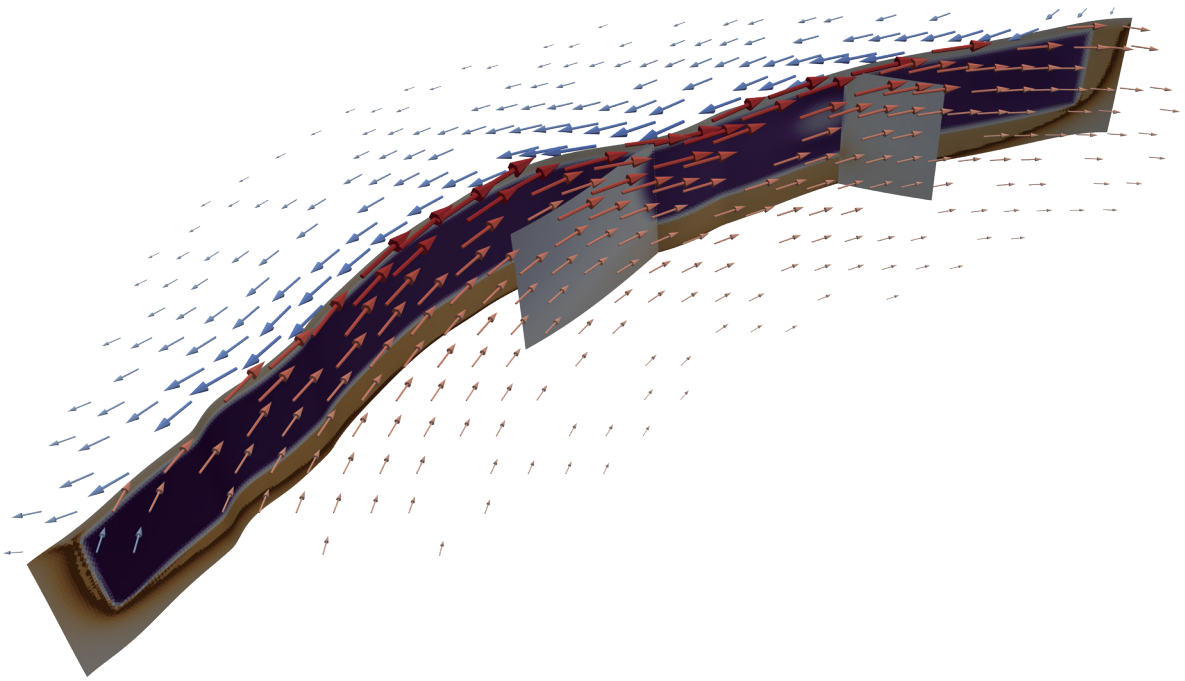
PyQuake3D

3D BOUNDARY ELEMENT METHODS FOR EARTHQUAKE MODELING

User Manual

Version 1.0.0



Rongjiang Tang[1] and Luca Dal Zilio[2]

PyQuake3D: A Python tool for 3-D earthquake sequence simulations of seismic and aseismic slip

[1] rongjiang@csj.uestc.edu.cn

[2] luca.dalzilio@ntu.edu.sg

Contents

1 Introduction

`PyQuake3D` is an open-source, Python-based Boundary Element Method (BEM) framework designed to simulate sequences of earthquakes and aseismic slip (SEAS) on geometrically complex three-dimensional (3D) fault systems governed by rate- and state-dependent friction laws. It supports fully arbitrary fault geometries embedded in either a uniform elastic half-space or full-space medium, including non-planar surfaces, fault stepovers, branches, and roughness. This document provides a general overview of `PyQuake3D`, including its main capabilities, usage instructions, and a detailed description of the required input parameters.

Two implementations of the code are currently available:

- **GPU-accelerated version**: Utilizes Python's `ProcessPoolExecutor` for parallel evaluation of Green's functions (kernels), and leverages GPU-based linear algebra libraries (`CuPy`) to accelerate dense matrix–vector operations.
- **H-matrix + MPI version**: Reduces memory footprint and computational cost using hierarchical matrix (H-matrix) compression via Adaptive Cross Approximation (ACA). Distributed parallelism is implemented with MPI (`mpi4py`) to enable efficient large-scale simulations on high-performance computing (HPC) platforms.

