ChatGPT

OpenWInD Package: Detailed Review and Integration Guide

Overview of OpenWInD (Open Wind Instrument Design)

OpenWInD is an open-source Python library (developed at Inria) for modeling and simulating wind musical instruments. It provides a virtual prototyping environment where instrument geometries (e.g. clarinet bores and toneholes) can be defined and their acoustic responses computed. Importantly, OpenWInD supports **frequency-domain (harmonic)** analysis as well as **time-domain** (transient) simulations and even sound synthesis for various wind instruments [1]. This means you can calculate things like input impedance curves (resonance frequencies) *and* simulate how an instrument would sound when "played" with a given excitation. The library uses one-dimensional physics (wave propagation in tubes) with high fidelity: it includes effects like **visco-thermal losses** in the air and can handle open/closed toneholes and other components for realistic results [2]. In fact, OpenWInD has been described in research as a *"virtual workshop"* or an *"optimal design tool"* for instrument makers [3] – exactly aligning with the goal of a digital clarinet prototyping lab.

OpenWInD is written in pure Python and structured as a toolbox with multiple modules, each targeting a part of the instrument modeling process. Below, we provide a **granular, line-by-line oriented review** of the package's structure and key components, explaining what each part does. We will pay special attention to how these features can be **integrated** into your Frappe v15 web app and where to focus on **optimization** for real-time interaction. All features of OpenWInD are covered since they can all be relevant in a comprehensive clarinet design platform.

(Note: OpenWInD 0.12.0 is the version referenced in this review. It is licensed under GPLv3, which is something to keep in mind for integration.)

Package Structure and Key Modules

OpenWInD's code is organized into several modules and sub-packages, each responsible for a different aspect of instrument simulation and design. Here is an outline of the main components:

- `openwind.design` – **Geometry design module**. Defines parametric shapes for instrument bore sections (cylinders, cones, exponential flares, etc.) and provides tools to construct the geometry of the bore (main tube) and toneholes. This is where you describe the clarinet's physical dimensions.
- `openwind.continuous` – **Physical components and models**. Implements the continuous 1D physics of each part of the instrument: e.g., straight pipe segments, tonehole elements, junctions, radiation (sound radiation at open ends), and excitators (like a clarinet's reed). These are the building blocks that form the acoustic network of the instrument.
- `openwind.frequential` – **Frequency-domain solver and analysis**. Contains classes to assemble and solve the acoustic wave equations in the frequency domain (using methods like Transfer-Matrix

Method and 1D Finite Element Method) and utilities to post-process frequency responses (e.g. finding resonance peaks). This is used to compute input impedance curves, resonance frequencies, and other frequency-domain characteristics.

- `openwind.temporal` – **Time-domain simulation**. Implements time-stepping solvers and models for simulating sound wave propagation over time. Includes models of dynamic components like reeds (for clarinets or brass) and time-domain equivalents of pipes, toneholes, etc., enabling sound synthesis and transient behavior simulations.
- `openwind.discretization` – **Discretization utilities (FEM)**. Defines low-level mesh and element classes for the one-dimensional Finite Element Method. The frequency-domain solver can use these to build matrices for complex bore shapes. This module handles dividing the bore into elements and computing shape functions, etc., for FEM calculations.
- `openwind.inversion` – **Inverse design (optimization)**. Tools to solve the "inverse problem": starting from desired acoustic targets, adjust or estimate the instrument's geometric parameters. This includes classes like `InverseFrequentialResponse` which can optimize the bore shape so that the simulated impedance matches a target (e.g. measured instrument or desired frequency profile). SciPy optimization routines (least-squares, etc.) are used here under the hood.
- `openwind.technical` – **Utilities and integration tools**. A grab-bag of useful utilities:
- `instrument_geometry.py` for parsing and assembling the full instrument geometry (bore + toneholes) from input data or files into OpenWInD's internal model.
- `fingering_chart.py` which may define standard fingerings (which holes open/closed for each note) for instruments – useful for analyzing a clarinet's scale.
- `parser.py` to load geometry from text files or other formats (if you have instrument description files).
- `ow_to_cadquery.py` and `ow_to_freecad.py` to convert OpenWInD instrument models into 3D CAD representations (using CadQuery or FreeCAD). This is great for creating actual 3D models of the designed clarinet geometry.
- `player.py` and `score.py` for higher-level simulation control – for instance, to simulate playing a musical score on the instrument (time-domain simulation sequencing notes).
- Other helpers like `default_excitator_parameters.py` (preset values for excitators), `temporal_curves.py` (maybe for plotting or analyzing time signals), etc.
- **Top-level modules**:
- `impedance_computation.py` and `impedance_tools.py` – likely convenience functions to compute input impedance easily given an instrument model, and tools for analyzing impedance (e.g., to compute resonance frequencies or input admittance).
- `compute_transfer_matrix.py` – possibly a utility to compute the transfer matrix of a given segment (useful in frequency domain analysis if using TMM).
- `algo_optimization.py` – implements some custom optimization algorithms (like a Levenberg-Marquardt variant or gradient-based methods) used by the inversion module. In general, OpenWInD relies on SciPy for optimization, but this provides additional control or logging (it defines a `HomemadeOptimizeResult` class and functions to print convergence info, etc.).
- `simu_anim.py` – a utility that might create animations of simulations (for example, animating mode shapes or wave propagation for visualization – useful for teaching/demonstration).
- `macro_OW2Freecad.py` – a script to generate a FreeCAD macro from an OpenWInD instrument (for users who want to open the design in FreeCAD's GUI).

With this structure in mind, let's dive into each major part of the package, explaining how it works line-by-line (conceptually) and noting integration and optimization points along the way.

Geometry Design: Bore and Tonehole Definition (`openwind.design`)

Designing a clarinet (or any wind instrument) in OpenWInD starts with defining the **geometry of the bore and toneholes**. The `openwind.design` module provides a flexible way to describe the shape of tube sections using parametric curves. The key concept here is the **radius profile** $r(x)$ of a section, where x is the position along that section's length. OpenWInD's design classes let you specify how the radius changes along a segment of the instrument.

Important classes in this module include:

- `DesignShape` (abstract base class): This defines the interface for any bore shape. It requires a method `get_radius_at(x_norm)` which returns the radius at a normalized position (0 to 1) along the section. All specific shape classes inherit from this. For example, if you have a section from position `x_start` to `x_end` along the bore, the shape will describe the radius for 0 (at `x_start`) to 1 (at `x_end`). The `DesignShape` classes also often implement methods to get the cross-sectional area, plot the shape, etc., but primarily they encapsulate the radius function.

- `Circle`: Despite the name, this corresponds to a **cylindrical or constant-curvature section**. It uses a circle-arc formula to smoothly connect two radius endpoints with constant curvature. In simpler terms, if you want a *cylinder* (constant radius), that's a special case of a circle (zero curvature). If you want a gentle curve in radius (like a very slight flare), the circle shape can do that. Essentially, you provide two points (e.g., start and end radius and the section length) and it fits an arc of a circle such that the radius changes smoothly between those points. This is useful for designing the main bore in segments or the connection between different diameters (like between the clarinet's cylindrical body and the flaring bell).

- `Cone`: A **conical section** where radius changes linearly with x. If you need a straight cone frustum shape (common in brass instruments or the clarinet's mouthpiece tenon, etc.), this class will give a radius profile that is linear from one end to the other.

- `Exponential`: An **exponential horn** profile. The radius grows (or decreases) exponentially along the length. Exponential horn shapes are often used for clarinet bells or brass bells to achieve certain acoustic impedance characteristics (exponential horns have certain impedance and radiation properties). Using this shape, you can model a classic exponential flare.

- `Bessel`: A **Bessel horn** profile. Bessel horns have a profile based on Bessel functions (commonly used in loudspeaker horns or some instrument bells for specific spreading of sound). If extremely accurate control of wavefront expansion is needed, a Bessel profile might be used. This likely won't be the first choice for a clarinet (which typically has an exponential or custom bell), but the option exists.

