Exploration of Strongly-typed Units:

A Case Study from Digital Audio

ROTH MICHAELS

Roth Michaels (he/him)
Principal Software Engineer
Native Instruments

You can do it!

NATIVE INSTRUMENTS

iZOTOPE

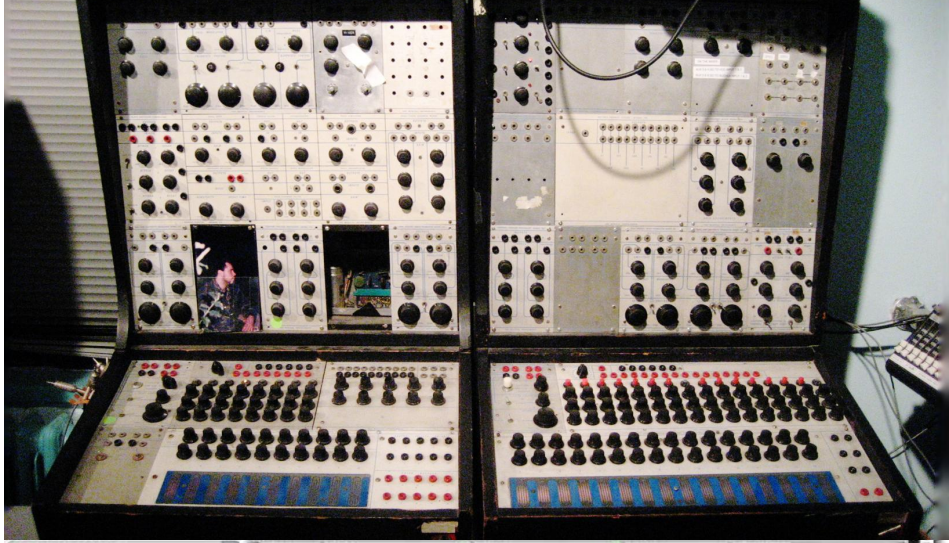
Plugin Alliance

BRAINWORX


Inst 3
Manuell
Vergleichen
Widerrufen
Wiederholen
Ansicht:
Editor
7 SKIES, Magnificence - THE DRILL
FILTER
CUTOFF
RESO
ENV AMT
AMP
LAYER 1
LAYER 2
THE DRILL
FLT ENV
ATTACK
DECAY
SUSTAIN
RELEASE
AMP ENV
Inst 4
gr808
808 Center
Mode
[mono]
808 Sides
Panning
[attack: fast]
[releaseeeeee]
Reverb: [Cloud Grain]
Size
Amount:
Rev Lo-Cut:
Rev Hi-Cut:
Presets: 1 2 3
Clay and Kelsy
Bouncy FX: [off]
Speed ModW:
gr808
kick
Kontakt
Inst 2
MS20 Drums
Cutoff
Saturation
Reverb
Delay
Delay Time
Peter Flint:
MS20 Drums
Kontakt

About me...

- Studied music composition
- Programmed my own tools
- Many years polyglot consultant
- Involved in early days of Swift Evolution
- 7 years writing C++ for digital audio
- Artisanal unit vocabulary types



https://www.vintagesynth.com/misc/tassman4.php

What do units matter?

What do units matter?

...making it to Mars.

