ChatGPT

OpenWInD Package: Comprehensive Code Review and Guide

Overview of OpenWInD Toolbox

OpenWInD (Open Wind Instrument Design) is an open-source Python library developed by Inria for the modeling and simulation of wind musical instruments. It provides tools to compute the acoustic response of an instrument from its geometrical description. In particular, OpenWInD can calculate the **input impedance** (or admittance) of a wind instrument over a range of frequencies (frequency-domain analysis) and can also simulate the **time-domain sound** of instruments (such as brass and single-reed woodwinds) by coupling the bore acoustics with a mouthpiece model. The library incorporates advanced physical effects: thermo-viscous losses in the air, various end radiation conditions, and non-linear excitation models (e.g. a clarinet reed or brass player's lips). It even supports *inversion* techniques to optimize or reconstruct instrument designs by comparing measured and simulated responses.

OpenWInD is organized into several modules, each handling a different aspect of the simulation pipeline. At a high level, using OpenWInD involves the following steps:

1. **Geometry Definition** – The instrument's bore profile and tonehole dimensions are described. OpenWInD provides classes to represent shapes (cylindrical, conical, etc.) and to parse geometry input files into a structured format.
2. **Physical Modeling** – The geometric description is converted into a network of acoustic **elements** (tubular sections, junctions for toneholes or bore discontinuities, radiation loads at open ends, and an excitation at the mouthpiece). These elements, together with their physical properties (e.g. impedance of each segment), form a **netlist** representation of the instrument's acoustic system.
3. **Discretization** – The continuous model can be discretized (using one-dimensional finite element or finite difference methods) into a numerical system suitable for computation. Alternatively, for certain computations, analytical transfer-matrix formulations are used for each element.
4. **Frequency-Domain Solver** – Given the assembled instrument model, OpenWInD can solve for the input impedance spectrum across frequencies. This involves either cascading transfer matrices of each element or solving a linear system derived from the finite element model at each frequency. The result is typically an impedance curve showing resonances (peaks) at the instrument's natural frequencies.
5. **Time-Domain Solver** – OpenWInD can also simulate the instrument's sound output over time when driven by a player's excitation. For example, a clarinet can be modeled with a reed excitator; the simulation computes the pressure and flow in the instrument bore over successive time steps, producing an output waveform (which can be used to generate sound).
6. **Analysis and Inversion** – The library includes tools to analyze the results (e.g. identifying peak frequencies, which correspond to playable notes) and even to perform optimization. Users can mark certain geometry parameters as variables, and OpenWInD's inversion algorithms (e.g. using a Levenberg–Marquardt approach) will adjust those parameters to achieve desired acoustic targets (like matching a measured impedance curve).

Under the hood, OpenWInD is structured into subpackages that mirror these stages. In the rest of this guide, we will **review the code line by line** (in a logical grouped manner) to understand each part of the package in detail. We will cover how the code handles instrument geometry definition (including clarinet bores and toneholes), how it builds the acoustic network, how the frequency and time solvers work, and how one can use the inversion tools for virtual prototyping. Throughout, we will relate the functionality to practical clarinet design where relevant.

Geometry Design and Representation (openwind.design)

The design module defines how the instrument's shape is described in the code. It provides abstract classes and concrete implementations for different bore profile shapes, as well as parameter classes to manage fixed or variable geometric parameters.

DesignShape Base Class and Derived Shapes

At the heart of the design module is the `DesignShape` abstract class (defined in `design_shape.py`). This class represents the radius profile **r(x)** of a section of the instrument's bore, over a normalized length from 0 to 1. Key aspects of `DesignShape` in the code:

- It defines an abstract method `get_radius_at(x_norm)` which returns the radius at a given normalized position along that segment. All specific shape classes must implement this. For example, if `x_norm = 0` it should return the radius at the start of the segment, and at `x_norm = 1` the radius at the end.
- It also defines `get_diff_radius_at(x_norm, diff_index)` for computing derivatives of the radius with respect to design parameters (used in optimization), as well as helper methods like `get_endpoints_position()` and `get_endpoints_radius()` to retrieve the physical start/end coordinates and radii of the segment.
- Utility methods such as `plot_shape()` (for visualization), `cut_shape(start, stop)` (to truncate a shape segment), and conversion methods between actual length and normalized coordinate (`get_position_from_xnorm` and vice versa) are provided.
- A `__repr__` and `__str__` are implemented to give a readable description of the shape (including its type and dimensions) when printed.

Concrete shape classes derive from `DesignShape` and implement the radius function. In the OpenWInD codebase, several shape types are supported (some are defined in separate files in the **design** package):

- **Linear (Conical) Segment:** A linear interpolation of radius between two ends (essentially a conical frustum). This might not appear as a separate class named "Linear" in the code, because a linear segment can be represented either by a general class or simply by using the base `DesignShape` with two end radii. However, when parsing input, the keyword `"linear"` is recognized (see below in the InstrumentGeometry section). A linear segment's radius function is `r(x) = r1 + (r2 - r1)*x_norm` (straight-line change). If `r1 == r2`, this is a **cylinder** (constant radius section).
- **Circle:** The code defines a class `Circle(DesignShape)` in `circle.py`. Despite the name, this does not mean a cylindrical tube, but rather it denotes a curvature defined by part of a circle's circumference. The documentation comment calls it "a slightly fat curve" when a negative parameter is given. Essentially, a circle segment uses an arc of a circle to smoothly connect two radii. The constructor likely takes a parameter (e.g. a curvature radius or an angle) influencing how "bulged"

the profile is. In the input file format (see InstrumentGeometry below), a `"circle"` segment requires an additional parameter (e.g. `-10` in the example) which adjusts the curvature of the radius profile. This shape allows for flaring or contracting sections in a smooth, non-linear way (useful for designing *musical bells* or barrel sections).

- **Exponential:** A class (probably `Exponential(DesignShape)`) that implements an exponential horn profile. In the input format, `"exponential"` is a recognized type that likely uses the formula `r(x) = r1 * exp(k*x)` (or something similar) to smoothly increase or decrease radius exponentially from r1 to r2 over the length.
- **Bessel:** Another specialized horn shape. Bessel horns are known in acoustics for certain bore profiles (governed by Bessel functions). The input type `"bessel"` requires a parameter alpha. The code likely has a `Bessel(DesignShape)` class that uses the given alpha parameter in the radius function (Bessel horns follow a specific differential equation). This shape is typically used for brass instrument bells.
- **Spline:** The class `Spline(DesignShape)` (defined in `spline.py`) allows a free-form shape passing through a set of intermediate points. The input format for `"spline"` expects a series of `(x_i, r_i)` internal points in addition to the end points. The Spline class likely constructs a polynomial or piecewise polynomial (spline) that fits these radius points. This gives the user fine control over the bore profile curve.
- **Cone:** In OpenWInD's context, *cone* may refer to an exact conical section. There might be a class `Cone(DesignShape)` that enforces a linear radius increase exactly matching a specified length and end radii (which is essentially similar to the linear case). It's possible that `"linear"` and `"cone"` are treated equivalently in input (or the code might not differentiate since a linear taper is a cone). However, the presence of `Cone` in the import (`from openwind.design import Spline, Cone, Bessel, Circle, Exponential`) confirms a `Cone` class exists. It likely just implements `get_radius_at` as a linear function from r1 to r2 (like linear), possibly included for semantic clarity.

Each of these shape classes provides `get_radius_at` and possibly overrides other methods if needed (for example, `Circle` might need to handle the case where one end radius equals the other differently with the curvature parameter). The shapes also have a method `is_TMM_compatible()`, which presumably returns True if the shape can be handled by the simpler Transfer-Matrix Method analytically. Likely simple shapes (linear/cylinder, maybe exponential) are TMM-compatible, whereas a spline or complex shape might not be (requiring numeric discretization instead). The code method `is_TMM_compatible` in `DesignShape` probably checks the class or type.

DesignShape Parameterization: Notice that shapes in OpenWInD can incorporate *design parameters*. The values like length and radii can be ordinary floats or special objects that represent variables for optimization. This is handled by the design parameter classes (next section). In the shape definitions, the radius endpoints (r1, r2) might be stored as `DesignParameter` objects. The `get_diff_radius_at(x_norm, diff_index)` method then computes how the radius changes if one of those parameters is varied, enabling the gradient computations for the inversion module.

Design Parameters: Fixed and Variable (design.design_parameter)

In `design_parameter.py`, OpenWInD defines classes for handling numeric parameters that might be subject to optimization (variation). The idea is to tag certain dimensions (like a tube length or a hole diameter) as adjustable. The key classes include:

- **DesignParameter:** A base class representing a numerical value associated with a shape or geometry element. It likely holds a current value and possibly bounds or a flag if it's variable. For example, a `DesignParameter` might just wrap a float and have methods to get the value or gradient index.
- **FixedParameter:** Inherits `DesignParameter` – represents a parameter that is *not* to be changed during optimization. It might simply store the fixed value. If an inversion or optimization routine runs, fixed parameters would remain constant. Typically, when parsing input, any plain number (like `0.01`) is wrapped as a FixedParameter internally.
- **VariableParameter:** Inherits `DesignParameter` – represents a parameter that *can* be varied. In the input file format, the user can prefix a number with `~` to indicate it's variable. There are several options:
- Just `~0.01` means variable with no explicit bounds (the code will instantiate a VariableParameter with that initial value and no bounds).
- With bounds syntax, e.g. `1e-3<~0.01<1e-1`, the code will set lower bound = 1e-3, upper bound = 1e-1 on that parameter. If only a lower bound is given (`1e-3<~0.01`), it's unbounded above, etc.
- These are parsed in `InstrumentGeometry` (it looks at the string and decides which DesignParameter subclass to use).
- **OptimizationParameters:** This is likely a container class that collects all `VariableParameter` instances in the instrument. `InstrumentGeometry` instantiates one of these (named `optim_params`) which holds the list of all variable parameters and perhaps provides indexing for them. It might also help in mapping the gradient indices to the actual parameters.

Additionally, there are specialized parameter classes for holes: - **VariableHolePosition** and **VariableHoleRadius:** These are mentioned in the import lines. They likely derive from `VariableParameter` but with special behavior. The `~...%` syntax in the input (as noted in the docstring excerpt) indicates a parameter relative to the main bore. For example, a hole position given as `~0.01%` might mean the hole's position is not an absolute value but a fraction relative to something (possibly relative to the nearest bore segment or as a percentage offset). The code notes that if bounds are given for these, *non-linear constraints* are instantiated – perhaps to ensure the hole stays on the physical bore section. Essentially, `VariableHolePosition` and `VariableHoleRadius` handle the case where changing a hole's position or radius might depend on the base bore geometry. They likely override how the parameter value is interpreted (for example, a relative hole position might scale with instrument length).

