťového prostoru. Obvykle se jedná o další údaj pro index řádku nebo sloupce. Existují i speciální případy, kdy není potřeba těchto indexů, a to například u diagonální matice.

Existuje více metod, kterými lze uložit řídkou matici do paměti, přičemž každý formát je výhodnější někdy jindy, v závislosti na strukturalizaci nul v matici.

2.2 Uložení řídké matice

Primárním úkolem při práci s řídkými maticemi je jejich úsporné uložení do paměti počítače bez nepotřebných nul. Nulový prvek složitého datového typu zabírá v paměti stejně velké místo jako prvek nenulový. Proto existují řídké formáty, které se ukládání nulových prvků vyvarují. V řídkém formátu však musí být všechny údaje pro pozdější možnost dekódování matice a provádění různých matematických operací .

Obecně platí, že čím složitější datový typ je obsahem matice, tím mají nuly v matici větší význam. Potom je výhodnější použít některý z úsporných datových formátů pro uložení řídkých matic.

Příklad:

V případě klasické „husté“ matice typu double, kde každé číslo obvykle zabere 8B (8 bytů) v paměti, je třeba si uvědomit, že 8 bytů paměti zabere i uložená nula, a to zcela zbytečně. Tímto způsobem je např. $8*n*m$ bytů, kde n je počet řádků a m počet sloupců. V případě, že je těchto nul v matici více, vyplatí se uložit ji úsporněji, tedy bez nul.

2.3 Příklad řešení řídkých matic

Jednou z nejrozsáhlejších maticových úloh byl výpočet dominantního vlastního vektoru matice řádu $n = 2.7\ x\ 10^9$ (jednalo se o tzv. Google matrix - Googlovskou matici), kde počet řádků a sloupců matice odpovídal počtu webových stránek indexovaných vyhledávačem Google. Pro uložení jednoho řádku této matice v hustém formátu by bylo zapotřebí cca 20,1 GB, pro uložení celé matice pak cca 39290171 TB. Při představě, že s takto obsáhlou maticí provádíme nejen její ukládání, ale i různé operace, se nabízí myšlenka, zda by se matice nedala uložit jinak. Googlovská matice má většinu prvků nulových (prvek $a_{i,j}$ je nenulový, pokud webová stránka s indexem i obsahuje hypertextový odkaz na webovou stránku s indexem j). Pro uložení (nejen) takto rozsáhlých matic se tedy používají tzv. **řídké formáty**. Převzato z [4].

Řídká matice může reprezentovat například i tabulku číselných proměnných. Pokud si chci uchovat pouze obsah tabulky ve smyslu úsporného uchování dat, a spousta prvků tabulky je nulových nebo nevyplněných, potom lze tabulku uložit do matice v řídkém formátu.

3 FORMÁTY PRO ULOŽENÍ ŘÍDKÝCH MATIC

3.1 Souřadnicový formát

Souřadnicový formát je pravděpodobně nejjednodušší z hlediska implementace. U tohoto formátu ukládáme pro každou nenulovou hodnotu tři údaje, a to**: index řádku,** na kterém se číslo nachází, **index sloupce,** na kterém se číslo nachází, a samotnou **hodnotu,** kde index řádku a sloupce mohou být například hodnoty typu *unsigned int* (obvykle čtyřbajtová proměnná). Pokud bych však chtěl pracovat s většími rozměry pole, potom je třeba indexy ukládat do v proměnných typu *unsigned long int* nebo *unsigned long long int*. Do těchto proměnných se mohou ukládat indexy v případě, že rozměry matice přesáhnou 4B, což nám dovolí maximálně 4394967296 řádků a sloupců. V případě, že budeme opět zvažovat datový typ *double*, uvažujme 8B, znamená to, že na uložení je třeba $\boldsymbol{8 * k + 4 * 2 * k}$ bytů, kde *k* je počet ukládaných nenulových prvků. Převzato z [3].

Ze zápisu je vidět, že nyní na uložení jednoho čísla nepotřebujeme 8B, jako v případě „husté“ matice, ale 12B. Toto sice není žádoucí, ale počet zapsaných prvků klesne, protože nulové hodnoty v matici **nejsou** zapisovány, čímž vzniká paměťová úspora. Platí pak podmínka z rovnice (1). Za předpokladu jejího splnění je tento formát výhodné upřednostnit. Tento formát je nejvýhodnější v případech, kdy je v matici enormní počet nul, v krajním případě, je-li v každém řádku pouze jedna hodnota.

$$\frac{(\boldsymbol{8+4}) * \boldsymbol{k}}{\boldsymbol{2a}} < 8 * m * n \qquad (1)$$

Uspořádané trojice (*index řádku, index sloupce* a *hodnota*) mohou být v paměti seřazeny různým způsobem (po řádcích, po sloupcích nebo náhodně). Náhodné seřazení je sice rychlejší, ale přístup k datům je pracnější, protože je nutné sekvenci prohledat.

3.2 CSR formát

Dalším z formátů pro uložení řídké matice je tzv. **CSR** formát **(compressed sparse rows)**, v překladu: komprimované řídké řádky. Matice se (obdobně jako u souřadnicového formátu) uloží do tří polí. První pole v sobě obsahuje všechny nenulové hodnoty matice srovnané po řádcích. U tohoto formátu je nutné, aby byly prvky uspořádané po řádcích, kvůli kódování třetího pole. Druhé pole obsahuje sloupcové indexy jednotlivých hodnot. Třetí pole pak říká, při kterém prvku začíná nový řádek. Tento formát by měl být funkční, i když na řádku nebude žádná hodnota.

Uvažujme následující příklad [4]:

$$A = \begin{bmatrix} 10 & & 3 & \\ & 8 & & 4 \\ 2 & 2 & 6 & \\ & & & 5 \end{bmatrix},$$

Obrázek 1 Matice určená k uložení pomocí CSR formátu

Jednotlivá pole CSR formátu pak jsou:

`Data = [10,3,8,4,2,2,6,5]`	hodnoty nenulových prvků
`Sloupce = [1,3,2,4,1,2,3,4]`	pole sloupcových indexů
`Řádky = [1,3,5,8]`	pole čísel, která značí číslo začínající na novém řádku

Pokud ukládáme *k* prvků, pak je k uložení matice v CSR formátu potřeba $2 * k + n$ hodnot, kde *n* je počet řádků.

Tato tři pole v tomto případě budou realizovány formou vektorů, do kterých se budou ukládat jednotlivé hodnoty a indexy.

K uložení matice typu double je pak potřeba $8 * k + 2 * k + 2 * n$, kde ukládáme *k* prvků a *n* je počet řádků.

Tento formát bude použit v knihovně. Na uložení jednotlivých polí budou použity *vektory* z knihovny „`vector.h`".

3.3 CSC formát

Dalším z formátů pro uložení řídké matice je tzv. **CSR formát (compressed sparse columbs)**, v překladu: komprimované řídké sloupce. Matice se analogicky jako u CSR formátu uloží do tří polí. První pole obsahuje všechny nenulové hodnoty matice srovnané po sloupcích. U tohoto formátu je nutné, aby byly prvky uspořádané po sloupcích, kvůli kódování třetího pole. Druhé pole obsahuje sloupcové indexy jednotlivých hodnot. Třetí pole pak říká, při kterém prvku začíná nový řádek. Tento formát by měl být funkční, i když na řádku nebude žádná hodnota.

Uvažujme následující příklad [4]:

$$A = \begin{bmatrix} 10 & & 3 & \\ & 8 & & 4 \\ 2 & 2 & 6 & \\ & & & 5 \end{bmatrix},$$

Obrázek 2 Matice určená k uložení pomocí CSC formátu.

Jednotlivá pole CSC formátu pak jsou

`Data = [10,2,8,2,3,6,4,5]` hodnoty nenulových prvků
`Řádky = [1,3,2,4,1,2,3,4]` pole řádkových indexů
`Sloupce = [1,3,5,7]` pole čísel, která značí číslo začínající v novém sloupci

Pokud ukládáme *k* prvků, pak je k uložení matice v CSR formátu potřeba $2k + n$ čísel, kde *n* je počet řádků.

3.4 Diagonální matice

Diagonální matice je naprosto unikátní typ matice, kde jsou hodnoty uloženy pouze v hlavní nebo vedlejší diagonále. Takto zapsanou matici lze uložit do jednorozměrného pole.

$$A = \begin{bmatrix} 10 & & & \\ & 8 & & \\ & & & \\ & & & 5 \end{bmatrix}$$

Obrázek 3 Matice určená k uložení pomocí diagonálního tvaru.

Matice pak bude uložena takto:

```
Hodnoty=[10,8,0,5];
HlavniDiagonala=true;
```

U tohoto případu je enormně výhodné uložení matice do tohoto formátu. Pro srovnání: uvažujme matici o rozměrech $10 * 10$ uchovávající datovou proměnnou double. Počet bytů potřebných pro uložení matice ve dvourozměrném poli je $8 * 10 * 10$, což je **800 bytů,** a v případě diagonální matice je potřeba 2B pro uchování rozměrů matice, 1 byte pro zjištění, zda se jedná o hlavní nebo vedlejší diagonálu, a $8 * 10$ bytů pro uchování hodnot. Pro uchování hodnot je třeba 80B.

3.5 Trojúhelníková matice

Je to řídká matice, kde hodnoty jsou pouze nad (*pod)* hlavní (*vedlejší*) diagonálou.

Předpokládám její uložení do jednorozměrného pole, přičemž musím znát orientaci trojúhelníkové matice a její rozměry. Matice bude ukládána jako hustá (včetně nul), ale pouze její polovina a diagonálu.

$$A = \begin{bmatrix} 10 & & 3 & \\ & 8 & & 4 \\ & & 6 & \\ & & & 5 \end{bmatrix}$$

Obrázek 4 Matice určená k uložení pomocí trojúhelníkového tvaru.

Matice pak bude uložena například takto:

```
HlavniDiagonala=true; NadDiagonalou=true;

Hodnoty=[10,0,3,0;8,0,4;6,0;5];
```

$$\frac{8 * n * n}{2} + 8 * n \qquad (2)$$

K uložení matice typu *double* potřebuji (2) bajtů, kde *n* je počet řádků a počet sloupců. U trojúhelníkové matice by měl být počet řádků a počet sloupců stejný. Formát musí obsahovat informace o orientaci, jestli se jedná o trojúhelník nad nebo pod hlavní nebo vedlejší diagonálou.

4 PROBLEMATIKY ŘEŠENÍ ŘÍDKÝCH MATIC

Problém řešení a ukládání řídkých matic je poměrně známá věc, zabývá se jím již více prací. Z dohledaných zdrojů jsem však zjistil, že každá práce se na tuto problematiku dívá trochu jinak.

Jednu z metod řešení řídkých matic používá systém MATLAB. Ten ukládá řídké matice do souřadnicového formátu a prvky ukládá po sloupcích [3].

Všechny mnou dohledané vytvořené návrhy prací s řídkými maticemi se ukládaly do předem známého řídkého formátu, a posléze se autor snažil usnadnit práci s nimi např. pomocí numerických metod [4]. Já jsem se rozhodl vyřešit tento problém jinou metodou, a to tak, že se formáty budou měnit v průběhu chodu programu. Moje práce neklade předně důraz na výpočetní rychlost, ale spíše na paměťovou náročnost. Paměťovou optimalizaci považuji za stěžejní a podmiňuji tomu i rychlost některých operací. Pokud by uživatel chtěl rychlejší operace bez paměťové úspory, může použít dvourozměrné pole.

4.1 Knihovny LAPACK, CLAPACK, CppLAPACK

Knihovna Lapack, v překladu - *Linear Algebra PACKage*, je softwarová knihovna pro řešení numerické lineární algebry. Knihovna byla původně napsána v prostředí FORTRAN 77 a později Fortran 90. Knihovna poskytuje nástroje pro řešení systémů lineárních rovnic, lineárních nejmenších čtverců, problémy vlastních čísel atd… Kromě toho se zabývá také různým uložením matic včetně řídkých matic. Převzato z [6]. Některé ze způsobů uložení matic pomocí LAPACKu jsou znázorněny v tabulce 1.

Tabulka 1 Seznam možných uložení matic pomocí LAPACKu. Převzato z [6]

Jméno	**Popis**
BD	Bidiagonal matrix
DI	Diagonal matrix
GB	Band matrix
GT	Tridiagonal Matrix General Matrix
HB	(complex) Hermitian matrix Band matrix
HE	(complex) Hermitian matrix
HG	upper Hessenberg matrix, generalized problem (i.e. a Hessenberg and a Triangular matrix)
HP	(complex) Hermitian matrix, Packed storage matrix
HS	upper Hessenberg matrix
OP	(real) Orthogonal matrix, Packed storage matrix
OR	(real) Orthogonal matrix

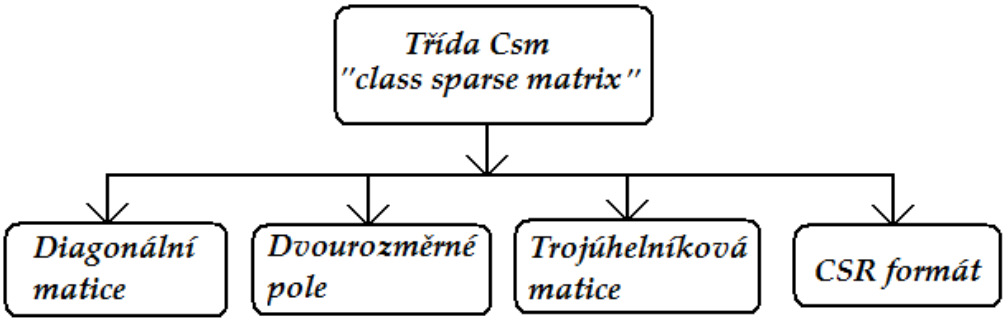
PB	Symmetric matrix or Hermitian matrix positive definite band
PO	Symmetric matrix or Hermitian matrix positive definite
PP	Symmetric matrix or Hermitian matrix positive definite, Packed storage matrix
PT	Symmetric matrix or Hermitian matrix positive definite Tridiagonal matrix
SB	(real) Symmetric matrix Band matrix
SP	Symmetric matrix, Packed storage matrix
ST	(real) Symmetric matrix Tridiagonal matrix
SY	Symmetric matrix
TB	Triangular matrix Band matrix
TG	triangular matrices, generalized problem (i.e., a pair of triangular matrices)
TP	Triangular matrix, Packed storage matrix
TR	Triangular matrix (or in some cases quasi-triangular)
TZ	Trapezoidal matrix
UN	(complex) Unitary matrix
UP	(complex) Unitary matrix, Packed storage matrix

Jak je vidět, knihovna LAPACK nabízí široké spektrum možností.
Při tvorbě knihovny jsem se inspiroval těmito formáty a zakomponoval jsem do knihovny formáty pro uložení matice jako diagonální a trojúhelníkové matice.

Kromě knihovny LAPACK existují různá rozšíření jako CLAPACK, což je modifikace programu plně použitelná v jazyce C. Obdobně je i CppLAPACK, který je plně použitelný v C++.

5 ŘEŠENÍ KNIHOVNY PRACUJÍCÍ S ŘÍDKÝMI MATICEMI

V této kapitole je popsána systematika knihovny pro uchování řídkých matic. Strukturalizace třídy, která uchovává řídkou matici v některém z výhodných formátů pro její uložení. Aplikační možnosti matice jsou obdobné jako u klasické matice uložené ve dvourozměrném poli. Uložení řídké matice by vždy mělo být ve formátu, který je pro ni nejvýhodnější.



Obrázek 5 Blokové schéma třídy pro práci s maticemi

Knihovna pro matematické výpočty s řídkými maticemi obsahuje jednu třídu, která se jmenuje `Csm "class sparse matrix"`. Tato třída je napsána jako šablona. To znamená, že algoritmy jsou psány obecně nezávisle na datových typech. To umožňuje napsat kód pouze jedenkrát a použít jej pro různé datové typy. Šablona má jeden parametr „`T`". Do něj se pak nakopíruje datový typ, který je obsahem matice.

Třída zahrnuje základní údaje o matici, jako jsou rozměry matice, počet nenulových prvků, výhodnosti jednotlivých formátů, informaci o aktivním formátu a také 4 části pro uchování matic samotných.

První z formátů ukládá pouze hlavní nebo vedlejší diagonálu matice. Pro paměťové uložení je vytvořen ukazatel na datové typy stejné jako je šablona, kam se uloží diagonála matice. Dále třída obsahuje informaci o tom, zda se jedná o hlavní nebo vedlejší diagonálu.

Druhý formát ukládá matici do dvourozměrného pole. Tento formát je obsažen v matici, protože se nemusí nutně jednat o řídkou matici. Potom je nejvhodnější uložit matici do dvojitého ukazatele na datový typ stejného s prvky v matici.

Třetím z formátů je uložení matice ve formě trojúhelníku. Trojúhelníková matice může být zapsána do třídy čtyřmi různými způsoby. Záleží na tom, ve které oblasti se vyskytují nenulové prvky. Mohou se vyskytovat pod nebo nad hlavní a vedlejší diagonálou. Formát je realizován informacemi o diagonálách a dvourozměrným polem. Toto dvourozměrné pole se ovšem nevytváří čtvercové, ale ve trojúhelníkovém tvaru v závislosti na orientaci trojúhelníkové matice.

Čtvrtým formátem je řídký formát CSR (*compressed sparse rows*). Tento formát je realizován formou tří vektorů z knihovny "`vector.h`". Do jednoho se ukládají nenulové prvky, do druhého jejich sloupcové indexy a do třetího vektoru se ukládají informace o řádkovém rozložení.

Všechny čtyři formáty jsou ukládány podle teoretických standardů popsaných v kapitole 3.

Dále jsou ve třídě implementovány různé metody pro získávání informací o výhodnostech a schopnosti uchování dat v jednotlivých formátech. Třída obsahuje čtyři konstruktory, čtyři metody pro nastavení dílčích parametrů matice, čtyři metody pro získání dílčích informací o matici a pět různých metod pro zjišťování stavu a schopnosti ukládat matici do různých formátů. Obsahem jsou i metody pro výpočty s maticemi, jako je výpočet inverzní matice, výpočet determinantu matice nebo výpočet transponované matice.

Knihovna obsahuje dva druhy možných setrů, což jsou metody nastavující jednotlivé proměnné matice, pomocí nichž lze matici naplnit hodnotami. Metoda `Set()`, která umožňuje matici naplnit datovými hodnotami je uživatelsky pohodlnější. Uživatel se nemusí starat o to, v jakém formátu je matice uložena, prostě matici plní prvky. Tato metoda má však tu nevýhodu, že konvertuje formáty, pokud je to třeba a kontroluje správnost zvoleného formátu. Tento proces kontrol a konverzí trvá relativně dlouho, a proto jsem napsal i druhou metodu `Setter()`, která se tímto nezabývá. Její správné použití je však komplikovanější. Metoda přijímá jeden parametr navíc, a to informace o zvoleném formátu. Na základě této informace je vytvářena matice. Pokud zvolíte špatný formát, do kterého na dané pozice nelze zapisovat, data nebudou zapsána.

Knihovna je kvůli další možné úpravě rozdělena do více souborů. Soubory se jmenují podle toho, jaké metody nebo operátory se v ní vyskytují. Pro správné použití knihovny musíte ke svému projektu připojit soubor "`sparse_matrix.h`" a všechny ostatní soubory musejí být ve složce projektu.

5.1 Seznam konstruktorů řídké matice

Matici je možné vytvořit implicitně, z dvourozměrného pole při známé velikosti pole, jako kopii jiné řídké matice, a parametricky v určitém formátu jako prázdné pole:

```
Csm();
Csm(T** matice, unsigned int rozmerx, unsigned int rozmery);
Csm(const Csm<T>& matice);
Csm(int format,unsigned int radku,unsigned int sloupcu);
~Csm();
```

5.2 Seznam metod a operátorů umožňující práci s maticemi

Metody pro získání hodnot, rozměrů, formátu matice:

```
T Get(unsigned int radek, unsigned int sloupec) const;
unsigned int GetRadky()const;
unsigned int GetSloupce()const;
int GetFormat()const;
```

Metody pro nastavení hodnot, rozměrů:

```
void SetRadky(unsigned int value);
void SetSloupce(unsigned int value);
void Setter(const T& hodnota,unsigned int radek,unsigned int sloupec,int pom)
void Set(const T& hodnota,unsigned int radek,unsigned int sloupec);
```

Matematické metody:

```
T determinant(int rad);
Csm<T> VytvorInverzni();
Csm<T> Transponuj();
```

Metody pro kontrolu a změnu formátů:

```
void prepocet();
int zjistiFormat(Csm<T>& vysledek,const Csm<T>& matice,int operace) const;
int zjistiFormat2(Csm<T>& vysledek,T hodnota,int operace)const;
int ZjistiFormat3();
bool konverze(int format);
```

operátory:

+	Plus, členské, nečlenské, s hodnotou, s maticí.
-	Mínus, členské, nečlenské, s hodnotou, s maticí.
*	Krát, členské, nečlenské, s hodnotou, s maticí.
/	Děleno, s maticí, hodnotou.
<<	Stream pro výpis matice.

