User Manual

Common Tasks

View PRE or POST wafer profiles

1. Upload a PRE or POST `.sbw` file.
2. Select `Thickness` or `Flatness` from the dropdown menu.
3. Select `PRE` or `POST` in the segmented control.
4. Select one or more **Slots** from the multiselect dropdown menu. If multiple slots are selected, the plots are displayed in order.
5. (Optional) Check `Mask notch` in the sidebar to mask notch and filter out outlier values.
6. Click **Plot**.

View REMOVAL wafer profile

1. Upload **both** PRE and POST `.sbw` files.
2. Select `Thickness` or `Flatness` from the dropdown menu.
3. Select `REMOVAL` in the segmented control.
4. Select PRE slots and POST slots. If counts differ, the slots are paired in order.
5. (Optional) Check `Overlay line charts` in the sidebar to show PRE/POST on top of REMOVAL line charts.
6. Click **Plot**.

View average wafer profiles

1. Check `Average Profile`.
2. In PRE/POST profile mode: plots the average radial profile for the selected profile mode.
3. In REMOVAL profile mode: plots the average removal profile.

User Interface

Top controls

- Upload PRE .sbw | Upload POST .sbw: load `.sbw` files.
- `Thickness | Flatness` dropdown menu: select graph mode.
- `PRE | POST | REMOVAL` segmented control: select profile mode.
- `Average Profile` checkbox: switch to average profile mode.

Sidebar — Display controls

- `Color clip low (%)`: slide to set the lowest percentile used for color range to reduce the effect of a notch (outlier values) skewing colors (default 0.5).
- `Color clip high (%)`: slide to set the highest percentile used for color range (default 100).
- `Mask notch`: replaces notch (outlier values) (beyond $k \times MAD$, default k = 4) with NaN to prevent notch from skewing colors.
- (`REMOVAL` only) `Overlay line charts`: overlays PRE/POST lines on removal line plots.

Angle selection

- **Angle slider**: slide to select an angle for a single-angle line chart.

The angle and direction at which the wafer has been line-scanned is indicated by the arrow shown on the icon of a wafer on the top right of the chart.

Controls and interactions

- **Hover**: shows x/y/z values.
- **Pan/Zoom**: use mouse to drag/scroll in plots.
- **Turnable rotation**: use mouse to drag and turn the surfaces.
- (`REMOVAL` only) ▶ / ◀ **button**: to switch between 2D and 3D plots for PRE and POST.

Code Explanation

Utility Functions

`reset_plot()`

A reset switch to control session state.

```
def reset_plot(flag_key: str):
    st.session_state[flag_key] = False
```

Streamlit reruns when the user interacts with the application (e.g., selecting new slots). `st.session_state[flag_key] = False` ensures that plotting is initiated by the **Plot** button.

The function is used in `st.multiselect`, for example:

```
    ...
    plot_key = f"do_plot_{profile_mode}"
    sel = st.multiselect("Slots", labels, default=None, key=f"{profile_mode}_slots",
                         on_change=reset_plot, args=(plot_key,))
```

Changing slot options resets session state, requiring the **Plot** button to be clicked again. `on_change=reset_plot` invokes `reset_plot()` to be run whenever the widget's value (slots in this case) is changed. `arg=(plot_key,)` then passes `plot_key=f"do_plot_{profile_mode}"` as the argument to `reset_plot()`. As a result, when the widget's value is changed, `st.session_state[plot_key]=False`.

```
    if st.button("Plot", key=f"plot_btn_{profile_mode}"):
        st.session_state[plot_key] = True
    ...

    if st.session_state.get(plot_key, False):
        if not sel_keys:
            st.warning("Choose at least one slot.")
        else:
            if avg_profiles:
```

Then, the code above checks if `st.session_state[plot_key]` is `True`. Plotting is initiated only when the user has selected slots and clicked the **Plot** button.

