Univerzita Pardubice

Fakulta elektrotechniky a informatiky

Řešení soustav lineárních rovnic na signálovém procesoru

Bc. Štěpán Novák

Diplomová práce
2010

zadání 1

Prohlášení autora

Tuto práci jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně.

V Třebíči dne 20. 8. 2010

Bc. Štěpán Novák

Abstrakt

Práce se zabývá řešením soustav lineárních rovnic v oboru komplexních čísel na signálovém procesoru. Obsahuje popis vlastností procesoru TigerSHARC ADSP-TS201S od firmy Analog Devices. V další části jsou rozebrány metody pro řešení soustav lineárních rovnic a odzkoušeny v programu MATLAB. Poté jsou provedeny analýzy, které určují vlastnosti jednotlivých metod s ohledem na potřeby signálového procesoru. V poslední části je popsána implementace metody založené na QR rozkladu pomocí modifikovaného Gram-Schmidtova algoritmu do signálového procesoru a jsou porovnány dosažené výsledky.

Klíčová slova

Řešení soustava lineárních rovnic, ADSP-TS201S, DSP, signálový procesor, QR rozklad, Gaussova eliminace, Gram-Schmidtův algoritmus, Householderovo zrcadlení, SVD, LU rozklad, run-time library, VisualDSP++

Abstract

This work deals with solving systems of linear equations in the field of complex numbers for digital signal processor. It includes description of properties of Tiger-SHARC Processor ADSP-TS201S from company Analog Devices. The next part deals with methods for solving linear equations and tested in MATLAB. Afterward the analysis is accomplished to determine the characteristics of the different methods with regard to the needs of DSP. The last part describes the implementation of the method based on QR decomposition using the modified Gram-Schmidt algorithm to the signal processor and obtained results are discussed.

Keywords

Solving linear equations, ADSP-TS201S DSP, signal processor, QR decomposition, Gaussian elimination, Gram-Schmidt algorithm, Householder reflection, SVD, LU decomposition, run-time library, VisualDSP++

Poděkování

Chci poděkovat všem lidem, kteří přispěli k vypracování diplomové práce, především vedoucímu projektu, panu Ing. Martinovi Hájkovi za pomoc při řešení dílčích úkolů, za čas, po který se mi věnoval v rámci odborných konzultací a za poskytnutí studijních materiálů.

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Řešení soustav lineárních rovnic patří k nejstarším matematickým problémům a v dnešní době má stále velké využití. Problematika jejich výpočtů je velice široká s mnoha možnostmi přístupu k jejich řešení. Tím se zabývá i tato práce, a to v oboru komplexních čísel. Soustavy rovnic mají být realizovány na signálovém procesoru ADSP-TS201S pro právě vyvíjený radar na měření výšky sněhu. Procesor, který nalézá nejčastější uplatnění ve zpracování číslicově reprezentovaných signálů, je velice komplikovaný a s mnoha specifickými vlastnostmi.

Hlavním požadavkem diplomové práce je tedy správný výběr jedné matematické metody pro řešení soustav lineárních rovnic a její efektivní implementaci do procesoru. Důraz je kladen na vysokou míru optimalizace, rychlost a stabilitu algoritmu pro výše zmíněný procesor.

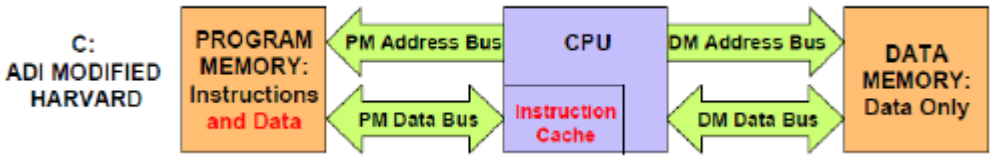
V první kapitole je popis procesoru ADSP-TS201S a rozbor jeho základních vlastností. Druhá obsahuje obecný popis problematiky řešení soustav lineárních rovnic, rozbor možných výskytů budoucích problémů a způsoby řešení soustav s komplexními čísly. Třetí se zabývá rozborem konkrétních metod řešení s praktickými ukázkami algoritmů v programu MATLAB. Ve čtvrté kapitole je provedena analýza metod řešících soustavy lineárních rovnic s ohledem na potřeby signálového procesoru. V poslední kapitole je popsána samotná implementace do DSP a zhodnocení dosažených výsledků.

1 Popis signálového procesoru

1.1 Úvod do architektury DSP

Digitální signálový procesor (DSP) je procesor navržený především ke zpracování digitálně reprezentovaných signálů. Velice často je nutné signál zpracovávat v reálném čase, z čehož plynou vysoké nároky na datovou propustnost a rychlosti výpočtů. Z těchto důvodů je uzpůsobena vnitřní struktura DSP oproti jiným typům procesorů. Nejčastější využití nalézá ve zpracování obrazu a zvuku, kde se vyskytují matematické operace jako například filtrace signálu, FFT[1], korelace, konvoluce a práce s maticemi. Operace lze velice efektivně řešit, protože vlastnosti procesoru jsou pro tyto účely přizpůsobeny.

Většina DSP je postavena na modifikované Harvardské architektuře (Obr. 1), kde je sice paměť programu a dat oddělena, ale je využito společného adresového prostoru. Díky společnému adresnímu prostoru se paměť jeví jako jeden celek, ale ve skutečnosti bývá většinou rozdělena na několik částí. Pokud procesor obsahuje více sběrnic, umožňuje to do jednotlivých částí přistupovat současně a tím urychlit přenosy dat.



Obr. 1: Modifikovaná Harvardská architektura [14]

Vysokých výkonů signálových procesorů je dosahováno především hardwarovou podporou některých operací. Většina dnešních DSP obsahuje více jak jeden výpočetní blok. Výpočetním blokem se rozumí část procesoru, která obsahuje jednotky na provádění matematických operací, například aritmeticko-logickou jednotku, násobička a další jednotky dle konkrétního typu DSP.

Aby bylo možné využít výše zmíněných vlastností (více sběrnic napojených na různé části paměti a více výpočetních jednotek), musí procesor umět vykonávat několik činností v jednom instrukčním cyklu. Toho je dosaženo tím, že procesor dokáže zpracovat velmi dlouhé instrukční slovo (VLIW), které může obsahovat až několik instruk-

1 Rychlá Furrierova transformace

cí. Díky VLIW je možné v jednom instrukčním cyklu například vynásobit dvě čísla a výsledek sečíst s akumulátorem. Během sčítaní a násobení je možné ještě načítat a ukládat data (z paměti do systémových registrů) pro následující, respektive předchozí instrukční krok. [14] [15]

1.2 Procesor Analog Devices ADSP-TS201S

Jedná se o 128 bitový signálový procesor se statickou superskalární architekturou[2] častěji nazývanou VLIW (Very Long Instruction Word) (Kapitola 1.2.2) , který obsahuje 24 Mb interní DRAM paměti (Kapitola 1.2.5) a dokáže pracovat s taktovací frekvencí 600 MHz (doba jednoho instrukčního cyklu trvá 1,67 ns). Má dva na sobě nezávislé výpočetní bloky obsahující několik matematických výpočetních jednotek (ALU, CLU, MUL, SHIFTER), které jsou popsány v následující kapitole (Kapitola 1.2.6). Procesor dokáže pracovat s čísly s pevnou a plovoucí řádovou čárkou (Kapitola 1.2.1). Pro komunikaci mezi jednotlivými bloky procesoru jsou k dispozici čtyři dvojice sběrnic (32 b adresová, 128 datová), které je možné využívat nezávisle a ve stejný časový okamžik (Kapitola 1.2.4).

1.2.1 Datové typy

Jak bylo napsáno výše, procesor umožňuje práci se dvěma aritmetikami, a to čísly s pevnou řádovou čárkou a plovoucí řádovou čárkou. Pro pevnou řádovou čárku je možné volit mezi různou délkou datového slova 8, 16, 32, 64 bitů. Při práci s 8, 16 bitovým datovým typem je možné procesor využít tak, aby bylo dosaženo většího výpočetního výkonu (Kapitola 1.2.6). Pro plovoucí řádovou čárku je možné použít 32 b typ single dle normy IEEE-754 (pro naše potřeby budeme používat označení float) a rozšířený 40 b datový typ, který má o 8 b větší mantisu, ale v paměti zabírá celých 64 b. Datový typ 64 b (double dle IEEE-754) není hardwarově podporován a je ho možné použít pouze za pomoci softwarové emulace. Pokud je potřeba použít datový typ double, výkon procesoru se rapidně sníží a byl by takřka nepoužitelný pro výpočet velkých soustav lineárních rovnic. Pro sčítaní v tomto formátu platí rovnice (1), analogicky pro násobení (2).

2 Hlavní rozdíl mezi VLIW a superskalární architekturou je, že u super skalární jsou řízeny paralelní instrukce procesorem a u technologie VLIW určuje paralelizaci instrukcí programátor, popřípadě kompilátor.

$$A = M * z^E$$
$$E1 \geq E2 \text{ -> } M1zE1 \pm M2zE2 = (M1 \pm M2zE2–E1)\ zE1 \tag{1}$$

$$M_1 z^{E1} * M_2 z^{E2} = (M_1 * M_2)\ z^{E1+\ E2} \tag{2}$$

Jednoduchým experimentem v jazyce C bylo zjištěno, že sčítání trvá zhruba 60 hodinových cyklů a násobení přibližně polovinu. Což je oproti formátu float, kde obě instrukce trvají jeden instrukční cyklus obrovský nárůst. Proto je podmínkou, aby pro výpočet soustav lineárních rovnic dostačovala přesnost dána 32 bitovým formátem (s jednoduchou přesností). Po zhodnocení výše uvedených faktorů se jeví jako nejlepší volba pro řešení soustav lineárních rovnic datový typ float (32 b s plovoucí řádovou čárkou). [16]

1.2.2 VLIW – velmi dlouhé instrukční slovo

Procesor dokáže zpracovávat až čtyři 32-bitové instrukce v jednom instrukčním cyklu, které je možné v programu řetězit tak, aby vznikla jedna 128 b. To však má některá omezení a pravidla, která je potřeba striktně dodržovat. V jednom instrukčním cyklu není možné dvakrát zapisovat data do stejného registru. Pokud je načítáno z paměti do registrů (nebo ukládáno do paměti), tak dále již není možné s tímto registrem pracovat. V jedné instrukci také není možné mít podmínku a na základě jejího řešení volat podprogram. Následující jednoduchý příklad v jazyce symbolických adres demonstruje, jak je možné využívat VLIW. Jednotlivé instrukce jsou odděleny středníkem, dvěma středníky je ukončeno 128 b instrukční slovo.

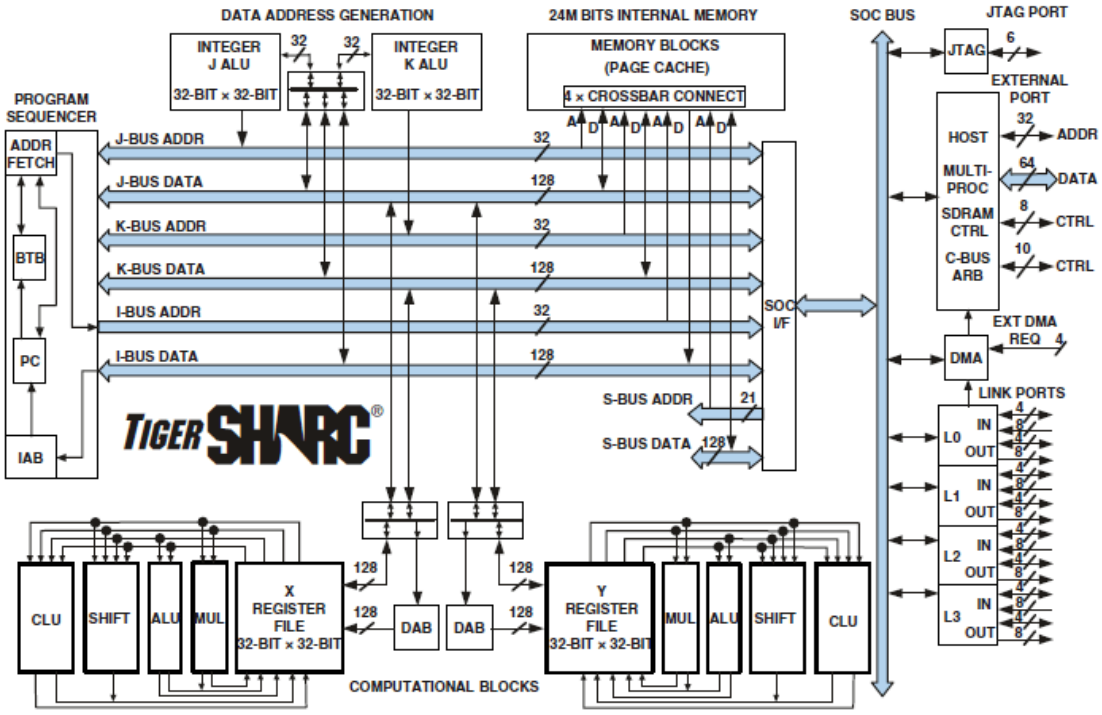
```
     j0 = vectf;;
     k0 = resf;;
     j31 = 0;;
     r10 = [j31 + scalf];;
     lc0 = N/2; r0 = l[j0 += 2];;
     fr16 = r0 + r10; r0 = l[j0 += 2];;
soucetf:
     if nlc0e, jump soucetf; l[k0 += 2] = r16; fr16 = r0 + r10;
     r0 = l[j0 += 2];;
```

V registru *r10* je uložen skalár a do registru *r0* jsou postupně načítána jednotlivá čísla vektoru z paměti DSP. Před cyklem je nutné načíst data, protože procesor nedoká-

že v jednom instrukčním cyklu data načíst z paměti do registru a pracovat s nimi. Poté následuje cyklus, který je vytvořen podmínkou *if nlc0e* a navěštím *soucetf*. Podmínka detekuje poslední člen vektoru, pokud se o něj nejedná, tak program pokračuje na návěští. V samotné smyčce je paralelní instrukce, která uloží výsledek součtu z předchozího instrukčního cyklu do paměti, provede součet členů vektoru a skaláru, poslední instrukcí jsou před načtena dat pro následující iteraci. Další instrukce po dokončení cyklu, uloží z poslední iterace výsledek do paměti procesoru. [18]

1.2.3 Blokové schéma procesoru

Na obrázku (Obr. 2) je znázorněno blokové schéma procesoru. Obsahuje čtyři dvojce sběrnic, kde vždy každá dvojce obsahuje jednu datovou a jednu adresovou sběrnici.



