ChatGPT

Implementing a Two-Sheet Abel–Jacobi Map Pipeline

Introduction: Two Sheets in the Abel–Jacobi Map

The Abel–Jacobi map sends points on a Riemann surface to points on its Jacobian by integrating a basis of g holomorphic differentials from a base point to the given point. On a *two-sheeted* Riemann surface (e.g. a hyperelliptic curve given by $y^2=f(x)$), each x-coordinate corresponds to two possible points (sheets) with $y = \pm\sqrt{f(x)}$. Crossing a branch cut causes the value of y (and hence any differential containing y) to flip sign [1]. A **faithful** pipeline must account for this two-sheet structure, whereas a simpler "AJ-lite" version may have assumed a single sheet or ignored sign flips. We will implement two Jupyter notebooks:

- **Notebook 1:** Precompute an integration lookup table handling branch cuts and two sheets (producing integrals I_{plus} and I_{minus} for each point).
- **Notebook 2:** Train a model that uses this two-sheet lookup table. The model will output g learned point coordinates and *sheet selection logits* for each, then sum their integrals (choosing I_{+} or I_{-} per point) to obtain the full Abel–Jacobi map. All other aspects (grid size, network trunk, loss terms) remain consistent with the existing AJ-lite pipeline to enable direct comparisons.

Notebook 1: Two-Sheet Integration Table Generator

1. Grid Setup and Branch Cut Definition

First, define the complex grid of points z where integrals will be computed (same resolution and range as before). Also define the branch cuts in the complex z-plane. Typically, for a two-sheeted Riemann surface, branch cuts connect pairs of branch points (roots of $f(x)$). For example, in a hyperelliptic curve $y^2=f(x)$ with branch points a_i, we might choose straight-line cuts between (a_{2j-1}, a_{2j}) for $j=1\ldots g+1$, or another canonical cut system. Represent each cut as a line segment or curve and have a function to test if a path segment crosses a cut.

2. Path Integration with Sign Flips

Use the same integration strategy as the one-sheet pipeline, but incorporate sign tracking. For each differential ω_k ($k=0,\dots,g-1$), we will integrate along a continuous path from the base point to the target grid point. We maintain a **sign variable** `sigma` initialized to +1 (assuming the base point lies on the "+" sheet). As we move along the path in small steps (e.g. along grid lines), detect if the line segment crosses any branch cut. Each time a branch cut is crossed, flip the sign: `sigma *= -1`. This models the fact that moving to the other sheet in a two-sheeted surface reverses the sign of $\sqrt{f(x)}$ [1]. Then, for an integration step $t \to t+\Delta t$, accumulate the integral for each differential as:

$$ I_k \mathrel{+}= \sigma \cdot \omega_k(t)\,\Delta t~, $$

where $\omega_k(t)$ is evaluated with the current sheet's sign. By the end of the path, σ indicates on which sheet we have landed (plus if $\sigma=+1$, minus if $\sigma=-1$).

3. Computing $I_{\mathbf{+}}$ and $I_{\mathbf{-}}$ for Each Point

We need two versions of the integral for each grid point z: - **$I_{\text{plus}}(z)$:** the g-dimensional integral $\int_{\text{base}}^z \boldsymbol{\omega}$ where the path is chosen (or deformed) such that the endpoint z is reached on the **+ sheet**. In practice, you can define a canonical path that yields an even number of branch-cut crossings (net $\sigma=+1$ at the end). Use the above stepwise integration with sign flips to compute $I_{\text{plus},k}(z)$ for each differential. - **$I_{\text{minus}}(z)$:** the integrals for arriving at the **– sheet** of the same point. We can use the *same physical path* in the x-plane but include one extra sign flip at the very end to simulate switching sheets. In implementation, a simple way is to compute $I_{\text{minus}}(z)$ as if you crossed one additional branch cut at z. For example, after obtaining $I_{\text{plus}}(z)$, multiply it by –1: $$I_{\text{minus},k}(z) = -\,I_{\text{plus},k}(z)$$ for each k, if all differentials change sign upon sheet flip (as is true for standard hyperelliptic differentials like $x^m dx/y$ which contain one power of y). This corresponds to a tiny loop around z crossing onto the other sheet without further x displacement. More generally, you could explicitly integrate a final infinitesimal segment crossing a cut at z. Both methods ensure that I_{minus} represents the integrals landing on the opposite sheet relative to I_{plus}.

