Boost Accumulators

What is the Boost Accumulators Library?

Provides powerful wrappers for performing real-time statistical operations on datasets.

What Type of Operations?

- Min, Max, Count
- Mean, Median, Mode
- Variance
- Density, Moments, Tails
- Much more

Boost Accumulators

- Header-only library
- Heavily utilizes Boost Meta-Programming Library
- All `accumulators` are either $O(1)$ or $O(N)$ with N as a fixed number (ex: number bins in CDF).

What is this good for?

This demo is focused on using this library, which wraps Boost Accumulators, to enable useful troubleshooting and debugging of systems which are difficult to profile.

- Often, we have performance-constraints which prevent building with debugging symbols.
- There can be bugs which occur infrequently, thus its difficult to find patterns.
    - **ex:** Blocking due to muxes or other race conditions.
- There can be extensive noise in the data, making it difficult to find trends.
    - **ex:** Measuring compression ratios of imagery algorithms.
    - **ex:** Functions working on many threads at once.
- The problem grows slowly, making it difficult to identify the problem in the first place.
    - **ex:** Memory leaks and algorithms with $O(n^2)$ performance.

What is this not good for?

- This will **never** beat the power using `awk`, `sed`, and `grep` on log files. It's just faster.
    - Use Pyplot to actually visualize trends.
- It won't beat a profiler or debugger. This is designed for cases where we can't use either effectively.

Simple Example

1. Determine the features you want to use

```
// A solid "general-purpose" feature set
typedef boost::accumulators::stats<boost::accumulators::tag::mean,
                                   boost::accumulators::tag::min,
                                   boost::accumulators::tag::max,
                                   boost::accumulators::tag::sum,
                                   boost::accumulators::tag::count,
                                   boost::accumulators::tag::variance> FEATURE_SET;
```

2. Build the `accumulator_set` object.

```
boost::accumulators::accumulator_set<double,FEATURE_SET> acc;
```

3. Add data to the `accumulator_set`.

```
acc( some_value );
```

Simple Example

4. Query specific values

```
double mean   = boost::accumulators::mean( acc );
double stddev = std::sqrt( boost::accumulators::variance( acc ) );
double min    = boost::accumulators::min( acc );
double max    = boost::accumulators::max( acc );
```

Tips for using Boost Accumulators

1. Always wrap code with classes to ease copy-paste.
2. Use `typedefs` and namespaces to simplify your logic.
    - Don't be tempted to use `using namespace boost::accumulators`. ***BAD***!
3. Create print functions to standardize output.
4. Recommend printing in a log-friendly or analysis-friendly way
    - Use space-deliminated patterns with a logger that prints timestamps if using with `awk`, `grep`, `select-string` or other shell tools.
    - For Python analysis, write as JSON or use Boost `property_tree` for output.
5. Don't over-complicate your wrapper code unless you are comfortable with templates.

A Few Notes About Template Meta-Programming in C++

- Don't do it unless you are **amazing** with C++ and have a **high** tolerance for pain.
- It can cause ***incredible*** pain and suffering:
    - It creates vague and useless compilation errors. Botched recusion in templates can create thousands of lines of replicated errors.
    - Templates don't get compiled until you *instanciate* an object with the given template parameters. You may go 5 years without knowing it fails on `uint8_t` but passes with an `int8_t`.
    - IDEs are utterly useless for doing anything once you create a template.
    - Once you start creating `typedef`, you tend to create dozens more.
    - You will inevitably forget to wrap a template in a namespace or call `using namespace <>` and re-learn why that's bad.

A Few Notes About Template Meta-Programming in C++

- It can do incredible things
  - Enforce extreme type safety. Stop using `double` and create `mass` or `weight` types.. https://www.boost.org/doc/libs/1_82_0/libs/mpl/doc/tutorial/tutorial-metafunctions.html
  - You can force the compiler to solve problems for you: (Fibonacci example) https://stackoverflow.com/questions/908256/getting-template-metaprogramming-compile-time-constants-at-runtime
  - You can statically bind vtables for extra performance in Polymorphic classes. https://en.wikipedia.org/wiki/Curiously_recurring_template_pattern

`lib-accumulator`

Simple wrapper around Boost Accumulators

- `Accumulator` class for managing features and printing output.
- `Stats_Aggregator` class for tracking stats over groups of accumulators.
- `Stopwatch` class for easier timing.
- Thread-safe, so you can insert even if inside a pool.

For all demos, these can be treated as the same fundamental thing. The goal is to identify *why* to use accumulators, not which one.

Demo 1 : Using Accumulators to track runtime performance.

example: `src/demo1.cpp`

- Imagine a function that does a lot of work and it repeated many times.

```
// Do a big loop
for( size_t i=0; i < number_iterations; i++ )
{
    // Get the start time
    auto start_time = std::chrono::steady_clock::now();

    Do_Complex_Task();

    // Compute task time
    auto operation_time = std::chrono::duration_cast<std::chrono::milliseconds>(std::chrono::steady_clock::now() - start_time);
    acc_worker( operation_time.count() );
}
```

- Notes:
    - This accumulator call is often faster than logging due to no string-building.
    - You add 2-3 lines of instrumentation code for a specific region.
    - This runs in `Release` or `Debug` builds. No need to profile.

Demo 1 : Using Accumulators to track runtime performance.