Obr. 2: Blokové schéma ADSP-TS201S [15]

Procesor se skládá ze dvou nezávislých matematických výpočetních jednotek (Computation Blocks) X a Y, které jsou připojeny přes přepínač ke sběrnicím J a K. První funkcí přepínače je výběr sběrnice, která se bude využívat pro komunikaci s pamětí. Druhou funkcí je, že umožňuje načítaná data poslat přes DAB (paměť FIFO), kde jsou správně přizpůsobena špatně zarovnaná data do čtveřic 32-bitových čísel (operace s krátkým nebo velmi dlouhým slovem).

Další částí je program sequencer, který slouží k řízení procesoru podle jednotlivých instrukcí programu. Data address generation slouží ke generovaní adres pro výpočetní

jednotky přistupující do paměti. DSP má 24 Mb paměť, ta je společná pro data i program a je připojena ke všem sběrnicím. Poslední části procesoru jsou komunikační obvody: JTAG, externí porty, sběrnice pro připojení externí paměti a multiprocesorové zapojení. Jednotka SOC, slouží ke komunikaci s I/O perifériemi. [18]

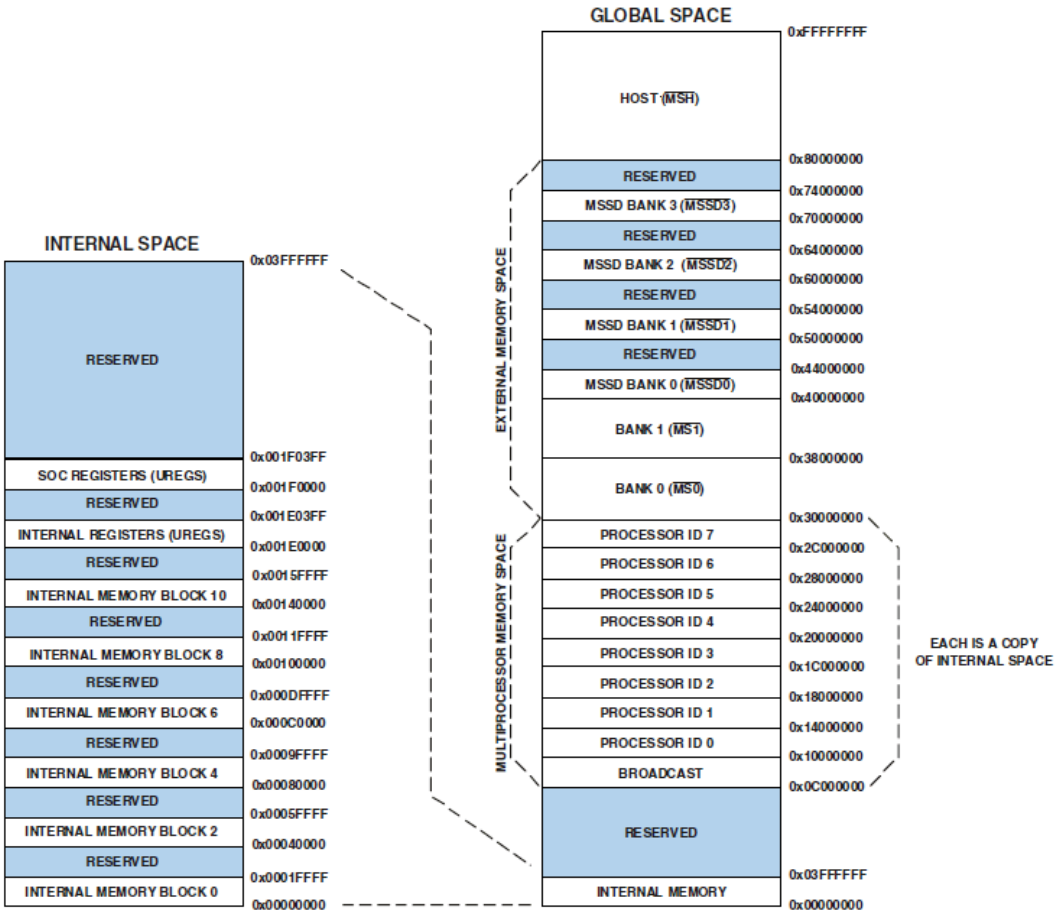
1.2.4 Sběrnice

ADSP-TS201S má čtyři datové 128 b sběrnice a čtyři 32 b adresové sběrnice, kterými je možné docílit datového toku 33 Gb/s. Všechny sběrnice (Obr. 2) jsou propojeny s interní pamětí procesoru, program sequencerem. Ostatní bloky ke komunikaci využívají vždy jen některou z nich. Sběrnice J slouží k načítaní jednotlivých instrukcí z paměti. I a K využívají výhradně výpočetní bloky. Poslední slouží především ke komunikaci jednotky SOC s pamětí. Všechny sběrnice jsou nezávislé a je možné je využívat souběžně. Je to jedna z hlavních výhod DSP, která napomáhá k dosažení vynikajících výpočetních výsledků. V praxi to znamená, že oba výpočetní bloky mohou komunikovat současně s dvěma různými bloky paměti po J a K sběrnici a přitom souběžně ze třetího bloku může program sequencer načítat VLIW instrukci.

1.2.5 Paměť

Na obrázku (Obr. 3) je znázorněna mapa paměti procesoru. Procesor umožňuje adresovat 4 G slov 32-bitové paměti. Z toho 24 Mb je interní paměť, zbytek adres je určen pro externí paměť nebo sdílenou paměť pro multiprocesorové zapojení. K paměti je možné současně přistupovat ze všech čtyřech sběrnic.

Interní paměť je rozdělena do šesti bloků po 4 Mb. Každý ze šesti bloků je rozdělen na horní a dolní polovinu paměti. K oběma polovinám je možné přistupovat souběžně v jednom instrukčním cyklu, a proto každá z nich obsahuje vyrovnávací paměť. Ta slouží k urychlení ukládaní a načítaní dat, protože frekvence, na které pracuje paměť, je poloviční oproti frekvenci jádra procesoru. Tento horní a spodní blok je dále ještě rozdělen na polovinu (sub-array). Každá polovina má svoji vlastní 2 Kb paměť (page buffer). Pokud je přistupováno k paměti, je vždy do page bufferu načten/uložen celý blok 64 x 32-bitového čísla. Page buffer také umožňuje detekovat, zda je konkrétní část paměti využívána nebo je k ní možné přistupovat. Při návrhu programu pro DSP je velice důležité s těmito vlastnostmi počítat a data vhodně rozdělit tak, aby se k nim dalo s co největší mírou pravděpodobnosti přistupovat. [18]



Obr. 3: Mapa paměti ADSP-TS201S [15]

Do jednoho 4Mb bloku je možné uložit maximálně 128 tis. 32 b čísel. Takže je teoreticky možné vytvořit maximální velikost dvourozměrného pole 32-bitových slov o velikosti 113 x 113 čísel. Tato hodnota je orientační, protože každý blok obsahuje ještě několik vyhrazených míst v paměti, které využívá kompilátor, pokud je program napsán v jazyce C/C++. Detailní rozdělení paměťového prostoru je potřeba určit v souboru *linker description file*, kde lze detailně zvolit umístění a velikost jednotlivých sekcí, kam linker umístí části kódu programu a data. [17] [18]

1.2.6 Výpočetní blok

Jak již bylo řečeno, procesor obsahuje dva na sobě nezávislé výpočetní bloky, které mají označení X a Y (Obr. 2). Oba dva bloky jsou připojeny ke dvěma datovým sběrnicím, to jim umožňuje souběžnou komunikaci s pamětí. Každý blok obsahuje jednotku na zpracování komunikačních signálů (CLU), aritmeticko-logickou jednotku (ALU), násobičku (MUL), jednotku na bitové operace (SHIFT) a sadu 32 registrů.

CLU provádí specializované operace pro zpracování komunikačních signálů, především usnadňují práci s dekódováním (například CMDA). Jednotka také umožňuje vý-

počty s komplexními čísly a pro operace násobení a sčítání dokáže nahradit i jednotku ALU, nevýhodou však je, že tyto operace dokáže jen s čísly s pevnou řádovou čárkou.

Jednotka SHIFT slouží k bitovým operacím. Obsahuje sadu funkcí na jejich rotace, blokové posuny, znaménkové operace a jednoduché bitové operace. Práce s bitovými operacemi je možné provádět pouze s čísly uloženými ve formátu s pevnou řádovou čárkou.

Aritmeticko-logická jednotka (ALU) dokáže pracovat s operandy ve formátech s pevnou i s plovoucí řádovou čárkou se znaménkem i bez znaménka. Slouží k výpočtu základních matematických operací jako je například sčítaní, odečítaní (i s carry bitem), výpočet absolutní hodnoty a negace, určování minima (maxima), výpočet střední hodnoty atd. ALU jednotka má podporu i pro saturační aritmetiku, což znamená, že pokud dojde k přetečení registru vlivem matematické operace, je tato událost zaznamenána do speciálního (stavového) registru AV.

Násobička (MUL) slouží pro MAC (Multiply-accumulate) operace. V jednom instrukčním cyklu dokáže vynásobit dvě čísla a výsledek sečíst s akumulátorem. Dokáže pracovat s pevnou i plovoucí řádovou čárkou. Násobení je možné provádět i v komplexním oboru čísel, ale pouze pro formát v pevné řádové čárce.

Pokud by postačovalo výpočty provádět v pevné řádové čárce s přesností 16 b nebo 8 b, je možné využít speciální funkce jednotek ALU a MUL. Tato jednotka vždy pracuje s 32-bytovým blokem dat, do něj se vejdou čtyři 8 b čísla, a proto je možné v jednom instrukčním cyklu provést čtyři operace s osmi čísly (analogicky pro 16 b, kde je možné provést dvě operace). Podmínkou je, že čtveřice 8 b čísel musí být v paměti uložena po sobě. Tato vlastnost může podstatně zvednout výpočetní výkon procesoru.

Jak již bylo řečeno ADSP-TS201S má dva nezávislé bloky, mezi které je možné rozložit výpočetní operace. Procesor podporuje MIMD (Multiple Instruction stream, Multiple Data stream), to umožňuje buď jedním příkazem v jediném instrukčním cyklu provést dvě stejné operace (každou v jiném výpočetním bloku), například vynásobit dvakrát dvě čísla, nebo pomocí dvou příkazů v jednom instrukčním cyklu provést v každém bloku různé operace, a to vždy s odlišnými daty.

Následující přehled (Tabulka 1) orientačně představuje čas potřebný pro výpočet několika základních matematických úloh. [18]

Tabulka 1: Výkon procesoru TS201S

32b float algoritmus	čas
1024p cFFT	15.64 μs
[8 x 8] x [8 x 8] Násobení matic	2.33 μs
FIR filtr (na vzorek)	0.83 ns
cFIR filtr (na vzorek)	3.33 ns

1.2.7 Program sequencer, adresový generátor

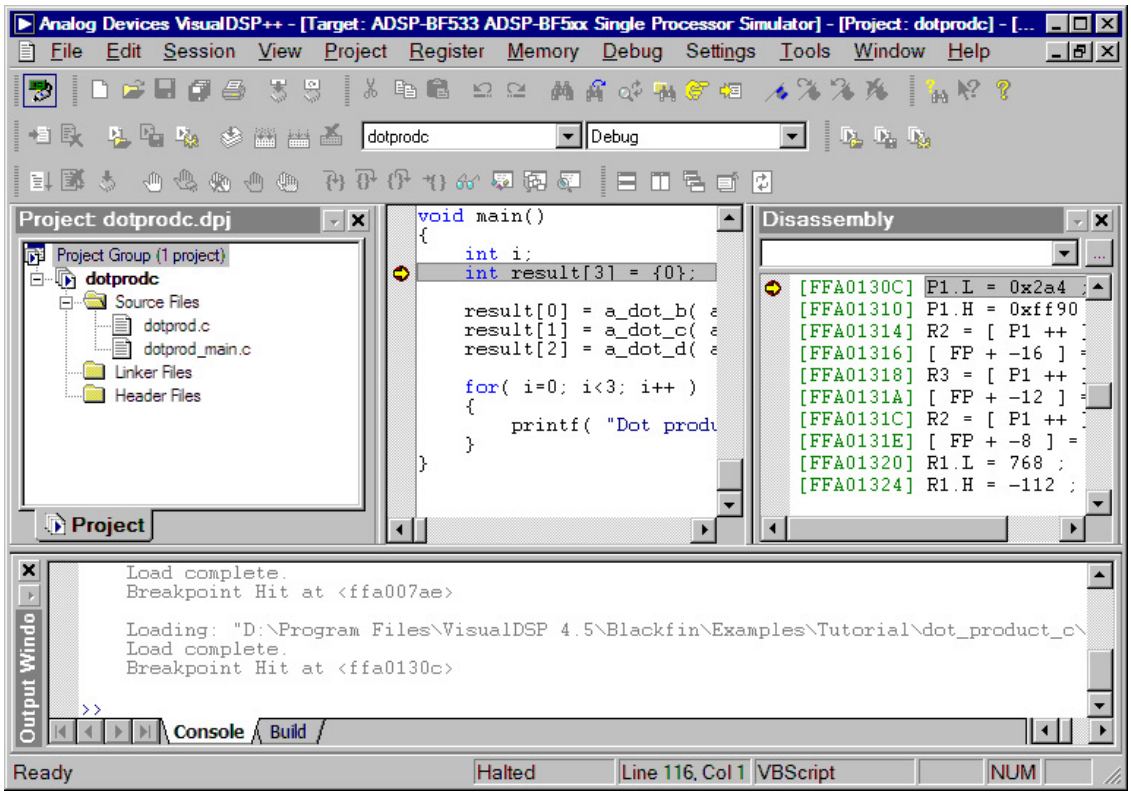
Program sequencer slouží k řízení procesoru dle jednotlivých instrukcí programu. Postupně jsou data načítána z paměti, dekódována a vykonávána v jednotlivých blocích procesoru. Skládá se z IAB (Instruction Alignment Buffer), jedná se o vstupní vyrovnávací paměť pro přicházející instrukce. V této paměti se také instrukce zarovnávají do bloků po čtyřech 32 b instrukcích. Jakmile je zaplněno celé 128-bitové instrukční slovo, jsou instrukce dále distribuovány mezi výpočetní blok, IALU a program sequencer.