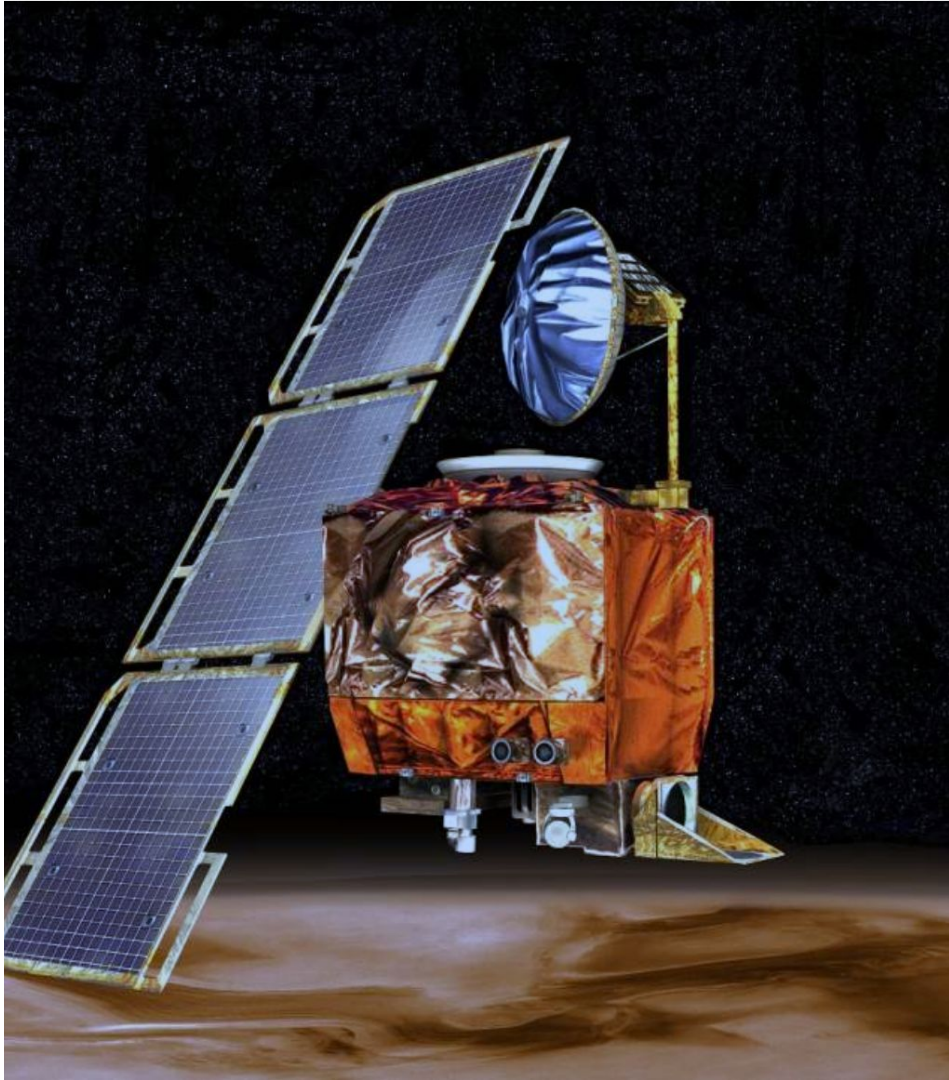
Mars climat orbiter

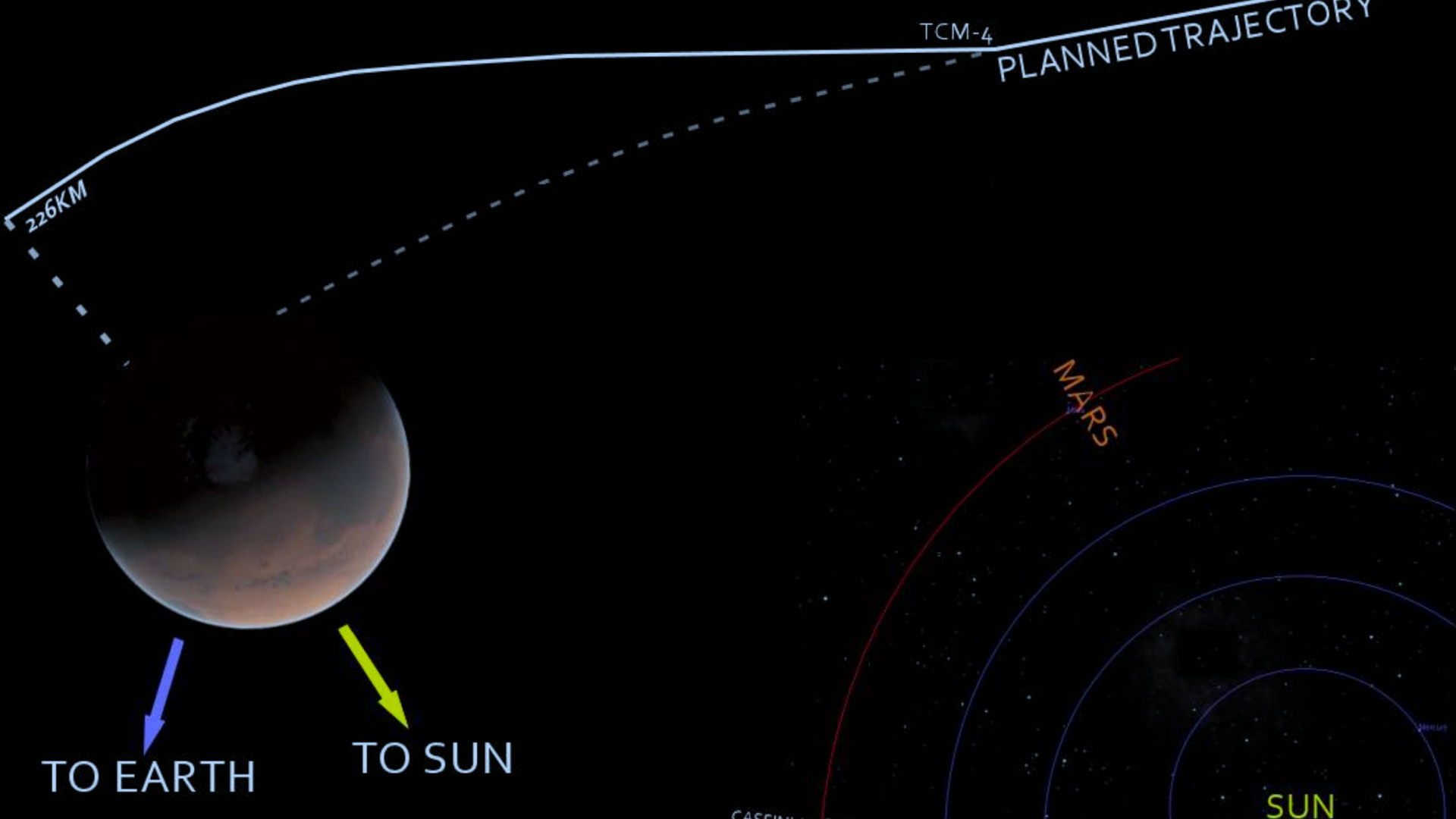
Launch December 11, 1998

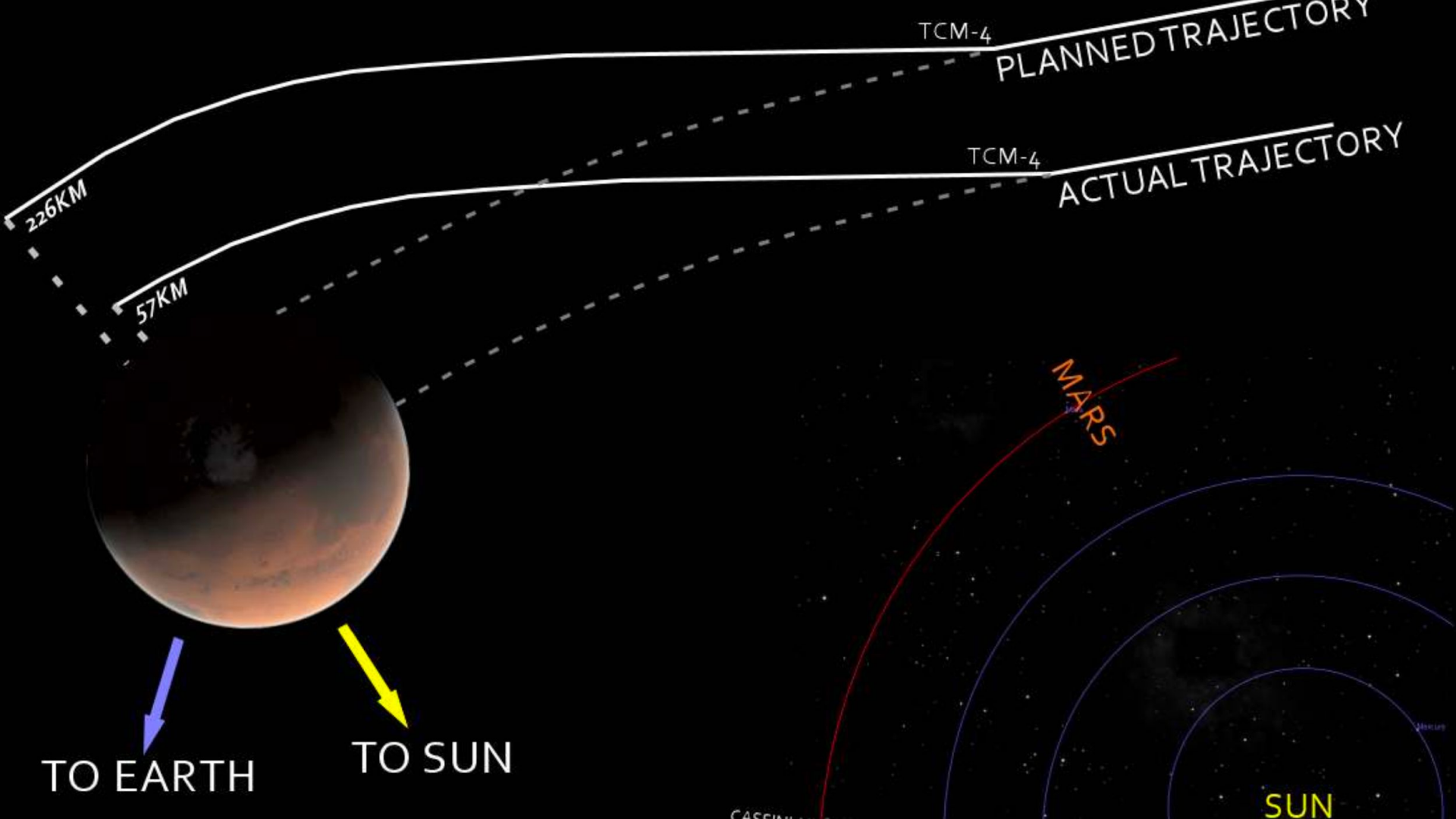
Cost $327.6 million

Expected to reach Mars September 23, 1999

Pound-force-seconds vs newton-seconds

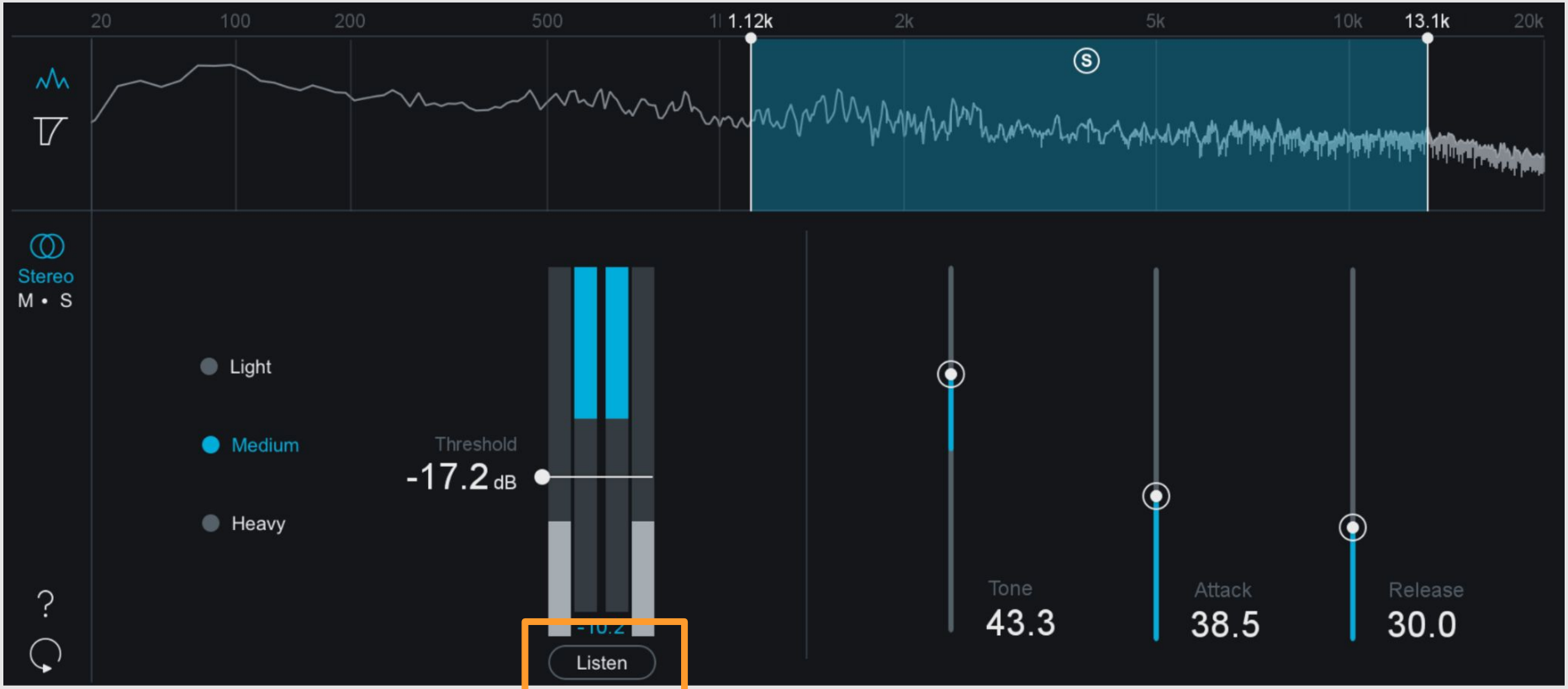



TCM-4
PLANNED TRAJECTORY
226KM
MARS
TO EARTH
TO SUN
SUN


TCM-4
PLANNED TRAJECTORY
TCM-4
ACTUAL TRAJECTORY
226KM
57KM
MARS
TO EARTH
TO SUN
SUN

What do units matter?

... “is it supposed to make sound?”


20
100
200
500
1k
1.12k
2k
5k
10k
13.1k
20k
S
Stereo
M • S
Light
Medium
Heavy
Threshold
-17.2 dB
Listen
Tone
43.3
Attack
38.5
Release
30.0

Not life and death, but audio bugs can be dangerous

- Hearing loss
- Damaged speakers
- Crashes during live performance

I tried boost::units...

...and gave up.

```
struct meter_base_unit : base_unit<meter_base_unit, length_dimension, 1> { };
struct kilogram_base_unit : base_unit<kilogram_base_unit, mass_dimension, 2> { };
struct second_base_unit : base_unit<second_base_unit, time_dimension, 3> { };

typedef make_system<
    meter_base_unit,
    kilogram_base_unit,
    second_base_unit>::type mks_system;

/// unit typedefs
typedef unit<dimensionless_type,mks_system>      dimensionless;

typedef unit<length_dimension,mks_system>        length;
typedef unit<mass_dimension,mks_system>          mass;
typedef unit<time_dimension,mks_system>          time;

typedef unit<area_dimension,mks_system>          area;
typedef unit<energy_dimension,mks_system>        energy;
```

o BOOST_UNITS_STATIC_CONSTANT is provided in boost/units/static_constant.hpp to facilitate ODR- and thread-safe c
efinitions:

```
/// unit constants
BOOST_UNITS_STATIC_CONSTANT(meter,length);
BOOST_UNITS_STATIC_CONSTANT(meters,length);
BOOST_UNITS_STATIC_CONSTANT(kilogram,mass);
BOOST_UNITS_STATIC_CONSTANT(kilograms,mass);
BOOST_UNITS_STATIC_CONSTANT(second,time);
BOOST_UNITS_STATIC_CONSTANT(seconds,time);

BOOST_UNITS_STATIC_CONSTANT(square_meter,area);
BOOST_UNITS_STATIC_CONSTANT(square_meters,area);
BOOST_UNITS_STATIC_CONSTANT(joule,energy);
BOOST_UNITS_STATIC_CONSTANT(joules,energy);
```

for textual output of units is desired, we can also specialize the base_unit_info class for each fundamental dimension tag:

```
template<> struct base_unit_info<test::meter_base_unit>
{
    static std::string name()               { return "meter"; }
    static std::string symbol()             { return "m"; }
};
```

My journey to mp-units...

https://github.com/mpusz/mp-units

My journey to to mp-units...