Základní popis metod je napsán v dokumentaci vytvořené pomocí programu doxygen, viz Příloha 2. V následující kapitole jsem podrobně popsal všechny složitější metody.

6 POPISY METOD

V této kapitole se pokusím nastínit principy složitějších funkcí, vysvětlit jejich chování a zdůvodnit některá řešení.

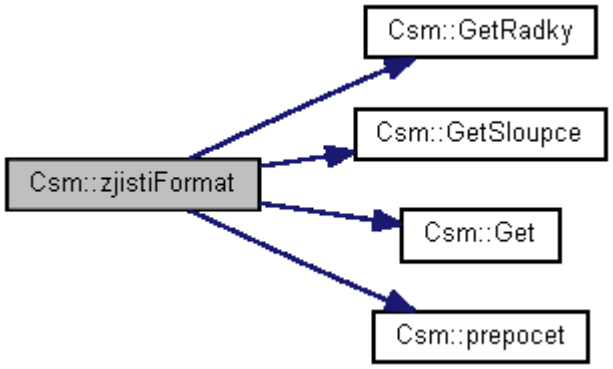
6.1 Metody pro zjišťování nejvýhodnějšího formátu

```
int zjistiFormat(Csm<T>& vysledek,const Csm<T>& matice,int operace) const;
int zjistiFormat2(Csm<T>& vysledek,T hodnota,int operace)const;
int ZjistiFormat3();
```

Knihovna obsahuje celkem tři metody, pomocí kterých je možno vyhodnotit formát, do kterého je nejvýhodnější uložit matici. Také však musí být možné do tohoto formátu matici uložit bez ztráty dat.

Př.: Pokud je nejvýhodnější uložit matici do diagonálního tvaru, avšak matice není diagonální, uloží se matice do druhého nejvýhodnějšího formátu, pokud je to možné.

Volané funkce z metod:



Obrázek 6 Seznam funkcí volaných z metod typu `zjistiFormat()`

Metody `GetRadky()`, `GetSloupce()` a `Get()` jsou používány pro získání dat ze vstupní matice.

Metody využívají funkci `prepocet()`, která zjistí, s jakou výhodností je možné uložit matici do toho či onoho formátu. Tato metoda však nebere ohled na to, zda-li je to možné učinit bez ztráty dat. Všechny tři metody typu `zjistiFormat()` pracují podobně, ale s rozdílnými parametry.

Nyní popíši jednotlivé metody zvlášť.

Metoda `zjistiFormat()` má tři parametry:

1. Matice výsledku
2. Zdrojová matice
3. Číslo požadované operace (`+,-,*,/`) viz Příloha 1.

Metoda vyhodnotí na základě požadované operace, do kterého formátu a jak je možné matici uložit. Informace o způsobu uložení se již uloží do matice výsledku, do které předpokládám, že se bude matice následně ukládat. To znamená, že do výsledku jsou již nastaveny informace o počtech prvků, informace o diagonálách a rozměry matice. Samotná data ukládána nejsou.

Metoda `ZjistiFormat2()` má také tři parametry:

1. Matice výsledku
2. Zdrojová hodnota
3. Číslo požadované operace (`+,-,*,/`) viz Příloha 1.

Rozdíl je v tom, že při vyhodnocování hodnot na jednotlivých pozicích se nejedná o operaci dvou matic, ale o operaci matice s jednou hodnotou. Všechny ostatní vlastnosti metody jsou shodné s metodou `ZjistiFormat()`.

Poslední metodou je metoda `ZjistiFormat3()`. Ta nemá žádný parametr. Tato metoda je volána v případě, když chci přepočítat, jestli je stále nejvýhodnější ukládat hodnoty do stávajícího nebo jiného formátu.

Návratová hodnota má u všech funkcí stejný význam a informuje, do kterého formátu se má matice uložit. Pokud se však po provedení této metody rozhodnu uložit matici do jiného formátu, mohu tak stále učinit.

Metody typu `ZjistiFormat()` jsou používány ve většině operátorů a konverzních funkcích.

Je důležité si uvědomit, že tyto metody nevytvářejí matice a neukládají prvky. Metody provádějí v jednotlivých operátorech operaci naprázdno.

Uvedu příklad:

Předpokládám součet dvou řídkých matic. Zavolám si funkci `ZjistiFormat()`, které matice vnitřně sečte, ovšem výsledek neuloží. Vrátí pouze informaci, kam mám matici nyní uložit. Vzhledem k tomu, že metoda používá funkci `Setter()`, je důležité si uvědomit, že dokud jsem nezkontroloval celý obsah matice, stále nevím, kam ji budu ukládat. Proto se ukládání součtu děje ve funkci `Setter()`, když už vím, kam mám matici uložit.

Knihovna umožňuje i naplnění hodnotami pomocí `Set()`, kde toto dvojí sčítání není nutné. Takto je to koncipováno proto, abych předešel neustálým kontrolám formátů a konverzím interních formátu.

6.2 Metoda přepočet

```
void prepocet();
```

Tato metoda zjišťuje, kolik bytů je třeba pro uložení matice do jednotlivých formátů.

Protože knihovna není psána pro konkrétního uživatele, nevím, zda uživatele knihovny zajímá pouze počet dat uložených v matici, nebo i ukazatele, které se alokují při tvorbě polí potažmo vektorů.

Pokud tedy bude někdo chtít velikosti matic vyhodnocovat podle jiných aspektů, stačí změnit tuto krátkou metodu. Na tuto metodu se pak odkazují všechny ostatní složitější přepočetní metody.

6.3 Metoda pro konvertování formátů

Metoda zkonvertuje matici do požadovaného formátu. Pokud převádím matici např. z dvourozměrného pole do diagonálního tvaru, konverze bude provedena, ale přijdu o veškerá data, která byla uložena mimo hlavní diagonálu. Chci-li mít jistotu, že o data nepřijdu, pak je správné použití jako předávaný parametr předat výsledek jedné z metod `ZjistiFormat()`. Pokud je třeba převést matici na diagonální nebo trojúhelníkový tvar, musí být matice čtvercová.

Metoda je používána v metodě `Set()`, která mění prvky matice. Metoda konverze je jedna z nejdelších a nejnáročnějších metod. Proto není volána při každé změně, ale vyhodnocovací funkce, které taktéž trvají relativně dlouho, jsou volány při každém padesátém zápisu. Na jejich základě pak může nebo nemusí být zavolána konverze. Toto neplatí v případě, že zapisuji prvek mimo diagonálu a matice umožňuje ukládání prvků pouze do diagonály. V tom případě je konverze volána ihned, aby bylo možné uložit prvek.

Metoda konverze při změně formátu postupně konvertuje data z jednoho formátu do druhého. Nepoužívá při tom však destruktor, nýbrž odstraňuje jednotlivé prvky po částech, a to pokaždé jinak v závislosti na tom, „ze kterého do jakého“ formátu budou data konvertována.

Uvedu dva příklady konverzí matic ze současného formátu do jiného:

1.) Diagonální tvar-> CSR formát

Při tomto převodu formátů se ihned vytvoří všechny vektory pro uložení matice do CSR formátu, posléze se překopírují a následně se odstraní diagonální matice. Výsledek je tedy obdobný, jako kdybych používal pomocnou proměnnou a posléze destruktor. Odstraňování po prvku nebo po částech v tomto případě není implementováno, protože diagonální tvar je ve skutečnosti pouze jednorozměrné pole vytvořené pomocí `new()` a operátor `new()` nemá pro jednorozměrný ukazatel funkci měnící délku pole. Protože se zde jedná jen o jednorozměrné pole, nezabýval jsem se tvorbou funkce, která by tyto změny délek (alokací) prováděla.
Nevýhodou je tedy existence jednoho pole o délce počtu řádků matice navíc.

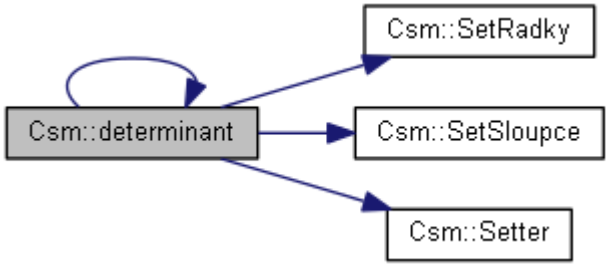
Výhodou je časová úspora strávená postupným odstraňováním a vytvářením polí/vektorů.

2.) Trojúhelníková matice-> CSR formát

Při tomto převodu formátů se nejprve vytvoří vektor informující o pozicích na řádku. Vektor pro uchování pozic sloupců a vektor pro uchování hodnot se pak tvoří postupně. Vektory se vytvoří pouze tak dlouhé, podle toho, kolik je prvků v **prvním řádku.** Po překopírování hodnot z prvního řádku se uvolní místo po prvním řádku v trojúhelníkové matici. Protože se v tomto případě jedná o dvojrozměrné pole, mohu jednotlivé řádky odstraňovat postupně. Není tedy nutné, aby byly najednou vytvořeny celé dvě matice. Existují vždy jen ty části matice, aby nedošlo ke ztrátě dat, přičemž data se odstraňují vždy po zpracování dat na jednom řádku. Obdobně se chovají konverze ostatních formátů vyjma diagonálních.

6.4 Metoda pro výpočet determinantu matice

Metoda počítá determinant matice Laplaceovým rozvojem, popřípadě Sarrusovým pravidlem. Parametrem funkce je řád matice.
Volané funkce z determinantu:



Obrázek 7 Seznam metod volaných z metody `determinant()`

Metody `SetRadky()`, `SetSloupce()` a `Setter()` jsou používány pro vytvoření subdeterminantů.
Determinant v rámci Laplaceova rozvoje také volá pro výpočet subdeterminantů opět metodu determinant, ale s nižším řádem. Jedná se tedy o rekurzivní metodu.

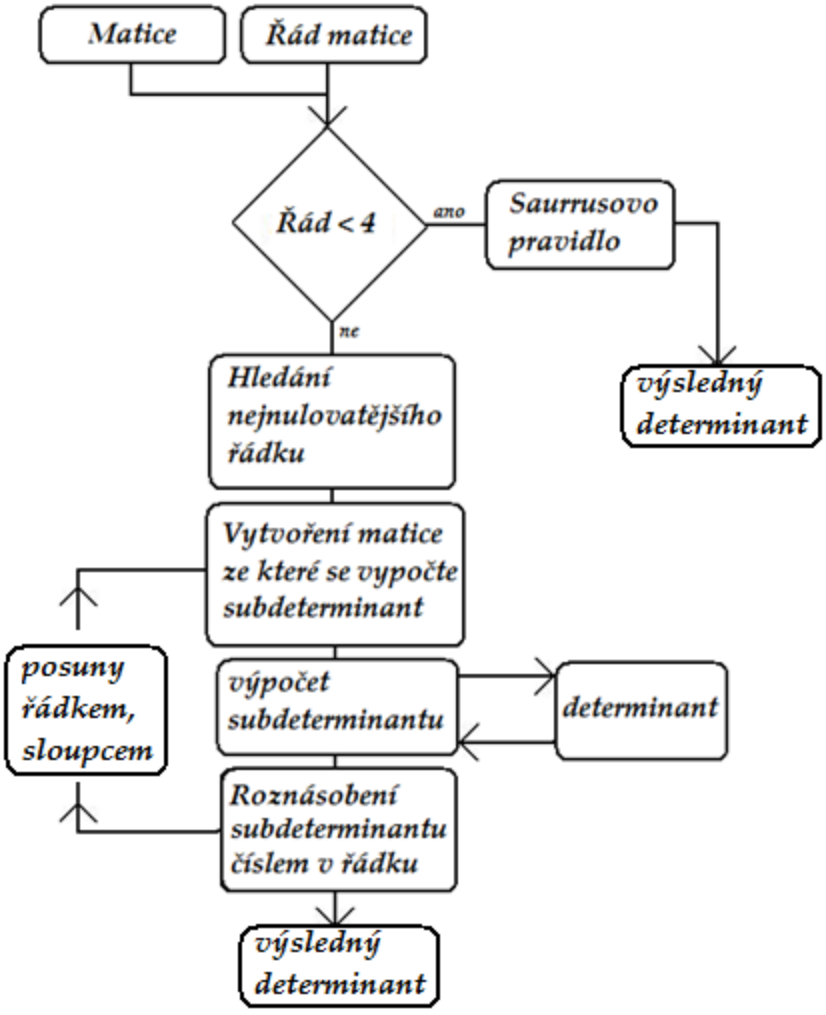
Způsob řešení:

Pro výpočet determinantu využívám Saurrusovo pravidlo, pro determinanty vyššího řádu než třetího používám Laplaceův rozvoj neboli Kofaktorovou metodu. Pomocí Laplaceova rozvoje můžeme rozvinout determinant podle řádku či podle sloupce, což umožňuje nám vypočítat determinant *n*-tého řádu. Potom rozvoj podle podle *i*-tého řádku umožňuje zapsat determinant řádu *n* jako součet determinantů řádu $(n-1)$. Zvolený řádek vyberu tak, aby v něm bylo co nejvíce nul.

Vztah pro výpočet determinantu:

$$\det A = \sum_{j=0}^{n} a_j^i C_i^j \qquad (3)$$

kde $\boldsymbol{C}$ jsou kofaktory, tedy -1^{j+i} krát determinant matice, která vznikne z matice $\boldsymbol{A}$ odstraněním i-tého řádku a j-tého sloupce. Taková matice se nazývá submatice a k ní příslušný determinant se nazývá subdeterminant. Ze vzorce je zřejmé, že je nejvhodnější využívat k rozvinutí matice řádek nebo sloupec, který obsahuje hodně 0. Převzato z [8].



Obrázek 8 Vývojový diagram algoritmu výpočtu determinantu

Popis algoritmu:

Nejprve vyhodnotím, jestli výpočet determinantu lze realizovat Saurrusovým pravidlem nebo Laplaceovým rozvojem. Pokud zjistím, že řád matice je vyšší než 3, tak nejprve nalézám řádek s největším počtem nul. Posléze po vynechání vyhledaného řádku a prvního sloupce vytvořím matici pro výpočet subdeterminantu, kterou uložím do řídkého formátu CSR. U této matice pak zjišťuji její determinant. Pro výpočet determinantu se opět volá funkce determinant, ovšem nyní s maticí o řád menší. Postupnými výpočty se vždy na subdeterminant zavolá determinant o řád nižší, až do momentu, kdy se dostanu na řád 3, kdy je výpošet realizován Saurussovým pravidlem.

Podle vzorce pak výsledný subdeterminant vynásobím číslem v řádku a posunu se na další nenulový prvek v řádku. To znamená, že vynechávám jiný sloupec a vytvářím novou matici pro výpočet determinantu. Po sečtení všech sčítanců dostávám výsledný determinant.

Výpočet determinantu pomocí Laplaceova rozvoje je při velkých rozměrech hodně časově náročný, a to hlavně za předpokladu, že by prvky v matici nebyly z větší části nulové a nejednalo by se tedy o řídkou matici. Jelikož však předpokládám, že velká část prvků bude nulová, bude tato metoda použitelná.

Laplaceův rozvoj jsem si vybral, protože jsme jím řešili determinanty vyšších řádů v předmětech BMA.

Jelikož při výpočtech zohledňuji především paměťovou náročnost, jsou jednotlivé matice pro subdeterminanty okamžitě odstraňovány po jejich výpočtu. To znamená, že k matici řádu *n* je vytvořena vždy maximálně jedna matice s řádem *n-1*. Navíc jsou uloženy v řídkých formátech. Bohužel neustálé odstraňování a vytváření těchto subdeterminantů zpomaluje výpočet.

Při výpočtech není u subdeterminantů zjišťováno, zda bude výhodné uložit matice do toho či onoho formátu, protože kontrolní algoritmy jsou relativně časově náročné, vzhledem k počtu opakování a době, po kterou je matice vytvořena. Jako výchozí je vždy volen formát řídký CSR. Vycházím z předpokladu, že matice je řídká.

6.5 Metoda pro výpočet inverzní matice

Tato metoda vytvoří inverzní matici pomocí vztahu (4) .

$$A_{ij}^{-1} = \frac{1}{\det A} * \text{adj } A_{ij} \qquad (4)$$

Výsledná inverzní matice je vytvořena na základě podílu determinantu s adjungovanou maticí z matice A.
Vytvořit inverzní matici není možné v případě, je-li determinant zvolené matice nulový.

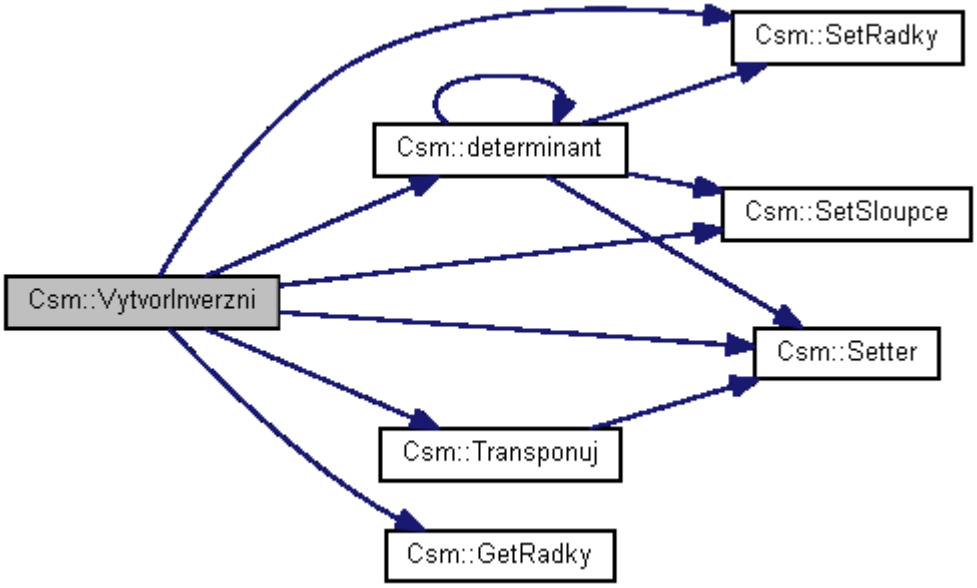
Adjungovaná matice je pak vytvářena následovně:

$$\operatorname{adj}\mathbf{A} = \begin{pmatrix} (-1)^{1+1}\mathbf{M}_{11} & (-1)^{2+1}\mathbf{M}_{12} & \cdots & (-1)^{n+1}\mathbf{M}_{1n} \\ (-1)^{1+2}\mathbf{M}_{21} & (-1)^{2+2}\mathbf{M}_{22} & \cdots & (-1)^{n+2}\mathbf{M}_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ (-1)^{1+n}\mathbf{M}_{n1} & (-1)^{2+n}\mathbf{M}_{n2} & \cdots & (-1)^{n+n}\mathbf{M}_{nn} \end{pmatrix}^T$$

Obrázek 9 Grafický popis výpočtu adjungované matice

Kde $\boldsymbol{M_{ij}}$ jsou determinanty matice, která vznikne vyškrtnutím *i*-tého řádku a *j*-tého sloupce z původní matice. Převzato z [7].

Metoda pro výpočet inverzní matice volá tyto funkce:



Obrázek 10 Volané funkce z metody `VytvorInverzni()`

Metody `SetRadky()`, `SetSloupce()`, a `Setter()` se používají pro vytvoření nové adjungované/ inverzní matice.

Pro výpočet adjungované matice jsou pak využívány metody pro transponování matice a pro výpočet determinantu.

Popis algoritmu:

Obdobně jako u determinantu, vytvářím nejprve matice pro výpočty subdeterminantů, které jsou potřeba pro výpočet adjungované matice. Tyto subdeterminanty uložím do matice a násobím -1^{j+i}. Následně matici transponuji a dostávám tak adjungovanou matici. Adjungovanou matici netestuji na vhodnost uložení, ale volím jako výchozí formát CSR. Nakonec násobím 1/determinant s adjungovanou maticí.

Budete-li pracovat s maticí typu *int,* nedočkáte se pravděpodobně správného výsledku, protože determinant bude zaokrouhlen stejně jako výsledek výrazu 1/determinant na celé číslo. Potřebujete-li pracovat s inverzní maticí nebo dělením matic, doporučuji vytvořit si matice zapouzdřující datový typ s vyšší přesností.

Alternativní metoda pro výpočetní algoritmus inverzní matice: Gauss-Jordanova eliminační metoda. Toto řešení však není implementováno.

6.6 Parametrický setter

Tato metoda umožňuje naplnit pole prvek po prvku a je úzce spjata s metodami `ZjistiFormat()`, které by měly předcházet použití tohoto settru.