`PyQuake3D` is designed to scale from local workstations to HPC clusters, making it a flexible and extensible platform for researchers, students, and educators interested in exploring the physics of earthquake cycles across a wide range of spatial and temporal scales.

2 Contribution

PyQuake3D was developed by Rongjiang Tang and Luca Dal Zilio, who implemented the core framework, including the Boundary Element Method for simulating seismic cycles on geometrically complex 3D faults governed by a regularized rate-and-state friction governed by aging law. The software can couple the boundary element governing equation and the finite difference fluid pore pressure diffusion equation, thus support for pore pressure varing with slip due to inelastic processes including dilatancy and pore compaction.

The software also features MPI-based hierarchical matrix compression and GPU acceleration for scalable performance. We welcome contributions from the research community. Please follow the contribution guidelines and ensure consistency with the existing codebase. The project is under active development, and we encourage feedback and collaborative extensions. PyQuake3D is distributed under an open-source license to support reproducibility and broad adoption in earthquake science. If you use PyQuake3D in your research, please cite the corresponding reference to acknowledge the original work.

3 Theoretical background

3.1 Governing Equations

To solve the earthquake dynamic simulation problem using boundary integral equations, we assume the fault plane is embedded in an elastic half-space or full-space with homogeneous and constant elastic moduli and constant tectonic loading rate is imposed across the whole fault interface. The elastic stress transfer due to slip on the fault is described in Equations 1 (shear stress) and 2 (normal stress), with the radiation damping assumption [Rice, 1993]:

$$\tau_i = \tau_0 - \sum_{j=1}^{n} k_{ij}^s \left(u_j - V_{pl}t\right) - \frac{\mu}{2c_s}\frac{\partial u_i}{\partial t} \tag{1}$$

$$\bar{\sigma}_i = \bar{\sigma}_0 + \sum_{j=1}^{n} k_{ij}^N \left(u_j - V_{pl}t\right) \tag{2}$$

where V_{pl} is the imposed tectonic slip rate, μ is the shear modulus, c_s is the shear wave speed, and u_j is the slip at the j-th element. The kernels k_{ij}^s and k_{ij}^N represent the shear and normal stiffness matrices, respectively. The last term in Equation 1 captures radiation damping and approximates inertial effects, which is adopted to avoid the unbounded slip velocity that would otherwise develop as a consequence of instability in a quasi-static model [Rice, 1993].

To compute the k_{ij}^s and k_{ij}^N , we employ analytical formulas for static stress induced by triangular dislocations in a homogeneous elastic full-space and half-space, as described by [Nikkhoo and Walter, 2015]. Since our objective is to simulate three-dimensional complex non-planar fault geometries, optimizations specific to planar faults, such as constructing K in the Fourier domain [Rice, 1993] and leveraging translational invariance to compute stresses as convolutions using the Fast Fourier Transform, are not applicable. Instead, we utilize CPU-based multiprocessing or MPI to accelerate the computation of Green's functions required for generating the stiffness matrix.

To construct the differential equation, we take the time derivative of equation Equation 1 and 2, while considering the external stress loading.

$$\frac{d\tau_i}{dt} = -\sum_{j=1}^{N} k_{ij}^s \left(V_j - V_{pl}\right) + \dot{\tau}_i - \frac{\mu}{2c_s}\frac{dV_i}{dt} \tag{3}$$

$$\frac{d\sigma_i}{dt} = \sum_{j=1}^{N} k_{ij}^N V_j + \dot{\sigma}_i \tag{4}$$

Where the $\dot{\tau}_i$ and $\dot{\sigma}_i$ represent the tectonic loading rates for shear and normal stress on the i-th fault cell, respectively.

