ChatGPT

MediaPipe 3D座標からVRM Humanoidボーンへの変換ガイド

ダンス動画などでMediaPipeの3Dポーズ推定結果をVRM形式のHumanoidアバターに適用するための手順を説明します。ここでは、**MediaPipe Poseの3Dワールド座標**（各フレームごとのXYZ位置）から、VRM（glTF Humanoid）形式の**17個の主要ボーン**（hips, spine, chest, neck, head, 両腕・両脚の各関節）を正確に制御する方法を解説します。Pythonスクリプトによるオフライン処理を前提とし、実装者がそのまま使える実用的な技術手順を示します。

座標系の違いと変換方法

MediaPipeのワールド座標系と**VRM(glTF)の座標系**にはいくつかの相違があります。まずMediaPipe Poseの3Dワールドランドマークは、**原点が左右の腰の中心**（ヒップの中央）にあり、単位はメートルです [1]。MediaPipeの座標軸はカメラ視点に合わせており、一般に以下のように設定されています（デフォルトでは右手系）：

- **X軸**: 画像の横方向。カメラから見て右方向が正（人物から見れば左右逆になる点に注意）。
- **Y軸**: 垂直方向。上方向が正（実世界の重力に逆らう方向）。
- **Z軸**: 奥行き方向。原点（腰）から見た奥行きで、**カメラに近づく方向で負の値**になります（つまり値が小さいほどカメラに近い） [2]。

一方、**VRM（glTF）の座標系**は3D汎用フォーマットのものです。glTFでは**右手系**で**+Yが上方向**、**+Zが前方向**（モデルの正面）で定義されています [3]。また「-Xが右方向」とされ、モデルの正面が+Z軸を向くよう規定されています [3]。この違いにより、MediaPipeの出力をそのままVRMアバターに適用すると、軸の向きがずれて不正確なポーズになります。

座標系の合わせ方: MediaPipe座標をVRM(glTF)座標に変換するには軸の符号や並進を調整します。一般的な変換は次の通りです：

- **原点の一致**: MediaPipeの原点は腰中心ですが、VRMモデルのhipsボーンも通常は骨盤付近が原点となります。あらかじめVRMモデルをTポーズの初期姿勢にしておき、MediaPipeの原点に相当する位置（腰の中心）がVRMのhipsに重なるようにスケール・位置合わせを行います（多くの場合スケール1単位=1メートルで問題ありません）。
- **軸の反転**: カメラ視点とモデル視点の違いから、**X軸とZ軸を反転**する必要があります。具体的には、MediaPipe座標 `[x, y, z]` を VRM座標に取り込む際に `x' = -x`、`z' = -z` とします。これにより、**カメラ座標系での右向き**を**モデル座標系での左向き**に、**カメラに向かう方向**を**モデル前方(+Z)方向**に変換できます。Y軸（上下）は両者で上方向が一致しているためそのままで構いません。
- **左右軸の確認**: VRMはUnity Humanoid互換でありUnity内部は左手系ですが、VRM自体(=glTF)は右手系です [4]。一般には上記のX,Z反転で正しく対応できますが、モーションが左右逆に見える場合は軸の取り違えを疑いましょう（Unityで動かす場合はさらに変換を挟む必要がありますが、本手順ではVRM(glTF)ファイル自体を直接操作します）。

以上の変換を各フレームの全てのランドマーク座標に適用し、以降の計算ではVRMと同じ座標系上で骨の向きを計算します。

ボーン毎の回転推定方法

各ボーン（関節）の姿勢（回転）を求めるには、MediaPipeの各ランドマークの位置から**ボーン方向ベクトル**を算出し、それを基にボーンの**回転**を推定します。基本的なアプローチは「**2ベクトル法**」と呼ばれるものです。これは、あるボーンについて**2つの方向ベクトル**（主方向と副方向）を定義し、それらをもとにボーンの3次元的な向きを決定する方法です [5]。以下に各ボーンの回転計算手順を説明します：

1. **主方向ベクトル（Bone方向）**：ボーンを構成する親関節と子関節の位置から、**ボーンの向き**を表すベクトルを計算します。例えば「左上腕 (Left UpperArm)」ボーンなら、`v1 = 肩関節(L_shoulder)の位置 → 肘関節(L_elbow)の位置` のベクトルを取ります。この正規化ベクトルがボーンの軸方向になります。ボーン軸は通常、そのボーンのローカル座標系で特定の軸（例えばローカルZ軸）に対応させます。

2. **副方向ベクトル（平面方向）**：ボーンのねじれ（ロール回転）を決めるために、もう1本のベクトルを定義します。これは主方向ベクトルと組み合わせて**ボーンの回転平面**を決めるものです。一般的には、主方向と隣接する別のランドマークとのベクトルを使います。例えば上腕であれば、「肩→肘」のベクトルに加え、「肩→手首」あるいは「肘→手首」のベクトルを用いて平面を定義します。具体的には、左上腕の場合は `v2 = 肘(L_elbow)の位置 → 手首(L_wrist)の位置` のベクトルを考えます。肩・肘・手首が作る平面上に上腕が動くため、この平面をボーンの基準面にします。

