Osci Drawing

Inhalt der Aufgabe

Oszilloskope sind hauptsächlich dafür bekannt, dass sie zum Visualisieren des Verlaufs von elektrischen Spannungen verwendet werden. Analoge Oszilloskope schießen mit einem Elektronenstrahl auf den Anzeigebildschirm, der durch Fluoreszenz dort aufleuchtet, wo die Elektronen auftreffen.
Mittels Elektromagneten kann der Elektronenstrahl durch zwei elektrische Signale umgeleitet werden. Ein Signal bestimmt die Verschiebung in x-Richtung, das andere in y-Richtung.
Im klassischen Anwendungsfall ist das x-Signal ein Sägezahn-Signal, das den Elektronenstrahl "langsam" von links nach rechts fahren lässt und dann sehr schnell wieder nach links springt. Dadurch, dass während dieser Links-Rechts-Bewegung das zweite Signal die y-Verschiebung (Höhe auf dem Bildschirm) bestimmt, wird der Spannungsverlauf des zweiten Signals auf dem Bildschirm sichtbar.

Es können aber auch beide Signale beliebig gewählt werden und somit die Position des Elektronenstrahls beliebig auf dem Bildschirm gesteuert werden.
Die selben beiden Signale können aber auch gleichzeitig als Töne interpretiert werden, also als ein Stereo-Audiosignal. Zum Beispiel bestimmt der Spannungsverlaufs des einen Signals die x-Position auf dem Oszilloskop und gleichzeitig Töne für den linken Lautsprecher, während der Spannungsverlauf des anderen Signals die y-Position des Elektronenstrahls manipuliert und als Ton auf dem rechten Lautsprecher ausgegeben wird.

Einige Leute haben sich eingehend damit beschäftigt, Stereosignale zu erzeugen, die einerseits interessante Bilder auf einem Oszilloskop zeichnen, sich gleichzeitig aber auch musikalisch anhören. Dieses Video (https://www.youtube.com/watch?v=4gibcRfp4zA) gibt einige interessante Einblicke dazu.

Das Hauptziel dieser Aufgabe ist es, eine `SignalFactory` -Klassen zu implementieren, die einige Methoden anbietet, die bei der Erstellung von Signalen helfen, um letztendlich verschiedene Muster und Effekte auf dem Oszilloskopbildschirm anzuzeigen.
Um die erzeugten Signale gut untersuchen zu können werden vorher zwei Plotting-Varianten implementiert. Einerseits wird mit `SignalTimePlotter` eine Klasse implementiert, die den Verlauf von Signalen über die Zeit plottet (ähnlich der klassischen Verwendung eines Oszilloskops), andererseits erlaubt `Osci2DPlotter` die Bewegung des Elektronenstrahls eines Oszilloskops, die durch ein Stereosignal verursacht wird, zu visualisieren.
Damit die geplotteten Bilder auch betrachtet werden können, wird zuvor ein Exporter implementiert, der diese als PNG-Dateien abspeichern kann.

Das implementierte 2D-Plotting funktioniert wunderbar für "statische" Zeichnungen, sobald aber Bewegung stattfindet, ist schwierig zu erkennen was passiert, weil **alle** Elektronenstrahlpositionen im Plot eingezeichnet sind. Deshalb wird zudem ein Exporter implementiert, der es erlaubt Stereosignale als Audiodateien abzuspeichern und diese dann entweder auf einem echtem Oszilloskop oder einem Oszilloskop-Emulator anzeigen zu lassen.

Viel Spaß!

Allgemeine Hinweise

- Die zu implementierenden Klassen sind bereits als Skelette vorgegeben, daher kompilieren die Tests von Anfang an.
- Ersetzen Sie die `throw new UnsupportedOperationException()` -Statements durch Ihre Implementierungen.
- Ein nicht bestander Tests bedeutet, dass sich ein Fehler in Ihrem Code befindet. Umgekehrt bedeutet ein bestander Test allerdings nicht zwangsläufig, dass der getestet Code keine Fehler enthält.
- Alle in der Aufgabenstellung geforderten Klassen und Methoden müssen implementiert werden, es dürfen zudem auch weitere Hilfsklassen und -methoden geschrieben werden. Allerdings ist es nicht möglich weitere abstrakte Methoden zu abstrakten Klassen hinzuzufügen.

1. Die abstrakte Klasse `Signal`

Diese Aufgabe interpretiert ein Signal folgendermaßen:

- Ein Signal beschreibt ein oder mehrere Werteverläufe über die Zeit.
- Der Signalverlauf wird nicht kontinuierlich gespeichert. Stattdessen definiert sich der Verlauf über die Werte (vom Typ `double`), die das Signal an gewissen Zeitpunkten annimmt (Samplepunkte).
- Samplepunkte sind zeitlich äquidistant verteilt.
- Die Samplerate gibt an, wieviele Samplepunkte eine Sekunde des Signals darstellen. Eine höhere Samplerate erlaubt eine genauere Beschreibung von Signalen, fordert aber natürlich mehr Speicherplatz.
- Wir unterscheiden endliche und unendliche Signale:
  - Beide sind nicht für negative Samplepunkt-Indizes definiert.
  - Ein endliches Signal mit Größe `n` ist für die Samplepunkt-Indizes von `0` bis `n-1` definiert.
  - Ein unendliches Signal ist für alle Samplepunkt-Indizes größer gleich `0` definiert.
- Wie bereits erwähnt wurde, kann ein Signal mehrere Werteverläufe beschreiben. Ein Velauf wird als Channel bezeichnet. Für ein Signal mit `n` Channels sind die Channel-Indizes `0` bis `n-1` gültig.

Diese Eigenschaften sind in der abstrakten Klasse `Signal` im Package `de.uniwue.jpp.oscidrawing` dargestellt. Es kann viele verschiedene Implementierungen von `Signal` geben, die verschiedenste Signalverläufe beschreiben, aber folgende Methoden sind ausreichend um generisch mit diesen zu interagieren:

- `public abstract boolean isInfinite()`
  Gibt zurück, ob es sich um ein endliches oder ein unendliches Signal handelt.
  Diese Methode muss von der jeweiligen Implementierung überschrieben werden.
- `public abstract int getSize()`
  Falls `isInfinite()` `true` liefert, ist der Rückgabewert von `getSize` nicht definiert und zu ignorieren. Ist `isInfinite()` allerdings `false`, dann wird die Anzahl der Samplepunkte dieses endlichen Signals zurückgegeben.
  Diese Methode muss von der jeweiligen Implementierung überschrieben werden.

- `public abstract int getChannelCount()`
  Gibt die Anzahl der Channels des Signals zurück.
  Diese Methode muss von der jeweiligen Implementierung überschrieben werden.
- `public abstract int getSampleRate()`
  Gibt die Samplerate des Signals zurück.
  Diese Methode muss von der jeweiligen Implementierung überschrieben werden.
- `public abstract double getValueAtValid(int channel, int index)`
  Gibt den Wert zurück, den der Werteverlauf von Channel `channel` an dem Samplepunkt `index` hat.
  Diese Methode kann dabei davon ausgehen, dass `channel` ein gültiger Channel-Index und `index` ein gültiger Samplepunkt-Index ist. Für einen Aufruf mit ungültigen Werten ist der Rückgabewert, bzw. das Verhalten der Methode nicht definiert.
  Diese Methode muss von der jeweiligen Implementierung überschrieben werden.
- `public double getDuration()`
  Falls `isInfinite()` `true` liefert, ist der Rückgabewert von `getDuration` nicht definiert und zu ignorieren. Ist `isInfinite()` allerdings `false` , dann soll diese Funktion die Gesamtdauer des Signal zurückgeben.
  Diese Methode ist nicht implementierungsabhängig und kann direkt in der abstrakten Klasse implementiert werden. Berechnen Sie die Gesamtdauer aus der Anzahl der Samplepunkte und der Samplerate und geben Sie das Ergebnis zurück.
- `public double getValueAt(int channel, int index)`
  Gibt den Wert des Signals von Channel `channel` an Stelle `index` zurück, falls das Signal dort definiert ist. Falls `channel` oder `index` ungültig sind, soll stattdessen `0` zurückgegeben werden.
  Diese Methode ist nicht implementierungsabhängig und kann direkt in der abstrakten Klasse implementiert werden.

2. Image-Export

Die Klasse `ImageExporter` aus dem Package `de.uniwue.jpp.oscidrawing.io` soll nur eine einzige Methode zur Verfügung stellen:

- `public static boolean writeToPNG(String pathWithoutSuffix, BufferedImage img)`
  Diese Methode soll an `pathWithoutSuffix` ein `".png"` anhängen und `img` unter diesem Dateipfad als PNG-Bild abspeichern. Falls dabei irgendetwas schief geht, z.B. an den gegebenen Pfad nicht geschrieben werden kann, soll `false` zurückgegeben werden, ansonsten soll `true` zurückgegeben werden.