- P1386 A Standard Audio API for C++: Motivation, Scope, and Basic Design
  - Wanted standard vocabulary types
    - Multichannel buffers: std::mdspan (?)
    - Frequency, sample time
- Searched for prior art
  - Found mp-units and…
  - P1935 A C++ approach to physical units
- All before Au was released

Are physical unit libraries for us?

...yes!

Are physical unit libraries for us?

- “Physical units don’t cover our use cases”
  - Not just about “physical” quantities/units
- “Don’t handle non-linear units”
  - nholthaus units has a technique
  - being explored in mp-units / Au
- How are we going to deal with 3rd party float APIs

hello, units...

The Downcasting Facility

```
#include <units/isq/si/length.h>
#include <units/isq/si/time.h>
#include <units/isq/si/speed.h>

using namespace units::isq;

constexpr Speed auto avg_speed(Length auto d, Time auto t)
{
  const auto s = d / t;
  std::cout << s << "\n";
  return s;
}
```

https://www.youtube.com/watch?v=l0rXdJfXLZc

mp-units: Lessons Learned and a New C++ Library Design (Mateusz Pusz)

Tuesday, October 3 • 14:00 - 15:00

The Au Library: Handling Physical Units Safely, Quickly, and Broadly

Log in to save this to your schedule, view media, leave feedback and see who's attending!

https://sched.co/1Qtf Tweet Share

Log in to leave feedback.

We present Au: a new open-source C++ units library, by Aurora. If you've rejected other units libraries because they couldn't meet your needs, check out Au (pronounced, "ay yoo"). It combines cutting-edge developer experience (fast compilation, simple and readable compiler errors) with wide accessibility (C++14 compatibility, single-header delivery option). You can be up and running with Au in your project --- in any build system --- in less time than it takes to read this abstract!

The Au Library: Handling Physical Units Safely, Quickly, and Broadly (Chip Hogg)

Conventions in the slides

- constexpr or inline may be left off for slides
- using mp_units in all slides
- ni:: / tp:: for example code
  - unless defined on the slide

Conventions in the slides

- constexpr or inline may be left off for slides
- using mp_units in all slides
- ni:: / tp:: for example code
  - unless defined on the slide
-

hello, units

```
using namespace mp_units::si::unit_symbols;
using namespace
  mp_units::international::unit_symbols;

constexpr quantity v1 = 110 * (km / h);
// 110 km/h
constexpr quantity v2 = 70 * mph;
// 70 mi/h
```

hello, units

```
auto v3 = avg_speed(220. * isq::distance[km],
                    2 * h);
// 110.0 km/h
auto v4 = avg_speed(isq::distance(140. * mi),
                    2 * h);
// 70.0 mi/h
```

hello, units

```
constexpr auto avg_speed(auto d, auto t) {
  return d / t;
}

auto v3 = avg_speed(220. * isq::distance[km],
                    2 * h);
// 110.0 km/h
auto v4 = avg_speed(isq::distance(140. * mi),
                    2 * h);
// 70.0 mi/h
```

hello, units

```
auto avg_speed(QuantityOf<isq::length> auto d,
               QuantityOf<isq::time> auto t) {
  return d / t;
}
auto v3 = avg_speed(220. * isq::distance[km],
                    2 * h);
// 110.0 km/h
auto v4 = avg_speed(isq::distance(140. * mi),
                    2 * h);
// 70.0 mi/h
```

hello, units

```
auto avg_speed(QuantityOf<isq::length> auto d,
               QuantityOf<isq::time> auto t) {
  return d / t;
}
auto v3 = avg_speed(220. * isq::distance[km],
                    2 * h);
// 110.0 km/h
auto v4 = avg_speed(isq::distance(140. * mi),
                    2 * h);
// 70.0 mi/h
```

hello, units

```
auto avg_speed(QuantityOf<isq::length> auto d,
               QuantityOf<isq::time> auto t) {
  return isq::speed(d / t);
}
auto v3 = avg_speed(220. * isq::distance[km],
                    2 * h);
// 110.0 km/h
auto v4 = avg_speed(isq::distance(140. * mi),
                    2 * h);
// 70.0 mi/h
```

hello, units

```
auto v4 = avg_speed(isq::distance(140. * mi),
                    2 * h);
// 70.0 mi/h

auto v5 = v3.in(m / s);
// 30.5556 m/s
auto v6 = value_cast<m / s>(v4);
// 31.2928 m/s
auto v7 = value_cast<int>(v6);
// 31 m/s
```

Are units real?

Are units real?

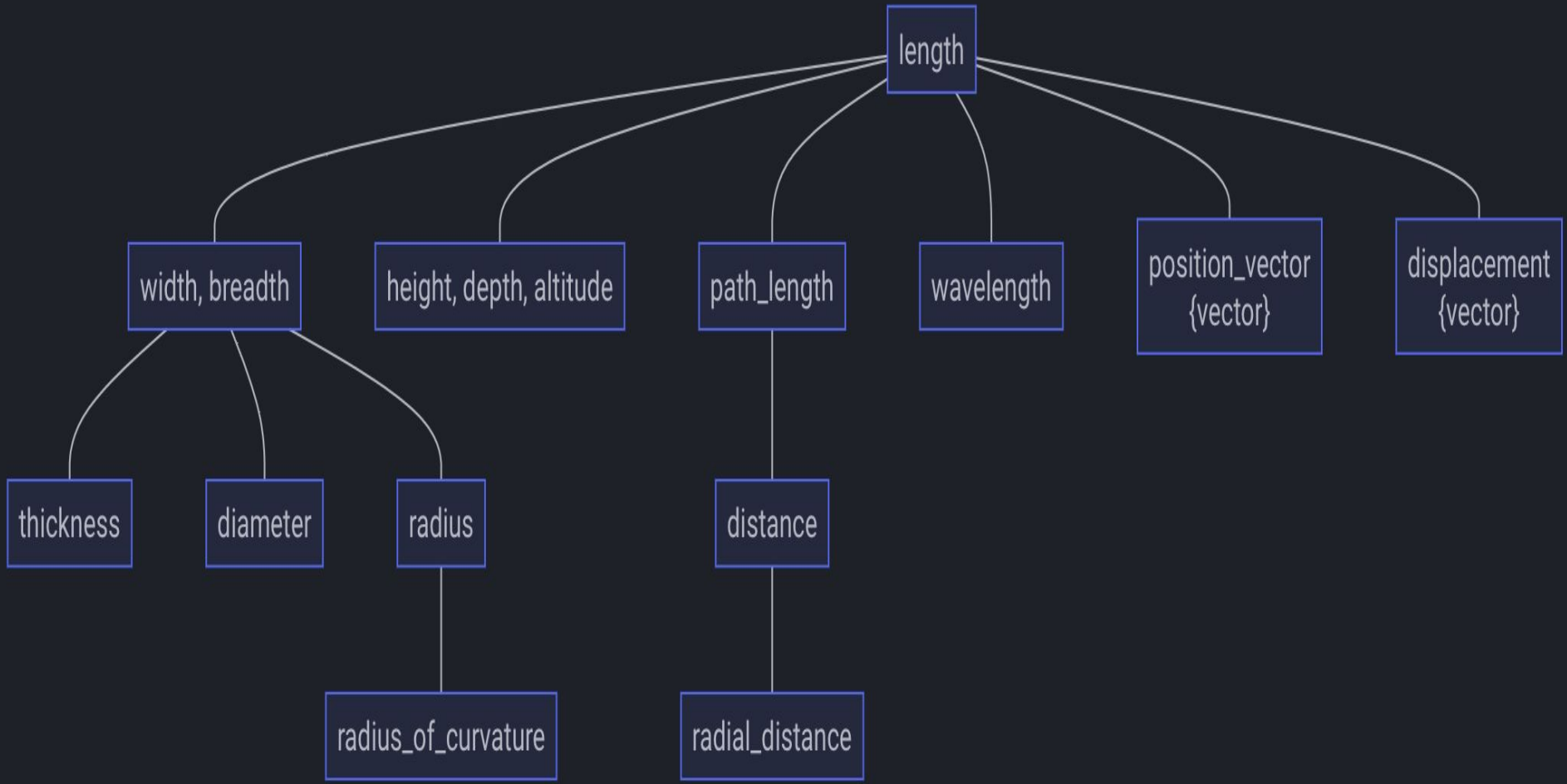
```
F = m * a
```

quantity kinds

```
auto avg_speed(QuantityOf<isq::length> auto d,
               QuantityOf<isq::time> auto t) {
  return isq::speed(d / t);
}
auto v3 = avg_speed(220. * isq::distance[km],
                    2 * h);
// 110.0 km/h
auto v4 = avg_speed(isq::distance(140. * mi),
                    2 * h);
// 70.0 mi/h
```

Quantity: “property of a phenomenon, body, or substance, where the property has a magnitude that can be expressed as a **number** and a **reference**. a reference can be a **measurement unit**...”