- Create a print function which wraps all of the relevant calls

```
std::string print_accumulator( const boost::accumulators::accumulator_set<double,FEATURE_SET>& acc,
                               const std::string&                                              units )
{
    std::string gap( 4, ' ' );
    std::stringstream sout;

    // Recommend setting precision for when you print doubles.
    sout << std::setprecision(4);
    sout << std::fixed;

    sout << gap << "Count : " << boost::accumulators::count( acc ) << " entries" << std::endl;
    sout << gap << "Mean  : " << boost::accumulators::mean( acc ) << " " << units << std::endl;
    sout << gap << "Min   : " << boost::accumulators::min( acc ) << " " << units << std::endl;
    sout << gap << "Max   : " << boost::accumulators::max( acc ) << " " << units << std::endl;
    sout << gap << "StdDev: " << std::sqrt( boost::accumulators::variance( acc ) ) << " " << units << std::endl;
    sout << gap << "Sum   : " << boost::accumulators::sum( acc ) << " " << units << std::endl;

    return sout.str();
}
```

Demo 1 Results

```
Final Results After 1000 Iterations
    Count : 1000 entries
    Mean  : 102.5610 ms
    Min   : 90.0000 ms
    Max   : 115.0000 ms
    StdDev: 6.0851 ms
    Sum   : 102561.0000 ms
```

What does this tell us?

1. Mean is 102 ms with a small Standard-Deviation. This method is very consistent.
2. **Min** and **max** are relatively close to **mean**, so no random noisy results.
3. A total of 102 seconds was spent in this function for the duration of the loop.

What does this tell us?

```
Final Results After 1000 Iterations
    Count : 1000 entries
    Mean  : 102.5610 ms
    Min   : 0.1000 ms
    Max   : 300.0000 ms
    StdDev: 24.0851 ms
    Sum   : 82561.0000 ms
```

1. Mean is 102 ms with a very large Standard-Deviation.
   - This method is inconsistent and may have logic that needs review.
2. **Min** is ***effectively*** zero. You may have an edge case or something not working properly.
3. **Max** is 3x the mean for what was previously a very tight result.
   - Look for deadlocks, resource starvation, or other multi-threading woes.
   - If max is considerably far from the mean than the min, you likely have a deadlock. Algorithms typically have a fairly centered mean.

Demo 2 - Analyzing Compression Performance

In this very lazy example, a batch of random images are created and written to disk. The size is measured of the image and compared against the expected size. Timing info is also tracked. Multiple image formats are considered.

1. Create Accumulators

```
auto timing_acc      = acc::Accumulator<acc::FULL_FEATURE_SET,double>::create( "ms" );
auto compression_acc = acc::Accumulator<acc::FULL_FEATURE_SET,double>::create( "%" );
```

Demo 2 - Analyzing Compression Performance

2. Dispatch Threads

```
{
    Thread_Pool pool( max_threads );
    pool.init();

    std::deque<std::future<void>> jobs;
    for( size_t worker = 0; worker < number_images; worker++ )
    {
        // See Next Slide
    }

    for( auto& job : jobs ){ job.get(); }
    pool.shutdown();
}
```

Demo 2 - Analyzing Compression Performance

3. Measure compression time

```
jobs.push_back( pool.submit([=, &timing_acc, &compression_acc, &format ]() {
                int id = worker;

                acc::Stopwatch<> timer;
                Compress_Imagery( id,
                                  image_size,
                                  output_dir,
                                  compression_acc,
                                  format );
                timing_acc.insert( (double)timer.stop().count() );
            }));
```

Demo 2 - Analyzing Compression Performance

3. Measure compression rate

```cpp
// 1. Create Image
// 2. Get expected image size (uncompressed)
const size_t expected_size = img_size.width * img_size.height * 3;

// 3. Create output image path
// 4. Build image
// 5. Smooth with median filter to help dilute noise
// 6. Write image

// 7. Compute compression percentage
double file_ratio = std::filesystem::file_size( output_path ) / (double)expected_size;
comp_acc.insert( file_ratio * 100 );
```

Demo 2 - Analyzing Compression Performance

```
Timing Accumulator: .jpg
    Count. . . . . . . : 2000 entry(s).
    Mean . . . . . . . : 452.417000
    Min. . . . . . . . : 230.000000 ms
    Max. . . . . . . . : 601.000000 ms
    StdDev . . . . . . : 61.779617 ms
    Variance . . . . . : 3816.721111 ms
    Sum. . . . . . . . : 904834.000000 ms
    Last Entry. . . .: 230.000000 ms

Compression Accumulator.jpg
    Count. . . . . . . : 2000 entry(s).
    Mean . . . . . . . : 15.368703
    Min. . . . . . . . : 15.323849 %
    Max. . . . . . . . : 15.417599 %
    StdDev . . . . . . : 0.013912 %
    Variance . . . . . : 0.000194 %
    Sum. . . . . . . . : 30737.406491 %
    Last Entry . . . . : 15.374389 %
```

Demo 2 - Analyzing Compression Performance

Timing Info (2000 images compressed)

Stat	JPEG	PNG	TIF
Mean	423 ms	2711 ms	410 ms
Min	212 ms	2485 ms	212 ms
Max	601 ms	7803 ms	635 ms
StdDev	61 ms	420 ms	98 ms
Variance	3789 ms	176730 ms	9506 ms
Sum	838 s	1013 s	820 s

Demo 2 - Analyzing Compression Performance

Compression Performance (2000 images compressed)

Stat	JPEG	PNG	TIF
Mean	15.36%	56.44%	71.66%
Min	15.32%	56.32%	71.55%
Max	15.41%	56.58%	71.78%
StdDev	0.01%	0.04%	0.04%
Variance	0.00%	0.00%	0.00%
Sum	junk	junk	junk

Demo 2 - Analyzing Compression Performance

- This demo was designed to be as brutal as possible on compression algorithms.
- JPEG is clearly the highest performer. JPEG gives impressive results at consistent timing.
- TIF gives okay results with LZW with very efficient timing.
- PNG has consistent results, but wildly different timing.

Demo 3 - Finding bugs in noisy or complex environments

This demo