Ocean wave measurements (C Implementation - On Glider)

Prerequisites

Assumes GCC compiler
Make runs with C11

Basics

Most functionality assume access to a global table of floating point values called `Table`.
`Table` is defined in `/src/v2/array/data.h`
`Table` named as such because it is a rectangular array or equally sized float arrays.
from data.h:

```
F32 Table[ROWS][COLS] = {0};
```

`ROWS` controls the number of rows available to the program, and `COLS` controls the length of each row.
Inputs and Outputs to functions are generally either an `Index` or a `Coord`. An `Index` is simple unsigned integer used to specify a row of `Table`. Thus, `Index` is defined as:

```
typedef u32 Index;
```

`Coord` is used to specify individual values in Table. Thus `Coord` is defined as:

```
typedef struct { Index row; Index col } Coord;
```

In much of the code you will see two helper macros `ROW` and `POS`. These are used to help annotate the intention of the code.
`ROW(x) => Index x`
`POS(x,y) => Coord { row = y, col = x }`

Most functions have the interface:

```
Index function(Index source_row_1, ... Index source_row_N, Index
target_row)
```

Functions take as input the index of zero, one, or more rows in the table.
Functions return as outout the index of the row wherein the result was written.

For example, given the operation:

```
A = B + C - D
```

This expression must be written such that the target of each operation is explicit.

```
B + C → A - D → A
```

Which, when written using the Array.h functions:

```
Index A = ROW(0);
Index B = ROW(1);
Index C = ROW(2);
Index D = ROW(3);

Sub(
   Add(B, C, A),        // Add returns Index A
   D,
   A
);
```

Every function in array is written such that the input source arrays are not mutated as a result of the operation.

Types

You will see custom types sprinkled throughout the code base. They are defined in `/src/v2/common.h`. In addition to `Index` and `Coord` there are...

```
I8 I16 I32 I64      signed integer types
U8 U16 U32 U64      unsigned integer types
```

```
F32 F64             floating point types
C64 C128            complex floating point structs


// complex values are represented as follows
typedef struct { F32 real; F32 imag; } C64;
typedef struct { F64 real; F64 imag; } C128;

// word sized unsigned integer
typedef size_t UZ;
```

Input Data

`Data.h` also defines a seperate memory table called 'Input'.

```
#define INPUTS 3
#define INPUT_MAX (1<<12)
F32 Input[INPUTS][INPUT_MAX] = {0};
```

`Input` was created for the CS team to mock the functionality described by the ECE team. That is, it is expected that the Glider will accumulate a large array of data points to an SD card. This data will them be read into memory. Due to the constraints of the gliders onboard CPU capabilities, this data is processes in chunks using the Welch method. The Input memory table is used for testing the C code locally on desktop computers, but is made redundant by the Input arrays provided by the ECE implementation.

File Tour

`main.c`

`Main` calls three functions: `read_csv`, `process`, and `print_table`

`read_csv` exects as input a well formatted CSV files with following header:

```
t, x, y, z
```

`read_csv` expects this format, because this is the format of the output of the Python program (described below).
The header is discarded, then each row is parsed into 4 data points. The time (t) value is discarded.

Each the x, y, and z columns are then translated from ascii to 32-bit floats.
`read_csv` writes the

- `x` column to `Input[0]`
- `y` column to `Input[1]`
- `z` column to `Input[2]`

Before describing the function `process` which concerns the bulk of the programs functionality, I will describe `print_table` which is used for viewing the state of the program.

`print_table` is defined in `/src/v2/array/meta.h`

```
print_table(max_columns, max_rows)
```

`max_columns` and `max_rows` specify the size of the view window into the global `Table`.

`process` runs after read_csv. `process` includes operations that move data from the `Input` buffers described above into the main `Table` described above. `Table` is only operating on windows of data from `Input` buffer. The data read from `Input` is placed in rows 0, 1, 2 (x, y, z respectively).