- `Spline`: A **spline-interpolated shape**. This allows a completely custom profile defined by a set of points (radii at certain positions) and uses spline interpolation to create a smooth curve. If you have irregular bore shapes or data from an actual instrument's ream (perhaps measured radii at

intervals), you can use a spline to fit those. In design exploration, a spline might be used for fine-tuned custom shapes or imported bore data.

- **Design parameters**: The design module also includes concepts of variable parameters (`VariableParameter`) and some evaluation utilities (`eval_`, `diff_`). These are likely used to treat certain dimensions as variables that can be optimized. For example, you might designate the length or radius of a section as a variable parameter so that the inversion/optimization routines can adjust it. Under the hood, these might wrap numerical values and keep track of gradients or changes for use in the optimization algorithms.

How these pieces come together: To define a **main bore**, you would typically break the clarinet's bore into a sequence of sections (e.g., barrel, upper joint, lower joint, bell) or even smaller segments if needed. Each segment can be assigned a shape type. For instance, a simplified clarinet main bore might be: - 1st segment: cylindrical (constant radius ~15 mm) for the majority of the bore, - last segment: an exponential flare for the bell (expanding radius towards the open end).

In code, you might do something like:

```
from openwind.design import Circle, Exponential
# Define a 60 cm cylinder (approx clarinet length minus bell)
cyl_section = Circle(r1=7.5e-3, r2=7.5e-3, length=0.6)  # radii in meters
# Define a 7 cm exponential bell, radius from 7.5mm to say 30mm at the end
bell_section = Exponential(r1=7.5e-3, r2=15e-3, length=0.07)
# (This is just an illustration; actual clarinet radii vary and the bell flare
might be more complex)
```

Each shape class typically takes parameters like starting radius `r1`, ending radius `r2`, and section length. Internally, it will set up the math to compute radius at any point. For example, `Cone.get_radius_at(x_norm)` likely does `return r1 + (r2 - r1)*x_norm` (linear interpolation). The `Circle` will use a bit of geometry to maintain curvature. The `Exponential.get_radius_at` might do `r1 * exp(k * x_norm)` with `k` chosen so that at x_norm=1 it reaches r2, etc. The exact lines in the code define these formulas, but conceptually that's what's happening.

For **toneholes or side tubes**, the design of their **chimneys** (the drilled hole's bore) can also use these shapes. Typically, a tonehole is short and can be approximated as a short cylinder of a certain diameter (and maybe a slight flaring at the open end if undercut). If needed, you could specify a small conical or curved shape for the tonehole if you want to include undercutting. The design classes are flexible enough to handle those as well.

Integration tip: In a Frappe-based web app, you would expose controls for these geometric parameters to the user. For example, you might have fields for each segment's length and diameter, or a selection of bell profile type (cylindrical vs exponential). When the user inputs these, you will create the corresponding `DesignShape` objects in Python. If the user doesn't care about choosing a shape formula, you can default to something (like cylinder for body, exponential for bell) and just let them tweak numeric values.

Optimization note: Treating geometry parameters as `VariableParameter` can allow automatic optimization. If your goal is to auto-tune the instrument, you can parameterize the bore (e.g., lengths of sections or diameters at certain points) as variables. OpenWInD's inversion module can then adjust those. We will discuss this more in the optimization section, but the design module is where those parameters originate. One suggestion is to keep the number of variable parameters as low as necessary for optimization (each variable adds a dimension to the search space). For example, you might choose just a few key parameters: the bore diameter (if you want to optimize cylindrical bore size), the bell flare coefficient, and perhaps tonehole positions or diameters. This balances flexibility with complexity.

Instrument Assembly and Components (Bore, Toneholes, Excitators)

Once the basic shapes are defined, they need to be assembled into a full **instrument model** with acoustic elements. OpenWInD provides higher-level classes to do this assembly, bridging the design geometry to the physical simulation components.

Instrument Geometry Parsing (`InstrumentGeometry` class)

The central class that brings everything together is `InstrumentGeometry` (found in `openwind.technical.instrument_geometry`). This class acts as a parser/builder: it takes in definitions of the main bore and the holes (or valves, for brass) and creates the corresponding network of acoustic elements. In practice: - You provide `InstrumentGeometry` with either file names or data structures containing the bore profile and the hole specifications. - It reads these and constructs the instrument.

For example, if you have a file describing the main bore radius at various points along its length, `InstrumentGeometry` will read it and create a series of `Pipe` elements (from `openwind.continuous.pipe`) that represent each segment of the bore. Likewise, for each tonehole defined (with parameters like position along the bore, diameter, chimney length, etc.), it will create a `Tonehole` object (from `openwind.continuous.tonehole`) at the appropriate location.

Internal structure: Within `InstrumentGeometry`, there are helper classes defined: - `Hole` – likely a simple class to hold a tonehole's geometric info (the shape of the hole's tube and its position on the main bore, plus a label or identifier). For example, `Hole.shape` could be a `DesignShape` for the hole's cross-section (usually a cylinder), `Hole.position` is the distance from the top of the bore, and `Hole.label` an identifier (like "Hole 1" or note name). - `BrassValve` – similar concept but for a valve's tubing on brass instruments (like the extra tubing added when a trumpet valve is pressed). In a clarinet context you won't use valves, but the class exists in the code for completeness.

`InstrumentGeometry.__init__` likely does the following steps (though not explicitly seen without the full code, this is inferred from typical usage and documentation): 1. **Parse main bore data:** If you passed a filename, it loads it (possibly via `technical.parser`). If you passed a list/array of geometric data, it uses that directly. The data format might be a sequence of points (distance, diameter) or segments. It then uses the `openwind.design` shapes to create the profile. Simpler case: it might assume the data is already piecewise linear or in segments. In any case, it will create one or more `Pipe` elements that represent sections of the bore. 2. **Parse holes data:** Similar approach – parse each hole (or key) specification. Each hole likely needs: position on bore, diameter (bore of the hole), chimney length (the length of the drilled

hole), and whether it's open or closed (for a given fingering, but by default the instrument geometry might assume all holes open or define keys separately). 3. **Construct the acoustic network:** Using the continuous models: - For the main bore: it will chain the `Pipe` elements to form the main duct. If the bore diameter changes, there may be a `Junction` element at the interface of two pipes with different radii (to account for reflection due to an abrupt or gradual change – OpenWInD's continuous model likely handles continuous radius changes internally if using FEM, but for TMM an abrupt change would be a junction). - For each tonehole: it will insert a `Tonehole` object at the correct point. The `Tonehole` class (in `openwind.continuous.tonehole`) is a compound component that includes: - A **T-junction** connecting the main bore and the tonehole's entrance. (In acoustical terms, a tonehole is like a side branch: acoustically a three-port junction – one port to incoming wave, one to outgoing wave, one to the hole). - A short **chimney pipe** representing the hole's tube (with the given diameter and length). - An **open-end radiation** impedance at the end of the hole (accounting for the fact that the hole opens to the air, radiating sound). The Tonehole class likely builds these sub-components and internally is able to switch between open vs closed state (closing a tonehole is essentially like capping the side branch so no flow goes out – openwind might simulate that by making the radiation an infinite impedance or by removing the branch connection). - If there are keys or complicated mechanisms (like multiple toneholes under a key, or register vents), those might be handled by either defining multiple holes or by using the fingering chart to specify their state per note.

1. **Excitator (mouthpiece/reed):** The instrument assembly might also include adding an excitator at the input end. For example, a clarinet has a mouthpiece+reed. In OpenWInD, the mouthpiece + reed can be modeled using an Excitator object (from `openwind.continuous.excitator`, e.g., `Reed1dof`). The excitator would connect at the beginning of the bore. However, `InstrumentGeometry` might not automatically add it; the excitator is often added when you actually run a simulation (e.g., the `FrequentialSolver` might allow specifying a mouth impedance or a source, and the `TemporalSolver` will definitely need an excitator to drive the system). We will touch on excitators more in the simulation sections. Just be aware that the *instrument* geometry primarily covers the passive resonator (bore + holes), while the exciter (reed, or lips, or air-jet for flute) is separate.