Overall, the design module's parameter system allows the rest of the code to treat geometry values uniformly. Whether a length or radius is fixed or variable, it can call `eval_(parameter)` (the code likely defines `eval_ = float()` or similar for extracting the numeric value) and it can call `diff_(parameter)` to get the derivative identifier if needed. Indeed, in `design_shape.py` we see references to `eval_` and `diff_` imported from `openwind.design` – these might be utility functions that evaluate a parameter or get its index for differentiation.

Summary: The design module provides a flexible way to describe any wind instrument bore as a series of shape segments with well-defined radius profiles. For a clarinet, for instance, one might describe the **main**

bore as a cylinder (constant radius ~7.5 mm) for the body, maybe a slightly conical *barrel* at the input, and a flaring bell which could be modeled as an exponential or spline section. Each segment would be represented by a `DesignShape` (Circle or Exponential, etc.), and their lengths and radii could be marked variable if we plan to optimize tuning. This design description is the first step, which then feeds into constructing the actual instrument model.

Instrument Geometry and Fingering (openwind.technical)

Once the basic shapes are defined, the next step is assembling the overall instrument geometry: combining the main bore sections and any side holes or valves, and specifying the fingering (which holes are open or closed for various notes). The **technical** module contains classes to handle this: the `InstrumentGeometry` class for parsing and holding the geometry, as well as classes for holes, valves, and fingering charts. It essentially bridges the gap between user-provided geometric data and the physical model.

InstrumentGeometry Class: Parsing and Structuring the Geometry

The `InstrumentGeometry` class (defined in `technical/instrument_geometry.py`) is a high-level container that takes input data (either from lists or from files) and produces structured objects representing the instrument:

Inputs: The constructor `InstrumentGeometry(main_bore, holes_valves=None, fingering_chart=None, unit='m', diameter=False, allow_long_instrument=False)` can accept either file paths or already-parsed lists:

- `main_bore`: this can be a filename (pointing to a text file describing the bore profile) or a Python list of segments (each segment itself being a list of parameters). For convenience, if the user passes a list of lists (like `[[0.0, 0.5, 0.007, 0.015, 'cone'], ...]` meaning a cone from x=0 to x=0.5 m expanding from 7mm to 15mm radius), the class will interpret it directly. If a filename is given, it will read the file (which should follow the expected format).
- `holes_valves`: similarly, an optional file or list describing each tone hole and brass valve (if any). Clarinet would have many holes; a trumpet would have valves (pistons) in this data.
- `fingering_chart`: optional file or list for fingerings. If not provided, by default the instrument is considered with **no fingering info**, which the code treats as "everything open" (all holes open by default) – meaning the computed impedance would correspond to the instrument with all toneholes open (this actually corresponds to the **highest** pitch pipe configuration, typically not how instruments are normally played unless it's like a flute with all open holes). In practice, one would usually provide a fingering chart so specific resonances (notes) can be studied.

File Format: The documentation in the code gives a clear description of the expected file formats for these inputs:

- **Main Bore Geometry File:** Each line defines one segment of the bore. The format can be either:
- `x1  x2  r1  r2  type [param...]` – This fully specifies a segment from position x1 to x2 (meters along the bore axis from the mouth end), with radius r1 at x1 and r2 at x2. The `type` is one of `linear`, `spline`, `circle`, `exponential`, `bessel` as described earlier, and additional

parameters may follow if required by that shape (e.g. the curvature for circle, alpha for bessel, internal points for spline).

- A shorter form `x   r` – This acts as a continuation: it means from the last segment's end to this new `x` coordinate, the radius changes linearly from the previous segment's last radius to this new `r`. Essentially, it's a shortcut to append a linear segment without repeating the previous end as the new start. The example given shows at the bell: after listing segments up to x=2.6, a line `2.7    2.1e-2` means "at x=2.7, radius is 0.021 m" and since the last specified point was x=2.6 with radius 0.02, it linearly goes to 0.021 at 2.7 (a short linear segment). This is a convenience in the file format.
- Comments can appear (lines starting with `#` are ignored) and special header lines starting with `!` can set `unit = mm` or `diameter = True` to override the defaults. If `diameter=True`, it means the values given for radii are diameters and should be halved to get radii internally.

For example, a clarinet main bore file might start like:

```
# Clarinet bore profile
0.0   0.065   0.0075   0.0075   linear        # cylindrical mouthpiece/barrel
0.065 0.5     0.0075   0.0073   linear        # slight taper in upper joint
0.5   0.6     0.0073   0.0073   circle   -10  # a tuning bulge or curve
... (more segments) ...
0.65  0.7     0.0073   0.015    exponential   # bell flare
```

Each line would be parsed and corresponding `DesignShape` objects (with appropriate `FixedParameter` or `VariableParameter` for each numeric value) would be created.

- **Holes and Valves File:** This lists each tonehole or valve. The first line is a header naming the columns. Recognized columns include:
- `label`: optional name of the hole (e.g. "A4" or "speaker_key"). If not given, the code will auto-label holes (e.g. Hole1, Hole2, ...).
- `variety`: either "hole" or "valve". If not specified, it defaults to hole.
- `position`: the location along the main bore axis where the center of the hole or valve branch connects (in meters from the mouth, presumably).
- `length`: the length of the side tube (chimney for a tonehole, or the bypass tube for a valve). For a tonehole, this is the chimney that goes from the main bore to the instrument's exterior. For a brass valve, it's the extra tubing engaged by the valve.
- `radius`: the radius of that side tube (chimney or valve tubing). Currently, the code notes only cylindrical side tubes are supported, so radius is constant along it.
- `type`: shape type of the side tube – currently only `'linear'` supported, which essentially means a straight cylindrical or conical chimney. Since only cylindrical is supported, this likely isn't used much (the length and equal radii imply a cylinder).
- `reconnection` (for valves only): the position on the main bore where the valve's extra tubing reconnects. A brass valve typically diverts the air through an alternate loop that rejoins the main tubing later. For toneholes, this is not applicable (denoted with `/` or blank).

An example snippet from the doc:

```
label       variety position  radius   length   reconnection
g_hole      hole    0.10      1e-3     0.021    /
b_flat_hole hole    0.23      4.2e-3   2e-3     /
piston1     valve   0.31      2.5e-3   0.10     0.32
```

Here we have two toneholes: - `g_hole` at 0.10 m from the top, chimney radius 1 mm, chimney length 0.021 m. - `b_flat_hole` at 0.23 m, radius 4.2 mm, chimney length 2e-3 (2 mm – this looks like maybe a small register key or vent). And one valve (like a trumpet piston) at 0.31 m, radius 2.5 mm, length 0.10 m, reconnecting at 0.32 m (so the loop goes out at 0.31 and back in at 0.32).

The code supports an “alternate (old) format” for holes as well: just `x   l   r   type` per hole (position, length, radius, type). But that’s deprecated and labels might not be assigned consistently in that case.

- **Fingering Chart File:** This describes which holes are open or closed for each **note** (fingering configuration). The first line has column headers: the first column is “label” (for note names) followed by one column per defined fingering (note). The following lines have each hole’s name, and for each note either `o` or `x` indicating open or closed. For example:

  ```
  label        C4     D4     E4     F4     G4
  --------------------------------------------
  g_hole       x      o      o      o      o
  b_flat_hole  x      x      o      o      o
  ...
  ```

  If `g_hole` and `b_flat_hole` correspond to two toneholes, the above chart means:
- For note C4, both g_hole and b_flat_hole are closed (`x` meaning covered), which is typically the fingering for the lowest note (all holes closed).
- For D4, g_hole is open (`o`), b_flat_hole remains closed.
- For E4, both are open, etc.

This chart must align with the holes listed. The `FingeringChart` class will read this and store an internal representation (likely a list of fingerings, each fingering being a dict or list of booleans for each hole). Note names (like “C4, D4…”) can be arbitrary labels; they might not be musically interpreted by the code except for display. The code likely uses the index or label to allow selecting a fingering.

InstrumentGeometry internals: When `InstrumentGeometry` is instantiated, it performs the following steps (as deduced from the code):

1. **Parse Main Bore:** It reads the main bore input (file or list). If it’s a file, it splits lines, skips comments, handles `! unit` and `! diameter` instructions. It then goes through each data line:
2. If a line contains 5 or more columns (x1, x2, r1, r2, type, …), it creates the appropriate `DesignShape`. For radii and positions, it wraps the values into `DesignParameter` objects. If the text had a '~' or bounds, it will create a `VariableParameter`. It also likely accumulates all these in `self.optim_params` (an `OptimizationParameters` instance) via something like `optim_params.add(parameter)` if variable.

3. If a line contains 2 columns (x, r), it treats it as a continuation of the previous segment: i.e., set x1 = last x2, r1 = last r2, x2 = new x, r2 = new r, type = 'linear'. Then create a linear segment.
4. It builds a list `main_bore_shapes` of the resulting `DesignShape` objects, in order.

Also, a safety check: if `allow_long_instrument` is False (default), the code will check the total length and maybe issue an error or warning if the numbers seem too large (e.g., user accidentally gave mm values without specifying unit, resulting in a 1000x too large instrument). This is to catch common unit mistakes.

1. **Parse Holes/Valves:** If a holes/valves list or file is provided, it reads it. It differentiates entries by `variety` column to separate tone holes and valves:
2. For each hole line, it creates a `Hole` object (there is a class `Hole` defined in this module). The `Hole` class likely stores the chimney's shape (which for now is always a simple cylinder of given length and radius) and the position on the main bore. The code will create a `DesignShape` for the chimney (likely a `Circle` or just use `DesignShape` since linear and radius same at both ends => cylinder). It will store that shape in `Hole.shape`. It also stores the `position` along the main bore (possibly as a `DesignParameter` if variable) and maybe an ID or label.
3. For each valve line, it creates a `BrassValve` object. The `BrassValve` class (also defined in `instrument_geometry.py`) probably contains two main things: a *valve hole* (the connection where the valve divert starts) and a *reconnection point*. It likely includes two `Hole`-like parts: one for where air exits the main bore into the valve tubing, and one where it comes back. Additionally, a valve might be considered normally closed (off) or open; but since design is static, the geometry is fixed, and the fingering will determine if the valve is engaged (open path through the extra tube) or not (closed path).
4. The code separates all holes into `self.holes` list and valves into `self.valves` list for clarity.

The `Hole` class itself is straightforward: it has attributes for `shape` (the chimney shape), `position` (on main bore), and perhaps `label`. It may also contain a reference to the corresponding `DesignParameter` objects for position and radius if variable (or maybe it uses `VariableHolePosition` inside).

The `BrassValve` class likely contains: positions of the two connections, a shape for the extra tubing (similar to a hole's shape), and possibly some boolean or identifier. It might also inherit from `Hole` or a common base for simplicity, but likely it's distinct due to having two connections. The code would treat a valve's extra tube as just another branch in the network, but with two connection points.