3. Audio-Export

Die Klasse `AudioExporter` aus dem Package `de.uniwue.jpp.oscidrawing.io` soll zwei Methoden bereitstellen, die dazu dienen Signale als Audiodateien zu exportieren.
Ihr Javacode soll dabei nicht direkt Musikdateien, z.B. im MP3- oder WAV-Format, schreiben, sondern lediglich Binärdateien erzeugen, die alle Werte eines Channels eines Signals als rohe `float` -Werte enthält.
Zu implementieren sind:

- `public static boolean writeChannelToFile(String path, Signal signal, int channel)`
  Der Channel `channel` von `signal` soll ins Dateisystem nach `path` exportiert werden, wobei an `path` noch die Endung `.raw` angehängt werden soll.
  Falls `signal` unendlich ist, soll eine `IllegalArgumentException` geworfen werden.
  Falls `signal` endlich ist, dann konvertieren Sie die `double` -Werte der Samplepunkte des Channels `channel` nach `float` und schreiben die binäre Darstellung in Big-Endian-Darstellung in die Datei.
  Dazu kann z.B. die Klasse `DataOutputStream` mit der Methode `writeFloat` verwendet werden.
  Geben Sie `false` zurück, falls währenddessen ein Fehler auftritt. Ansonsten geben Sie `true` zurück.

- `public static boolean writeStereoToFiles(String path, Signal signal)`
  Diese Methode soll die beiden Channels einen Stereosignals in rohe Binärdateien schreiben.
  Falls `signal` unendlich ist, soll eine `IllegalArgumentException` geworfen werden.
  Falls `signal` nicht genau zwei Channels besitzt, soll eine `IllegalArgumentException` geworfen werden. Channel `0` von `signal` soll im Dateisystem unter `path + "left.raw"` gespeichert werden, Channel `1` soll unter `path + "right.raw"` gespeichert werden. Beim Speichern eines Channels gehen Sie genauso vor wie in `writeChannelToFile`.
  Geben Sie `false` zurück, falls währenddessen ein Fehler auftritt. Ansonsten geben Sie `true` zurück.

Optional: Der Rest dieses Abschnitts beschreibt, wie die Rohdateien in gültige Audiodateien umgewandelt werden können, um sie anzuhören oder auf einem Oszillopskop(-Emulator) anzuzeigen. Die später zu erzeugenden Signale können mit den Visualisierungen (nächster Abschnitt) ausreichend gut getestet werden.

Zum Erstellen der Audiodateien wird FFmpeg (https://ffmpeg.org/) verwendet.
Um aus der Rohdatei `signal.raw` eine Mono-Audiodatei `signal.wav` zu erzeugen:

```
ffmpeg -f f32be -ar 48000 -i signal.raw signal.wav
```

`-f f32be` gibt an, dass die Rohdatei 32-bit Floatwerte mit Big-Endian-Byteordnung enthält.
`-ar 48000` gibt an, dass die Samplerate 48000 beträgt. Dieser Wert muss an die verwendete Samplerate angepasst werden.

Um eine Stereo-Audiodatei zu erzeugen, müssen zu erst beide Monosignale mit obigen Befehl erzeugt werden. Dann können diese mit folgendem Befehl zusammengeführt werden.

```
ffmpeg -i left.wav -i right.wav -filter_complex "[0:a][1:a]join=inputs=2:channel_layo
ut=stereo[a]" -map "[a]" stereo.wav
```

Zum Visualisieren eines Stereo-Signals muss die Audiodatei einfach im Oszilloskop-Emulator (https://asdfg.me/osci/) geöffnet und abgespielt werden.

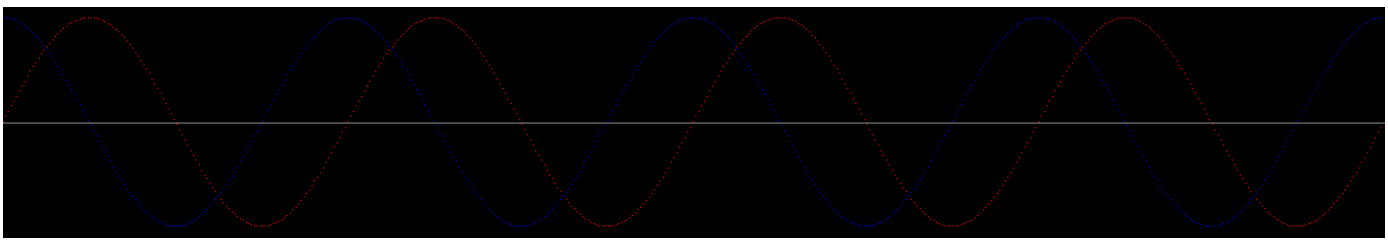
Alternativ kann statt den FFmpeg-Befehlen auch Audacity (https://www.audacity.de/), eine Audiobearbeitungsanwendung mit grafischer Oberfläche, verwendet werden.

4. Visualisierung

Signalverläufe über die Zeit plotten

Das Interface `SignalTimePlotter` im Package `de.uniwue.jpp.oscidrawing.visualization` beschreibt eine Schnittstelle, die es erlauben soll, den Verlauf der Samplepunktwerte von Channels eines Signals zu visualisieren.

Als Beispiel für solch einen Plot hier der Verlauf einer Sinus- und einer Cosinuswelle:



Schreiben Sie die Methode

```
public static SignalTimePlotter createSignalTimePlotter(int width, int height, double
valScale, double timeScale, Color bgcol, Color axiscol)
```

so um, dass sie eine Implementierung von `SignalTimePlotter` zurückgibt. Intern soll ein `BufferedImage` der Größe `width` x `height` gespeichert werden, auf dem mit Hilfe der Methode Signalverläufe eingezeichnet werden können.
Zu Beginn sollen fast alle Pixel des Bildes die Farbe `bgcol` haben, mit der Ausnahme der mittleren Pixelzeile, diese soll in `axiscol` gefärbt sein. Ist `height` ungerade, so ist die `y`-Koordinate der mittleren Zeile eindeutig. Ist `height` gerade, soll die "untere der beiden mittleren Zeilen" mit `axiscol` gefärbt werden (die Zeile mit größer `y`-Koordinate, denn die `y`-Achse läuft bei `BufferedImage` von oben nach unten).
`valScale` gibt an, wie die `y`-Achse skaliert werden soll. Z.B. werden Samplepunkte mit dem Wert `0` in der mittleren Zeile des Bildes gezeichnet. Punkte, die dem Wert `valScale` entsprechen, liegen in der obersten Zeile. Punkte, die dem Wert `-valScale` entsprechen, liegen in der untersten Zeile. Gleichermaßen beschreibt `timeScale` die Skalierung der `x-Achse`. `timeScale` ist demnach also die Grenze des Zeitintervalls, das auf dem Bild dargestellt werden soll. Das bedeutet also, dass der erste Samplepunkt (Sampleindex `0`, zum Zeitpunkt `0`) in der ersten, "linkesten" Pixelspalte liegt. Ganz rechts befindet sich jedoch der Samplepunkt, der zum Zeitpunkt `timeScale` gehört. Ist `timeScale` nun z.B. `1` kann im Bild genau eine Sekunde dargestellt werden und in der rechtesten Pixelspalte wird der Wert gezeichnet, der zum letzten Samplepunkt der ersten Sekunde des Signals gehört.

Die Methoden, die das Interface fordert, sollen sich wie folgt verhalten:

- `public int sampleIndexToImageXCoord(int sampleIndex, int sampleRate)`
  Gibt zurück, welche `x`-Koordinate ein Samplepunkt auf dem Bild hat.
  Sie können sich an dieser map-Funktion (https://www.arduino.cc/reference/en/language/functions/math/map/) orientieren. Das `x` entspricht hier dem `sampleIndex`.
  `in_min` ist der kleinste Samplepunktindex (also `0`).
  `in_max` ist der größte Samplepunktindex, der noch auf dem Bild zu sehen ist (`sampleRate` * `timeScale` - `1`).
  `out_min` ist die kleinste `x`-Koordinate (also `0`).
  `out_max` ist die größte `x`-Koordinate (`width` - `1`).
  Achten Sie darauf, dass die Rechnungen als Fließkommazahl-Rechnungen durchgeführt werden und erst das Endergebnis auf den betragsmäßig nächstkleineren `int` abgerundet wird.