BTB (Branch Target Buffer) je vnitřní vyrovnávací paměť, kde jsou uloženy různé větve programu, které vzniknou například podmíněnými skoky nebo cykly. Dokáže uchovat 128 instrukcí a čtyři různé větve. Tato metoda pomáhá urychlit práci procesoru. Pokud dojde ke změně předpokládané cesty, BTB aktualizuje frontu instrukcí. Program sequencer podporuje čtyři různé druhy operací, které narušují kontinuální tok. Změna způsobená podmínkou, ukončením cyklu, podmíněné skoky a hardwarově vyvolané přerušení (externí, časové, komunikační).

Posledním nepopsaným blokem procesoru je adresový generátor, který slouží k adresování datových přenosů po sběrnicích a také obsahuje dvě nezávislé jednotky IALU (Integer Arithmetic Logic Units) pro řešení operací s celočíselným datovým typem. Pomocí této jednotky je možné sčítat, násobit, provádět aritmetické posuny a logické operace a mnoho dalších elementárních, matematických funkcí. Umožňuje přímé a nepřímé adresovaní (postinkrementací, preinkrementací) a adresovaní na kruhový zásobník. [18]

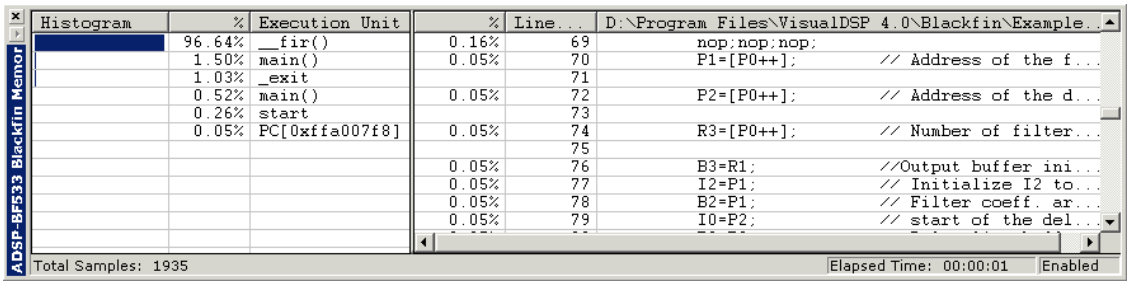
1.3 Vývojové nástroje

VisualDSP++ je program, určený k vývoji a simulování programů pro procesory od firmy Analog Devices. Umožňuje pracovat jak v jazyce symbolických adres, tak i v C/C++, nebo je vzájemně kombinovat. Pro podporu vývoje aplikací v jazyce C/C++ je připravena knihovna s více jak stem předprogramovaných funkcí. VisualDSP++ obsahuje všechny běžné nástroje jako je kompilátor s optimalizací kódu, linker a debugger. Následující obrázek znázorňuje vývojové prostředí VisualDSP++ (Obr. 4). V levé části se nachází manažer vytvořených projektů, v prostředním okně program v režimu debuggru, v pravé části přeložený kód z C/C++ do jazyka symbolických adres a v dolní části je výpis chybových hlášek a debuggru.



Obr. 4: Vývojové prostředí VisualDSP++

VisualDSP++ obsahují simulátor a emulátor programů. Simulace projektů umožňuje v softwarově vytvořeném prostředí testovat vytvořené algoritmy, jako by se nacházely přímo v procesoru. Emulátor slouží k testování projektů na skutečném procesoru s tím, že pomocí VisualDSP++ je možné řídit jeho funkce, zobrazovat a měnit obsah paměti a krokovat jeho běh přímo v procesoru.



Obr. 5: Profiler VisualDSP++

Velice přínosnou součástí debuggeru je profiler (Obr. 5), který umožňuje analyzovat efektivitu algoritmu a určit, jak vykonávání konkrétní části kódu zatěžuje procesor, tedy například kolik instrukčních cyklů trvá danná funkce. Hodnoty lze vyjádřit i procentuálně a graficky je zobrazit.

1.4 VisualDSP++ C/C++ library

K vývojovému prostředí VisualDSP++ jsou dostupné knihovny, které mají usnadnit a zefektivnit vytváření programů v jazyce C/C++ pro DSP. Knihovny jsou napsány v jazyce symbolických adres a jsou vysoce optimalizovány pro konkrétní typ procesoru. Lze je rozdělit na dvě části, běžné CRT knihovny dle normy C/C++ a takzvané DSP library. První obsahují běžné operace jako operace s pamětí, matematické operace a práce s řetězci.

DSP library jsou navrženy tak, aby odpovídaly nejčastějšímu použití ve zpracování signálů. Obsahují funkce pro práci s maticemi, komplexními čísly, filtrace signálu, FFT a mnoho dalších.

Všechny dostupné funkce jsou se zakoupením VisualDSP++ volně k použití a je možné je i podle potřeby na úrovni jazyka symbolických adres upravovat. Jejich největší výhodou je jejich vysoká optimalizace, která zaručuje maximální využití potenciálu procesoru v dílčích operacích. [19]

Souhrn dostupných funkcí DSP library s ohledem na předpokládané budoucí řešení soustav lineárních rovnic:

- transpmf() – transpozice matice
- matmmltf() – násobení dvou matic
- matmadd() – součet matic
- matmsubf() – rozdíl dvou matic

- vecdotf() – skalární součin matic
- matinvf() - výpočet inverzní matice

Všechny funkce jsou navrženy pro datový typ float (dostupné jsou i modifikace pro jiné typy s pevnou řádovou čárkou). Vstupem do těchto funkcí je pointer na místo v paměti konkrétní proměnné, totéž platí i pro výstupní hodnoty.

Pro ilustraci práce s funkcemi DSP library je v následujících dvou výpisech popsána hlavička funkce *matmmltf* a způsob jejího volání:

```
#include <matrix.h>

void matmmltf (
      const float *a; /* Pointer to input matrix a[][] */
      int n; /* Number of rows in matrix a[][] */
      int k; /* Number of columns in matrix a[][] */
      const float *b; /* Pointer to input matrix b[][] */
      int m; /* Number of columns in matrix b[][] */
      float *c; /* Pointer to matrix c[][] */
)
```

Z kódu je možné zjistit, že do funkce vstupují tři pointery na místo v paměti. V jazyce C však má pointer na dvourozměrné pole tvar ***A*. Důvodem proč je možné použít pointer na jednorozměrné pole je ten, že data jsou v paměti uložena po sobě (po prvním řádku matice následuje další řadek). Do funkce na výpočet součinu dvou matic proto stačí předat adresu prvního prvku **a* a rozměry matice *n* a *k* (řádky, sloupce). Pro druhou matici platí stejná pravidla s tím rozdílem, že stačí předat pouze jeden rozměr matice (jeden rozměr první matice musí být stejný s rozměrem druhé matice).

Následující kód demonstruje jak je možné z programu volat funkci z DSP library:

```
#include <matrix.h>

#define ROWS_1 4
#define COLS_1 8
#define COLS_2 2
double input_1[ROWS_1][COLS_1], *a_p = (double *) (&input_1);
double input_2[COLS_1][COLS_2], *b_p = (double *) (&input_2);
double result[ROWS_1][COLS_2], *res_p = (double *) (&result);

matmmlt (a_p, ROWS_1, COLS_1, COLS_2, b_p, res_p);
```

V prvním kroku je potřeba načíst knihovnu *matrix*, která obsahuje funkci *matmmlt*. V dalším jsou nadefinovány konstanty a uloženy do nich velikosti matic. Následně je vytvořena proměnná dvourozměrného pole, poté je vytvořen pointer na první prvek této proměnné. To je provedeno jak pro vstupní, tak pro výstupní matici. V posledním kroku je volána funkce (jelikož je typu void, nevrací žádnou hodnotu), výsledek je uložen do proměnné *result*. [19]

2 Řešení soustav lineárních rovnic

2.1 Základní věta a vymezení problému

Obecně může být soustava *m* lineárních rovnic s *n* proměnnými zapsána jako

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n &= b_m, \end{aligned} \tag{3}$$

kde proměnné x_n jsou neznámé, a_m jsou koeficienty soustavy rovnic a čísla b_n jsou absolutní čísla soustavy (pravá strana rovnice). Soustava se dá vyjádřit také v maticovém tvaru

$$\mathbf{Ax} = \mathbf{b}, \tag{4}$$

kde **A** je matice koeficientů typu *n* x *n*, **x**, **b** jsou vektory. Hodnost matice[3] **A** je označena symbolem r(**A**). Symbolem $\mathbf{A_b}$ označíme matici *n* x *n* + 1, která vznikne z matice **A** tak, že za *n* sloupců této matice se přidá vektor **b** jako (*n* + 1)-ní sloupec.

Věta 1. *Soustava rovnic (4) má řešení tehdy a jen tehdy, platí-li*

$$r(\mathbf{A}) = r(\mathbf{A_b}), \tag{5}$$

Předchozí větu a její důsledky lze dokázat Gaussovou eliminací.

Tato práce se zabývá způsobem jak nejefektivněji řešit velké soustavy rovnic s více jak 100 neznámými, proto je potřeba definovat pojem *efektivnost algoritmu*, která má dvě základní vlastnosti:

- Rychlost algoritmu (počet operací)
- Přesnost vypočteného řešení

U první je zcela zřejmé, že pro velké soustavy rovnic je potřeba volit algoritmy, které mají co nejmenší výpočetní složitost s ohledem na danou výpočetní jednotku. Druhá

3 Platí, že maximální počet lineárně nezávislých sloupců je roven maximálnímu počtu lineárně nezávislých řádků matice. Definice hodnosti je tedy platná také pro maximální počet lineárně nezávislých sloupců [2] .

vlastnost je velice důležitá z toho důvodu, že i malá chyba v zaokrouhlení u velké soustavy může v konečném důsledku způsobit špatné výsledky. Jelikož se obě dvě vlastnosti efektivnosti algoritmu prolínají, je potřeba při hledání optimálního řešení zvolit správný poměr i s vlastnostmi jednotlivých metod řešení. [1]

2.2 Obecné vlastnosti matic

V praxi se obvykle objevují dva základní typy koeficientů matic a to plné nebo řídké matice. Plnou maticí rozumíme takovou, která má velmi málo nulových prvků. Matice tohoto typu se často objevují při řešení statistických, fyzickálních a technických problémech. Řídké matice jsou ty, které obsahují velké množství nulových prvků v matici. Nejčastěji vznikají z numerického řešení parciálních diferenciálních rovnic.

K řešení rovnic na číslicových počítačích se využívá dvou základních pohledů. Přímé a iterační metody.

Přímé se využívají k řešení plných matic, ale někdy je vhodné je využít i pro řešení matic řídkých. Tyto metody vedou k přesnému řešení dané soustavy, avšak jsou často ovlivňovány chybou zaokrouhlení. Všechny tyto metody obsahují postupný proces založený na Gaussově eliminaci. Jelikož pozdější využití této diplomové práce má konkrétní zadání, tak v dalších textech jsou popsány a zkoumány výhradně přímé metody. Předpokládané vstupní matice koeficientů jsou čtvercové, plné matice s maximálním počtem 144 neznámých a jsou požadovány přesné výsledky.

Iterační metody se hodí především pro řešení řídkých soustav rovnic a velice velké matice. Při využití těchto metod dochází v každém cyklu ke zpřesňování výsledků. Obvykle je potřeba sledovat konvergenční kritéria, která určují dobu běhu algoritmu. Tyto metody mají velice dobrou efektivitu algoritmu, za prvé rychlost výpočtu a také nejsou tak náchylné na chyby způsobené zaokrouhlováním. Některé upravené metody lze použít i na určité typy plných matic, ale jejich hlavní nevýhodou je, že některé rovnice nemusí konvergovat ke správným výsledkům, tím metoda zkrachuje. Druhým úskalím je, že se přesně nedá určit doba běhu algoritmu, pouze obecný řád konvergence. Většina metod není konečná a je ji potřeba ukončit na základě předem daných podmínek. Mezi nejznámější iterační algoritmy patří Jacobiho metoda, Gaussova iterační metoda a metoda sdružených gradientů.