To solve the differential Equation 3 and 4, we need to incorporate the boundary conditions governed by the laboratory-derived rate- and state-dependent friction law (RSF) [Dieterich, 1979, Ruina, 1983]. The friction coefficient is given by a regularized aging law formulation:

$$f(V,\theta) = a \sinh^{-1}\left[\frac{V}{2V_0}\exp\left(\frac{f_0 + b\ln\left(\frac{V_0\theta}{d_c}\right)}{a}\right)\right] \tag{5}$$

$$\frac{d\theta}{dt} = 1 - \frac{V\theta}{d_c} \tag{6}$$

Where d_c represents the characteristic slip distance, V_0 the reference slip rate, f_0 the reference friction coefficient, θ the state variable. The parameters a and b are used to depict the direct and evolution effects of shear resistance during and following a velocity step.

To simplify, we replace the state variables θ with ψ

$$\psi = f_0 + b\ln\left(\frac{V_0\theta}{d_c}\right) \tag{7}$$

Then, we obtain the transformed friction relation:

$$\frac{\tau_i}{\sigma_i} = a \operatorname{arcsin}(h) \left(\frac{V_i}{2V_0} \exp\left(\frac{\psi_i}{a} \right) \right) \tag{8}$$

$$\frac{d\psi_i}{dt} = \frac{b}{d_c} \left[V_0 \exp\left(\frac{f_0 - \psi_i}{b} \right) - V_i \right] \tag{9}$$

Let's return to Equation 3. The $\frac{dV_i}{dt}$ can be replaced using the chain rule.

$$\frac{dV_i}{dt} = \frac{\partial V_i}{\partial \tau_i} \frac{d\tau_i}{dt} + \frac{\partial V_i}{\partial \sigma_i} \frac{d\sigma_i}{dt} + \frac{\partial V_i}{\partial \psi_i} \frac{d\psi_i}{dt} \tag{10}$$

Then we obtain final form of the shear stress variation

$$\frac{d\tau_i}{dt} = \left(1 + \frac{\mu}{2c_s} \frac{\partial V_i}{\partial \tau_i} \right)^{-1} \left[-\sum_{j=1}^{N} k_{ij}^s V_j + \dot{\tau}_i - \frac{\mu}{2c_s} \left(\frac{\partial V_i}{\partial \sigma_i} \frac{d\sigma_i}{dt} + \frac{\partial V_i}{\partial \psi_i} \frac{d\psi_i}{dt} \right) \right] \tag{11}$$

Shear stress can be decomposed in both strike-slip and dip-slip directions τ_1 and τ_2 , and V_1 and V_2 refer to the slip rates in both directions

$$\frac{d\tau_{1,i}}{dt} = \left(1 + \frac{\mu}{2c_s} \frac{\partial V_{1,i}}{\partial \tau_{1,i}} \right)^{-1} \left[-\sum_{j=1}^{N} k_{ij}^{s1} V_{1,j} + \dot{\tau_{1,i}} - \frac{\mu}{2c_s} \left(\frac{\partial V_{1,i}}{\partial \sigma_i} \frac{d\sigma_i}{dt} + \frac{\partial V_{1,i}}{\partial \psi_i} \frac{d\psi_i}{dt} \right) \right] \tag{12}$$

$$\frac{d\tau_{2,i}}{dt} = \left(1 + \frac{\mu}{2c_s} \frac{\partial V_{2,i}}{\partial \tau_{2,i}} \right)^{-1} \left[-\sum_{j=1}^{N} k_{ij}^{s2} V_{2,j} + \dot{\tau_{2,i}} - \frac{\mu}{2c_s} \left(\frac{\partial V_{2,i}}{\partial \sigma_i} \frac{d\sigma_i}{dt} + \frac{\partial V_{2,i}}{\partial \psi_i} \frac{d\psi_i}{dt} \right) \right] \tag{13}$$

Where

$$\frac{\partial V_{1,i}}{\partial \tau_{1,i}} = \frac{2V_0}{a\sigma_i} \exp\left(-\frac{\psi_i}{a} \right) \cosh\left(\frac{\tau_{1,i}}{a\sigma_i} \right) \tag{14}$$

$$\frac{\partial V_{2,i}}{\partial \tau_{2,i}} = \frac{2V_0}{a\sigma_i} \exp\left(-\frac{\psi_i}{a} \right) \cosh\left(\frac{\tau_{2,i}}{a\sigma_i} \right) \tag{15}$$

