دانشگاه تهران

پردیس دانشکده‌های فنی

دانشکده برق و کامپیوتر

گزارش آزمایش شماره 7

آزمایشگاه پردازش بی‌درنگ سیگنال‌های دیجیتال

پاییز 1400

نام و نام خانوادگی	علی ساعی زاده
شماره دانشجویی	810196477

فهرست

چکیده

پردازنده‌های دیجیتالی برای نمایش اعداد به ناچار از تعدادی بیت محدود استفاده می‌کنند که این موضوع منجر به ایجاد اثراتی در پیاده سازی الگوریتم‌ها می‌شود. این اثرات شامل کوانتزیاسون، سرریز، محاسبات صحیح و ... می‌گردد که در عمل چالش‌هایی به دنبال دارد. در این آزمایش به محاسبات صحیح و برخی از اثرات ناشی از نمایش اعداد با تعداد بیت محدود شامل کوانتزاسیون و سرریز بررسی می‌شود.

7-4-2 قسمت اول: پیاده سازی تابع کسینوس به صورت Fixed-Point

توضیح پیاده سازی

به دلیل اینکه اغلب پردازنده‌های DSP به صورت fixed-point هستند در این قسمت تابع کسینوس را به کمک بسط مک لورن پیاده سازی می کنیم که رابطه آن بصورت زیر است:

$$\cos\theta = 1 - \frac{1}{2!}\theta^2 + \frac{1}{4!}\theta^4 - \frac{1}{6!}\theta^6 + \cdots$$

پیاده سازی روش اول (fcos1)

در این قسمت از قطعه کد زیر برای پیاده سازی استفاده شد.

```
float fcoef[4]={1.0,-1/2.0,1.0/(2.0*3.0*4.0),
                -1.0/(2.0*3.0*4.0*5.0*6.0)};
float fcos1(float x)
{
   float out;
   out=fcoef[0];
   out+=fcoef[1]*x*x;
   out+=fcoef[2]*x*x*x*x;
   out+=fcoef[3]*x*x*x*x*x*x;
   return out;
}
```

شکل 1 کد روش اول پیاده سازی تابع کسینوس

با توجه به نحوه پیاده سازی این قسمت، به این دلیل که از جملات محدود بسط مک لورن استفاده شده است بنابراین تنها آرگومان ها با مقادیر کوچک قابل محاسبه هستند برای محاسبه آرگومان‌های بزرگ تر باید از جملات بیشتر بسط مک‌لورن استفاده شود.

پیاده سازی روش دوم (fcos2)

از کد زیر برای پیاده سازی قسمت دوم استفاده شد.

```
float fcoef[4]={1.0,-1/2.0,1.0/(2.0*3.0*4.0),
                -1.0/(2.0*3.0*4.0*5.0*6.0)};
float fcos2(float x)
{
   float out,x2;
   x2=x*x;
   out=x2*fcoef[3];
   out=(out+fcoef[2])*x2;
   out=(out+fcoef[1])*x2;
   out+=fcoef[0];
   return out;
}
```

شکل 2 کد روش دوم پیاده سازی تابع کسینوس

پیاده سازی روش سوم به صورت محاسبات ممیز ثابت (icos)

در این روش مقدار متغیر cosine 13 بیت به راست انتقال پیدا کرده است. دلیل آن است که به علت ضرب متغیر به فرمت 28 بیتی تبدیل می‌شود. اما فرمت استاندارد ما برای این قسمت 15 بیتی است. بنابراین پس از هر ضرب، 13 شیفت به راست می‌دهیم تا دوباره به فرمت 15 بیتی برسیم.

```
#define UNITQ15 0x7fff
short iCoef[4]={(short)(UNITQ15),(short)(-(UNITQ15/2.0)),
                (short)(UNITQ15/(2.0*3.0*4.0)),
                 (short)(-(UNITQ15/(2.0*3.0*4.0*5.0*6.0)))};
short icos(short x)
{  long cosine,z;
   short x2;
   z = (long)x * x;
   x2 = (short)(z>>15); // x2 has x(Q14)*x(Q14)
   cosine = (long)iCoef[3] * x2;
   cosine = cosine >> 13;
   cosine = (cosine + (long)iCoef[2]) * x2;
   cosine = cosine >> 13;
   cosine = (cosine + (long)iCoef[1]) * x2;
   cosine = cosine >> 13;
   cosine = cosine + iCoef[0];
   return((short)cosine);
}
```