V případě řešení konkrétních rovnic by šlo využít i kombinaci těchto metod. Přímou metodou spočítat odhad řešení s velkou nepřesností a poté použít iterační metodu

na zpřesnění výsledků. V tomto případě by byla s větší pravděpodobností zaručena konvergence i rychlost celého algoritmu. Dalším vhodným nasazením iteračních metod by mohlo být případ, pokud by byli předem známé tyto odhady, nebo se dali snadno odhadnout či spočítat z předchozích výsledků. Ale ani jedna z těchto variant nebyla, pro konkrétní využití této práce, prozatím prokázána. [1]

2.3 Podmíněnost a chyby při řešení soustav rovnic

Předpokládejme, že **A** je regulární matice[4], pak je řešení soustavy rovnic podle vzorce (6)

$$\mathbf{x} = \mathbf{A}^{-1}\,\mathbf{b}, \tag{6}$$

pokud není matice **A** normalizovaná[5] a je v ní například velké číslo a přesto budou mezivýsledky zaokrouhleny na stejný řád, bude docházet k chybě (Příklad 1.). Problém lze vyjádřit za pomoci rezidua r

$$\mathrm{r} = \mathbf{b} - \mathbf{A}\mathbf{x}_\mathbf{c}, \tag{7}$$

kde $\mathbf{x}_\mathbf{c}$ je vypočítané řešení. Má-li inverzní matice $\mathbf{A}^{-1}$ některé prvky řádově odlišné od ostatních muže být r velmi malé, i když se vektor $\mathbf{x}_\mathbf{c}$ liší od správného řešení $\mathbf{x}_\mathbf{t}$

$$\mathrm{r} = \mathbf{A}(\mathbf{x}_\mathbf{t} - \mathbf{x}_\mathbf{c}). \tag{8}$$

Velikost rezidua nám tedy určí nepodmíněnost soustavy rovnic.

Příklad 1. [1]

$$2x + 6y = 8$$
$$2x + 6{,}00001 = 8{,}00001$$
řešení: x = 1, y = 1

$$2x + 6y = 8$$
$$2x + 5{,}99999 = 8{,}00002$$
řešení: x = 10, y = -2

Malá chyba v zaokrouhlení způsobila úplně odlišné výsledky. Matice $\mathbf{A}^{-1}$ má prvky řádu 10^5, což ukazuje její špatnou podmíněnost. Aby byla matice normalizovaná, je nutno ji vynásobit 10^5, poté budou výsledky vycházet správné. [1]

4 Regulární matice (v některé literatuře též invertibilní) je taková čtvercová matice jejíž determinant je různý od nuly, tzn. $A \neq 0$ [2] .

5 Normalizovaná matice znamená, že všechny její prvky jsou podobného řádu.

Při řešení soustav lineárních rovnic dochází tedy ke třem různým chybám:

- chyby zaokrouhlení
- špatně podmíněná soustava
- chyby iteračních metod (v konečné iteraci neznáme přesný výsledek)

2.4 Matice s komplexními čísly

Jelikož vstupní data pro výpočet soustav lineárních rovnic nemusejí být pouze reálná čísla, ale mohou obsahovat i komplexní složky, je potřeba určit způsoby těchto výpočtů. Existují dva základní pohledy jak tento problém řešit. Při výpočtech využít buď algebraického tvaru komplexních čísel a nebo matici rozložit do fuzzy soustavy, kde se následně může řešit jako lineární rovnice v oboru reálných čísel.

Při použití algebraického tvaru je potřeba všechny výpočty provádět pomocí následujících vzorců. Je vidět, že vzorec pro podíl (12) je komplikovaný a pokud se bude často vyskytovat v algoritmu, značně se zhorší jeho efektivita. [3]

$$(a+ib)+(c+id)=(a+c)+i(b+d), \tag{9}$$

$$(a+ib)-(c+id)=(a-c)+i(b-d), \tag{10}$$

$$(a+ib)\cdot(c+id)=(ac-bd)+i(ad+bc), \tag{11}$$

$$\frac{a+ib}{c+id}=\frac{(a+ib)(c-id)}{(c+id)(c-id)}=\frac{(ac+bd)+i(bc-ad)}{c^2+d^2}=\left(\frac{ac+bd}{c^2+d^2}\right)+i\left(\frac{bc-ad}{c^2+d^2}\right). \tag{12}$$

Řešení fuzzy soustavy rovnic se provede tak, že komplexní čtvercovou matici **C** o N prvcích je možné rozložit na jinou čtvercovou matici o 2N reálných prvcích. Komplexní vektor **b** (pravá strana rovnice) se také rozloží s podobnými pravidly jako matice **C**. Poté nám vznikne soustava lineárních rovnic v oboru reálných čísel (Věta 2.).

Věta 2. *Soustava lineárních rovnic (13)*

$$\begin{aligned} c_{11}z_1 + c_{12}z_2 + \ldots + c_{1n}z_n &= w_1 \\ c_{21}z_1 + c_{22}z_2 + \ldots + c_{2n}z_n &= w_2 \\ &\vdots \\ c_{m1}z_1 + c_{m2}z_2 + \ldots + c_{mn}z_n &= w_m, \end{aligned} \tag{13}$$

kde jsou koeficienty čtvercové matice C = (cij), 1 ≤ i,j ≤ n komplexní čísla a wi 1 ≤ i,j ≤ n fuzzy čísla[6]*. Je možné soustavu dále vyjádřit jako (14):*

$$\sum_{j=1}^{n} c_i z_j = w_i \in 1,2..n \tag{14}$$

soustava (14) se rozloží na (15):

$$\begin{aligned} c_{ij} &= a_{ij} + ib_{ij} \\ z_i &= p_i + iq_i \\ w_i &= u_i + iv_i \end{aligned} \tag{15}$$

Pro řešení této soustavy rovnic se vytvoří následující matice (16)

$$\begin{aligned} \mathbf{V} &= [v_i] \\ \mathbf{U} &= [u_i] \\ \mathbf{A} &= [a_{ij}] \\ \mathbf{B} &= [b_{ij}] \\ \mathbf{P} &= [p_i] \\ \mathbf{Q} &= [q_i] \end{aligned} \tag{16}$$

a řeší se soustava rovnic (17).

$$\begin{bmatrix} A & -B \\ B & A \end{bmatrix} \begin{bmatrix} P \\ Q \end{bmatrix} = \begin{bmatrix} U \\ V \end{bmatrix} \tag{17}$$

Řešení vzorové soustavy rovnic demonstrující tuto metodu a ověření její věrohodnosti je předvedeno v následujícím algoritmu programu MATLAB.

6 Komplexní čísla jsou speciální variantou, ale také spadají do kategorie fuzzy čísel. Tato metoda je nejčastěji využívána pro řešení RLC obvodu v elektrotechnice.

```
C =
   6.5000 + 3.2000i   3.7000 + 8.1000i
  -2.2000 + 4.0000i  17.0000 -12.0000i

b =
  12.1000 + 3.7000i
   8.0000 - 5.2000i

S =
    6.5000    3.7000   -3.2000   -8.1000
   -2.2000   17.0000   -4.0000   12.0000
    3.2000    8.1000    6.5000    3.7000
    4.0000  -12.0000   -2.2000   17.0000

b2 =
   12.1000
    8.0000
    3.7000
   -5.2000

>> x = inv(C) * b
x =
   1.0031 - 0.5660i
   0.6017 - 0.1904i

>> z = inv(S) * b2
z =
    1.0031
    0.6017
   -0.5660
   -0.1904
```

Matice **C** (koeficienty) a vektor **b** pravé strany tvoří soustavu rovnic s komplexními čísly. Řešení této soustavy je uloženo ve vektoru **x**, které slouží jako referenční hodnota. Matice **S** a vektor **b2** je sestavena podle vztahů (16), (17) a poté řešena soustava lineárních rovnic a to pouze s reálnými čísly. Vektor **z** je výsledek řešení, jak je vidět podle upraveného vztahu (18) odpovídá vektoru **x**, kde **P** je reálná část a **Q** imaginární s řešením **x**.

$$\begin{bmatrix} P \\ Q \end{bmatrix} = \begin{bmatrix} p_1 \\ p_2 \\ q_1 \\ q_2 \end{bmatrix} \tag{18}$$

Pří řešení soustav lineárních rovnic s komplexními čísly je vždy potřeba posoudit, jakou variantu zvolit. Je zřejmé, že první metoda bude vhodnější pro algoritmus, kde se bude vyskytovat minimum dělení a menší množství násobení. Odhad složitosti bude záviset na konkrétní metodě. Druhá metoda má jasně dané, že složitost algoritmu se zvětší dvojnásobně, protože výsledná matice má dvojnásobný rozměr.

Existují také metody, které už předpokládají, že vstupní matice bude s komplexními čísly, ale jedná se především o modifikace iteračních metod. [4]

3 Metody řešící soustavy lineárních rovnic

Všechny zde zmíněné metody budou řešeny pouze v oboru reálných čísel. V předchozí kapitole (Kapitola 2.4) je znázorněno jak pracovat s komplexní soustavou rovnic, a proto všechny následující algoritmy mohou být upraveny i pro komplexní čísla. Při výběru metod a konkrétních algoritmů je již zohledňováno pozdější nasazeni v signálovém procesoru, který má specifické vlastnosti (Kapitola 1).

3.1 Gaussova eliminace

Jedná se o základní přímou metodu, která je stabilní a především je základem pro ostatní metody založené na rozkladu matic. V praxi se však nepoužívá, jelikož je zhruba 3x pomalejší [5] než jakákoliv jiná metoda založená na rozkladu a je velice citlivá na chyby způsobené zaokrouhlováním. Metoda nezohledňuje nepodmíněné soustavy rovnic, což se v daném zadání nedá zaručit.

$$[\mathbf{A}]\,[\mathbf{x}1 \cdot \mathbf{x}2 \cdot \mathbf{x}3 \cdot \mathbf{Y}] = [\mathbf{b}1 \cdot \mathbf{b}2 \cdot \mathbf{b}3 \cdot \mathbf{I}] \tag{19}$$

A, **Y** jsou čtvercové matice a **b**, **x** jsou sloupcové vektory, **I** je jednotková matice. Pomocí povolených úprav pro soustavy rovnic, což jsou výměna dvou řádků, vynásobení řádku nenulovým číslem, přičítání násobku některého řádku k jinému, je snahou levou stranu matici **A** upravit na matici **I**. Poté na pravé straně místo jednotkové matice zůstane řešení soustavy rovnic. Tento postup se někdy také nazývá řešení soustav rovnic pomocí inverzní matice, kde výsledný tvar soustavy rovnic je vzorec (6). [1] [5]

3.2 Zpětná substituce

Mějme matici **A**, kterou je možné upravit některou z metod na tvar

$$\begin{bmatrix} a'_{11} & a'_{12} & a'_{13} & a'_{14} \\ 0 & a'_{22} & a'_{23} & a'_{24} \\ 0 & 0 & a'_{33} & a'_{34} \\ 0 & 0 & 0 & a'_{44} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \end{bmatrix} \tag{20}$$

kde matice **A'** je horní trojúhelníková a **b'** je pravá strana. Přičemž platí, že $\mathbf{A} \neq \mathbf{A'}$. Za těchto předpokladů je možné pomocí vzorce (21) spočítat postupně všechny neznámou x_i.

$$x_i = \frac{1}{a'_{ii}} \left[b'_i - \sum_{j=i+1}^{N} a'_{ij} x_j \right] \tag{21}$$

Tuto metodu lze přepsat do následujícího algoritmu, který vypočte **X** dané soustavy rovnicí (6). [5]

```
function [x] = mgss(obj,Q,R,b);
    [m,n] = size(Q);
    b = Q'*b;
    x = zeros(n,1);
    x(n) = b(n)/R(n,n);
    for i = n-1:-1:1
      x(i) = (b(i) - R(i,i+1:n)*x(i+1:n))/R(i,i);
    end
end
```

3.3 LU dekompozice

Mějme čtvercovou regulární matici **A,** ta lze rozložit na soustavu (22)

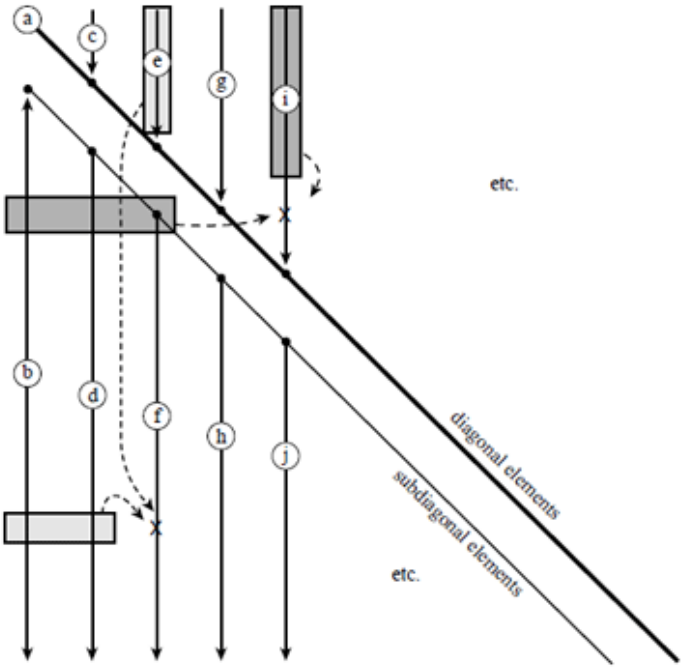
$$\mathbf{A} = \mathbf{LU}, \tag{22}$$

kde **L** je dolní trojúhelníková matice s jedničkami na hlavní diagonále a **U** je horní trojúhelníková matice. Pokud touto metodou chceme řešit soustavu lineárních rovnic musí se využít dvou rovnic (23): první je řešena dopřednou substitucí a následně druhá zpětnou substitucí (kapitola 3.2).

