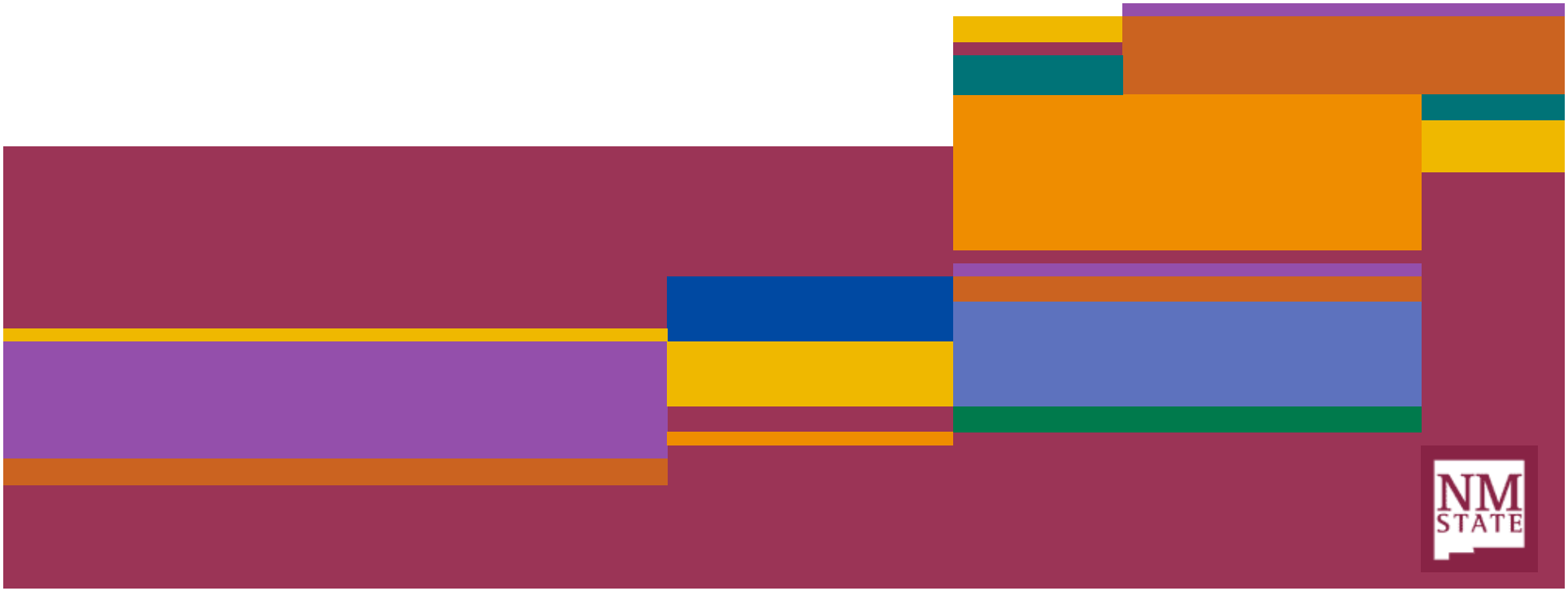
Signal Processing using TMS320C64x Digital Signal Processing Library (DSPLib) Part 1



TI - DSPLib

- TI provides a set of functions (DSPLib) which are C-callable, assembly-optimized routines
- See reference manual spru565b.pdf (download from TI website)
- See examples in spru884a.pdf (download from TI website)
- Advantages of using DSPLib functions:
  - Achieve execution speeds considerably faster than equivalent code written in standard ANSI C language
  - By using functions in TI DSPLIB, programmers can shorten application development time

Installing and Using DSPLib

- Add DSPLib by selecting dsp64x.lib from menu item *Project → Add Files to Project*
  - dsp64x.lib can be found at C:\ccstudio\c6400\include\lib\directory
- Make sure you link with the correct run-time support library and DSPLIB by having lrts6400.lib and ldsp64x.lib in linker command file (.cmd)
  - If first bullet is followed, this is automatic
- Header file for corresponding DSPLIB function is included in code (header name is same as function name)

Points to remember

- TI DSPLib functions typically operate over vector (array) operands
- Vector operands are composed of elements held in consecutive memory locations (may need address alignment along word- or double-word boundaries)
- Complex elements are assumed to be stored as real, imaginary, real, imaginary, etc.
- In-place computation is not allowed, unless noted, i.e. source operand is different than destination operand
- "restrict" keyword forces pointers to different memory addresses to avoid memory aliasing (compiler optimizations sometimes cause aliasing)

Memory Alias Disambiguation

```
void my_func (int *ptr_in, int *ptr_out)
{
        LDW         *ptr_in++, A0
        ADD         A0, 4, A1
        STW         A1, *ptr_out++
}
```

Compiler may think that *ptr_in and *ptr_out point to the same memory location leading to memory aliasing and hence the compiler may not optimize this piece of code. In order to avoid memory disambiguation use "restrict"

```
void my_func (int restrict *ptr_in, int *ptr_out)
{
        LDW         *ptr_in++, A0
        ADD         A0, 4, A1
        STW         A1, *ptr_out++
}
```