$$\frac{\partial V_{1,i}}{\partial \sigma_i} = -\frac{2V_0\tau_{1,i}}{a\sigma_i^2} \exp\left(-\frac{\psi_i}{a} \right) \cosh\left(\frac{\tau_{1,i}}{a\sigma_i} \right) \tag{16}$$

$$\frac{\partial V_{2,i}}{\partial \sigma_i} = -\frac{2V_0\tau_{2,i}}{a\sigma_i^2} \exp\left(-\frac{\psi_i}{a} \right) \cosh\left(\frac{\tau_{2,i}}{a\sigma_i} \right) \tag{17}$$

$$\frac{\partial V_{1,i}}{\partial \psi_i} = -\frac{2V_0}{a} \exp\left(-\frac{\psi_i}{a} \right) \sinh\left(\frac{\tau_{1,i}}{a\sigma_i} \right) \tag{18}$$

$$\frac{\partial V_{2,i}}{\partial \psi_i} = -\frac{2V_0}{a} \exp\left(-\frac{\psi_i}{a} \right) \sinh\left(\frac{\tau_{2,i}}{a\sigma_i} \right) \tag{19}$$

We use backslip method ([Heimisson, 2020]) with a plate rate V_{pl} for plate motion loading, such that $V_{1,i}$ and $V_{2,i}$ can by replaced by $V_{1,i} - V_{pl,i}$ and $V_{2,i} - V_{pl,i}$. By substituting Equation 14 to 19 into Equation 12 and 13, Equation 4, 9, 12 and 13 can form a system of ordinary differential equations with a dimensional size of $4N$:

$$\frac{dy}{dt} = f(y) \tag{20}$$

$$y = (\psi_1, \ldots, \psi_N, \tau_{1,1}, \ldots, \tau_{1,N}, \tau_{2,1}, \ldots, \tau_{2,N}, \sigma_1, \ldots, \sigma_N) \tag{21}$$

We solve these equations using the Dormand-Prince 5th-order Runge–Kutta method with adaptive time stepping [Press et al., 2007]. After each iteration, other important variables such as the slip velocity, shear traction amplitude and rake angle are updated by the following formula:

$$\tau = \sqrt{\tau_1^2 + \tau_2^2} \tag{22}$$

$$\mathrm{V}_1 = 2V_0 \cdot \exp\left(-\frac{\psi}{a}\right) \cdot \sinh\left(\frac{\tau_1}{\sigma}\right) \tag{23}$$

$$\mathrm{V}_2 = 2V_0 \cdot \exp\left(-\frac{\psi}{a}\right) \cdot \sinh\left(\frac{\tau_2}{\sigma}\right) \tag{24}$$

$$V = \sqrt{V_1^2 + V_2^2} \tag{25}$$

$$\mathrm{rake} = \arctan\left(\frac{\tau_2}{\tau_1}\right)$$

To implement the adaptive time step, we calculate the time step for the next iteration h_{n+1} based on the relative error between the 4- and 5-order Runge-Kutta formulas.

$$h_{n+1} = S h_n \left(\frac{\epsilon_0}{\epsilon_k}\right)^{0.2} \tag{26}$$

where the safety factor S = 0.9, and $\epsilon_0 = 10^{-4}$ is the set threshold to determine whether the desired precision has been achieved. The relative error is given by

$$\epsilon_k = \max\left(\left|\frac{y_{n+1} - y^*_{n+1}}{y_{n+1}}\right|\right) \tag{27}$$

Here y^*_{n+1} is the result of the fifth order computation and y_{n+1} is the result of the fourth order computation. For each Runge–Kutta iteration, we first check if $C = \frac{\epsilon_0}{\epsilon_k}$ is less than 1. If it is, we update the time step using Equation 26 and then constrain the $h_{n+1} = \min(1.5h_n, h_{n+1})$. Otherwise, we update the step size using 26 then constrain the $h_{n+1} = \max(0.5h_n, h_{n+1})$. The Runge-Kutta iteration is re-executed until the C becomes less than 1.0 or the maximum number of iterations is reached.