Funkce má 4 parametry:

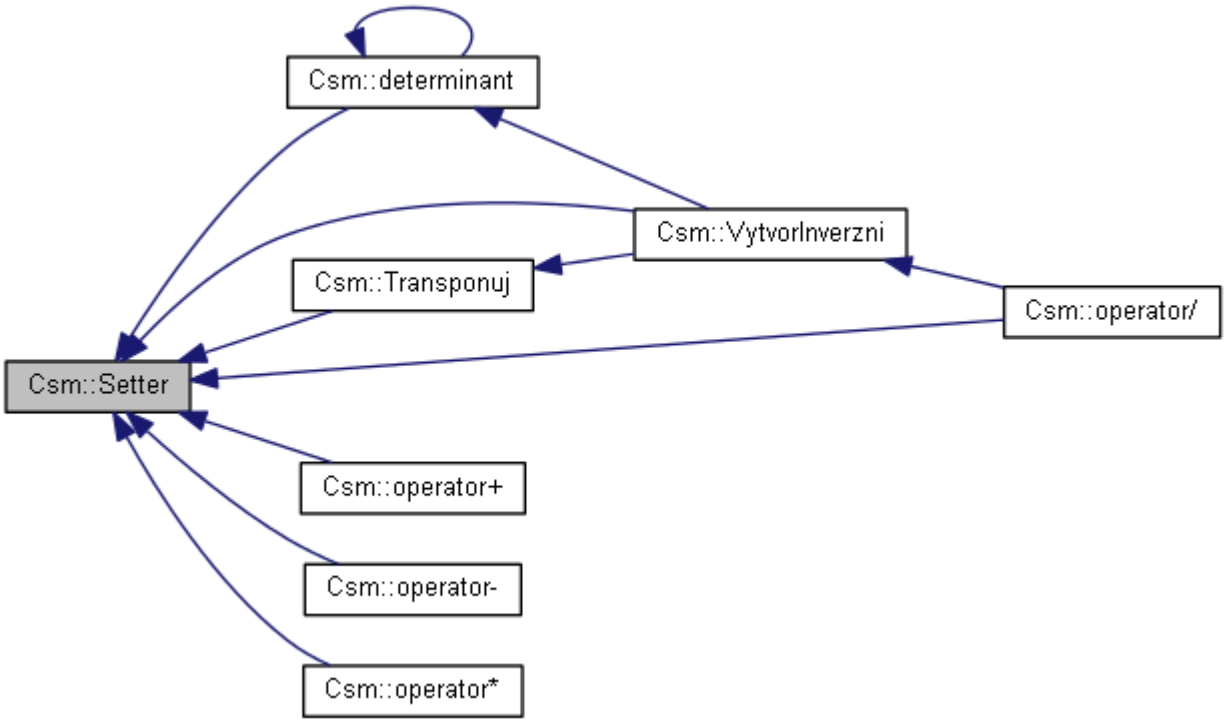
1.) Hodnota, kterou chci uložit na danou pozici

2.) Souřadnice řádku

3.) Souřadnice sloupce

4.) Informace o formátu, do kterého chci uložit matici. Tento parametr vy měl vzejít z některé z funkcí `zjistiFormat()`.

Tuto funkci volají:



Obrázek 11 Seznam metod využívající metodu `setter()`

Tuto funkci volají téměř všechny ostatní metody nebo operátory, protože je mnohem jednodušší než funkce `Set()`, kterou by měl používat uživatel při naplňování pole. Neobsahuje totiž žádné kontroly formátů ani konverze. Používá se tehdy, je-li předem známo, do jakého formátu je třeba matici uložit.

Matice, kterou chcete naplnit, už musí být připravená. To znamená, že musí mít nenulové rozměry, a jedná-li se o diagonální nebo a trojúhelníkovou variantu matice, musí mít nastaveny i informace o diagonálách. Toto zajišťuje jakákoliv z metod `ZjistiFormat()`.

6.7 Bezparametrický setter

```
bool Set(const T& hodnota,unsigned int radek,unsigned int sloupec);
```

V tomto případě je pojmem bezparametrický myšleno pouze to, že metodě chybí vstupní informace o výsledném formátu.

Funkce má 3 parametry:

1.) Hodnota, kterou chci uložit na danou pozici

2.) Souřadnice řádku

3.) Souřadnice sloupce

Na rozdíl od metody `Setter()` nemá tato metoda vstupní parametr reprezentující informaci o tom, do jakého formátu chci uložit matici. Metoda si umí sama zjistit, do jakého formátu je nejvhodnější uložit matici. Tato metoda již není úzce spjata s metodami `ZjistiFormat()`, které nemusí předcházet naplňování matice. Dopředu však musí být zavolán některý z konstruktorů. V případě, že se jedná o implicitní konstruktor, musí být nastaveny ještě rozměry matice, aby bylo matici možné naplnit.

V případě, že naplňuji matici prvek po prvku, vytvoří se jako výchozí CSR formát. Po zapsání prvních 30ti prvků je zjišťováno, zda je vhodné matici zkonvertovat do jiného formátu, a pokud ano, učiním tak pomocí metody `konverze()`. Zjišťování vhodnosti konverze je kvůli časové úspoře následně prováděno při každém padesátém zápisu. Navíc po zapsání jednoho jediného prvku není žádoucí, aby byla matice konvertována kvůli paměťové úspoře. Výjimkou je případ, kdy díky nevhodnému současnému formátu matice (např. zapsání prvku mimo diagonálu u diagonálního formátu uložení matice) by mohlo dojít ke ztrátě dat. Potom matice mění formát okamžitě pomocí metody `konverze()`.

Tato metoda není využívána v ostatních funkcích popř. operátorech. Místo ní je používána metoda `Setter()`, která je značně jednodušší. Vzhledem k faktu, že v operátorech předem vím, do jakého formátu je potřeba matici uložit (nejsou postupně přidávány další prvky matice), není již nutno opětovně kontrolovat formát.

6.8 Metody pro nastavení a změnu rozměrů

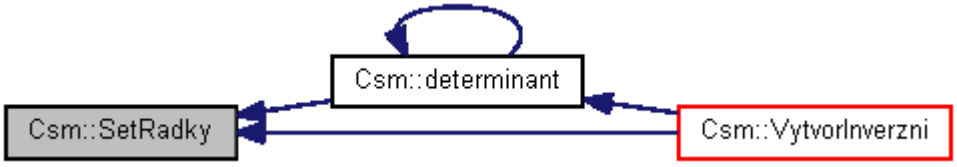
Knihovna má dvě metody pro nastavení rozměrů, a to `SetRadky()` a `SetSloupce()`. Tyto dvě metody umožňují nejen nastavit velikost matice, ale dokážou rozměry změnit a tím upravit rozměry i hodnoty v celé matici.

Není problém zvětšit matici. Pokud však chcete zmenšit rozměry, tyto dvě metody to sice umožní, přijdete však o nenulové hodnoty, které byly mimo nové rozměry. Tuto ztrátu je možně kontrolovat pomocí návratové hodnoty.

Popis funkce:

Při změně velikosti je matice automaticky převedena do CSR formátu, který jediný umožňuje změny rozměrů matice. CSR formát je použit vždy, protože pro formáty jako diagonální tvar a trojúhelníkový tvar jsou rozměry stěžejní a musí být stejné (počet řádků = počet sloupců), z toho důvodu je při změně formát změněn na CSR. Také s ohledem na skutečnost, že ukazatele vytvořené pomocí operátoru `new()` nemají předdefinovanou metodu pro změnu rozměrů. Bylo by nutné vytvořit jiný ukazatel. V jednom okamžiku by tedy existovala dvě pole hodnot. Obdobného efektu se dá docílit při zavolání parametrického konstruktoru. To je důvod, proč jsem uživateli neumožnil měnit rozměry v jiném než CSR formátu, kde jsou data ukládána do vektorů z knihovny `vector.h`. Tyto vektory umožňují změnu velikosti pomoví metody `resize()`. Není problém následně kdykoliv matici převést zpátky do původního nebo jiného formátu.

Funkce je volána:



Obrázek 12 Seznam metod využívající funkci `SetRadky()`

V metodách `determinant()` a `VytvorInverzni()` je používána funkce `SetRadky()` na nastavování rozměrů subdeterminantů.

6.9 Kontroly správného výpočtu metod

Jednotlivé metody, funkce a operátory byly otestovány a byla ověřena jejich správná funkčnost. Výsledky operací pak byly porovnány se stejným zadáním ve vývojovém prostředku MATLAB. Všechny mnou provedené testy jednotlivých operátorů a funkcí byly potvrzeny pomocí MATLABu. Samozřejmě s ohledem na přesnost datových typů, do kterých je ukládán výsledek. Například u výpočtu determinantu matice obsahující prvky typu `int`, je zaokrouhlení hodnot razantní a výsledek nevychází správně. Algoritmy jednotlivých metod a operátorů byly otestovány na maticích různých obsahů do rozměrů matic 10 x 10 při kontrole prvek po prvku. Zkušební příklady jsem si vytvořil sám, nebo

jsem si je nechal vygenerovat generátorem náhodných čísel. Byla otestována správná funkčnost matic i s nulovými hodnotami. Nejvíce testů bylo provedeno na funkci pro výpočet determinantu. Tato funkce při vyšších řádech nelze odkrokovat, protože je rekurzivní a celý výpočet má velké množství řádků. Proto je správnost algoritmu usouzena na základě shodných výpočtů se systémem MATLAB. Pro testování správnosti knihovny je v příloze na CD soubor `testovaci_program_matice.cpp`, pomocí kterého je možné zkontrolovat výsledky jednotlivých operací.

7 ÚVOD DO ČÍSEL S POHYBLIVOU ŘÁDOVOU DESETINNOU ČÁRKOU

Desetinná čísla se do paměti počítače zapisují ve formátu tzv. **mantisa/exponent,** což je vhodný způsob zápisu jak velmi velkých, tak i velmi malých desetinných čísel. Zápis je pak rozdělen následovně: $a \ x \ 2^b$, kde ***a*** je mantisa a ***b*** je exponent. Exponent je celé číslo. Mantisa je číslo, i desetinné, od 1 (včetně) do 10 (vyjma). Převzato z [5].

Značnou výhodou tohoto zápisu je, že čísla, která by byla zapsána v běžném zápisu a byla by velmi dlouhá, budou zapsána v krátké formě. Délka krátké formy je přímo úměrná přesnosti, tudíž počtu vyčíslených platných cifer.

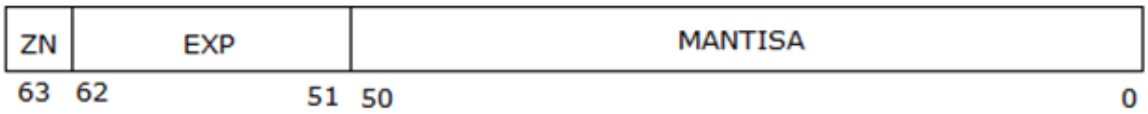
Jedním ze základních standardů pro uládání čísel s pohyblivou desetinnou čárkou, je standard IEEE 754 neboli standard IEEE pro dvojkovou aritmetiku v pohyblivé řádové čárce. Je to nejrozšířenější standard pro výpočty v pohyblivé řádové čárce, který používá mnoho mikroprocesorů a jednotek FPU.

IEE 754 definuje tyto formáty čísla:

single precision	jednoduchá přesnost	32 bitů
double precision	dvojnásobná přesnost	64 bitů
double extended precision	rozšířenou dvojitou přesnost	80 bitů

mimo jiné i další formáty, které jsou méně používané.

Uložení čísla pak vypadá v případě double precision následovně:



Obrázek 13 Způsob uložení čísla s pohyblivou řádovou čárkou

7.1 Návrh třídy zapouzdřující v sobě číslo s pohyblivou řádovou desetinnou čárkou

Předpokládám číslo uložené podobně, ne však stejně, jako ve formátu mantisa/exponent. Pro mantisu platí stejná pravidla. Exponent však nebude umocňovat dvojku, ale desítku. Výsledné číslo je pak získáno podle vztahu (5), kde *a* je mantisa a *b* je exponent.

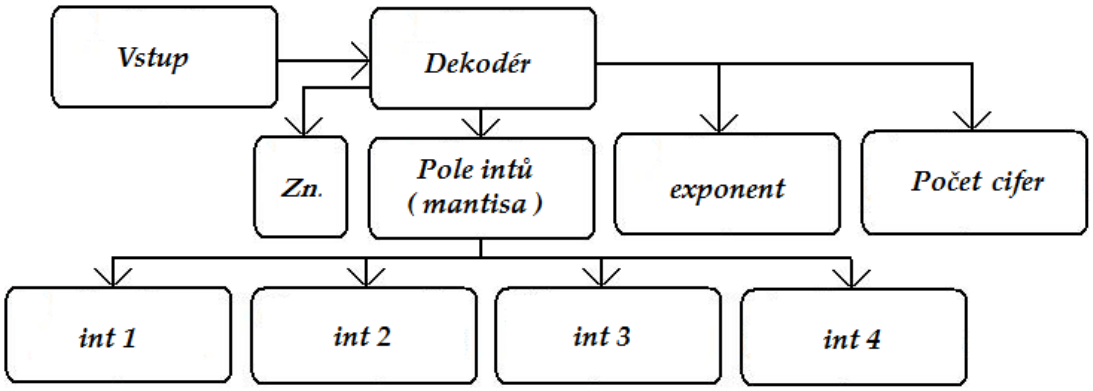
$$a \ x \ 10^b \qquad (5)$$

Mantisu bude realizovat jako pole celočíselných datových proměnných `MantiseType` např: *intových*, ve kterých bude **po částech** uložena mantisa proměnné. Uvažoval jsem také o uložení mantisy do pole znaků typu `*char`, ale toto řešení bude pravděpodobně méně praktické vzhledem k tomu, že se snažím vytěžit více uložených cifer na bajt. Při ukládání dvou cifer do jednoho bajtu (jako v případě `charu`) pak nemám žádnou výhodu oproti kódování v BCD kódu. Exponent bude vždy hodnota typu `int`. Tím, že bude exponent hodnota uložená v proměnné typu *int,* omezím maximální velikost přesného desetinného čísla na 10^{-32768} - 10^{32767}. [5]

Základní myšlenka a rozdíl od např. typu `double` je, že budu schopen uložit číslo s téměř nekonečnou přesností. Proto je mantisa realizovaná jako pole celočíselných datových proměnných, které se zvětšuje podle potřeby. Uložení vysokého počtu cifer v proměnné typu `double` není možné, ta je omezena na 8 bytů v paměti. Z toho 11 bitů je vymezeno pro uložení exponentu, a 52 bitů je vymezeno na mantisu. Datový typ `double` umožňuje v desítkové aritmetice uložit číslo s přesností na 15 desetinných míst. [5]

V mantise bude vždy uloženo číslo rozložené po určitém počtu cifer v závislosti na velikosti datového typu. Datový typ mantisy je možné měnit v knihovně. Při použití většího datového typu dosáhneme menšího počtu alokací. Mantisa je rozdělena na méně větších kousků. Při použití menšího datového typu dosáhneme zase větší paměťové úspory v případě, že je třeba uložit číslo na menší počet platných cifer. Nyní jsou předdefinované tři možnosti datových proměnných pro uložení mantisy (`int, long int, long long int`). Pro uložení mantisy však může být použit jakýkoliv ze **znaménkových** celočíselných datových typů. Typy musí být znaménkové, protože při odčítání dosahují jednotlivé dílčí rozdíly záporných hodnot. Pro použití bezznaménkových datových typů by se musel přepracovat algoritmus odčítání.

Při matematických operacích je pak pracováno s mantisou jako s celočíselným číslem, ze kterého jsou jednotlivé cifry vyčíslovány pomocí zbytku po dělení a celočíselného dělení.



Obrázek 14 Grafický popis uložení přesného desetinného čísla

Příklad 1:

Berme v úvahu, že chci vytvořit přesné číslo z řetězce, ve kterém je uloženo číslo "1234,123456789012000". Dekodér (v tomto případě konstruktor) zjistí počet platných cifer, které chceme uložit, a to 16. Na základě tohoto údaje naalokuje pole dvou proměnných typu `int` (dílčí mantisy). A dále do exponentu uloží hodnotu 3, protože 1,234 * 10^3 = 1234. Potom do paměťové buňky typu `int` uloží hodnotu rovnající se počtu platných cifer. To z toho důvodu, abych nemusel při každé operaci s číslem počítat, kolik cifer je uložených v mantise, což by zpomalilo výpočty. Do jednotlivých `intů` se nyní uloží cifry mantisy, a to vždy po devíti, vyjma poslední dílčí mantisy kam se uloží zbylý počet cifer.

Zvolený datový typ pro uložení mantisy je `int`, v tomto případě bude o velikosti 4B.

Výsledné uspořádání v paměti:

Mantisa (pole intů):
```
Int 1:     56789012
Int 2:     12341234
```

Exponent (int):
```
3
```

Počet cifer (int):
```
16
```

Znaménko (`bool`):
```
True
```

Celkový počet zabraných bytů za předpokladu, že `int` je 4 B*: 2*4+4+4+1=16* bytů.

Jestliže chci následně pracovat s jednotlivými ciframi (operátory +, -, *, /), využívám pro jejich vyčítání celočíselného zbytku po dělení a celočíselného dělení.

<u>Příklad 2</u>:

Z čísla z předchozího příkladu chci zjistit číslo s vahou 10^2. Z celkového počtu cifer a exponentu čísla zjistím, ve kterém z `intů` se číslo nachází, a následně se jej snažím vyčíslit. Dílčí mantisa je v tomto případě druhý `int` (mantisa[2]).

```
zbytek = dilci_mantisa % 10^8;  //123 412 345 % 100 000 000 = 23 412 345;
výsledek = zbytek / 10^7;       //23 412 345 / 10 000 000 = 2;
                                //výsledek = 2;
```

Tímto způsobem jsou získávány jednotlivé cifry ve všech operacích.

7.2 Seznam konstruktorů přesného čísla

Číslo je možné vytvořit implicitně, z řetězce číselných znaků včetně desetinné čárky nebo desetinné tečky, jako kopii, z proměnné typu `int`, `double` nebo `float`.

```
ENumber();
ENumber(const char* cislo);
ENumber(const ENumber& cislo);
ENumber(const float cislo);
ENumber(const double cislo);
ENumber(const int cislo);

~ENumber();
```

Číslo je možné vytvořit na základě těchto konstruktorů. Konstruktory využívají metody `VytvorZRetezce()`, `VytvorZInt()` atd… Těmito metodami se dá nastavit velikost čísla. Číslo se dá také načíst z konzole pomocí operátoru `<<`. Metody jsou ošetřeny tak, aby dokázaly správně vytvořit číslo i z ne zcela standardní posloupnosti znaků.

Například:

Číslo tvořím funkcí `VytvorZRetezce()` nebo pomocí streamu `<<`. Číslo, které se má vytvořit z řetězce `".567"` výsledek je `"0.567"` nebo `"976."` výsledek je `"976"`.

7.3 Seznam metod a operátorů umožňující práci s přesnými desetinnými čísly.

`MantiseType` je typ proměnné, do které je ukládána mantisa.

Metody pro získání dílčích informací o čísle:

```
int GetPocetCifer() const;
MantiseType* GetMantisu() const;
MantiseType GetCifruRadu(int rad) const;
```

Metody využívané při konstrukci čísla:

```
void SetMantisu(MantiseType* amantisa);
void VytvorZRetezce(const char* retezec);
void VytvorZInt(const int cislo);
void VytvorZDouble(const double cislo);
void VytvorMantisu(const int cifer_max);
void OdstranNuly();
```

Matematické metody:

```
ENumber abs() const;
ENumber umocni(const int mocnitel) const;
ENumber inverzni() const;
```

Streamy:

```
friend std::ostream& operator<<(std::ostream& oBuffer, const ENumber& cislo);
friend std::istream& operator>>(std::istream& iBuffer, ENumber& cislo);
```

operátory:

+	Plus
-	Mínus
*	Krát
/	Děleno

Logické operátory:

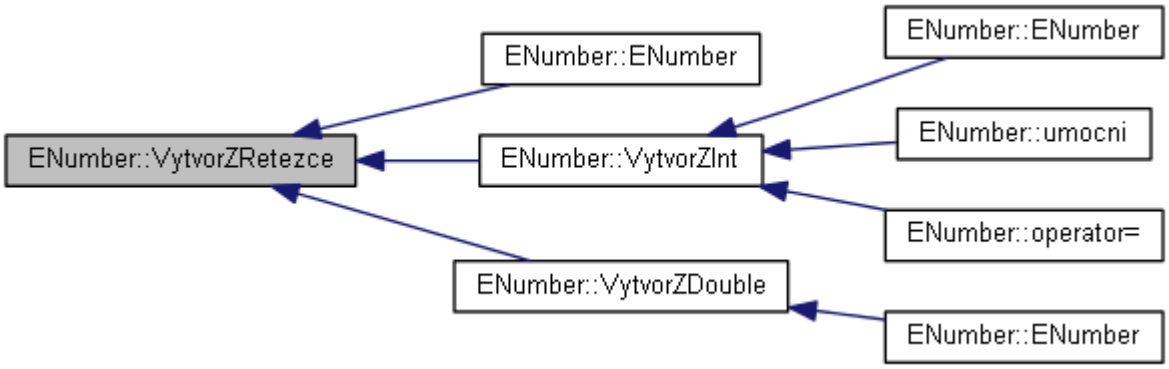
`==` , `!=` , `<` , `<=` , `>=` , `>` mezi dvěmi čísly, s datovým typem `int` a `double`.

