An Efficient and Versatile Signal Representation in the INET Physical Layer
Motivation

- **INET 4** already provides several signal representations
  - Unit disk, scalar and dimensional
  - Dimensional model is based on **MiXiM**
- Problems with current dimensional model
  - **10+ times slower** than equivalent scalar model
  - Has open bugs which are very difficult to fix
  - Eager computation model makes it harder to speed up
  - Iterator API makes it difficult to extend
Benefits

- Representation for any kind of signals (time + frequency domains)
  - OFDM, FHSS, UWB, etc.
- Mix different wireless technologies arbitrarily
- Comparable performance to equivalent scalar representation and scale well for others
- Scale to large networks with small memory footprint
- Live visualization of transmission medium spectrum (space + time + frequency domains)
Live Demos

IEEE 802.11 – WIFI
IEEE 802.15.4 – WPAN
Hypothetical UWB
Network and Configuration

- No changes to the network

```network CrosstalkShowcaseBaseNetwork
{
    submodules:
    physicalEnvironment: PhysicalEnvironment;
    configurator: Ipv4NetworkConfigurator;
    radioMedium: Ieee80211DimensionalRadioMedium;
    visualizer: IntegratedVisualizer;
    probe: Probe;
    sender1: AdhocHost;
    receiver1: AdhocHost;
    sender2: AdhocHost;
    receiver2: AdhocHost;
}
```

- Easily switch from scalar to multidimensional model

```.
*.radioMedium.analogModel.typename = "DimensionalAnalogModel"
*.radioMedium.backgroundNoise.typename = "IsotropicDimensionalBackgroundNoise"
**.wlan[*].radio.typename = "Ieee80211DimensionalRadio"

**.analogModel.attenuateWithCarrierFrequency = false
**.errorModel.snirMode = "mean"
**.receiver.snirThresholdMode = "mean"
```
OFDM signal

- Represent arbitrary signal spectrum using interpolation

```python
**.transmitter.frequencyGains = "left
  c-b*1.5 -40dB
  linear
  c-b -28dB
  linear
  c-b*0.5-1MHz -20dB
  linear
  c-b*0.5+1MHz 0dB
  linear
  c+b*0.5-1MHz 0dB
  linear
  c+b*0.5+1MHz -20dB
  linear
  c+b -28dB
  linear
  c+b*1.5 -40dB
  right"
```
FHSS signal

- Transmitted signal spreads both in time and frequency

- SNIR is also represented in time and frequency domains
- Error models can vary from statistical to symbol level
UWB signals

- Describe details of UWB signals in the time domain

- Compute path loss, interference and reception in the time domain
Multidimensional Mathematical Function API

- `getRange()`, `getDomain()`
- `isFinite(Interval)`, `isZero(Interval)`
- `getValue(Point)`, `getIntegral(Interval)`
- `getMin(Interval)`, `getMax(Interval)`, `getMean(Interval)`
- `add(IFunction)`, `subtract(IFunction)`
- `multiply(IFunction)`, `divide(IFunction)`
- `print(Stream, Interval)`

- `partition(Interval, Callback)`

```cpp
template<typename R, typename D>
class INET_API IFunction : public IntrusivePtrCounter<IFunction<R, D>>
```
Mathematical Function Properties

- Primitive and composite functions (one or more domain dimensions)
- Extensible implementation with user defined functions
- Use physical units (use C++ type system to ensure dimensional correctness; self documentation)
- Small objects (reduce memory footprint)
- Shared pointers (simplify memory management and sharing)
- Lazy computation (eliminate unused intermediate results)
- Optional caching (reuse results)
Partitioning to Primitive Functions I.

- Represent a function with piecewise primitive functions

- Primitive mathematical functions:
  - Constant (over all dimensions)
  - Linear (in 1 dimension, constant in the others)
  - Bilinear (linear in 2 dimensions, constant in the others)
  - Reciprocal (in 1 dimension, constant in the others)
Partitioning to Primitive Functions II.

- Partitioning nested functions with subdivision

- Partitioning in 2 or more dimensions

$h = f + g$
Algebraic Operations I.

- **Addition/subtraction (e.g. summing total interference)**
  - constant ± constant = constant
  - linear ± constant = linear
  - linear ± linear = linear
  - reciprocal ± anything = *not supported*
  - etc.