1. **Parse Fingering Chart:** If provided, the `fingering_chart` input is processed by the `FingeringChart` class (found in `technical/fingering_chart.py`). The InstrumentGeometry will create a `FingeringChart` object (if none given, it might default to an "all open" chart internally). The `FingeringChart` class likely holds:
2. A list of **hole names** (in the order they appear in the holes list).
3. A list of **note names** (the labels of each fingering configuration).
4. A 2D structure (list of lists or similar) of booleans indicating open/closed for each hole in each fingering.
5. Possibly methods to query a particular fingering by note name or index, and to apply it.

For example, `FingeringChart.is_open(hole_name, note_name)` might return True/False if that hole is open for that fingering.

In the code, `InstrumentGeometry` sets up the `FingeringChart` by passing the list of hole labels and the parsed table of 'o'/'x'. It stores the result in `self.fingering_chart`.

1. **Optimization Parameter Aggregation:** During parsing, every time a `VariableParameter` is created (for a length, radius, hole position, etc.), it should be added to the instrument's `optim_params` (an `OptimizationParameters` instance). This object then contains references to all variable knobs in the instrument. It may also group them by type or by segment. This is later used by inversion routines to easily get the vector of current parameter values and to apply updates.

After construction, an `InstrumentGeometry` instance has the following accessible attributes: - `main_bore_shapes`: list of `DesignShape` objects describing the bore, in order from mouth to end. - `holes`: list of `Hole` objects (could be empty if no toneholes). - `valves`: list of `BrassValve` objects (could be empty for woodwinds like clarinet which have none). - `fingering_chart`: a `FingeringChart` object (or possibly `None` if not provided, but likely an empty chart meaning all holes open). - `optim_params`: an `OptimizationParameters` instance collecting all variable parameters (if any).

It also likely provides methods such as `write_files(filename_prefix)` to output the geometry into OpenWInD CSV format files (the code snippet in `parsing_tools.py` shows `instru_ow.write_files(file_ow)`). This would allow converting from one format to OpenWInD's format or saving modifications.

Important: The `InstrumentGeometry` by itself is just a data structure – it does *not* compute any acoustics. It's the input to the next stage (InstrumentPhysics). But it encapsulates all info needed: the bore profile and where the holes/valves are, plus how the instrument is fingered for different notes.

For a **clarinet prototyping scenario**, one would create an `InstrumentGeometry` by specifying: - The bore (mostly cylindrical, with perhaps a tapered barrel and flared bell). - Tone holes (positions along the bore, their diameters and lengths – chimney lengths might be the wall thickness of the clarinet or tonehole lengths). - A fingering chart mapping each hole to notes. For example, a clarinet's fingering chart is more complex (involving keys that cover multiple holes and register keys, but one can start with simple open/closed per hole for basic fingerings). Once this is set up, the `InstrumentGeometry` can be fed into the physics builder to simulate acoustic behavior.

FingeringChart and Player

The **FingeringChart** class (in `fingering_chart.py`) manages the matrix of hole states. It likely provides methods to get the state of all holes for a given note, or vice versa. For instance, it might have: - `chart.get_fingering(note_label)` returning a dictionary or list of open/closed values for each hole. - `chart.get_open_holes(note_label)` or similar utility.

One particularly useful method could be `tabulate()` (the code import shows `from openwind.technical.fingering_chart import tabulate`), possibly to print or format the chart data. The `FingeringChart` class might also be iterable or indexable by note.

In the **continuous model building stage**, the fingering chart plays a role: when assembling the acoustic network, the code needs to know which holes are **open or closed** for the configuration being simulated. For example, if we want the input impedance for fingering “C4” (all holes closed in our example), the model should treat each tonehole as closed (which usually means it behaves like an inactive side branch, possibly just a trapped air cavity or effectively no branching). If a hole is **open**, it means the side branch connects to the outside air (introducing a radiative loss at that branch).

OpenWInD can handle different fingerings in computations. The `ImpedanceComputation` class (which we’ll discuss later) has a parameter `note` or uses the `Player` to specify which note’s fingering to use for the impedance calculation. The code will then configure the instrument physics accordingly (likely by closing or opening certain tonehole connections in the network).

The **Player** class (from `technical.player.py`) represents how the instrument is excited or played. It might seem odd to have a “Player” in a technical module, but think of it this way: - In frequency-domain analysis, the simplest “player” is one who produces a steady sound at each frequency. By default, OpenWInD uses `Player()` with no arguments, which according to the docs corresponds to a **unitary flow input** at the mouthpiece for each frequency【64†source】. In other words, it drives the instrument with a fixed volume velocity of 1 (or a pressure source) to measure impedance. The Player in this context holds the *boundary condition* at the input. Possibly `Player` has options to instead specify a different driving condition (like impose a pressure instead of flow, which would effectively compute admittance instead). - In time-domain simulation, the Player could encapsulate how the instrument is played over time – for example, how the blowing pressure varies, or if the fingering changes mid-simulation. It might include a *score* or sequence of events (the import in technical `from .score import Score` suggests there might be a Score class to schedule changes like notes or blowing). For a clarinet, one might simulate an increasing mouth pressure to initiate a note, or slurring between notes by opening/closing holes at certain times – the Player/Score could handle that.

From the code, `Player` is imported in `openwind/__init__.py` from technical, meaning it’s a user-facing class. Likely `Player()` can be constructed with parameters like mouth pressure (for time domain) or just left default.

Internally, `Player` may also keep track of *which fingering is currently active* (for multi-note simulations). If not provided, maybe it defaults to all holes open (since “no fingering” case).

In summary, **technical.InstrumentGeometry** and its related classes handle everything about the instrument’s **physical makeup (shape + holes)** and **how it’s played (fingerings)**. After creating an InstrumentGeometry, the next step is to feed it into the **InstrumentPhysics** to build the computational model.

Continuous Physical Model Assembly (openwind.continuous)

The continuous module is where the **physics** of sound propagation in the instrument are set up. It takes the geometric description and creates a network of elements that represent the acoustic system. The primary class here is `InstrumentPhysics` (openwind.continuous.instrument_physics), which builds the **netlist** of components and assigns physical properties, such as cross-sectional areas, wave impedances, and loss factors.

Before diving into `InstrumentPhysics`, let's clarify the types of components used to model a wind instrument acoustically:

- **Pipes:** Segments of the bore (main tube or tonehole chimneys) through which sound waves propagate. These are essentially one-dimensional waveguides characterized by length, cross-sectional area (possibly varying with x), and wave propagation properties (speed of sound, etc.). Each pipe will have two ends, which can connect to other components.
- **Junctions:** Connection points where multiple waveguides meet. Examples:
- A simple junction between two sections of different diameter (like a step discontinuity in the bore) – acoustically, this causes reflection due to impedance mismatch and possibly can be modeled as a two-port scattering element.
- A three-way junction (T-joint) where a tonehole branches off from the main bore. This connects the main bore upstream segment, the main bore downstream segment, and the side hole pipe. Such a junction has three ports.
- For a valve (which might be more complex), possibly a 4-port junction: main bore in, main bore out, valve branch out, valve branch in (though valves might also be handled by two separate T-junctions with a short main bore segment between representing the valve being "open" or "closed" state).
- **Radiation loads:** Terminations of waveguides open to the air. The open ends (bell end of clarinet, open tonehole chimneys, etc.) present an acoustic impedance that is frequency-dependent (radiation impedance). These are modeled as one-port elements connected to a pipe end, representing the sound radiating out. Different models (unflanged, flanged, etc.) are available.
- **Excitation (Source):** The input from the player – for frequency domain, it's an ideal source driving the bore; for time domain, it's a non-linear element like a reed or lips or a simple pressure/flow source.

All these elements are handled in code as objects, and their connectivity is managed by a `Netlist` graph.

Netlist and Connectors

OpenWInD uses a **graph-based approach** (like an electrical circuit analogy) to represent the instrument. The class `Netlist` (in `continuous/netlist.py`) manages this structure. Key points about `Netlist` from the code:

- It inherits from `UserDict` via a custom `MyDict` class, but conceptually, it stores a collection of **pipes** and **connectors**.
- The `Netlist` class is essentially a container that holds:
- `pipes`: a dictionary of all pipe elements in the instrument. Each entry likely maps a pipe object to its two end-nodes (or uses some labeling scheme). Actually, the code shows `pipes: dict(tuple(Pipe, PipeEnd))` – possibly meaning it stores a tuple of (Pipe, PipeEnd) for each entry (which is a bit unclear, maybe it pairs each pipe with its ends).
- `connectors`: a dict of all connectors (junctions, radiation loads, excitators) in the network. Each connector will have references to one or more `PipeEnd` that it is connected to.
- `ends`: a list of all `PipeEnd` instances in the netlist. A `PipeEnd` is a small class (defined in netlist.py) representing one end of a pipe and its connection. It holds reference to the pipe and whether it's the "plus" or "minus" end.

The code defines an enum `EndPos` with values `MINUS` and `PLUS` to label the two ends of a pipe. This convention is important because certain elements (like diodes in circuits, or here perhaps direction-sensitive conventions for how we number ends) require consistent orientation. For acoustic pipes, one can label one end as "upstream (-)" and the other "downstream (+)" arbitrarily, but consistently, so that when connecting, for example, a junction, you know which side is which.

- **NetlistConnector:** In `netlist.py`, `NetlistConnector` is a base class for anything that can connect to pipe ends. The code comment says connectors may attach to one or several pipe ends. The three main subclasses (as hinted by the Netlist docstring) are:
- `PhysicalRadiation` (connects to one pipe end, representing an open end termination).
- `PhysicalJunction` (connects to 2 or 3 pipe ends – e.g., two for a simple discontinuity or three for a T-junction).
- `Excitator` (connects to one pipe end – the input mouthpiece connection).

`NetlistConnector` likely stores a list of pipe ends it is connected to and possibly has a label or type identifier.

- The Netlist provides methods:
- `add_pipe(pipe)` – adds a new Pipe object to the network, returns the two new `PipeEnd` objects that represent its ends. Internally, it will create two PipeEnd (one for each side, labeling one as MINUS end, the other PLUS maybe) and append them to `self.ends` list and add to `self.pipes` dict.
- The connectors (like junctions, radiations) are probably added via methods inside `InstrumentPhysics` rather than direct `Netlist` methods. But Netlist might have helper methods to attach connectors.

- `reset()` – clears the netlist (useful if rebuilding).

- The Netlist docstring provides a conceptual ASCII diagram of a typical structure:

```
    Excitator  <----Pipe---->  Junction  <----Pipe---->  Junction  <----
Pipe---->  Radiation
             PipeEnd      PipeEnd      PipeEnd      PipeEnd
PipeEnd      PipeEnd
```

  This shows, for example, a main bore divided by junctions (perhaps tonehole T-junctions or just segment joints) and terminated by radiation at the rightmost end, with an excitation at the leftmost end.