3. **外積による軸の算出**: 上記の2つのベクトルからボーンのローカル座標系の3軸を求めます。まず **主軸**を単位ベクトル化し、これを仮にボーンローカルのZ軸（前後方向）とします。次に、主軸と副ベクトルの外積（クロスプロダクト）を取り、その方向をローカルのY軸（あるいはX軸）に割り当てます [6] [5]。最後に、その2つの軸の外積を再度計算して残るX軸（またはY軸）を得ます。こうして3本の互いに直交する単位ベクトル（**ボーンの基底**）が決まります。この基底を並べたものがボーンの**回転行列**となります。例えば:

4. `z_axis = normalize(v1)`（ボーン軸方向）
5. `x_temp = normalize(v2)`（第二の方向を一旦正規化）
6. `y_axis = normalize(z_axis × x_temp)`（主軸と副方向の外積で垂直方向を定義）
7. `x_axis = normalize(y_axis × z_axis)`（直交系を完成させるため残りを計算）
これにより、**x_axis, y_axis, z_axis**がボーンのローカル空間における各軸のワールド方向を示します。この3軸を列（または行）に持つ3x3行列がボーンの回転行列になります。

8. **特殊ケースの処理**: 2つのベクトルが**平行またはほぼ一直線上**に並んでいる場合、外積がゼロベクトルになり軸が定まりません（自由にねじれ得る状態、例えば腕が完全に伸びきって肘と手首が一直線になる場合）。この場合の**安定化対策**として、次のような処置をします [7]:

9. 平面を定義する副ベクトルを別の基準に変える。例えば腕が伸びきっているなら、手首ではなく**体幹側の基準**を使う（肩〜腰の方向など）か、**固定の世界軸**（例: ワールドの上方向 (0,1,0) など）を代わりに使用

します。ボーン軸と固定軸の外積で平面を定義すれば、とりあえずねじれを0度と仮定した姿勢が得られます。

10. 直前フレームの回転を参考に補正する。オフラインで連続フレームを扱う場合、前後のフレームで回転の連続性を保つようにし、突然の反転を防ぎます。例えば前フレームのボーンのロール角を維持するよう補間することで、不安定な姿勢変化をなだらかにします。

11. 外積計算時の**向きの一貫性**にも注意します。外積は2つのベクトルの順序で向きが反転します。骨格ごとに**右手系/左手系**のどちらで基底を構築するかを統一し、期待する軸の向きと逆になった場合は軸を反転するよう調整します [8]。例えば、ボーンローカル座標系が右手系になるように、適宜外積の順序や符号を揃えます。これによって、すべてのボーンで一貫した回転軸の取り方が保証され、モーション中に不自然に骨がひっくり返るのを防ぎます。

12. **各ボーンごとの具体例**:　主なボーンについて、使用する2ベクトルの組み合わせと計算方法の例を挙げます。

13. **Hips（腰）**：原点かつルートとなるHipsボーンは、全身の向きと傾きを決めます。MediaPipeでは左右の腰（hip）ランドマークが得られるため、まず**左右の腰を結ぶベクトル**（左→右）を計算しこれを骨盤の水平方向とします。また**腰中心→上半身中心**（腰中央→首あるいは胸の中心）のベクトルを計算し、これを垂直方向の基準とします。この2つから腰の回転平面を定義します。具体的には、左右腰を結ぶ方向をローカルX軸、腰から上半身への方向をローカルY軸（上方向）に設定し、その外積でローカルZ軸（前方向＝正面方向）を求めます [6]。こうすることで、腰（hips）のボーンは人物の骨盤の傾き・向きに一致します。

14. **Spine/Chest（背骨・胸）**: 背骨（spine）や胸（chest）ボーンは胴体の姿勢を表します。MediaPipeでは脊柱に相当する明確なランドマークはありませんが、肩や腰の位置から大まかな姿勢を推定できます。例えば**腰中心→首**のベクトルを胴体の軸とし、**左右肩を結ぶベクトル**を胴体の横方向基準とします。この2つから胴体の回転（前傾・側傾・捻り）を計算します。Spineボーン（下部背骨）は腰に近く胸に繋がるので、腰→胸の方向と左右腰の方向を使うなどします。Chestボーン（上部胸郭）は胸→首の方向と左右肩の方向で計算すると良いでしょう。いずれも胴体の捻じれ角は肩と腰の向きの差から推定できます。
15. **UpperArm（上腕）**: 上述のように**肩→肘**を主軸、**肘→手首**を副軸として計算します。これにより腕が曲がる面を特定し、上腕の回転（肩関節の屈伸とひねり）を求めます。2ベクトルがほぼ直線になる（腕が伸びきる）場合は、副軸として例えば**肩→中指先**方向や**肩→反対肩の方向**など、代替の参照を用いると安定します。
16. **LowerArm（前腕）**: 前腕（肘〜手首）は、**肘→手首**を主軸とし、**手首→中指先**（またはMediaPipeのINDEX指先ランドマーク等）を副軸にすると、手首の向き（回内外）まで推定できます。ただしMediaPipe Pose単体では指先の向き情報が乏しいため、Holisticの手ランドマークも使うか、前腕のロールはゼロに保つ（手のひらが常に下を向く等仮定）こともあります。精密な手首の回転には手のランドマーク21点から推定される**手首座標系**を使うのがお勧めです。
17. **UpperLeg（大腿）**: **腰→膝**を主軸、**膝→足首**を副軸として計算します。膝が曲がる面（脚の屈曲面）を定義するため、副軸として**左右の膝の方向**や**腰→膝→足首の平面の法線**を使います。膝が完全に伸びきった場合は不安定になるので、その際は脚のひねりは前フレームから補間するか0に固定します。
18. **LowerLeg（下腿）**: **膝→足首**を主軸、**足首→つま先**方向を副軸として、足首（足関節）の回転を計算します。MediaPipe Poseにはつま先（foot index）や踵（heel）のランドマークがあるため、例えば**足首→つま先**ベクトルが足先の向き（左右方向）を示します。これと下腿軸を用いれば足首の回内外や屈伸を求められます。