`average_profile()`

Compute average radial profile by combining both + r and - r sides.

```
def average_profile(Z_line: np.ndarray) -> np.ndarray:
    Z_line = np.asarray(Z_line, dtype=float)
   if Z_line.size == 0:
       return np.array([])
   Z_full = np.vstack([Z_line, Z_line[:, ::-1]])
```

```
    with np.errstate(all='ignore'):
        return np.nanmean(Z_full, axis=0)
```

`Z_full = np.vstack([Z_line, Z_line[:, ::-1]])` stacks the original (+ r) array and the mirrored (- r) array vertically (`::-1` reverses the sequence). Then, the function returns the average of the stacked arrays. (`np.errstate(all='ignore')` suppresses warning messages.)

SBW File Parsing and Cleaning

`parsecleansbw()`

Parse (using `parsesbw()`) and clean (using `cleansbw()`) the `.sbw` file uploaded by the user, and return it in a cleaned dict format.

```
def parsecleansbw(uploaded_bytes: bytes) → Dict[str, Any]:
    import tempfile
    obj = None
    with tempfile.NamedTemporaryFile(delete=False, suffix=".sbw") as tmp:
        tmp.write(uploaded_bytes)
        tmp_path = tmp.name
    try:
        obj = parsesbw(tmp_path)
        return cleansbw(obj)
    finally:
        try:
            os.unlink(tmp_path)
        except Exception:
            pass
```

- `parsesbw()` parses the raw `.sbw` file (using a file path) and returns an `sbwinfo` object.
- `cleansbw()` takes an `sbwinfo` object and converts it into a clean dictionary.
- `parsecleansbw()` bridges these two functions: it takes the raw file bytes uploaded by the user, writes them into a temporary `.sbw` file on disk, so that it can use `parsesbw()` to parse that file and use `cleansbw()` to convert it into a clean dictionary, which is the final output of this function. In essence, this function returns the output of `cleansbw()`, and its input is the uploaded file bytes instead of a file path.

Wafer Matrix & Slot Caching

`Thkmatrix()` & `Flatmatrix()`

Build a 2D thickness/flatness matrix with rows = Angle and columns = Radius.

```
def Thkmatrix(wafer):
    r = np.asarray(wafer.get('Radius', []), dtype=float)
    theta = np.asarray(wafer.get('Angle', []), dtype=float)
    profiles = wafer.get('Profiles', [])
    nt, nr = len(theta), len(r)
    Thk = np.full((nt, nr), np.nan, dtype=float)
    for i in range(nt):
        line = np.asarray(profiles[i], dtype=float) if i < len(profiles) else np.array([], dtype=float)
        if line.ndim == 2 and line.shape[1] > 0:
            Thk[i, :min(nr, line.shape[0])] = line[:nr, 0]
        else:
            Thk[i, :min(nr, line.size)] = line.ravel()[:nr]
    return r, theta, Thk

def Flatmatrix(wafer):
    r = np.asarray(wafer.get('Radius', []), dtype=float)
```

```
    theta = np.asarray(wafer.get('Angle', []), dtype=float)
    profiles = wafer.get('Profiles', [])
    nt, nr = len(theta), len(r)
    Flat = np.full((nt, nr), np.nan, dtype=float)
    for i in range(nt):
        line = np.asarray(profiles[i], dtype=float) if i < len(profiles) else np.array([], dtype=float)
        if line.ndim == 2 and line.shape[1] > 1:
            Flat[i, :min(nr, line.shape[0])] = line[:nr, 1]
        else:
            Flat[i, :min(nr, line.size)] = line.ravel()[:nr]
    return r, theta, Flat
```

This function loops over every angle i and retrieves the corresponding `Thk` (`Flat`) data at every r. `Thkmatrix()` (`Flatmatrix()`) takes `Thk` (`Flat`) data from the first (second) column of `line`, a 2D array representing one scan line.