It also emphasizes that **order matters** for connections: connectors expect their PipeEnds in a particular order (like you cannot swap ends arbitrarily for a multi-port connector, just as you can't swap pins on an electrical component without consequences). So the code likely ensures when adding a connector, it passes the pipe ends in a consistent convention (e.g., for a T-junction connecting main bore and hole: always [main_bore_minus_end, main_bore_plus_end, side_hole_end] in that order, or similar).

Pipes and Physical Elements

Pipe class (continuous.pipe.Pipe): Each `Pipe` object represents a section of the instrument's tube (either a main bore segment or a tonehole chimney or a valve tube). The class is defined in `continuous/pipe.py`. Its `__init__` likely takes: - `design_shape`: the `DesignShape` describing the radius along that segment (so it knows the profile). - `temperature`: the local temperature or a function for temperature distribution along the tube (in Celsius). By default, InstrumentPhysics passes a constant (like 20°C or 25°C) unless a temperature gradient is specified. The code allows `temperature` to be a callable function of x (for a gradient along main bore). - `label`: a string label for the pipe (helpful for debugging or output, e.g., "MainBore1", "Hole3", etc.). - `scaling`: an object of class `Scaling` (continuous.scaling.Scaling). This is used for nondimensionalization of units. OpenWInD can work in nondimensional form internally for numerical stability (scaling lengths by a reference, etc.). The `scaling` object likely holds conversion factors. For simplicity, we can consider that `scaling` might default to physical units (no scaling) unless specified. - Possibly other flags like `spherical_waves`: The documentation of ImpedanceComputation mentions an option `spherical_waves`. If true, the pipe should account for spherical wave propagation (as in horns). In a conical bore, waves are not plane but spherical – one effect is a end correction and cutoff frequency. The Pipe class may have a boolean for spherical waves, or determine it from the shape (perhaps if the shape is significantly flaring or if explicitly turned on).

What does a `Pipe` do? Likely: - Compute cross-sectional **area** as a function of x (from the shape). - Compute an effective **wave number** along the pipe, possibly piecewise if losses or temperature vary. - Provide methods `get_physics()` or similar: for the frequency solver, it might produce element matrices (for finite element) or its own transfer matrix. The code shows a method `get_losses()` and `is_spherical_waves()` etc. It probably precomputes some properties: - If losses are included, it might pick which formulation to use (there might be a base class or composition with a `ThermoviscousModel` class for losses). - If spherical waves option is true, maybe adjust the propagation model inside (e.g. Webster's horn equation vs plane wave). - Possibly the `Pipe` also discretizes itself if needed (like dividing into smaller elements if a higher-order method is needed). However, more likely the **Mesh** class handles discretization (see later).

Important: Each `Pipe` will ultimately be part of the global system. In a network context, a pipe element connecting two junctions or connectors can be represented by scattering or impedance parameters. For TMM (transfer matrix method), a pipe's behavior between its two ends can be summarized by a 2x2 transfer matrix at a given frequency (if uniform or if an analytic solution exists for that shape). For the finite element approach, each pipe might be broken into multiple elements (like smaller sections) and assembled in a global matrix.

Junction classes (continuous.junction): The file `continuous/junction.py` likely defines classes for different types of junctions: - `PhysicalJunction` base class (maybe abstract). - `JunctionDiscontinuity`: for a 2-port junction connecting two pipes of different radii (i.e., a simple diameter change in the main bore). If discontinuity masses (an extra inertance due to sudden expansion/ contraction) are considered (`discontinuity_mass=True` option), this junction will include an additional lumped acoustic mass or compliance to model the effect. - `JunctionTjoint`: a 3-port representing a tonehole junction (a T). It connects an upstream main bore segment, a downstream main bore segment, and a side hole pipe. The physics of a T-junction involve conservation of flow and continuity of pressure, plus possibly a "end correction volume" if `matching_volume=True` (the code's option to include matching volume presumably means adding the small volume at the junction intersection as part of the branch inertance). - `JunctionRadiation` might not be needed because radiation is separate, but they might have one for formalism (though PhysicalRadiation is likely separate). - Possibly others like `JunctionToneholeClosed` – however, a closed tonehole might just be handled by putting a very high impedance or a cap at the end of the chimney (which is essentially a radiation with no radiation, like a rigid termination).

From ImpedanceComputation parameters: - `matching_volume=False` by default means they do *not* include the common volume at a T-junction in the calculations (some models add a small compliance due to the cavity formed at the junction). - The code likely toggles some behavior in `JunctionTjoint` when `matching_volume=True`.

Radiation classes: In the continuous package, we saw multiple radiation-related classes: - `PhysicalRadiation`: a general class for radiation impedance at an open end. It likely has subclasses or specific implementations for different models. - They import `RadiationPade`, `RadiationNoncausal (Silva's model perhaps)`, `RadiationPulsatingSphere`, etc. The **radiation_model.py** probably has a function `radiation_model(category)` that given a name like 'unflanged' returns an appropriate PhysicalRadiation instance (maybe one that uses a Padé approximant to the known radiation impedance of an unflanged pipe end). A flanged (infinite baffle) end has a different impedance. They also have a “non-causal” model (Silva’s model) which might capture time-domain behavior by impulse response (non-causal meaning the impulse response has a non-instant onset due to pre-calculation, probably to ensure a proper phase). - For frequency domain impedance, radiation impedance $Z_R(f)$ can be expressed analytically or via approximations for each frequency. For time domain, convolution with an impulse response is needed (hence diffusive or non-causal models). - Each PhysicalRadiation connects to one PipeEnd (the end of a pipe that is open). The InstrumentPhysics will add a PhysicalRadiation at: - The bell end of the main bore (unless the instrument is closed at end, e.g., a clarinet’s mouthpiece end is effectively closed by the reed except for the small aperture – but in modeling, the mouthpiece end is handled by the excitator, not radiation). - Any open tonehole chimney end (top of each open tonehole). - Possibly at valve outlets if needed (though for a valve, if engaged, it leads back in, if not engaged, one end might be closed which could be treated as no radiation). - The `radiation_category` parameter to ImpedanceComputation can be a string (like 'unflanged', 'flanged'), or a dictionary if different holes have different conditions, or even a custom PhysicalRadiation object. InstrumentPhysics uses this to instantiate the correct radiation model for each open termination.

Excitator (continuous.excitator): This is the source at the mouthpiece end: - For frequency domain simulation of input impedance, the excitator is typically a simple linear source. OpenWInD might implement that as an `Excitator` object that imposes a flow (volume velocity) of 1 at the input. Possibly the `Player` object is used here: when InstrumentPhysics is created, it takes a `player` parameter (which defaults to `Player()` if not given). This `Player` might have an attribute or method to create an Excitator. - For instance, `Player` could have something like `.create_excitator(note_frequency)` or simply hold a type ('flow' or 'pressure' source). The code does `from .excitator import create_excitator, Excitator, Flow, Flute, Reed1dof, Reed1dof_Scaled` in continuous **init**, suggesting a factory function `create_excitator(player)` that returns the appropriate Excitator object. - By default, likely `create_excitator` returns an instance of `Flow` (which might be a subclass of Excitator representing a prescribed flow source). The `Flow` class could be a trivial linear source imposing Q = 1 (and thus measuring p, the impedance as p/1). - For time domain, the excitator is a full dynamic model: e.g., `Reed1dof` for clarinet. The `Reed1dof` class (in `continuous/excitator.py`) represents a single degree-of-freedom reed (like a mass-spring-damper oscillator) coupled to the mouthpiece. It will have parameters like reed mass, stiffness, damping, mouth pressure, reed opening area, etc., and it computes the volume flow into the instrument based on the pressure difference (mouth pressure minus mouthpiece pressure) and the reed dynamics. This is nonlinear (it will clip the flow when reed closes). - The `Reed1dof_Scaled` variant is probably the same equations but formulated in nondimensional variables (for numerical stability or to allow larger time steps). - There's also `Flute` (for a flute jet – which might model the delay and nonlinearity of an air jet) and possibly `Lips` (though the code didn't explicitly show Lips, possibly not implemented or named differently). - All excitators derive from `Excitator` base class, which is a subclass of `NetlistConnector` (connecting to one pipe end). The excitator in the netlist attaches at the very beginning of the first main bore segment (position x=0). Typically, the "minus" end of the first pipe (at x=0) is connected to an Excitator, and the "plus" end of that pipe goes into the bore.

Thermoviscous losses (continuous.thermoviscous_models): Sound propagation in narrow tubes like clarinet bore or toneholes is affected by thermal and viscous losses at the walls. OpenWInD includes multiple models for these: - The parameter `losses` can be `False` (no losses, treat as lossless propagation), `True` (likely use a default model, probably the "bessel" model by default as they suggest), or a string specifying the model: - `'bessel'`: Possibly uses the exact solution for waveguides with thermal and viscous boundary layers (which involves Bessel functions for the radial dependency of pressure/flow in a cylindrical tube). This could be a frequency-domain approach that adjusts the propagation constant and characteristic impedance for each frequency based on fluid properties. - `'wl'`: Perhaps "Wirsing-Lagarrigue" or "waterfall" – not sure, maybe a specific approximation method. - `'keefe'`: A model by Douglas Keefe (who published models for woodwind tonehole impedance and losses). - `'diffrepr'`: Diffusive representation – an approach where the frequency-dependent loss is approximated by additional state variables in time domain (for use in time stepping, making the system a higher-order but linear system). `'diffrepr+'` mentioned indicates a variant with extra variables explicitly. - `'minikeefe'`: Possibly a simplified Keefe model.

In the code, `thermoviscous_models.py` likely defines a function `losses_model(selection)` that returns a configured `ThermoviscousModel` object for a given selection. The `ThermoviscousModel` class would hold parameters like viscosity, thermal conductivity, and compute how they affect wave propagation. InstrumentPhysics, when it creates each Pipe, would attach a ThermoviscousModel to it if losses are requested. Then `Pipe.get_losses()` might retrieve this to adjust calculations. For frequency domain, this could mean modifying the complex wave number; for time domain, introducing damping terms or additional filters.

Scaling (continuous.scaling.Scaling): This class manages nondimensional scaling. If OpenWInD operates in nondimensional units internally, `Scaling` would have reference values (length scale, time scale, pressure scale, etc.) and methods to convert physical units to scaled units and back. For example, they might choose the instrument length or speed of sound to scale frequencies. The details aren't crucial for usage, but in code you will see `self.scaling` passed around to all components so that they compute dimensionless parameters consistently.