After constructing these components, you get an `InstrumentGeometry` instance that contains the full specification of the clarinet. This likely includes: - A list of `Pipe` sections for the bore (with their lengths and radii), - A list of `Tonehole` objects with their placement, - Possibly information about the input (mouth) and output (bell) ends.

From here, this instrument geometry can be fed into the simulation engines.

Integration tip: You might not need to use `InstrumentGeometry` if you prefer to build the continuous model directly in code, but it is very convenient especially if you have a standard format for instrument definition. For your app, you could allow advanced users to upload a text file (or copy-paste data) describing a bore profile and holes, then pass it to `InstrumentGeometry` to construct the model. For more interactive use, you can bypass files and directly create an `InstrumentGeometry` by passing Python lists/arrays for the bore and hole definitions. For example:

```
# Example of constructing instrument geometry from Python lists (pseudo-code):
bore_data = [
```

```
    # length (m), diameter (m) pairs or perhaps (position, diameter) points
    (0.0, 0.015),  # start radius 7.5mm
    (0.50, 0.015), # 50cm of 15mm diameter cylindrical
    (0.57, 0.020), # flare to 20mm by 57cm (maybe the bell section)
    (0.60, 0.030)  # end at 60cm with 30mm diameter
]
holes_data = [

# Each hole might be defined by: position (m from top), diameter (m), length
(m), maybe chimneys shape
    (0.15, 0.006, 0.02),  # hole at 15cm, 6mm diam, 2cm chimney
    (0.23, 0.006, 0.02),
    # ... etc for all toneholes
]
instrument = InstrumentGeometry(bore_data, holes_data)
```

The exact expected format might differ (some tools specify holes with additional parameters like if it's open or closed in a default fingering), but you can adapt to what `InstrumentGeometry` expects. Once you have this object, you can then proceed to analysis.

Continuous Physics Components (`openwind.continuous`)

This module contains the classes that represent the actual physical behavior of each part of the instrument. After using `InstrumentGeometry`, you effectively get a collection of these components connected together. Key classes here include:

- `Pipe` (in `continuous.pipe`): Represents a cylindrical (or tapered) pipe section of the bore. It will be characterized by a length and a radius (or a radius profile if non-uniform). Internally, if using the Transfer Matrix Method (TMM), a pipe has a known matrix relation between pressure/flow at its ends. If using FEM, the pipe will be meshed (via `discretization`) and contribute to the global matrices. There might be subclasses or variations like `PipeLossy` or `PipeRough` for including wall roughness or other loss models (we saw files like `tpipe_lossy.py`, `tpipe_rough.py` in temporal, and similarly in frequency domain might handle losses separately). For design purposes, a normal `Pipe` will already include visco-thermal losses if you enable those in the solver.

- `Junction` (`continuous.junction`): A general connection element where two or more ducts meet. For example, at a diameter discontinuity, a junction connects a larger pipe to a smaller pipe (two-port junction). For a tonehole, a three-port junction connects the main bore and the side hole. The `Junction` class handles the boundary conditions and continuity of pressure/flow at that point. OpenWInD likely has specialized junction subclasses:

- `JunctionDiscontinuity` (for an area step change in the main bore),
- `JunctionTJoint` (T-shaped junction for toneholes),
- `JunctionSimple` (maybe a simplified model of a small hole or register vent),

- etc. (We saw files like `frequential_junction_switch.py`, `frequential_junction_tjoint.py`, etc., which suggests different junction configurations in frequency domain.)

- `Tonehole` (`continuous.tonehole`): As mentioned, this is a composite that likely contains a junction + short pipe + radiation. In code, when you instantiate a Tonehole, it will internally create:

- A `FrequentialJunction` (in freq domain) or `T-junction` (in time domain) linking the main bore to the hole.
- A short `Pipe` representing the tonehole's chimney.

- A `Radiation` impedance at the hole's open end to simulate radiation from that opening. The Tonehole class probably manages whether it's open or closed by, for instance, setting the hole's pipe length to effectively zero or the radius to zero when closed, or by switching boundary conditions. In some models, an open tonehole is an acoustic load on the main bore (a shunt impedance), while a closed tonehole behaves like a inertial stub (if the pad is closed, the air in the hole still moves a bit but doesn't radiate). OpenWInD might handle closed holes by treating them as a capped pipe (so still a stub load). These details are handled internally by the physics classes.

- `Radiation` (`continuous.radiation_model` and related): This models how sound radiates out of an open end (be it the main bell or an open tonehole). Acoustic radiation impedance is frequency-dependent (and for a bell, directional), but in a 1D model, they incorporate an impedance at the end that approximates the radiation load. OpenWInD includes various radiation models:

- `radiation_pulsating_sphere` (perhaps a simple spherical radiation model),
- `radiation_silva` (maybe based on Silva's model for flanged openings or similar),
- `physical_radiation` (a presumably more advanced model possibly using measured data or a full solution),

- `radiation_from_data` (perhaps allows using externally provided radiation impedance data). The clarinet's bell radiation might be handled by a specific model or by a combination (e.g., an expansion plus an end correction). The toneholes radiation likely uses a simpler model (like an unflanged pipe radiation formula). For integration, usually you won't need to tweak these – they are set up by default when building the instrument, but it's good to know they exist in case you want to experiment with different end models.

- `Excitator` (`continuous.excitator`): This is the model for the *source* of excitation – e.g., a reed, a mouth pressure for a flute, or lip for brass. OpenWInD has an abstract `Excitator` class and specific ones:

- `Reed1dof` (one-degree-of-freedom reed model)【42†】: Used for clarinet/saxophone cane reeds and for brass lips. It's a lumped model (mass-spring-damper) that can open/close and modulate airflow nonlinearly. It takes parameters like reed stiffness, mass, damping, mouth pressure, etc. There is also likely a simpler static nonlinear model for reeds.
- Possibly an `AirJet` or `Flue` excitator for flutes (the file `tflute.py` in temporal suggests a flute jet model).

- `MouthPressureCondition` (perhaps a boundary condition class that just provides a pressure source without reed dynamics – could be used for a simplified model or for calibrating input impedance by forcing an oscillation).

In frequency-domain analysis, the excitator is typically not included (because we usually compute *passive* input impedance with no active oscillation). Instead, we treat the input as closed or open for impedance calculation. However, in time-domain, the excitator is crucial to simulate actual sound. We'll cover usage in the temporal section.

Visco-Thermal Losses and Realism: The continuous models incorporate losses (as indicated by research references [4]). In practical terms, this means that the `Pipe` elements have an option to include viscous friction and thermal conduction loss in the air. This is often done via a complex frequency-dependent component (e.g., by adjusting the wave number or impedance with boundary layer effects). OpenWInD likely does this automatically for you. There might be a global flag or simply the default behavior, but it's good to note that the simulation won't assume lossless pipes (unless you intentionally turn losses off for some reason). This makes the results more "real world" (e.g., impedance peaks will have finite height and bandwidth, as in a real clarinet).

Integration considerations: When integrating these components through `InstrumentGeometry`, you mostly deal with high-level parameters. But if your app wants to allow detailed control (for advanced users), you could expose some of these lower-level options: - e.g., a toggle for "include thermoviscous losses" (though default should be yes, since it's more accurate). - or the ability to choose a radiation model (maybe an "accuracy" setting: simple vs advanced radiation, which might trade off computation speed slightly). - or adjusting reed parameters (strength of reed, mouthpiece lay, etc.) if you allow time-domain play simulation.

For most design purposes, however, you won't tweak those – you'll stick to geometry. The continuous module ensures that given a geometry, the physics is properly represented.

In summary, after this assembly step, we have a complete model of the clarinet's acoustical system: a network of pipes, junctions, and toneholes that corresponds to the instrument's design. Now, we can simulate this network either in the frequency domain (to get resonance frequencies, impedance curves) or in the time domain (to get actual sound or time response). We'll examine each in turn.