3.2 Hierarchical Matrix Compression and MPI Parallelization in PyQuake3D

According to Börm's foundational work [Börm et al., 2003], PyQuake3D implements a Python-based H-matrix framework from the ground up. The implementation, contained in the `Hmatrix.py` module, supports MPI-based parallel acceleration and is designed with modularity, making it easily separable and adaptable for use in other applications.

The core idea of the H-matrix is to apply low-rank approximation to far-field submatrices while keep the dense near-field submatrices. Therefore, the essential purpose is to decompose the original matrix in a reasonable and efficient manner and to identify the submatrices suitable for low-rank approximation. The structure of the H-matrix is built upon a cluster tree and a block tree. The construction begins with generating a cluster tree based on the element index, which is then used to form the block cluster tree through pairwise combinations of the clusters. The implementation of H-matrices in *PyQuake3D* includeds four parts: cluster tree construction, block cluster tree generation, low-rank approximation, as well as MPI acceleration.

3.3 Cluster and Block tree construction

A simple method of building a cluster tree is based on geometry-based splittings of the index set. The unit coordinates in 3D space are the basis e_x, e_y, e_z of the canonical unit vectors. The following algorithm will split a given cluster $\tau \subset I$ into two sons such that the points with canonical coordinate x_i are separated by a hyper-plane (Algorithm 1).

Algorithm 1 Geometric Splitting of an Index Cluster

procedure SPLIT(τ, **var** τ_1, **var** τ_2)
▷ Choose a direction for geometrical splitting of the cluster τ
  for $j := 1$ **to** d **do**
    $\alpha_j := \min\{\langle e_j, x_i\rangle : i \in \tau\}$ ▷ $\langle\cdot,\cdot\rangle$ is the $\mathbb{R}^d$ Euclidean product, $d = 3$
    $\beta_j := \max\{\langle e_j, x_i\rangle : i \in \tau\}$
  end for
  $j_{\max} := \arg\max\{\beta_j - \alpha_j : j \in \{1, \ldots, d\}\}$
▷ Split the cluster τ in the chosen direction
  $\gamma := (\alpha_{j_{\max}} + \beta_{j_{\max}})/2$
  $\tau_1 := \emptyset$; $\tau_2 := \emptyset$
  for $i \in \tau$ **do**
    if $\langle e_{j_{\max}}, x_i\rangle \leq \gamma$ **then**
      $\tau_1 := \tau_1 \cup \{i\}$
    else
      $\tau_2 := \tau_2 \cup \{i\}$
    end if
  end for
end procedure

The cluster tree can be used to define a block tree by forming pairs of clusters recursively, and an admissibility condition is constructed by the following procedure (Algorithm 2):

For a pair of index clusters (τ, σ), the corresponding submatrix is $A_{\tau,\sigma}$. These matrix blocks are organized into a block cluster tree, which guides the hierarchical representation of A. If the clusters t and s are both non-leaf nodes with children t_1, t_2 and s_1, s_2 respectively, the submatrix A_{ts} can be further decomposed into four submatrices (Figure 1a).

Algorithm 2 Build BlockTree

procedure BUILDBLOCKTREE($\tau \times \sigma$)
  begin
  if $\tau \times \sigma$ is not admissible and $|\tau| > C_{\text{leaf}}$ and $|\sigma| > C_{\text{leaf}}$ **then**
    begin
    $S(\tau \times \sigma) := \{\tau' \times \sigma' : \tau' \in S(\tau),\ \sigma' \in S(\sigma)\}$
    for $\tau' \times \sigma' \in S(\tau \times \sigma)$ **do**
      BuildBlockTree($\tau' \times \sigma'$)
    end for
    end
  else
    $S(\tau \times \sigma) := \emptyset$
  end if
  end
end procedure

3.4 Admissibility Condition

Matrix blocks $A_{\tau,\sigma}$ are classified into *admissible* and *inadmissible* based on the geometric configuration of τ and σ. we need an admissibility condition that allows us to check if a candidate $(\tau \times \sigma)$ allows for

a suitable low rank approximation (Algorithm 2).