InstrumentPhysics: Building the Graph

Now, the `InstrumentPhysics` class (in `continuous/instrument_physics.py`) orchestrates the creation of the netlist from an `InstrumentGeometry`. Its constructor signature is

`InstrumentPhysics(instrument_geometry, temperature=..., player=..., losses=..., radiation_category=..., spherical_waves=False, discontinuity_mass=True, matching_volume=False, convention='PH1', nondim=False)` – plus possibly more. Many of these correspond to what we've discussed: - `instrument_geometry`: the `InstrumentGeometry` object built earlier. - `temperature`: can be a constant (float) or a function; default might be 20 or 25°C. If a function is provided, the code will evaluate it across the bore. The doc warns that if temperature varies, it assumes linear gradient along main bore and uniform in side holes (and for valves, sets gradient such that the endpoints match the main bore values). - `player`: a `Player` object specifying the excitation. This will influence what type of Excitator is created and possibly which fingering to use (the Player might contain a desired note/fingering). - `losses`: as above, bool or string for loss model. - `radiation_category`: type of radiation model (string like 'unflanged' etc., or a dict mapping each open end to a model if needed). - `spherical_waves`: bool or special string ('spherical_area_corr') – if true, treat propagation as spherical in conical sections. If `'spherical_area_corr'` is specified, they even include an "area correction" in addition to just the end correction. - `discontinuity_mass`: bool, default True – whether to include an acoustic mass lump at diameter discontinuities (to account for the inertia of air expanding into a cavity, known correction at a sudden expansion). - `matching_volume`: bool, default False – whether to include the small volume in a T-junction where the branch connects (tends to add compliance). - `convention`: possibly 'PH1' or 'PH2', indicating how they define positive direction for pressure/flow. (PH might stand for Pressure-Head convention 1 or 2, but not entirely sure; it's likely internal detail affecting sign conventions). - `nondim`: bool, whether to nondimensionalize the system (if True, they'll use the Scaling class to scale all values, which is needed especially for time-domain diffusive representations to avoid numerical issues).

InstrumentPhysics.init does approximately the following:

1. **Initialize empty Netlist:** `self.netlist = Netlist()`. This is the graph that will be populated.
2. **Set up Scaling:** If `nondim=True`, it creates a `Scaling` object (maybe based on instrument length or speed of sound). If false, it might use a dummy scaling where all scales are 1 (so values remain physical).
3. **Thermoviscous Model:** Determine the losses model. If `losses` is False, maybe set model to None or an ideal model. If True or a string, call `ThermoviscousModel = losses_model(losses)` to get a configured model instance.
4. **Go through main bore segments:** For each shape in `instrument_geometry.main_bore_shapes`:
5. Compute the temperature distribution for that segment. If temperature is constant, just pass the value. If it's a function, sample it at segment ends or create a small function for within segment. Possibly they break a long segment if temperature gradient is significant.
6. Create a `Pipe` object: e.g., `pipe = Pipe(shape, temperature_segment, label="MainPipe#N", scaling=self.scaling, losses_model=ThermoviscousModel, spherical=spherical_waves flags…)`.
7. Add this pipe to the netlist: `ends = self.netlist.add_pipe(pipe)` which returns two `PipeEnd` objects (one for each end). For the first segment added:
    - The upstream end is the start of instrument (mouthpiece end). Save that end for later to attach excitator.
    - The downstream end goes to either next segment or if it's the last segment, to radiation.
8. For subsequent segments:
    - Before connecting to previous, if there is a **change in diameter** from the previous segment's end to this segment's start (which there will be, unless radii match exactly), a **junction** is

needed. Specifically, a two-port junction representing the discontinuity. The code will compare the previous segment's end radius and this segment's start radius. If different and `discontinuity_mass=True`, it will instantiate a `JunctionDiscontinuity` (which likely inherits PhysicalJunction).
  - This junction will connect the previous segment's downstream PipeEnd and the new segment's upstream PipeEnd.
  - It will compute reflection/transmission coefficients or added inertance. The Netlist needs to register this connector: something like `self.netlist.connectors[label] = junction_obj` and assign junction_obj's ports to those pipe ends.
  - If the radii are equal (perfectly matched, continuous taper), one might argue no junction needed. The code might still place a trivial connector or simply directly connect the pipe ends (but since their formalism likely always uses connectors between pipes, they might always put a connector, possibly an "Identity" junction or just skip if not needed).
  - If radii difference is zero, the discontinuity could be omitted (two pipe sections of same radius could be merged into one longer pipe theoretically, but they keep segments separate as defined by user because the shape type might have changed or a hole may attach in between).

9. Continue this for all main bore parts. At the very end of the main bore (bell end if open instrument like clarinet), attach a **radiation connector**:
   - Determine the appropriate `PhysicalRadiation` model from `radiation_category` (if that is a dict, maybe check if this particular end has a special condition; usually the bell is one type and toneholes use another – possibly all unflanged except if user wants to model the bell's flange).
   - Create the radiation connector and attach it to the last PipeEnd. Add to netlist connectors.
10. **Add toneholes (side holes):** This part is more involved:
11. For each `Hole` in `instrument_geometry.holes`:
    - The `Hole` provides `position` along the main bore and its `shape` (the chimney pipe). We need to insert this into the main bore.
    - Find which main bore segment corresponds to that position. If a hole is at 0.10 m, the code will find the main bore pipe that spans this location (say a pipe from 0.0 to 0.3 m covers it).
    - In many acoustic models, a tonehole is treated as if the main bore is *cut* at that point, introducing a T-junction. **OpenWInD likely splits the main bore pipe at the hole position**: i.e., if a main pipe ran from 0.0 to 0.3, and hole at 0.1, it will break it into two pipes: 0.0–0.1 and 0.1–0.3. The splitting ensures a node at 0.1 where the side hole connects. The code likely accomplishes this by:
    - Possibly calling something on the pipe or netlist to split a pipe. Or, more straightforward, they might have initially *discretized the main bore by every hole position*. The instrument geometry knows all hole positions, so InstrumentPhysics could have created main bore pipes not exactly as user segments but further split at hole positions. It's not explicitly mentioned, but to insert a junction, you need a node.
    - Another approach: Represent the whole main bore as one continuous pipe but allow connecting the hole via a `Tonehole` connector that internally uses an interpolation to the correct position. However, that's complicated and less robust. It's likely easier to split.
    - Once a node at the hole's position is available, they create a `Pipe` for the hole's chimney (similar to main bore pipes but shorter, radius = hole radius).
    - Add that pipe via `netlist.add_pipe(hole_pipe)` and get its two ends. One end is at the junction with main bore, the other end is the external opening of the hole.

- Now create a **T-junction connector** (`PhysicalJunction` or specifically `JunctionTjoint`) to connect the three ends: main bore upstream, main bore downstream, and the hole's internal end. The `JunctionTjoint` will use the diameters of main bore and hole, and if `matching_volume=True` possibly include a small compliance.
- Mark that junction with a label (like "Hole_junct_HoleName") and add to connectors.
- If the hole is **open** (meaning we want to simulate it open), then at the **external end** of the hole chimney pipe we should attach a radiation connector (since the hole opens to the air). If the hole is **supposed to be closed**, then that external end should be capped – which can be modeled as a high impedance. The simplest way is to attach a `PhysicalRadiation` but for a closed end, radiation model could be set to "reflective/closed". The code may simply handle closed holes by using a radiation model of an *infinite impedance* (no energy loss, full reflection). There might not be a dedicated class for closed end, they could just use a special case in radiation or a different type of connector.
- Another way: define a connector that connects the chimney end to nothing (like an acoustic cap). Perhaps the code uses a `RadiationPade` with parameters that approximate a closed end (which physically is an infinite reflection with a specific 180° phase shift, effectively adding an end correction of about 0.61 * radius as inertive load).
- The decision of open vs closed comes from the **fingering chart**. InstrumentPhysics likely takes a `Player` which may specify a note. If a note is given, the InstrumentPhysics can query the fingering chart: e.g., `fg = instrument_geometry.fingering_chart; for each hole: is_open = fg.is_open(hole.label, note)`. For open ones, do radiation; for closed ones, treat as closed.
- If no fingering chart or note given (everything open default), then all holes are open by default (the code's assumption for "no fingering" case).
- The code also may incorporate the effect of *open holes that are covered by keys but not in use*. Actually, if a hole is not listed in fingering chart (meaning it's always open or doesn't matter?), probably all listed holes are assumed closed unless noted open for each fingering.

12. In summary, for each tonehole we add a branch pipe and a junction. **Important**: The code defines a class `Tonehole(NetlistConnector)` in `continuous/tonehole.py` which "models a complete tonehole as combination of a junction, chimney pipe, and radiation". This hints that in the **time-domain solver**, they aggregate a tonehole's elements into one combined component for efficiency (the comment says "for temporal simulation with locally implicit schemes"). But in the frequency domain assembly, they might either use that or just handle it piecewise.
    - Possibly, `InstrumentPhysics` when building the network for time domain will use `Tonehole` class: i.e., create one Tonehole object that contains the junction, pipe, radiation internally, and connects properly. For frequency domain, maybe they add them separately or still could use the same combined approach.
    - Since the code explicitly has Tonehole class, it might be used to group the three sub-components so that in the netlist `connectors` dict, a Tonehole entry is a single connector connecting main bore and environment via the chimney. The Tonehole class's `__init__` likely takes `junct, pipe, rad` (as we saw) and a label, etc., and inherits from NetlistConnector so that the netlist sees it as one multi-connection component. This is a design choice to possibly allow certain implicit integration where treating the hole as one unit is beneficial.
    - For understanding, we can still think of it as separate pieces logically.
13. **Add valves:** If any `BrassValve` in instrument_geometry:

14. A valve has two connection points on the main bore (where it diverts and where it rejoins). The approach could be:
    - Split the main bore at the divert point and at the reconnect point, so we isolate the section of main bore that would be replaced when valve is open.
    - Add a pipe for the valve's bypass tubing (from divert to reconnect, presumably same length as given).
    - Use two 3-port junctions: one at the divert point (splitting into main bore forward and valve path) and one at the reconnect (merging valve path back in). These two junctions plus the valve pipe form a loop.
    - But controlling whether the valve is open or closed is tricky: if valve is closed (not pressed), the air should *not* go through the valve tube but continue straight. That could be modeled by giving the valve tube or junction an extremely high impedance when closed or, more straightforwardly, by simply *not connecting* the valve path in the model if closed (i.e., effectively treat the side branch as capped at the divert junction, and the main bore continuous).
    - The fingering chart likely includes valves too (with 'o' meaning valve open – pressed – meaning extra tubing engaged, 'x' meaning valve closed – not pressed). The code class `BrassValve` might have a method to configure itself as open or closed. Possibly they instantiate the valve's internal structure always, but if closed, they might short-circuit the side path (maybe by making the valve pipe length zero or something? But more likely by altering the junction).
    - The presence of `BrassValve` class suggests it contains the geometric info, but the actual network building could reuse the tonehole logic if we treat a valve as an extra side branch connecting two points. It's a bit more complex, but presumably handled similarly by splitting and connecting.
15. **Attach Excitator (Source):** Finally, the upstream end of the first main bore pipe needs an excitation:
16. Use the provided `player` to decide what to do. If we are doing an impedance computation, `player` by default is just a steady unit flow source. The code likely calls `exc = create_excitator(player)` which returns an instance of `Excitator` (for example, if player is default, it returns a `Flow` excitator).
17. The excitator will be connected to the very first PipeEnd (the one at x=0 of main bore). The `Netlist` gets this connector added. Typically, that pipe end is then "terminated" by the excitator, meaning it's not free.
18. In time domain, if the player is configured for a reed, `create_excitator` might return a `Reed1dof` object. That object contains state variables (reed displacement, velocity, etc.). It connects at the mouth: one side of the reed sees mouth pressure (an input parameter from the player, which might be in Player as a constant or time-varying blowing pressure), the other side sees the acoustic pressure in the bore at the entry. The coupling yields a flow into the bore. This excitator is nonlinear and requires solving at each time step (we'll cover time solver later).