Table

To minimize memory use, rows are reused by intermediate operations.

```
TABLE

Row  0 ← Raw X acceleration data
Row  0 ← Rolling mean of X acceleration data
Row  0 ← Biased with Hann window
Row  0 ← FFT(X), FFT of Row 0

Row  1 ← Raw Y acceleration data
Row  1 ← Rolling mean of Y acceleration data
Row  1 ← Biased with Hann window
Row  1 ← FFT(Y), FFT of Row 1

Row  2 ← Raw Z acceleration data
Row  2 ← Rolling mean of Z acceleration data
Row  2 ← Biased with Hann window
```

```
Row  2 ← FFT(Z), FFT of Row 2


Row  3 ← PSD(X, X), PSD of Row  0 and Row  0
Row  4 ← PSD(Y, Y), PSD of Row  1 and Row  1
Row  5 ← PSD(Z, Z), PSD of Row  2 and Row  2

Row  6 ← PSD(X, Y), PSD of Row  0 and Row  1
Row  7 ← PSD(X, Z), PSD of Row  0 and Row  2
Row  8 ← PSD(Y, Z), PSD of Row  1 and Row  2

// The Welch method sees the PSD performed on successive
// overlapping windows of the input data
Row  9 ← Accumulator for PSD XX
Row 10 ← Accumulator for PSD YY
Row 11 ← Accumulator for PSD ZZ

Row 12 ← Accumulator for PSD XY
Row 13 ← Accumulator for PSD XZ
Row 14 ← Accumulator for PSD YZ


// Wave Coefficients
Row 15 ← Frequency Space
Row 16 ← Scalar Coefficients
Row 17 ← a0
Row 18 ← Temporary Row for intermediate calculations

Pos 0,16 ← m0
Pos 1,16 ← m1
Pos 2,16 ← mm1
Pos 3,16 ← te
Pos 4,16 ← m2
Pos 5,16 ← tp

Row 19 ← denom (denominator)
Row 20 ← a1
Row 21 ← b1

Row 22 ← denom2 (denominator 2)

Pos  6,16 ← dp
Pos  7,16 ← Hs
Pos  8,16 ← Ta
Pos  9,16 ← wave energy ratio
Pos 10,16 ← Tz
Pos 11,16 ← PeakPSD
```

```
Pos 12,16 ← dp true

Row 22 ← A2
Row 23 ← B2
```

driver.h

`Process` is defined in `/src/v2/driver.h`

```
void process(
   UZ input_max,
   F32* x_input,
   F32* y_input,
   F32* z_input,
   F32 freq
)
```

As mentioned in the description of `main`, process expects 3 pointers to 3 memory buffers (x, y, z raw acceleration data). As well as the logical length of these input buffers `input_max`. `input_max` is the length of each input buffer, not the combined length of all input buffers. freq is the sampling frequency, for example: 1.5 corresponds to 1.5 Hz.

`process` performs the following calculations to get the PSD using the welch method

1. Load a window of raw acceleration data from input arrays into `Table`
2. Perform rolling mean on raw acceleration data
3. Bias (hann) the window of data
4. Calculate the X,Y,Z FFTs
5. Calculate the XX, YY, ZZ, XY, XZ, YZ PSDs from the FFTs

Once all the data is processed, the welch PSD is used by the `WaveCoeffients` function which is called at the end of `process`

processdata.h

`WaveCoeffients` is defined in `/src/v2/processdata.h`

`processdata.h` defines 4 functions

```
Index Rolling_mean(
   UZ window_size,
```

```
    Index s_r,
    Index t_r
)

Index Bias(Index t_r)

Index CalcPSD(
    float freq,
    Index s_r_x,
    Index s_r_y,
    Index t_r
)

Index WaveCoeffients(float freq)
```

`Rolling_mean`, `Bias`, `CalcPSD` are called by `process` described above `WaveCoeffients` is the only function of the 4 defined in `processdata.h` that implicitly modifies rows in `Table`. You can think of it as the second stage in the data processesing pipeline after `process`.

`WaveCoefficients` uses most of the row operators defined in `/src/v2/array/code.h` to calculate the wave coefficients using the PSDs. The calculation of these coefficents mirrors their numpy python implementation as closely as possible. You can find the specifics of where each coefficent is located in `Table` by reading the section 'Table' above.

data.h

`data.h` is located at `/src/v2/array/data.h`
`data.h` is a fairly short file, but it controls the size of the program memory. `data.h` was largely addressed in the section 'Basics', I will reiterate here. `Table` is a rectangular array of floats. `Table` is used for both arrays of real 32-floats, and arrays of 64-bit complex floats (32 bit real part, 32 bit imag part).

`POW_OF_2` determines the width of the welch window. This window must be a power of 2 and must be less than or equal to INPUT_MAX.

`Input` is a 3 row buffer for the raw x, y, z acceration data.

```
#define POW_OF_2 5
#define ROWS 24
#define COLS (1<<POW_OF_2)

F32 Table[ROWS][COLS] = {0};
```

```
#define INPUTS 3
// used by process fuction
#define INPUT_MAX (1<<12)
F32 Input[INPUTS][INPUT_MAX] = {0};
```

code.h

`code.h` is located at `/src/v2/array/code.h`
`code.h` contains 34 operators that each implicitly rely on `Table`. As described above, these operators pass around indexes to the rows in `Table` so that they can be composed together. Operators do not mutate source rows (`s_r`). Operators mutate target rows (`t_r`).