A relatively general and practical admissibility condition for clusters in $\mathbb{R}^d$ can be defined by using bounding boxes: We define the canonical coordinate maps

$$\pi_k : \mathbb{R}^d \to \mathbb{R}, \quad x \mapsto x_k,$$

for all $k \in \{1, \ldots, d\}$ *(d=3 in 3D space). The bounding box for a cluster* τ *is then given by*

$$Q_\tau := \prod_{k=1}^{d} [a_{\tau,k}, b_{\tau,k}], \quad \text{where} \quad a_{\tau,k} := \min(\pi_k \Omega_\tau) \quad \text{and} \quad b_{\tau,k} := \max(\pi_k \Omega_\tau).$$

Obviously, $a_{\tau,k}$ and $b_{\tau,k}$ are the minimum and maximum value in the k-th dimension of the coordinate set Ω_τ, we have $\Omega_\tau \subseteq Q_\tau$, so we can define the admissibility condition

$$\min\{\operatorname{diam}(Q_\tau), \operatorname{diam}(Q_\sigma)\} \leq \eta \operatorname{dist}(Q_\tau, Q_\sigma) \tag{28}$$

We can compute the diameters and distances of the boxes by

$$\operatorname{diam}(Q_\tau) = \left(\sum_{k=1}^{d} (b_{\tau,k} - a_{\tau,k})^2 \right)^{1/2} \quad \text{and} \tag{29}$$

$$\operatorname{dist}(Q_\tau, Q_\sigma) = \left(\sum_{k=1}^{d} \left(\max(0, a_{\tau,k} - b_{\sigma,k})^2 + \max(0, a_{\sigma,k} - b_{\tau,k})^2 \right) \right)^{1/2}. \tag{30}$$

3.5 Low-Rank Approximation of Admissible Blocks

When calculating the stress Green's function, we need to avoid constructing the full dense BEM matrix by Low-Rank Approximation. This can be helpful for reducing memory costs and, in some situations, actually results in a faster solver too. Singular value decomposition (SVD) is an extremely efficient approximation, but it still suffers from the need to compute the entire matrix block in the first place, an $O(n^2)$ operation!

The most useful solution for our setting is the adaptive cross approximation (ACA) method [Bebendorf, 2000, Rjasanow and Steinbach, 2007], but most real-world application use either *ACA with partial pivoting* or the *ACA+* algorithm.[Grasedyck, 2005]. The basic idea of ACA+ is to approximate a matrix with a rank 1 outer product of one row and one column of that same matrix, and then iteratively use this process to construct an approximation of arbitrary precision. ACA+ uses orthogonal projections and recompression to better control the error and avoid poor pivot choices, and improves the stability and accuracy of the standard ACA.

3.6 Parallelization of H-matrix Construction and Matrix-Vector Multiplication with MPI

We construct the cluster tree, block cluster tree, and establish the overall H-matrix framework. This includes the initial distribution of the index sets and the assignment of matrix blocks. Once the hierarchical structure is built, the matrix blocks are distributed across different MPI processes for parallel computation of elements (e.g., `MPI_send`, `MPI_recv`). We adopt a dynamic task allocation strategy in MPI to mitigate load imbalance issues that may arise from static task assignment (Figure 1d).

Each process is responsible for computing the entries of its assigned matrix blocks. For admissible blocks, a low-rank approximation is performed using the ACA algorithm. For non-admissible blocks, typically corresponding to leaf nodes in the block cluster tree, are the full dense matrix computed by evaluating all pairwise Green's function interactions between source and target elements(Figure 1c). .

During matrix-vector multiplication, each process independently computes the product of its local matrix blocks with the corresponding portion of the input vector. The final global result vector is then obtained by summing the local contributions across all processes (`MPI_reduce`) (Figure 1e).