Frequency-Domain Acoustic Analysis (`openwind.frequential`)

Frequency-domain analysis is fundamental for instrument design because it tells us the **resonant frequencies** of the bore – which correspond to the notes the instrument can play (for a given fingering) and their tuning. OpenWInD's frequential module is responsible for computing things like the input impedance $Z(f)$ or input admittance of the instrument as a function of frequency. By analyzing $Z(f)$, we can identify peaks that indicate the resonances (standing wave modes) of the instrument [4]. For a clarinet, the sequence of impedance peaks (starting from the lowest) corresponds to the playable tones (the fundamental and its harmonics for a given fingering). Designers aim for those peaks to align with desired musical frequencies.

Key elements of the frequential analysis in OpenWInD include:

- `FrequentialSolver`: This is likely a class that takes an `InstrumentGeometry` (or directly the continuous components) and sets up the equations to solve in the frequency domain. It probably has methods like `solve()` or `compute_impedance()` which perform the calculation. The solver must assemble the global system representing the instrument:
- If using **Transfer Matrix Method (TMM)**: it multiplies transfer matrices of each section to get a total transfer function from mouth to radiating end. TMM is very fast for piecewise cylindrical or conical segments and is often used for quick calculations.
- If using **Finite Element Method (FEM)**: it builds a system of linear equations (sparse matrices) using the mesh from `discretization` and solves for pressures given a source. FEM is more flexible (handles arbitrary shapes like a smooth Bessel or measured profile) and can more precisely include losses for any shape. It is a bit heavier computationally than TMM, but still quite feasible for 1D problems.

- OpenWInD actually implements both and even has a unified approach for simple shapes [5]. In practice, the code might automatically choose TMM for sections that are cylindrical or conical, and resort to FEM for complex profiles. Or possibly the user can choose a mode. The research paper cited [5] indicates that they ensure a *"seamless formulation"* combining cylinders and cones in TMM, and they quantify when TMM vs FEM is more efficient or accurate. Essentially, **for standard clarinet geometries (mostly cylindrical with a conical bell and small tonehole branches)**, the TMM will be very fast and reasonably accurate. If high precision is needed (or if using exotic shapes), FEM can be used. The solver likely has an option or automatically does this by inspecting the shape objects.

- **Solving for impedance:** To get the input impedance, typically one would:

- Apply a sinusoidal pressure at the mouth end (or a volume velocity) and compute the resulting flow or pressure, then compute $Z = p / U$ (pressure over volume flow) at the frequency of interest.
- Repeat this for a range of frequencies.

The `FrequentialSolver` probably allows you to specify a frequency range or a list of frequency points. It then either sweeps through them solving one by one (if using FEM, solving a linear system for each frequency) or directly computes an analytic expression (if using TMM, one can get a formula or just compute each frequency anyway since TMM is fast).

It might store the results in arrays (frequency vs impedance or admittance). Possibly it also can compute modal parameters (some research lines mention computing modal frequencies and shapes via an eigenvalue problem [4], which might be an advanced feature where they solve for poles of the system rather than sweeping frequency – this is more advanced and not strictly needed for basic design, but know that academically, OpenWInD can extract those).

- **Visco-thermal losses in freq domain:** As mentioned, these are included. For the user, the result is that impedance peaks are damped (finite Q). No action needed, but it ensures simulation matches reality (e.g., clarinet's higher harmonics have lower peaks due to losses).

- `peakfinder` (`frequential.peakfinder`): This utility likely scans a computed impedance or admittance curve and finds the local maxima. It would return the frequencies (and possibly

magnitudes and bandwidths) of each resonance peak. This is extremely useful for automating tuning analysis. For instance, if you simulate the clarinet with all holes closed (like lowest note), peakfinder can find the first peak (which should correspond to the fundamental fingering's note frequency, say around 147 Hz for low E on a clarinet) and higher peaks (which might correspond to the third harmonic etc., since clarinet, being cylindrical and closed at one end, has predominantly odd harmonics). If the frequencies are off from the desired values, you know you need to adjust geometry.

Similarly, for a different fingering (some holes open), you would get a different set of peaks. By comparing those to a target scale (like equal-tempered scale frequencies A4=440Hz etc.), you can quantify intonation errors.

- **Other frequential components**: The package has classes for each element in frequency domain:
- `FrequentialPipeTMM` vs `FrequentialPipeFEM` – implementations of a pipe for each method.
- `FrequentialJunction*` classes for different junction types.
- `FrequentialRadiation` for radiation impedance frequency-dependent behavior.
- `FrequentialSource` perhaps to define how the excitation is applied in frequency domain (like a source impedance or a known volume velocity at the input).
- `FrequentialPressureCondition` might represent a boundary condition of fixed pressure (or open end at pressure = 0 reference).
- `tmm_tools.py` might include helper functions specifically for transfer-matrix computations (like composing matrices, etc.).
- `FrequentialInterpolation` might be used to interpolate results or to create a continuous impedance function from computed points (maybe used in the inversion process to compare to target smoothly).

As a user of OpenWInD, you normally wouldn't need to directly manipulate these low-level classes; the `FrequentialSolver` and `InstrumentGeometry` handle it. However, understanding that they exist helps if you ever need to extend or debug the behavior (for example, if you suspect an issue with how a tonehole is being modeled at high frequencies, you might look into `FrequentialJunctionTJoint` etc.).

Performing a frequency analysis in code: After you have your `instrument = InstrumentGeometry(...)`, doing a frequency analysis might look like:

```
from openwind.frequential import FrequentialSolver
solver = FrequentialSolver(instrument)  # pass in the instrument geometry
freqs = np.linspace(50, 2000, 1000)    # say 50 Hz to 2000 Hz range
impedance = solver.get_impedance(freqs)
```

(Actual API may differ, but conceptually). The result might be an array of complex impedance values for each frequency. You could then find peaks:

```
from openwind.frequential import peakfinder
peaks = peakfinder.find_peaks(abs(impedance), freqs)
```

The `peakfinder` likely returns frequencies at which local maxima occur. If it's sophisticated, it might also do curve-fitting to refine peak frequency or get Q-factor.

Integration tip: For a **real-time design app**, you will likely use the frequential analysis in two ways: 1. **On-demand calculation:** When the user adjusts a parameter (say moves a tonehole), you can recompute the impedance curve and update the displayed resonance frequencies or a chart. To keep it real-time, you might not always compute a dense 0-2kHz sweep with 1000+ points; you could compute fewer points or use prior results as an initial guess. However, given modern computation and the 1D nature, even 1000 frequency points solving via TMM is very fast (on the order of milliseconds). FEM would be slower but still might be okay if mesh is moderate. If performance is an issue, see optimization notes below. 2. **Automated checks/optimization:** When the user triggers an "optimize tuning" action, you would use the frequency analysis to drive the optimization (comparing peaks to targets). This might involve many repeated impedance computations (every time the optimizer tweaks a parameter). To optimize this, you can reduce the frequency grid to just around expected peaks or use the modal extraction method (solving eigenvalue problem for resonance frequencies directly, which OpenWInD can do). That can speed up convergence since you directly get peak frequencies rather than scanning.

A practical integration approach is to **pre-compute baseline results** (for an initial design) and then **update incrementally**. For example, if a user changes one tonehole diameter slightly, the high-frequency peaks might shift more than the low-frequency ones. You could in principle reuse the previous impedance as a starting point or only recalc a part of the spectrum. OpenWInD itself doesn't automatically do partial updates – you would just recompute – but given the speed, that's okay. If you needed to optimize further: - Running computations in a background task (Celery/Redis in Frappe context) is wise for heavy operations to avoid blocking the web request. The user can get a loading indicator or partial results streaming in.

Optimization notes (frequency domain): - *Choose the solving method appropriately:* If OpenWInD allows a toggle between TMM and FEM, use TMM for quick feedback during interactive tweaking (since clarinets are largely cylindrical, TMM will be accurate for low-to-mid frequencies). Use FEM or a finer computation if needed for final verification or if the bore shape is very non-standard. - *Frequency resolution:* Don't oversample the frequency axis more than needed. For example, if you only care about identifying peaks, a step of 5-10 Hz might be enough, plus maybe refine around peaks. The `peakfinder` might handle interpolation to get sub-Hz accuracy if needed. A coarse sweep can be done in real-time, and a finer sweep can be done as a secondary step or on demand. - *Parallel analysis:* If you want to evaluate *all fingerings* of a clarinet (i.e., the impedance curve for each fingering), note that these are independent simulations on slightly different instrument configurations (open/closed holes). You can parallelize those across multiple threads or processes. OpenWInD itself might not do that, but you can manage it. For example, if optimizing a whole set of notes, evaluate their impedance peaks concurrently to save time. In Python, true multi-threading is limited by the GIL, but heavy computations in numpy release the GIL. Alternatively, use the Python multiprocessing module or job workers to compute different fingerings in parallel. This is an advanced optimization, but worth noting if you anticipate computing dozens of impedance curves repeatedly (e.g., optimizing an entire scale's tuning simultaneously).

