MUSICA

Music theory in Scala

Contents

1. Real Intervals and Cents Intervals

`musica.math.RealInterval, CentsInterval`

A real interval is interpreted as a proportion of two positive, real-valued frequencies (or two lengths). It has two attributes: for frequencies f_1 and f_2, *value* = f_1 / f_2 and *cents* = $1200^{2} \log (f_1 / f_2)$. A value of 2 maps to 1200 cents, and value 1 to 0 cents. Values smaller then 1 map to negative cents. Values cannot be zero or negative. Two real intervals are equal if their value and their cents are equal.

Musica offers the following algebra for real intervals:

Given real intervals *a* and *b* then their sum interval *c* = *a* + *b* is defined as:
c.cents = *a.cents* + *b.cents* and consequently: *c.value* = *a.value* * *b.value*
and their subtraction is *c* = *a* - *b*:
c.cents = *a.cents* - *b.cents* and consequently: *c.value* = *a.value* / *b.value*

The negation of an interval: *c* = -*a* is defined as:
c.cents = -*a.cents* and consequently: *c.value* = 1/*a.value*

Real intervals can be (right) multiplied (and divided) by a Double: *c* = *a* * *k*, *d* = *a* / *k*
c.cents = *a.cents* * *k* and consequently: *c.value* = $a.value^{k}$
d.cents = *a.cents* / *k* and consequently: *d.value* = $a.value^{-k}$

Real intevals can be applied to a frequency *f* to obtain a new frequency: *g* = *a on f*
g = *f* * *a.value*

It is possible to normalize a real interval, which obtains the octave-equivalent interval with a value between 1 and 2 and cents between 0 and 1200.
b = *a.normalize(): b.cents = a.cents modulo 1200 (for whole values)*

The trait RealInterval is implemented principally by the immutable class CentsInterval. By default, operations on a real interval produce objects of the class CentsInterval.

Factory objects and implicit conversion are used to create RealIntervals and CentsIntervals. The (Double) parameter always indicates the size of the interval in cents.

Examples:

```
val a = CentsInterval( 200.0 )
val b = RealInterval( 1100 )  //  creates a CentsInterval
val c: RealInterval  = 1000.0  // implicit conversion

val d = (a + b) * 3 - (c / 2.5) + 230.0
println (d.value)  // 8.6238...
println (d.cents)  // 3730.0

val e = d.normalize
println (e.value)  // 1.0779...
println (e.cents)  // 130.0

println(d on 1.5) // 12.9357...

println(100 + d)  // 3830.0 - implicit conversion to double
```

2. Pure Intervals and Rationals

`musica.math.PureInterval, Rational`

Pure intervals can be expressed by the proportion of two positive integers: n / m. Musica offers a special algebra for pure intervals that allows exact computations. The class PureInterval implements the trait RealInterval, so pure intervals also have a value and cents. They can be added to or subtracted from any other real interval, but the result will be a cents interval and no exact computations will be possible on the result.

Pure intervals are extensions of the class Rational. It means that at their construction, n and m are divided by their graetest common divider to obtain the numerator (*numer*) and denominator (*denom*). The valueof a rational is a double equal to $1.0 * n / m$. Two rationals are equal when their numerators and denominators are both equal. The class Rational implements 4 operators:

$[n/m] +/- [k/l] = [(n*l +/- k*m) / m*l]$; $[n/m] * [k/l] = [n*k / m*l]$; $[n/m] / [k/l] = [n*l / m*k]$

For Pure Intervals, the algebra differs from the Rational algebra:

addition: $[n / m] + [k / l] = [(n * k) / (m * l)]$ (is a Rational mutliplication)

subtraction: $[n / m] – [k / l] = [(n * l) / (m * k)]$ (is a Rational division)

negation: $- [n / m] = [m / n]$

Pure intervals can be right-multiplied by integers:

$[n / m] * i = [n^i / m^i]$ (for $i >= 0$) $= [m^i / n^i]$ (for $i < 0$)

observe that $[n / m] * (-i) = - ([n / m] * i)$

If Doubles are used or division is applied, a CentsInterval will be produced.

When a Pure Interval is applied to a Double frequency, the result is a Double:

$[n / m]$ on $r = n * r / m$

If a Pure Interval is applied to an Int or a Rational frequency, the result is a Rational:

$[n / m]$ on $[a / b] = [n * a / m * b]$

Pure intervals can be normalized: which means that they are divided or multplied by powers of 2 such that the result lies between values 1 and 2.