- `public int signalValToImageYCoord(double val)`
  Gibt zurück, welche `y`-Koordinate ein Samplepunkt auf dem Bild hat.
  Sie können sich auch hier wieder an der map-Funktion (https://www.arduino.cc/reference/en/language/functions/math/map/) orientieren. Das `x` entspricht hier dem `val`.
  `in_min` ist der Samplepunktwert, der ganz oben im Bild gezeichnet wird (`valScale`).
  `in_max` ist der Samplepunktwert, der ganz unten im Bild gezeichnet wird (`-valScale`).
  `out_min` ist die `y`-Koordinate der obersten Pixel (also `0`).
  `out_max` ist die `y`-Koordinate der untersten Pixel (`height` - `1`).
  Achten Sie darauf, dass die Rechnungen als Fließkommazahl-Rechnungen durchgeführt werden und erst das Endergebnis auf den betragsmäßig nächstkleineren `int` abgerundet wird.

- `public void drawSignalAt(Signal signal, int channel, int index, Color col)`
  Diese Methode soll den Samplepunkt mit dem Index `index` aus dem Channel `channel` von `signal` auf dem Bild in der Farbe `col` einzeichnen.
  Bestimmen Sie dazu die Pixelkoordinaten des Samplepunktes. Liegen diese innerhalb der Bildes, soll der entsprechende Pixes mit `col` gefärbt werden. Liegen sie außerhalb des Bildes, soll diese Methode nichts tun (auch keine Exceptions werfen).

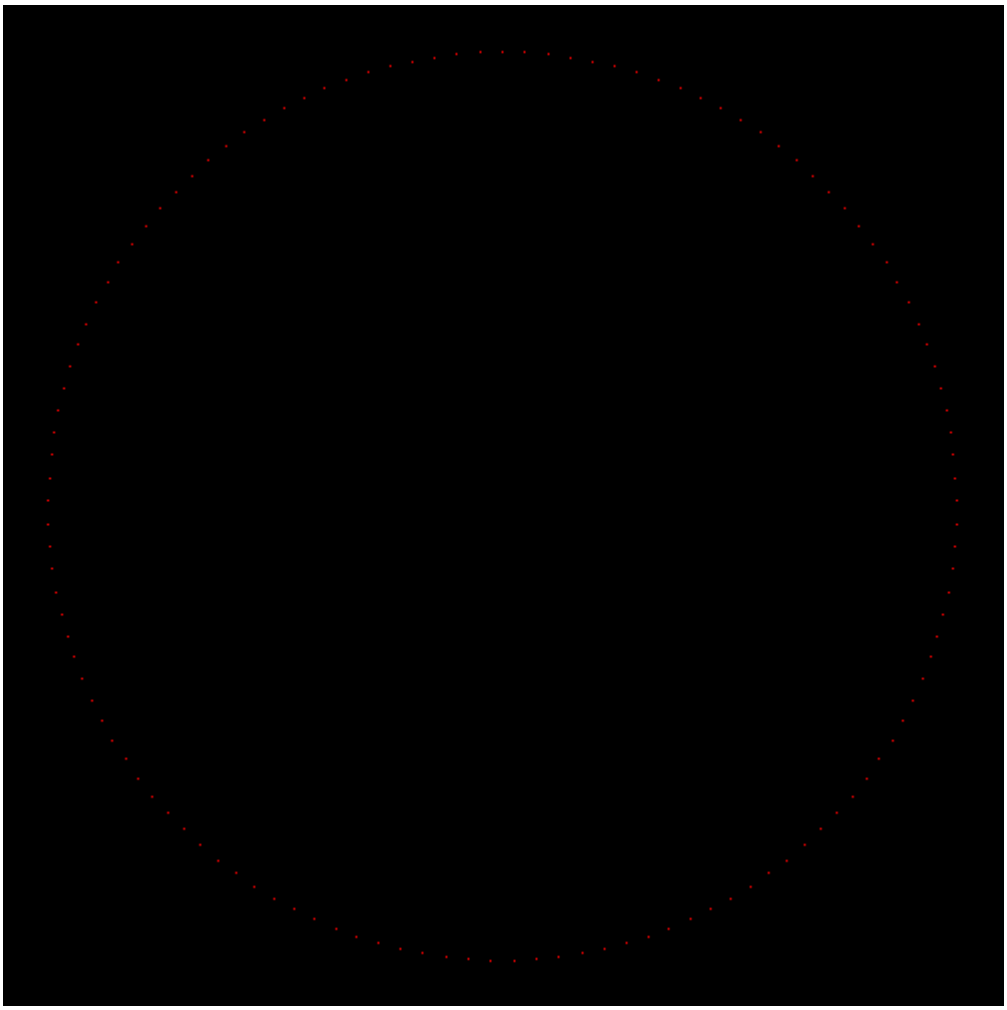
- `public void drawSignal(Signal signal, int channel, Color col)`
  Diese Methode soll den Verlauf des Channels `channel` von `signal` in der Farbe `col` auf dem Plot einzeichnen.

- `public void drawSignal(Signal signal, Color... colors)`
  Diese Methode soll alle Channels von `signal` in das Bild einzeichnen. Der `i`-te Channel soll dabei in der Farbe `colors[i]` geplottet werden.
  Falls die Anzahl der übergebenen Farben nicht mit der Anzahl der Channels von `signal` übereinstimmt, soll eine `IllegalArgumentException` geworfen werden.
- `public BufferedImage getImage()`
  Diese Methode gibt das Bild zurück, auf dem gezeichnet wurde.

2D-Oszilloskop-Plot eines Stereosignals

Das Interface `Osci2DPlotter` im Package `de.uniwue.jpp.oscidrawing.visualization` beschreibt eine Schnittstelle, die es erlauben soll, alle Positionen zu visualisieren, die der Elektronenstrahl eines Oszilloskops einnimmt, wenn die beiden Channels des Stereosignals als `x`- und `y`-Ausrichtung verwendet werden.

Zum Beispiel ergeben die Sinus- und Cosinuswellen aus dem Beispiel des vorherigen Abschnitts einen Kreis:



Schreiben Sie die Methode

```
public static Osci2DPlotter createImageCreator(int size, double scale, Color bgcol)
```

so um, dass sie eine Implementierung von `Osci2DPlotter` zurückgibt. Intern soll ein `BufferedImage` der Größe `size` x `size` gespeichert werden, auf dem gezeichnet werden soll.
Zu Beginn sollen alle Pixel des Bildes die Farbe `bgcol` haben.
`scale` gibt die Skalierung an, die für `x`- und `y`-Achse identisch ist.
Die Methoden, die das Interface fordert, sollen sich wie folgt verhalten:

- `public int signalValToImageXCoord(double val)`
  Gibt zurück, welche `x`-Koordinate ein Samplepunkt auf dem Bild hat.

Sie können sich auch hier wieder an der map-Funktion (https://www.arduino.cc/reference/en/language/functions/math/map/) orientieren. Das `x` entspricht hier dem `val`.
`in_min` ist der Samplepunktwert, der ganz links im Bild gezeichnet wird (`-scale`).
`in_max` ist der Samplepunktwert, der ganz rechts im Bild gezeichnet wird (`scale`).
`out_min` ist die `x`-Koordinate des linkesten Pixels (also `0`).
`out_max` ist die `x`-Koordinate der rechtesten Pixels (`size - 1`).
Achten Sie darauf, dass die Rechnungen als Fließkommazahl-Rechnungen durchgeführt werden und erst das Endergebnis auf den betragsmäßig nächstkleineren `int` abgerundet wird.

- `public int signalValToImageYCoord(double val)`
  Gibt zurück, welche `y`-Koordinate ein Samplepunkt auf dem Bild hat.
  Sie können sich auch hier wieder an der map-Funktion (https://www.arduino.cc/reference/en/language/functions/math/map/) orientieren. Das `x` entspricht hier dem `val`.
  `in_min` ist der Samplepunktwert, der ganz oben im Bild gezeichnet wird (`scale`).
  `in_max` ist der Samplepunktwert, der ganz rechts im Bild gezeichnet wird (`-scale`).
  `out_min` ist die `y`-Koordinate des obersten Pixels (also `0`).
  `out_max` ist die `x`-Koordinate der untersten Pixels (`size - 1`).
  Achten Sie darauf, dass die Rechnungen als Fließkommazahl-Rechnungen durchgeführt werden und erst das Endergebnis auf den betragsmäßig nächstkleineren `int` abgerundet wird.
- `public void drawSignalAt(Signal signal, int index, Color col)`
  Zeichnet einen Punkt für das Wertepaar, das das Stereosignal `signal` am Samplepunkt `index` annimmt.
  Falls `signal` nicht genau zwei Channels hat, soll eine `IllegalArgumentException` geworfen werden.
  Ansonsten verwenden Sie den Wert des `0`-ten Channels um die `x`-Koordinate des einzufärbenden Pixels zu bestimmen, sowie den Wert des `1`-ten Channels um die `y`-Koordinate zu berechnen. Falls der bestimmte Pixel innerhalb des Bildes liegt, soll dieser im Bild auf die Farbe `col` geändert werden. Falls der Pixel außerhalb liegt, soll die Methode nichts weiter tun (auch keine Exception werfen).
- `public void drawSignal(Signal signal, Color col)`
  Diese Methode soll alle Wertepaare des Stereosignals `signal` in das Bild einzeichnen.
  Falls `signal` ein unendliches Signal ist, soll eine `IllegalArgumentException` geworfen werden.
  Ansonsten zeichnen Sie die Wertepaare an allen Samplepunkten wie in `drawSignalAt` beschrieben mit der Farbe `col` in das Bild.
- `public BufferedImage getImage()`
  Diese Methode gibt das Bild zurück, auf dem gezeichnet wurde.