شکل 3 کد روش سوم پیاده سازی تابع کسینوس

نتایج

خروجی کد برای ورودی های مختلف به شکل زیر بدست آمد که به کمک جدول کامل می‌شود.

```
real cos(0.000000): 1.000000 fcos1: 1.000000 fcos2: 1.000000 icos: 1.000000
real cos(0.100000): 0.995000 fcos1: 0.995004 fcos2: 0.995004 icos: 0.995056
real cos(-0.800000): 0.696700 fcos1: 0.696703 fcos2: 0.696703 icos: 0.696799
real cos(19.373156): 0.866000 fcos1: -67746.656250 fcos2: -67746.656250 icos: 0.809168
real cos(-0.523592): 0.866000 fcos1: 0.866029 fcos2: 0.866029 icos: 0.866054
real cos(1.570796): 0.000000 fcos1: -0.000894 fcos2: -0.000894 icos: -0.000732
real cos(3.141592): -1.000000 fcos1: -1.211352 fcos2: -1.211353 icos: 0.653462
```

شکل 4 خروجی کد پیاده سازی روش های مختلف

جدول 1 نتیجه تست پیاده سازی‌های مختلف برای تابع کسینوس

θ	$\cos\theta$	fcos1	fcos2	icos
0	1.0000	1.000000	1.000000	1.000000
0.1	0.9950	0.995004	0.995004	0.995056
-0.8	0.6967	0.696703	0.696703	0.696799
19.373155	0.8660	-67746.656250	-67746.656250	0.809168
-0.523592	0.8660	0.866029	0.866029	0.866054
1.5707963	0.0000	-0.000894	-0.000894	-0.000732
3.141592	-1.0000	-1.211352	-1.211353	0.653462

همانطور که در قست های قبل اشاره شد، دو روش اول به علت اینکه از جملات محدود بسط مک لورن استفاده می‌کنند در آرگومان های بزرگ دچار مشکل می‌شوند که در آرگومان 19.373155 این موضوع مشخص شده است.

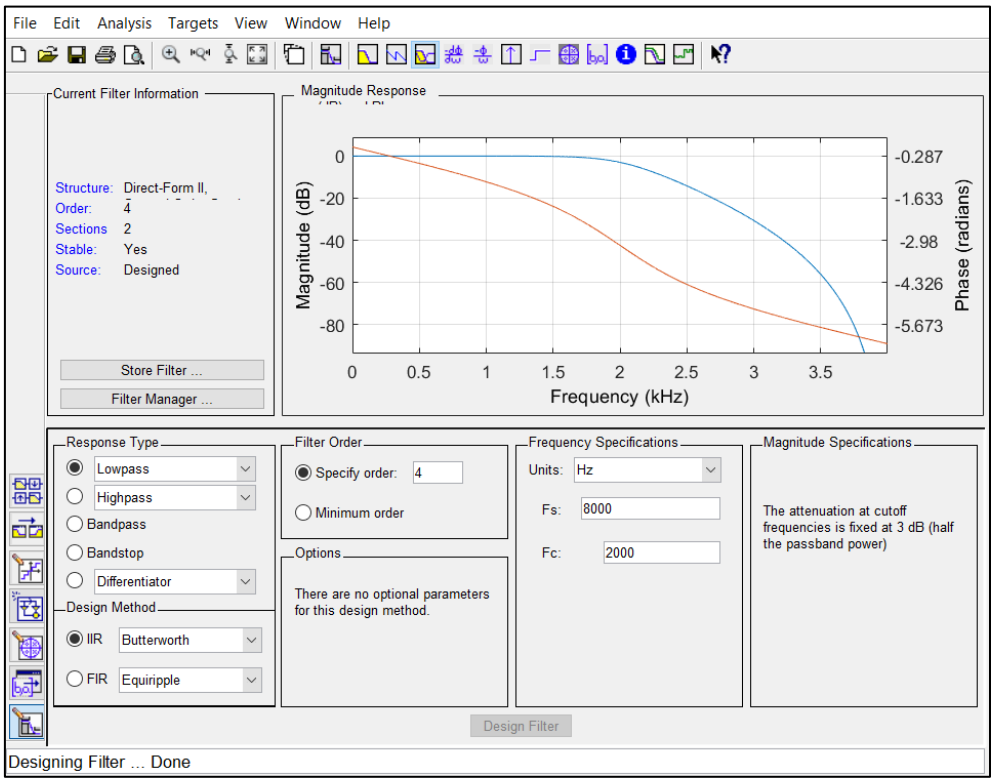
همچنین در آرگومان 3.141592 خطا برای همه روش به نسبت بقیه آرگومان ها زیاد است اما در روش سوم خطا قابل توجه تر است دلیل آن است که همانطور در دستور کار اشاره شد ورودی باید بین -2 و 2 باشد تا با فرمت استفاده شده قابل محاسبه باشد.