ISO Guide 99:2007

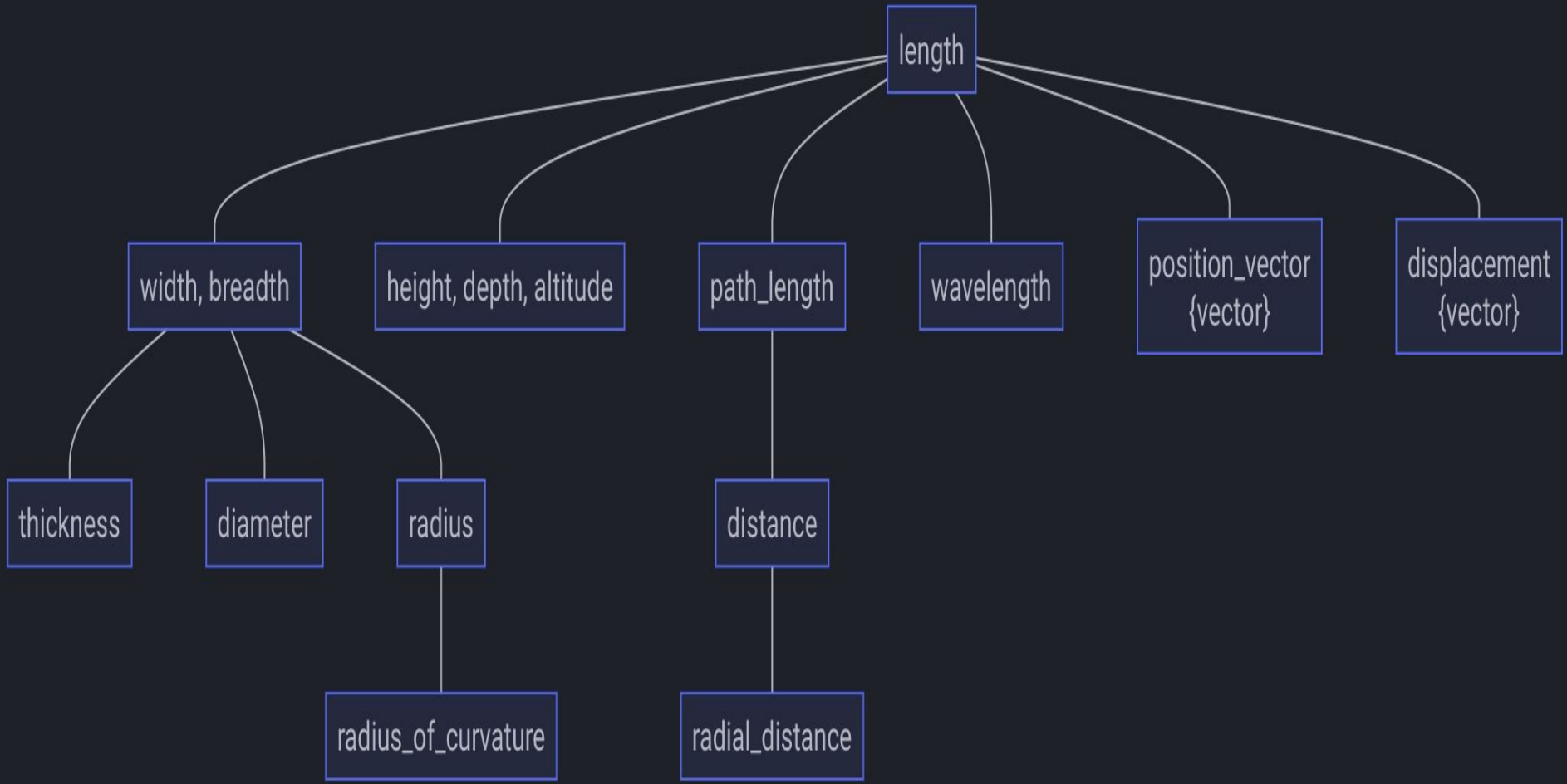

length
width, breadth
height, depth, altitude
path_length
wavelength
position_vector {vector}
displacement {vector}
thickness
diameter
radius
distance
radius_of_curvature
radial_distance

quantity kinds

```
class SquarePrism {
public:
  constexpr SquarePrism(quantity<isq::width[m]> width,
                        quantity<isq::height[m]> height)
    : m_w{width}
    , m_h{height} {}

  constexpr auto Volume() const { return m_w * m_w * m_h; }

private:
  quantity<isq::width[m], float> m_w;
  quantity<isq::height[m], float> m_h;
};
static_assert(
  SquarePrism{1 * m, 1 * m}.Volume() == 1 * pow<3>(m)
);
```

quantity kinds

```
auto p1 = SquarePrism{1 * m, 1 * m};
auto p2 = SquarePrism{1.0 * isq::width[m],
                      1.0 * isq::height[m]};

auto p3 = SquarePrism{1.0 * isq::width[km],
                      1.0 * isq::height[km]};

auto p4 = SquarePrism{isq::width(1.0 * isq::length[m]),
                      1.0 * isq::height[m]};
auto p5 = SquarePrism{
            quantity_cast<isq::width>(1.0 * isq::height[m]),
            1.0 * isq::height[m]};

auto p6 = SquarePrism{
            quantity_cast<isq::width>(1.0 * s),
             1.0 * isq::height[m]};
```


length
width, breadth
height, depth, altitude
path_length
wavelength
position_vector {vector}
displacement {vector}
thickness
diameter
radius
distance
radius_of_curvature
radial_distance

quantity kinds

```
class SquarePrism {
public:
  // ...
  constexpr void Scale(quantity<isq::width[m] /
                                isq::width[m]> scaleWidth,
                       quantity<isq::height[m] /
                                isq::height[m]> scaleHeight)
  {
    m_w *= scaleWidth;
    m_h *= scaleHeight;
  }
  // ...
};
```

quantity kinds

```
class SquarePrism {
public:
  // ...
  constexpr void Scale(quantity<one> scaleWidth,

                       quantity<one> scaleHeight)

  {
    m_w *= scaleWidth;
    m_h *= scaleHeight;
  }
  // ...
};
```

quantity kinds

```
class SquarePrism {
public:
  // ...
  constexpr void Scale(double scaleWidth,

                       double scaleHeight)

  {
    m_w *= scaleWidth;
    m_h *= scaleHeight;
  }
  // ...
};
```

quantity kinds

```
class SquarePrism {
public:
  // ...
  constexpr void ScaleWidth(double scaleWidth) {
    m_w *= scaleWidth;
  }

  constexpr void ScaleHeight(double scaleHeight) {
    m_h *= scaleHeight;
  }
  // ...
};
```

Digital audio

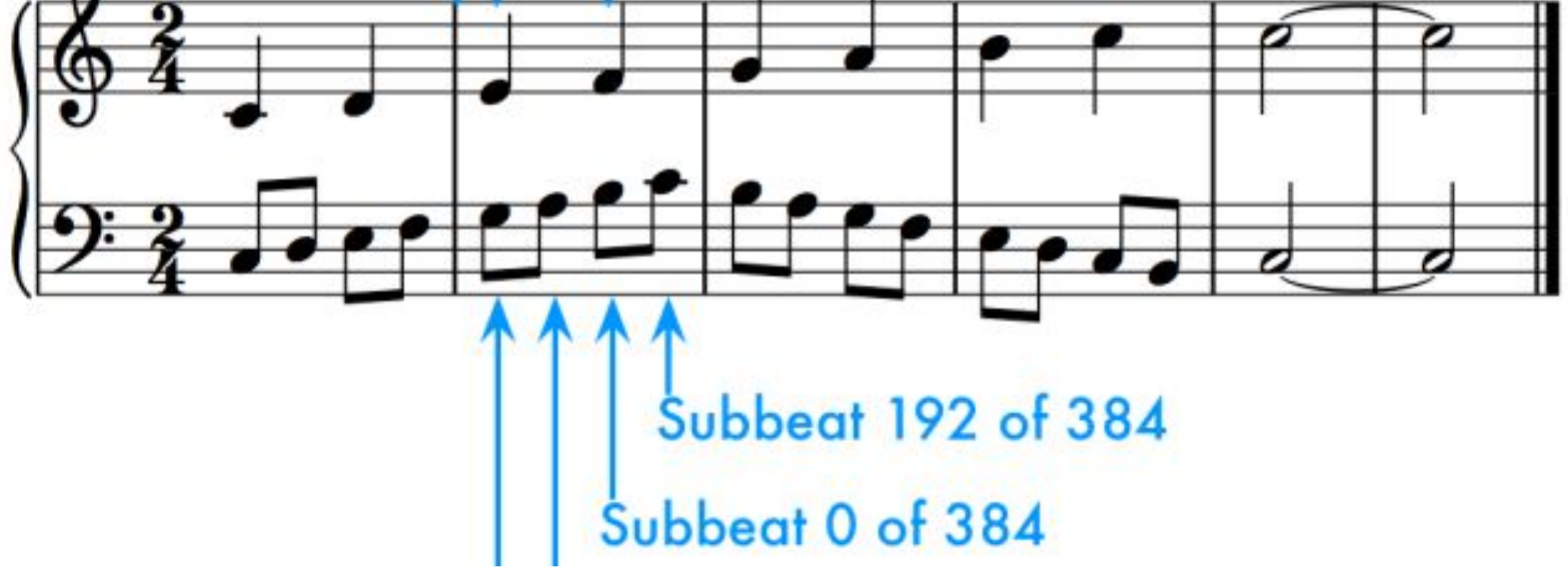
…

Digital audio considerations…

- Mostly operating on `float`
- Don’t want unnecessary conversions
- Use 3rd party APIs (`float, unsigned`)
- “Modulation rate” is a frequency (Hz)