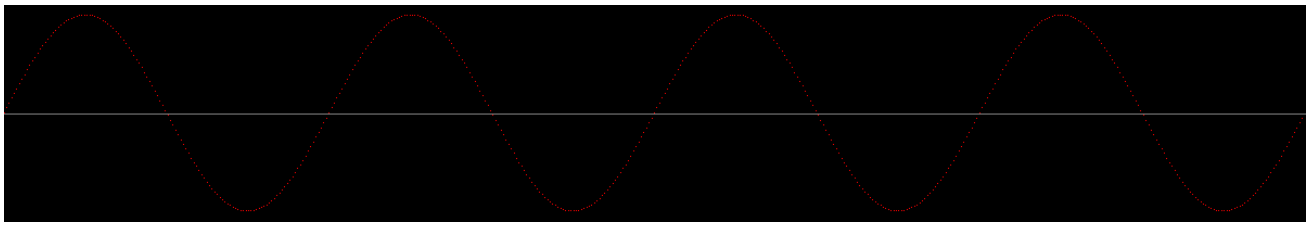
5. Signalerzeugung

Die Klasse `SignalFactory` im Package `de.uniwue.jpp.oscidrawing.generation` bietet einige Methoden zur Erzeugung von Signalen. Es bietet sich oft an zur Implementierung einer Factory-Methode andere Factory-Methoden zu verwenden.
Verwenden Sie die vorher implementierten Visualisierungstools um Ihre Signale zu untersuchen.

- `public static Signal fromValues(double[] signalData, int sampleRate)`
  Diese Methode erlaubt es eine Menge von `double`-Werten, gegeben als Array, als ein Signal darzustellen.
  Falls `sampleRate` nicht positiv ist, soll eine `IllegalArgumentException` geworfen werden.
  Das zu erzeugende Signal soll endlich sein, soviele Samplepunkte enthalten wie `signalData` Werte und genau einen Channel besitzen. Die Samplerate ist durch `sampleRate` gegeben. Der `i`-te Samplepunkt von Channel `0` soll dem `i`-ten Werten von `signalData` entsprechen.

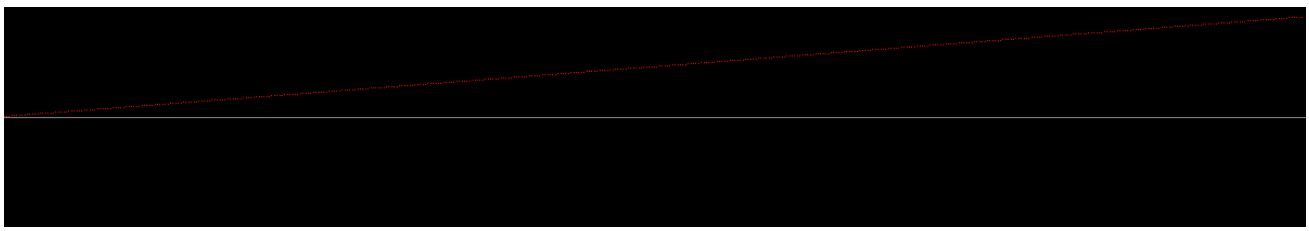
- `public static Signal wave(DoubleUnaryOperator function, double frequency, double duration, int sampleRate)`
  Diese Methode soll periodische Signale mit einer Periodenlänge von 2π erzeugen, zum Beispiel Sinus- oder Cosinus-Signale.
  Falls `frequency` nicht positiv ist, soll eine `IllegalArgumentException` geworfen werden.
  Falls `duration` nicht positiv ist, soll eine `IllegalArgumentException` geworfen werden.
  Falls `sampleRate` nicht positiv ist, soll eine `IllegalArgumentException` geworfen werden.
  Das zu erzeugende Signal soll endlich, mit Größe `sampleRate` * `duration` , sein, genau einen Channel besitzen und die Samplerate `sampleRate` haben. Zum Berechnen der Signalwerte gehen Sie folgendermaßen vor:
  - Berechnen Sie die Schrittweite `step` . Diese beschreibt, wie gestreckt oder gestaucht Werte in die Methode `function` eingesetzt werden müssen, um die gewünschte Frequenz zu erreichen. Es ist `step` = (`frequency` * 2π)/ `sampleRate` .
  - Berechnen Sie den `i` -ten Samplepunkt von Channel `0` als `function.applyAsDouble(i*step)` .

  Beispielsweise sieht für den Aufruf `wave(Math::sin, 4, 1, 500)` der Plot des Signals so aus:



- `public static Signal rampUp(double duration, int sampleRate)`
  Diese Methode soll ein Signal zurückgeben, das "den Velauf einer Rampe" besitzt, also über die Dauer `duration` von `0` linear bis `1` ansteigt.
  Falls `duration` nicht positiv ist, soll eine `IllegalArgumentException` geworfen werden.
  Falls `sampleRate` nicht positiv ist, soll eine `IllegalArgumentException` geworfen werden.
  Das zu erzeugende Signal soll endlich, mit Größe `sampleRate` * `duration` , sein, genau einen Channel besitzen und die Samplerate `sampleRate` haben.
  Falls die Größe des Signals kleiner als `2` wäre, z.B. weil die Dauer des Signals kurz und die Samplerate niedrig ist, soll stattdessen eine `IllegalArgumentException` geworfen werden.
  Zum Berechnen der Signalwerte gehen Sie folgendermaßen vor:
  - Es sei im Folgenden `samples` die Anzahl der Samplepunkte (also die Größe des Signals).
  - Der `i` -te Samplepunkt soll den Wert `i` /(`samples` - `1`) haben.

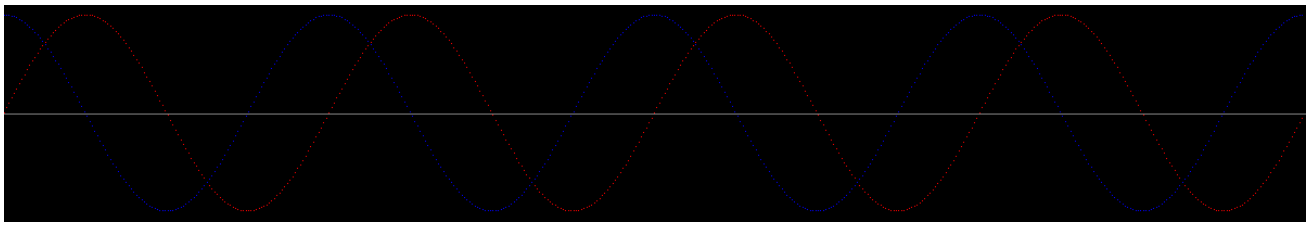
  Beispielsweise sieht für den Aufruf `rampUp(1, 500)` der Plot des Signals so aus:



- `public static Signal combineMonoSignals(List<Signal> signals)`
  Diese Methode soll mehrere Monosignale (Signale mit genau einem Channel) zu einem einzigen Signal mit mehreren Channels kombinieren.
  Falls `signals` `null` ist, soll eine `NullPointerException` geworfen werden.
  Falls `signals` leer ist, soll eine `IllegalArgumentException` geworfen werden.
  Falls `signals` Signale mit unterschiedlichen Sampleraten enthält ist, soll eine `IllegalArgumentException` geworfen werden.
  Falls `signals` ein Signal enthält, das kein Monosignal ist, soll eine `IllegalArgumentException` geworfen werden.