- **Multiplication (e.g. applying transmission power)**
  - constant * constant = constant
  - constant * linear = linear
  - linear * linear = *not supported*
  - etc.
Algebraic Operations II.

- Division (e.g. calculating SNIR)
  - constant / constant = constant
  - constant / linear = reciprocal
  - linear / constant = linear
  - linear / linear = reciprocal
  - reciprocal / reciprocal = not supported
  - Etc.

- Various additional algebraic optimizations for 0 and 1 constant values
Functions Operating on Functions

- **Shifting** along the domain axes (e.g. for applying transmission central frequency and start time to a signal)
- **Approximating** by sampling and using interpolation between samples (e.g. for using a frequency dependent attenuation function)
- **Integrating** over one dimension (reduces dimensions by 1; e.g. for computing the signal power over the spectrum)
- **Memoizing** (caching results to speed up further computations)
Isotropic Background Noise

```cpp
template<typename R, typename D>
class INET_API ConstantFunction : public FunctionBase<R, D>
{
    protected:
    const R r;
}

template<typename R, typename D>
class INET_API DomainLimitedFunction : public FunctionBase<R, D>
{
    protected:
    const Ptr<const IFunction<R, D>>& f;
    const Interval<R> range;
    const typename D::I domain;
}

auto f = makeShared<ConstantFunction<WpHz, Domain<simsec, Hz>>>(p / b);
auto g = makeFirstQuadrantLimitedFunction(f);
```
Signal with Non-trivial Spectrum I.

```cpp
template<typename R, typename X>
class INET_API OneDimensionalBoxcarFunction : public FunctionBase<R, Domain<X>>
{
    protected:
    const X lower;
    const X upper;
    const R r;
}
```

```cpp
template<typename R, typename X>
class INET_API OneDimensionalInterpolatedFunction : public FunctionBase<R, Domain<X>>
{
    protected:
    const std::map<X, std::pair<R, const IInterpolator<X, R>> *> rs;
}
```

```cpp
template<typename R, typename X, typename Y>
class INET_API OrthogonalCombinatorFunction : public FunctionBase<R, Domain<X, Y>>
{
    protected:
    const Ptr<const IFunction<R, Domain<X>>> f;
    const Ptr<const IFunction<double, Domain<Y>>> g;
}
```
Signal with Non-trivial Spectrum II.

- No center frequency and start time
- Reused among several transmissions
- May be memoized if very complex

```cpp
map<Hz, pair<double, const IInterpolator<Hz, double> *>> fs;
auto f = makeShared<OneDimensionalInterpolatedFunction<double, Hz>>(fs);
auto g = makeShared<ConstantFunction<double, Domain<Hz>>>(f->getIntegral());
auto h = f->divide(g);
auto i = makeShared<OneDimensionalBoxcarFunction<double, simsec>>(start, end, l);
auto j = makeShared<OrthogonalCombinatorFunction<double, simsec, Hz>>(i, h);
auto k = makeShared<ConstantFunction<WpHz, Domain<simsec, Hz>>>(p / b);
auto l = k->multiply(j);
auto m = makeFirstQuadrantLimitedFunction(l);
```
Signal Transmission

template<typename R, typename D>
class INET_API ShiftFunction : public FunctionBase<R, D>
{
    protected:
    const Ptr<const IFunction<R, D>> f;
    const typename D::P s;
}

auto shift = Point<simsec, Hz>(startTime, centerFrequency);
auto transmission = makeShared<ShiftFunction<WpHz, Domain<simsec, Hz>>>(signal, shift);
Reception with Constant Attenuation

Using the center frequency

```
auto shift = Point<simsec, Hz>(propagationTime, 0);
auto f = makeShared<ShiftFunction< HpHz, Domain<simsec, Hz>>>(transmission, shift);
auto a = makeShared<FrequencyDependentAttenuationFunction>(tgain, rgain, tp, rp);
auto loss = a->getValue(Point<simsec, Hz>(0, cf));
auto c = makeShared<ConstantFunction<double, Domain<simsec, Hz>>>(loss);
auto reception = f->multiply(c);
```
R. with Frequency Dependent Attenuation

```
auto shift = Point<simsec, Hz>(propagationTime, 0);
auto f = makeShared<ShiftFunction<WpHz, Domain<simsec, Hz>>>(transmission, shift);
auto attenuation = makeShared<FrequencyDependentAttenuationFunction>(tgain, rgain, tp, rp);
auto g = makeShared<ApproximatedFunction<double, Domain<simsec, Hz>, 1, Hz>>(lower, upper, step, interpolator, attenuation);
auto reception = f->multiply(g);
```
Interference