$$\begin{aligned} \mathbf{Ly} &= \mathbf{b} \\ \mathbf{Ux} &= \mathbf{y} \end{aligned} \tag{23}$$

Pokud předpokládáme, že matice **L** má na hlavní diagonále samé jedničky ($l_{ii} = 1$, i = 1, 2 .. j) poté jsou ostatní prvky vypočteny podle vzorce (24), (Obr. 6)

$$\begin{aligned} u_{ij} &= a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}, \quad i = 1, \ldots, j, \\ l_{ij} &= \frac{1}{u_{jj}} \left(a_{ij} - \sum_{p=1}^{j-1} l_{ip} u_{pj} \right), \quad i = j+1, j+2, \ldots, n. \end{aligned} \tag{24}$$



Obr. 6: Algoritmus na vytvoření trojúhelníkových matic LU rozkladem [5]

Postupně je počítán první řádek matice **U**, potom první řádek sloupce **L**, druhý řádek matice **U**, atd. Přestože je matice **A** regulární, mohou se objevit na diagonále nulové prvky. Eliminace dovoluje vzájemně přehodit řádky matice, ale po této změně už nebude vycházet trojúhelníková matice. Proto je následně potřeba rozklad zleva násobit permutační maticí **P** (25), aby byla zachována rovnost

$$\mathbf{A} = \mathbf{PLU}. \tag{25}$$

```
function [lu, pvt] = LUfactor (obj,A)
    [nrow ncol] = size (A);
    if ( nrow ~= ncol )
       disp ( 'Není ctvercova matice' );
       return;
    end;
    for i=1:nrow
        pvt(i) = i;
    end;
    for i = 1 : nrow - 1
        t =  min ( find ( abs(A(pvt(i:nrow),i)) ==
           max(abs(A(pvt(i:nrow),i))) ) + i-1 );
        if ( t ~= i )
           temp = pvt(i);
           pvt(i) = pvt(t);
           pvt(t) = temp;
        end;
        if ( A(pvt(i),i) == 0 )
           disp ( 'LUfactor error: matrix is singular' );
           lu = A;
           return
        end;

        for j = i+1 : nrow
            m = -A(pvt(j),i) / A(pvt(i),i);
            A(pvt(j),i) = -m;
            A(pvt(j), i+1:nrow);
            A(pvt(j), i+1:nrow) = A(pvt(j), i+1:nrow) + m *
                 A(pvt(i), i+1:nrow);
        end;
    end;
    lu = A;
end
```

Pokud by byla matice **A** pozitivně definitní[7], bylo by možné nahradit řešení rovnic Choleského dekompozicí, která je mnohem efektivnější než jakýkoliv jiný algoritmus, ale u matic vytvořených z fyzikálních jevů to nelze předpokládat. [5]

7 Pozitivně definitní matice má všechna vlastní čísla kladná.

3.4 QR dekompozice

Mějme čtvercovou matici **A**, která lze rozložit na matici **Q** a **R** (26)

$$\mathbf{A} = \mathbf{QR}, \tag{26}$$

kde **Q** je ortogonální matice (27) a **R** je horní trojúhelníková matice

$$\mathbf{Q}^{\mathrm{T}}\mathbf{Q} = 1, \tag{27}$$

pokud je **A** regulární má matice **R** na hlavní diagonále kladná čísla a rozklad bude jednoznačný. Regulárnost matice v praktickém řešení je předpokládána.

Pro rozklad lze použít tři nejznámější metody: Gram-Schmidtův algoritmus, Householderovo zrcadlení a Givensova rotace. První dvě jsou v pozdějším textu zanalyzovány, ale Givensova rotace se příliš nehodí pro aplikování do signálového procesoru.

Pokud chceme řešit soustavu lineárních rovnic pomocí QR rozkladu platí vztah (28)

$$\mathbf{R}\,\mathbf{x} = \mathbf{Q}^{\mathrm{T}}\,\mathbf{b}. \tag{28}$$

Jelikož **R** je trojúhelníková matice je možné tuto rovnici snadno řešit zpětnou substitucí (Kapitola 3.2). [5]

3.4.1 Modifikovaný Gram-Schmidtův algoritmus

Gram-Schmidtův ortogonalizační proces spočívá v následujícím, pokud báze vektoru není ortogonální lze z ní vytvořit novou, která už ortogonální bude.

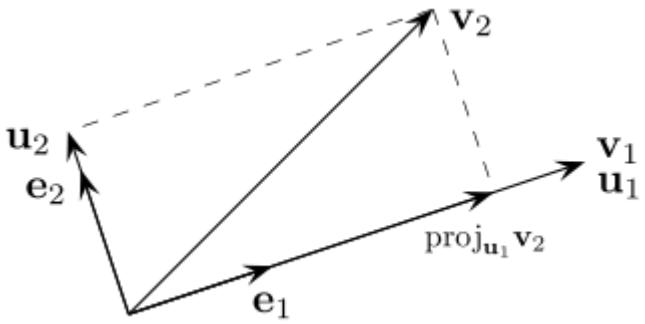
Věta 3. *Jsou-li vektory* $\boldsymbol{v_k}$ *lineárně nezávislé vektory, pak existují jejich takové lineární kombinace* $\boldsymbol{u_k}$*, které jsou vzájemně ortogonální.*

Aby bylo možné vektor $\mathbf{u_k}$ spočítat, musíme si nadefinovat operátor (29)

$$\mathrm{proj}_{\mathbf{u}}(\mathbf{v}) = \frac{\langle \mathbf{v}, \mathbf{u} \rangle}{\langle \mathbf{u}, \mathbf{u} \rangle}\mathbf{u}, \tag{29}$$

kde <**v**, **u**> je skalární součin. Tento operátor je projekcí vektoru **v** do ortogonálního vektoru **u**. Výpočet vektor $\mathbf{u_k}$ je vidět na obrázku (Obr. 7) a spočítá se podle vztahu (30)

$$\mathbf{u}_k = \mathbf{v}_k - \sum_{j=1}^{k-1} \mathrm{proj}_{\mathbf{u}_j}(\mathbf{v}_k), \qquad \mathbf{e}_k = \frac{\mathbf{u}_k}{\|\mathbf{u}_k\|}. \tag{30}$$



Obr. 7: První dva kroky ortogonalizace [7]

Tuto metoda je aplikována v následujícím algoritmu v programu MATLAB podle. [6] [7]

```
%QR rozklad - Modified Gram-Schmidt Orthogonalization
function [q,r] = GramSchmidt(obj,A)
    [m, n] = size(A);
     q = zeros(m, n);
     r = zeros(n, n);
     for k = 1:n
         r(k,k) = norm(A(1:m, k));
         if r(k,k) == 0
             break;
         end
         q(1:m, k) = A(1:m, k) / r(k,k);
         for j = k+1:n
             r(k, j) = dot(q(1:m, k), A(1:m, j));
             A(1:m, j) = A(1:m, j) - r(k, j) * q(1:m, k);
          end
      end
end %QR GS
```

Uvedený algoritmus v této podobě vypadá velice náročný na výpočet kvůli vnořené smyčce, ale lze ho upravit, do maticového tvaru. Poté je možné metodu lépe srovnat s ostatními a později i aplikovat do DSP s pomocí DSP library. [12]

```
function [Q,R] = GramSchmidtDSP(obj,A);
    [m,n] = size(A);
    Q = A;
    R=zeros(n);
    for k = 1:n
        R(k,k) = norm(Q(:,k));
        Q(:,k) = Q(:,k)/R(k,k);
         R(k,k+1:n) = Q(:,k)'*Q(:,k+1:n); %vektor x matice
         Q(:,k+1:n) = Q(:,k+1:n) - Q(:,k)*R(k,k+1:n);
     end
end
```

3.4.2 Householderova transformace

Věta 4. *Každá matice A s reálnými čísly lze pomocí* $\mathbf{H}_s$ *matic rozložit na součin* ***QR*** *(31)*

$$\mathbf{H}_s \cdots \mathbf{H}_2\mathbf{H}_1\mathbf{A} = \mathbf{Q}^T\mathbf{A} = \begin{cases} \begin{pmatrix} \mathbf{R}_1 \\ 0 \end{pmatrix} & m > n, \\ (\mathbf{R}_1,\ 0) & m < n, \\ \mathbf{R} & m = n. \end{cases} \tag{31}$$

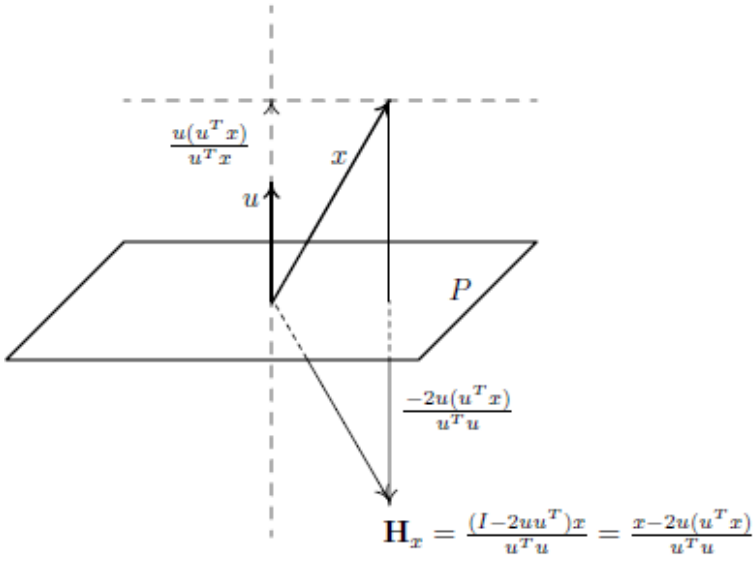
Matice $\mathbf{H}_s$ vznikne v každém kroku tak, že jsou postupně nulovány sloupce pod hlavní diagonálou a vznikne upravená matice $\mathbf{A'}_k$. Přitom v každém kroku platí (32)

$$\mathbf{H}_s\mathbf{A} = \mathbf{A'}_k. \tag{32}$$

Aby bylo možné spočítat **H** potřebujeme znát vektor Householderova zrcadlení **v**, který vznikne ze vzorce (33)

$$\mathbf{v} = \mathbf{x} + ||\mathbf{x}||\ \mathbf{e}, \tag{33}$$

kde **x** je nulovaný vektor a **e** je bázový vektor příslušné délky.



Obr. 8: Householderova transformace (zrcadlení) [9]

Princip Householderovy matice je názorně vidět z obrázku (Obr. 8) a obecného vzorece (34)

$$\mathbf{H}_k := \mathbf{E}_{n-k+1} - 2\frac{u_{n-k+1}u_{n-k+1}^T}{u_{n-k+1}^T u_{n-k+1}} \tag{34}$$

Následně už je velice snadné vytvořit matice **Q**, **R**. Matici **R** = **A'** v posledním kroku, kdy jsou vynulované koeficienty pod hlavní diagonálou. Jelikož matice $\mathbf{H_s}$ jsou ortogonální je možné z nich vytvořit matici **Q** podle vzorce (35), pokud je **Q** potřeba pro další výpočty, jako je například i řešení soustav lineárních rovnic.

$$\mathbf{Q^T} = \mathbf{H_s H_{s+1}} \,..\, \mathbf{H_2 H_1}, \tag{35}$$

Soustava rovnic se následně snadno dopočítá podle vztahu (28) (Kapitola 3.2). Na následujícím kódu je znázorněn algoritmus v programu MATLAB. Tato metoda se využívá i pro SVD rozklad s tím rozdílem, že se matice **A'** upravuje na bidiagonální[8] . [9] [8]

8 Bidiagonální matice má nuly všude kromě hlavní diagonály a superdiagonály.

```
%QR rozklad - Modified Gram-Schmidt Orthogonalization
function [q,r] = GramSchmidt(obj,A)
    [m, n] = size(A);
    q = zeros(m, n);
    r = zeros(n, n);
    for k = 1:n
        r(k,k) = norm(A(1:m, k));
        if r(k,k) == 0
            break;
        end
        q(1:m, k) = A(1:m, k) / r(k,k);
        for j = k+1:n
            r(k, j) = dot(q(1:m, k), A(1:m, j));
            A(1:m, j) = A(1:m, j) - r(k, j) * q(1:m, k);
        end
    end
end

%upravena verze pro DSP
function [Q,R] = GramSchmidtDSP(obj,A);
    [m,n] = size(A);
    Q = A;
    R=zeros(n);
    for k = 1:n
        R(k,k) = norm(Q(:,k));
        Q(:,k) = Q(:,k)/R(k,k);
        R(k,k+1:n) = Q(:,k)'*Q(:,k+1:n);
        Q(:,k+1:n) = Q(:,k+1:n) - Q(:,k)*R(k,k+1:n);
    end
end
```