19. **Foot（足）**: VRMの足ボーン（foot）は足首の向きを指します。下腿と似ていますが、必要に応じて**踵→つま先**の方向を加味し、足裏が地面と接地する角度を算出します。ダンスでは足先の接地は重要なので、床面が既知の場合は足裏が床と平行になるよう補正するテクニックもあります。
20. **Neck/Head（首・頭）**: 首は胸→頭部の方向を軸とし、左右肩を結ぶ方向を基準に回転を求めます。頭部についてはMediaPipe Poseでは正面向きかどうかの情報が不足しますが、Holisticの**顔ランドマーク**（鼻先や耳の位置など）を利用すれば細かい頭の回転（うなずき・首振り・傾き）が得られます。例えば**首→鼻**のベクトルを頭の前方向の軸とし、**左耳→右耳**のベクトルを頭の横軸として計算する方法があります。顔の向きも骨の回転としてQuaternionに反映させます（VRMでは首ボーンと頭ボーンの2箇所で分担）。Holisticのデータがない場合、鼻先と首の位置関係や肩の向きから大まかに頭の向きを推定することになります。

以上のように、各ボーンごとに適切な2つのランドマーク間ベクトルを選び、それらからボーンの**回転行列**（もしくは軸と角度）を算出します。この際、MediaPipeのボーン長とVRMモデルのボーン長が異なることがありますが、回転の計算には影響しません。必要であればボーン長はスケールで調整できますが、Humanoidではスケールより**ポーズ（回転）**を合わせることが重視されます。

回転行列からクォータニオンへの変換

各ボーンの回転行列が求まれば、それをVRMに適用する**クォータニオン**（Quaternion）に変換します。クォータニオンは回転のズレ（ジンバルロック）を防ぎスムーズに補間できる表現です。変換手順と留意点は以下の通りです：

- **回転行列の作成**: 前節の手順で得たボーンの基底（三つの単位ベクトル）から3x3回転行列を構成します。例えば、`R = [x_axis, y_axis, z_axis]`を列ベクトルに持つ行列（または行ベクトルに持つ転置行列）とします。ここでx_axis等はワールド座標系での方向ベクトルです。この行列はボーンの**ワールド空間での向き**を表しています。

- **Quaternionへの変換**: 回転行列をクォータニオンに変換するには、直接数式を使う方法とライブラリを使う方法があります。ライブラリを使う場合、Pythonでは`scipy.spatial.transform.Rotation`などで`R.from_matrix(matrix).as_quat()`とすることで`[x, y, z, w]`形式のクォータニオンが得られます。この順序はglTFが要求する形式（xyzw）に合致します。独自に実装する場合は、**軸-角度**から求める方法や**直接式**があります。効率的かつ安定な求め方として、**2ベクトルから直接クォータニオンを計算する公式**を利用することもできます [9]。例えば単位ベクトルuをvに回すクォータニオンは次式で与えられます：

$$q = \Big(1 + u \cdot v,\; u \times v\Big)$$

（必要なら正規化） [10]。上式は四元数$(w, \mathbf{xyz})$の形式で表しています。実装上は、スカラー部$w = 1 + \mathbf{\hat{u}}\cdot\mathbf{\hat{v}}$、ベクトル部$\mathbf{xyz} = \mathbf{\hat{u}} \times \mathbf{\hat{v}}$として計算し、最後に正規化します [10]。実際には全身の回転行列からも同様に計算できますが、この公式は**2つのベクトルの回転**には高速で安定です。全身の回転行列から変換する場合も、数式自体は単純なので直接実装可能です。

- **数値安定性と正規化**: 回転行列→クォータニオン変換では、浮動小数点誤差によりクォータニオンの長さが1から僅かにズレることがあります。これを放置するとアニメーション中にスケーリングのような不具

合を招くため、**必ず正規化（ノルムを1にする）**します[11]。また、クォータニオンは符号反転しても同じ回転を表すため、フレーム間で符号がコロコロ変わると連続性に問題が出ます。そこで**クォータニオンの符号を統一**する工夫も有用です。一般には「隣接フレームでw成分が負になったら全体を-1倍する」あるいは「前フレームとの内積が負なら反転する」などで符号の連続性を保ちます。これにより、補間時に大きくジャンプしない滑らかな回転になります。

- **ジンバルロック回避**: クォータニオンはオイラー角と違いジンバルロックの問題を直接起こしませんが、計算過程でオイラー角を使わないようにすることがポイントです。可能な限り直接回転行列やベクトルから計算し、XYZオイラー角を介さないことで安定した回転が得られます。どうしてもオイラー角を使う場合は、順序（例えば`XYZ`順回りなど）に注意し、特定の角度で失われる自由度が出ないようにします。