Musical tempo

...MIDI example (demo)



https://ericjknapp.com/2019/09/26/midi-measures/

```
namespace ni {
inline constexpr struct BeatCount
  : quantity_spec<BeatCount, dimensionless, is_kind> {} BeatCount;

inline constexpr struct BeatDuration
  : quantity_spec<BeatDuration, isq::time> {} BeatDuration;

inline constexpr struct Tempo
  : quantity_spec<Tempo,
                  isq::frequency,
                  BeatCount / isq::time> {} Tempo;

inline constexpr struct QuarterNote
  : named_unit<"q", one, kind_of<BeatCount>> {} QuarterNote;

inline constexpr struct WholeNote
 : named_unit<"w", mag<4> * QuarterNote> {} WholeNote;
}
```

```
namespace ni {
inline constexpr struct BeatCount
  : quantity_spec<BeatCount, dimensionless, is_kind> {} BeatCount;

inline constexpr struct BeatDuration
  : quantity_spec<BeatDuration, isq::time> {} BeatDuration;

inline constexpr struct Tempo
  : quantity_spec<Tempo,
                  isq::frequency,
                  BeatCount / isq::time> {} Tempo;

inline constexpr struct QuarterNote
  : named_unit<"q", one, kind_of<BeatCount>> {} QuarterNote;

inline constexpr struct WholeNote
 : named_unit<"w", mag<4> * QuarterNote> {} WholeNote;
}
```

```
namespace ni {
inline constexpr struct BeatCount
  : quantity_spec<BeatCount, dimensionless, is_kind> {} BeatCount;

inline constexpr struct BeatDuration
  : quantity_spec<BeatDuration, isq::time> {} BeatDuration;

inline constexpr struct Tempo
  : quantity_spec<Tempo,
                  isq::frequency,
                  BeatCount / isq::time> {} Tempo;

inline constexpr struct QuarterNote
  : named_unit<"q", one, kind_of<BeatCount>> {} QuarterNote;

inline constexpr struct WholeNote
 : named_unit<"w", mag<4> * QuarterNote> {} WholeNote;
}
```

```
namespace ni {
inline constexpr struct BeatCount
  : quantity_spec<BeatCount, dimensionless, is_kind> {} BeatCount;

inline constexpr struct BeatDuration
  : quantity_spec<BeatDuration, isq::time> {} BeatDuration;

inline constexpr struct Tempo
  : quantity_spec<Tempo,
                  isq::frequency,
                  BeatCount / isq::time> {} Tempo;

inline constexpr struct QuarterNote
  : named_unit<"q", one, kind_of<BeatCount>> {} QuarterNote;

inline constexpr struct WholeNote
 : named_unit<"w", mag<4> * QuarterNote> {} WholeNote;
}
```

```
namespace ni {
inline constexpr struct BeatCount
  : quantity_spec<BeatCount, dimensionless, is_kind> {} BeatCount;

inline constexpr struct BeatDuration
  : quantity_spec<BeatDuration, isq::time> {} BeatDuration;

inline constexpr struct Tempo
  : quantity_spec<Tempo,
                  isq::frequency,
                  BeatCount / isq::time> {} Tempo;

inline constexpr struct QuarterNote
  : named_unit<"q", one, kind_of<BeatCount>> {} QuarterNote;

inline constexpr struct WholeNote
 : named_unit<"w", mag<4> * QuarterNote> {} WholeNote;
}
```

```
namespace ni {
inline constexpr struct BeatCount
  : quantity_spec<BeatCount, dimensionless, is_kind> {} BeatCount;

inline constexpr struct BeatDuration
  : quantity_spec<BeatDuration, isq::time> {} BeatDuration;

inline constexpr struct Tempo
  : quantity_spec<Tempo,
                  isq::frequency,
                  BeatCount / isq::time> {} Tempo;

inline constexpr struct QuarterNote
  : named_unit<"q", one, kind_of<BeatCount>> {} QuarterNote;

inline constexpr struct WholeNote
 : named_unit<"w", mag<4> * QuarterNote> {} WholeNote;
}
```

```
namespace ni {
inline constexpr struct HalfNote
  : named_unit<"h", mag<2> * QuarterNote> {} HalfNote;

inline constexpr struct DottedHalfNote
  : named_unit<"h.", mag<3> * QuarterNote> {} DottedHalfNote;

inline constexpr struct EigthNote
  : named_unit<"8th", mag<ratio{1, 2}> * QuarterNote> {} EigthNote;

inline constexpr struct DottedQuarterNote
  : named_unit<"q.", mag<3> * EigthNote> {} DottedQuarterNote;

inline constexpr struct QuarterNoteTriplet
  : named_unit<"qt", mag<ratio{1, 3}> * HalfNote> {} QuarterNoteTriplet;

inline constexpr struct SixteenthNote
  : named_unit<"16th", mag<ratio{1, 2}> * EigthNote> {} SixteenthNote;
// ...
}
```

```
namespace ni::unit_symbols {

inline constexpr auto n_wd = 3 * HalfNote;
inline constexpr auto n_w = WholeNote;
inline constexpr auto n_hd = DottedHalfNote;
inline constexpr auto n_h = HalfNote;
inline constexpr auto n_qd = DottedQuarterNote;
inline constexpr auto n_q = QuarterNote;
inline constexpr auto n_qt = QuarterNoteTriplet;
inline constexpr auto n_8thd = DottedEigthNote;
inline constexpr auto n_8th = EigthNote;
inline constexpr auto n_16th = SixteenthNote;

}
```

```
namespace ni::unit_symbols {

inline constexpr auto n_wd = 3 * HalfNote;
inline constexpr auto n_w = WholeNote;
inline constexpr auto n_hd = DottedHalfNote;
inline constexpr auto n_h = HalfNote;
inline constexpr auto n_qd = DottedQuarterNote;
inline constexpr auto n_q = QuarterNote;
inline constexpr auto n_qt = QuarterNoteTriplet;
inline constexpr auto n_8thd = DottedEigthNote;
inline constexpr auto n_8th = EigthNote;
inline constexpr auto n_16th = SixteenthNote;

}
```

Conventions in the slides

- constexpr or inline may be left off for slides
- using mp_units in all slides
- ni:: / tp:: for example code
  - unless defined on the slide
- ni::unit_symbols

```
namespace tp {
float GetTempo();

unsigned GetPPQN();

unsigned GetTransportPos();
}
```

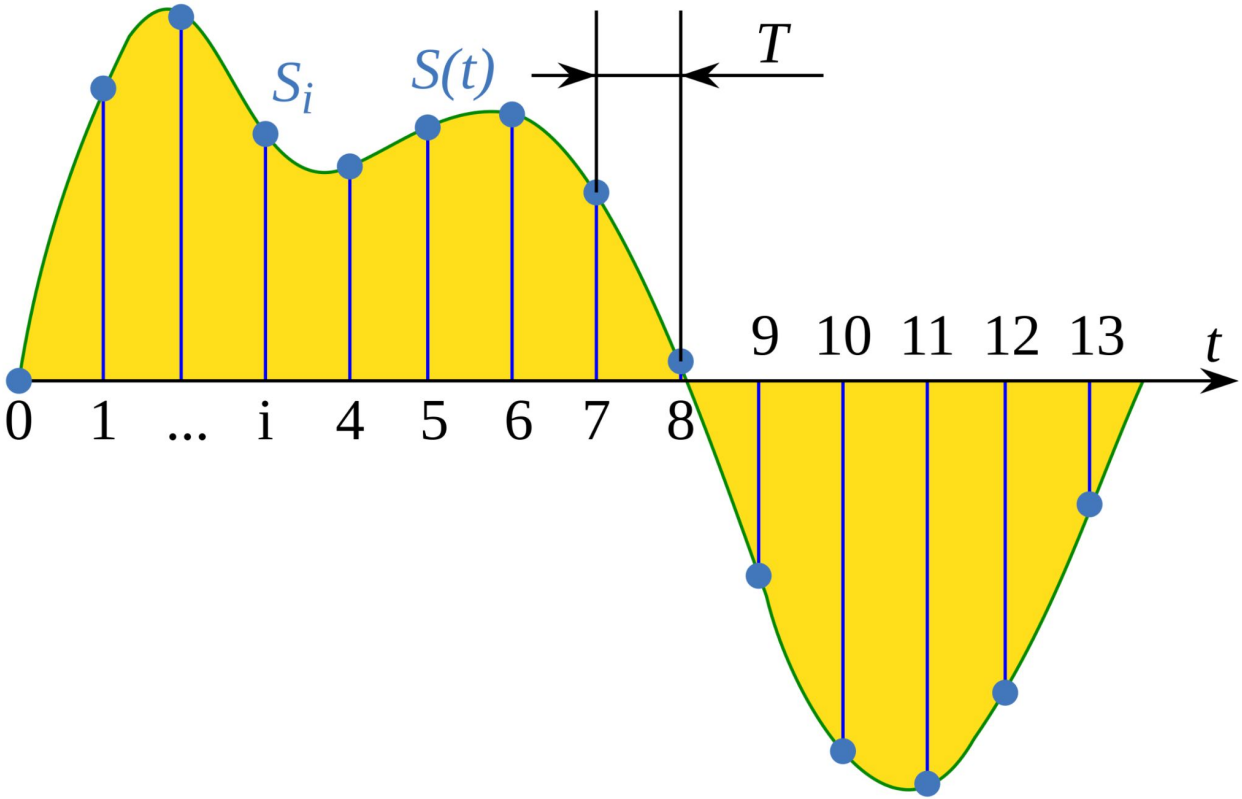
```
namespace tp {
float GetTempo();

unsigned GetPPQN();

unsigned GetTransportPos();
}

namespace ni {
quantity<BeatsPerMinute, float> GetTempo() {
  return tp::GetTempo() * BeatsPerMinute;
}
quantity<MIDIPulsePerQuarter, unsigned> GetPPQN() {
  return tp::GetPPQN() * MIDIPulsePerQuarter;
}
quantity<MIDIPulse, unsigned> GetTransportPos() {
    return tp::GetTransportPos() * MIDIPulse;
};
}
```

```
auto tempo = ni::GetTempo();
// 110 bpm
auto reverbBeats = 1 * n_qd;
// 1 q.
auto reverbTime = reverbBeats / tempo;
// 0.818182 s

auto pulsePerQuarter = value_cast<float>(ni::GetPPQN());
// 960 ppqn
auto transportPosition = ni::GetTransportPos();
// 15836 p
auto transportBeats = (transportPosition / pulsePerQuarter).in(n_q);
// 16.4958 q
auto transportTime = (transportBeats / tempo).in(s);
// 8.99773 s
```

Digital sampling...