3.5 SVD dekompozice

V dnešní době jde o velice perspektivní metodu, která se využívá ke kompresím dat, úpravám obrazu či zvuku, ale lze pomocí ní řešit i soustavy rovnic (4) a to s velice velkou úspěšností. Tam kde, většina metod selhává, SVD dává velice dobré výsledky. Základem této metody je rozklad matice **A** podle vztahu (36)

$$\mathbf{A} = \mathbf{PBQ}^{\mathrm{T}}, \tag{36}$$

kde **P** a **Q** jsou ortogonální matice a **B** je matice diagonální, v praxi se však požívá zřídka. **B** většinou tvoří jako v našem případě matici bidiagonální nebo tridiagonální. Princip je velice podobný QR rozkladu, dokonce se dají používat i stejné metody, které jsou však lehce poupraveny. Snahou není získat horní trojúhelníkovou matici jako u QR rozkladu, ale matici bidiagonální. To je provedeno postupným vynulováním pod hlavní diagonálou a následném vynulování nad superdiagonálou. Postup je vidět na následujícím zdrojovém kódu. Je logické, že tak musí vzniknout dva vektory Householderova zrcadlení **v**, **u** v každém kroku.

```
%Householderovo zrcadleni na bidiagonalni matici
function [A, V, U] = upbid(obj,A)
    [m, n] = size(A);
    if m ~= n
        error('Matrix must be square')
    end
    if tril(triu(A),1) == A
        V = eye(n-1);
        U = eye(n-2);
    end
    V = []; U = [];
    for k=1:n-1
        x = A(k:n,k);
        v = SVD.Housv(x);
        l = k:n;
        A(l,l) = A(l,l) - 2*v*(v'*A(l,l));
        v = [zeros(k-1,1);v];
        V = [V v];
        if k < n-1
            x = A(k,k+1:n)';
            u = SVD.Housv(x);
            p = 1:n;
            q = k+1:n;
            A(p,q) = A(p,q) - 2*(A(p,q)*u)*u';
            u = [zeros(k,1);u];
            U = [U u];
        end
    end
end
```

```
function u = Housv(x)
    % Householder reflection unit vector u from the vector x.
    m = max(abs(x));
    u = x/m;
    if u(1) == 0
        su = 1;
    else
        su = sign(u(1));
    end
    u(1) = u(1)+su*norm(u);
    u = u/norm(u);
    u = u(:);
end
```

Následně je možné z matice **V**, **U** vytvořit podobně jako u QR ortogonalizace ortogonální matice **P** a **Q**. Přičemž matice **P** je ve tvaru (37)

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & p_{i,j} & p_{i,j} \\ 0 & p_{i,j} & p_{i,j} \end{bmatrix} \tag{37}$$

Poté je možné řešit soustavu rovnic (38) například zpětnou substitucí

$$\mathbf{Bx} = \mathbf{PQ}^{\mathrm{T}}\mathbf{b}. \tag{38}$$

```
function Q = Housprod(V)
     [m, n] = size(V);
      Q = eye(m)-2*V(:,n)*V(:,n)';
      for i=n-1:-1:1
          Q = SVD.Houspre(V(:,i),Q);
      end
end

function P = Houspre(u, A)
     [n, p] = size(A);
      m = length(u);
      if m ~= n
          error('Dimensions of u and A must agree')
      end
      v = u/norm(u);
      v = v(:);
      P = [];
      for j=1:p
          aj = A(:,j);
          P = [P aj-2*v*(v'*aj)];
      end
end
```

SVD rozklad je však obecně považován za výpočetně nejsložitější metodu a pokud je naším cílem rychlost, nebude určitě naší volbou. Její největší výhoda tkví především v její stabilitě. [5] [10] [11]

4 Analýza algoritmů

4.1 Definice složitosti algoritmu

Asymptotická složitost je způsob klasifikace počítačových algoritmů. Určuje náročnost algoritmu tak, že zjišťuje jakým způsobem se bude chování algoritmu měnit v závislosti na změně velikosti vstupních dat. Zapisuje se pomocí „velké O notace“, jako $O(f(N))$. Tato veličina nejčastěji slouží jako kritérium pro porovnávání kvality algoritmu pro její pozdější praktické použití. Náročnost algoritmu musí být menší než $A + B * f(N)$, kde A a B jsou vhodně zvolené konstanty a N je veličina popisující velikost vstupních dat. Je tedy zanedbána multiplikativní i aditivní konstanta. Asymptotická složitost se dá vyjádřit křivkou z $O(f(N))$, která bude zastřešovat skutečnou složitost algoritmu.

Jsou rozeznávány dva typy složitosti: časová, prostorová. Časová odpovídá počtu vykonaných operací za jednotku času. Podle doby, za kterou se vykoná jedna operace, je možné tuto hodnotu přepočítat na čas. Prostorová odpovídá potřebám na velikosti datové paměti. Obě kritéria jsou společně svázána, pokud se snažíme optimalizovat časovou složitost, většinou to klade větší nároky na paměť.

Z následující tabulky je vidět, jak se zvyšování asymptotické složitosti s velikostí vstupních struktur rapidně projevuje na době výpočtu (Tabulka 2).

Tabulka 2: *Doba provádění algoritmu v závislosti na velikosti vstupních dat (1 mil. operací za s)*

	N						
f(N)	20	40	60	80	100	500	1000
N	20 µs	40 µs	60 µs	80 µs	100 ms	0,5 ms	1 ms
N log N	86 µs	0,2 ms	0,35 ms	0,5 ms	0,7 ms	4,5 ms	10 ms
N^2	0,4 ms	1,6 ms	3,6 ms	6,4 ms	10 ms	0,25 s	1 s
N^3	8 ms	64 ms	0,22 s	0,5 s	1 s	125 s	17 min
2^n	1 s	11,7 dní	36 tis. let				
N!	77 tis. let						

Cílem vytvoření efektivního algoritmu je vytvořit program s co nejmenší složitostí. Pro malá N, jak je známo z teorie algoritmizace platí, že rozdíly mezi časem výpočtu

jednotlivých tříd složitostí budou malé, ale pokud je uvažována větší vstupní struktura, jsou nároky na čas potřebný pro výpočet diametrálně větší. Jak je vidět z tabulky, tak pro algoritmy s vysokou složitostí (větší než kvadratické) není řešením zvyšování výkonu výpočetní jednotky, ale optimalizace kódu a snaha o snížení výpočetní složitosti. [13]

4.2 Generátor zadání

Pro úspěšnou analýzu algoritmů a jejich porovnání je zapotřebí generátoru náhodných čísel, ze kterých se poté sestaví matice koeficientů a vektor pravé strany soustavy lineárních rovnic. Jelikož nejsou známé žádné informace o vstupních datech pro řešení soustav rovnic, byly vytvořeny dva generátory. První z nich generuje čísla rovnoměrného rozdělení pravděpodobnosti (39)

$$A_{ii} = a + (b - a) * U, \tag{39}$$

kde U jsou vygenerovaná čísla programem MATLAB v rozmezí od 0 do 1, konstanty a, b jsou meze generovaných čísel.

Další generátor generuje čísla podle normálního rozdělení pravděpodobnosti podle vzorce (40)

$$A_{ii} = X * \text{směrodatná odchylka} + \text{střední hodnota}, \tag{40}$$

kde X se spočítá podle (40)

$$X = \sqrt{-2 \ln U}\, \cos(2\pi V), \tag{41}$$

kde U a V jsou opět vygenerovaná čísla programem MATLAB v rozmezí od 0 do 1.

V následujícím kódu je zobrazen algoritmus pro generování náhodných čísel. [21]

```
% unirmniho rozlozeni pravdepodobnosti
function [unifRoz,unifRozB] = vygenerujUniformRP(obj,a,b)
      %a, b jsou okrajove body intervalu
       U = rand(obj.N,obj.N);
       unifRoz = a + (b - a) * U;
       U = rand(obj.N,1);
       unifRozB = a + (b - a) * U;
end
```

```
% normalniho rozlozeni pravdepodobnosti
function normRoz =
      vygenerujNormRP(obj,stredniHodnota,smerodatnaOdchylka)
       U = rand(obj.N,obj.N);
       V = rand(obj.N,obj.N);
       normRoz = (sqrt(-2 * log(U)).* sin(2 * pi  * V)) *
            smerodatnaOdchylka + stredniHodnota;
end
```

4.3 Obecné poznatky ke složitosti analyzovaných algoritmů

K výše uvedeným algoritmům je možné v odborné literatuře najít obecné vztahy pro jejich asymptotickou časovou složitost. Pro řešení soustav rovnic je podle [5] zapotřebí přibližně $O(N^3)$ operací, pro každý algoritmus jsou však drobné rozdíly. Ale pokud uvažujeme nasazení algoritmu v DSP je tato hodnota pouze velice orientační. Je zapotřebí také uvažovat o jaké operace se jedná a jejich řešení v DSP. I když se na první pohled může zdát, že daný algoritmus je efektivní, může obsahovat například dělení, odmocniny, goniometrické a logaritmické funkce, které je velice obtížné řešit a zvyšují výpočetní náročnost. Proto je snahou co nejvíce využívat možnosti procesoru, které jsou hardwarově podporovány, tím se podstatně zvýší výpočetní výkon (Kapitole 1).

Aby bylo zjištěno jak jsou jednotlivé metody skutečně náročné vzhledem k použité technologii DSP, byly s dříve uvedenými algoritmy (Kapitola 3) provedeny experimenty v programu MATLAB a důkladně prozkoumány jednotlivé kroky. Následně byla odhadnuta složitost algoritmu v závislosti na použitých operacích, jejich četnosti a vlastností procesoru.

4.4 Modifikovaný Gram-Schmidtův algoritmus, QR rozklad

Algoritmus obsahuje jednu hlavní a jednu vnořenou smyčku. Hlavní smyčka je závislá na velikosti matice, takže bude probíhat N krát. V tomto cyklu jsou dvě kritické operace a to výpočet normy vektoru, která se vypočítá podle vzorce (42)

$$\|\mathbf{x}\| := \sqrt{x_1^2 + \cdots + x_n^2}. \tag{42}$$

kde se vyskytuje odmocnina. Ostatní operace jako umocnění a sčítání se dají velice snadno v DSP paralelizovat a spočítat za velmi krátkou dobu. A druhý bude výpočet dělení vektoru skalárem. Tyto operace se však vyskytuji pouze N krát, ale hlavní výpočet-

ní náročnost stojí na vnořeném cyklu, který bude proveden N^2. V tomto cyklu však nejsou žádné kritické operace, obsahuje 2x součin dvou matic a odečtení dvou matic velikosti N. Tyto operace jsou však ideální pro zpracování v DSP.

Pokud je uvažován upravený kód (Kapitola 3.4.1), na první pohled je zřejmé, že bude efektivnější pro DSP. Neobsahuje totiž vnořenou smyčku, ale operace jsou převedeny na práci v maticovém tvaru, který bude daleko více vyhovovat procesoru. Obsahuje jednu smyčku o N cyklech s výpočtem normy, dělením vektoru skalárem, dvě maticové násobení a jeden maticový rozdíl.

```
for k = 1:n
    R(k,k) = norm(Q(:,k));
    Q(:,k) = Q(:,k)/R(k,k);
     R(k,k+1:n) = Q(:,k)'*Q(:,k+1:n); %vektor x matice
     Q(:,k+1:n) = Q(:,k+1:n) - Q(:,k)*R(k,k+1:n);
 end
```

4.5 Householderova transformace, QR rozklad

Tato metoda se skládá ze dvou po sobě následujících cyklech, každý má přitom N kroků. První obsahuje dělení vektoru skalárem, 2x násobení matice o velikosti N s vektorem a jedno násobení dvou čísel. Ve druhém cyklu je výpočet podle vzorce (34), který obsahuje jedno dělení vektoru skalárem, roznásobení matic a její sestavení.

```
for k = 1:n
    r(k,k) = norm(A(1:m, k));
    if r(k,k) == 0
        break;
    end
    q(1:m, k) = A(1:m, k) / r(k,k);
    for j = k+1:n
        r(k, j) = dot(q(1:m, k), A(1:m, j));
        A(1:m, j) = A(1:m, j) - r(k, j) * q(1:m, k);
    end
 end
```

4.6 LU rozklad

Tento algoritmus pro rozklad matice obsahuje dva cykly, z toho jeden je vnořený do druhého. Vnější obsahuje velice málo operací, které nejsou zvláště náročné a dají se v DSP řešit pomocí bitových posunů. Vnitřní cyklus, kde dochází k eliminaci ob-

sahuje N^2 operací. Dochází při tom i k dělení dvou čísel, což není vhodná operace pro procesor a násobení dvou vektorů.

```
for i = 1 : nrow - 1
    t =  min ( find ( abs(A(pvt(i:nrow),i)) ==
       max(abs(A(pvt(i:nrow),i))) ) + i-1 );
    if ( t ~= i )
       temp = pvt(i);
       pvt(i) = pvt(t);
       pvt(t) = temp;
    end;
    for j = i+1 : nrow
        m = -A(pvt(j),i) / A(pvt(i),i);
        A(pvt(j),i) = -m;
        A(pvt(j), i+1:nrow);
        A(pvt(j), i+1:nrow) = A(pvt(j), i+1:nrow) + m *
             A(pvt(i), i+1:nrow);
    end;
end;
```

4.7 SVD rozklad

Pro základ algoritmu je použit již popsaný QR rozklad pomocí Householderovy transformace. Pro výpočet ortogonálních matic je zvolen algoritmus, kde je potřeba N^2 operací násobení vektorů. Jelikož jsou vypočteny dvě ortogonální matice je potřeba tento cyklus provést dvakrát. V SVD algoritmu se vyskytuje minimum operací, které nejsou vhodné pro DSP, ale celková náročnost je mnohem větší než u předchozích algoritmů.

4.8 Výsledky analýzy

Z předchozích měření a důkladném zkoumání výše uvedených možností se jeví jako nejefektivnější algoritmy QR rozkladu. Metody založené na tomto principu všeobecně vynikají lepši stabilitou než LU rozklad, nebo Gaussova eliminace. Jsou schopny eliminovat i některé problémy způsobené průběžným zaokrouhlováním a nepodmíněností matice. V tomto směru nejvíce vyniká SVD, ale její výpočetní i paměťové nároky jsou obrovské a nehodí se pro aplikaci v DSP.

Householderův QR rozklad a Modifikovaný Gram-Schmidtův (MGS) rozklad s upraveným kódem pro DSP jsou si velice podobné svojí efektivností algoritmu. Hou-

seholder však obsahuje o něco více operací, které budou zatěžovat procesor, ale vyniká větší stabilitou. MGS má většinu svých operací v maticové formě, k tomu se velice hodí využit funkcí od Analog Devices pro práci s maticemi.