template<typename R, typename D>
class INET_API SumFunction : public FunctionBase<R, D>
{
    protected:
        std::vector<Ptr<const IFункцион<R, D>>> fs;

    f = makeShared<SumFunction<WpHz, Domain<simsec, Hz>>>();
    f->addElement(backgroundNoise);
    for (auto reception : interferingReceptions)
        f->addElement(reception);

interference

just multiple additions

bg. noise

reception

reception
Signal to Noise and Interference Ratio

```
template<typename R, typename X, typename Y>
class INET_API TwoDimensionalBoxcarFunction : public FunctionBase<R, Domain<X, Y>>
{
  protected:
    const X lowerX;
    const X upperX;
    const Y lowerY;
    const Y upperY;
    const R r;
```

```
auto filter = makeShared<TwoDimensionalBoxcarFunction<
    double, simsec, Hz>>(
    startTime, endTime, startFrequency, endFrequency, 1);
auto f = reception->multiply(filter);
auto g = interference->multiply(filter);
auto snir = f->divide(g);
```

- SNIR
- division
- multiplication
- reception
- interference
- TwoDimensionalBoxcar
- bandpass filters
Transmission Medium Spectrum Visualization

```cpp
class INET_API PropagatedTransmissionPowerFunction : public FunctionBase<WpHz, Domain<m, m, m, simsec, Hz>>
{
    protected:
    const Ptr<const IFunction<WpHz, Domain<simsec, Hz>> transmissionPowerFunction;
    const Point<m, m, m> startPosition;
    const mps propagationSpeed;
}

class INET_API SpaceAndFrequencyAttenuationFunction : public FunctionBase<double, Domain<m, m, m, m, m, m, Hz>>
{
    protected:
    const Ptr<const IFunction<double, Domain<Quaternion>> transmitterAntennaGainFunction;
    const Ptr<const IFunction<double, Domain<mps, Hz>> pathLossFunction;
    const Ptr<const IFunction<double, Domain<m, m, m, m, m, m, Hz>> obstacleLossFunction;
    const Point<m, m, m> startPosition;
    const Quaternion startOrientation;
    const mps propagationSpeed;
}

auto f = makeShared<PropagatedTransmissionPowerFunction>(transmission, position, speed);
auto attenuation = makeShared<SpaceAndFrequencyAttenuationFunction>(
    antennaGain, pathLoss, obstacleLoss, position, orientation, speed);
auto g = makeShared<ApproximatedFunction<double, Domain<m, m, m, simsec, Hz>, 4, Hz>>(
    lower, upper, step, interpolator, attenuation);
auto receptionPowerFunction = f->multiply(g);
mediumPowerFunction->addElement(receptionPowerFunction);
```

*WPAN and WIFI*  
*WIFI and UWB*  
*WIFI crosstalk*
Using SNIR in the Error Model

• Using a single SNIR value for the whole signal

```cpp
auto lower = Point<simsec, Hz>(startTime, centerFrequency - bandwidth / 2);
auto upper = Point<simsec, Hz>(endTime, centerFrequency + bandwidth / 2);
auto interval = Interval<simsec, Hz>(lower, upper, @b11);
double min = snir->getMin(interval);
double mean = snir->getMean(interval);
```

- Simple error models but less accurate
- Problem with minimum SNIR: spike noise may cause reception failure
- Problem with mean SNIR: substantial noise may doesn’t cause reception failure

• Using a single SNIR value for each physical layer symbol

```cpp
for (auto time = startTime; time < endTime; time += symbolTime) {
    for (auto frequency = startFrequency; frequency < endFrequency; frequency += subcarrierBandwidth) {
        auto lower = Point<simsec, Hz>(time, frequency);
        auto upper = Point<simsec, Hz>(time + symbolTime, frequency + subcarrierBandwidth);
        auto interval = Interval<simsec, Hz>(lower, upper, @b11);
        double min = snir->getMin(interval);
        double mean = snir->getMean(interval);
    }
}
```

- More complicated error models but more accurate
Questions and Answers