8 POPISY METOD A OPERÁTORŮ

V této kapitole se pokusím nastínit principy složitějších funkcí a vysvětlit jejich chování.

8.1 Metoda pro vytvoření přesného čísla z řetězce znaků.

Tuto funkci volají:



Obrázek 15 Seznam metod volajících funkci `VytvorZretezce()`

Tato metoda je využívána takřka ve všech konstruktorech a u operátorů =.

Metoda má jeden parametr a to řetězec `char` znaků. Na tento řetězec jsou kladeny následující požadavky: musí obsahovat pouze znaky `0` – `9`, `-` , desetinnou tečku,`\0`, nebo mezeru, která není umístěna mezi ciframi. Může obsahovat i desetinnou čárku, ta bude převedena na tečku.

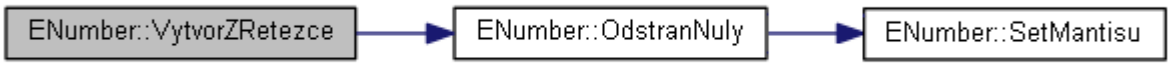
Popis algoritmu:

Nejprve jsou odstraněny přebytečné mezery před a za řetězcem. Tyto mezery mohou vzniknout například při použití funkce `sprintf()`.

Dále je zjištěno, jestli řetězec obsahuje znaménko, desetinnou tečku nebo čárku. Pokud ano, je řetězec překopírován bez čárky a znaménka a je nastavena hodnota exponentu a znaménka.

Nyní je vytvářena mantisa čísla na základě takto upraveného řetězce. Mantisa je reprezentována jako ukazatel na datové proměnné typu `MantiseType`, což může být `int`, `long long int`, `short` atd… Do jednotlivých proměnných (dílčích mantis) je vždy uložen striktně daný počet cifer (kromě poslední proměnné) podle jejich velikosti. Počet těchto dílčích mantis je vypočítáván z celkového počtu cifer.

Tato metoda při tvorbě čísla využívá další metody:



Obrázek 16 Seznam metod volaných z `VytvorZRetezce()`

Mantisa čísla je nejprve vytvořena i s nutnými přesahy, jelikož dopředu nezkoumám, kolik nul je v řetězci nevýznamných. To jsou nuly, které jsou na konci čísla a za nimiž již není žádná nenulová hodnota s nižší vahou. Tyto nuly je třeba odstranit. Na to jsou používány metody: `OdstranNuly()`, ta odstraní nuly, které jsou uloženy navíc, a `SetMantisu()`, která odstraní přebytečné alokace paměti vymezené na mantisu.

Metoda `VytvorZRetezce()` je využívána téměř ve všech konverzních operátorech. Například při porovnání čísla s `intem` je porovnání realizováno jako porovnání dvou přesných čísel, přičemž z `intu` je tvořeno přesné číslo pomocí metody `VytvorZInt()`, která obsahuje metodu `VytvorZRetezce()`.

Metody `VytvorZFloat()`, `VytvorZDouble()` a `VytvorZInt()` využívají tuto metodu. Datová proměnná je vždy převedena na řetězec a pomocí metody `VytvorZRetezce()` převedena na přesné číslo.

8.2 Metoda pro nastavení mantisy čísla

```
void SetMantisu(MantiseType* amantisa);
```

Mantisa může být nastavena na základě mantisy z jiného čísla.

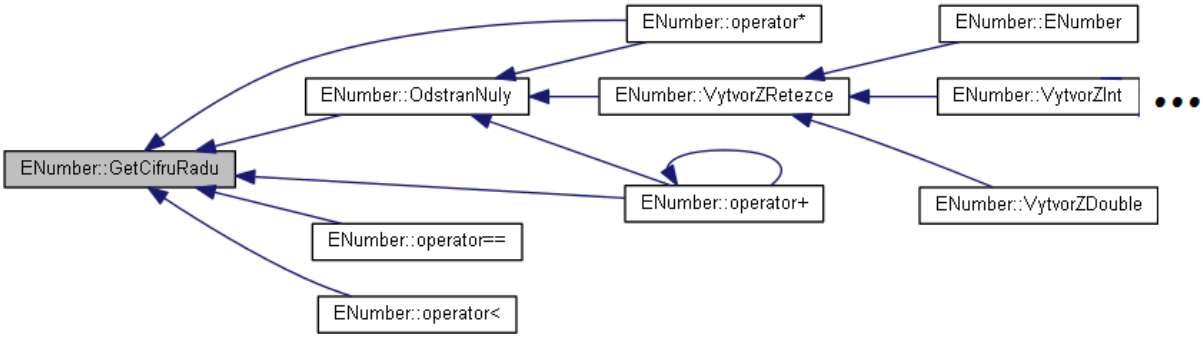
Tuto metodu využívají téměř všechny operátory. Kromě toho, že nastaví mantisu čísla, umí tato metoda i realokovat mantisu a odstranit tím přebytečné alokace.

8.3 Metoda pro vyčtení cifry čísla dané váhy

```
MantiseType GetCifruRadu(int rad) const;
```

Tato funkce má jeden parametr, který udává váhu cifry, kterou chci vrátit.

Tato metoda je volána:



Obrázek 17 Seznam metod používající `GetCifruRadu()`

Tato metoda je využívána hlavně v operátorech. Pomocí této metody jsem schopen například sčítat dvě čísla tak, že sčítám jednotlivé cifry se stejnou váhou a přičítám k nim přenosy.

Popis algoritmu metody:

Metoda si podle váhy číslice zjistí, ve které z dílčích mantis je požadovaná cifra. Cifru je potřeba vyjmout z celého obsahu datové proměnné typu `MantiseType`.

Příklad:

`MantiseType` je v tomto případě proměnná typu `int` o velikosti 4B. Potom do dílčí mantisy ukládám maximálně 9 cifer. Např. číslo 123456789. Nyní chci získat číslici s vahou 10^4, což je v tomto případě „6". Výslednou cifru získám pomocí zbytku po dělení čísla vyššího řádu a celočíselného dělení vahou.

```
zbytek = dilci_mantisa % 10^4;   //123456789 % 10000 = 6789;
výsledek = zbytek / 10^3;        //6789 / 1000 = 6;
                                 //výsledek = 6;
```

Tímto způsobem jsou pak získávány cifry čísla u všech základních operací.

Alternativou tohoto řešení by mohlo být uložení mantisy v dílčích mantisách pomocí BCD kódu. Poté by se nemuselo používat celočíselné dělení a zbytku dělení, které je pro výpočetní výkon poměrně náročné. Při použití BCD kódu by bylo možné cifry separovat pomocí bitových posunů, jelikož jedna číslice je reprezentována vždy čtyřmi bity. A to hodnotami 0000 až 1001.

Bohužel jsem se rozhodl pro alternativu s celočíselným dělením, což ubírá knihovně na výpočetní rychlosti. Povedlo se mi však z toho vytěžit možnost ukládání většího počtu cifer v rámci jedné proměnné. Pro BCD kód platí, že do jednoho bajtu se vlezou 2 desítkové cifry. Pro používané kódování platí rovnice (6), kde n je počet bajtů datové proměnné typu `MantiseType`.

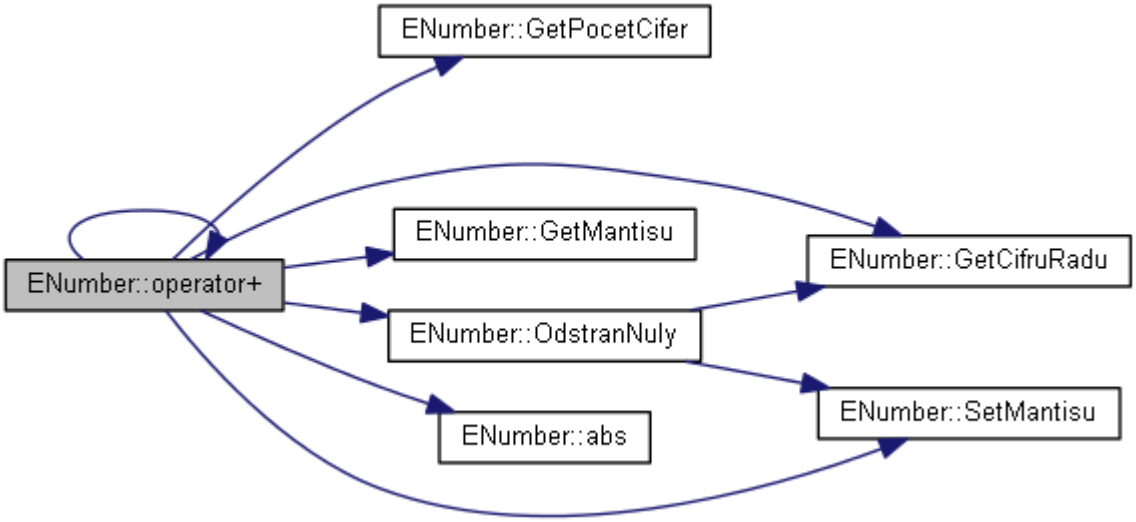
$$počet\ cifer = 2 * n + \frac{n}{3} \qquad (6)$$

8.4 Operátor plus

```
ENumber operator+(const ENumber& cislo)const;
```

Tento operátor sčítá dvě přesná čísla typu ENumber.

Tato funkce volá:



Obrázek 18 Seznam metod, které využívá operátor +

Operátor využívá většinu výše popsaných metod pro výpočet součtu čísel.

Popis algoritmu operátoru +:

$$\begin{array}{r} 8234 \\ +3234 \\ \hline 11468 \end{array}$$

Obrázek 19 Sčítání dvou čísel

Algoritmus pracuje obdobně jako klasické sčítání, které provádím pod sebou.
Resp. sčítání je realizováno od nejnižší váhy, jako součet cifer se stejnou vahou. K součtu je pak přičítán přenos.
Operátor nejprve zjistí nejnižší váhu cifer, které bude sčítat. Pro vyčíslení cifry je používána funkce `GetCifruRadu()`, tato metoda vrací `"0"` v případě, že cifra dané váhy v čísle není, resp. je nulová. Takže mohu sčítat čísla i přes to, že mají různou nejnižší váhu čísla. Pro mantisu je vymezeno místo počítající i s přenosem čísla na nejvyšším řádu.

Cifry jsou tedy sčítány od nejnižší váhy. Při součtu může dojít k jednomu druhu přenosu. U násobení pak ke dvěma.

1.) Přenos způsobený tím, že součet je větší než 9.

    Přenos se potom přičte k součtu cifer vyššího řádu.

2.) Přenos je způsobený tím, že číslo v mantise překročilo stanovený počet cifer.

    Tento přenos nastane, protože násobení je realizováno jako postupné sčítání a sčítána je jedna cifra z prvního čísla s prozatímním součtem uloženým v dílčí mantise. Při překročení počtu cifer je nejvyšší cifra z dílčí mantisy odstraněna a přenos je nastaven na 1.

Tímto postupným sčítáním jednotlivých cifer získám správný výsledek. Během získávání výsledku zjišťuji, jestli výsledné cifry na nejnižších a nejvyšších řádech nejsou nuly, které nejsou potřeba ukládat do mantisy. Na základě těchto údajů pak zmenším počet platných prvků a zavolám metody `SetMantisu()` a `OdstranNuly()`, které realokují mantisu a přebytečné nuly odstraní.

Jelikož znaménko čísla je jeho součástí, funguje operátor + také jako odečítání. Princip je obdobný jako u součtu. Nejprve jsou porovnána obě čísla, abych odečítal menší číslo od většího. Posléze jsou čísla odečítána a místo přenosů je pracováno s tzv. výpůjčkami.

Výsledné znaménko je uloženo nakonec podle znamének obou čísel a jejich velikosti.

8.5 Výpočet inverzní hodnoty čísla

```
ENumber inverzni() const;

B = A.inverzni();
```

Metoda vrací hodnotu `1/A`

Tuto funkci volají:

Obrázek 20 Seznam operátorů volající metodu `inverzni()`

Tato metoda je využívána při podílu dvou čísel.
Podíl je pak realizován jako `A*(1/B);`

Popis algoritmu:

Algoritmus pracuje obdobně jako u dělení.

$$1 : 125 = 0{,}0080$$
$$1000 : 125 = 8 \qquad exp = -3$$
$$(1000 - 125 * 8) * 10 : 125 = 0$$

Obrázek 21 Ukázka klasického dělení

Jednotlivé cifry jsou získávány jako celočíselný násobek dělitele, který se vejde do desetinásobku zbytku po dělení.

Nejprve se vytvoří dělenec o řád vyšší než dělitel. To znamená, že v případě čísla 125 začínám dělit rovnou číslo 1000 a nikoli jedničku. Nyní pomocí operátoru >= zjistím, kolikrát se do tohoto čísla (zbytku) vejde dělitel, a výsledek zapíši. Potom dopočítám zbytek po dělení. Zbytek po dělení pak zvětším o řád a znovu provádím porovnání, kolikrát se dělitel vejde do zbytku. Jestliže je zbytek menší než setina dělitele (minimálně dvě jdoucí nuly po sobě) je dělení ukončeno. Pokud jsou čísla nesoudělná více než na určitou přesnost (přesnost dělitele + 20 cifer), je číslo zaokrouhleno na počet cifer dělitele rozšířeného o 20 platných cifer.

Na konci je třeba mantisu upravit do správného tvaru. U všech operací jako `+`, `-`, `*` je výsledek počítán od nejnižších řádů. Mantisa je také zapsána od nejnižších řádů. Při dělení však byla mantisa zapisována od nejvyšších vah, takže na konci je mantisa přeskládána do správného tvaru.

Metoda `inverzni()` je využívána v operátoru dělení. Vzhledem k nutnému zaokrouhlení na neznámý konečný počet platných cifer, může vzniknout chyba při dělení. Jedna z možností je zvýšit přesnost nesoudělných čísel. Další možností, která by připadala v úvahu, ale bohužel jsem se k tomu již nedostal, je zapisovat do čísel informace o periodicitě. S tímto údajem by se pak daly eliminovat některé zdroje chyb. Také by se potom nemuselo ukládat číslo celé, ale pouze první perioda.

9 ZHODNOCENÍ NÁVRHŮ

Součástí návrhu je i zhodnocení paměťové náročnosti a rychlosti výpočtů.

Moje hardwarová konfigurace je:

CPU:	AMD Turion™ dual-core 2.0 GHz
RAM:	DDR2 2 x 2 GB
GRAFICKÁ KARTA:	ATI Mobility Radeon™ HD 3470
OS:	Windows Vista (64 bit)
Vývojové prostředí:	Windows Visual Studio 2010 (32 bit)

9.1 Paměťové zhodnocení

Pro kontrolu paměťové správnosti, eliminaci memory leaků a správného odstraňování objektů byly použity tyto nástroje:

Intel Parallel Studio 2011

Konkrétně nástroj **Intel Parallel Inspector** – tento nástroj je určen k zjištění špatných destrukcí objektů, chybějících destrukcí, částečných destrukcí a indikaci paměťových úniků.

Pomocí tohoto nástroje se mi povedlo eliminovat velké množství chyb a částečně odstraněných objektů.

Microsoft Visual Studio 2010 Premium

Konkrétně z nástrojové sady **Analyzes** .NET Memory Allocation. Bylo nutné program zkompilovat jako CLR aplikaci.

Knihovna `check.h`

Posledním nástrojem pro indikaci a kontrolu memory leaků a počty zabrané dynamické paměti jek knihovna `check.h` která je výsledem bakalářské práce Pavla Sabatky a je využívána ve výuce předmětu BPPC**.** Tato knihovna nabízí indikaci neodalokované paměti, špatně odalokované paměti, informace o alokované paměti a také informaci o špičkovém množství vymezené paměti v průběhu programu. Dovoluje také zjišťovat průběžný stav alokované paměti pomocí funkce `memory_stat()`.

Pomocí této knihovny byly neustále kontrolovány všechny paměťové operace. Podle knihovny `check.h` byly také nastaveny informace o zabrané paměti v jednotlivých formátech. Jedná se samozřejmě pouze o dynamickou paměť. Byly také odstraněny přebytečné alokace v některých metodách a operátorech například u streamů.

Knihovnou `check.h` bylo ověřeno odstraňování matice ve funkci `konverze()` a také bylo ověřeno správné odstraňování matic pro výpočet subdeterminantů u metody `determinant()`.

Příklad:

Program počítající determinant matice:

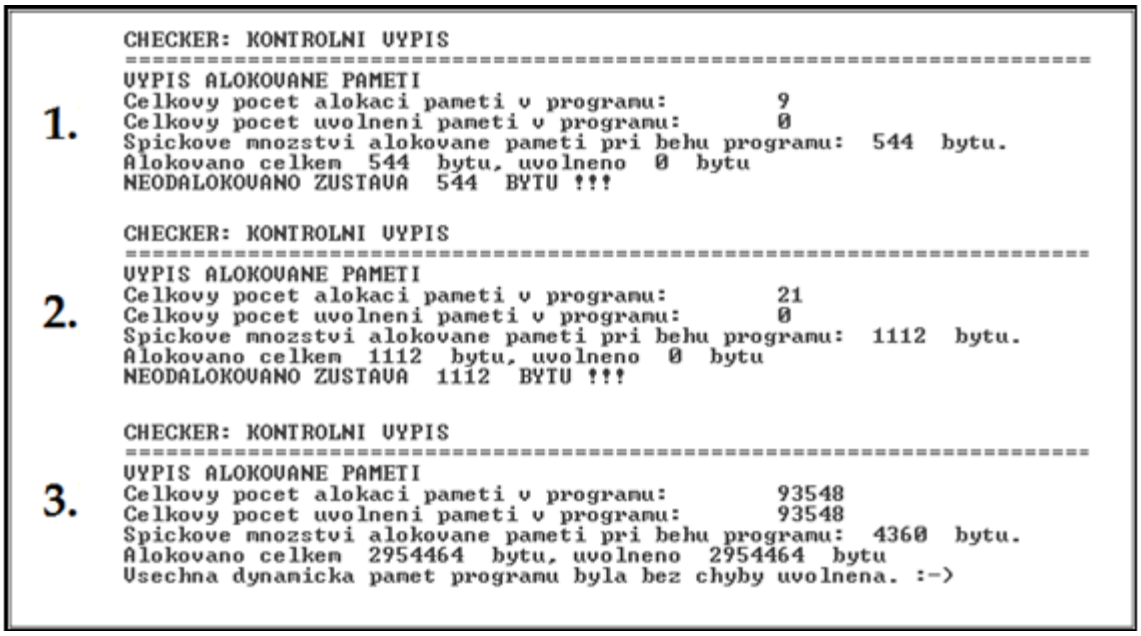
```
#define U double
#define X 8
int main()
{
	int i,j;
	double x;
	U **pole=nullptr;				// alokace matice cisel

	pole = new U*[X];				// vytvoreni matice cisel
	for(i=0;i<X;i++)
		pole[i] = new U[X];

	for(i=0;i<X;i++)				// naplneni matice cisel
	{
		for(j=0;j<X;j++)
		{
			pole[i][j]=i+j;
		}
	}
	memory_stat();
	Csm<U> matice(pole,X,X);			// vytvoreni ridke matice s cisly
	memory_stat();					// PRVNI VYPIS PAMETI
	matice.prepocet();
	cout << matice;					// vypis matice cisel
	memory_stat();					// DRUHY VYPIS PAMETI
	x=matice.determinant(8);

	for(i=0;i<X;i++)
		delete[] pole[i];
	delete[] pole;
	pole=nullptr;
	return 0;
}							// TRETI VYPIS PAMETI
```

Obrázek 22 zaznamenává jednotlivé výpisy provedené v průběhu programu.



Obrázek 22 Výpisy stavu paměti při výpočtu determinantu pomocí `check.h`

Popis paměťových výpisů:

1.) První výpis informuje, kolik bytu si pro sebe vymezilo dvourozměrné pole, ze kterého budu vytvářet matici. Hodnota je 544 bytů, což odpovídá `8*8*sizeof(double)+32`.