`build_SlotCache()`

Take `wafer_dict` and build `SlotCache`.

```
def build_SlotCache(wafer_dict) -> SlotCache:
    r, theta, Thk = Thkmatrix(wafer_dict)
    _, _, Flat = Flatmatrix(wafer_dict)
    Rmax = finite_max(r, 0.0)
    if theta.size and r.size:
        theta_full = (np.concatenate([theta, theta + np.pi]) % (2*np.pi))
        Thk_full = np.vstack([Thk, Thk[:, ::-1]]) if Thk.size else np.empty((0, 0))
        Flat_full = np.vstack([Flat, Flat[:, ::-1]]) if Flat.size else np.empty((0, 0))
        T, Rm = np.meshgrid(theta_full, r, indexing='ij')
        X_mir = Rm*np.cos(T)
        Y_mir = Rm*np.sin(T)
    else:
        Thk_full = np.empty((0, 0))
        Flat_full = np.empty((0, 0))
        X_mir = np.empty((0, 0))
        Y_mir = np.empty((0, 0))
    return SlotCache(
        r=r, theta=theta, Thk=Thk, Flat=Flat,
        Rmax=Rmax, X_mir=X_mir, Y_mir=Y_mir, Thk_mir=Thk_full, Flat_mir=Flat_full)
```

This function uses the following polar-coordinate identity:

$$(r, \theta) \equiv (-r, \theta + \pi)$$

`theta_full = (np.concatenate([theta, theta + np.pi]) % (2*np.pi))` extends `theta` by mirroring it across the wafer `theta + np.pi` while `% (2*np.pi)` ensures that angles stay in the range $[0, 2\pi)$. Then, `Thk_full = np.vstack([Thk, Thk[:, ::-1]]) if Thk.size` stacks the original (+ r) array and the mirrored (- r) array vertically (`::-1` reverses the sequence). This way, the mirrored rows are stacked under the original rows to form a full 0 - 360° matrix.

Plot Utilities

`robust_clip()`

Clip values based on `p_lo` (lowest percentile) and `p_hi` (highest percentile) to reduce the effect of a notch (outlier values) skewing colors.

```
def robust_clip(Z: np.ndarray, p_lo: float, p_hi: float):
    Zf = Z[np.isfinite(Z)]
    if Zf.size == 0:
        return Z, 0.0, 1.0
    vmin = float(np.nanpercentile(Zf, p_lo))
```

```
    vmax = float(np.nanpercentile(Zf, p_hi))
    if not np.isfinite(vmin): vmin = 0.0
    if not np.isfinite(vmax): vmax = vmin + 1.0
    if vmin >= vmax:
        vmax = vmin + 1e-9
    return np.clip(Z, vmin, vmax), vmin, vmax
```

The function limits the input array `Z` to values within the range (`vmin`, `vmax`). `vmin = float(np.nanpercentile(Zf, p_lo))` and `vmax = float(np.nanpercentile(Zf, p_hi))` computes the lowest percentile and the highest percentile of the data, respectively, from `p_lo` and `p_hi` , both of which are set by the user through color clip sliders (`Color clip low (%)` / `Color clip high (%)`) in the sidebar.

```
with st.sidebar:
    st.markdown("**Display controls**")
    p_lo = st.slider("Color clip low (%)", 0.0, 5.0, 0.5, 0.5)
    p_hi = st.slider("Color clip high (%)", 95.0, 100.0, 100.0, 1.0)
    if p_hi <= p_lo:
        p_hi = min(100.0, p_lo + 0.5)
```

By default, `p_lo` is set to 0.5 and `p_hi` is set to 100—only values below the 0.5th percentile are clipped.

`masknotch()`

Mask notch (outlier values) in the array using Median Absolute Deviation (MAD).

```
def masknotch(Z: np.ndarray, k: float=4):
    Zm = np.asarray(Z, dtype=float).copy()
    m = np.isfinite(Zm)
    if not m.any():
        return Zm
    Zf = Zm[m]
    med = float(np.nanmedian(Zf))
    mad = float(np.nanmedian(np.abs(Zf - med))) * 1.4826
    if mad == 0 or not np.isfinite(mad):
        return Zm
    out = np.abs(Zm - med) > (k * mad)
    Zm[out] = np.nan
    return Zm
```

Any finite values whose absolute distance from the median is greater than $k \times MAD$ are marked as outliers and are replaced with `NaN`. k is the outlier threshold, which has been set to 4. (A lower k is a stricter filter that would mask more data as outliers.) The function returns a copy of the input array with outliers replaced with `NaN`.