Now that we have frequency-domain capabilities to get resonance info and guide design adjustments, the next aspect is the time-domain simulation, which allows us to actually "play" the instrument model – a valuable feature for sound and behavior verification.

Time-Domain Simulation and Sound Synthesis (`openwind.temporal`)

While frequency analysis is used for tuning and design, time-domain simulation addresses **how the instrument actually sounds and behaves when played**. OpenWInD's temporal module provides a solver that can simulate the dynamic interaction of the air, the instrument, and the excitator (like the clarinet's reed) over time. This essentially performs a virtual play test of the instrument design.

Key components in the `openwind.temporal` package:

- `TemporalSolver` (likely in `temporal.temporal_solver`): This would be the main class managing the time simulation. It sets up the time integration (probably an explicit or implicit scheme) for the partial differential equations / ODEs representing the instrument. It would integrate variables like pressure and flow in each segment, and the motion of the reed (if a reed excitator is used), over small time steps. The solver needs to enforce stability and accuracy – possibly using an energy-conserving scheme or a standard approach as described by the OpenWInD authors in their publications [6] (they have work on time-domain simulation of dissipative reed instruments).

- **Time-step components**: Each element in the continuous model has a time-domain counterpart:

- `tpipe` / `tpipe_lossy` / `tpipe_rough`: models of a pipe segment for time stepping, with or without losses, and possibly a separate model if wall roughness is considered (roughness can cause scattering and additional losses).
- `ttonehole`: a time-domain tonehole model. This likely handles the opening/closing of the tonehole and the small cavity effect when closed.
- `treed1dof` (`temporal.treed1dof`): the time-domain reed model. This is a critical piece for clarinet simulation. It's typically a non-linear equation that relates the pressure difference across the reed to the reed opening area and thus the flow. It involves the reed's equation of motion (mass * acceleration + damping * velocity + stiffness * displacement = pressure force difference). The code implements this and couples it to the acoustic wave in the bore. There might also be `treed1dof_scaled` (perhaps a scaled version for numerical stability) and possibly separate handling for a lip-reed vs a single reed if needed, but likely unified.
- `tpressure_condition` and `tflow_condition`: these might be boundary conditions at the input – e.g., a prescribed mouth pressure or volume flow. In a simulation, you often set a mouth pressure waveform (like a constant or a slowly ramped pressure to start a note) and let the reed/bore system oscillate. Alternatively, one can model a controlled flow source. The `player` utility might wrap this to simulate a musician blowing with a certain pressure profile.
- `tradiation`: time-domain radiation boundary condition at the output (bell or open holes). Possibly an convolutional model or a digital filter that approximates the frequency-dependent radiation impedance in time.
- `tsimplejunction`, `tjunction`: time domain junctions (for toneholes, etc.).
- `execute_score`: possibly a function to step through a sequence of notes (like changing fingerings or mouth pressure over time, according to a musical score input).
- `recording_device`: maybe a simulated microphone or way to record the output during simulation (could output a waveform).

- `utils.py`: some utility functions, maybe for unit conversions or waveform generation.

- **How time simulation works:** To simulate a clarinet note, you would:

- Define the instrument (bore + holes, as before).
- Choose a fingering (which holes are open/closed). This might involve toggling the Tonehole objects' state. OpenWInD likely has a way to programmatically set fingering, or you might rebuild the instrument with certain holes closed (i.e., treat them as just a sealed part of the bore).
- Attach an excitator model. For a clarinet, you would use `Reed1dof`. You must set parameters for the reed: for example, reed natural frequency, opening area at rest, stiffness, mass, and the mouth pressure applied. There is probably a `default_excitator_parameters.py` which provides reasonable defaults (for a clarinet reed, etc.), so you're not working blind.
- Set initial conditions (usually at rest: no sound, reed in equilibrium).
- Start the simulation by gradually increasing mouth pressure to simulate blowing. The solver will begin time-stepping. If conditions are right (mouth pressure above a threshold), the reed will start oscillating and a self-sustained oscillation will establish at one of the instrument's resonance frequencies (for clarinet, typically the first peak for lower registers, or a higher mode if you "overblow").
- The simulation will output the pressure at the bell or some microphone point over time – essentially the sound wave. You can record this to an array (and even play it back as audio if scaled to audio range).

The simulation can also cover transients like attack and decay of notes, and if the fingering or blowing changes over time, you can simulate a melody or articulation (the `score.py` likely helps orchestrate that, by changing fingerings or mouth pressure according to a timetable or musical notation).

- **Using time simulation for prototyping:** In a prototyping app, time-domain simulation might be used for a few purposes:
- **Auralization**: Allow the user to **hear** what the designed clarinet might sound like. For instance, after designing, you could simulate a few sustained notes or a scale and provide an audio output. This can be very impressive for a maker to hear the difference a design change makes (though caution that a simple 1D reed model might not capture everything about timbre, it gives a rough idea).
- **Transient behavior**: Some design aspects like how quickly a note speaks (attack time) or stability of higher registers can be evaluated. For example, if toneholes are too large or placed such that certain notes overblow too easily or have weak fundamentals, a time sim could reveal that by showing the instrument jumping to a higher mode or having a slow build-up.

- **Nonlinear effects**: Frequency-domain analysis is linear and doesn't show things like amplitude, saturation, or non-harmonic oscillation. The time domain (with the nonlinear reed) will show how the instrument saturates, how loud it can get for a given mouth pressure, etc.

- **Performance considerations for time sim**: Time simulations are computationally heavier than frequency sweeps. They require small time steps (e.g., to resolve frequencies up to 2kHz, you might need a time step of < 1/4000 s for stability/accuracy, so maybe 1e-4 s). To simulate 1 second of sound, that's 10,000 steps. The equations per step involve updating each pipe section and solving the coupling with the reed. This is still quite doable in Python thanks to vectorization (the pipes can be updated as an array operation perhaps), but it's not instantaneous. A 1-second simulation might take several seconds to compute, depending on step size and complexity. This is fine for an "on-

demand" feature (like clicking a Play button), but not for continuous real-time as the user drags sliders. So you likely wouldn't run the time solver on every small design change.

A strategy is to use time sim as a *secondary* tool: once a design looks good via frequency analysis, the user can request a sound simulation. That kicks off a computation (possibly on the server, maybe as a background job if it takes noticeable time) and then returns an audio file or a graph of the waveform. If using Frappe, you could have an endpoint to fetch a synthesized audio (maybe base64-encoded WAV) to play in the browser.

- **Integration with user inputs**: The user might specify things like "blowing pressure" or "note to play". You can provide controls for these. For example, a dropdown of fingering or note name, a slider for how hard to blow. These map to parameters in the simulation (which holes are open, and the mouth pressure value). You'd then run the simulation for perhaps a fixed duration (e.g., 0.5s to hear a note) and output the result.

- **Visualization**: Apart from audio, you can visualize aspects of the time simulation:

- For instance, how the reed opening changes over time, or how the pressure wave moves in the bore (OpenWInD might allow exporting an animation or at least the data along the bore vs time). The `simu_anim.py` likely can create an animation of pressure distribution along the instrument over time. In a web app, you might not use that directly (since it could output a matplotlib animation), but you could generate a video/gif to show the user or translate the concept into a WebGL animation if extremely ambitious.
- A simpler visualization is plotting the waveform of the sound or its spectrum (FFT). E.g., show the user the harmonics content of the produced sound, which relates to timbre.