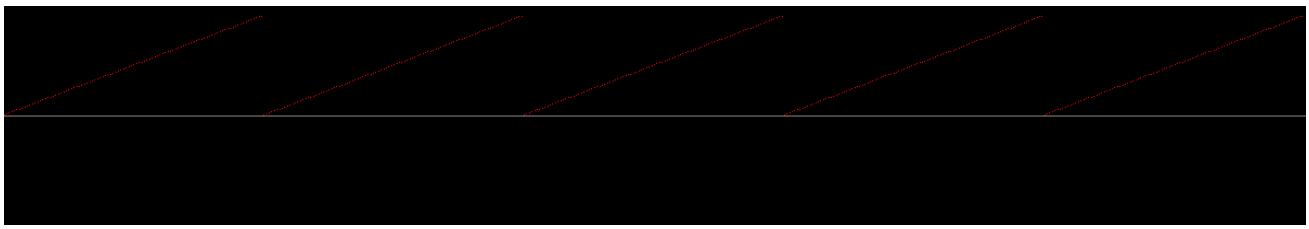
Das kombinierte Signal soll so groß wie das kürzeste Monosignal sein. Das bedeutet folglich, dass das neue Signal nur unendlich ist, wenn alle Signal in `signals` unendlich sind. Das Signal hat genauso viele Channels wie `signals` `Signal`-Objekte enthält. Die neue Samplerate ist die selbe wie die der Signale aus `signals`. Der neue Werte von Channel `channel` an Stelle `index` soll der selbe sein wie der Wert des Signals aus `signals` mit Index `channel` von Channel `0` an Stelle `index`.

- `public static Signal combineMonoSignals(Signal... signals)`
  Diese Methode soll genauso funktionieren wie `public static Signal combineMonoSignals(List<Signal> signals)`, mit dem Unterschied, dass die Monosignale als *vararg*-Argument übergeben werden.
- `public static Signal stereoFromMonos(Signal left, Signal right)`
  Diese Methode soll zwei Monosignale zu einem Stereosignal (Signal mit zwei Channels) kombinieren. Die Kombination funktioniert genauso wie in `combineMonoSignals`, aber wegen der Wichtigkeit von Stereosignalen für das Zeichnen auf dem Oszilloskop wird dafür diese Methode mit explizitem Namen implementiert.
- `public static Signal extractChannels(Signal source, int... channels)`
  Diese Methode soll ein neues Signal erzeugen, dessen Channels sich durch das in `channels` gegebene Mapping aus den Channels von `source` zusammensetzen.
  Dementsprechend übernimmt das neue Signal auch Unendlichkeit, Größe und Samplerate vom ursprünglichen Signal. Das neue Signal besitzt "Länge von `channels`"-viele Channels. Der neue Werte von Channel `channel` an Stelle `index` soll der selbe sein wie der Wert des alten Signals von Channel `channels[channel]` an Stelle `index`.
  Dabei sind einige Fehlerfälle zu beachten:
  Falls `source` `null` ist, soll eine `NullPointerException` geworfen werden.
  Falls `channels` Werte enthält, die keine gültigen Channel-Indizes in `source` sind, soll eine `IllegalArgumentException` geworfen werden. Beispiel:
  Sei `s3` ein Signal mit drei Channels, dann liefert
  - `extractChannels(s3, 0)` ein Monosignal, das als Channel 0 den Channel 0 von `s3` besitzt.
  - `extractChannels(s3, 2)` ein Monosignal, das als Channel 0 den Channel 2 von `s3` besitzt.
  - `extractChannels(s3, 2, 1, 0)` ein Signal mit drei Channels, nämlich den Channels von `s3`, aber in umgekehrter Reihenfolge.
- `public static Signal circle(double frequency, double duration, int sampleRate)`
  Diese Methode soll ein Signal zurückgeben, das auf dem Oskilloskop einen Kreis zeichnet.
  Falls `frequency` nicht positiv ist, soll eine `IllegalArgumentException` geworfen werden.
  Falls `duration` nicht positiv ist, soll eine `IllegalArgumentException` geworfen werden.
  Falls `sampleRate` nicht positiv ist, soll eine `IllegalArgumentException` geworfen werden.
  Um auf dem Oszilloskop zu zeichnen muss das Signal ein Stereosignal sein. Ein Kreis ergibt sich genau dann, wenn ein Channel eine Sinus-Welle enthält, während der andere eine Cosinus-Welle mit selber Frequenz darstellt. Prinzipiell können die Channel vertauscht werden, allerdings soll sich hier die Sinus-Welle im Channel `0` befinden.
  Es soll `frequency` für die Frequenz der beiden Wellen gewählt werden. Daraus ergibt sich, dass der Elektronenstrahl des Oszilloskops `frequency` oft pro Sekunde den Kreis abfährt.
  Die Amplitude der Wellen bestimmt den Radius des Kreises. In diesem Fall soll die "Default"-Amplitude von `1` verwendet werden.
  Weiterhin soll, vermutlich selbstverständlich, das Signal endlich sein, eine Größe von `duration` * `sampleRate` und eine Samplerate von `sampleRate` haben.
  Beispielsweise sieht für den Aufruf `circle(4, 1, 500)` der Plot des Signals so aus:

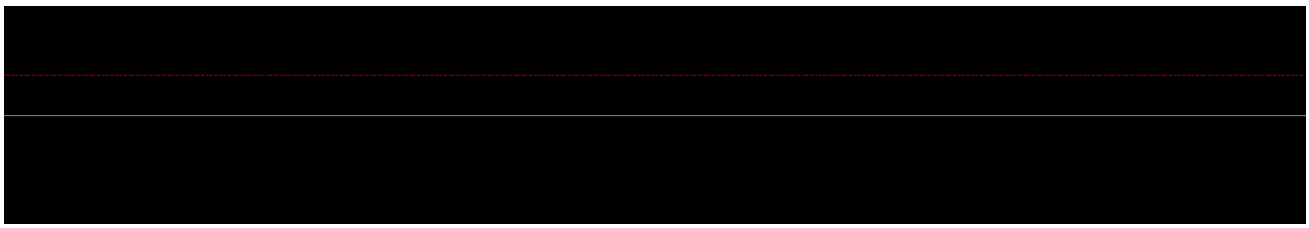


Wobei der rote Plot den Channel `0` und der blaue Plot den Channel `1` visualiert.

- `public static Signal cycle(Signal signal)`
  Nun geht es endlich darum ein unendliches Signal zu erzeugen. Diese Methode soll ein unendliches Signal zurückgeben, das das gegebene `signal` "in Dauerschleife abspielt".
  Falls `signal` `null` ist, soll eine `NullPointerException` geworfen werden.
  Das neue Signal soll unendlich sein. Dementsprechend ist das Verhalten von `getSize` nicht definiert und kann beliebig gewählt werden. Die Anzahl der Channel und die Samplerate werden von `signal` übernommen.
  `signal` darf auch ein unendliches Signal sein. In diesem Fall kann der Wert für den Channel `channel` und Samplepunkt `index` direkt von `signal` abgeleitet werden.
  Ist `signal` endlich und `index` größer oder gleich der Größe von `signal`, so muss `index` mittels Modulorechnung so modifiziert werden, dass er zwischen `0` (inklusive) und der Größe von `signal` (exklusive) liegt.
  Beispielsweise sieht für den Aufruf `cycle(rampUp(0.2, 500))` der Plot der ersten Sekunde des Signals so aus:



- `public static Signal infiniteFromValue(double value, int sampleRate)`
  Diese Methode soll ein unendliches Signal erzeugen, das einen Channel besitzt, der an jedem Samplepunkt den Wert `value` hat.
  Das neue Signal ist also unendlich, das Verhalten von `getSize` ist undefiniert, das Signal hat genau einen Channel, als Samplerate ist `sampleRate` zu wählen und an jeder gültigen Stelle besitzt das Signal den Wert `value`.
  Dementsprechend unspektakulär ist ein Plot dieses Signals (mit `value = 0.4`):



- `public static Signal take(int count, Signal source)`
  Diese Methode ist dazu gedacht um Signale zu kürzen, auf die gegebene Länge `count`.
  Falls `count` negativ ist, soll eine `IllegalArgumentException` geworfen werden.
  Das neue Signal ist endlich mit der Größe `count`. Anzahl der Channel und Samplerate werden von `source` übernommen.
  Falls die Größe von `source` größer als `count` ist, wird das `source` abgeschnitten. Für jeden Channel `channel` und `index` kann also der Wert direkt aus `source` bestimmt werden.
  Falls `source` weniger Samplepunkte als `count` besitzt, sollen alle Werte für Samplepunkt-Indizes, die größer oder gleich der Größe von `source` sind `0` sein (`source` mit Nullen auf die neue Größe auffüllen).