ボーン階層とローカル回転の算出

上述の手順で求めた各ボーンの回転（クォータニオン）は、**ワールド空間**での回転として導出されています。しかし、VRM（Humanoid）のボーン変換は階層構造になっており、各ボーンは**親ボーンに対するローカル回転**で指定する必要があります。そこで、求めたワールド回転から親子関係を考慮してローカル回転に変換します。

手順としては、**親ボーンのワールド回転の逆（逆クォータニオン）を右から掛ける**ことで子ボーンのローカル回転を求めます[12]。具体的には、親ボーンのクォータニオンをq_p、子ボーンのワールド回転クォータニオンをq_cとすると、子のローカル回転q_{local}は以下で得られます:

$$q_{local} = q_p^{-1} * q_c$$

ここでq_p^{-1}は親クォータニオンの逆（共役）で、単位クォータニオンの場合は成分を反転するだけです（(x,y,z,w) → $(-x,-y,-z,w)$）。演算$*$はクォータニオンのハミルトン積です。注意として、*glTF*におけるノードの変換は親から子へ適用されるため、上記のように親の逆を左から掛ける*（もしくは右掛け、クォータニオンの表現方法によりますが、ここでは(x,y,z,w)形式を前提に左側から積を取るとします）ことで子のローカル表現に変換できます。例えば、hipsの回転がq_{hips}、spineのワールド回転がq_{spine}なら、spineノードに格納すべきローカル回転は $q_{hips}^{-1} * q_{spine}$ となります。同様にchest（胸）のローカル回転は $q_{spine}^{-1} * q_{chest}$、首 (neck) は $q_{chest}^{-1} * q_{neck}$、頭 (head) は $q_{neck}^{-1} * q_{head}$ といった具合です。四肢も同様で、上腕は肩（あるいは胸）の逆を掛け、前腕は上腕の逆、手は前腕の逆、大腿はhipsの逆、下腿は大腿の逆、足は下腿の逆…となります。

Tips: クォータニオンの積順序は実装言語やライブラリで異なることがあります。一般には`q_total = q_parent * q_local`で子のワールド回転となる定義（右乗算）を採用していれば、上記のように逆を左から掛ける式になります。誤って逆順にしてしまうとローカル回転が逆になってしまうため、テスト時には単一ボーンで簡単な回転を与えて検証するとよいでしょう。例えば、hipsを90度回した状態で子ボーンを0にした場合に、出力で子が90度回っていなければ順序がおかしいことが分かります。

親子関係に従った計算の順序: 実装上は、ルートであるhipsボーンのクォータニオンはそれ自体がローカル回転になります（親がいないため）。それ以外のボーンは、親のワールド回転が計算済みである必要があります。そこで、あらかじめ**ボーン階層順**（例えばhips→spine→chest→neck→head、およびhips→upperLeg→lowerLeg→foot、chest→shoulder→upperArm→lowerArm→handの順）に各ボーンのワールド回転を計算しておきます。その上でルート以外について上記の式でローカル回転に変換します。階層順に計