تمامی روش ها، به ازای آرگومان های کوچک با دقت خوبی عمل می‌کنند.

توضیح پیاده سازی

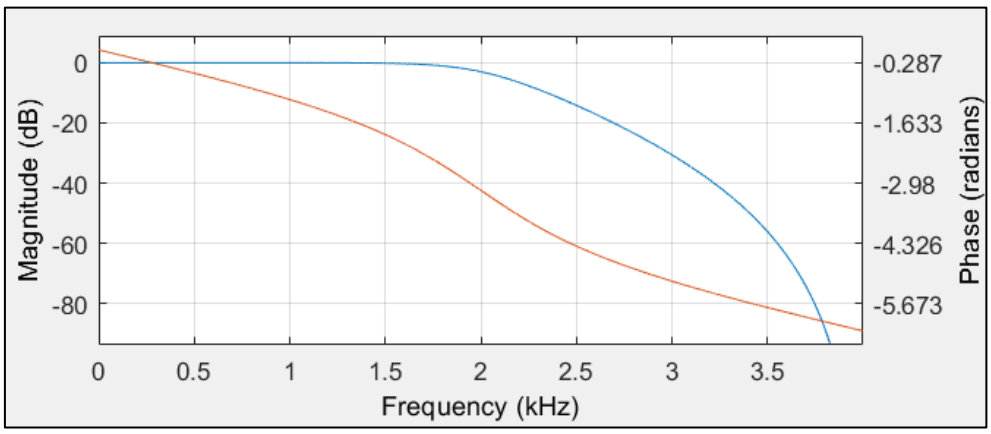
در این قست یک فیلتر IIR به صورت fixed-point طراحی و تست شد. مشخصات این فیلتر یک فیلتر پایین گذر با فرکانس قطع 2KHz است و فرکانس نمونه برداری نیز 8 KHz می‌باشد.

این فیلتر مرتبه 4 درمتلب به کمک ابزار fdatool طراحی به شکل زیر طراحی شد.



شکل 5 طراحی فیلتر در محیط **fdatool**

پاسخ دامنه و فاز این فیلتر به شکل زیر بدست آمد. مشخصات و ضرایب آن در فایل IIR_Filter.fcf ذخیره شد.