- `public static Signal drop(int count, Signal source)`
  Diese Methode ist dazu gedacht den Anfang eines Signals zu verwerfen, nämlich die ersten `count` Samplepunkte.
  Falls `count` negativ ist, soll eine `IllegalArgumentException` geworfen werden.
  Falls `source` unendlich groß ist, ist das resultierende Signal immer noch unendlich groß und das Verhalten von `getSize` somit undefiniert. Falls `source` endlich groß ist, wird die Größe von `source` um `count`, bzw. möglicherweise auf `0` reduziert (ein Signal von Größe `0` ist nicht sehr nützlich, aber durchaus legal).
  Anzahl der Channel und Samplerate werden von `source` übernommen. Der Wert von Channel `channel` am Samplepunkt `index` soll der selbe sein wie `source` von Channel `channel` am Samplepunkt `index` + `count`.
- `public static Signal transform(DoubleUnaryOperator function, Signal source)`
  Ein `DoubleUnaryOperator` ist ein Interface, welches eine Methode

  ```
  double applyAsDouble(double operand)
  ```

  fordert.
  Die gegebene Implementierung dieses Interfaces `function` soll dazu verwendet werden jeden Signalwert von `source` zu transformieren.
  Falls `function` `null` ist, soll eine `NullPointerException` geworfen werden.
  Falls `source` `null` ist, soll eine `NullPointerException` geworfen werden.
  (Un-)endlichkeit, Größe, Channelzahl und Samplerate sollen identisch zu denen von `source` sein. Der Wert von Channel `channel` am Samplepunkt `index` soll der Wert von `source` von Channel `channel` am Samplepunkt `index` eingesetzt in `applyAsDouble` von `function` sein.
- `public static Signal scale(double amplitude, Signal source)`
  Diese Methode soll ein Signal zurückliefern das jeden Signalwert von `source` um `amplitude` streckt oder staucht. Falls `source` `null` ist, soll eine `NullPointerException` geworfen werden.
  (Un-)endlichkeit, Größe, Channelzahl und Samplerate sollen identisch zu denen von `source` sein. Der Wert von Channel `channel` am Samplepunkt `index` soll der Wert von `source` von Channel `channel` am Samplepunkt `index`, multipliziert mit `amplitude` sein.
- `public static Signal reverse(Signal source)`
  Diese Methode soll ein ein Signal zurückgeben, das die Samplepunkte von `source` ist umgekehrter Reihenfolge enthält.
  Falls `source` `null` ist, soll eine `NullPointerException` geworfen werden.
  Falls `source` unendlich ist, soll eine `IllegalArgumentException` geworfen werden.
  Folglich soll auch das neue Signal endlich sein. Größe, Channelzahl und Samplerate sollen identisch zu denen von `source` sein. Der Wert von Channel `channel` am Samplepunkt `index` soll der Wert von `source` von Channel `channel` am Samplepunkt ("Größe von `source` "- 1 - `index`) sein.
- `public static Signal rampDown(double duration, int sampleRate)`
  Diese Methode soll sich so verhalten wie `rampUp`, mit dem Unterschied, dass die Werte im einzigen Channel linear von `1` auf `0` abfallen soll.
- `public static Signal merge(BiFunction<Double, Double, Double> function, Signal s1, Signal s2)`
  Eine `BiFunction<Double, Double, Double>` ist ein Interface, welches die Methode

  ```
  Double apply(Double t, Double u)
  ```

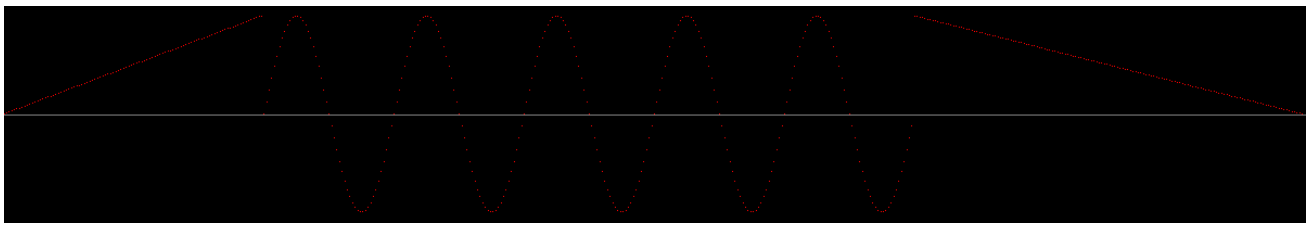
  fordert.
  Die gegebene Implementierung des Interfaces `function` soll dazu verwendet werden die Signalwerte der Signale `s1` und `s2` zu kombinieren.
  Falls `s1` `null` ist, soll eine `NullPointerException` geworfen werden.
  Falls `s2` `null` ist, soll eine `NullPointerException` geworfen werden.
  Falls `function` `null` ist, soll eine `NullPointerException` geworfen werden.

Falls `s1` und `s2` unterschiedliche Sampleraten haben, soll eine `IllegalArgumentException` geworfen werden.
Falls `s1` und `s2` unterschiedliche Channelanzahlen haben, soll eine `IllegalArgumentException` geworfen werden.
Sind die Signale von unterschiedlicher Größe, wird die kleinere Größe als neue Größe für das neue Signal gewählt. Folglich ist das resultierende Signal nur dann unendlich, falls sowohl `s1` als auch `s2` unendlich ist. Zahl der Channels und die Samplerate sollen von `s1` und `s2` übernommen werden.
Das neue Signal soll im Channel `channel` am Samplepunkt `index` das Resultat von `function` haben, wenn die Werte von `s1` und `s2` jeweils von Channel `channel` und Samplepunkt `index` eingesetzt werden (Wert von `s1` als erstes Argument.)

- `public static Signal add(Signal s1, Signal s2)`
  Diese Methode soll die Signale `s1` und `s2` mergen, indem die Signalwerte addiert werden.
  Falls `s1` `null` ist, soll eine `NullPointerException` geworfen werden.
  Falls `s2` `null` ist, soll eine `NullPointerException` geworfen werden.
  Falls `s1` und `s2` unterschiedliche Sampleraten haben, soll eine `IllegalArgumentException` geworfen werden.
  Falls `s1` und `s2` unterschiedliche Channelanzahlen haben, soll eine `IllegalArgumentException` geworfen werden.
  Sind die Signale von unterschiedlicher Größe, wird die kleinere Größe als neue Größe für das neue Signal gewählt. Folglich ist das summierte Signal nur dann unendlich, falls sowohl `s1` als auch `s2` unendlich ist. Zahl der Channels und die Samplerate sollen von `s1` und `s2` übernommen werden.
- `public static Signal mult(Signal s1, Signal s2)`
  Diese Methode soll die Signale `s1` und `s2` mergen, indem die Signalwerte multipliziert werden.
  Falls `s1` `null` ist, soll eine `NullPointerException` geworfen werden.
  Falls `s2` `null` ist, soll eine `NullPointerException` geworfen werden.
  Falls `s1` und `s2` unterschiedliche Sampleraten haben, soll eine `IllegalArgumentException` geworfen werden.
  Falls `s1` und `s2` unterschiedliche Channelanzahlen haben, soll eine `IllegalArgumentException` geworfen werden.
  Sind die Signale von unterschiedlicher Größe, wird die kleinere Größe als neue Größe für das neue Signal gewählt. Folglich ist das summierte Signal nur dann unendlich, falls sowohl `s1` als auch `s2` unendlich ist. Zahl der Channels und die Samplerate sollen von `s1` und `s2` übernommen werden.
- `public static Signal append(List<Signal> signals)`
  Diese Methode soll eine Liste von Signalen zu einem Signal zusammenführen, indem die Signale in gegebener Reihenfolge aneinandergehängt werden. Falls `signals` `null` ist, soll eine `NullPointerException` geworfen werden.
  Falls `signals` leer ist, soll eine `IllegalArgumentException` geworfen werden.
  Falls ein Signal aus `signals`, das nicht das letzte Signal ist, unendlich ist, soll eine `IllegalArgumentException` geworfen werden.
  Falls die Signale in `signals` nicht alle die gleichen Sampleraten haben, soll eine `IllegalArgumentException` geworfen werden.
  Falls die Signale in `signals` nicht alle die gleichen Channelanzahlen haben, soll eine `IllegalArgumentException` geworfen werden.
  Das neue Signal soll genau dann unendlich sein, wenn das letzte Signal in `signals` unendlich ist. Ist dies nicht der Fall, ist das Signal endlich, wobei die Größe die Summe der Größe der Signale aus `signals` ist.
  Zahl der Channels und die Samplerate sollen übernommen werden.
  Sei `s_i` mit `i = 0`, `1`, `2`,..., `n` die Größe des Signals an Stelle `i` Der Wert von Channel `channel` an Stelle `index` soll sein:
    - Falls `index` < `s_0`, dann Wert des ersten Signals von Channel `channel` an Stelle `index`