FFT

The first operator I will describe is `FFT`. I am specifically highlighting this operator so that if and when it is modified/replaced it can be done with the least amount of pain and suffering.

`FFT` relies on the file `FFT.h` located at `/src/v2/array/FFT.h`

```
Index FFT(Index s_r, Index t_r)
```

`FFT` reads a source row `s_r` and writes the FFT output to a target row `t_r`. As with all operators, the target row `t_r` is also returned by the operator for easy of function composition
`FFT` contains two local `C64` static arrays used to perform the FFT. Their size is also determined by `COLS` which is defined in `data.h`.

```
static C64 TMP[COLS] = {0};
static C64 data[COLS] = {0};
```

FFT.h

`FFT.h` is located at `/src/v2/array/FFT.h`
`FFT.h` contains a custom FFT implementation purpose built to be easy to use and to use no dynamic memory.

some things to note:

`lil_FFT` is the interface function used by the `FFT` operator defined in `code.h`. lil_FFT is the function that would need to be replaced should you decide to use a vendor specific FFT implementation. BTW, 'lil' in `lil_FFT` just means little.

```
... in code.h in the FFT operator
for (UZ i = 0; i < COLS; i++) {
   data[i].real = Table[s_r][i];
}
lil_FFT(data, TMP, COLS);              <--------- lil_FFT called
here
for (UZ i = 0; i < COLS/2; i++) {
   ((C64*) Table[t_r])[i] = data[i];
}
...
```

`gcc_log2` is a custom log2 function that relies on GCC specific intrinsics. ARM cpus have an O(1) operation called `clz` (count of leading zeros) that can be used to calculate integer log2.

back to `code.h`

`Op`

`code.h` defines an set of enums `Op`, that is used to dispatch a few crudely defined higher order functions scan, fold, etc.

```
typedef enum { ADD, SUB, DIV, MUL } Op;
```

`Mov`

`Mov` copies a source row into a target row

```
Index Mov(Index s_r, Index t_r)
```

`Iota`

`Iota` generates an ascending series of number from `0` to `COLS-1` in a target row

```
Index Iota(Index r)
```

Scale

`Scale` multiplies each cell of row `s_r` by `scalar` and places the result in row `t_r`

```
Index Scale(F32 scalar, Index s_r, Index t_r)
```

ScaleCell

`ScaleCell` multiplies a single cell `s` by `scalar` and places the result in row `t_r`

```
Coord ScaleCell(F32 scalar, Coord s, Coord t)
```

Inc

`Inc` increments each cell in row `s_r` by `scalar` and places the result in row `t_r`

```
Index Inc(F32 scalar, Index s_r, Index t_r)
```

IncCell

`IncCell` increments cell `s` by `scalar` and places the result in cell `t`

```
Coord IncCell(F32 scalar, Coord s, Coord t)
```

Reciprocal

`Reciprocal` creates the reciprocal of each cell in row `s_r` and places the result in row `t_r`

```
Index Reciprocal(Index s_r, Index t_r)
```

Add

`Add` zips two rows `s_r_a` and `s_r_b` with the addition operation and places the result in row `t_r`

```
Index Add(Index s_r_a, Index s_r_b, Index t_r)
```

Sub

`Sub` zips two rows `s_r_a` and `s_r_b` with the subtraction operation and places the result in row `t_r`

```
Index Sub(Index s_r_a, Index s_r_b, Index t_r)
```

Mul

`Mul` zips two rows `s_r_a` and `s_r_b` with the multiplication operation and places the result in row `t_r`

```
Index Mul(Index s_r_a, Index s_r_b, Index t_r)
```

Div

`Div` zips two rows `s_r_a` and `s_r_b` with the division operation and places the result in row `t_r`

```
Index Div(Index s_r_a, Index s_r_b, Index t_r)
```

ComplexMul

`ComplexMul` zips two rows of complex C64 data `s_r_a` and `s_r_b` with the complex multiplication operation and places the result in row `t_r`

```
Index ComplexMul(Index s_r_a, Index s_r_b, Index t_r)
```

ComplexScale

`ComplexScale` multiplies each cell of row `s_r` by a complex `scalar` and places the result in row `t_r`

```
Index ComplexScale(C64 scalar, Index s_r, Index t_r)
```