19. If the instrument is something like an organ pipe (no reed, just a mouth), they might have an excitator type that imposes a pressure function or waveform.

20. **Finalize InstrumentPhysics:** After constructing the netlist with all components, `InstrumentPhysics` likely stores references for convenience:

21. `self.main_bore_pipes` list, `self.hole_pipes` list, etc., or directly it can always access from netlist.
22. It might build a **global system matrix** or leave that to the solver. Typically, InstrumentPhysics might just set up the netlist and not yet assemble matrices; the FrequentialSolver or TemporalSolver does the assembly/solve.
23. However, some pre-computation might be done here: e.g., computing the lengths, speeds, cross-sectional areas discretized etc. But likely, heavy computations are deferred to solver so that if a parameter changes (for optimization), they can rebuild quickly.

In summary, **InstrumentPhysics converts the high-level instrument description into a full network of acoustic elements** ready for analysis. This includes: - Dividing the bore into segments at appropriate points (including at toneholes and valves). - Creating pipe objects for each segment and each tonehole chimney (with appropriate loss models and wave models). - Creating connectors for every junction: between consecutive bore segments (discontinuities), at tonehole branch points (T-junctions), at valve branches, at bore termination (radiation), at tonehole open ends (radiation), and at the input (excitation). - The end result is a `Netlist` of interconnected components representing the instrument.

For a **clarinet** specifically: - The main bore will be a series of cylindrical and slightly tapered pipes connected by very small junctions (most segments will match diameters except where the bore suddenly flares like the bell – there a discontinuity junction accounts for a large expansion to the flared bell section). - Each tonehole (including register key) will introduce a T-junction with a chimney pipe and an opening. The fingering (say we want the impedance for all holes closed for the lowest note) will cause each tonehole branch to effectively end in a **closed condition** (which is like a dead-end resonator off the main bore, affecting impedance but not radiating). OpenWInD will include the shunt impedance of each closed tonehole (chimney + cap) in the network, which is crucial for accurately predicting how closed but uncovered toneholes slightly shift pitches (the chimneys act as side branches). - The mouthpiece end is effectively a closed end with a reed. In the impedance computation scenario, if we just want input impedance, we usually treat the mouthpiece as closed (no flow except a tiny source to probe impedance). The default Player might impose a flow at the entry – that is mathematically equivalent to measuring impedance (Z = p/U). So the excitator in that case is essentially injecting a test flow of 1 and measuring pressure. If one wanted to incorporate a mouthpiece volume or compliance (some models do separate the mouthpiece chamber), that would require adding an extra volume element. The current code doesn't explicitly mention mouthpiece volume, but one could model it as a short segment at the beginning if needed.

At this point, we have a fully specified **continuous model** of the instrument. Next, we look at how OpenWInD computes results from this model in both frequency and time domains.

Discretization and Finite Element Mesh (openwind.discretization)

OpenWInD states that it uses one-dimensional **Finite Element Methods** for modeling【6†source】. The discretization module contains code to break the continuous model (which is essentially a set of differential equations on each pipe) into a solvable numerical form.

Key components likely in the **discretization** package: - `Mesh` (openwind.discretization.mesh): A class that generates and holds the mesh of the entire instrument. It would take the `InstrumentPhysics` (with its netlist) and create finite elements for each pipe, possibly at a certain granularity. - `Element` (openwind.discretization.element): Represents a single finite element segment of a pipe. It might contain matrices (mass, stiffness) for that element if doing FEM. - `glQuad` (maybe Gauss-Lobatto Quadrature? The file glQuad.py suggests integration routines, likely providing shape function integrals). - `discretized_pipe` (maybe a higher-level object combining elements along a pipe). - Perhaps others for assembling global matrices.

In frequency domain analysis, one common approach is to solve the *Helmholtz equation* along the bore with appropriate boundary conditions. The 1D wave equation with losses can be turned into a system of linear equations for pressure/flow at the nodes (if using FEM) or one can directly compute transfer matrices for each uniform segment. OpenWInD likely offers both: - If the instrument is piecewise cylindrical or conical and losses are off, one can use analytic transfer matrices (fast). If complex shapes or losses are on, they might default to numeric solutions (FEM). - The parameter `compute_method` in FrequentialSolver (as mentioned in docs) might allow switching between methods. Possibly `compute_method='freqdomain_fe'` vs `'tmm'` or similar.

The `Mesh` class probably has methods to: - Divide each Pipe into smaller segments if needed. The number of elements might depend on `l_ele` (element length) or `nb_sub` (number of subdivisions per design segment) provided in options. There's also `order` which could refer to polynomial order of each finite element (maybe linear vs quadratic shape functions). - The mesh needs to place nodes at every connection (junction) because that's where boundary conditions are applied. So those positions are fixed nodes. Additionally, long pipes may be subdivided to capture wave effects if using higher frequency or need more resolution. - Once the mesh is established, it assembles global matrices. Perhaps: - A global mass matrix **M** and stiffness (or impedance) matrix **K** such that for a given angular frequency ω, the equation looks like $(K - \omega^2 M) p = f$ (for pressure p and source f). But since they solve in frequency domain directly for steady-state, they might formulate it as an impedance matrix equation. - More straightforward: they might assemble directly an admittance or impedance matrix at the input from network point of view rather than a full FEM matrix inversion. However, given the complexity (especially with toneholes), a matrix approach with nodal pressures might be simpler.

- Because they integrate losses and possibly frequency-dependent terms (the ThermoviscousModel might add complex terms depending on ω), the assembly probably happens inside the FrequentialSolver for each frequency (unless using diffusive reps, which would allow a frequency-independent state-space form).

- The `Element` class likely handles computing the local element matrices given a segment of pipe. If the pipe radius is varying, maybe they approximate it as linear variation within the element or take average cross-section. If the element length is small, a linear variation is fine. If using a higher polynomial order, shape function can approximate radius variation.

- `discretized_pipe.py` might create an internal representation of a pipe broken into elements, maybe precomputing any needed integrals.

For a user of OpenWInD, these details are under the hood. The high-level solver will manage them. We might not need to go into each line, but it's good to know: **Example:** If one chooses a frequency range up to 2 kHz for a clarinet, and clarinet length is ~0.6 m, a rule of thumb is to have about 6-10 points per wavelength. At 2 kHz, wavelength ~0.17 m, so maybe an element size of ~1-2 cm is adequate for accuracy.

OpenWInD might either choose automatically or allow the user to set `l_ele` (element length) or `nb_sub` (number of elements per geometric segment).

In addition, in the **time domain**, the discretization is crucial for stability: the spatial step must satisfy Courant condition relative to time step, etc. The code has specific classes in **temporal** that might implement their own discretization (like a digital waveguide or FDTD approach). But possibly they reuse the same mesh and just use a time integrator.

To summarize, the discretization module builds numerical elements from the continuous model: - If using FEM, it provides matrices for pipes and assembles them with boundary conditions from junctions and radiations. - If using TMM, some parts of discretization may be bypassed (TMM would multiply analytic matrices instead). - Since the code has a unified approach, likely they implemented FEM to handle everything generally, but possibly still have some analytic shortcuts.

Now that we have either an analytical or numerical representation, we move to solving for acoustic responses.

Frequency-Domain Acoustic Solver (openwind.frequential)

The **frequential** subpackage contains the solver that computes the instrument's acoustic response in the frequency domain. The main class is likely `FrequentialSolver` (imported in openwind as well). It takes an `InstrumentPhysics` instance (the built network) and a set of frequencies, and computes something like the driving-point impedance at the input for each frequency.

Key steps in frequency domain solution: 1. **Matrix Assembly:** Using the instrument's netlist, set up equations. One typical method: - Use nodal pressure unknowns at each junction, and possibly two unknowns per pipe if needed (but you can reduce unknowns by using continuity in pipes). - Alternatively, use a modal expansion or wave expansion in each pipe element (like FEM does). - In any case, for each frequency ω, incorporate the effect of each element: * For a pipe element, relate pressure/flow on one end to the other. If doing matrix solve, each element contributes to global M and K as mentioned. * For a junction, apply continuity of pressure (all connected ends have same pressure node) and conservation of flow (sum of flows = 0 at that node, adjusted for sign). * For radiation at an open end, impose the radiation impedance as a boundary condition: it relates the pressure at that node to the outgoing wave (or in FEM, it adds a complex impedance load at that node – often done by adding a stiffness-like term $p/(U) = Z_R$, or one can attach one-port elements). * For a closed end (like a closed tonehole or mouthpiece cap), impose zero flow (Neumann BC) which in a matrix is like disconnecting or adding a large impedance. * For the source: If we want input impedance, we impose a unit flow at the mouth (or unit pressure and measure flow). The typical arrangement is to treat the source as a current source: inject 1 volume velocity at the input node and solve for pressure at that node (the resulting pressure gives impedance Z = p/1). - All these form a linear system $A(\omega) x = b$. The unknown vector x might represent pressures at junctions and flows in some elements. The right-hand side b is non-zero only at the source (representing the impressed source). - Solve for x (which gives pressure at the input node among others). - Compute input impedance $Z = p_{in}/U_{in}$ (with $U_{in}=1$, $Z = p_{in}$ directly). - Repeat for each frequency in the list. This approach, while possibly computationally heavy if many frequencies, is straightforward and can leverage sparse matrices.