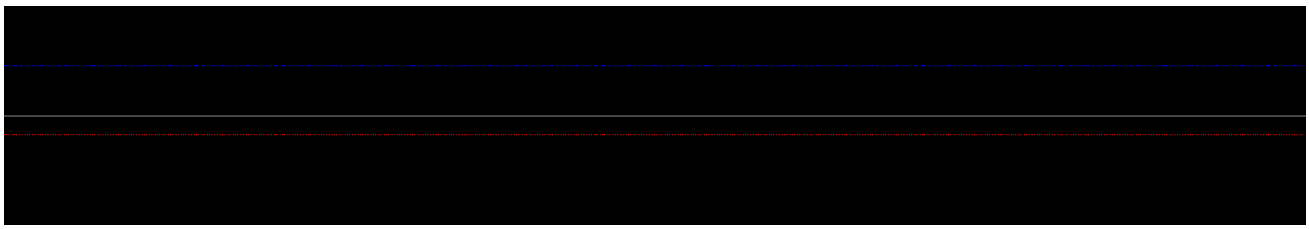
    - Sonst, falls `index` < `s_0` + `s_1` , dann Wert des zweiten Signals von Channel `channel` an Stelle `index` - `s_0`
    - Sonst, falls `index` < `s_0` + `s_1` + `s_2` , dann Wert des dritten Signals von Channel `channel` an Stelle `index` - `s_0` - `s_1`
    - ...
    - Sonst, der Wert des letzten Signals von Channel `channel` an Stelle `index` - `s_0` - `s_1` -...- `s_(n-1)`

  Beispielsweise sieht für den Aufruf `append(List.of(rampUp(0.2, 500), wave(Math::sin, 10, 0.5, 500), rampDown(0.3, 500)))` der Plot des Signals so aus:



- `public static Signal append(Signal... signals)`
  Diese Methode soll genauso funktionieren wie `public static Signal append(List<Signal> signals)` , mit dem Unterschied, dass die Signal als *vararg*-Argument übergeben werden.

- `public static Signal translate(List<Double> distances, Signal signal)`
  Diese Methode soll das Signal `signal` um ein festes Offset verschieben, welches durch `distances` geben ist. Jeder Eintrag in `distances` beschreibt die Verschiebung in einem Channel. Falls `signal` `null` ist, soll eine `NullPointerException` geworfen werden.
  Falls `distances` `null` ist, soll eine `NullPointerException` geworfen werden.
  Falls `distances` nicht genauso viele Elemente wie `signal` Channels besitzt, soll eine `IllegalArgumentException` geworfen werden.
  Das neue Signal soll Endlichkeit, Größe, Channelanzahl und Samplerate von `signal` übernehmen.
  Der Wert von Channel `channel` am Samplepunkt `index` soll gleich dem Wert von `signal` von Channel `channel` am Samplepunkt `index` sein, addiert mit Wert, der an der `channel` -ten Stelle in `distances` steht.
  Beispielsweise sieht für den Aufruf `translate(List.of(-0.2, 0.5), stereoFromMonos(infiniteFromValue(0, 500), infiniteFromValue(0, 500)))` der Plot der ersten Sekunde des Signals so aus:



  Wobei der rote Plot den Channel `0` und der blaue Plot den Channel `1` visualiert.

Die letzte zu implementiertende Methode in `SignalFactory` soll ein Stereosignal erzeugen, das den Elektronenstrahl des Oskilloskop mit konstanter Geschwindigkeit entlang eines Pfades bewegt. Dazu sollen zuerst zwei Hilfsklassen implementiert werden.

Die Klasse `Point`

Die Klasse `Point` aus dem Package `de.uniwue.jpp.oscidrawing.generation.pathutils` wird verwendet um Punkte im zweidimensionalen Raum zu verwalten. Es sind die folgenden Methoden zu implementieren:

- `public Point(double x, double y)`
  Erstellt ein neues `Point` -Objekt.

- `public double getX()`
  Gibt den im Konstruktor erhaltenen `x` -Wert zurück.
- `public double getY()`
  Gibt den im Konstruktor erhaltenen `y` -Wert zurück.
- `public double distanceTo(Point p)`
  Gibt die euklidische Distanz zum Punkt p zurück.
- `public String toString()`
  Gibt eine textuelle Darstellung des Punktes in folgender Form zurück:

  ```
  Point{x=<x>, y=<y>}
  ```

  wobei `<x>` und `<y>` durch die entsprechenden Attributwerte ersetzt werden sollen.
- `public Point interpolateTo(Point p, double factor)`
  Sei `xd` der Vektor, der von diesem Punkt zum Punkt `p` zeigt. Der zurückgegebene Punkt soll "dieser Punkt"+ `factor` * `xd` sein. Insbesondere gilt dann:
  Falls `factor` gleich `0` ist, soll der zurückgegebene Punkt die selben Koordinaten wie dieser Punkt haben.
  Falls `factor` gleich `1` ist, soll der zurückgegebene Punkt die selben Koordinaten wie der Punkt `p` haben.

Die Klasse `Line`

Die Klasse `Line` aus dem Package `de.uniwue.jpp.oscidrawing.generation.pathutils` wird verwendet um Linien zwischen zwei Punkten im zweidimensionalen Raum zu verwalten. Es sind die folgenden Methoden zu implementieren:

- `public Line(Point p1, Point p2)`
  Erstellt ein neues `Line` -Objekt.
- `public double getStart()`
  Gibt den im Konstruktor erhaltenen `p1` -Wert zurück.
- `public double getEnd()`
  Gibt den im Konstruktor erhaltenen `p2` -Wert zurück.
- `public double length()`
  Gibt die Länge der Linie zurück.
- `public Point getPointAt(double percentage)`
  Gibt den Punkt zurück, der an `percentage` des Weges vom Startpunkt zum Endpunkt liegt. Insbesondere gilt dann:
  Falls `percentage` gleich `0` ist, soll der zurückgegebene Punkt die selben Koordinaten wie der Startpunkt haben.
  Falls `percentage` gleich `1` ist, soll der zurückgegebene Punkt die selben Koordinaten wie der Endpunkt haben.
- `public String toString()`
  Gibt eine textuelle Darstellung der Linie in folgender Form zurück:

  ```
  Line{p1=<p1>, p2=<p2>}
  ```

  wobei `<p1>` und `<p2>` durch die entsprechenden Attributwerte ersetzt werden sollen.

Mit diesen Hilfsmitteln kann nun die letzte Methode der `SignalFactory` implementiert werden:

- `public static Signal fromPath(List<Point> points, double frequency, int sampleRate)`
  Diese Methode soll ein Stereosignal zurückgeben, das den Elektronenstrahl des Oszilloskops entlang der Punkte, die in `points` enthalten sind, bewegt. Dabei soll Geschwindigkeit möglichst konstant gehalten werden. `frequency` gibt an, wie oft pro Sekunde das Signal wiederholt werden könnte. Allerdings soll das zurückgegebene Signal nur ein einziges Mal den Pfad entlangfahren.

Falls `frequency` nicht positiv ist, soll eine `IllegalArgumentException` geworfen werden.
Falls `sampleRate` nicht positiv ist, soll eine `IllegalArgumentException` geworfen werden.
Falls `points` leer ist oder nur einen einzelnen Punkt enthält, soll eine `IllegalArgumentException` geworfen werden.
Gehen Sie folgendermaßen vor um ein entsprechendes Signal zu erzeugen:

- Berechnen Sie die `duration` als Kehrwert der `frequency` .
- Erzeugen Sie `lines` , die alle Linien des Pfades enthält (diese Liste ist dann um `1` kleiner als `points`).
- Erzeugen Sie `lineLengths` , die für jede `Line` aus `lines` die Länge der Linie speichert.
- Berechnen Sie `pathLength` als Summe aller Linienlängen.
- Erzeugen Sie eine Liste `normalizedLineLengths` , die für jede `Line` aus `lines` die Länge der Linie geteilt durch `pathLength` speichert.
- Erzeugen Sie eine Liste `pointsPerLine` , die für jede `Line` aus `lines` speichert, wieviele Samplepunkte diese Linie erhalten soll.
  Sei dabei `l` eine `Line` und `nLen` die normalisierte Länge von `l` . Dann kann die Dauer `lDur` , die der Elektronenstrahl auf dieser Linie verbringt, berechnet werden also `duration` * `nLen` . Weiter kann dann mit `lDur` * `sampleRate` berechnet werden, wievele Samplepunkte die Linie erhalten soll. Das Ergebnis ist allerdings ein Fließkommawert. Runden Sie diesen ab und speichern das Ergebnis als `Integer` in der Liste `pointsPerLine` ab.
- Erzeugen Sie eine Liste `interpolatedPoints` , die alle Samplepunkte, gespeichert als `Point` , enthält. Beginnen Sie mit einer leeren Liste. Für jede Linie aus `lines` , führen Sie aus:
  - Es sei `numPoints` die Anzahl der Samplepunkte der Linie.
  - Erzeugen Sie eine Liste `indices` , die die Ganzzahlen von `0` (inklusive) bis `numPoints` (exklusive) enthält.
  - Erzeugen Sie eine Liste von `Double` mit dem Namen `lineProgress` , die die Elemente aus `indices` geteilt durch `numPoints` enthält.
  - Erzeugen Sie eine Liste mit dem Namen `interpolatedPointsOfLine` , die die Punkte enthält, die `getPointAt` der Linie zurückliefert, wenn man die Elemente von `lineProgress` einsetzt.
  - Fügen Sie alle Elemente von `interpolatedPointsOfLine` am Ende von `interpolatedPoints` ein.