The factory object PureInterval can be used to produce Pure Intervals. It also contains a set of often used intervals:

```
val Prime = PureInterval(1, 1)
val Octave = PureInterval(2, 1)
val Fifth = PureInterval(3, 2)
val Fourth = PureInterval(4, 3)
val MajorThird = PureInterval(5, 4)
val MinorThird = PureInterval(6, 5)
val MajorSeventh = PureInterval(7, 6)
val MinorSeventh = PureInterval(8, 7)
val MajorTone = PureInterval(9,8)
val MinorTone = PureInterval(10,9)
val SemiTone = PureInterval(16,15)
val PythagoreanComma = (Fifth * 12) - (Octave * 7)
val SyntonicComma = (Fifth * 4) - (MajorThird + (Octave * 2))
val EnharmonicComma = Octave - (MajorThird * 3)
```

Examples:

Creating a Pure interval:

```
val a = PureInterval( 48,36 )
println(a)        // 4/3
println(a.value)  // 1.3333...
println(a.cents)  // 498.04449
```

Some operations:

```
val b = (PureInterval.Fifth * 4) - (PureInterval.MajorThird +
       (PureInterval.Octave * 2))
println(b) // 81/80 – the Syntonic Comma
```

Conversion to CentsInterval when:

```
val c = PureInterval(23,18)
println ( c * 1.2 )  // 509.237... c - multiply by double
println ( c + 230.4) // 654.764... c - adding a cents interval, but:
println ( 100 + c)   // 524.366...  - implicit conversion to cents (Double)

val f = RealInterval(c) // explicit conversion
println(f) //  424.364... c
```

Applying to frequencies:

```
println (c on 3)             // 23/6
println (c on Rational(3,2)) // 23/12
println (c on 1.5)           // 1.916666....
```

2.1. Just Intonation, Eitz notation

There are three functions provided in the PureInterval factory object to compute Just-intonation intervals, using limit-3, limit-5 and limit-7 just intonation:

```
def JILimit7(f3: Int, f5: Int, f7: Int): PureInterval = {
      (Fifth * f3 + MajorThird * f5 + MajorSeventh * f7).normalize }
def JILimit5(f3: Int, f5: Int): PureInterval = {
      (Fifth * f3 + MajorThird * f5).normalize }
def JILimit3(f3: Int): PureInterval = {
      (Fifth * f3).normalize }
```

Example:

```
val e = PureInterval.JILimit5(4,-1)
println(e) // 81/80 – again the Syntonic comma
```

It is possible to use Eitz' notation for pure intervals in the Euler/Riemann *Tonnetz*. Use the same notations as for classic notes (see next chapter); the octave-indication is used for the Eitz-layers.

```
println(EitzInterval("C0"))    // 1/1
println(EitzInterval("E-1"))   // 5/4 = pure third
println(EitzInterval("E+1"))   // 6561/
println(EitzInterval("Bb+1"))  // 9/5
 println(EitzInterval("F##+4")) // 68630377364883/42949672960000
```

The Eitz-layers (= number of Syntonic comma's) can also be indicated separately, the octaves in the note string are ignored and should be left out:

```
println(EitzInterval("E", -1)) // 5/4 = pure third
```

Non-integer numbers can be used as well, but then cents intervals are produced:

```
println(EitzInterval("D", -1/3.0)) // third comma: 196.741... c
```

3. Classic Notes

`musica.symbol.ClassicNote`

Classic notes are most easily created using the notation: note name, deviation, octaves, in which the note names are 'A' to 'G', the deviation is one ore more '#' for sharps and one or more 'b' for flats. Octaves are indicated with '0' for the central octave (0 is default), +1, +2, etc for higher octaves and -1, -2, etc for lower octaves. Use the factory object ClassicNote:

```
val n = ClassicNote("C#+1")
val a: ClassicNote = "Abb"
```

You can use numerical parameters as well. Use step, deviation and octave. The default for step and octave is 0. Beware that the step starts with 0 for "C" and only values 0 to 6 are allowed. Deviation and octave can be any integer:

```
println( ClassicNote(2,-1,1) ) //  Eb+1
```

You can use fields *step*, *dev* and *octave* to retrieve the properties of a note.

```
println( "" + n + " "+ n.step +" "+ n.dev+ " " + n.octave ) // C#+1 0 1 1
```

The chromatic (12-scale) value of a note is indicated by the field *chr* and the corresponding midi pitch code with the field *midicode*:

```
println( "" + a + " " + a.chr + " " + a.midicode)  // Abb 7 67
```

Two notes are called enharmonic if their chromatic value is equal:

```
println( a.isEnharmonic("G")) // true

println( a == "G") // false
```

(Observe that implicit conversion from String to ClassicNote is applied.)

Classic notes can be normalized, which means setting the octave to 0.

```
println( n.normalize ) // C#
```

Classic notes can also be created by using the Circle of Fifths. At position 0 is the "C". All notes produced this way are normalized.

```
println (ClassicNote.FifthCircle(2))    // D
println (ClassicNote.FifthCircle(-13))  // Gbb
```

You can also retrieve the position on the Circle of Fifth for a given classic note (ignoring the octave), using *fifth:*

```
println( a.fifth ) // -11
```

4. Classic Intervals

`musica.symbol.ClassicInterval`

Classic intervals can be created using the standard notation: "P" for pure, "M" for major, "m" for minor, "A" and "d" for augmented and diminished, followed by the number of steps (1 is unison):

```
val s = ClassicInterval("P5")   // pure fifth
val t: ClassicInterval = "m16"  // minor sixteenth
```

In musica there are some additions to this notation. You can indicate an interval downwards by adding a minus sign.

```
val u: ClassicInterval = "-A4" // augmented fourth down
```