Categories in DSPLIB

- The routines contained in DSPLib are organized into the following categories:

1. Adaptive filtering
2. Autocorrelation
3. Filtering and convolution
4. Math
5. Matrix/vector operations
6. FFT
7. Miscellaneous

Arguments and Conventions Used

Arguments	Description
x,y	Argument reflecting input data vector.
r	Argument reflecting output data vector
nx,ny,nr	Arguments reflecting the size of vectors x,y, and r, respectively. For functions in the case nx = ny = nr, only nx has been used across
h	Argument reflecting filter coefficient vector (filter routines only)
nh	Argument reflecting the size of vector h
w	Argument reflecting FFT coefficient vector (FFT routines only)

Example 1: Find maximum value (and index) in an array

```
/* Maximum value and index of maximum value of a vector */
#include "C:\CCStudio\c6400\dsplib\include\dsp_maxval.h"
#include "C:\CCStudio\c6400\dsplib\include\dsp_maxidx.h"
int main (void)
{
int indxMax;
const short nx=48;
short  k,maxVal, x[48];
for(k=0;k<nx;k++)
   x[k]=-k;
   x[25]=100;

   maxVal = DSP_maxval(x,nx);
   indxMax = DSP_maxidx(x,nx);
   return 0;
}
```

Inner (dot) product of vectors and sum-of-squares of vector

- `int DSP_dotp_sqr(int G, short *x, short *y, int *r, int nx)`

  G (see below)

  x[nx], y[nx] - input vectors
  r - dot product of x and y
  nx - Number of elements (must be multiple of 4, ≥12)

Description: Compute dot product of x[ ] and y[ ] and store result in r. Also square each element of y[ ] and sum in G. G is passed back to calling function

Example 2: Inner (dot) product of vectors and sum squared elements of vector

```
/* Vector Dot product and Sum of Squares of the elements
   of one of the vectors */

#include "C:\CCStudio\c6400\dsplib\include\dsp_dotp_sqr.h"

int main (void)
{
int r=0, g=0;
short  x[12] = {12,11,10,9,8,7,6,5,4,3,2,1};
short  y[12] = {1,2,3,4,5,6,7,8,9,10,11,12};

   g = DSP_dotp_sqr(g,x,y,&r,12);
   return 0;
}
```

Matrix Multiplication

- void DSP_mat_mul(const short * restrict x, int r1, int c1, const short * restrict y, int c2, short * restrict r, int qs)

  x [r1*c1] - Pointer to input matrix of size r1 x c1
  r1 - Number of rows in matrix x
  c1 - Number of columns in matrix x (also number of rows in y)
  y [c1*c2] - Pointer to input matrix of size c1 x c2
  c2 - Number of columns in matrix y
  r [r1*c2] -Pointer to output matrix of size r1*c2
  qs - Final right shift (scaling) to apply to result

  Description: This function computes "r = x*y" for matrices x and y. All intermediate sums have 32-bit precision and no overflow checking is performed. Results are right-shifted by a user-specified amount, and then truncated to 16 bits

Matrix Transpose

- void DSP_mat_trans (const short *x, short rows, short columns, short *r)

  x[rows*columns] - pointer to input matrix, rows x columns
  rows - number of rows (must be a multiple of 4)
  columns - number of columns (must be a multiple of 4)
  r[columns*rows] - pointer to output matrix, columns x rows

  Description: This function transposes the input matrix x[ ] and writes the result to matrix r[ ].

Example 3: Matrix multiplication and transpose

```
/* Multiplication and Transpose of Matrices */
#include "C:\CCStudio\c6400\dsplib\include\dsp_mat_mul.h"
#include "C:\CCStudio\c6400\dsplib\include\dsp_mat_trans.h"
int main (void)
{

int rx=2,cx=2,cy=3,m,n;
short x[2][2] = {{20,50},{30,80}};
short y[2][3] = {{10,20,30},{40,50,60}};
short Y[4][4]={{1,2,3,4},{5,6,7,8},{1,9,2,8},{3,7,4,6}},R[4][4],r[2][3];
   DSP_mat_mul(*x, rx, cx, *y, cy, *r,0);
   DSP_mat_trans(*Y,4,4,*R);
 return (0);
}
```

Other DSPLib Math and Matrix Functions

Function

- `int DSP_dotprod(short *x, short *y, int nx`
- `short DSP_minval (short *x, int nx)`
- `void DSP_neg32(int *x, int *r, short nx)` **32-bit vector negate**
- `int DSP_vecsumsq (short *x, int nx)` **Sum-of-squares**
- `void DSP_w_vec(short *x, short *y, short m, short *r, short nr)`

Weighted vector Sum

- `void DSP_mul32` **element-by-element vector multiply**
- `void DSP_recip16(short *x, short *rfrac, short *rexp, short nx)`

element-by-element reciprocal