算することで、親の姿勢に引きずられた子の回転を正しくローカル表現にできます。もし順序が前後すると、未計算の親を逆に使ってしまうことになるので注意してください。

Python実装例（擬似コード）

上記の理論に基づいた実装の流れを、Python風の疑似コードで示します。これにより具体的なイメージをつかめます。必要に応じてnumpyなどの数値ライブラリを利用するとよいでしょう。

```
import numpy as np

# 単位ベクトルへの正規化関数
def normalize(v):
    norm = np.linalg.norm(v)
    return v / norm if norm != 0 else v

# 2つのベクトルから回転行列を求める（ボーンの基底を計算）
def compute_rotation_matrix(v1, v2):
    z_axis = normalize(v1)                          # 主軸（ボーン方向）
    # 副軸が主軸と平行に近い場合の処理
    x_temp = normalize(v2)
    if np.linalg.norm(np.cross(z_axis, x_temp)) < 1e-6:
        # 平行に近い：別の基準ベクトルを使用（例：ワールドY軸）
        if abs(z_axis[1]) < 0.9:
            # z軸がYと十分異なるならY軸を使用
            x_temp = np.array([0.0, 1.0, 0.0])
        else:
            # z軸がY軸とほぼ平行ならZ軸を使用
            x_temp = np.array([0.0, 0.0, 1.0])
    # 直交基底を計算
    y_axis = normalize(np.cross(z_axis, x_temp))  # zと副軸の垂直方向（ローカルY）
    x_axis = normalize(np.cross(y_axis, z_axis))  # YとZからローカルX
    # 3x3回転行列を組み立て（列ベクトルがローカル軸）
    R = np.column_stack((x_axis, y_axis, z_axis))
    return R

# クォータニオンの計算：回転行列から求める場合
from scipy.spatial.transform import Rotation as R
def matrix_to_quat(Rmat):
    # scipyは [x, y, z, w] の順でクォータニオンを返す
    quat = R.from_matrix(Rmat).as_quat()
    # 正規化（念のため）
    return quat / np.linalg.norm(quat)

# あるいは、2つの方向ベクトルから直接クォータニオンを計算する関数
def quaternion_from_two_vectors(u, v):
```

```
    u = normalize(u)
    v = normalize(v)
    dot = np.dot(u, v)
    # 真逆方向の場合の処理
    if dot < -0.999999:
        # uに直交する適当な軸を取る
        ortho = np.cross(np.array([1,0,0]), u)
        if np.linalg.norm(ortho) < 1e-6:
            ortho = np.cross(np.array([0,1,0]), u)
        ortho = normalize(ortho)
        # 180度回転（piラジアン）
        return np.concatenate((ortho * 0.0, [ -1.0 ]))  # w=-1は180度（x,y,zは0）
    # 並行な場合（回転なし）
    if dot > 0.999999:
        return np.array([0.0, 0.0, 0.0, 1.0])
    # それ以外の場合
    axis = np.cross(u, v)
    quat = np.concatenate((axis, [1.0 + dot]))
    quat = quat / np.linalg.norm(quat)
    return quat

# メイン処理: MediaPipeランドマーク（world座標）からボーン回転計算
# ※ landmarksは例えば {"LEFT_SHOULDER": [x,y,z], ...} のようなdictとする
def solve_pose_to_vrm(landmarks):
    # まずMediaPipe→VRM座標変換（各座標に x→-x, z→-z を適用）
    mp_to_vrm = lambda p: np.array([-p[0], p[1], -p[2]])
    lm = { name: mp_to_vrm(np.array(coord)) for name, coord in
landmarks.items() }

    # 各ボーンのワールド回転quaternionを計算
    rotations_world = {}

    # Hips（腰）: 左右ヒップ中心の姿勢
    hip_center = (lm["LEFT_HIP"] + lm["RIGHT_HIP"]) / 2.0
    # hipsボーンの向き: 腰->上半身 & 腰の左右
    v1 = lm["SPINE"] - hip_center if "SPINE" in lm else lm["NECK"] - hip_center
    v2 = lm["RIGHT_HIP"] - lm["LEFT_HIP"]
    R_hips = compute_rotation_matrix(v1, v2)
    rotations_world["Hips"] = matrix_to_quat(R_hips)

    # Spine（背骨下部）
    if "SPINE" in lm and "CHEST" in lm:
        v1 = lm["CHEST"] - lm["SPINE"]           # 背骨下->上方向
        v2 = lm["RIGHT_HIP"] - lm["LEFT_HIP"]    # 腰の左右方向（胸の捻り基準）
        R_spine = compute_rotation_matrix(v1, v2)
        rotations_world["Spine"] = matrix_to_quat(R_spine)
    # Chest（胸部）
```

```
    if "CHEST" in lm:
        v1 = lm["NECK"] - lm["CHEST"]            # 胸->首方向
        v2 = lm["RIGHT_SHOULDER"] - lm["LEFT_SHOULDER"]  # 肩の左右方向
        R_chest = compute_rotation_matrix(v1, v2)
        rotations_world["Chest"] = matrix_to_quat(R_chest)

    # Neck（首）
    if "NECK" in lm:
        v1 = lm["HEAD"] - lm["NECK"]             # 首->頭方向
        v2 = lm["RIGHT_SHOULDER"] - lm["LEFT_SHOULDER"]  # 肩左右
        R_neck = compute_rotation_matrix(v1, v2)
        rotations_world["Neck"] = matrix_to_quat(R_neck)

    # Head（頭）: 首->鼻方向と左右耳方向を使う（Holistic前提）
    if "NOSE" in lm and "LEFT_EAR" in lm and "RIGHT_EAR" in lm:
        v1 = lm["NOSE"] - lm["NECK"]             # 頭の前方向近似: 首->鼻