Následující tabulka (Tabulka 3) udává náročnost jednotlivých algoritmů v PC změřených profilerem v programu MATLAB. Vstupem byla čtvercová matice o velikosti N = 288. Měřené výpočty neobsahují zpětnou substituci, ale pouze faktorizaci dané metody (QR, SVD, LU). Doba zpětné substituce byla oproti faktorizaci zanedbatelná a průměrně trvala 7 ms.

Tabulka 3: Efektivita algoritmu v programu MATLAB

Metoda	Čas [s]
QR Modifikovaný Gram-Schmidt upravený pro DSP	0.256
QR Modifikovaný Gram-Schmidt	2.812
QR Householder	0.818
LU	1.083
SVD s Householderovou ortogonalizací	120.647

Tyto naměřené hodnoty nejsou přímo vypovídající pro aplikování algoritmu v DSP, ale přesto mohou pomoci s orientačním zhodnocením. Je vidět, že nejlepších výsledků bylo dosaženo za pomocí QR rozkladů a to především MGS upraveným pro signálový procesor. Velice dobře si také vedla metoda Householderova. SVD dekompozice dopadla podle očekávání nejhůře a i přes dobrý výkon testovacího PC (Intel Core2-duo 1.6GHz 64-bitový MATLAB) trval výpočet skoro dvě minuty. Těmito výsledky byl potvrzen předchozí rozbor algoritmů a pro aplikování v DSP byla vybrána metoda QR rozkladu s modifikovaný Gram-Schmidtovým algoritmem upraveným pro DSP.

5 Algoritmus pro signálový procesor

Pro práci s DSP bylo snahou co nejvíce využít knihovnu run-time library, s co nejmenším množstvím pomocných datových manipulací, která by měla zaručovat co možná největší využití potenciálu procesoru. Celý kód je napsán v programovacím jazyce C. Algoritmus je zastřešen funkcí, kterou je možné připojit k jakémukoli programu v procesoru TigerSHARC.

5.1 Popis funkce reseniSoustavy()

Vstupem funkce je čtvercová matice koeficientů **A**, vektor pravé strany **b** a velikost matice N. Výstupem bude vektor řešení soustavy rovnic **X**. Tato funkce nevyžaduje jako vstupní hodnoty data, ale pouze pointer na první číslo matice. Data jsou v paměti uložena po řádcích, a proto je možné z velikosti matice N dopočítat začátky konkrétních řádků. Výstupní vektor je uložen na N pozic od počáteční adresy proměnné X. [18] [19]

```
void reseniSoustavy(pA, pb, N, pX);

float *pA; /* Pointer na matici A[][] */
float *pb; /* Pointer na vektor b[] */
int N; /* Počet radku matice A[][] */
float *pX; /* Pointer na vystupni matici X[][] */
```

Způsob využití této funkce znázorňuje následující kód:

```
include <stdio.h>
#include "reseniSoustavy.h"

/* velikost matice */
#define M 3

/* definice matic pro QR rozklad */
section ("data1") float A[M*M] = {
          13.3624, 3.41897, 29.7283,
          32.4345, 65.7153, 95.7367,
          93.6374, 46.3307, 24.8074 };
```

```
section ("data1") float b[M] = {
          10.2059,
          46.8854,
          1.92390
};    //prava strana

section ("data1") float X[M*M];

int main( int argc, char *argv[] )
{
     float *pA  = (float *) &A;   // Vytvoreni pointeru na A[][]
     float *pX = (float *) &X;    // Vytvoreni pointeru na vX[][]
     float *pb = (float *) &b;    // Vytvoreni pointeru na b[]
     reseniSoustavy(pA,pb,M,pX); //reseni soustavy rovnic
     return 0;
}
```

V první části jsou vytvořeny proměnné typu pole a uloženy do nich data (matice koeficientů **A**, pravá strana **b** a prázdný vektor výsledků **X**). V hlavním programu jsou vytvořeny pointry na tyto proměnné a zavolána funkce reseniSoustavy(), která vypočte řešení a uloží ho do *X*.

5.2 Program pro řešení soustav lineárních rovnic

5.2.1 QR rozklad

Algoritmus se skládá ze dvou základních částí. První část je QR rozklad matice (druhá část zpětná substituce), kterou si nyní popíšeme (z kódu byly pro přehlednost odstraněny komentáře, jsou však součástí přílohy).

```
transpmf (pA,N,N,pQ);
for (k = 0; k < N; k++)
{
    memcpy(pvX,pQ + k*N,N);
    *(pR + k*N + k) = vecdotf (pvX,pvX,N);
    *(pR + k*N + k) = sqrtf (*(pR + k*N + k));
    for (i = 0; i < N; i++)
    {
        *(pQ + k*N + i) = *(pQ + k*N + i) / *(pR + k*N + k);
```

```
    }
    if (k + 1 < N)
    {
        matmmltf (pR + k*N + k + 1, N - k, 1, pQ + k*N, N, pv
        memcpy(pX,pQ + k*N + N, N*(N - k - 1));
        matmsubf (pX,pvX,N - k - 1, N, pX);
        memcpy (pQ + k*N + N, pX, N*(N - k - 1));
    }
}
```

Základem algoritmu je cyklus s předem známou délkou, která bude rovna N, kde N je velikost matice. Před samotným výpočtem rozkladu je provedena transpozice vstupní matice **A** a výsledek uložen do **Q**. Důvodem této transpozice, která není součástí algoritmu pro MATLAB je, že všechny výpočetní operace pracují se sloupci. Běžné uložení matice v DSP je takové, že jsou řádky postupně uloženy po sobě, ale kdyby se pracovalo se sloupci, muselo by se po každém členu vždy přeskočit N číslic. Po provedení transpozice je k dispozici v paměť s čísly uspořádanými po sloupcích po sobě. Pokud tyto kroky nebudou provedeny, značně by se zvýšila složitost výpočtu.

Po provedení nezbytné úpravy paměti následuje cyklus *for* o N krocích. První tři instrukce spočítají normu vektoru k-tého sloupce z matice **Q**. První instrukce *memcpy* zkopíruje vektor do pomocné proměnné vX. Tato kopie je provedena, protože následující funkce nedokáže pracovat s jakoukoliv adresou (jak bylo dříve zmíněno, proměnné jsou uspořádány do čtveřic). Následující příkaz roznásobí členy vektoru a sečte je, poté je pomocí funkce *sqrt* provedena odmocnina. Výsledkem je norma vektoru, která je uložena do **R** na k-tou pozici na hlavní diagonále.

Dalším krokem je výpočet normy vektoru, jsou vyděleny jednotlivé členy k-tého sloupcového vektoru matice **Q**. Pro tuto operaci neexistuje funkce, tak je ji potřeba složit z dělení čísla jiným číslem a cyklem, který tuto operaci postupně provede na všechny členy vektoru.

V posledním cyklu QR rozkladu nejsou potřeba provádět následující operace, proto se zde nachází podmínka, která nám tuto vlastnost zaručí. Další prováděnou funkcí je násobení dvou matic. Násobí se k-tý sloupec transponované matice (vektor) **Q'** s maticí **Q**, kde jsou vybrány sloupce od pozice k + 1 do velikosti matice N. Velikost výsledku tohoto násobení přesně odpovídá velikosti pravé strany od hlavní diagonály k-tého

řádku matice **R**. Tímto krokem je dopočítán první řádek této matice, v příštích krocích postupně vznikne trojúhelníková matice.

V poslední části je výsledek **R** (řádkový vektor od hlavní diagonály doprava) vynásoben s k-tým řádkem matice **Q**. Výsledek tohoto násobení je odečten od matice **Q** sloupce od pozice k + 1 do velikosti matice N. Tímto je aktualizována matice **Q** a následuje další iterace. Příkazy *memcpy* připravují data pro odčítaní, protože funkce *matsub* potřebuje vždy čísla zarovnaná po čtveřicích, toho však není vždy možné dosáhnout (záleží na velikosti vstupní matice, zda je dělitelná čtyřmi).

Po dokončení N iterací cyklu QR rozkladu zůstane v proměnné **Q** ortogonální matice k matici **A** a v **R** horní trojúhelníková matice.

5.2.2 Zpětná substituce

V následujícím algoritmu je provedena zpětná substituce pro výpočet řešení **X**.

```
matmmltf (pQ,N,N,pb,1,pvX);
*(pX + N - 1) = *(pvX + N - 1) / *(pR + N*N - 1);
for (k = N - 1; k > 0; k--) {
    matmmltf (pX + k, 1, N - k, pR + k*N - N + k, 1, pX + k -
       1);
    *(pX + k - 1) = *(pvX + k - 1) - *(pX + k - 1);
    *(pX + k - 1) = *(pX + k - 1) / *(pR + k*N - 1 - N + k);
}
```

V prvním kroku je vynásobena matice **Q** s vektorem pravé strany **b**, tímto násobením je řešena pravá strana rovnice (28). Jelikož **R** je horní trojúhelníková matice a pravá strana je vektor, je možné x_1 spočítat pouhým vydělením dvou čísel r_1/b_1 . V následujícím kroku je x_1 vektorově vynásobena s pravou stranou od hlavní diagonály k-tého řádku matice **R**. Výsledkem je číslo, které je odečteno od b_k a znovu je možné řešit rovnici o jedné neznáme pomocí dělení. Výsledkem je x_k člen řešení, po N – 1 krocích bude vektor **X** obsahovat řešení soustavy rovnic.

5.3 Alternativní řešení soustav lineárních rovnic

Knihovna VisualDSP++ DSP library obsahuje funkci *matinvf*, která vypočítá inverzní matici pomocí Gauss-Jordanovi eleminace. Pokud je známa inverzní matici, je možné snadno spočítat řešení soustavy lineárních rovnic pomocí vztahu (6). V následujícím výpise je zobrazen kód pro ADSP-TS201S.

```
matinvf (pA,N,pQ);
transpmf (pQ,N,N,pR);
matmmltf (pb,1,N,pR,N,pX);
```

V prvním řádku se vypočítá inverzní matice k **A** a uloží do proměnné Q. V následujícím kroku je provedena transpozice matice **Q**. Poté je vynásobena vektorem pravé strany rovnice **b**. Výsledný vektor je uložen do proměnné X.

V následující kapitole je tato metoda porovnána s řešením pomocí QR rozkladu.

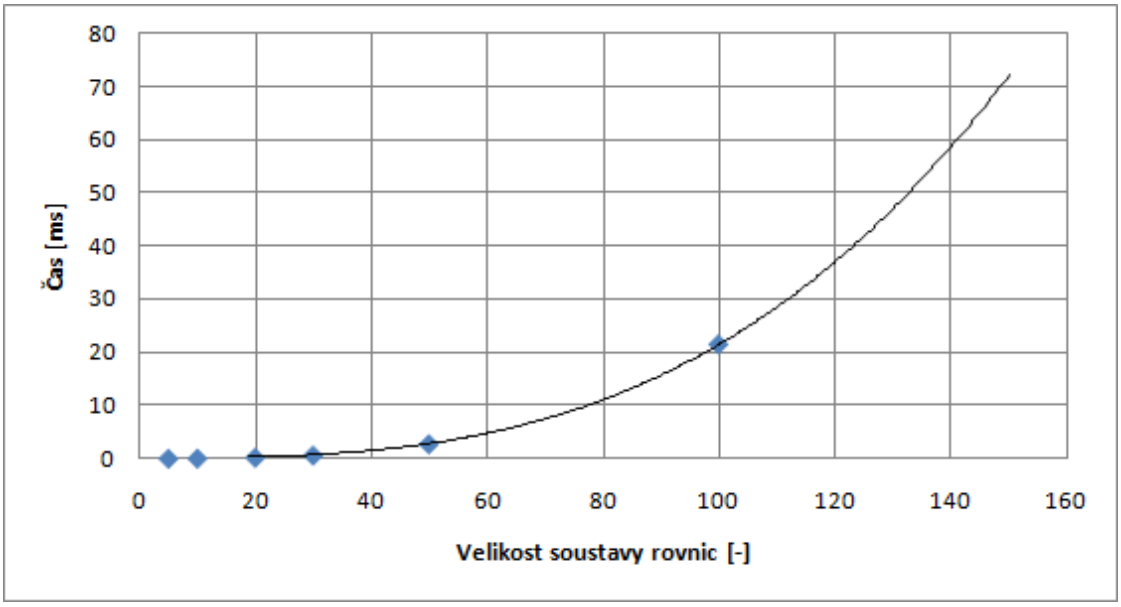
6 Testy vytvořeného SW

6.1 Měření rychlosti

Po úspěšné implementaci algoritmu do procesoru byla provedena reálná měření, která mají za cíl zjistit použitelnost tohoto řešení v praxi. Následující tabulka zobrazuje výsledky měření, pro různé velikosti soustavy rovnic (Tabulka 4). Výsledky jsou pro řešení soustav rovnic s reálnými čísly, pokud je uvažováno v komplexním oboru, je potřeba velikost N vydělit dvěma.

Tabulka 4: Řešení různých soustav lineárních rovnic v DSP (v závorce je velikost matice v oboru komplexních čísel)

N	5	10 (5)	20 (10)	30 (15)	50 (25)	100 (50)
instrukce	3600	17390	113500	364205	1623544	12815799
Čas	6 µs	29 µs	189,5 µs	608 µs	2,7 ms	21,4 ms



Obr. 9: Odhad doby výpočtu velkých soustav rovnic.