...new time and value quantities and units

Digital sampling



By Д.Ильин: vectorization - File:Signal Sampling.png by Email4mobile (talk), CC0, https://commons.wikimedia.org/w/index.php?curid=98587159

```
namespace ni {
inline constexpr struct SampleCount
  : quantity_spec<SampleCount, dimensionless, is_kind> {} SampleCount;

inline constexpr struct SampleDuration
  : quantity_spec<SampleDuration, isq::time> {} SampleDuration;

inline constexpr struct SamplingRate
  : quantity_spec<SamplingRate,
                  isq::frequency,
                  SampleCount / isq::time> {} SamplingRate;

inline constexpr struct Sample
  : named_unit<"Smpl", one, kind_of<SampleCount>> {} Sample;

  namespace unit_symbols {
  inline constexpr auto Smpl = Sample;
  }
}
```

```
namespace tp {

float GetSampleRate();

}
```

```
namespace tp {

float GetSampleRate();

}

namespace ni {

quantity<SamplingRate[si::hertz], float> GetSampleRate() {
  return tp::GetSampleRate() * si::hertz;
}

}
```

```
auto sr1 = ni::GetSampleRate();
// 44100 Hz
auto sr2 = 48000.f * (Smpl / s);
// 48000 Smpl/s

auto samples = 512 * Smpl;
// 512 Smpl

auto sampleTime1 = (samples / sr1).in(s);
// 0.01161 s
auto sampleTime2 = (samples / sr2).in(ms);
// 10.6667 ms

auto sampleDuration1 = (1 / sr1).in(ms);
// 0.0226757 ms
auto sampleDuration2 = (1 / sr2).in(ms);
// 0.0208333 ms
```

```
namespace ni {
inline constexpr struct UnitSampleAmount
  : quantity_spec<UnitSampleAmount,
                  dimensionless, is_kind> {} UnitSampleAmount;
//! non-negative
inline constexpr auto Amplitude = UnitSampleAmount;
//! non-negative
inline constexpr auto Level = UnitSampleAmount;
//! non-negative
inline constexpr struct Power
  : quantity_spec<Power, Level * Level> {} Power;

// Ethiers?
inline constexpr struct SampleValue
  : named_unit<"PCM", one, kind_of<UnitSampleAmount>> {} SampleValue;

namespace unit_symbols {
  inline constexpr auto Smpl = Sample;
  inline constexpr auto pcm = SampleValue;
}
}
```

Non-negative level and power...

...oh and what about decibels? (demo)

```
inline constexpr struct validated_tag {
} validated;

template <std::movable T, std::predicate<T> Validator> class validated_type {
  T value_;

public:
  using value_type = T;

  static constexpr bool validate(const T& value) { return Validator()(value); }

  constexpr validated_type() : value_(0) {}

  constexpr explicit validated_type(const T& value) noexcept(
    std::is_nothrow_copy_constructible_v<T>) requires std::copyable<T> : value_(value) {
    gsl_Expects(validate(value_));
  }

  constexpr explicit validated_type(T&& value) noexcept(std::is_nothrow_move_constructible_v<T>)
    : value_(std::move(value)) {
    gsl_Expects(validate(value_));
  }

  constexpr validated_type(const T& value, validated_tag) noexcept(
    std::is_nothrow_copy_constructible_v<T>) requires std::copyable<T> : value_(value) {}

  constexpr validated_type(T&& value,
                           validated_tag) noexcept(std::is_nothrow_move_constructible_v<T>)
    : value_(std::move(value)) {}

  constexpr explicit(false)
  operator T() const& noexcept(std::is_nothrow_copy_constructible_v<T>) requires std::copyable<T> {
    return value_;
  }

  constexpr explicit(false) operator T() && noexcept(std::is_nothrow_move_constructible_v<T>) {
    return std::move(value_);
  }

  constexpr T& value() & noexcept = delete;
  constexpr const T& value() const& noexcept {
    return value_;
  }
  constexpr T&& value() && noexcept {
    return std::move(value_);
  }
  constexpr const T&& value() const&& noexcept {
    return std::move(value_);
  }

  bool operator==(const validated_type&) const requires std::equality_comparable<T>
  = default;
  auto operator<=>(const validated_type&) const requires std::three_way_comparable<T>
  = default;
};

template <typename T, typename Validator>
inline constexpr bool mp_units::is_scalar<validated_type<T, Validator>> = mp_units::is_scalar<T>;

template <typename T, typename Validator>
inline constexpr bool mp_units::treat_as_floating_point<validated_type<T, Validator>> =
  mp_units::treat_as_floating_point<T>;

template <typename CharT, typename Traits, typename T, typename Validator>
std::basic_ostream<CharT, Traits>&
operator<<(std::basic_ostream<CharT, Traits>& os,
           const validated_type<T, Validator>& v) requires requires {
  os << v.value();
}
{ return os << v.value(); }

template <typename T, typename Validator>
struct MP_UNITS_STD_FMT::formatter<validated_type<T, Validator>> : formatter<T> {
  template <typename FormatContext>
  auto format(const validated_type<T, Validator>& v, FormatContext& ctx) {
    return formatter<T>::format(v.value(), ctx);
  }
};
```

```
namespace ni {

template <std::floating_point ValueType>
inline constexpr auto is_non_negative = [](ValueType x) { return x >=
0.f; };

template <std::floating_point ValueType>
using is_non_negative_t = decltype(is_non_negative<ValueType>);

template <std::floating_point ValueType>
using non_negative = validated_type<ValueType,
is_non_negative_t<ValueType>>;

template <std::floating_point SampleType = float>
using SamplePowerQuantity =
  quantity<Power[pow<2>(SampleValue)], non_negative<SampleType>>;

}
```

```
namespace ni {

template <std::floating_point SampleType = float>
using SamplePowerQuantity =
  quantity<Power[SampleValue * SampleValue], non_negative<SampleType>>;

}



auto sampleValue = -0.4f * ni::SampleValue;
// -0.4 PCM
auto power1 = ni::SamplePowerQuantity<float>(sampleValue * sampleValue);
// 0.16 PCM²

auto power2 = ni::SamplePowerQuantity<float>(
                    -0.2 * pow<2>(ni::SampleValue));
```

Non-linear quantity/unit relationships

- ISO 80000 guidance for logarithmic units
  - e.g. Sound pressure level
- Decibels need special handling beyond log
- Leave other non-linear relationships to the user
  - Free functions instead of customization?

```
const auto u_0 = 0.775 * V;
const auto v_0 = 1.0 * V;

auto v1 = 0.9 * V;
auto v2 = 1.2 * V;

quantity<dBu> dbu1 = 20.0 * std::log10(v1 / u_0);
quantity<dBu> dbu2 = 20.0 * std::log10(v2 / u_0);

quantity<dBV> dbv1 = 20.0 * std::log10(v1 / v_0);
quantity<dBV> dbv2 = 20.0 * std::log10(v2 / v_0);

quantity<dB> db1 = dbu1 - dbu2;
auto dbx = dbu1 + dbu2;

quantity<dBV> dbv3 = dbv1 + db1;
quantity<dBV> dbv4 = dbv1 - dbu1;

quantity<dBFS> dbfs1 = 20.0 * std::log10(0.8 * pcm / 1.0 pcm);
quantity<dBFS> dbfs2 = dbfs1 + db1; // ???
```

```
namespace ni {
void SetLevel(quantity<pcm> level);
}

const auto l_0 = 1.0 * pcm;
const auto p_0 = 1.0 * pow<2>(pcm);

auto l = 0.9 * V;
auto p = l * l;

quantity<dBFS> dbfs_root_power = 20.0 * std::log10(l / l_0);
ni::SetLevel(dbfs_root_power);

quantity<dBFS> dbfs_power = 10.0 * std::log10(p / p_0);
ni::SetLevel(dbfs_power);
```

```
template <typename Scalar = float>
class DecibelFS : public boost::additive<DecibelFS<Scalar>,
                                         DecibelGain<Scalar>>,
                  public boost::totally_ordered<DecibelFS<Scalar>> {
public:
  static_assert(std::is_floating_point_v<Scalar>);

  constexpr explicit DecibelFS(Scalar quantity);
  explicit DecibelFS(RootPowerLevel<Scalar> amplitudeFS);

  Scalar Quantity() const;

  friend constexpr DecibelFS& operator+=(DecibelFS&, DecibelGain<Scalar>);
  friend constexpr DecibelFS& operator-=(DecibelFS&, DecibelGain<Scalar>);
  friend constexpr DecibelGain<Scalar> operator-(DecibelFS, DecibelFS);
  friend constexpr bool operator==(DecibelFS, DecibelFS);
  friend constexpr bool operator<(DecibelFS, DecibelFS);
};
```

```
template <typename Scalar = float>
class DecibelFS : public boost::additive<DecibelFS<Scalar>,
                                         DecibelGain<Scalar>>,
                  public boost::totally_ordered<DecibelFS<Scalar>> {
public:
  static_assert(std::is_floating_point_v<Scalar>);

  constexpr explicit DecibelFS(Scalar quantity);
  explicit DecibelFS(RootPowerLevel<Scalar> amplitudeFS);

  Scalar Quantity() const;

  friend constexpr DecibelFS& operator+=(DecibelFS&, DecibelGain<Scalar>);
  friend constexpr DecibelFS& operator-=(DecibelFS&, DecibelGain<Scalar>);
  friend constexpr DecibelGain<Scalar> operator-(DecibelFS, DecibelFS);
  friend constexpr bool operator==(DecibelFS, DecibelFS);
  friend constexpr bool operator<(DecibelFS, DecibelFS);
};
```