        v2 = lm["RIGHT_EAR"] - lm["LEFT_EAR"]    # 頭部の左右軸
        R_head = compute_rotation_matrix(v1, v2)
        rotations_world["Head"] = matrix_to_quat(R_head)

    # Left UpperArm（左上腕）
    v1 = lm["LEFT_ELBOW"] - lm["LEFT_SHOULDER"]
    v2 = lm["LEFT_WRIST"] - lm["LEFT_ELBOW"]
    R_leftUpperArm = compute_rotation_matrix(v1, v2)
    rotations_world["LeftUpperArm"] = matrix_to_quat(R_leftUpperArm)

    # Left LowerArm（左前腕）
    v1 = lm["LEFT_WRIST"] - lm["LEFT_ELBOW"]
    # 副軸: 手首から中指先方向（Holisticの手ランドマークがあれば使用）
    if "LEFT_INDEX" in lm:
        v2 = lm["LEFT_INDEX"] - lm["LEFT_WRIST"]
    else:
        v2 = np.array([1.0, 0.0, 0.0])  # 仮: とりあえずX軸基準（要調整）
    R_leftLowerArm = compute_rotation_matrix(v1, v2)
    rotations_world["LeftLowerArm"] = matrix_to_quat(R_leftLowerArm)

    # Left Hand（左手）: ここでは省略（手首はLowerArmで表現済み、Handボーンは指先基準で回すかも）

    # Left UpperLeg（左大腿）
    v1 = lm["LEFT_KNEE"] - lm["LEFT_HIP"]
    v2 = lm["LEFT_FOOT"] - lm["LEFT_KNEE"]   # 膝->足首
    R_leftUpperLeg = compute_rotation_matrix(v1, v2)
    rotations_world["LeftUpperLeg"] = matrix_to_quat(R_leftUpperLeg)

    # Left LowerLeg（左下腿）
    v1 = lm["LEFT_ANKLE"] - lm["LEFT_KNEE"]
```

```
    v2 = lm["LEFT_FOOT_INDEX"] - lm["LEFT_ANKLE"] if "LEFT_FOOT_INDEX" in lm
else np.array([1,0,0])
    R_leftLowerLeg = compute_rotation_matrix(v1, v2)
    rotations_world["LeftLowerLeg"] = matrix_to_quat(R_leftLowerLeg)

    # Left Foot（左足）
    if "LEFT_HEEL" in lm and "LEFT_FOOT_INDEX" in lm:
        v1 = lm["LEFT_FOOT_INDEX"] - lm["LEFT_ANKLE"]  # 足の前方軸
        v2 = np.array([0.0, 1.0, 0.0])                 # 上方向（仮に垂直）
        R_leftFoot = compute_rotation_matrix(v1, v2)
        rotations_world["LeftFoot"] = matrix_to_quat(R_leftFoot)

    # 右側（Right）も同様に計算...（省略）
    # rotations_world["RightUpperArm"] = ...
    # rotations_world["RightUpperLeg"] = ...
    # ... etc.

    # ワールド回転からローカル回転に変換
    rotations_local = {}
    rotations_local["Hips"] = rotations_world["Hips"]  # hipsはルートなのでそのまま
    # 以下、親の逆クォータニオンを適用
    if "Spine" in rotations_world:
        q_inv = quat_inv(rotations_world["Hips"])
        rotations_local["Spine"] = quat_mul(q_inv, rotations_world["Spine"])
    if "Chest" in rotations_world:
        q_inv = quat_inv(rotations_world.get("Spine", rotations_world["Hips"]))
        rotations_local["Chest"] = quat_mul(q_inv, rotations_world["Chest"])
    if "Neck" in rotations_world:
        q_inv = quat_inv(rotations_world["Chest"])
        rotations_local["Neck"] = quat_mul(q_inv, rotations_world["Neck"])
    if "Head" in rotations_world:
        q_inv = quat_inv(rotations_world.get("Neck", rotations_world["Chest"]))
        rotations_local["Head"] = quat_mul(q_inv, rotations_world["Head"])
    # 四肢
    q_inv = quat_inv(rotations_world["Chest"])   # 腕の親はChestと仮定（モデルにより
肩ボーンがあるが）
    rotations_local["LeftUpperArm"] = quat_mul(q_inv,
rotations_world["LeftUpperArm"])
    q_inv = quat_inv(rotations_world["LeftUpperArm"])
    rotations_local["LeftLowerArm"] = quat_mul(q_inv,
rotations_world["LeftLowerArm"])
    # 脚（親はHips）
    q_inv = quat_inv(rotations_world["Hips"])
    rotations_local["LeftUpperLeg"] = quat_mul(q_inv,
rotations_world["LeftUpperLeg"])
    q_inv = quat_inv(rotations_world["LeftUpperLeg"])
    rotations_local["LeftLowerLeg"] = quat_mul(q_inv,
```

```
rotations_world["LeftLowerLeg"])
    # ...以下略（Right側も同様に）

    return rotations_local

# クォータニオン演算関数の例（xyzw形式）
def quat_inv(q):
    x,y,z,w = q
    return np.array([-x, -y, -z, w])
def quat_mul(q1, q2):
    # Hamilton積: (x1,y1,z1,w1)*(x2,y2,z2,w2)
    x1,y1,z1,w1 = q1; x2,y2,z2,w2 = q2
    return np.array([
        w1*x2 + x1*w2 + y1*z2 - z1*y2,
        w1*y2 - x1*z2 + y1*w2 + z1*x2,
        w1*z2 + x1*y2 - y1*x2 + z1*w2,
        w1*w2 - x1*x2 - y1*y2 - z1*z2
    ])
```