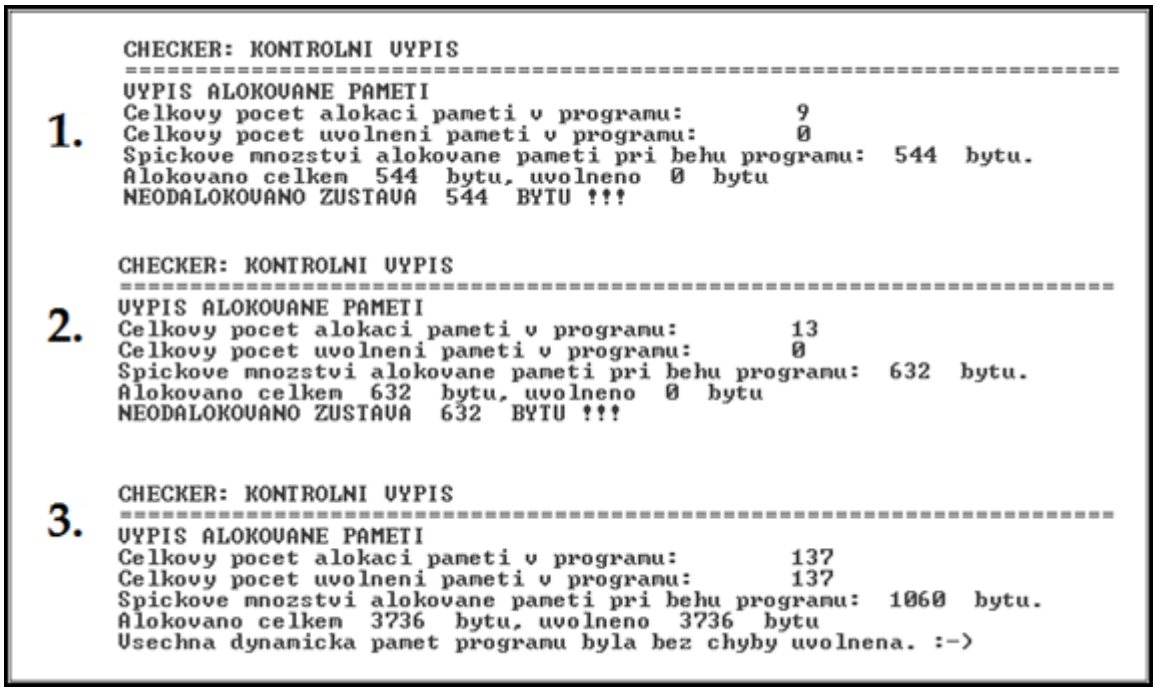
2.) Při druhém výpisu existují současně dvourozměrné pole a matice a celkový počet zabraných bytů je 1112. Což znamená, že řídká matice v tomto případě zabírá 1112-544 = 564 bytů. V tomto případě se jedná o matici bez většího počtu nul, proto není možná žádná paměťová úspora dat.

3.) Třetí výpis je volán na konci programu. Před jeho průběhem byl počítán determinant matice osmého řádu, která v sobě neobsahovala téměř žádné nuly. Jak je vidět v průběhu počítání determinantu, bylo nealokované veliké množství dynamické paměti. Jedná se o subdeterminanty, které jsou počítány v determinantu. Došlo k více než devadesáti tisícům alokacím a v průběhu výpočtu determinantu byly celkem zabrány téměř 3 MB paměti. Avšak maximální počet zabrané paměti v jednom okamžiku je 4,36 KB. Což vzhledem k tomu, že 1,11 KB zabírají samy objekty, nepředstavuje výraznější nárůst paměti.

I přesto je poměrně znepokojující velikost paměti, která byla celkově zabrána. Také výpočetní rychlost je pomalá, protože vytváření a zanikání takového množství objektů vyžadují výpočetní čas. Pro svou obhajobu bych chtěl zdůraznit, že algoritmus je navržen pro výpočet determinantu **řídké** matice, kde tento problém do jisté míry zaniká.

Nyní upravím červeně vyznačenou část z předchozího zdrojového kódu, ve které se pole plní hodnotami tak, aby matice byla diagonální čili řídká. Zbytek kódu zůstává stejný.

```
    for(i=0;i<X;i++)            // naplneni matice cisel
    {
        for(j=0;j<X;j++)
        {
            if(i==j)
                pole[i][j]=i+j;
            else
                pole[i][j]=0;
        }
    }
```



Obrázek 23 Výpisy stavu paměti při výpočtu determinantu řídké matice pomocí knihovny `check.h`

Popis paměťových výpisů:

1.) První výpis zůstává stejný. Pole si opět vymezilo 544 bytů.
2.) Ze druhého výpisu je již vidět paměťová úspora, kterou jsem docílil uložením pole do řídké matice. Ta zabírá 632 - 544 = 88 bytů paměti.
3.) Ve třetím výpisu je vidět, že počet alokované paměti oproti předchozímu případu značně klesl. Pro výpočet determinantu bylo navíc vytvořeno v jednom okamžiku až 428 bytů, což je ve srovnání s 88 byty, které matice zabírá hodně. To je

způsobeno tím, subdeterminanty nejsou testovány na nejvýhodnější formát uložení (kvůli časové úspoře), ale jsou přímo ukládány do CSR formátu. Tento formát je sice úsporný, ale zabírá více paměťového prostoru než diagonální tvar. Proto při výpočtu determinantu s diagonální maticí není výpočet tak výhodný jako je samo uložení.

9.1.1 Paměťová náročnost metody počítající determinant matice

Nejnáročnější metodou na paměť je funkce `determinant()`. Tato funkce je napsána tak, aby ji bylo možné použít pro výpočet determinantu **řídké** matice velkého řádu. V tabulce 2 je zaznamenán přesný počet alokované paměti v průběhu programu počítající determinant matic různých velikostí.

Tabulka 2 Počty využívané paměti při počítání determinantu matice o různých rozměrech a různém obsahu

Rozměr Matice	Nulová matice	Pouze nenulové prvky	Nenulové prvky v diagonále
[počet prvků]	[B]	[B]	[B]
1	80	90	90
4	124	172	148
9	184	292	196
16	868	2452	1128
25	1132	15452	2488
36	1420	97756	4440
49	1732	692648	7060
64	2068	5554293	10404
81	2428	50012368	14936
100	2812		20008
225	5092		62556
400	7972		145556
900	15532		467920
1600	25492		1118220
2500	37872		2185940
10000	135652		16541644
40000	511252		129877040

V grafu 1 je znázorněna závislost využité paměti v průběhu programu na rozměrech matice. Viz Příloha 1. Z tabulky a grafu je vidět, že výpočty determinantu matice jsou značně méně paměťové náročné a zároveň rychlejší, když je matice naplněna řídkým počtem nenulových prvků. Skok v grafu je způsoben změnou výpočetního algoritmu matice pro čtvrtý řád a vyšší. Do třetího řádu je používáno Saurussovo pravidlo od vyšších řádů pak kofaktorová metoda. Kofaktorová metoda je náročnější na spotřebu paměti.

9.1.1 Paměťová náročnost třídy pro uložení řídké matice.

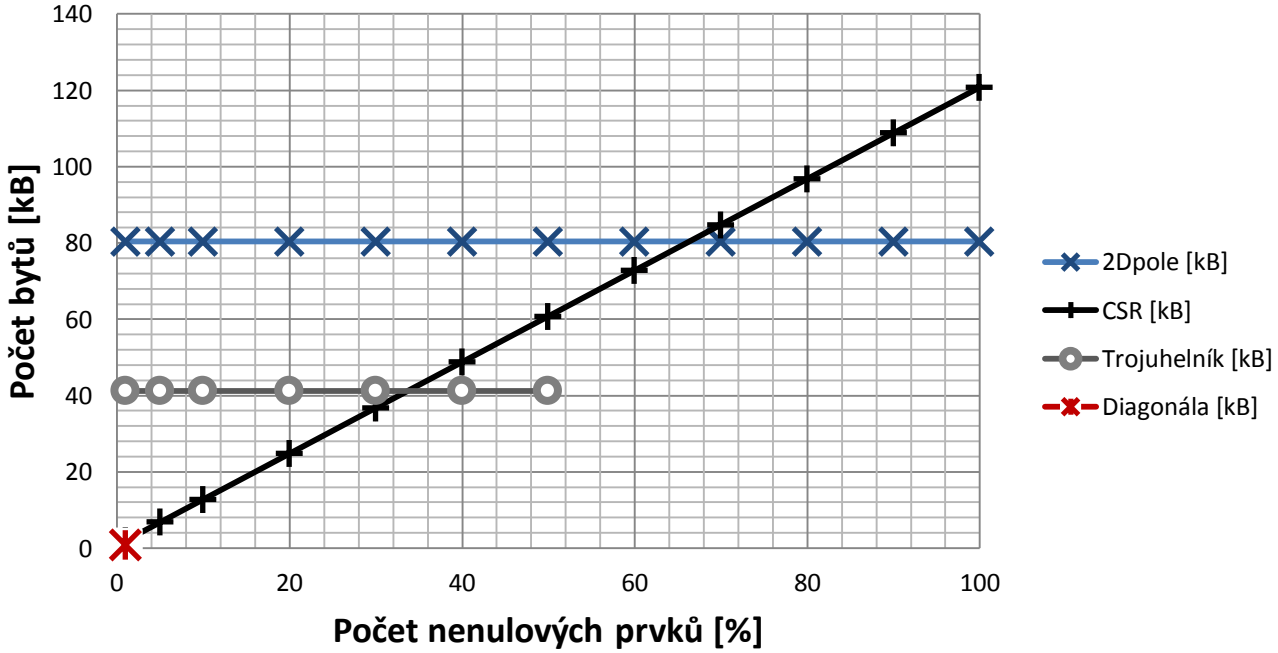
Co se týče konkrétního počtu zabraných bytů jednotlivými formáty řídké matice, dodržel jsem teoretické předpoklady z kapitoly 3.

Skutečné hodnoty jsou zaznamenány viz Tabulka 3 a vyneseny do grafu 2**.** Jedná se o matici o rozměrech 100 x 100. V grafu 2 jsou vyneseny počty zabrané paměti v jednotlivých formátech. Oproti teoretickým předpokladům jsou ve třídě ještě statické proměnné (rozměry atd…) a také 3 vektory pro CSR formát. Tyto vektory i přes svou nulovou délku zabírají místo v paměti. Počet zabraných bytů paměti přesně počítá metoda `přepočet()`.

Tabulka 3 Závislosti zabrané paměti jednotlivými formáty v závislosti na hustotě nenulových prvků

nenulových prvků	2Dpole	CSR	Trojúhelník	Diagonála
[%]	[kB]	[kB]	[kB]	[kB]
1	80,4	2	41,2	0,8
5	80,4	6,8	41,2	
10	80,4	12,8	41,2	
20	80,4	24,8	41,2	
30	80,4	36,8	41,2	
40	80,4	48,8	41,2	
50	80,4	60,8	41,2	
60	80,4	72,8		
70	80,4	84,8		
80	80,4	96,8		
90	80,4	108,8		
100	80,4	120,8		

Graf 2 Závislost počtu zabrané paměti v jednotlivých formátech na počtu nenulových prvků v matici.



9.1.2 Paměťová náročnost třídy pro uložení přesného čísla

Třída pro uchování přesného čísla s pohyblivou řádovou desetinnou čárkou je paměťově náročná
v závislosti na požadovaném počtu platných cifer. Se zvyšující se přesností se zvětšuje i počet zabrané paměti viz Tabulka 4.

Tabulka 4 Počty zabraných bytů v závislosti na přesnosti čísla u `double` a `ENumber`

Počet desetinných míst	`double`	uložení mantisy čísla v:		
		`short` **2B**	`Int` **4B**	`long long int` **8B**
[B]	**[B]**	**[B]**	**[B]**	**[B]**
2	8	7	9	13
5	8	9	9	13
10	8	11	13	13
15	8	13	13	13
20	nelze	15	17	21
30	nelze	21	21	21

40	nelze	25	25	29
50	nelze	31	29	29
70	nelze	39	37	37
100	nelze	55	53	53
1000	nelze	505	453	453

9.2 Výpočetní náročnost maticové knihovny

Pro zjišťování výpočetní náročnosti byl použit nástroj ze sady Intel Parallel Studio 2011 a to Intel Paraller Amplifier.

Bohužel jsem neměl možnost tímto nástrojem porovnat žádnou jinou knihovnu zabývající se mou problematikou. Porovnal jsem tedy pouze časy strávené zpracováním jednotlivých funkcí. Nejdéle trvaly streamovací operátory a výpočet determinantu. Proto jsem se zaměřil hlavně na úpravu algoritmu determinantu, který popisuji v kapitolách 5 a 8. Porovnám-li však výpočet determinantu se systémem MATLAB, který zvládá taktéž maticové výpočty, tak jsem se bohužel na úroveň MATLABu nedostal. Výpočetní náročnost je samozřejmě ovlivněna velikostmi matic, ale hlavně jejich obsahy. Pokud matice obsahují mnoho nul, jsou výpočty rychlejší a to hlavně protože bude možné použít jednodušší formát, se kterým lze rychleji pracovat.

9.3 Výpočetní náročnost knihovny pro přesná čísla

Téměř veškerá výpočetní doba je strávena při získávání cifer z mantisy. Mantisa čísla je ukazatel na celočíselné datové proměnné. Z těchto proměnných se jednotlivé cifry vyčíslují celočíselným dělením a zbytku po dělení. A dělení není zrovna nejjednodušší operace ve srovnání například s bitovými posuny. Bitovými posuny by bylo možné jednotlivé cifry například v případě kódování cifer v BDC kódu. Kde jedna cifra zabírá striktně 4 bity.

Je to do jisté míry chybný způsob implementace knihovny. Pokud bych navrhoval knihovnu znovu, uchýlil bych se k ukládání cifer pomocí BCD kódu. Na druhou stranu se mi povedlo ušetřit paměťový prostor ve velikosti mantisy, protože ukládám čísla úsporněji.

10 PŘÍKLADY ŘÍDKÝCH MATIC S PŘESNÝMI ČÍSLY

Příklady pro naplnění řídké matice `Csm` prvky typu `ENumber`.

Jsou zde uvedeny dva základní způsoby, kterými je nejvhodnější matici naplnit.
Tyto příklady platí pro naplnění matice obecně jakýmkoliv datovým typem, stačí pouze zaměnit šablonu. V příloze na CD je testovací program, který demonstruje většinu ze základních operací.

Příklad 1:

```
#include "sparse_matrix.h"
#include "exact_number.h"

int main()
{
	int i,j;
	Csm<ENumber> matice;			// vytvoreni ridke matice
	matice.SetRadky(5);			// nastaveni rozmeru matice
	matice.SetSloupce(5);
	for(i=0;i<5;i++)			// naplneni matice
	{
		for(j=0;j<5;j++)
		{
			matice.Set( (rand()%100)/10.0 , i , j);
		}
	}
	cout << matice;				// vypis matice

	return 0;
}
```

V tomto příkladu nejprve nastavím rozměry matice a posléze matici naplním.

V tomto případě naplňuji matici náhodnými ENumber čísly od 0 – 10 vytvořenými konverzním konstruktorem z `double`.

V tomto případě je matice uložena v CSR formátu, protože kontrola zapsaných prvků se automaticky provádí při zápisu každého padesátého prvku. Přičemž 50 prvků nebylo zapsáno.

Pokud trváte na nejúspornějším formátu ihned, lze změnu formátu zajistit pomocí metody `ZjistiFormat3()`.

```
Matice je ulozena ve formatu CSR

4,1     6,7     3,4     0       6,9
2,4     7,8     5,8     6,2     6,4
0,5     4,5     8,1     2,7     6,1
9,1     9,5     4,2     2,7     3,6
9,1     0,4     0,2     5,3     9,11
```

Obrázek 24 Výsledné zobrazení matice

```
matice.konverze(matice.ZjistiFormat3());
```

Matice bude po provedení tohoto zápisu převedena do formátu `2DPole`, který je pro tuto matici výhodnější než CSR formát.

Příklad 2:

```
#include "exact_number.h"
#include "check.h"

int main()
{
	int i,j;
	ENumber **pole=nullptr;				// alokace matice cisel

	pole = new ENumber*[5];				// vytvoreni matice cisel
	for(i=0;i<5;i++)
		pole[i] = new ENumber[5];

	for(i=0;i<5;i++)				// naplneni matice cisel
	{
		for(j=0;j<5;j++)
		{
			pole[i][j]=i+j;
		}
	}
	Csm<ENumber> matice(pole,5,5);			// vytvoreni ridke matice s cisly

	for(i=0;i<5;i++)				// odstraneni pole
		delete[] pole[i];
	delete pole;
	cout << matice;					// vypis matice cisel
}
```

V tomto příkladu nejprve vytvářím dvourozměrné pole prvků typu `ENumber`.

Posléze bude vytvořena matice pomocí konverzního konstruktoru z dvourozměrného pole a hodnoty budou vypsány do konzole.

Výsledek je rovnou uložen do nejúspornějšího formátu, ale je nutné si uvědomit, že zároveň existuje i předem vytvořené dvourozměrné pole, které však bude odstraněno co nejdříve.

11 ZÁVĚR

Nastudoval jsem problematiku uložení řídkých matic v paměti počítače. Principů pro ukládání a práci s řídkými maticemi je velké množství. Některé jsem vybral a objasnil v kapitole 3. Každý z principů uložení řídkých matic se stává výhodným za jiných podmínek. Některé principy uložení matic jsou určeny na specificky orientované matice. Ty se pro uložení řídké matice nemohou použít ve všech případech, ale pokud je to možné, přinášejí velké výhody. Na základě těchto informací jsem si zvolil vhodné formáty ukládání řídkých matic a implementoval je do knihovny.

Vytvořil jsem knihovnu realizující třídu pro úsporné uložení řídkých matic v paměti počítače. Knihovna je psána jako šablona, což znamená, že obsahem matice může být jakýkoliv datový typ, který zvládá tytéž matematické operace, které budou prováděny s maticemi. Knihovna umožňuje ukládat matice do čtyř různých formátů: řídkého formátu CSR, diagonálního tvaru, trojúhelníkového tvaru nebo do dvourozměrného pole jako hustou matici. Knihovna nabízí základní matematické maticové operace jako je výpočet determinantu, transponované matice a inverzní matice. Po dokončení matematické operace matice jako celku je výsledek vždy uložen ve formátu, který zabere v paměti nejméně místa. Knihovna je rozdělena do souborů `sparse_matrix.h`, kde jsou napsány hlavičky funkcí, a do více souborů s příponou `inl`, kde jsou ukládány jednotlivé algoritmy funkcí. Těchto souborů je celkem devět. Jednotlivé soubory obsahují vždy metody nebo operátory obdobného typu. Knihovna je rozdělena do těchto souborů pro lepší orientaci programátora v knihovně. Metody jsou totiž rozsáhlé a toto systematické rozdělení umožňuje lepší orientaci v kódu. Pro knihovnu jsem vygeneroval dokumentaci funkci třídy `Csm<T>`, což je datový typ pro uložení řídké matice, pomocí programu Doxygen, viz Příloha 2. Dokumentace obsahuje seznamy všech metod a operátorů a jejich stručný popis.

Nastudoval jsem principy uložení čísel s plovoucí desetinnou čárkou v paměti počítače, kde je číslo rozděleno na znaménko, exponent a mantisu. Na každou z těchto částí je vymezen určitý datový prostor v závislosti na použitém formátu standardu IEEE 754. S ohledem na tyto informace jsem navrhnul princip uložení přesného čísla. Ponechal jsem rozdělení na znaménko, mantisu a exponent, ale jejich význam není totožný se standardem IEEE 754, viz kapitola 7.1.

Byly vytvořeny základy knihovny pro uchování přesného datového typu, která je určena pro uložení čísel s plovoucí desetinnou čárkou. Třída obsahuje dvě stěžejní složky, exponent a mantisu. Exponent je celočíselná proměnná a mantisa je pole celočíselných datových typů, do kterých jsou po blocích ukládány části mantisy. Pole obsahující mantisu čísla se vytvoří vždy tak velké, aby do něj bylo možné ukládat všechny platné cifry. Mantisa se tedy zvětšuje v závislosti na počtu cifer. Minimální velikost mantisy je velikost jednoho bloku, což je jeden z celočíselných datových typů. Různé datové typy pro uložení mantisy jsou implementovány tak, aby byla možná optimalizace proměnných pro daný typ procesoru. Seznam a stručný popis knihovny je v dokumentaci třídy `ENumber`, viz Příloha 3. Pro zlepšení a zjednodušení výpočtů knihovny by se do knihovny mohly ještě přidat pomocné algoritmy indikující periodická čísla. Zápis by pak mohl být značně kratší a čísla by byla ještě přesnější. Nyní je tento problém řešen zvýšenou přesností čísla. Dalším

vylepšením by mohlo být přepracování algoritmů rozdílu a podílu tak, aby nebylo třeba používat na uložení mantisy znaménkové celočíselné datové typy.