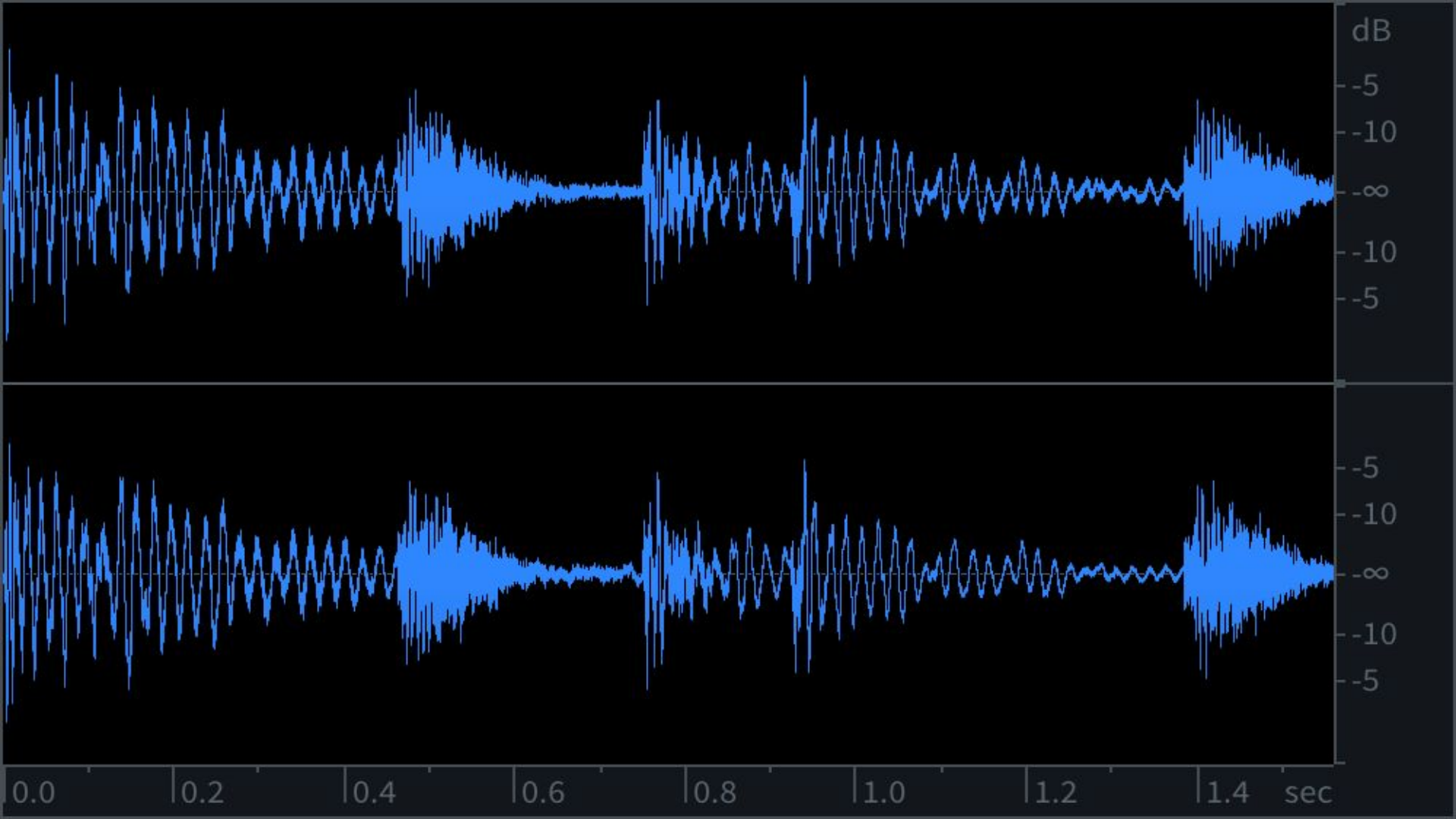
```
template <typename Scalar = float>
class DecibelFS : public boost::additive<DecibelFS<Scalar>,
                                         DecibelGain<Scalar>>,
                  public boost::totally_ordered<DecibelFS<Scalar>> {
public:
  static_assert(std::is_floating_point_v<Scalar>);

  constexpr explicit DecibelFS(Scalar quantity);
  explicit DecibelFS(RootPowerLevel<Scalar> amplitudeFS);

  Scalar Quantity() const;

  friend constexpr DecibelFS& operator+=(DecibelFS&, DecibelGain<Scalar>);
  friend constexpr DecibelFS& operator-=(DecibelFS&, DecibelGain<Scalar>);
  friend constexpr DecibelGain<Scalar> operator-(DecibelFS, DecibelFS);
  friend constexpr bool operator==(DecibelFS, DecibelFS);
  friend constexpr bool operator<(DecibelFS, DecibelFS);
};
```

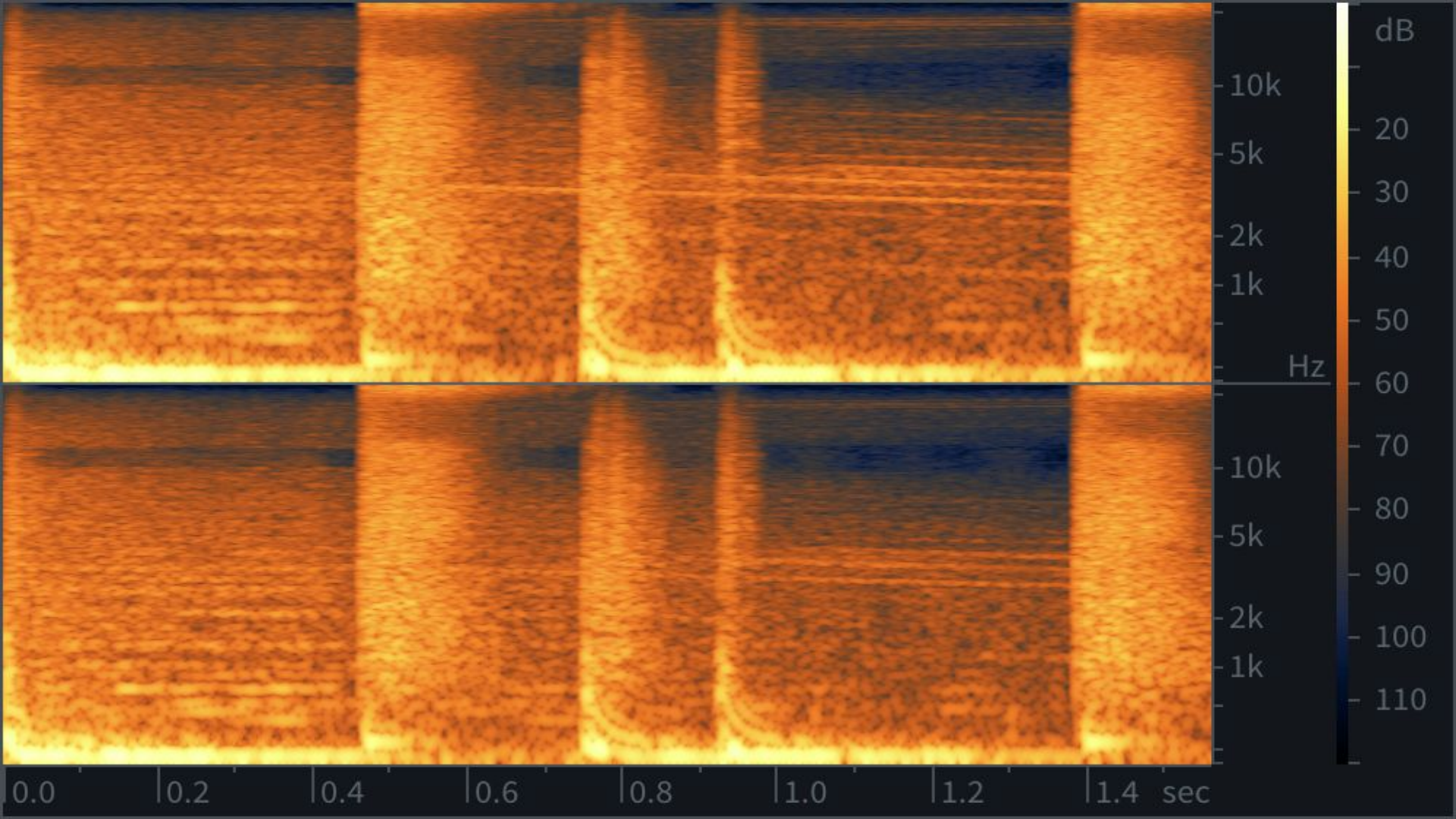
```
template <typename Scalar = float>
class RootPowerLevel : public boost::multiplicative<RootPowerLevel<Scalar>,
                                                    RootPowerGain<Scalar>>,
                       public boost::totally_ordered<RootPowerLevel<Scalar>>
{
public:
  static_assert(std::is_floating_point_v<Scalar>);

  constexpr explicit RootPowerLevel(Scalar quantity);
  explicit RootPowerLevel(DecibelFS<Scalar> decibelFS);

  Scalar Quantity() const;

  friend constexpr RootPowerLevel& operator*=(RootPowerLevel& lhs,
                                              RootPowerGain<Scalar> rhs);
  friend constexpr RootPowerLevel& operator/=(RootPowerLevel& lhs,
                                              RootPowerGain<Scalar> rhs);
  friend constexpr RootPowerGain<Scalar> operator/(RootPowerLevel lhs,
                                                   RootPowerLevel rhs);
  friend constexpr bool operator==(RootPowerLevel lhs, RootPowerLevel rhs);
  friend constexpr bool operator<(RootPowerLevel lhs, RootPowerLevel rhs);
};
```

```
template <typename Scalar = float>
class RootPowerLevel : public boost::multiplicative<RootPowerLevel<Scalar>,
                                                    RootPowerGain<Scalar>>,
                       public boost::totally_ordered<RootPowerLevel<Scalar>>
{
public:
  static_assert(std::is_floating_point_v<Scalar>);

  constexpr explicit RootPowerLevel(Scalar quantity);
  explicit RootPowerLevel(DecibelFS<Scalar> decibelFS);

  Scalar Quantity() const;

  friend constexpr RootPowerLevel& operator*=(RootPowerLevel& lhs,
                                              RootPowerGain<Scalar> rhs);
  friend constexpr RootPowerLevel& operator/=(RootPowerLevel& lhs,
                                              RootPowerGain<Scalar> rhs);
  friend constexpr RootPowerGain<Scalar> operator/(RootPowerLevel lhs,
                                                   RootPowerLevel rhs);
  friend constexpr bool operator==(RootPowerLevel lhs, RootPowerLevel rhs);
  friend constexpr bool operator<(RootPowerLevel lhs, RootPowerLevel rhs);
};
```

Non-linear quantity/unit relationships

- ISO 80000 guidance for logarithmic units
  - e.g. Sound pressure level
- Decibels need special handling beyond log
- Leave other non-linear relationships to the user
  - Free functions instead of customization?


dB
-5
-10
-∞
-10
-5
-5
-10
-∞
-10
-5
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
sec


10k
5k
2k
1k
Hz
10k
5k
2k
1k
dB
20
30
40
50
60
70
80
90
100
110
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
sec