> **Note:** By construction, $I_{\text{plus}}(z)$ and $I_{\text{minus}}(z)$ may differ essentially by a sign for each differential. On a two-sheeted surface, crossing to the other sheet flips $w = \sqrt{f(x)}$ to $-w$ [1], which flips the sign of each holomorphic 1-form that contains an odd power of w (true for the chosen basis). Thus, storing both versions is crucial for fidelity.

4. Iterative Integration Across the Grid

Organize the integration to reuse results and ensure consistency: - Start from the base point (where all integrals are zero by definition). The base should be a point on the + sheet (assign it as $(\text{base}_x, +)$). If the base lies on a grid boundary or a branch point, handle carefully (the base integrals vector is zero for plus sheet; minus sheet at base would be the negative of that, i.e. also zero). - Traverse the grid in a systematic order (e.g. row by row or using a breadth-first flood fill from the base) so that when computing integrals at a new point, you have a previously computed neighbor as a starting reference. At each small step between adjacent grid points: - Begin with the integral values of the neighbor as a starting point. - Integrate the differential over the small step (e.g. using Simpson's rule or a simple trapezoidal rule if step size is small). Use `sigma` flips if the step crosses a branch cut. - Add the result to the neighbor's integral to get the new point's integral. This way, errors don't accumulate significantly and you leverage continuity. - Track `sigma` dynamically for the path between each pair of neighboring points. You might need to reset `sigma` for each new integration from a neighbor (starting from that neighbor's ending sheet state). Essentially, you carry over the sheet state along a path as you propagate outward.

By following grid connectivity, you ensure each point's integrals are computed consistently relative to the base. If the grid is simply connected aside from cuts, any two paths to the same point differ by cycles around branch cuts (which correspond to period lattice vectors; since we are constructing a single-valued Abel–Jacobi map, we stick to one fixed cut system and path homotopy class).

5. Storing Results in a Tensor

For each grid point (identified by indices $[i,j]$ for its coordinate in the grid), store both versions of the integrals. The output can be a NumPy array (or PyTorch tensor, etc.) of shape **$(g,\;2,\;H,\;W)$**: - Index 0 of the second dimension for the "+" sheet integrals, index 1 for the "–" sheet integrals. - For example, `table[k, 0, i, j] = I_{\text{plus},k}(z_{ij})` and `table[k, 1, i, j] = I_{\text{minus},k}(z_{ij})`.

All grid points (i,j) thus have an entry for each differential k on both sheets. This doubles the storage compared to the one-sheet table, but $2 \times H \times W$ is manageable and necessary.

6. Making the Computation Resume-Safe

Compute the table in chunks or scanlines to allow resuming if interrupted (important for large grids or long integrations): - For example, iterate over rows: after finishing each row (or a set of rows), save the partial table to disk (e.g. in an HDF5 or NumPy `.npy` file). Maintain a checkpoint of the last completed index. - If the notebook stops or crashes, you can reload the saved table and resume from the next unfinished point instead of recomputing from scratch. - Use the same file structure or variable names as the previous integral notebooks so that the integration routine and saving/loading logic are familiar. For instance, if the old code used `integrals[h][w]` or similar, extend it to `integrals[sheet][h][w]` or an extra dimension for sheet. - Ensure determinism: always traverse in the same order and handle branch cuts consistently, so that resuming yields the same result as a single pass.

By mirroring the structure of prior notebooks (function organization, logging, checkpointing), it will be easier to integrate this code into the existing pipeline. After this step, you will have a lookup table that, given a point's approximate location and a choice of sheet, returns the g-component Abel–Jacobi integral from the base to that point.

Notebook 2: Training the Model with Two-Sheet Lookup

1. Loading the 2-Sheet Abel–Jacobi Table

Load the precomputed table from Notebook 1 (e.g. from file). This could be stored as a constant tensor in memory for quick access during training. In a PyTorch or TensorFlow model, you might load it as a buffer or use an interpolation layer. The table provides a function $F(x, y, \text{sheet}) \approx \int_{\text{base}}^{(x,y, \text{sheet})}\boldsymbol{\omega}$.