1. **Transfer Matrix Method (TMM):** Possibly implemented in parallel for cases where it's efficient:
2. TMM treats each element as a 2x2 matrix linking pressure/flow at one end of an element to the other: $\begin{pmatrix} p_- \ U_- \end{pmatrix} = T \begin{pmatrix} p_+ \ U_+ \end{pmatrix}$. For example, a uniform pipe with losses has known T matrix; a junction has scattering matrix that can be converted to transfer form by augmenting network. However, multi-port junctions complicate pure cascade.
3. They might specifically implement TMM for the main bore (cascading segments) and include toneholes via an equivalent network (which often in acoustics is done by computing an effective shunt impedance of the open hole).
4. Indeed, the presence of modules like `frequential_pipe_tmm.py`, `frequential_junction_tjoint.py`, `frequential_junction_switch.py` suggests they have coded analytic formulas:
    - `frequential_pipe_tmm.py`: likely functions to get the 2x2 matrix of a uniform or conical pipe for a given frequency (with or without losses).
    - `frequential_junction_tjoint.py`: formula for a T-junction's impedance or scattering (maybe the matrix for connecting a tonehole either open or closed).
    - `frequential_junction_switch.py`: possibly a helper that can *switch* the state of a tonehole between open and closed in the model without rebuilding everything, maybe by choosing an appropriate impedance for the branch (like open -> branch radiation, closed -> branch cap).
5. Using these, they could compute input impedance by starting at the mouthpiece and progressively adding elements: e.g., multiply transfer matrices of main bore segments and insert shunt impedance for each tonehole at its location (which is a common approach in impedance calculations of woodwinds: treat each open hole as a shunt to ground (radiation), each closed hole as a shunt to a resonant cavity (closed side branch), etc.).
6. This method can be very fast for initial design because you avoid large matrix solves for each frequency. However, it is less flexible if you have complex shapes (non-constant segments where no analytic formula exists, or if using high-loss models that don't have closed form).

Given OpenWInD's focus on being general and accurate, they probably rely on the FEM approach by default, but possibly the TMM is used when appropriate. The documentation suggests multiple models and robust results, so likely they ensure even a conical bore or a series of toneholes can be done with FE if needed. For everyday instrument sizes and moderate frequency resolution, the FEM approach is fine (especially with modern computing).

1. **FrequentialSolver class (frequential/frequential_solver.py):** This class likely:
2. Stores `instrument_physics` and frequency array.
3. Has options: `compute_method`, `l_ele`, `order`, `nb_sub`, `note` etc as mentioned.
4. In its constructor or a solve method, it will:
    - If compute_method indicates TMM and if all shape segments are TMM-compatible, use the analytic approach (there might be a branch in code to do that).
    - Otherwise, proceed to build a `Mesh` (if not already built) for the instrument. Possibly calls something like `mesh = Mesh(instrument_physics, l_ele, order, nb_sub)`.
    - The mesh would produce the global matrices or something like a system description.

5. Then for each frequency, or possibly all at once if they can do a multi-frequency solve (though usually easier one by one):
    - Configure the system for that frequency (if not frequency-independent).
    - If using diffusive representations for losses, it could be frequency-independent state-space, but let's assume simpler: they treat each ω separately.
    - Apply the source and solve linear equations (maybe using an LU factorization or so).
    - Store the resulting impedance (and possibly the entire transfer function if needed).
6. The results likely saved in `self.impedance` (as a numpy array of complex impedance values corresponding to input frequencies).
7. There might also be methods to compute other derived results:
    - `get_Zc_adim()` – maybe returns the characteristic impedance used for normalization (if nondimensional, they might output dimensionless impedance needing to multiply by reference).
    - Possibly functions to find resonances: but likely that's left to the user or the `peakfinder` module.
8. The `peakfinder.py` in frequential likely contains a function to find local maxima in the impedance magnitude and maybe Q-factors. This is useful for identifying note frequencies and bore tuning. The ImpedanceComputation or user can call that on the impedance result.

ImpedanceComputation high-level class: Finally, at the top level, `openwind.impedance_computation.ImpedanceComputation` provides a user-friendly interface. According to its docstring, it essentially wraps all the above steps automatically: - The user calls `ImpedanceComputation(fs, main_bore, holes_valves=[], fingering_chart=[], **options)`. - Inside `__init__`, it will: 1. Call `InstrumentGeometry(main_bore, holes_valves, fingering_chart, unit=..., diameter=..., allow_long_instrument=...)` to parse the geometry. 2. Create an `InstrumentPhysics(geometry, temperature=..., player=..., losses=..., radiation_category=..., spherical_waves=..., discontinuity_mass=..., matching_volume=..., nondim=...)`. 3. Create a `FrequentialSolver(phys, frequencies=fs, compute_method=..., l_ele=..., order=..., nb_sub=..., note=...)`. 4. Call the solver's solve method (or it might solve in the constructor or lazily on first use). 5. Store the resulting impedance data in `self.impedance` and other attributes like `self.Zc` (maybe characteristic impedance for normalization). - It essentially **"bypasses several classes"** by doing all of this inside, so the user doesn't have to manually call each. However, it still exposes options that are passed through to those classes via `**kwargs`. - It likely has methods like `plot_impedance()` for convenience, or properties for resonance frequencies. But fundamentally, it computes the impedance.

Using **ImpedanceComputation**, a user can get the input impedance curve of an instrument with just a few lines of code. For example, to compute a clarinet's impedance:

```
freqs = np.linspace(50, 2000, 1000)  # frequency range
imp_calc = ImpedanceComputation(freqs, "clarinet_bore.txt",
"clarinet_holes.txt", "clarinet_fingerings.txt",
                                losses=True, radiation_category='unflanged',
note="C4")
Z_in = imp_calc.impedance
```

This would produce the complex impedance as a function of frequency for the clarinet with all holes closed (C4 fingering). One could then find the peaks of `|Z|` which correspond to the resonance frequencies (roughly the playing frequencies for that fingering). Indeed, OpenWInD was used in educational contexts to illustrate how hole positions affect resonance frequencies.

The **Peak Finder** (`openwind.frequential.peakfinder`) likely provides a way to automatically get those peak frequencies and possibly their magnitudes and bandwidths, which could be used to evaluate tuning and timbre.

Time-Domain Simulation (openwind.temporal)

Beyond impedance curves, OpenWInD can simulate the *sound waveform* of an instrument when blown by a player or excited in a certain way. This is crucial for “virtual prototyping” because it allows one to **hear** the effect of design changes (especially for brass and reeds where nonlinear effects determine whether and how the instrument will play a note).

The **temporal** subpackage contains the classes and functions for time integration: - `TemporalSolver` (openwind.temporal.temporal_solver): analogous to FrequentialSolver, but for time domain. - Models of junctions and pipes tailored for time domain (maybe simpler because one deals with time stepping of PDEs). - Possibly uses a digital waveguide model or a finite-difference scheme on the 1D wave equation. - We also see classes like `tjunction.py`, `treed1dof_scaled.py`, `tflow_condition.py`, `recording_device.py` in the temporal folder.

Time-domain simulation approach: 1. They will use the instrument’s netlist (same as before) but now solve the wave equation in time. Typically, one could use an explicit finite difference (leapfrog or similar) or an implicit method if including losses (to avoid instability or stiffness from the small boundary layers). 2. The presence of `treed1dof_scaled.py` suggests the method for a tree of 1D waveguides with a 1-DOF nonlinearity (the reed). Possibly this is an implementation of a specific method (maybe a form of *K-method* or *digital waveguide network*). - The term "tree" indicates they handle branching (toneholes form a tree off the main bore). - They likely split the network at junctions and propagate waves in each branch. - A *locally implicit scheme* might be used at the tonehole junctions to handle the very short side branch delays without reducing global time step (tonehole chimneys can be short and cause stability issues if explicit). - The Tonehole class hint about "locally implicit" suggests that at each tonehole junction, they might solve a small implicit problem for the junction + chimney in each time step, rather than doing the whole system implicitly (this can improve stability for stiff parts). 3. The `TemporalSolver` takes `InstrumentPhysics` and an `Excitator` (like Reed1dof) and integrates over a given time: - It will set up initial conditions (usually at rest, pressure = 0 everywhere). - Then for each time step (like 1/(sampling_rate)): * Compute pressure/flow updates along each pipe segment. If using a digital waveguide, split step: propagate waves, then apply junction conditions. * Update the excitator: for a reed, calculate reed force and flow given current mouth pressure and bore pressure at mouth. * Update radiation: at open ends, outgoing wave is partially reflected (some as feedback, rest lost – implemented as filter or simple coefficient for small time steps). * Use a `RecordingDevice` (class in temporal.recording_device) to sample the output. For example, one might record the pressure at the instrument’s bell or a short distance outside to simulate microphone recording. Or record mouthpiece pressure to see playing frequency stabilization. * Possibly use

`SimulationTracker` (openwind.tracker.SimulationTracker) to print progress and estimated remaining time in a user-friendly way during a long simulation (as indicated by that class's docstring).

- The `simulate(duration, main_bore, holes_valves, fingering_chart, ...)` function in `temporal_simulation.py` is a high-level wrapper like ImpedanceComputation but for time. It likely:
- Constructs InstrumentGeometry from inputs.
- Constructs InstrumentPhysics with a *non-linear excitator* (depending on `Player` or additional arguments like type of instrument).
- Creates a TemporalSolver with certain time step (maybe determined internally from stability criteria or a default like 44100 Hz sampling).
- Runs the simulation loop for the given duration (in seconds).
- Returns or writes the output waveform (maybe as a NumPy array or an audio file). It might also return the time axis or allow the user to access the RecordingDevice's stored data.

The excitator models in time: - **Reed1dof:** The clarinet reed is often modeled as a mass-spring with damping. Its equation of motion might be something like $M \ddot{y} + R \dot{y} + K y = \Delta P * A_{eff}(y) $, where y is reed tip displacement, ΔP is pressure difference (mouth minus mouthpiece), and $A_{eff}(y)$ is the effective opening area as a function of displacement (which saturates when reed hits mouthpiece lay). The flow into the instrument U is then a nonlinear function of both reed displacement (opening area) and pressure difference (like an orifice flow). - The simulation must solve this coupled with the acoustic wave equation each time step. Possibly they use an implicit method for this coupling because of stiffness (the reed mass is small, can be high frequency). - If using `Reed1dof_Scaled`, they might scale variables to avoid extremely small time steps. The code likely solves the reed equation via something like Newton iteration if implicit. - **Lip (Brass) model:** Not explicitly seen in code by name, but if they have `Flute` and `Reed`, they might have a brass lip model not yet in open code or combined under Excitator as well. If not, maybe time simulation for brass uses an equivalent approach with reed model but different parameters (lips can be seen as a 1DOF valve too). - **Flute model:** Usually a feedback loop where the jet delay and amplification cause oscillation. The `Flute` class likely implements something known (like the edge tone model or an empirical nonlinear function linking mouth pressure, mouth hole flow, and oscillation).

The **RecordingDevice** class likely allows you to specify *where* to record. E.g., you might record pressure at the bell end inside the bore (to get the internal pressure wave that would radiate). Or even simulate two microphones at different distances. But given it's 1D, probably recording at the bell yields something proportional to radiated sound.

Example of usage: If one wanted to simulate a clarinet sustained note:

```
simulate(0.5, clarinet_bore, clarinet_holes, clarinet_fingering,
player=Player(blowing_pressure=2000))
```

(This is hypothetical; one would need to specify how to set blowing pressure in Player). The simulation would run for 0.5 seconds and produce the pressure waveform of the resulting sound (likely the periodic oscillation at the clarinet's pitch for that fingering, with possibly some attack transient). The output could then be listened to by converting to audio.