Interfacing with 3rd-party APIs

…

```
namespace tp {

class Processor {
public:
  virtual void Process(std::span<float> buf) = 0;
};

}
```

```
namespace ni {

using ProcessQuantityT =
mp_units::quantity<UnitSampleAmount[SampleValue], float>;

template <typename T>
concept InPlaceProcessor = requires(T t) {
  t.Process(std::span<ProcessQuantityT>{});
};

template <typename T> class ProcessorImpl : public tp::Processor {
public:
  void Process(std::span<float> buf) {
    m_impl.Process({reinterpret_cast<ProcessQuantityT*>(buf.data()),
                                                    buf.size()});
  }
private:
  T m_impl{};
};
}
```

```
namespace ni {
class LevelPrinter {
public:
  void Process(std::span<ProcessQuantityT> buf) {
    std::cout << '{';
    for (auto i = 0u; i < buf.size(); ++i) {
      if (i != 0u) {
        std::cout << ", ";
      }
      std::cout << buf[i];
    }
    std::cout << "}\n";
  }
};
}

int main() {
  auto processor = ni::ProcessorImpl<ni::LevelPrinter>{};
  auto buf = std::array{1.f, 2.f, 0.4f, 0.2f};
  processor.Process(buf); // {1 PCM, 2 PCM, 0.4 PCM, 0.2 PCM}
}
```

```
namespace tp {

void AddConstant(std::size_t n, float* srcDst, float x) {
  std::transform(srcDst, srcDst + n, srcDst, [&](float val) {
    return val + x;
  });
}

void MulConstant(std::size_t n, float* srcDst, float x) {
  std::transform(srcDst, srcDst + n, srcDst, [&](float val) {
    return val * x;
  });
}

}
```

```
namespace tp {
void AddConstant(std::size_t n, float* srcDst, float x);
}

namespace ni {

template <auto R1, auto R2>
auto AddConstant(std::span<quantity<R1, float>> src,
                 quantity<R2, float> x) requires requires {
                                          { src[0] + x } -> Quantity;
                                         } {
  tp::AddConstant(src.size(),
                  reinterpret_cast<float*>(src.data()),
                  x.numerical_value_in(x.unit));
  return std::span{reinterpret_cast<decltype(src[0] + x)*>(src.data()),
                   src.size()};
}

}
```

```
namespace tp {
void MulConstant(std::size_t n, float* srcDst, float x);
}

namespace ni {

template <auto R>
auto MulConstant(std::span<quantity<R, float>> src,
                 float x) requires requires {
                             { src[0] * x } -> Quantity;
                           } {
  tp::MulConstant(src.size(),
                  reinterpret_cast<float*>(src.data()),
                  x);
  return std::span{reinterpret_cast<decltype(src[0] * x)*>(src.data()),
                                                        src.size()};
}

}
```

```
namespace tp {
void MulConstant(std::size_t n, float* srcDst, float x);
}

namespace ni {

template <auto R1, auto R2>
auto MulConstant(std::span<quantity<R1, float>> src,
                 quantity<R2, float> x) requires requires {
                                          { src[0] * x } -> Quantity;
                                        } {
  tp::MulConstant(src.size(),
                  reinterpret_cast<float*>(src.data()),
                  x.numerical_value_in(x.unit));
  return std::span{reinterpret_cast<decltype(src[0] * x)*>(src.data()),
                   src.size()};
}

}
```

```
auto input = std::array{1.f, 2.f, 3.f, 4.f};
// {1.0f, 2.0f, 3.0f, 4.0f}

tp::MulConstant(input, 10.f);
// {10.0f, 20.0f, 30.0f, 40.0f}

tp::AddConstant(input, -1.f);
// {9.0f, 19.0f, 29.0f, 39.0f}

tp::MulConstant(input, 2.f);
// {18.0f, 38.0f, 58.0f, 78.0f}
```

```
auto input = std::array{1.f * pcm, 2.f * pcm, 3.f * pcm, 4.f * pcm};
// {1 PCM, 2 PCM, 3 PCM, 4 PCM}

auto output1 = ni::MulConstant<pcm>(input1, 10.f);
// {10 PCM, 20 PCM, 30 PCM, 40 PCM}

auto output2 = ni::AddConstant<pcm, pcm>(output1, -1.f * pcm);
// {9 PCM, 19 PCM, 29 PCM, 39 PCM}

auto output3 = ni::MulConstant<pcm, pcm>(output2, 2.f * pcm);
// {18 PCM², 38 PCM², 58 PCM², 78 PCM²}
```

```cpp
auto input = std::array{1.f * pcm, 2.f * pcm, 3.f * pcm, 4.f * pcm};
// {1 PCM, 2 PCM, 3 PCM, 4 PCM}

auto output1 = ni::MulConstant<pcm>(input1, 10.f);
// {10 PCM, 20 PCM, 30 PCM, 40 PCM}

auto output2 = ni::AddConstant<pcm, pcm>(output1, -1.f * pcm);
// {9 PCM, 19 PCM, 29 PCM, 39 PCM}

auto output3 = ni::MulConstant<pcm, pcm>(output2, 2.f * pcm);
// {18 PCM², 38 PCM², 58 PCM², 78 PCM²}
```

```cpp
auto input = std::array{1.f * pcm, 2.f * pcm, 3.f * pcm, 4.f * pcm};
// {1 PCM, 2 PCM, 3 PCM, 4 PCM}

auto output1 = ni::MulConstant<pcm>(input1, 10.f);
// {10 PCM, 20 PCM, 30 PCM, 40 PCM}

auto output2 = ni::AddConstant<pcm, pcm>(output1, -1.f * pcm);
// {9 PCM, 19 PCM, 29 PCM, 39 PCM}

auto output3 = ni::MulConstant<pcm, pcm>(output2, 2.f * pcm);
// {18 PCM², 38 PCM², 58 PCM², 78 PCM²}
```

```
namespace ni {
template <class T>
concept FloatQuantity = Quantity<T> &&
                        std::same_as<typename T::rep, float>;

template <class T>
concept FloatQuantitySpan =
  FloatQuantity<typename T::element_type> &&
                std::same_as<T, std::span<typename T::element_type,
                                          T::extent>>;

template <FloatQuantitySpan QA, FloatQuantity Q>
auto AddConstant(QA src, Q x) requires requires {
                                  { src[0] + x } -> Quantity;
                                } {
  tp::AddConstant(src.size(),
                  reinterpret_cast<float*>(src.data()),
                  x.numerical_value_in(x.unit));
  return std::span{reinterpret_cast<decltype(src[0] + x)*>(src.data()),
                                                           src.size()};
} }
```

```
namespace ni {
template <class T>
concept FloatQuantity = Quantity<T> &&
                        std::same_as<typename T::rep, float>;

template <class T>
concept FloatQuantitySpan =
  FloatQuantity<typename T::element_type> &&
                std::same_as<T, std::span<typename T::element_type,
                                          T::extent>>;

template <class T, class U>
requires(!FloatQuantitySpan<T>) && (!FloatQuantity<U>)
auto AddConstant(T&& src, U&& x) {
  return AddConstant(std::span{std::forward<T>(src)},
                     std::forward<U>(x)); }
}
```

```
namespace ni {
template <class T>
concept FloatQuantity = Quantity<T> &&
                        std::same_as<typename T::rep, float>;

template <class T>
concept FloatQuantitySpan =
  FloatQuantity<typename T::element_type> &&
                std::same_as<T, std::span<typename T::element_type,
                                          T::extent>>;

auto MulConstant(FloatQuantitySpan auto src, float x) requires
       requires{ { src[0] * x } -> Quantity; } {
  tp::MulConstant(src.size(),
                  reinterpret_cast<float*>(src.data()),
                  x);
  return std::span{reinterpret_cast<decltype(src[0] * x)*>(src.data()),
                                                           src.size()};
}

}
```

```
namespace ni {
template <class T>
concept FloatQuantity = Quantity<T> &&
                        std::same_as<typename T::rep, float>;

template <class T>
concept FloatQuantitySpan =
  FloatQuantity<typename T::element_type> &&
                std::same_as<T, std::span<typename T::element_type,
                                          T::extent>>;

template <FloatQuantitySpan QA, FloatQuantity Q>
auto MulConstant(QA src, Q x) requires requires {
                                         { src[0] * x } -> Quantity;
                                       } {
  tp::MulConstant(src.size(),
                  reinterpret_cast<float*>(src.data()),
                  x.numerical_value_in(Q::unit));
  return std::span{reinterpret_cast<decltype(src[0] * x)*>(src.data()),
                                                           src.size()};
} }
```

```
namespace ni {
template <class T>
concept FloatQuantity = Quantity<T> &&
                        std::same_as<typename T::rep, float>;

template <class T>
concept FloatQuantitySpan =
  FloatQuantity<typename T::element_type> &&
                std::same_as<T, std::span<typename T::element_type,
                                          T::extent>>;

template <class T, class U>
requires(!FloatQuantitySpan<T>) && (!FloatQuantity<U>)
auto MulConstant(T&& src, U&& x) {
  return MulConstant(std::span{std::forward<T>(src)},
                     std::forward<U>(x));
}

}
```

```
auto input = std::array{1.f * pcm, 2.f * pcm, 3.f * pcm, 4.f * pcm};
// {1 PCM, 2 PCM, 3 PCM, 4 PCM}

auto output1 = ni::MulConstant(input1, 10.f);
// {10 PCM, 20 PCM, 30 PCM, 40 PCM}

auto output2 = ni::AddConstant(output1, -1.f * pcm);
// {9 PCM, 19 PCM, 29 PCM, 39 PCM}

auto output3 = ni::MulConstant(output2, 2.f * pcm);
// {18 PCM², 38 PCM², 58 PCM², 78 PCM²}
```

What’s next?

Are physical unit libraries for us?

- Do all my examples work in Xcode 15?
- Should we use Au in the meantime?
- Contribute to decibel units in mp-units
- Experiment with FFT and complex numbers
- Papers (not by me):
  - P2980 A motivation, scope, and plan for a physical quantities and units library
  - P2981 Improving our safety with a physical quantities and units library
  - P2982 std::quantity as a numeric type

Want to try mp-units today?

...use a compatibility macro.

Thank you...

- To my (current and former) colleagues:
  - Alex Fink
  - Dr. Cory Goldsmith
  - Russell McClellan
  - Nico Toy
- And for this talk:
  - Mateusz Pusz

Thank you!

Roth Michaels (he/him)
Principal Software Engineer
roth.michaels@native-instruments.com
@thevibesman