If the model's input domain for points is continuous (not restricted to grid indices), implement a **differentiable interpolation** of the table: - Use bilinear or bicubic interpolation on the $(H \times W)$ grid for each sheet. For example, given a predicted continuous coordinate (u,v) (in normalized grid units) and a sheet choice s, interpolate the g integrals:
$$I_{\text{interp},k}(u,v,s) = \text{bilinear_interp}\big(table[k, s, \cdot,\cdot], (u,v)\big).$$ - This yields g values smoothly depending on (u,v) and allows gradients to flow through the coordinate predictions. The interpolation should be the same method used in the original AJ-lite pipeline (for consistency and fairness in comparison). - Confirm that the interpolation handles boundaries/extrapolation the same way as before (e.g. if a predicted point falls slightly outside the precomputed range, you might clamp it or apply a penalty – see below).

2. Model Outputs: Coordinates and Sheet Logits

Design the model's output heads to produce, for each of the g points in the divisor: - **Point coordinates:** likely two values (e.g. x and y positions on the grid, or real and imaginary parts of z) for each point. This could be $g\times2$ numbers output. Ensure the range/scale of these outputs matches the grid coordinate system (you may need to apply a sigmoid or tanh followed by scaling to the grid limits, as done in the original pipeline). - **Sheet logits:** one logit or score for each point indicating which sheet it should be on. We can use a single scalar that will be transformed into a probability of being on the "minus" sheet. For example, let l_j be the logit for point j: - Compute a sheet probability $p_j = \sigma(l_j)$ via a sigmoid (or use a two-way softmax if you prefer to explicitly model [plus, minus] probabilities). - Interpret $p_j \approx 1$ as choosing the minus sheet and $p_j \approx 0$ as the plus sheet. During inference you could take $s_j = \text{round}(p_j)$, but during training we **keep it continuous** to allow gradient-based learning.

This setup mirrors a typical classification layer (one binary classification per point), and it's an extension of the AJ-lite pipeline which likely only had coordinate outputs. The trunk network (shared backbone) remains unchanged; we simply add these extra outputs.

3. Forward Pass: Selecting I_{+} or I_{-} via Sheet Weights

In the forward pass, the model will use the predicted coordinates and sheet logits to assemble the Abel–Jacobi map prediction: 1. **Lookup integrals for each point:** For each point j (where $j=1,\dots,g$): - Obtain its predicted coordinate (u_j, v_j) (continuous indices or scaled position in the grid). - Compute the *plus* and *minus* integrals at that location by interpolating the lookup table: get vectors $\mathbf{I}*{j}^{(+)} = (I}^{+}, \dots, I_{j,g}^{+})$ and $\mathbf{I*{j}^{(-)} = (I)$ from }^{-}, \dots, I_{j,g}^{- `table[:, 0, :, :]` and `table[:, 1, :, :]` respectively. - Use the sheet probability p_j to mix these: $$\mathbf{I}*{j}^{\text{(selected)}} = (1 - p_j)\,\mathbf{I}}^{(+)} \;+\; p_j\,\mathbf{I*{j}^{(-)}.$$ This effectively chooses I for that point. If p_j is 0 or 1, it picks one sheet exactly; if it's in between, the model is effectively considering a superposition of sheets (which can be useful for gradient flow early in training). As training progresses, we expect p_j to push toward 0 or 1 for a clear sheet assignment. - This operation is differentiable. Gradients will flow into p_j (encouraging it to choose whichever sheet leads to lower loss) and into the coordinate (u_j,v_j) (moving the point to reduce error).}$ or $I_{-

1. **Sum the integrals:** Add up the contributions of all g points to form the total Abel–Jacobi map output: $$\mathbf{I}*{\text{total}} \;=\; \sum}^g \mathbf{I*{j}^{\text{(selected)}}.$$ Here each $\mathbf{I}$).}$ is a length-g vector (one entry per differential). The sum is also a g-dimensional vector, which is the predicted Abel–Jacobi coordinate (up to the period lattice). This summation exploits the linearity of integrals: integrating to g points and summing is equivalent to integrating a degree-g divisor (the Abel–Jacobi of $P_1+\cdots+P_g$ is $\sum_i \int_{\text{base}}^{P_i} \boldsymbol{\omega

2. **Output interpretation:** $\mathbf{I}*{\text{total}}$ can be compared to the target Abel–Jacobi coordinates. If the task is Jacobi inversion (recovering divisor points from a given AJ target), the network's prediction $\mathbf{I}$ should match the input target vector (with appropriate modulus if comparing in the fundamental domain). If the task is forward mapping (predicting AJ given points), the outputs can be directly compared to ground truth computed by classical means. In either case, the model now correctly handles the two-sheet ambiguity for each point.}