Obě dvě knihovny jsou vzájemně kompatibilní. Po integraci knihoven je možné provádět operace maticových výpočtů s přesnými desetinnými čísly. Pro použití knihovny pro přesná čísla je třeba připojit soubory `exact_number.h` a `exact_number.cpp` k projektu. Pro použití knihovny pro počítání s řídkými maticemi je třeba připojit soubory `sparse_matrix.h` a všechny soubory s příponou `inl` k projektu. Následně je třeba připojit hlavičkové soubory pomocí `#include`. Po připojení obou knihoven lze provádět vzájemné operace.

Jednotlivé funkce knihoven byly otestovány v obdobných nástrojích pro práci s maticemi a čísly. V MATLABu byla otestována správnost maticových výpočtů. Maticové výpočty byly otestovány s maticemi různých velikostí a obsahů do rozměrů 10x10. Kontrola správnosti výsledku byla prováděna prvek po prvku. Knihovna pro uchování přesného čísla byla otestována taktéž pomocí MATLABu, MS EXCEL a rovněž pomocí vědecké kalkulačky v OS Windows. Čísla byla otestována s přesností na 32 desetinných míst, což dovoloval nástroj vědecké kalkulačky. Pomocí MS EXCEL pak s přesností na 14 desetinných míst.

Pro ověření jednotlivých metod a operátorů knihovny byl vytvořen soubor `testovaci_program_matice.cpp`, který je na CD a demonstruje funkčnost jednotlivých metod a operátorů.

Existujícím řešením řídkých matic se zabývá i MATLAB. MATLAB používá pro ukládání řídkých matic souřadnicový formát, kde jednotlivé prvky jsou ukládány po sloupcích. Ve srovnání s MATLABem má knihovna pro uchování řídkých matic různé druhy formátů. MATLAB nabízí veliké množství předdefinovaných funkcí pro práci s maticemi. Knihovna nabízí metod méně. Kdybych měl pokračovat ve vylepšování knihovny, dopsal bych další metody, jako například vytvoření jednotkové matice, diagonální matice a operace matic pracující s jednotlivými sloupci nebo řádky. Dalším existujícím řešením je LAPACK, který zase nabízí veliké spektrum různých datových struktur, do kterých lze matice ukládat. Dalšími možnými rozšířeními knihovny pro matematické operace s řídkými maticemi by mohlo být přidání dalších řídkých formátů, do kterých by se mohly matice ukládat.

12 POUŽITÁ LITERATURA

[1] ECKEL, Bruce. *Myslíme v jazyku C++*. 1. vyd. Praha: Grada Publishing, 2006, 608 s. ISBN 80-247-1015-3.

[2] VIRIUS, Miroslav. *Jazyky C a C++*. 1. vyd. Praha: Grada, 2006, 518 s. Knihovna programátora (Grada). ISBN 80-247-1494-9.

[3] CÍHA, T.: *Řídké matice a jejich použití v numerické matematice*, MU, Brno, 2009

[4] Sparse matrix. In: *Wikipedia*: *the free encyclopedia* [online].]. Wikimedia Foundation, poslední aktualizace 2.12.2011 [cit. 2011-12-12]. Dostupné z: http://cs.wikipedia.org/wiki/Sparse_matrix

[5] Floating point. In: *Wikipedia*: *the free encyclopedia* [online]. Wikimedia Foundation, poslední aktualizace 18. 5. 2012 [cit. 2012-05-24]. Dostupné z: http://en.wikipedia.org/wiki/Floating_point

[6] LAPACK . In: *Wikipedia*: *the free encyclopedia* [online]. Wikimedia Foundation, poslední aktualizace 18. 3. 2012 [cit. 2012-4-4]. Dostupné z: http://cs.wikipedia.org/wiki/LAPACK

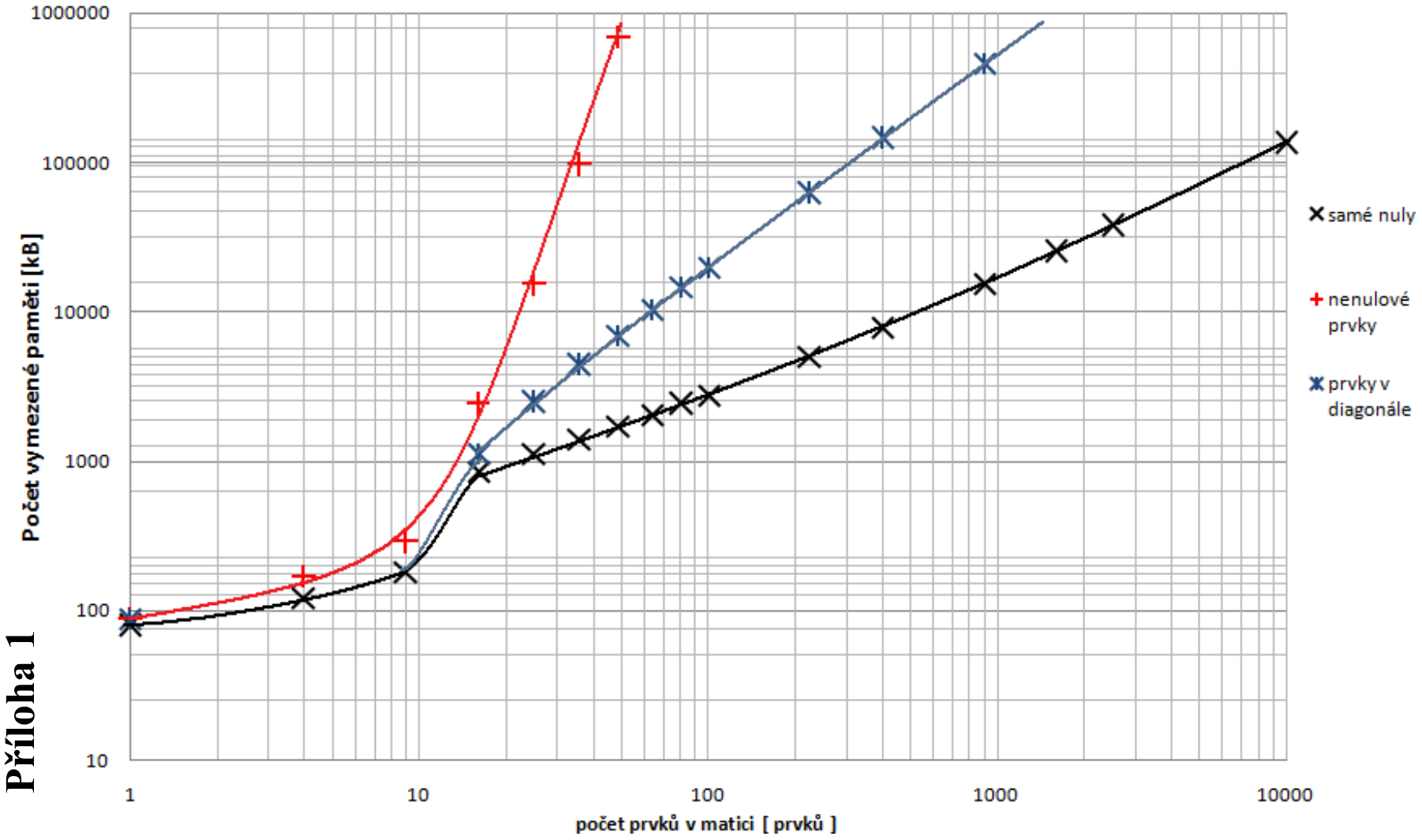
[7] Adjungovaná matice. In: *Wikipedia*: *the free encyclopedia* [online]. Wikimedia Foundation, poslední aktualizace 18.3.2012 [cit. 2012-4-4]. Dostupné z http://cs.wikipedia.org/wiki/Adjungovan%C3%A1_matice

[8] Determinant matice. In: *Wikipedia*: *the free encyclopedia* [online]. Wikimedia Foundation, poslední aktualizace 18.3.2012 [cit. 2012-4-4] Dostupné z http://cs.wikipedia.org/wiki/Determinant

13 SEZNAM PŘÍLOH

Příloha 1

Graf 1 Závislost vymezené paměti v průběhu programu na rozměrech maticepro matici naplněnou pouze nulami a matici naplněnounenulovými prvky.



Příloha 2 Dokumentace šablony třídy Csm<T>

Veřejné metody

Csm (T **matice, unsigned int rozmerx, unsigned int rozmery)
Csm (const **Csm**< T > &matice)
Csm (int format, unsigned int radku, unsigned int sloupcu)
std::vector< unsigned int > **GetCSRRadky** () const
std::vector< unsigned int > **GetCSRSloupce** () const
std::vector< T > **GetCSRHodnoty** () const
T ** **GetD2Hodnoty** () const
T ** **GetTrojpoleHodnoty** () const
T * **GetDiagHodnoty** () const
unsigned int **GetRadky** () const
unsigned int **GetSloupce** () const
size_t **Size** ()
TAktivni **GetFormat** () const
void **SetRadky** (unsigned int value)
void **SetSloupce** (unsigned int value)
T **Get** (unsigned int radek, unsigned int sloupec) const
void **Setter** (const T &hodnota, unsigned int radek, unsigned int sloupec, int pom)
void **Set** (const T &hodnota, unsigned int radek, unsigned int sloupec)
bool **konverze** (int format)
T ** **VytvorAPrevedNaPole** ()
T **determinant** (int rad)
Csm< T > **VytvorInverzni** ()
Csm< T > **Transponuj** ()
void **prepocet** ()
int **zjistiFormat** (**Csm**< T > &vysledek, const **Csm**< T > &matice, int operace) const
int **zjistiFormat2** (**Csm**< T > &vysledek, T hodnota, int operace) const
int **ZjistiFormat3** ()
Csm< T > **operator+** (T **matice)
Csm< T > **operator+** (const **Csm**< T > &matice)
Csm< T > & **operator+=** (**Csm**< T > &matice)
Csm< T > & **operator-=** (const **Csm**< T > &matice)
Csm< T > **operator-** (const **Csm**< T > &matice)
Csm< T > **operator*** (const **Csm**< T > &matice)
Csm< T > & **operator*=** (**Csm**< T > &matice)
Csm< T > & **operator=** (const **Csm**< T > &matice)
bool **operator==** (const **Csm**< T > &matice) const
bool **operator!=** (const **Csm**< T > &matice) const
Csm< T > **operator*** (T hodnota)
Csm< T > **operator+** (T hodnota)
Csm< T > **operator-** (T hodnota)

```
Csm< T > & operator-= (T hodnota)
Csm< T > & operator+= (T hodnota)
Csm< T > & operator*= (T hodnota)
Csm< T > operator/ (Csm< T > &matice)
Csm< T > operator/ (T hodnota)
```

Friends

```
Csm< T > operator* (T hodnota, const Csm< T > &matice)
Csm< T > operator- (T hodnota, const Csm< T > &matice)
Csm< T > operator+ (T hodnota, const Csm< T > &matice)
std::ostream & operator (std::ostream &oBuffer, const Csm< T > &matice)
```

template<typename T> class Csm< T >

Dokumentace k metodám

template<typename T > T Csm< T >::determinant (int *rad*)

Metoda vypocte determinant matice laplasovym rozvojem.

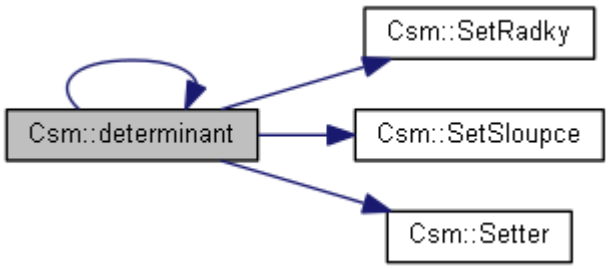
Parametry:

matice	Matice ze ktere chci zjistit jeji determinant.
rad	Rad matice.

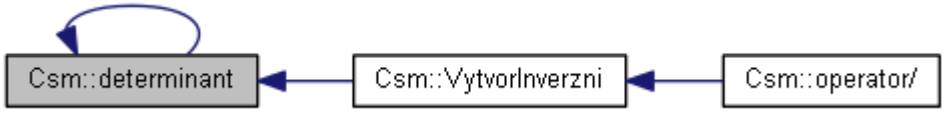
Návratová hodnota:

Vysledny determinant.

Tato funkce volá...



Tuto funkci volají...



template<typename T > T Csm< T >::Get (unsigned int *radek*, unsigned int *sloupec*) const

Metoda zjisti hodnotu prvku na pozadovanych souradnicich.

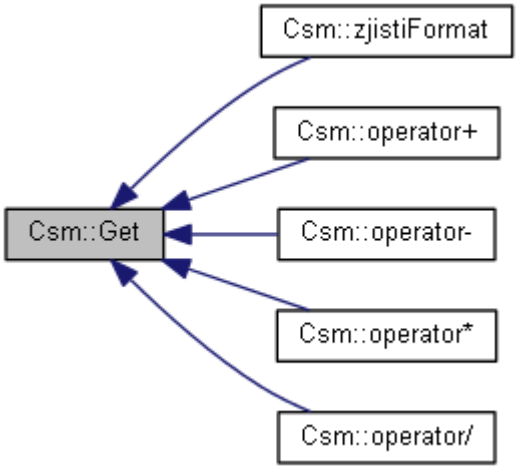
Parametry:

radek	index radku.
sloupec	index sloupce.

Návratová hodnota:

Vysledna hodnota.

Tuto funkci volají...



template<typename T > std::vector< T > Csm< T >::GetCSRHodnoty () const

Metoda vraci hodnoty ulozene ve specificke forme.

Návratová hodnota:

Hodnoty pro CSRformat ulozene ve vektoru.

Tuto funkci volají...

template<typename T > std::vector< unsigned int > Csm< T >::GetCSRRadky () const

Metoda vraci informace o radcich ulozene ve specificke forme.

Návratová hodnota:

Kodovani radku pro CSRformat ulozene ve vektoru.

Tuto funkci volají...

template<typename T > std::vector< unsigned int > Csm< T >::GetCSRSloupce () const

Metoda vraci informace o sloupcu ulozene ve specificke forme.

Návratová hodnota:

Kodovani sloupcu pro CSRformat ulozene ve vektoru.

Tuto funkci volají...

template<typename T > T ** Csm< T >::GetD2Hodnoty () const

Metoda vraci hodnoty ulozene ve specificke forme.

Návratová hodnota:

Hodnoty ulozene ve dvourozmernem poli.

Tuto funkci volají...

template<typename T > T * Csm< T >::GetDiagHodnoty () const

Metoda vraci hodnoty ulozene ve specificke forme.

Návratová hodnota:

Hodnoty ulozene ve vektoru reprezentujici diagonalu.

Tuto funkci volají...

template<typename T> TAktivni Csm< T >::GetFormat () const `[inline]`

Metoda zjisti aktualni format.

Návratová hodnota:

udaj o formatu ve kterem je ulozena matice 1-4 diag,pole,trojpole,CSR

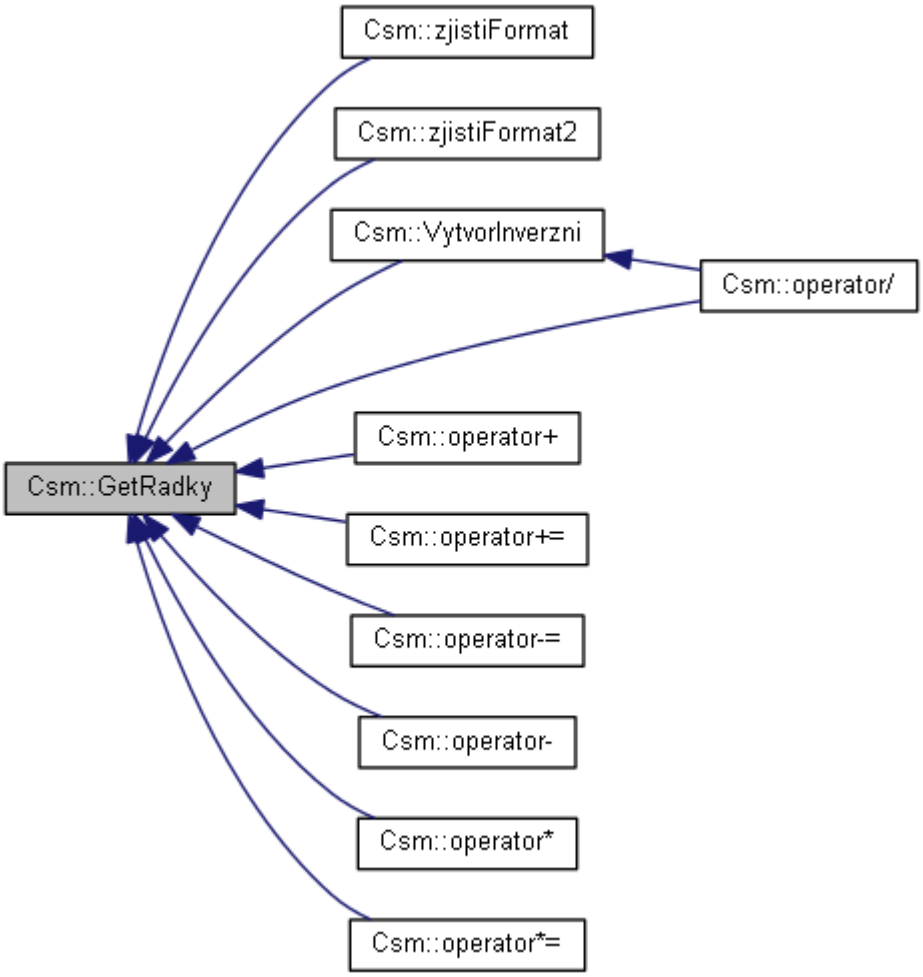
template<typename T > unsigned int Csm< T >::GetRadky () const

*Metoda zjisti pocet radku.

Návratová hodnota:

udaj o poctu radku.

Tuto funkci volají...



template<typename T > unsigned int Csm< T >::GetSloupce () const

Metoda zjisti pocet cloupcu.

Návratová hodnota:

udaj o poctu sloupcu.

Tuto funkci volají...

Obdobně jako u GetRadky()

template<typename T > T ** Csm< T >::GetTrojpoleHodnoty () const

Metoda vraci hodnoty ulozene ve specificke forme.

Návratová hodnota:

Hodnoty ulozene v trojuhelnikovem poli.

Tuto funkci volají...

template<typename T > bool Csm< T >::konverze (int *format*)

Metoda, ktera zkonvertuje matici do pozadovaneho formatu. POZOR! pokud prevadim matici napr. z 2Dpole do diagonalniho tvaru prijdu o nenulove prvky mimo diagonalu. Pokud chcete prevest matici na diagonalni/trojuhelnikovy tvar, je treba, aby matice byla ctvercova.

Parametry:

format	Pozadovany vysledny format.

Návratová hodnota:

True=povedlo se, false=nepovedlo se.

template<typename T > bool Csm< T >::operator!= (const Csm< T > & *matice*) const

Operator porovnani dvou matic. Porovnava i adresy vnitrnich poli, pokud se vrati true, potom A!=A. Jedna se o kontrolu jestli se nejedna o tutez matici.

Parametry:

Zdrojova	matice.

Návratová hodnota:

Vysledek true=Matice neni totozna, false=matice je totozna.

template<typename T > Csm< T > Csm< T >::operator* (const Csm< T > & *matice*)

Operator soucinu dvou ridkych matic.

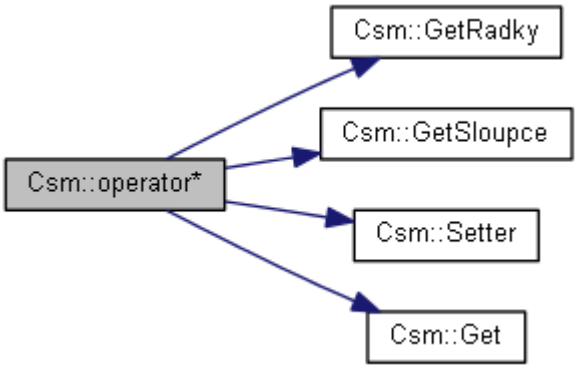
Parametry:

Matice	urcena k soucinu.

Návratová hodnota:

Vysledek soucinu, temp. hodnota.

Tato funkce volá...



template<typename T > Csm< T > Csm< T >::operator* (T *hodnota*)

Operator soucinu ridke matice s promennou.

Parametry:

Hodnota	urcena k soucinu.

Návratová hodnota:

Vysledek soucinu. Temp. hodnota.

Tato funkce volá...

template<typename T > Csm< T > & Csm< T >::operator*= (Csm< T > & *matice*)

Operator soucinu dvou ridkych matic.

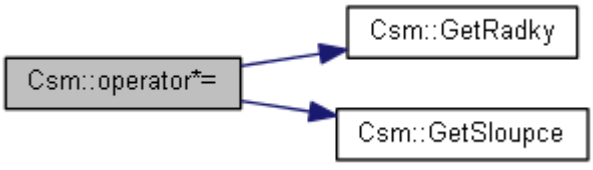
Parametry:

Matice	urcena k soucinu.

Návratová hodnota:

Vysledek soucinu.