SetImag

SetImag sets the imaginary part of each complex value in row s_r to imag and places the result in row t_r

```
Index SetImag(float imag, Index s_r, Index t_r)
```

SetReal

SetReal sets the real part of each complex value in row s_r to real and places the result in row t_r

```
Index SetReal(float real, Index s_r, Index t_r)
```

Shift

Shift moves every cell to the right by shift amount if the value is positive, or to the left by shift amount if the value is negative and places the result in row t_r. If shift is 0, then Shift calls Mov.

```
Index Shift(I32 shift, Index s_r, Index t_r)
```

Sqrt

Sqrt performs the sqrt operation on each cell in row s_r and places the result in t_r

```
Index Sqrt(Index s_r, Index t_r)
```

SqrtCell

SqrtCell performs the sqrt operation on cell s and places the result in cell t.

```
Coord SqrtCell(Coord s, Coord t)
```

GetCell

`GetCell` returns the floating point values located at cell `s`

```
float GetCell(Coord s)
```

DivCell

`DivCell` divides a single cell `s_a` by `s_b` and places the result in cell `t`

```
Coord DivCell(Coord s_a, Coord s_b, Coord t)
```

MovCell

`MovCell` moves cell `s` to cell `t`

```
Coord MovCell(Coord s, Coord t)
```

SetCell

`SetCell` sets a cell `t` with floating point value `val`.

```
Coord SetCell(float val, Coord t)
```

ArcTanCell

`ArcTanCell` performs the atan2f function on cell `s_a` and `s_b` and places the result in cell `t`

```
Coord ArcTanCell(Coord s_a, Coord s_b, Coord t)
```

MaxCoord

`MaxCoord` scans row `s_r` and returns the position of the cell in row `s_r` that contains the greatest value.

```
Coord MaxCoord (Index s_r)
```

MaxCell

`MaxCell` uses `MaxCoord` to locate the max cell in row `s_r` and places the value of that cell in cell `t`.

```
Coord MaxCell (Index s_r, Coord t)
```

MaxCoordReal

`MaxCoordReal` is similar to MaxCoord, but specifically looks only at the real part of a row of complex values in row `s_r`. `MaxCoordReal` returns the position of the cell in a complex row that contains the largest real part.

```
Coord MaxCoordReal(Index s_r)
```

RadToDegreeCell

`RadToDegreeCell` converts a cell `s` from radians to degress and places the result in `t`

```
Coord RadToDegreeCell(Coord s, Coord t)
```

ModCell_immediate

`ModCell_immediate` performs uses fmodf to perform a modulo operation on cell `s` with `mod` and places the result in `t`

```
Coord ModCell_immediate(Coord s, float mod, Coord t)
```

Scan

`Scan` performs one of 4 operations (ADD, SUB, MUL, DIV) `op` on row `s_r`. Scan perfoms an operation between two cells [n] and [n+1] in a row and places the result of the operation in [n+1], thus accumulating the result of the computation through the array. `Scan` places the result of this operation in row `t_r`.

```

```

```
Index Scan(Op op, Index s_r, Index t_r)
```

FOLD

`FOLD` is similar to `Scan` but does not accumulate the results in a row, instead the row is reduced to a single value (this would be the last value if scan was used). `FOLD` performs one of 4 operations (ADD, SUB, MUL, DIV) `op` on row `s_r` and places the result in cell `t`.

```
Coord FOLD(Op op, Index s_r, Coord t)
```

ComplexConj

`ComplexConj` creates the complex conjugate of row `s_r` and places the result in `t_r`.

```
Index ComplexConj(Index s_r, Index t_r)
```

FreqSpace

`FreqSpace` populates row `t_r` with ascending multiples of frequency `freq`

```
Index FreqSpace(float freq, Index t_r)
```

Load

`Load` moves data from a F32 buffer `input` into row `t_r`. `Load` reads from the input buffer starting at index `offset`. `Load` does not have any means of performing bounds checking on the input buffer, thus caller must ensure that the input buffer + offset is greater than or equal to the length `COLS`.

```
Index Load(F32* input, Index offset, Index t_r)
```

Cos

`Cos` peforms cosf on each cell in row `s_r` and places the result in row `t_r`.

```
Index Cos(Index s_r, Index t_r)
```

Tutorial:

Run python program to generate CSV file from CDIP data

```
python ./Driver.py
"./ncFiles/067.20201225_1200.20201225_1600_output.nc"
```

Run make in `/C` to compile the C code and run the resulting program

```
make
```