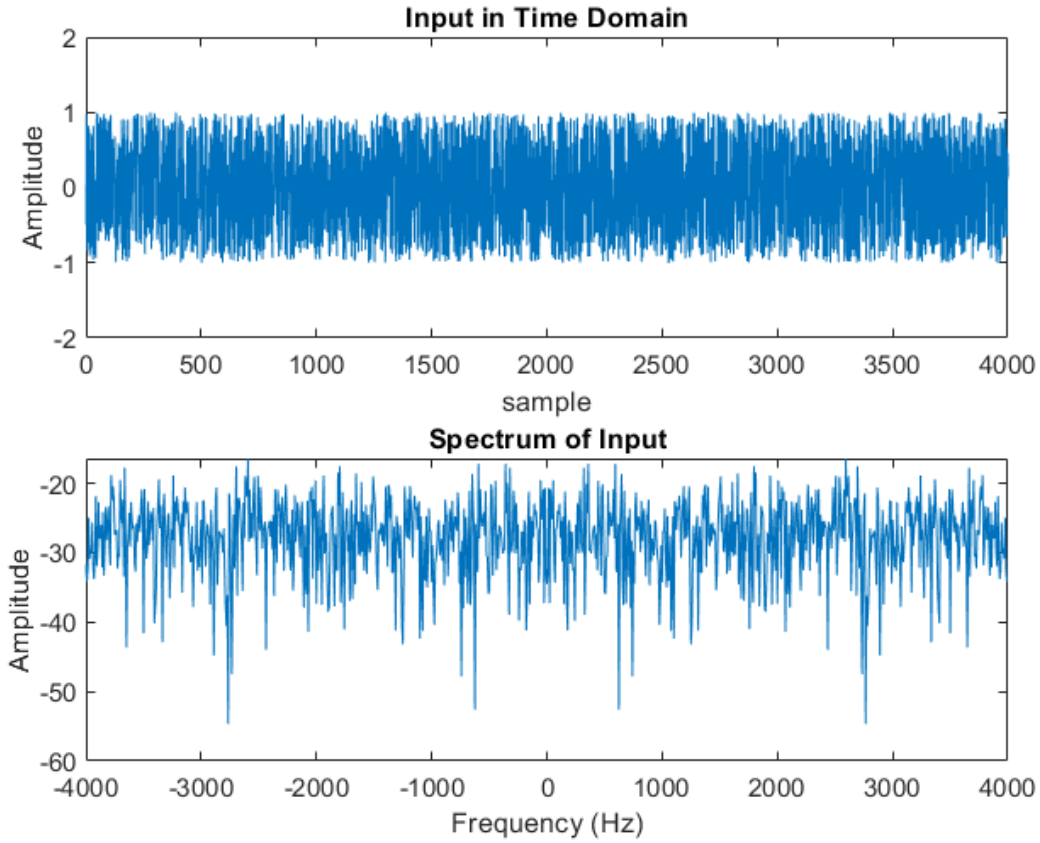
شکل 6 اندازه و پاسخ فیلتر بدست آده

برای تولید سیگنال از کد gen_random.py استفاده شد که 4000 مقادیر رندوم را بین -1 و 1 تولید و در فایل input.txt جهت ورودی دادن به کد C ذخیره می‌کند.

کد فیلترینگ سیگنال در فایل Lab7_2.c قابل مشاهده است. خروجی این قسمت در فایل out.txt ذخیره می‌شود تا در نرم‌افزار MATLAB بررسی شود.