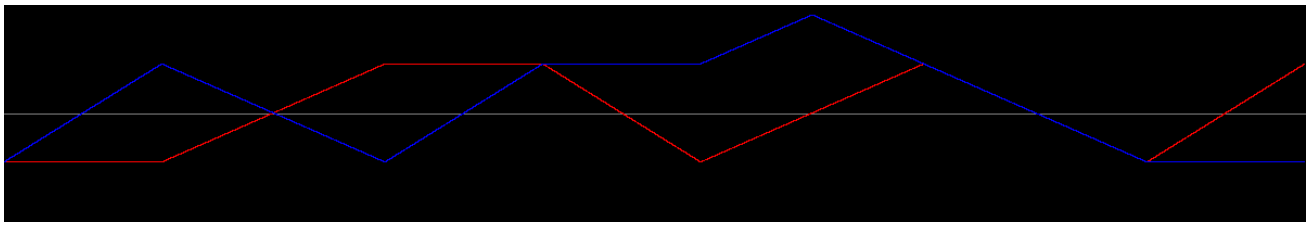
  Nun enthält `interpolatedPoints` alle Informationen die für das zu erzeugende Signal notwendig sind.

Geben Sie ein Stereosignal zurück, dessen Channel `0` die `x` -Koordinaten der Punkte enthält und dessen Channel `1` die `y` -Koordinaten der Punkte enthält.

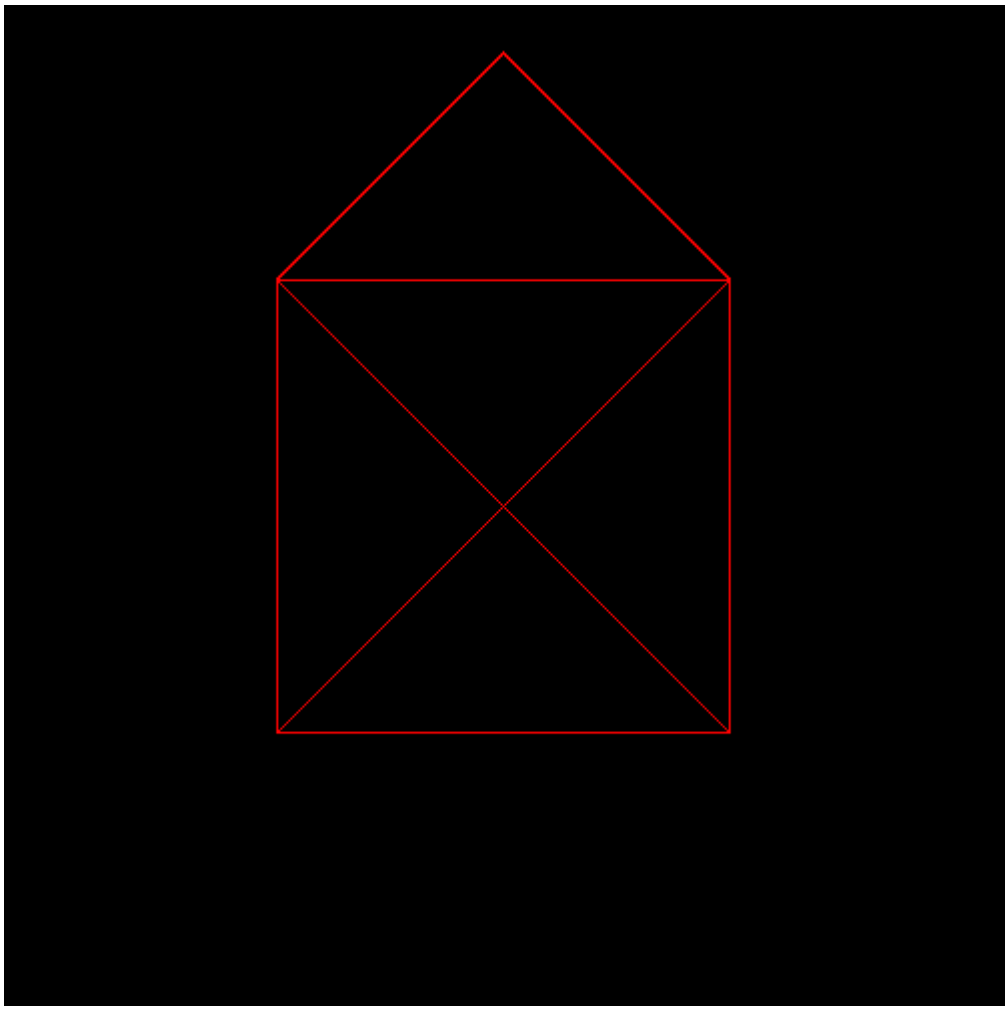
Beispielsweise ergeben die Punkte

```
(-0.5,-0.5)
(-0.5, 0.5)
( 0.5,-0.5)
( 0.5, 0.5)
(-0.5, 0.5)
(   0,   1)
( 0.5, 0.5)
(-0.5,-0.5)
( 0.5,-0.5)
```

folgendes Signal (Channel `0` -> rot, Channel `1` -> blau):



Auf dem Oszilloskop ergibt sich dann das folgende Bild:



Path-Import

Da es relativ unkomfortabel ist im Code eine Liste von Punkten zu erstellen um sich daraus Signale erzeugen zu lassen, soll ein Importer geschrieben werden, der Punkte aus einem Textformat einlesen kann.
Das Format für Punkte ist sehr simpel und sieht zum Beispiel folgendermaßen aus:

```
0.2165206508135169,-0.1639549436795995
0.041301627033792254,-0.06883604505632035
-0.17146433041301623,-0.018773466833541974
-0.3241551939924906,-0.053817271589486904
```

Jede Zeile beschreibt einen Punkt. Zuerst gibt eine Fließkommazahl die `x`-Koordinate des Punktes an, dann folgt ein Komma, bevor eine weitere Fließkommazahl die `y`-Koordinate des Punktes bestimmt.

Implementieren Sie in der Klasse `PathImporter` im Package `de.uniwue.jpp.oscidrawing.io` diese beiden Methoden:

- `public static Optional<List<Point>> fromString(List<String> lines)`
  Diese Methode erhält eine Liste von Strings, von denen jeder eine Zeile einer Textdatei darstellen soll. Falls alle Zeilen dem eben beschriebenen Format entsprechen, soll eine Liste der durch die Zeilen

beschriebenen Punkte, verpackt in ein `Optional`, zurückgegeben werden.
Falls eine der Zeilen nicht dem beschriebenen Format entspricht und deswegen nicht in einen `Point` übersetzt werden kann, soll stattdessen ein leeres `Optional` zurückgegeben werden.

- `public static Optional<List<Point>> fromFile(String path)`
  Diese Methode bekommt mit `path` eine Pfad zu einer Datei übergeben. Der Inhalt dieser Datei ist einzulesen und jede Zeile in einen Punkt zu übersetzen.
  Falls irgendetwas dabei schief geht, z.B. die Datei nicht existiert, nicht gelesen werden kann oder der Inhalt nicht dem geforderten Format entspricht, soll ein leeres `Optional` zurückgegeben werden.
  Andernfalls soll eine in einen `Optional` verpackte Liste von `Point` zurückgegeben werden, die dem Inhalt der Datei entspricht.

Im Ordner `resources` finden Sie die Pfad-Datei `samplePath.txt`, mit der Sie Ihren Importer testen können.

Optional: Eigenes Signal erzeugen

In der `SignalFactory` existiert noch die Methode `myCoolSignal`-Methode. Diese Methode muss vorhanden sein, damit die Tests kompilieren, es ist aber ausreichend die `return null`-Default-Implementierung unverändert zu lassen.

Wer Spaß daran hatte Signale zu erzeugen und die Factory-Methoden zu sinnvoller Anwendung bringen möchte, ist eingeladen hier ein kreatives Signal zu erzeugen und zurückzugeben.
Die "Abgaben" werden am Ende des Praktikums ausgewertet.
Falls Sie dabei Dateien einlesen, platzieren Sie diese im `resources`-Ordner und lesen sie wie hier beschrieben (https://mkyong.com/java/java-read-a-file-from-resources-folder/) ein (`resources` als `Resources Root` markieren).