Tato funkce volá...



template<typename T > Csm< T > & Csm< T >::operator*= (T *hodnota*)

Operator soucinu ridke matice s promennou.

Parametry:

Hodnota	urcena k soucinu.

Návratová hodnota:

Vysledek soucinu.

template<typename T > Csm< T > Csm< T >::operator+ (T ** *matice*)

Operator souctu dvou ridkych matic. Predpoklad ze pole je dostatecne velke.

Parametry:

Matice	urcena k secteni.

Návratová hodnota:

Vysledek souctu, temp. hodnota.

Tato funkce volá...

template<typename T > Csm< T > Csm< T >::operator+ (const Csm< T > & *matice*)

Operator souctu dvou ridkych matic.

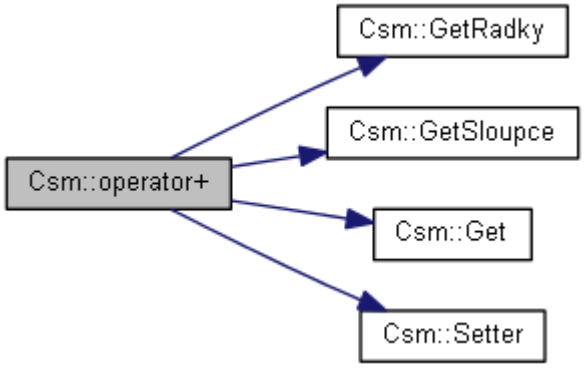
Parametry:

Matice	urcena k secteni.

Návratová hodnota:

Vysledek souctu, temp. hodnota.

Tato funkce volá...



template<typename T > Csm< T > Csm< T >::operator+ (T *hodnota*)

Operator souctu ridke matice s promennou.

Parametry:

Hodnota	urcena k souctu.

Návratová hodnota:

Vysledek souctu. Temp. hodnota.

Tato funkce volá...

template<typename T > Csm< T > & Csm< T >::operator+= (Csm< T > & *matice*)

Operator souctu dvou ridkych matic.

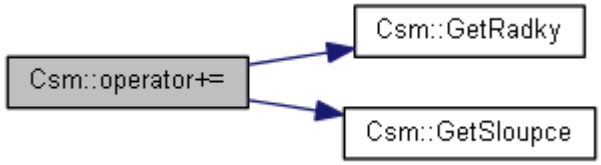
Parametry:

Matice	urcena k secteni.

Návratová hodnota:

Vysledek souctu.

Tato funkce volá...



template<typename T > Csm< T > & Csm< T >::operator+= (T *hodnota*)

Operator souctu ridke matice s promennou.

Parametry:

Hodnota	urcena k souctu.

Návratová hodnota:

Vysledek souctu.

template<typename T > Csm< T > Csm< T >::operator- (const Csm< T > & *matice*)

Operator rozdilu dvou ridkych matic.

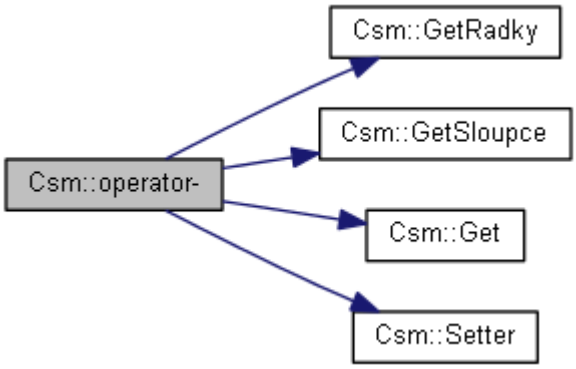
Parametry:

Matice	urcena k rozdilu.

Návratová hodnota:

Vysledek rozdilu, temp. hodnota.

Tato funkce volá...



template<typename T > Csm< T > Csm< T >::operator- (T *hodnota*)

Operator rozdilu ridke matice s promennou.

Parametry:

Hodnota	urcena k rozdilu.

Návratová hodnota:

Vysledek rozdilu, temp. hodnota.

Tato funkce volá...

template<typename T > Csm< T > & Csm< T >::operator-= (const Csm< T > & *matice*)

Operator rozdilu dvou ridkych matic.

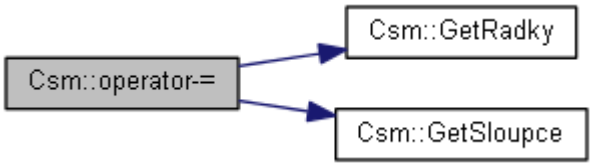
Parametry:

Matice	urcena k rozdilu.

Návratová hodnota:

Vysledek rozdilu.

Tato funkce volá...



template<typename T > Csm< T > & Csm< T >::operator-= (T *hodnota*)

Operator rozdilu ridke matice s promennou.

Parametry:

Hodnota	urcena k rozdilu.

Návratová hodnota:

Vysledek rozdilu.

template<typename T > Csm< T > Csm< T >::operator/ (Csm< T > & *matice*)

Operator podilu dvou ridkych matic.

Parametry:

Matice	urcena k podilu.

Návratová hodnota:

Vysledek podilu, temp. hodnota.

template<typename T > Csm< T > Csm< T >::operator/ (T *hodnota*)

Operator podilu promenne s ridkou matici.

Parametry:

hodnota	Hodnota urcena k podilu.

Návratová hodnota:

Vysledek podilu.

Tato funkce volá...

template<typename T > Csm< T > & Csm< T >::operator= (const Csm< T > & *matice*)

Operator rovna se. Pro Zkopirovani matice.

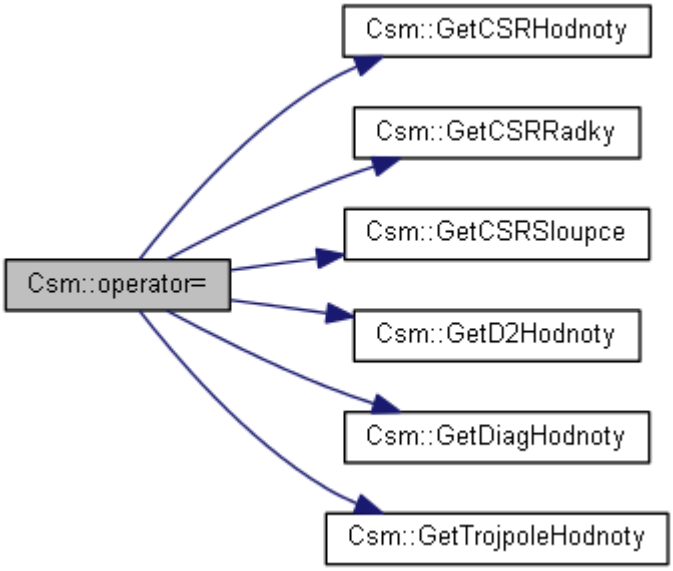
Parametry:

matice	Zdrojova matice.

Návratová hodnota:

Vysledek.

Tato funkce volá...



template<typename T > bool Csm< T >::operator== (const Csm< T > & *matice*) const

Operator porovnani dvou matic. Porovnava i adresy vnitrnich poli, pokud se vrati true, potom A==A. Jedna se o kontrolu jestli se jedna o tutez matici.

Parametry:

matice	Zdrojova matice.

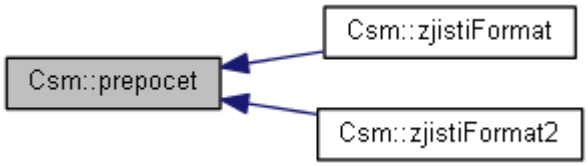
Návratová hodnota:

Vysledek true=Matice je totozna, false=matice neni totozna.

template<typename T > void Csm< T >::prepocet ()

Zjisti jak je vyhodne ulozit matici do jednotlivych formatu bez ohledu na to, jestli je to mozne. Zaroveň nastavuje hodnoty vyhodnosti do matice.

Tuto funkci volají...



template<typename T > void Csm< T >::Set (const T & *hodnota*, unsigned int *radek*, unsigned int *sloupec*)

Metoda ulozi hodnotu do pozadovaneho formatu a pozadovane souradnice.

Parametry:

hodnota	Hodnota, musi byt stejneho datoveho typu jako typ matice.
radek	souradnice radku na ktery chci zapsat hodnotu. Indexovano od nuly.
sloupec	souradnice sloupce na ktery chci zapsat hodnotu. Indexovano od nuly.
pom	informace o formatu do ktereho ukladam, musim pouzit zpravny format, pokud chci format zmenit musim pouzit metodu konverze.

Tuto funkci volají...

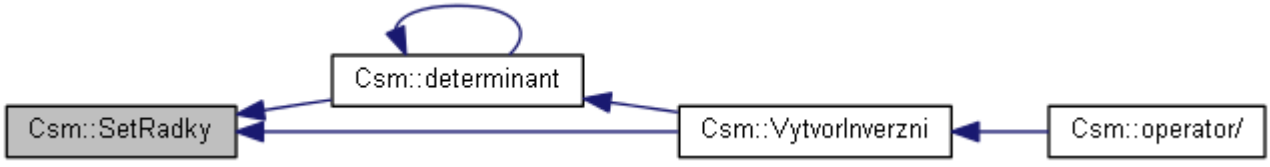
template<typename T > void Csm< T >::SetRadky (unsigned int *value*)

Metoda nastavi pocet radku matice. Pokud nebyl doposud nastaven rozmer je mozne jej nastavit bez ohledu na format. *Pokud jiz byl rozmer nastaven a chcete jej zmenit, tak se vase matice prevede do CSR formatu a zmeni se jeji rozmery.

Parametry:

value	Nova hodnota poctu radku.

Tuto funkci volají...



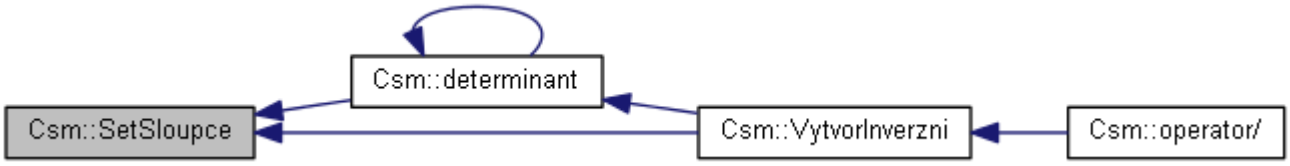
template<typename T > void Csm< T >::SetSloupce (unsigned int *value*)

Metoda nastavi pocet sloupcu matice. Pokud nebyl doposud nastaven rozmer je mozne jej nastavit bez ohledu na format. *Pokud jiz byl rozmer nastaven a chcete jej zmenit, tak se vase matice prevede do CSR formatu a zmeni se jeji rozmery.

Parametry:

value	Nova hodnota poctu sloupcu.

Tuto funkci volají...



template<typename T > void Csm< T >::Setter (const T & *hodnota*, unsigned int *radek*, unsigned int *sloupec*, int *pom*)

Metoda ulozi hodnotu do pozadovaneho formatu a pozadovane souradnice.

Parametry:

hodnota	Hodnota, musi byt stejneho datoveho typu jako typ matice.
radek	souradnice radku na ktery chci zapsat hodnotu. Indexovano od nuly.
sloupec	souradnice sloupce na ktery chci zapsat hodnotu. Indexovano od nuly.
pom	informace o formatu do ktereho ukladam, musim pouzit zpravny format, pokud chci format zmenit musim pouzit metodu konverze.

template<typename T > Csm< T > Csm< T >::Transponuj ()

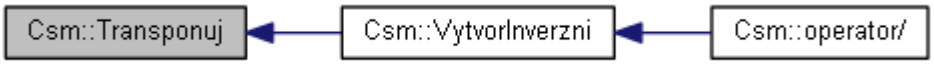
Metoda transponuje matici.

Návratová hodnota:

Transponovana matice.

Tato funkce volá...

Tuto funkci volají...



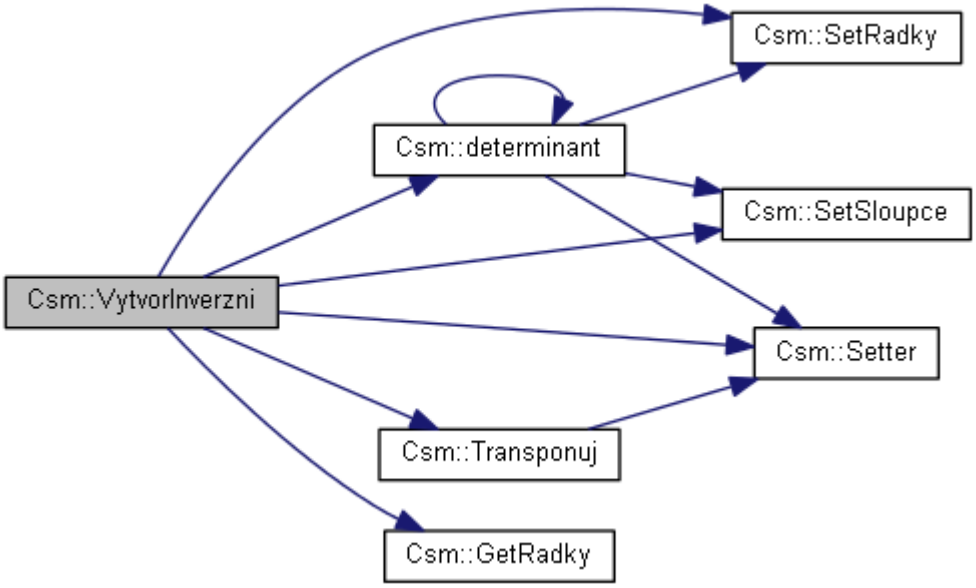
template<typename T > Csm< T > Csm< T >::VytvorInverzni ()

Metoda vypocte inverzni matici ze soucasne matice.

Návratová hodnota:

Inverzni matice.

Tato funkce volá...



Tuto funkci volají...

template<typename T > int Csm< T >::zjistiFormat (Csm< T > & *vysledek*, const Csm< T > & *matice*, int *operace*) const

Zjisti do jakeho formatu je nejvyhodnejsi ulozit vysledek operace dvou matic vysledek je zmenen a nastavi se do nej informace o rozmerech poctu prvku a informace o diagonalach.

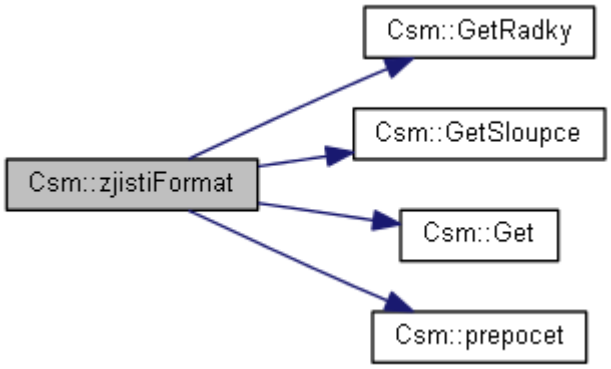
Parametry:

vysledek	vysledna matice do ktere se nastavi informace o rozmerech, diagonalach, poctu prvku.
hodnota	matice se kterou se bude provadet pozadovana operace.
operace	pozadovana matematicka operace, 1=+, 2=-, 3=*.

Návratová hodnota:

format ktery je nejvhodnejsi pro ulozeni matice.

Tato funkce volá...



template<typename T > int Csm< T >::zjistiFormat2 (Csm< T > & *vysledek*, T *hodnota*, int *operace*) const

Zjisti do jakeho formatu je nejvyhodnejsi ulozit vysledek operace matice a cisla, vysledek je zmenen a nastavi se do nej informace o rozmerech poctu prvku a informace o diagonalach.

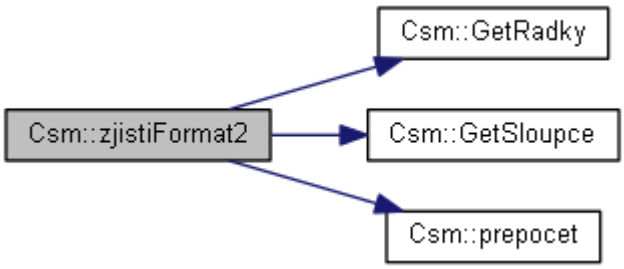
Parametry:

vysledek	vysledna matice do ktere se nastavi informace o rozmerech, diagonalach, poctu prvku.
hodnota	hodnota se kterou se bude provadet pozadovana operace.
operace	pozadovana matematicka operace, 0=zadna operace,vysetreni pouze samotne matice, 1=+, 2=-, 3=*.

Návratová hodnota:

format, ktery je nejvhodnejsi pro ulozeni matice. 1-4 Diag, 2Dpole, Trojuhelnik, CSR.

Tato funkce volá...



template<typename T > int Csm< T >::ZjistiFormat3 ()

Zjisti do jakeho formatu je nejvyhodnejsi ulozit vysledek soucasne matice. Zmeni informace o diagonalach.

Návratová hodnota:

format, ktery je nejvhodnejsi pro ulozeni matice. 1-4: Diag, 2Dpole, Trojuhelnik, CSR.

Dokumentace k friends

template<typename T> Csm<T> operator* (T *hodnota*, const Csm< T > & *matice*) [friend]

Operator soucinu promenne s ridkou matici.

Parametry:

hodnota	Hodnota urcena k soucinu.
matice	Matice urcena k soucinu.

Návratová hodnota:

Vysledek soucinu.

template<typename T> Csm<T> operator+ (T *hodnota*, const Csm< T > & *matice*) [friend]

Operator souctu promenne s ridkou matici.

Parametry:

hodnota	Hodnota urcena k souctu.
matice	Matice urcena k souctu.

Návratová hodnota:

Vysledek souctu.

template<typename T> Csm<T> operator- (T *hodnota*, const Csm< T > & *matice*) [friend]

Operator rozdilu promenne s ridkou matici.

Parametry:

hodnota	Hodnota urcena k rozdilu.
matice	Matice urcena k rozdilu.

Návratová hodnota:

Vysledek rozdilu.

Dokumentace pro tuto třídu byla generována z následujících souborů:

- sparse_matrix.h
- kontroly_formatu.inl.h
- konverze.inl.h, metody_setry.inl.h
- matematicke_metody.inl.h
- metody_getry.inl.h, operatory.inl.h

Příloha 3 Dokumentace třídy ENumber

Některé z vývojových diagramů volaných a volajících metod zde není uveřejněna, protože díky jejich rozsáhlosti nebyli přehledné. Jsou k dispozici na CD ve webové dokumentaci.

Veřejné metody

```
ENumber ()
ENumber (const char *cislo)
ENumber (const ENumber &cislo)
ENumber (const float cislo)
ENumber (const double cislo)
ENumber (const int cislo)
~ENumber ()
int GetPocetCifer () const
MantiseType * GetMantisu () const
void SetMantisu (MantiseType *amantisa)
MantiseType GetCifruRadu (int rad) const
ENumber abs () const
void OdstranNuly ()
ENumber inverzni () const
void VytvorZRetezce (const char *retezec)
void VytvorZInt (const int cislo)
void VytvorZDouble (const double cislo)
void VytvorMantisu (const int cifer_max)
ENumber umocni (const int mocnitel) const
ENumber operator+ (const ENumber &cislo) const
ENumber operator+ () const
ENumber operator- (const ENumber &cislo) const
ENumber operator- () const
ENumber operator* (const ENumber &cislo) const
ENumber operator* (const int cislo) const
ENumber operator/ (const ENumber &cislo) const
ENumber & operator= (const ENumber &cislo)
ENumber & operator+= (const ENumber &cislo)
ENumber & operator-= (const ENumber &cislo)
ENumber & operator*= (const ENumber &cislo)
ENumber & operator/= (const ENumber &cislo)
ENumber & operator= (const int cislo)
ENumber & operator= (const double cislo)
ENumber & operator= (const char *retezec)
bool operator== (const ENumber &cislo) const
bool operator!= (const ENumber &cislo) const
bool operator<= (const ENumber &cislo) const
bool operator>= (const ENumber &cislo) const
bool operator< (const ENumber &cislo) const
bool operator> (const ENumber &cislo) const
bool operator> (const double cislo) const
bool operator< (const double cislo) const
```

```
bool operator<= (const double cislo) const
bool operator>= (const double cislo) const
bool operator== (const double cislo) const
bool operator!= (const double cislo) const
bool operator> (const int cislo) const
bool operator< (const int cislo) const
bool operator<= (const int cislo) const
bool operator>= (const int cislo) const
bool operator== (const int cislo) const
bool operator!= (const int cislo) const
```

Veřejné atributy

- MantiseType * **mantisa**

Friends

- std::ostream & **operator<<** (std::ostream &oBuffer, const **ENumber** &cislo)
- std::istream & **operator>>** (std::istream &iBuffer, **ENumber** &cislo)

Dokumentace konstruktoru a destruktoru

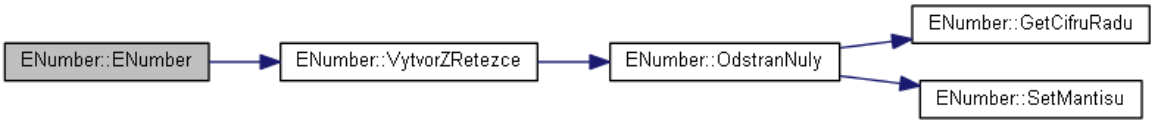
ENumber::ENumber ()

Implicitni konstruktor

ENumber::ENumber (const char * *retezec*)

Konverzni konstruktor z retezce

Tato funkce volá...