One more feature: Fingering Chart integration – The `fingering_chart.py` in technical likely contains a mapping of note names or finger numbers to a combination of open/closed holes. If it specifically includes clarinet, it could list standard Boehm system fingerings (like which holes open for each note from E3 up to say C6). If so, you can use it to automate switching the instrument state. For example, you could do `InstrumentGeometry.apply_fingering(fingering_name)` to close/open the correct holes, then compute the impedance or simulate sound. If the OpenWInD library doesn't have clarinet fingerings built-in, you may implement your own chart in your app. But having it would save time and ensure consistency.

Summing up time-domain: This capability is extremely powerful for a "digital clarinet lab" but should be used thoughtfully in integration. It's best for qualitative assessment and impressing the user (hearing the instrument, seeing time response). For the core design loop (geometry -> tuning), the frequency domain is faster and more directly tied to tuning. Time domain comes in when needed to verify or demonstrate playability.

Inverse Design and Optimization (`openwind.inversion`)

One of your key goals is **automatic optimization of the instrument geometry based on simulations** – essentially letting the computer figure out how to tweak the clarinet's dimensions to meet target performance (like hitting the exact desired note frequencies). OpenWInD directly supports this through its inversion module, which is a standout feature for reducing R&D time.

The main idea of the inversion (inverse problem) is: *given a desired acoustic response, find the geometric parameters that produce it*. In practice, you might specify something like "I want the first impedance peak at 147 Hz (D3), the second peak at 294 Hz (D4, a twelfth above, since clarinet overblows a twelfth), etc., for this fingering" and let the algorithm adjust lengths/diameters to get there.

Important classes/functions in this context:

- `InverseFrequentialResponse` (in `openwind.inversion.inverse_frequential_response`): This class extends the `FrequentialSolver` and adds the ability to define a **cost function** based on the difference between simulated results and target specifications. For example, the cost function could be the sum of squared differences between the simulated impedance peaks and the target frequencies (and possibly amplitudes) that you want. Or it could be difference in entire impedance curve shape if matching a measured instrument. The class likely allows you to set what observation/feature to match (they mention "observation on the simulated impedance" vs "the same observation on a reference"). Observations could be something like "the frequencies of the first N peaks" or "the impedance magnitude at certain frequencies". The inversion class then provides methods to compute the gradient of this cost w.r.t. design parameters and uses an optimizer to adjust the parameters.

- `Observation` (in `openwind.inversion.observation`): Possibly defines what we are observing in the frequency response. For instance, an observation might be "peak frequencies", "peak magnitudes", "impedance at specific frequency", etc. It likely has some predefined options (the mention of `implemented_observation` hints at multiple observation types). For clarinet design, the most obvious observation is peak frequency positions.

- **Optimization routines**: The inversion uses SciPy's optimizers (as seen by imports of `scipy.optimize.minimize`, `least_squares`, etc.) as well as custom algorithms (`homemade_minimization` in `algo_optimization.py`). They likely set up a Jacobian (sensitivity of impedance features to each geometric parameter) and use a Levenberg-Marquardt or similar algorithm for efficient solving. The code defines things like gradient tolerance, cost tolerance, max iterations, etc., and prints messages about convergence. The `InverseFrequentialResponse` class probably wraps this all so you as a user can simply call something like `inverse = InverseFrequentialResponse(instrument, observation=..., target=...)` then `result = inverse.optimize(parameters_initial_guess)`.

- **Design parameters linking**: How do the geometric parameters tie in? This is where the earlier `VariableParameter` from the design module comes in. The instrument geometry must be parameterized. For example, you might define:

- Parameter 1 = length of barrel,
- Parameter 2 = diameter of bore,
- Parameter 3 = diameter of first tonehole,
- etc. These would be fed into the instrument model instead of hard-coded numbers. Possibly `InstrumentGeometry` or the continuous classes support being updated or built from parameter values. The inversion class then tweaks these parameters. There might be a need to provide a

function that, given a parameter set, rebuilds the instrument model (or updates it) and computes the impedance. This function is used by the optimizer repeatedly.

OpenWInD might automate some of this if you create the instrument using certain classes. For instance, if you use `design.VariableParameter` for a dimension, the system might internally know how to adjust the geometry. If not, you can still do it by writing a small wrapper function for the optimizer that takes parameter values, creates a new InstrumentGeometry, runs FrequentialSolver, and computes cost. It will be slower (re-building instrument each iteration), but for a handful of parameters it's manageable.

- **Constraints**: The inversion code imports `Bounds`, `LinearConstraint`, `NonlinearConstraint`, indicating you can put constraints on the parameters. This is **important in practice**: you don't want the optimizer to produce absurd or non-manufacturable results, like a negative hole diameter or a bore length that is too short. With constraints, you can enforce limits (e.g., hole diameters between, say, 4mm and 8mm, or bore length fixed if you want). You might also tie parameters together (linear constraint: e.g., total instrument length fixed while distributing amongst sections). Nonlinear constraints could be used for things like "hole i must be at least 1 cm away from hole j" (to avoid overlap).

- **Using the inversion in your app:** Suppose you want to implement an "Auto-Optimize" button for clarinet tuning. The steps could be:

- **Choose parameters**: Select a small set of geometry parameters to vary. For example, you might allow optimization of **barrel length**, **bell flare parameter**, and a couple of **tonehole positions** (or maybe all tonehole positions within a certain range). More parameters means more freedom but also a harder optimization problem. It's wise to start with fewer.
- **Define targets**: For each parameter set, you need a target acoustic outcome. If focusing on tuning, pick a few notes and their target frequencies. For instance, ensure that with all holes closed, the first impedance peak = target low E frequency; with the first hole open (next fingering), its first peak = target F#, etc. You can run multiple inversions or a combined cost that sums over multiple fingering configurations. The inversion module might not natively handle multiple fingerings at once – you might have to define a composite cost yourself that simulates each fingering and accumulates error. Since OpenWInD is Python, you can script this: define a cost function that calls FrequentialSolver for each fingering of interest, finds the peak, compares to target, and sums squared differences.
- **Run optimizer**: Use SciPy's `least_squares` or OpenWInD's `InverseFrequentialResponse`. If using OpenWInD's, you might do something like:

```
inv = InverseFrequentialResponse(instrument, observation="peaks",
target=target_values)
result = inv.least_squares(x0=initial_params)
```

  where `x0` is an initial guess array of parameter values. The result will contain optimized `x` (parameters) and info about convergence.
- **Update design**: Take the optimized parameters and update your instrument geometry accordingly. Then re-run a fresh simulation to confirm the performance and show the user the outcome (e.g., "After optimization, your note frequencies are ..." perhaps all within a few cents of target).

During this process, it's important to communicate with the user, since optimization might take a bit of time (a few seconds to minutes depending on complexity). You can show a progress indicator or logs (the `print_cost` in `algo_optimization.py` prints iteration, cost, gradient, etc. – capturing those prints to show in the UI could be useful feedback so user sees it's working).

Optimization tips and considerations: - **Initial guess**: The outcome heavily depends on starting from a reasonable design. The better your initial guess (perhaps based on an existing clarinet design), the more likely the optimization will converge to a good result quickly. If you start from a random or very off design, the algorithm might get stuck in a bad local minimum or take a long time. - **Scaling**: The `InverseFrequentialResponse` likely handles scaling of parameters or cost, but ensure your parameters are scaled roughly similarly for the optimizer. For example, one parameter in the range of 0.001 (m) and another in tens (like a number of toneholes, hypothetically) could cause issues. Usually length and diameter in meters are all around 1e-3 to 1e-1 range, which is fine. - **Multiple objectives**: If trying to tune multiple notes at once, sometimes one design parameter affects several notes (especially tonehole positions can have complex effects). The optimization might need to balance conflicting requirements (like you can't independently tune every note with just a couple of parameters – trade-offs happen, just as in real instrument making). You might need to prioritize or iterate: e.g., tune the primary scale first, then adjust another parameter for one troublesome note, etc. There's no single solution if too few parameters; conversely with many parameters you can perfectly fit many targets but the solution might be non-unique or unrealistic (overfitting). - **Verification**: Always verify the optimized design with a fresh simulation. Also consider secondary criteria: e.g., maybe it hit the frequencies right, but what if one hole ended up extremely large or in a weird spot? The constraints help avoid extremes, but also use your domain knowledge (the output geometry should still resemble a playable clarinet – if something looks off, you might need to restrict that in the optimization or adjust weighting). - **Speed**: Each iteration of optimization calls the frequency solver (possibly dozens or hundreds of times). If each call is 100ms, and you need 100 iterations, that's 10s – not bad. But if it's slower, it could be minutes. Using a coarser frequency grid or fewer target points can speed it up. Also consider using the **Jacobian** if available: SciPy least_squares can use analytic Jacobian if provided – OpenWInD might provide gradient calculations (the mention of gradient in `HomemadeOptimizeResult` suggests they do compute it). If Jacobians are available, optimization converges in far fewer iterations than using finite differences. OpenWInD likely does have analytic sensitivities for peak positions with respect to geometry via adjoint or finite difference internally. Trust their implementation if using the built-in inversion class.