上記は簡易な擬似コードですが、MediaPipeのランドマーク名や利用するライブラリによって書き換える必要があります。また、17ボーン全てを網羅するにはRight側の処理やHandボーン、肩ボーン（clavicle）がある場合の処理など追加が必要です。各所に検討すべき点（例えば肩ボーンがあるモデルではChest→Shoulder→UpperArmと親子関係が1段深くなる、など）がありますので、モデルのヒエラルキーに応じて修正してください。

VRM/glTFへの出力方法

得られた各ボーンのローカル回転クォータニオンを、実際にVRMファイルに適用してみましょう。VRMはglTF 2.0形式の拡張ですから、基本的には**glTFのノード変換**としてボーン回転を書き込めばOKです。以下に推奨手順とツールを示します。

- **ライブラリの利用**: Python環境でglTF/VRMを扱うには、`pygltflib`等のライブラリが便利です。これを使うと、VRMファイル（拡張子.vrmはglbと同様のバイナリ形式です）を読み込み、ノードツリーを操作できます。
- **ノードへのアクセス**: VRMにはHumanoidボーンの対応関係が`extensions`内に定義されていますが、単純にノード名でボーンを探して更新しても構いません。例えば、多くのVRMではHumanoidボーンのノード名が定型的についている（hips, spine, chest, upperArm.L 等）ことが多いです。`pygltflib`なら、`gltf = GLTF2().load('model.vrm')`で読み込み、`gltf.model.nodes`リストから該当ノードを探します。ノードの`rotation`プロパティがクォータニオン（xyzw）ですので、そこを書き換えます。
- **回転値の適用**: 上述の`rotations_local`辞書（ボーン名→クォータニオン）を使い、VRMの各対応ノードに代入します。例:

```
from pygltflib import GLTF2
gltf = GLTF2().load("input_model.vrm")
# Humanoid拡張からボーン名とノードindexの対応を取得（簡便のため直接探索も可）
node_index_by_name = {node.name: idx for idx, node in
enumerate(gltf.model.nodes)}
for bone_name, quat in rotations_local.items():
    idx = node_index_by_name.get(bone_name)
    if idx is not None:
        # pygltflibではクォータニオンはリスト4要素で指定
        gltf.model.nodes[idx].rotation = [ float(quat[0]), float(quat[1]),
float(quat[2]), float(quat[3]) ]
gltf.save("output_pose.vrm")
```

これで新しいVRMファイル `output_pose.vrm` に各ボーンの回転が書き込まれます。もし**既存のVRMモデルに対してポーズを適用**したい場合、上書き保存せず**アニメーションとして出力**する方法もあります。その場合はglTFの `animations` セクションを利用して各ノードの回転チャネルにキーを打つ必要があります。ただし、それは複雑になるため、静的ポーズであれば上記のように直接ノード回転を書き換える方法が簡単です。

- **Humanoid拡張との対応**:　VRMのHumanoid拡張にはボーンの対応表（humanBone）が含まれており、ノード名が一部モデルごとに異なることもあります。可能ならHumanoid拡張をパースして対応するノードを探すと汎用性が増します。 `gltf.model.extensions["VRM"]["humanoid"]["humanBones"]` 配下に各ボーンのnode番号が記載されています。それらを使って対応付けると安全です。

- **three-vrm等での利用**: PythonでVRMを書き出した後、それをUnityやThree.jsで読み込んでモーション再生するケースが多いでしょう。Three.js環境では公式の `three-vrm` ライブラリがVRMモデルを扱えます。three-vrmでは、VRMのHumanoidボーンが `VRMHumanoid` クラス経由で操作できます。今回求めたクォータニオンを適用する際、Three.jsでは左手系→右手系の換算や軸回転の差異に注意が必要ですが、基本的な回転値自体はVRM内部に正しく書き込まれていれば表示可能です。Unityの場合はUniVRMでVRMを読み込めますが、Unityは左手系なので**X軸を反転**する必要があります [4] 。幸い今回の手順でX,Z軸を反転しているため、Unity上でも正しく再生できるはずです。

- **モーションのBake**: もしダンスのように **全フレームのアニメーションを適用**したい場合、各フレームごとに上記計算を行い、得られたクォータニオンをglTF　Animationにキーとして追加していくことになります。pygltflibでもanimationsを構築できますが、フォーマット構造に注意が必要です（ノードのrotationに対する時間配列と値配列を設定し、samplerとchannelを作成）。オフライン処理なら、フレーム毎にVRMを書き出してFBX等に変換する方法も考えられますが、glTFのままAnimation拡張を書く方が直接的です。

- **確認**: 出力したVRMファイルを実際にビューアー（例えばWindows用のVRMビューア、Three.js + three-vrmの簡易アプリ、またはUnity Editor上）で開いてみて、ポーズが期待通りになっているか確認します。特に腕や脚が真逆の方向に曲がってしまう場合、軸反転やクォータニオンの順序がおかしい可能性があります。

デバッグと精度検証のポイント

最後に、モーション適用のデバッグや精度確認のためのTipsをいくつか紹介します。

- **骨軸の可視化**: 計算したボーン軸が正しいか確かめるため、簡単なプロットを行うと理解が深まります。例えばMatplotlibでMediaPipeのランドマークを3Dプロットし、そこに各ボーンの軸ベクトル（基底の方向）を線で描画してみます。正しく計算できていれば、上腕の軸は肘の方向を向き、副軸は手首方向に概ね沿うはずです。異常に傾いている軸があれば、計算の元になったランドマークの指定ミス（左右取り違えなど）を疑います。

- **ボーン長のチェック**: VRMモデルのボーン長（例えば上腕の長さ）とMediaPipe推定のボーン長は異なることがあります。今回の手順では回転のみを適用するため長さの違い自体は問題になりません。ただし、極端に長さが違うと見た目に不自然になる場合があります。対策として、VRMモデル側のTポーズをMediaPipe側の初期姿勢に合わせてスケール調整するか、もしくはMediaPipeの座標を一度モデルの大きさにスケーリングしてから回転計算するとよいでしょう。例えば、モデルと人物の身長を合わせるために全ランドマークを一定のスケール比で縮尺してから計算します。

- **初期姿勢のバイアス**: VRMモデルの初期姿勢がMediaPipeのデータと異なる場合、オフセットを補正する必要があります。多くのVRMモデルはTポーズですが、中にはAポーズ（腕が下がり気味）になっているものもあります。また首や足先のデフォルト回転が入っていることもあります。これらは**初期回転の差分**として補正クォータニオンを各ボーンに掛けておく必要があります。例えばモデルがAポーズなら、肩関節を数度上げる初期回転を逆に適用し、MediaPipeのTポーズ相当と一致させます。この調整は**最初のフレーム**でモデルと推定姿勢を重ねてみて、各ボーンにどの程度のずれがあるかを計算するとよいでしょう。その差をすべてのフレームの計算結果に乗じれば、常にモデルに対して正しいオフセットで動かせます。