ENumber::ENumber (const ENumber & *cislo*)

Kopykonstruktor

Parametry:

retezec	zdrojovy retezec.

Tato funkce volá...

ENumber::ENumber (const float *cislo*)

Konverzni konstruktor z float

Parametry:

cislo	zdrojovy cislo.

ENumber::ENumber (const double *cislo*)

Konverzni konstruktor z double

Parametry:

cislo	zdrojovy cislo.

ENumber::ENumber (const int *cislo*)

Konverzni konstruktor z int

Parametry:

cislo	zdrojovy cislo.

ENumber::~ENumber ()

Destruktor

Dokumentace k metodám

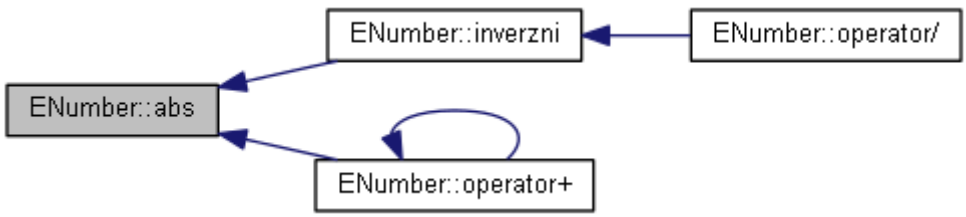
ENumber ENumber::abs () const

Metoda vraci absolutni hodnotu cisla.

Návratová hodnota:

Absolutni hodnota cisla.

Tuto funkci volají...



ENumber::MantiseType ENumber::GetCifruRadu (int *rad*) const

Metoda vrati cifru ktere ma cislo na danem radu.

Parametry:

rad	Vaha cisla, ktere chci vycist.

Tuto funkci volají...

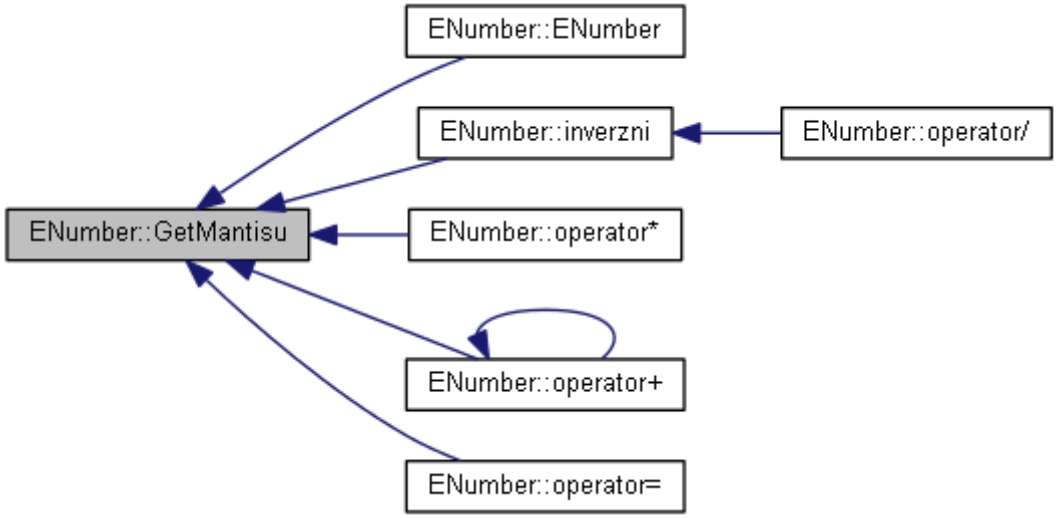
ENumber::MantiseType * ENumber::GetMantisu () const

Metoda vrati celou mantisu cisla.

Návratová hodnota:

mantisa cisla.

Tuto funkci volají...



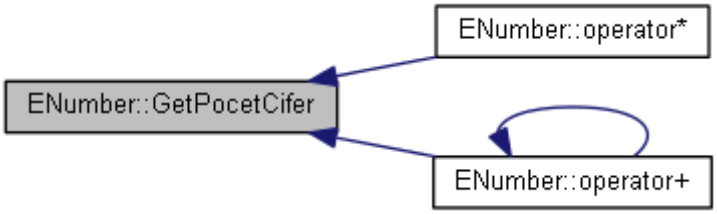
int ENumber::GetPocetCifer () const

Metoda zjisti pocet cifer v cisle.

Návratová hodnota:

Vysledek.

Tuto funkci volají...



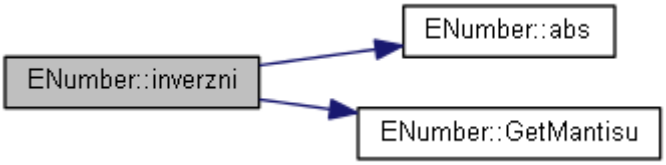
ENumber ENumber::inverzni () const

Metoda vytvori cislo 1/cislo. Nemeni cislo, ale vraci inverzni hodnotu.

Návratová hodnota:

inverzni hodnota.

Tato funkce volá...

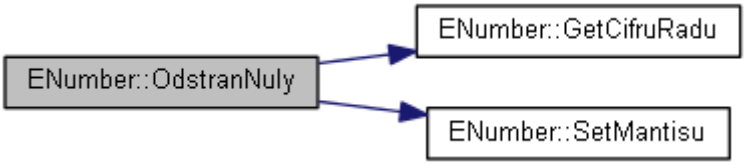


Tuto funkci volají...

void ENumber::OdstranNuly ()

Metoda odstrani vsechny nuly, ktere jsou na nejnizsich radech. Metoda zmeni(zmensi) cislo.

Tato funkce volá...



bool ENumber::operator!= (const ENumber & *cislo*) const

Operator nerovna se.

Parametry:

cislo	Porovnavany objekt.

Návratová hodnota:

pravda/nepravda

bool ENumber::operator!= (const double *cislo*) const

Operator nestejne velke nez double.

Parametry:

cislo	Porovnavany objekt.

Návratová hodnota:

pravda/nepravda

bool ENumber::operator!= (const int *cislo*) const

Operator nestejne velke nez int.

Parametry:

cislo	Porovnavany objekt.

Návratová hodnota:

pravda/nepravda

ENumber ENumber::operator* (const ENumber & *cislo*) const

Operator soucinu dvou cisel.

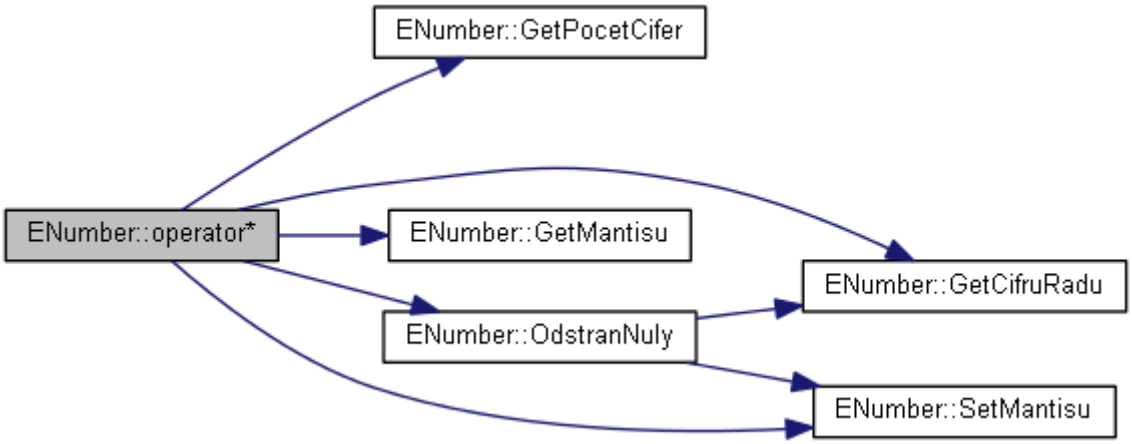
Parametry:

cislo	nasobitel

Návratová hodnota:

vysledek soucinu.

Tato funkce volá...



ENumber ENumber::operator* (const int *cislo*) const

Operator soucinu cisla a intu.

Parametry:

cislo	*nasobitel.*

Návratová hodnota:

vysledek soucinu.

ENumber & ENumber::operator*= (const ENumber & *cislo*)

Operator krat rovna se.

Parametry:

cislo	nasobici prvek.

Návratová hodnota:

odkaz na vysledek operace.

ENumber ENumber::operator+ (const ENumber & *cislo*) const

Operator souctu dvou cisel.

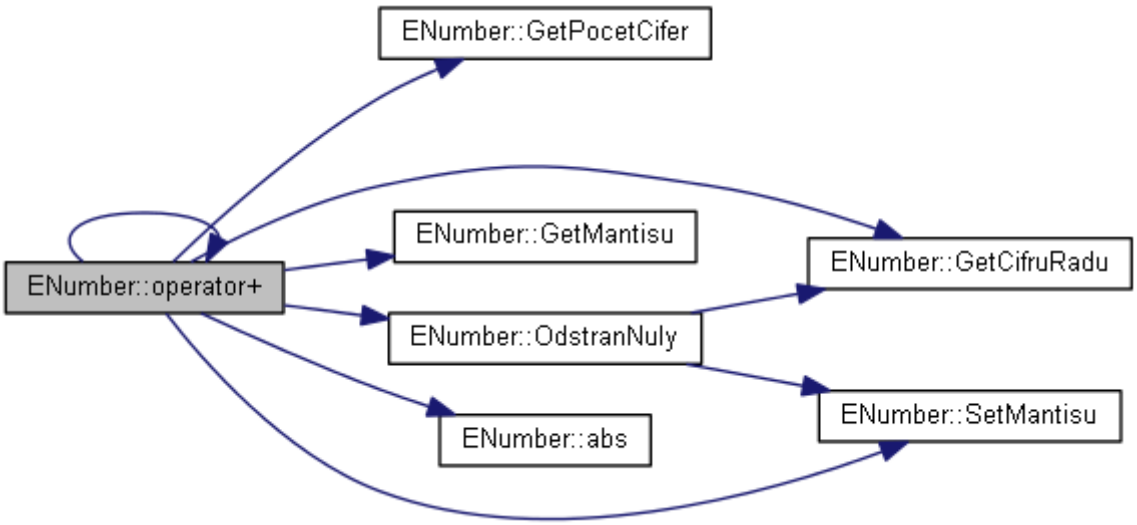
Parametry:

cislo	scitanec

Návratová hodnota:

vysledek souctu.

Tato funkce volá...



Tuto funkci volají...

ENumber & ENumber::operator+= (const ENumber & *cislo*)

Operator plus rovna se.

Parametry:

cislo	scitany prvek.

Návratová hodnota:

odkaz na vysledek operace.

ENumber ENumber::operator- (const ENumber & *cislo*) const

Operator rozdilu.

Parametry:

cislo	Odecitany prvek.

Návratová hodnota:

vysledek rozdilu.

ENumber ENumber::operator- () const

Operator vraci znamenkovou negaci.

Návratová hodnota:

opacnou cislo s opacnym znamenkem.

ENumber & ENumber::operator-= (const ENumber & *cislo*)

Operator minus rovna se.

Parametry:

cislo	odecitany prvek.

Návratová hodnota:

odkaz na vysledek operace.

ENumber ENumber::operator/ (const ENumber & *cislo*) const

Operator podilu dvou cisel. Vyuzivanasobeni inverzni hodnotou.

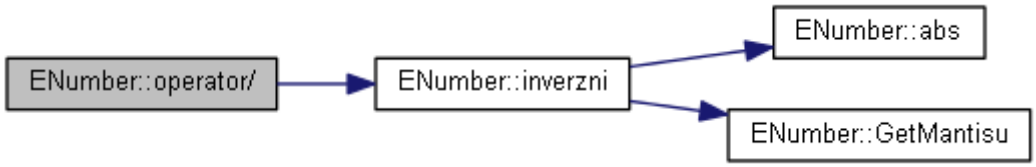
Parametry:

cislo	*delitel.*

Návratová hodnota:

vysledek podilu.

Tato funkce volá...



ENumber & ENumber::operator/= (const ENumber & *cislo*)

Operator lomeno rovna se.

Parametry:

cislo	delenec.

Návratová hodnota:

odkaz na vysledek operace.

bool ENumber::operator< (const ENumber & *cislo*) const

Operator mensi nez.

Parametry:

porovnavany	objekt.

Návratová hodnota:

pravda/nepravda

Tato funkce volá...

bool ENumber::operator< (const double *cislo*) const

Operator mensi nez double.

Parametry:

cislo	porovnavany objekt.

Návratová hodnota:

pravda/nepravda

bool ENumber::operator< (const int *cislo*) const

Operator mensi nez int.

Parametry:

cislo	porovnavany objekt.

Návratová hodnota:

pravda/nepravda

bool ENumber::operator<= (const ENumber & *cislo*) const

Operator mensi rovno.

Parametry:

cislo	Porovnavany objekt.

Návratová hodnota:

pravda/nepravda

bool ENumber::operator<= (const double *cislo*) const

Operator mensi nebo rovno nez double.

Parametry:

cislo	Porovnavany objekt.

Návratová hodnota:

pravda/nepravda

bool ENumber::operator<= (const int *cislo*) const

Operator mensi nebo rovno nez int.

Parametry:

cislo	Porovnavany objekt.

Návratová hodnota:

pravda/nepravda

ENumber & ENumber::operator= (const ENumber & *cislo*)

Operator rovna se.

Parametry:

cislo	Dosazovany prvek.

Návratová hodnota:

dosazeny prvek.

Tato funkce volá...

ENumber & ENumber::operator= (const int *cislo*)

Operator rovna se z intu.

Parametry:

cislo	Dosazovany prvek.

Návratová hodnota:

vysledek rozdilu.

ENumber & ENumber::operator= (const double *cislo*)

Operator rovna se z double.

Parametry:

cislo	Dosazovany prvek.

Návratová hodnota:

vysledek rozdilu.

ENumber & ENumber::operator= (const char * *retezec*)

Operator rovna se z intu.

Parametry:

cislo	Dosazovany prvek.

Návratová hodnota:

vysledek rozdilu.

bool ENumber::operator== (const ENumber & *cislo*) const

Operator porovnani dvou cisel.

Parametry:

cislo	Pozadovany objekt.

Návratová hodnota:

pravda/nepravda

Tato funkce volá...

bool ENumber::operator== (const double *cislo*) const

Operator shodne s double?

Parametry:

cislo	porovnavany objekt.

Návratová hodnota:

pravda/nepravda

bool ENumber::operator== (const int *cislo*) const

Operator shodne s int?

Parametry:

cislo	porovnavany objekt.

Návratová hodnota:

pravda/nepravda

bool ENumber::operator> (const ENumber & *cislo*) const

Operator vetsi nez.

Parametry:

cislo	porovnavany objekt.

Návratová hodnota:

pravda/nepravda

bool ENumber::operator> (const double *cislo*) const

Operator vetsi nez double.

Parametry:

cislo	porovnavany objekt.

Návratová hodnota:

pravda/nepravda

bool ENumber::operator> (const int *cislo*) const

Operator vetsi nez int.

Parametry:

cislo	porovnavany objekt.

Návratová hodnota:

pravda/nepravda

bool ENumber::operator>= (const ENumber & *cislo*) const

Operator vetsi rovno.

Parametry:

cislo	porovnavany objekt.

Návratová hodnota:

pravda/nepravda

bool ENumber::operator>= (const double *cislo*) const

Operator vetsi nebo rovno nez double.

Parametry:

cislo	porovnavany objekt.

Návratová hodnota:

pravda/nepravda

bool ENumber::operator>= (const int *cislo*) const

Operator vetsi nebo rovno nez int.

Parametry:

cislo	porovnavany objekt.

Návratová hodnota:

pravda/nepravda

void ENumber::SetMantisu (MantiseType * *amantisa*)

Metoda nastavi mantisu cisla pomoci jine mantisy. Tato metoda se muze pouzit na odstraneni prebytecne naalokovanych bunek mantisy tak, ze se dopredu zmensi pocet platnych cifer.

Parametry:

amantisa	Zdrojova mantisa.

ENumber ENumber::umocni (const int *mocnitel*) const

Metoda pro umocneni cisla.

Parametry:

mocnitel	pouze kladna mocnina.

Návratová hodnota:

umocnene cislo.

void ENumber::VytvorZDouble (const double *cislo*)

Metoda vytvori cislo z intu. metoda vyuziva metodu **VytvorZRetezce()**

Parametry:

cislo	Zdrojove cislo.

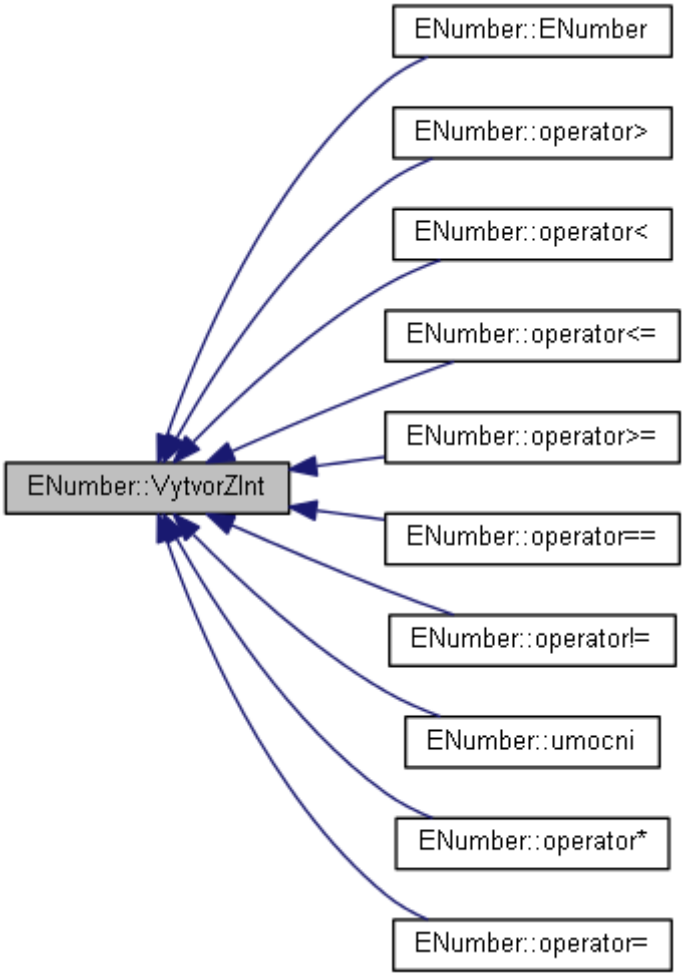
void ENumber::VytvorZInt (const int *cislo*)

Metoda vytvori cislo z intu. metoda vyuziva metodu **VytvorZRetezce()**

Parametry:

cislo	Zdrojove cislo.

Tuto funkci volají...



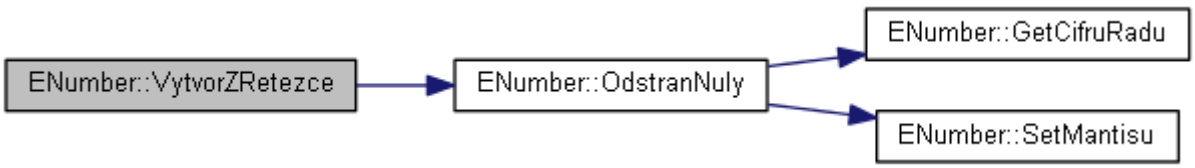
void ENumber::VytvorZRetezce (const char * *retezec*)

Metoda vytvori cislo z char retezce cislic.

Parametry:

retezec	Zdrojovy retezec.

Tato funkce volá...



Dokumentace k friends

std::ostream& operator<< (std::ostream & *oBuffer*, const ENumber & *cislo*) [friend]

Operator << pro vypis do console.

Parametry:

oBuffer	Stream urceny k vypisu.
cislo	Zdrojove cislo urcene k vypisu.

Návratová hodnota:

oBuffer.

std::ostream& operator>> (std::istream & *iBuffer*, ENumber & *cislo*) [friend]

Operator >> pro nacteni z console.

Parametry:

iBuffer	Stream vstupnich znaku
cislo	Zdrojove cislo urcene k naplneni.

Návratová hodnota:

oBuffer.

Dokumentace pro tuto třídu byla generována z následujících souborů:

- exact_number.h
- exact_number.cpp