4. Maintaining Training Structure and Losses

To ensure a fair comparison with the prior AJ-lite pipeline, **do not change** aspects like the neural network trunk architecture, optimizer, or basic loss definitions except as needed for the new outputs: - **Primary loss:** Continue to use the same loss for the Abel–Jacobi map prediction as before (e.g. mean squared error between $\mathbf{I}_{\text{total}}$ and the true AJ coordinates, or another appropriate distance on the torus). The introduction of two-sheet handling doesn't change how the final output is evaluated; it simply improves accuracy for cases where points were on the minus sheet. - **Regularization on sheets:** If we have knowledge of the correct sheet choices for training data, we could add a supervised classification loss (like a binary cross-entropy for each p_j). However, in an unsupervised inversion scenario, typically we don't know the sheet a priori. Instead, we can encourage *discrete* sheet decisions by adding a small penalty for p_j not being near 0 or 1. For instance, add a term $\lambda \sum_j 4\,p_j(1-p_j)$ to the loss (this is zero when p_j is 0 or 1, and maximal at $p_j=0.5$). This will gently push the network to make a clear choice for each sheet, preventing it from forever using fractional values. The coefficient λ can be small so as not to overwhelm the main loss. - **Coordinate penalties:** Keep any existing penalties on the point coordinates. In the AJ-lite pipeline, there might have been constraints to stabilize training, such as: - **Bound enforcement:** If a predicted (u_j,v_j) goes outside the precomputed grid, you might clamp it or add a loss term to push it inside. Maintain the same strategy here. - **Ordering or symmetry breaking:** Often, the g points in a divisor can be permuted without changing the sum. To avoid redundant solutions, the original pipeline might have sorted points by, say, increasing real part of x, or added a loss to penalize out-of-order points. Ensure the same mechanism is applied so that the new model doesn't suffer from permutation instability. If previously points were sorted by construction, do the same now. If not, maybe a small penalty for $u_1 < u_2 < \cdots < u_g$ violations was used – keep it in place. - **Collision penalty:** If two predicted points coinciding was a concern (which could lead to singular behavior in inversion), a penalty to discourage $|(u_i - u_j, v_i - v_j)|$ being too small might exist. Continue using it if so. - **Training procedure:** Use the same learning rate schedule, batch size, etc. The new parameters (sheet logits) will be learned along with coordinates. It may help to initialize sheet logits to a neutral value (e.g. 0 logits => $p_j=0.5$) so the model starts with no strong sheet preference and can explore both. This is analogous to the AJ-lite start (which effectively assumed plus sheet for all – now we let the network decide).

By preserving these aspects, the training dynamics remain comparable. The difference is that the model now has the capacity to represent solutions involving points on the second sheet, which the AJ-lite pipeline could not.

5. Verification and Comparison

During training and after, verify that the new pipeline matches the old one's behavior in the limit where sheet logits choose the plus sheet for all points. In cases where the AJ-lite was valid (no sheet flips needed), the two pipelines should produce nearly identical results (aside from minor differences due to interpolation or regularization). Then test scenarios that *require* minus-sheet contributions (for example, where the correct divisor has one or more points on the second sheet). The new model should be able to fit those, whereas the old model would have shown a systematic error. By keeping everything else identical, any improvement in accuracy can be attributed to the faithful two-sheet treatment.

Conclusion

We implemented a two-sheet Abel–Jacobi map pipeline with minimal changes to the existing framework: Notebook 1 builds a $(g,2,H,W)$ integration table accounting for branch cut sign flips (so each point has integrals for both sheets), and Notebook 2 uses that table in a neural model that learns not only the point coordinates but also which sheet each point lies on. This yields a **faithful Abel–Jacobi map** evaluation, properly reflecting the double-sheeted Riemann surface structure while remaining compatible with the original pipeline's architecture and hyperparameters. With this in place, the model can be trained and its results directly compared to the AJ-lite pipeline, demonstrating the impact of accurately handling the two-sheet structure.

Sources:

- Teleman, C. *Riemann Surfaces* – Example of a two-sheeted surface (hyperelliptic curve) where crossing a branch cut flips the sign of w [1], illustrating the need for sign flips in integration.

---

[1] math.berkeley.edu
https://math.berkeley.edu/~teleman/math/Riemann.pdf