Integration in Frappe: - Run the optimization as a background job to avoid blocking the UI. Provide the user with a way to initiate it and then poll for results or refresh when done. - Use the results to update the design fields in your app's database (so the new optimized geometry is saved). - Perhaps log the before-and-after in a report so the user can see how each parameter changed and how much the frequencies improved. - If the user wants to fine-tune further, they could adjust weighting of targets or fix some parameters and rerun optimization. Your UI could allow selecting which aspects to optimize (maybe a checkbox "optimize this parameter or keep it fixed").

OpenWInD's inversion tools essentially give you the "automatic luthier" ability – a huge advantage to shorten R&D. By leveraging them, your digital lab can not only simulate designs but also actively suggest improvements.

Integration with Frappe Framework (v15)

Now, let's focus on how to bring all these capabilities into your app built on the Frappe framework (version 15). Frappe is a full-stack framework (Python backend, built-in database, and a client-side system) often used for apps like ERPNext, but it's flexible enough for any web application. Here are detailed suggestions for integration:

1. Installing and Managing OpenWInD: Make sure the OpenWInD package (and its dependencies like NumPy, SciPy, Matplotlib, CadQuery if needed) are added to your Frappe app's environment. In Frappe, you might include it in `requirements.txt` so that when you install the app, pip installs openwind. Since OpenWInD is on PyPI, this is straightforward (`openwind>=0.12.0`). Keep an eye on version updates; as of writing, 0.12.1 is out [7], so consider using the latest for bug fixes.

2. Backend Structure: In Frappe, you can create server-side Python scripts (either in doctype controllers or separate module files) that will handle calculations: - **Design Data Models**: Define DocTypes for things like "Instrument Design" or "Clarinet Prototype" which store user inputs (bore lengths, diameters, etc., maybe one child table for holes). This way, each saved document corresponds to a design that can be revisited. - **Controller Methods**: Write whitelisted methods (or use Frappe's REST API) to perform simulations. For instance, a method `compute_impedance(design_docname)` that: - Fetches the design parameters from the document, - Constructs the OpenWInD `InstrumentGeometry`, - Runs `FrequentialSolver` for the relevant fingering (maybe all holes closed or a specific fingering passed as a parameter), - Returns the computed resonance frequencies or even the whole impedance curve. - Another method could be `optimize_design(design_docname)` that triggers the optimization routine (perhaps using RQ (Redis Queue) which Frappe uses for background jobs). - For time domain, a method like `simulate_sound(design_docname, fingering, pressure)` that runs a short time simulation and stores or returns the audio.

These methods can be invoked via AJAX from the client-side, allowing a dynamic web interface.

3. Real-time Interaction (Client-side): Using Frappe's front-end (which can use JS and HTML templating or even React/Vue if you integrate), you can create a page where: - The user adjusts sliders or inputs for geometry. You might use Frappe's form if using DocType, or a custom web form. - On change, use JavaScript to call the backend method (Frappe provides `frappe.call` for RPC calls). For example, when a user changes a hole diameter, you call `compute_impedance` for that design. - The backend returns data (e.g., an array of peaks or a small JSON with frequencies). The front-end JS then updates the displayed info (for instance, updates text showing "Resonant frequency = 148 Hz" or moves a marker on a chart). - If you want truly continuous feedback (like dragging a slider and see a graph update continuously), you may need to throttle the calls (avoid sending too many calls if user is dragging). Compute on mouseup or every few hundred ms. The frequency analysis is fast enough to do this in near-real-time if implemented carefully. - For visualizing the **impedance curve**, you can integrate a chart library. Frappe has Chart.js integration or you can use Plotly, etc. The backend can either send raw data points which the JS plots, or even send a pre-plotted image (Matplotlib can save a PNG and you display it). But dynamic JS plotting is nicer for interactivity (e.g., allow user to zoom). - For **3D geometry display**, consider using CadQuery + a viewer. One approach: use `ow_to_cadquery.OWtoCadQuery` to generate a CadQuery solid of the instrument. Then export it to a format usable in browser. CadQuery can produce a STEP or STL file. You could then use a JavaScript 3D viewer (like three.js) to display that model. There is also a library called `cqparts` or others to convert

CadQuery to Three.js JSON. Another simpler way: use VTK or JSCAD. However, initially you might skip 3D rendering and just show important dimensions or rely on user's imagination. But since a "prototyping lab" would benefit from seeing the instrument, a static 2D schematic (like a drawn bore profile with holes) could be generated easily (Matplotlib can draw the side view given the geometry). - **Fingering interface**: Provide controls for which note/fingering is being analyzed. This could be a dropdown of notes, or an interactive diagram of clarinet keys the user can toggle. When a fingering changes, call backend to recompute impedance for that fingering. The `InstrumentGeometry` combined with `fingering_chart` can simplify this: you might just call something like `set_fingering("G4")` then compute. Otherwise, manually instruct which holes open/closed for that fingering.

4. Background Tasks for Heavy Jobs: Frappe uses background workers for any long task: - Mark the optimization method as a long job. It can enqueue the job and immediately return a job ID. The front-end can poll for job completion or use socketio (if configured) to get a message when done. This way, the user can continue to use the interface or get a loading spinner rather than the whole site freezing. - The optimization job, once done, could update the DocType with the new optimized parameters and perhaps store the achieved frequencies. - Similarly, a time-domain simulation that runs for a few seconds of audio should be a background job. It can, for example, save the waveform to a file (WAV) or to the database (maybe as an attachment or base64 string). The client can then be prompted to load or play that audio.

5. Data Storage and Versioning: Each instrument design might go through iterations. You can leverage Frappe's versioning (if you use a DocType with Version, or you manually copy designs). It could be helpful to keep track of various designs (maybe one per user or multiple variants). Also consider allowing export/import of designs (e.g., in a JSON or CSV format) so users can share or archive outside the system.

6. User Experience Considerations: - Provide default designs: On first use, load a standard clarinet geometry (from literature or measured). This gives users a starting point to tweak rather than starting from scratch (which can be overwhelming and less meaningful without context). - Provide guidance or tooltips (for instance, hovering over a parameter explains it: "Tonehole diameter: affects venting and impedance of that hole; larger diameter raises venting cutoff making notes higher in pitch and louder but too large can weaken certain registers" – basically encode some expert knowledge in the UI). - Validate inputs: Before sending to OpenWInD, check basic validity (positive lengths, diameters, physically possible configurations). This prevents crashes or nonsensical results. For example, if a user accidentally sets bore length to 0 or negative, you can catch that and show an error before running. - Error handling: If OpenWInD does raise exceptions (say, the solver fails to converge at some parameter extreme), catch those and inform the user gracefully (e.g., "Simulation failed at those parameters, try a more realistic value"). - Logging: Keep logs of OpenWInD operations for debugging. Perhaps expose a "view log" if something goes wrong, which could contain the internal warnings or messages (OpenWInD prints messages like optimization stops or iterative progress).