Intervals with larger deviations are called "irregular" and can be created using "i()" and "I()" and a number indicating the size of the deviation. This number should be positive. "i" is used for intervals smaller than standard, "I" for intervals larger than standard.

```
val v: ClassicInterval = "i(12)4"
val w: ClassicInterval = "I(10)7"
```

It is possible to create Classic Intervals using numeric parameters as well. Use parameters step and deviation. Both parameters can be any integer. Step 0 is the unison interval. Negative values for step indicate an interval downwards. The meaning of the deviation differs for intervals that can or cannot be pure. For pure intervals, 0 means pure, -1 means diminished and +1 is augmented. All other values are irregular. For non-pure intervals, 0 means major, -1 minor, -2 diminished and +1 augmented, and all other values irregular.

```
println(ClassicInterval(0))     // P1
println(ClassicInterval(2,-1))  // m3
println(ClassicInterval(7,5))   // I(5)8
```

You can retrieve the step and deviation using fields *step* and *dev*:

```
println( "" + s + " "+ s.step +" "+ s.dev ) // P5 4 0
```

It is possible to retrieve the size of the interval (chromatic distance) and the number of complete octaves encompassed:

```
println( "" + t + " "+ t.size +" "+ t.octaves ) // m16 25 2
```

Use the function *name* to get a longer description:

```
println(t.name) // Minor Second 16va
```

Intervals are called *equal* if their step and deviation are equal. They are called *enharmonic* if they have the same size.

```
println(ClassicInterval("M2") isEnharmonic "d3") // true
println(ClassicInterval("M2") == "d3")           // false
```

Classic Intervals can be created from two notes (implicit String conversion):

```
println(ClassicInterval("C", "E"))    // M3
println(ClassicInterval("F+1", "C"))  //-P11
```

A number of classic intervals have been predefined in the factory object ClassicInterval:

```
def Prime = ClassicInterval(0)
def MinorSecond = ClassicInterval(1,-1)
def MajorSecond = ClassicInterval(1)
def MinorThird = ClassicInterval(2,-1)
def MajorThird = ClassicInterval(2)
def Fourth = ClassicInterval(3)
def Fifth = ClassicInterval(4)
def MinorSixth = ClassicInterval(5,-1)
def MajorSixth = ClassicInterval(5)
def MinorSeventh = ClassicInterval(6,-1)
def MajorSeventh = ClassicInterval(6)
def Octave = ClassicInterval(7)
```

Classic intervals can be normalized, bringing them back within one octave:

```
println(t.normalize.name) // Minor Second
println(u.normalize.name) // Diminished Fifth
```

Some operations involving classic intervals and notes have been defined. Classic Intervals can be negated, added, subtracted and multiplied by integers:

```
println( s + t - u*2 ) // A26
println( -s )// -P5
```

Intervals can be applied on notes, to produce notes:

```
println (s on "C#")       // G#
println (t below "Ab+2")  // G
```

The inverse of an interval is computed on the normalized version:

```
println (s invert) // P4
println (t invert) // M7
```

5. Classic Scales

`musica.symbol.ClassicScale`

A classic scale is simply a list of classic intervals, in increasing order. There are two ways to define a scale: by providing (a List of) ClassicIntervals, or by simply providing a String with comma-separated interval names (no whitespace). Scales can be applied to a note to create a list of notes:

```
val scale1 = ClassicScale("P1,M2,M3,P4,P5,M6,M7")
val scale2 = ClassicScale("P1","M2","M3","P4","P5","M6","M7")
val l: List[ClassicInterval] = List("P1","M2","M3","P4","P5","M6","M7")
val scale3 = ClassicScale(l)
println ( scale1 on "C#" )  // List(C#, D#, E#, F#, G#, A#, B#)
```

Function step can be used to retrieve the individual intervals on the infinite scale. Step(0) is equal to the firstt interval in the list (most often P1).

```
println(scale.step(-4))          // -P5
println(scale.step(-4) on "Ab")  // Db

for (i <- -4 to 1) {
   println( ""+i+": "+(scale.step(i)) +" "+(scale.step(i) on "Ab"))
}
  // -4: -P5 Db
  // -3: -P4 Eb
  // -2: -m3 F
  // -1: -m2 G
  // 0: P1 Ab
  // 1: M2 Bb
```

Of course, pentatonic or whole-note scales are possible too:

```
val pentatonic = ClassicScale("P1,M2,P4,P5,M6")
val wholenote = ClassicScale("P1,M2,M3,A4,A5,A6")
println ( pentatonic on "F" ) // List(F, G, Bb, C+1, D+1)
println ( wholenote on "C" )  // List(C, D, E, F#, G#, A#)
```

A few scales have been predefined in the factory object ClassicScale:

```
println (ClassicScale.Major on "C")          // List(C, D, E, F, G, A, B)
println (ClassicScale.Minor on "C")          // List(C, D, Eb, F, G, Ab, Bb)
println (ClassicScale.HarmonicMinor on "C")  // List(C, D, Eb, F, G, Ab, B)
println (ClassicScale.MelodicMinor on "C")   // List(C, D, Eb, F, G, A, B)
```