Větší soustavy rovnic nebylo možné nasimulovat, protože maximální velikost souvislé paměti v procesoru postačuje pouze pro čtvercovou matici o velikosti N = 113. Pro řešení větších rovnic by bylo nutné naprogramovat režii, která by se starala o adresování i do jiných bloků paměti. Z naměřených hodnot bylo zjištěno pomocí mocninné regrese, že časová asymptotická složitost je přibližně dána vztahem (43)

$$y = (6E{-}5)x^{2,748}. \tag{43}$$

Z této rovnice je patrné, že původní tvrzení $O(N^3)$ bylo správné. Odhadem za pomoci regrese je možné určit, že doba řešení největší předpokládané soustavy rovnic o N = 288 bude trvat odhadem 350 ms. Tato doba je velice podobná času, který stráví program MATLAB řešením této soustavy rovnic s dříve představenou konfigurací osobního počítače.

Na následujícím obrázku je znázorněno procentuální využití jednotlivých funkcí algoritmu vůči celkovému počtu vykonaných instrukcí (Obr. 10). Největším podílem je zastoupena funkce *matmmltf*, to znamená, že procesor při řešení soustavy rovnic stráví takřka 87% času vzájemným násobením dvou matic. Tato hodnota je velice pozitivní, protože zmíněnou operaci dokáže procesor velice efektivně řešit pomocí jednotky MUL. Funkce *memcpy*, je vykonávána 8% času, jedná se o pouhé kopírování dat v paměti. Funkce *vecvsubf (odčítání)*, která se vykonává 3% z celkového času, je analogií ke sčítaní. Ostatní funkce jako výpočet normy dělení a operace pro obsluhu procesoru jsou minoritní.

Histogram	%	Execution Unit
	86.32%	__matmmltf
	8.24%	____memcpyDD
	2.76%	__vecvsubf
	2.05%	reseniSoustavy(float*, float*, int, float*)
	0.35%	__vecdotf
	0.20%	__sqrtf
	0.03%	__matmsubf
	0.02%	______system_start
	0.01%	_mi_initialize
	0.01%	__getargv
	0.01%	main(int, char**)

Obr. 10: Vytížení použitých funkcí z VisualDSP++ run-time library.

Před kompilací vytvořeného projektu pro DSP byla nastavena automatická softwarová optimalizace kódu s důrazem na časovou úsporu. Po změření a porovnání výsledků bylo tímto módem dosaženo časové úspory výpočtu soustavy rovnic zhruba o 10%.

V následující tabulce je porovnáno dosažených rychlostí pomocí QR rozkladu a funkce *matinvf* z DSP library. Metodou založenou na QR rozkladu bylo však dosaženo o něco lepších výsledků (Tabulka 5).

Tabulka 5: Porovnání dosažených výsledků QR rozkladu s funkcí matinvf

Metoda / velikost matice (N)	10	30	50
QR rozklad	18390	364205	1623544
Run-time library (*matinvf*)	19195	417654	1900692

Z předchozí tabulky je vidět, že u QR rozkladu je dosaženo časové úspory zhruba 15% oproti funkci z DSP library, která je také maximálně optimalizovaná pro daný procesor.

6.2 Testy přesnosti výpočtů

Další důležitou částí analýzy je porovnání přesností dosažených výsledků. Jako referenční hodnoty jsou použity výsledky z programu MATLAB, které jsou vypočteny za pomoci datového typu double (64 b s plovoucí čárkou). V DSP byl použit single (32 b s plovoucí čárkou). Je zřejmé, že výsledky ze signálového procesoru budou vypočteny s určitou chybou způsobenou zaokrouhlením. Metoda pro řešení soustav rovnic založená na QR rozkladu, by však měla být proti těmto chybám odolná. V praxi to znamená, že v každé iteraci je počítáno s čísly, které ještě nejsou zatíženy chybou z předchozích výpočtů.

Pro dosažení konkrétních výsledků byla vygenerována stejná data v datovém formátu single. Jako referenční hodnota byla použita čtvercová matice o rozměru N = 30. Tento formát byl zvolen proto, aby nedocházelo už při načítaní čísel do DSP k jejich zaokrouhlení. Data byla z programu MATLAB vyexportována do souboru a vložena do paměti procesoru. Poté byly provedeny výpočty soustav lineárních rovnic. Výsledky byly vyexportovány do samostatných souborů a v programu Microsoft Excel porovnány. Porovnávané hodnoty jsou součástí přílohy. Z dosažených výsledků plyne předchozí tvrzení, všechna čísla řešení soustav lineárních rovnic jsou zatížena stejnou chybou. K rozdílným hodnotám dochází až na pátém místě mantisy datového formátu s plovoucí řádovou čárkou single (uvažováno v desítkové soustavě, kde se mantisa skládá z šesti čísel). Procentuální vyjádření průměrné chyby je rovno $5{,}56 \cdot 10^{-4}$%.

6.3 Zhodnocení dosažených výsledků a návrhy pro vylepšení

Z naměřených hodnot je patrné, že zvolená metoda je použitelná v signálovém procesoru a je dosahováno velice přesných a rychlých výpočtů. Tím, že se podařilo dosáhnout takřka 86% využití funkce *matmmltf* vůči ostatním operacím algoritmu, dá se předpokládat, že zvolená metoda založená na QR rozkladu, se výborně hodí pro aplikování v tomto signálovém procesoru. Přesto existuje několik možností, které by mohly přinést ještě o něco lepší výsledky s ohledem na časovou náročnost. Je možné přepsat dílčí části kódu do jazyka symbolických adres, tím by se dalo využít správného rozmístění dat v paměti a lepší paralelizace přenosu po sběrnicích. Další možností je, navrhnout lépe část odčítaní a vyhnout se funkci *memcpy*, tím by se mohlo ušetřit až 8% času, jak je opět zřejmé z předchozího srovnání (Obr. 10). Vyžadovalo by to přepsat následující funkci odčítaní do jazyka symbolických adres.

Jelikož jsou použité funkce vytvořené výrobcem, je jejich optimalizace pro procesor na nejvyšší možné úrovni, proto je otázkou, zda by zmíněné úpravy měly pozitivní účinek.

Pro řešení komplexních soustav rovnic se využívá dříve zmíněného rozšíření matice a poté se řeší soustava lineárních rovnic v oboru reálných čísel. Tento algoritmus není součástí zmíněné funkce *reseniSoustavy()*. Lze předpokládat, že data určená k řešení soustavy rovnic budou získána z externí komunikační sběrnice, a proto je bude možné do paměti procesoru rovnou uložit v požadovaném formátu (Kapitola 2.4). Stejným způsobem by bylo možné odstranit i prvotní transpozici v algoritmu, která má zaručit vhodnější uspořádání dat v paměti procesoru.

Závěr

Výsledkem této diplomové práce je souhrnný popis metod pro řešení soustav lineárních rovnic v oboru komplexních čísel, rozbor chyb, které mohou vznikat zaokrouhlováním a špatnou podmíněností soustavy při aplikaci v číslicových systémech.

Před implementací do signálového procesoru bylo potřeba provést analýzu, která se skládala z důkladného prozkoumání jednotlivých kroků metody pro řešení soustavy lineárních rovnic. Při těchto rozborech již byly brány v úvahu specifické vlastnosti signálového procesoru. Snahou bylo zvolit takovou variantu, která bude mít co nejvíce dílčích kroků podporovaných hardwarovou strukturou procesoru a celkově nejmenší asymptotickou časovou složitost. Pro tyto účely byly v programu MATLAB zanalyzovány metody a spočítány jednotlivé hardwarově podporované i nepodporované operace. Na základě těchto výsledků bylo rozhodnuto o metodě založené na QR rozkladu matice pomocí modifikovaného Gram-Schmidtova algoritmu.

Pro samotnou implementaci bylo využito vysoce optimalizovaných knihoven od firmy Analog Devices. Zároveň se podařilo vybrat takové funkce, které mají hardwarovou podporu, a tím bylo dosaženo takřka maximálního výkonu procesoru.

Hlavním přínosem diplomové práce je tedy naprogramovaný efektivní algoritmus pro zpracování soustav lineárních rovnic. Ten bude využit v signálovém procesoru, umístěném v radaru na měření výšky sněhu. Nedílnou součástí této práce je také rozbor metod s praktickými ukázkami v programu MATLAB a jejich zhodnocení v závislosti na aplikování v DSP.

Diplomová práce splnila všechny požadavky původního zadání a umožňuje v budoucnu její případné další rozvíjení. Také může eventuálně posloužit jako základ pro řešení soustav lineárních rovnic na podobných číslicových systémech.

Zdroje

[1] RALSTON, Anthony. *Základy numerické matematiky*. 2. Praha : Československá akademie věd, 1978. Řešení soustav lineárních rovnic, s. 636, 21-054-78

[2] Kategorie:Lineární algebra. *In Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, [cit. 2010-08-09]. Dostupné z WWW: <http://cs.wikipedia.org/wiki/Kategorie:Lineární_algebra>.

[3] Komplexní číslo. *In Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, [cit. 2010-08-10]. Dostupné z WWW: <http://cs.wikipedia.org/wiki/Komplexní_číslo>.

[4] J. J. Buckley, *Fuzzy Complex Numbers : Proceeding of ISFK*, Guangzhou, China, 1987, 697-700.

[5] H. PRESS, *William, et al. Numerical Recipes in C : The Art of Scientific Computing*. Second Edition. New York : CAMBRIDGE UNIVERSITY PRESS, 1992. 964 s.

[6] Golub, Gene H.; Van Loan, Charles F. (1996), *Matrix Computations* (3rd ed.), Johns Hopkins, ISBN 978-0-8018-5414-9.

[7] *Gram-Schmidt Orthogonalization* [online]. 2007 [cit. 2010-08-11]. The University of British Columbia. Dostupné z WWW: <http://www.ugrad.cs.ubc.ca/~cs402/handouts/handout08.pdf>.

[8] ZEMÁK, Petr. *Vybrané partie z aplikované matematiky : QR- ROZKLAD* [online]. Brno, 27 s. Masarykova univerzita Brno. Dostupné z WWW: <http://www.math.muni.cz/~xzemane2/QR-rozklad_[seminarni_prace_-_Petr_Zemanek].pdf>.

[9] *Householder Triangularization* [online]. 2007 [cit. 2010-08-11]. The University of British Columbia. Dostupné z WWW: <http://www.ugrad.cs.ubc.ca/~cs402/handouts/handout10.pdf>.

[10] NEUMAN, Edward. *Numerical Linear Algebra : SVD*. Southern Illinois University at Carbondale. 44 s.

[11] KOTAS, Petr. *Efektivní SVD a jeho využití při zpracování biometrických údajů*. Ostrava, 2009. 42 s. Diplomová práce. Technická univerzita Ostrava.

[12] BJORCK, Ake . *Gram–Schmidt Orthogonalization : 100 Years and More*. Shanghai : 2010. 50 s.

[13] VRANÝ, Jiří. [online]. 2008 [cit. 2010-08-23]. A*lgoritmy a datové struktury.* Dostupné z WWW: <http://www.nti.tul.cz/~vrany/ads/1_uvod.pdf>.

[14] HÁJEK, Martin. *Digitální signálové procesoryl.* Pardubice : Univerzita Pardubice, 2009. Studijní materiály k přednáškám.

[15] *TigerSHARC ADSP-TS201S : Data sheet*. Norwood, U.S.A. : Analog Devices, 2006. 48 s.

[16] HÁJEK, Martin. *Možnosti řešení soustavy lin. rovnic na DSP TigerSHARC.* Pardubice : Univerzita Pardubice, 2009. 2 s.

[17] ADSP-TS201 T*igerSHARC® Processor : Hardware Reference*. Revision 1.1. Norwood, U.S.A. : Analog Devices, 2004. 524 s.

[18] ADSP-TS201 T*igerSHARC® Processor : Programming Reference*. Revision 1.1. Norwood, U.S.A. : Analog Devices, 2005. 896 s.

[19] *VisualDSP++ 5.0 : C/C++ Compiler and Library Manual for TigerSHARC® Processors*. Revision 4.1. Norwood, U.S.A. : Analog Devices, 2008. 565 s.

[20] CALDWELL, Andrew; KOKALY-BANNOURAH, Maikel. S*HARC® DSPs to TigerSHARC® Processors Code Porting Guide*. Norwood, U.S.A. : Analog Devices, 2004. 64 s.

[21] KÁVIČKA, Antonín. *Pokročilé techniky modelování a simulace.* Pardubice : Univerzita Pardubice, 2009. Studijní materiály k přednáškám.

Přílohy

Obsah přiloženého CD-ROM disku

Na CD-ROM disku jsou tři složky. První obsahuje zdrojové kódy pro program MATLAB, kde jsou nasimulovány různé metody řešení soustav lineárních rovnic. Ve druhé složce se nachází projekt pro VisualDSP++, který obsahuje všechny potřebné funkce pro řešení soustav rovnic na DSP. Poslední obsahuje vypracovanou diplomovou práci v elektronické podobě, ve formátu PDF.

Srovnání dosažených přesností výsledků

Matlab	DSP
-0,322349000	-0,322341000
0,245674600	0,245659000
-0,779376400	-0,779362000
-0,463898400	-0,463894000
-0,925534500	-0,925508000
0,336643800	0,336637000
-0,288178100	-0,288159000
0,285017300	0,285012000
-0,631477100	-0,631462000
-0,951326400	-0,951299000
0,817924000	0,817921000
-0,509700400	-0,509690000
1,207747000	1,207720000
0,404998600	0,404996000
-0,170935800	-0,170934000
0,439111700	0,439099000
-0,348871800	-0,348864000
0,497174900	0,497178000
0,678871800	0,678852000
0,122489700	0,122478000
-0,625575700	-0,625564000
0,112332100	0,112335000
0,482509200	0,482495000
0,009215391	0,009222230
-0,279677900	-0,279679000
0,524696700	0,524682000
0,087020260	0,087009600
0,392322500	0,392307000
-0,207244900	-0,207245000
0,413744600	0,413749000