7. Frappe v15 specifics: Ensure compatibility: - Python 3 is used (OpenWInD requires Python 3, which is fine). - The server environment might not have a display for Matplotlib. If OpenWInD tries to show plots (like using `plt.show()` in some functions), that could cause issues. But since you won't use those plotting functions on the server (you'll likely generate data and plot on client, or explicitly use Agg backend to generate files), it should be okay. If needed, set Matplotlib to a non-GUI backend by default (like `matplotlib.use('Agg')`) to avoid it trying to open windows. - CadQuery and FreeCAD usage: If you use `ow_to_freecad`, note that FreeCAD is a heavy dependency and not usually on a server. Instead, `ow_to_cadquery` with CadQuery is pure Python. CadQuery relies on an OpenCASCADE kernel – which

should be pip-installable (cadquery package includes OCC). Just test this in your environment. If installation is troublesome, you can skip CAD generation or offload it to clients (or just let users download the geometry data). - Frappe UI could potentially embed a 3D viewer or an audio player. Use HTML5 `<audio>` tag to play audio results (after getting a URL or base64 audio). For 3D, use a `<canvas>` with three.js if going that route.

By integrating OpenWInD in this way, your app essentially becomes the GUI that OpenWInD itself lacks. You'll be providing instrument makers or researchers a friendly interface to a powerful simulation engine, all within a modern web app structure. This combination can indeed **significantly reduce R&D cost and time**, as designs can be tested and optimized virtually with far fewer physical prototypes.

Performance and Optimization Considerations

To ensure the digital prototyping lab feels responsive and efficient, here are some additional optimization-focused tips drawn from the code and general best practices:

- **Numerical Efficiency**: OpenWInD mostly uses NumPy for heavy calculations, which is quite fast. Ensure NumPy/SciPy are using optimized linear algebra libraries (BLAS, LAPACK). In a typical environment this is true, but if you're on a custom setup, using Intel MKL or OpenBLAS can speed up matrix operations significantly. This is relevant for FEM solves and large matrix ops in frequency domain. It's less of an issue for TMM (which is analytical) or small systems, but it's good hygiene.

- **Cache Reuse**: If you are doing repeated calculations on slightly varying designs, consider reusing parts of the computation:

- The FEM mesh and system matrices for the bore don't change if you only tweak one tonehole slightly. Currently, OpenWInD doesn't have an obvious incremental update mechanism (it likely rebuilds everything when you rerun solver). But you could cache the `InstrumentGeometry` or the `FrequentialSolver` object. For example, if user is tweaking one parameter slowly, you could keep the solver alive and just update that parameter in the model if possible, rather than reconstructing from scratch each time. This might require digging into OpenWInD internals to update a radius value and reassembling matrices partially. Given the complexity, you might opt to just recompute fully (clear approach) unless profiling shows a bottleneck there.

- A more straightforward cache: store results for certain configurations if they might repeat. Perhaps not very applicable unless user toggles between a few known states.

- **Parallel Processing**: We touched on this, but to reiterate: if you need to compute multiple scenarios (like whole scale fingerings) frequently (say to display a chart of impedance for every note at once), leverage parallelism. Python's multiprocessing can spawn separate processes each running OpenWInD on a different fingering. Just be mindful of memory (each process loads the instrument data, but that's small) and start-up overhead. For a one-off calculation of 20 fingerings, that's fine.

- **Matplotlib and Plots**: If generating plots on server (like impedance vs freq graphs), creating a Matplotlib figure can be somewhat heavy. Prefer to send raw data to client and use client-side plotting when possible. If you must generate images (e.g., for PDF reports or something), do it

asynchronously. Also, when using Matplotlib inside a web app, remember to close figures (`plt.close`) to free memory.

- **Memory usage**: The instrument models are not huge (a clarinet with maybe tens of elements). Even time simulation arrays (10k time steps * few hundred state variables) are a few million data points at most, which is fine (tens of MB of RAM). But if you simulate very long or with extremely fine resolution, memory can spike. Monitor if needed and set practical limits (no need to simulate more than a few seconds perhaps).

- **Error and Stability**: Some extreme designs might cause numerical instability (e.g., a very large diameter hole might make an impedance matrix ill-conditioned, or time simulation blow up if the time step is too large for a very short segment). The code likely includes warnings (we saw it imports `warnings` and uses them in adjust_instrument_geometry for example). Capture those warnings and handle them. For instance, if a certain configuration is on the edge of stability, you might warn the user "This design may produce unstable oscillations or numerical issues – consider adjusting parameters."

- **Floating Point and Precision**: Usually double precision (default in NumPy) is fine. There's no need for higher precision. But be careful with unit conversions (OpenWInD likely expects SI units: meters, seconds, Pascals). Ensure all inputs you feed are in the correct units to avoid ridiculously large/small numbers that waste precision or cause slow convergence.

- **Leverage the Community/Docs**: The OpenWInD documentation (available on Inria's site) likely has *how-to guides* and *tutorials*. Using those examples as a starting point can save time. For example, they might have a tutorial for a clarinet or saxophone design – you can adapt that code into your app. Also, being aware of known issues or planned improvements (via their GitLab or forums) can guide you (for instance, if a future version improves optimization or adds features, you might plan to upgrade).

- **GPL License**: A quick note beyond technical optimization – since OpenWInD is GPLv3, if your app is distributed (especially commercially or publicly), you must also open-source your code under GPLv3 compatible terms. If this is an internal tool or open-source anyway, it's fine. Just keep this in mind; integration of a GPL library imposes that licensing on the combined work. This isn't a technical optimization, but it can affect how you deploy or share the app.

Finally, remember that **physical feasibility** should guide the optimization. Sometimes pure numerical optimization might suggest a solution that is hard to implement (like a tonehole placed exactly where a key post might be, or a bore shape that's too complex to machine). As the developer of the tool, you can add heuristics or post-filters to the results (for instance, round tonehole diameters to standard drill sizes, or ensure tonehole spacing is adequate for fingers). Such considerations keep the "digital prototyping" grounded in the real-world constraints, further saving time when moving to actual production.

Conclusion

OpenWInD is a comprehensive toolkit covering everything from shape design to acoustic simulation and automatic optimization. In this detailed review, we walked through each part of the codebase and discussed

how it contributes to a digital clarinet prototyping lab: - The **design module** lets you define the instrument's geometry in detail. - The **assembly and continuous models** translate that into a physical network representation including all the important acoustic effects (toneholes, losses, radiation). - The **frequency-domain analysis** provides rapid feedback on tuning and resonance characteristics – essential for iterating designs. - The **time-domain simulation** adds realism, allowing you to "hear" the instrument and assess playability. - The **inversion/optimization** tools enable automated design improvements, potentially finding geometry configurations that meet desired targets with minimal manual trial-and-error. - Throughout, we considered integration with a Frappe web app, ensuring that these powerful but complex functions can be made accessible through a user-friendly interface, with real-time updates and efficient computation.

By carefully integrating OpenWInD into your Frappe v15 tech stack and following the optimization strategies, you can create an interactive platform where instrument makers or researchers design a clarinet virtually, get instant insights into its performance, and iterate quickly. This "virtual workshop" approach has been shown to accelerate the instrument design process [3], thereby **significantly reducing the cost and time of R&D**.

In sum, OpenWInD provides the physics engine and analytical depth, and your app will provide the user experience and orchestration. Together, they form a cutting-edge digital clarinet prototyping lab, turning what used to require weeks of crafting and testing into a matter of minutes of computation and adjustment – all while maintaining the accuracy needed for real-world confidence.

Sources:

- Chabassier, J., et al. *"Openwind, a virtual workshop assisting wind instrument makers."* (2022 focus article) – highlights OpenWInD capabilities [1] [2] [3].
- OpenWInD official documentation and repository (Inria, 2019-2025) – code analysis and comments.
- OpenWInD research references (JSV 2022, Acta Acustica 2019, etc.) – background on numerical methods used [8] [9].

---

[1] [2] [3] [4] [5] [6] [8] [9] Juliette Chabassier | Openwind, a virtual workshop assisting wind instrument makers
http://juliettechabassier.perso.math.cnrs.fr/public/focus/2022/virtual-workshop/

[7] openwind · PyPI
https://pypi.org/project/openwind/