Under the hood, a lot is happening each time step, but OpenWInD manages it with optimized code (C++ extensions or just efficient numpy? not sure, but possibly pure Python might be slow unless small step; there might be numba or something, but not mentioned explicitly).

The time solver ensures *numerical stability*. The locally implicit trick for toneholes: A tonehole can introduce very short time delays (the side branch is like a short stub). If treated explicitly, one might need a very small global time step to resolve it. Instead, they might handle the side-branch internal update implicitly (solving a small equation coupling the junction and the branch within a time step) which allows a larger global step. This is a known technique in some wave digital and finite-difference time domain methods to handle multi-scale issues.

Inversion and Optimization (openwind.inversion)

One standout feature of OpenWInD is the ability to perform **inversion** – that is, adjusting the instrument design to meet target acoustic criteria. This can greatly reduce R&D time by automating what was traditionally done by trial-and-error: e.g., moving toneholes or changing bore diameters to tune the instrument.

The inversion subpackage provides classes for defining an **inverse problem** and solving it: - `Observation` (inversion.observation.py): Possibly a class to hold measured data. For example, an observation might be a set of measured impedance curves or resonance frequencies that we want to match. It might store frequency points and complex impedance values measured from a real instrument prototype. - `InverseFrequentialResponse` (inversion.inverse_frequential_response.py): Likely the main class implementing the inversion for frequency-domain responses. It probably takes an `InstrumentGeometry` (with some parameters marked variable) and a target `Observation` (the desired impedance or resonance data). It then uses an optimization algorithm to adjust the variables. - `display_inversion.py`: Possibly utilities to show progress or results of the inversion (like plotting how the error decreases, etc.).

The inversion likely uses a variant of **Full Waveform Inversion (FWI)** approach, borrowing ideas from geophysics as mentioned. In practice, for instrument design, one might not invert the full frequency response curve (which is high-dimensional). Instead, one often sets specific targets like “the first impedance peak should be at X frequency, second at Y, etc.”. The code might allow either approach: - FWI: minimize the sum of squared differences between entire impedance spectra (or admittance) computed vs measured. - Or targeted: minimize differences in peak frequencies and heights.

To perform the inversion, gradients are needed. This is where those `diff_` methods and `VariableParameter` classes become crucial. The code can compute how a small change in a parameter (like a hole diameter or position) affects the impedance. It likely does this via **finite differences** or via an **adjoint method** for efficiency: - An adjoint approach (like used in geophysical FWI) would involve running a *adjoint simulation* to get gradients in one solve, but that may be overkill for a handful of parameters. - Given they have analytic differentiation for design shapes (like `get_diff_radius_at` and parameter classes), they might actually assemble a **Jacobian matrix** of how each resonance or impedance point shifts with each parameter by direct sensitivity formulas. - Possibly they do something simpler: finite-difference each variable by a small step and recompute impedance changes. But that could be slow if many frequencies and variables.

The `algo_optimization.py` file suggests they implemented a **Levenberg-Marquardt** algorithm (`LevenbergMarquardt` function) with gradient and Hessian or Jacobian information. LM is well-suited for least-squares problems (like matching resonance frequencies). The function likely expects a callback `get_cost_grad_hessian(x)` that returns the current mismatch (cost) and optionally gradient and Hessian. If Hessian is not provided, LM approximates it by JTJ (Jacobian^T * Jacobian). The code mentions `maxiter`, tolerances, etc. The `HomemadeOptimizeResult` class mimics SciPy's OptimizeResult for output (with fields like success, message, n_iter, etc.).

How these are used: - `AdjustInstrumentGeometry` class in technical (adjust_instrument_geometry.py) seems to be an older or higher-level interface: it likely sets up an inversion to adjust one instrument to match another's response. Possibly used for copying designs. - `InverseFrequentialResponse` likely uses `OptimizationParameters` from InstrumentGeometry to get the vector of variables, and sets up a cost function comparing simulated vs target response. - It then calls `algo_optimization.LevenbergMarquardt` with that cost function. The algorithm will iterate: - Each iteration: slightly modify the instrument's geometry parameters (like moving a hole or changing a diameter), then recompute the frequency response (via ImpedanceComputation internally, but more efficiently maybe only at specific frequencies of interest) and compute the error. - Use gradient (differences) to decide how to adjust parameters next. - Continue until error is minimized or parameters converge.

The code likely prints or logs the progress (maybe using `display_inversion` utilities to show current vs target curves, etc.).

In a clarinet design context, one could use inversion to *automatically tune hole positions*. For example, set up a target such that: - The impedance peak frequencies for notes A4, B4, C5, etc., match some desired values. - Mark the positions of those toneholes as VariableHolePosition (with maybe bounds so they don't move too far). - Run the inversion – the algorithm will adjust those positions in simulation until the impedance peaks align with the targets (i.e., instrument is in tune as desired). This is exactly the kind of optimization instrument makers would love to automate and is what OpenWInD aims to do.

Another inversion use: given an **input impedance measurement** of a real instrument, try to reconstruct the bore profile. This is a harder problem (many parameters), but one can parameterize the bore as, say, a series of cone/cylinder segments and let the algorithm adjust those radii and lengths to fit the measured impedance curve. This is akin to how acoustic pulse reflectometry works but done via iterative optimization. OpenWInD's general method could handle that too, if provided with a decent initial guess.

Additional Utilities and Interfaces

OpenWInD includes some additional tools and interfaces to help in using the library:

- **File Parsing Tools (openwind.parsing_tools):** This module contains functions to convert data from other instrument design programs into OpenWInD format. For instance, there is `convert_RESONANS_to_OW(filename)` to convert a **RESONANS** software output (a .dat format) into OpenWInD's lists for bore, holes, and fingering. This suggests that if a user has an instrument defined in the RESONANS program (a known tool for wind instrument acoustics), they can import it into OpenWInD easily. The parsing_tools also likely have functions for writing output CSV files, etc.

- **FreeCAD Integration (openwind.macro_OW2Freecad):** This is a macro or script intended to be used in FreeCAD (a 3D CAD software). It probably reads an OpenWInD instrument description and generates a 3D model (solid or surface) of the instrument bore and holes in FreeCAD. The code uses CadQuery (a Python CAD library) as indicated by trying to import `cadquery`. With this, one could automatically create a 3D CAD model of a designed clarinet, for example, to visualize it or even prepare it for manufacturing. The macro likely places a series of sketches or solids for each bore segment and hole. It's a one-way export (OpenWInD -> CAD). For instrument makers, this is useful to go from acoustic design to a geometrical design that can be prototyped (e.g., 3D-printed).

- **Fingering Chart Utilities:** We already covered FingeringChart and parser in technical. There might also be a class `Fingering` in that module (import shows `FingeringChart, Fingering`). Possibly `Fingering` is a small structure for a single fingering (with name and hole states).

- **Player and Score (technical.player, technical.score):** The `Score` class might allow scripting a sequence of notes (with durations, articulations). For example, one could simulate a scale by changing the fingering and perhaps mouth pressure dynamically. The Player might contain a Score or reference to one. The existence of these suggests one can simulate an *entire musical phrase* on the virtual instrument, not just a steady note. For instance, Player could be configured to start with fingering C4 and certain pressure, hold for some time, then smoothly transition to G4 fingering, etc., and the simulation would follow. This is advanced use, but demonstrates the flexibility of the framework.

- **SimulationTracker (openwind.tracker.SimulationTracker):** A utility to display progress of simulations. For long time-domain runs or heavy optimizations, this will print out elapsed time, estimated remaining time, etc., each few seconds (based on parameters like `min_delay`). It's purely for the user interface (does not affect simulation results).

- **Algorithmic optimization utilities:** We mentioned `algo_optimization.py` having LM implementation. It might also have other solvers or cost functions (maybe gradient descent as fallback, etc.). This is not directly used by typical users, but is called by inversion routines.

- **Animations (openwind.simu_anim):** There is a `simu_anim.py` at top-level. This might be a script or module to create animations of the pressure waves or mode shapes. Possibly it uses matplotlib to animate how pressure distribution changes over time or frequency. For teaching, visualizing standing wave patterns or wave propagation in time would be useful. We won't dive deep, but know that OpenWInD can produce animations (the name suggests something like simulation animation).

Finally, the **package's structure and flow** can be summarized by how a user might typically utilize it: 1. **Define instrument geometry** – e.g., load from file or define inline lists. 2. **Compute frequency response** – via `ImpedanceComputation` class or manually using InstrumentGeometry -> InstrumentPhysics -> FrequentialSolver. Analyze resonance peaks (maybe with peakfinder) to check tuning or acoustic characteristics. 3. **Simulate sound** – via `simulate()` function or manually constructing a TemporalSolver with a Player. Listen to or analyze the output waveform. 4. **Optimize design** – if needed, mark some geometry parameters as variable in the input (using `~` in the file or constructing VariableParameters), set target metrics, and use inversion classes to automatically adjust the design. Then go back to step 2 or 3 to see the improved design's results. 5. **Export design** – use FreeCAD macro to create a CAD model of the optimized instrument for fabrication, or save the geometry to share with colleagues.

Throughout this entire process, the code is structured to be **modular**, and we walked line by line through the logic of each key part. The emphasis on clear abstractions (DesignShape, InstrumentGeometry, Netlist, Solver, etc.) makes it easier to maintain and extend. For instance, one could add a new shape type (say a parabolic flare) by adding a new DesignShape subclass, and everything else would largely work with it. Or add a new excitator model for a different kind of instrument (say a double reed or an air reed) by subclassing Excitator.

OpenWInD's comprehensive approach truly provides a **digital prototyping lab** for wind instruments. By understanding its code and structure in granular detail, as we've done in this review, a developer or researcher can confidently use and modify it to experiment with clarinet designs or any wind instrument – potentially saving a huge amount of time and cost in R&D by virtually testing designs before any physical manufacturing. With this toolbox, one can iterate through design changes (bore profile adjustments, tonehole size/position tweaks) and immediately see their impact on tuning and sound, enabling informed decisions to achieve the desired acoustic performance.

Sources:

- Ernoult, A. *et al.*, "OpenWInD: A software to simulate wind instruments, as a tool for acoustic teachers," *Proc. of the 10th EAA Convention (Forum Acusticum)*, 2023 – (describes the OpenWInD library's capabilities)

- OpenWInD Documentation (Inria) – Code comments and usage examples within the OpenWInD source code (v0.12.0) for geometry file format and class behaviors.

- OpenWInD Changelog (Inria GitLab) – Noting new features such as brass valve support [1], confirming classes like `BrassValve` added.

---

[1] CHANGELOG.md · master · openwind / openwind · GitLab
https://gitlab.inria.fr/openwind/openwind/-/blob/master/CHANGELOG.md