- **出力クォータニオンの検証**: 算出したクォータニオンが正しいか、簡易に検証するには **再投影**してみる方法があります。例えば、VRMモデルの各ボーンについて、初期姿勢における子ボーン（または連結するランドマーク）の方向ベクトルを取り、そのベクトルにクォータニオンを適用して回転させます。得られたベクトルがMediaPipeの対応する現在のランドマーク方向と一致すれば正しく計算できています。例えば上腕なら、モデルのTポーズでの「肩->肘」の方向を取り、計算クォータニオンで回した結果がMediaPipeの肩->肘方向と同じになるか確認します。これが大きくズレている場合、回転計算に誤りがあります。

- **フレーム間のなめらかさ**: ダンスのような動きでは、MediaPipe推定値が多少ぶれることがあります。クォータニオン出力が**小刻みに振動する**ようなら、**スムージングフィルタ**を検討します。クォータニオンに直接フィルタをかけることは難しいですが、各ランドマーク座標に移動平均や一時的な補間を入れて揺れを低減する方法があります。また、クォータニオン間の球面線形補間（SLERP）を使って補正することも可能です。たとえば、前後フレームのクォータニオンq_{prev}, q_{next}の間を補間して滑らかな中間姿勢を作りノイズを低減します。ただし過度に平滑化するとキレのある動きが鈍ってしまうため、ダンス用途では高周波ノイズのみ除去する程度に留めます。

- **接地と物理的な調整**: VRMモデルの足が床にめり込んだり浮いたりしないか確認しましょう。MediaPipeはヒップ中心が原点ですが、VRMモデルではhipsボーンが重心よりやや上（腰）にあることが多いです。そのため、絶対位置で合わせると足の長さの差で浮く可能性があります。オフライン処理であれば、床面の高さを固定して**垂直方向のオフセット**をモデル全体に与えることも検討してください。例えば、一番下

のランドマーク（かかと位置など）が常にy=0（床）になるように全身を上下させると、足裏が床につきます。ダンスではこの調整で安定感が増します。

- **腕とひじの不自然さ**: 推定によっては肘が逆方向に曲がった姿勢になることがあります（いわゆるエラー姿勢）。こうした場合、ボーンの回転角度を検出して**非現実的な角度**であれば補正するロジックを入れることも重要です。例えば肘関節は通常0〜150度程度しか曲がらないので、もし計算結果が180度を超えるようなら前フレームとの補間でなだらかに戻す、あるいはそのフレームの値を無視する処理を入れます。同様に肩のひねりや膝の方向も制限を設けると破綻を防げます。

- **参照実装と比較**: MediaPipe→VRMの変換は「Kalidokit」というオープンソース実装が存在します [13] [14]。JavaScript実装ですが、アプローチは似ています。一度Kalidokitや類似のスクリプトでモーションを適用し、その結果と自前の結果を比較すると、軸の取り方や回転角のずれを確認できます。例えば同じフレームで両者のVRMポーズを比較し、手足の向きが違う場合は計算方法を見直すヒントになります。Kalidokitではスムーズなモーション用に若干イージング処理も入っているので、まったく同一にはなりませんが、骨の大まかな向きは一致するはずです。

以上、MediaPipeの3DランドマークからVRM Humanoidボーンの回転情報を生成する手順を詳細に説明しました。**座標系の変換**から**ボーンごとの回転計算**、**クォータニオン化と階層適用**、そして**実装上のポイントやデバッグ方法**まで網羅しています [1] [3]。このガイドを参考に実装を行えば、オフラインでMediaPipeの出力を高精度にVRMアバターへ反映できるでしょう。ぜひ実際にコードを書いて、仮想キャラクターにリアルなダンスモーションを与えてみてください。必要に応じて各種パラメータ（フィルタの強さ、オフセット角度など）を調整し、納得のいくモーションになるまでチューニングすることをおすすめします。

---

[1] python - Transform value 3D coordinate pose landmarks (Mediapipe) to real world value in meters? - Stack Overflow
https://stackoverflow.com/questions/72185599/transform-value-3d-coordinate-pose-landmarks-mediapipe-to-real-world-value-in

[2] Real-Time 3D Pose Detection & Pose Classification with Mediapipe ...
https://bleedaiacademy.com/introduction-to-pose-detection-and-basic-pose-classification/

[3] glTF™ 2.0 Specification - Khronos Registry
https://registry.khronos.org/glTF/specs/2.0/glTF-2.0.html

[4] Issue #257 · KhronosGroup/UnityGLTF - Coordinate System - GitHub
https://github.com/KhronosGroup/UnityGLTF/issues/257

[5] [6] [8] [12] ProtoFlux:Cross - Resonite Wiki
https://wiki.resonite.com/ProtoFlux:Cross

[7] [9] [10] [11] Quaternion from two vectors • RAW
https://raw.org/proof/quaternion-from-two-vectors/

[13] [14] github.com-yeemachine-kalidokit_-_2021-11-13_04-43-47 : yeemachine : Free Download, Borrow, and Streaming : Internet Archive
https://archive.org/details/github.com-yeemachine-kalidokit_-_2021-11-13_04-43-47