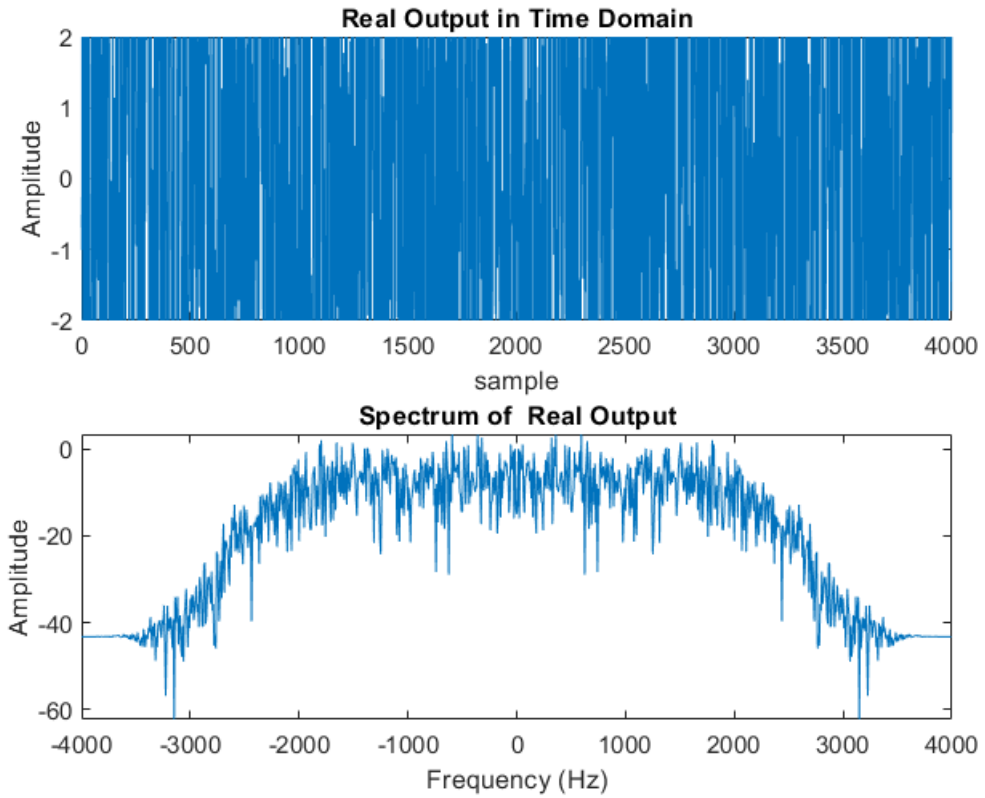
نتایج

سیگنال ورودی در حوزه زمان فرکانس به شکل زیر می‌باشد.



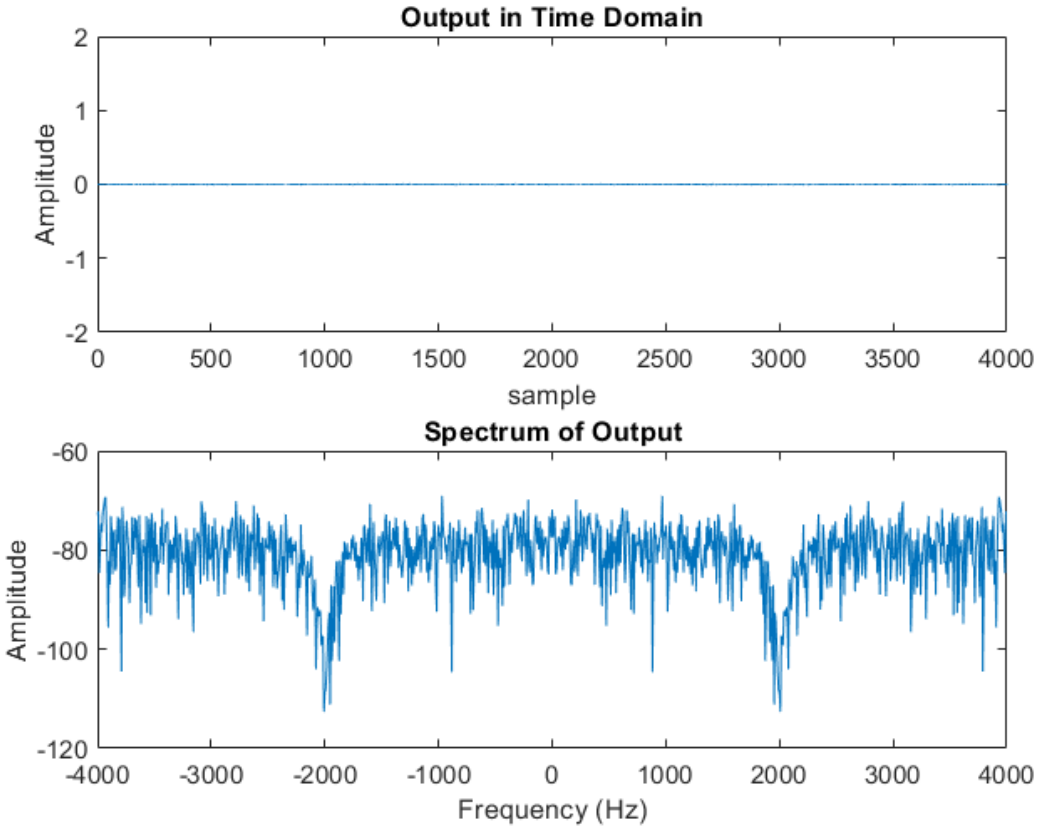
شکل7 سیگنال ورودی در حوزه زمان و فرکانس

همچنین خروجی به کمک دستور sosfilt متلب و ساتفاده از فیلتر محاسبه شد که در سکل زیر قابل مشاهده است. همانطور که انتظار می‌رفت فرکانس‌های بالا فیلتر شده اند.



شکل**8** سیگنال فیلتر شده توسط متلب

اما خروجی به کمک فیلتر fixed-point که در زبان C پیاده سازی شد به شکل زیر است که کاملا کارایی خودرا از دست داده است.



fixed-point سیگنال فیلتر شده توسط کد **9** شکل