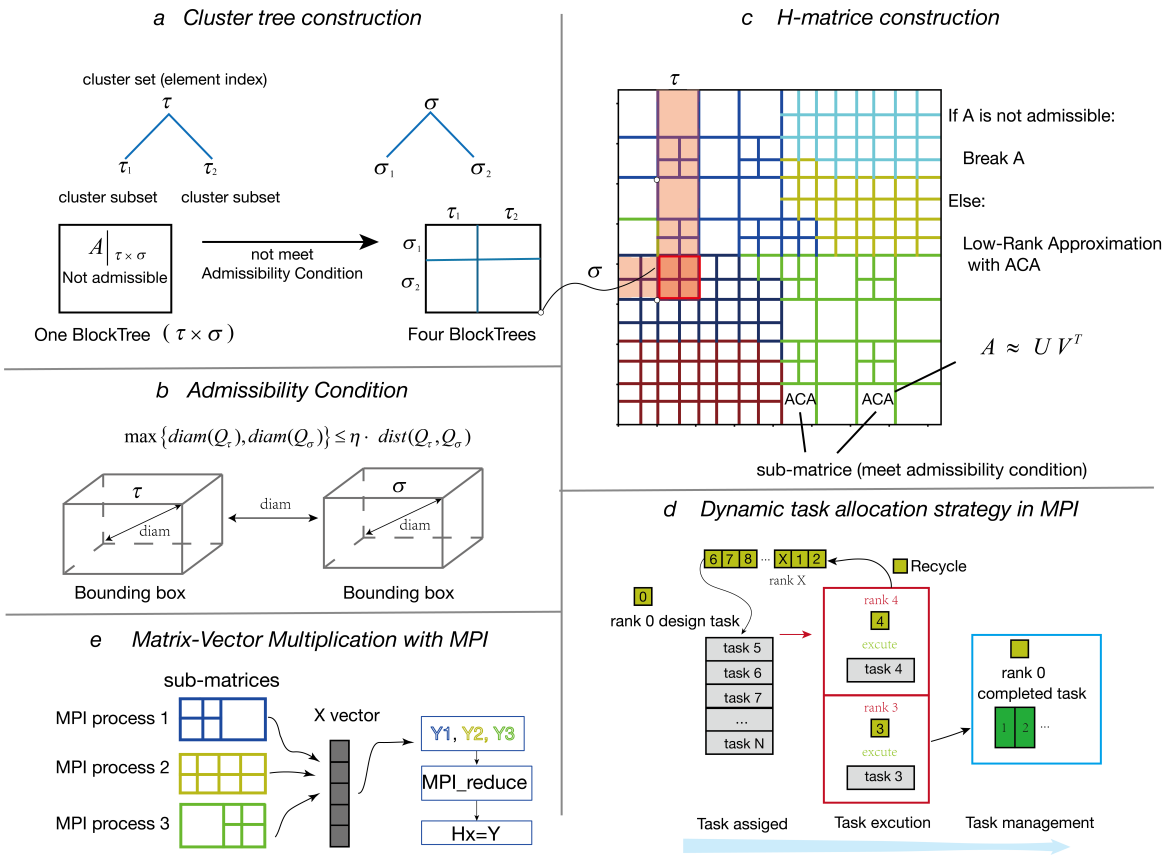


Figure 1: Construction of hierarchical matrix speed up by MPI

3.7 Definition of strike-slip and dip-slip direction

The rake angle is defined as the angle between the fault slip vector and the fault strike direction within the fault plane. In seismology, the strike direction is conventionally defined: when facing the dip direction of the fault plane, the strike is taken to be to the observer's right—this follows the right-hand rule. In *PyQuake3D*, the strike direction is determined by the ordering of the element's nodes—either clockwise or counterclockwise. To get the defined strike-slip direction for each element, we first calculate the normal vector of the element in the global coordinate system by

$$\vec{e_3} = \vec{v}_{ab} \times \vec{v}_{ac} \tag{31}$$

Based on the geometric relationship of $\vec{e_1}$ and $\vec{e_3}$ (Figures 2b,c), the three component of strike-slip direction of the fault cell (x-direction of the local coordinate system) can be obtained by

$$\begin{aligned} e_{11} &= e_{32} \\ e_{12} &= -e_{31} \\ e_{13} &= 0 \end{aligned} \tag{32}$$

Then we can easily get the last direction (dip-slip direction) of the fault element get by

$$\vec{e}_2 = \vec{e}_3 \times \vec{e}_1 \tag{33}$$

Figures 2d and e and d show that the definition of strike slip and dip slip directions obviously depends on the ordering of nodes, so when modeling, we need to first clarify the ordering direction of nodes, and then determine the definition of fault slip direction.

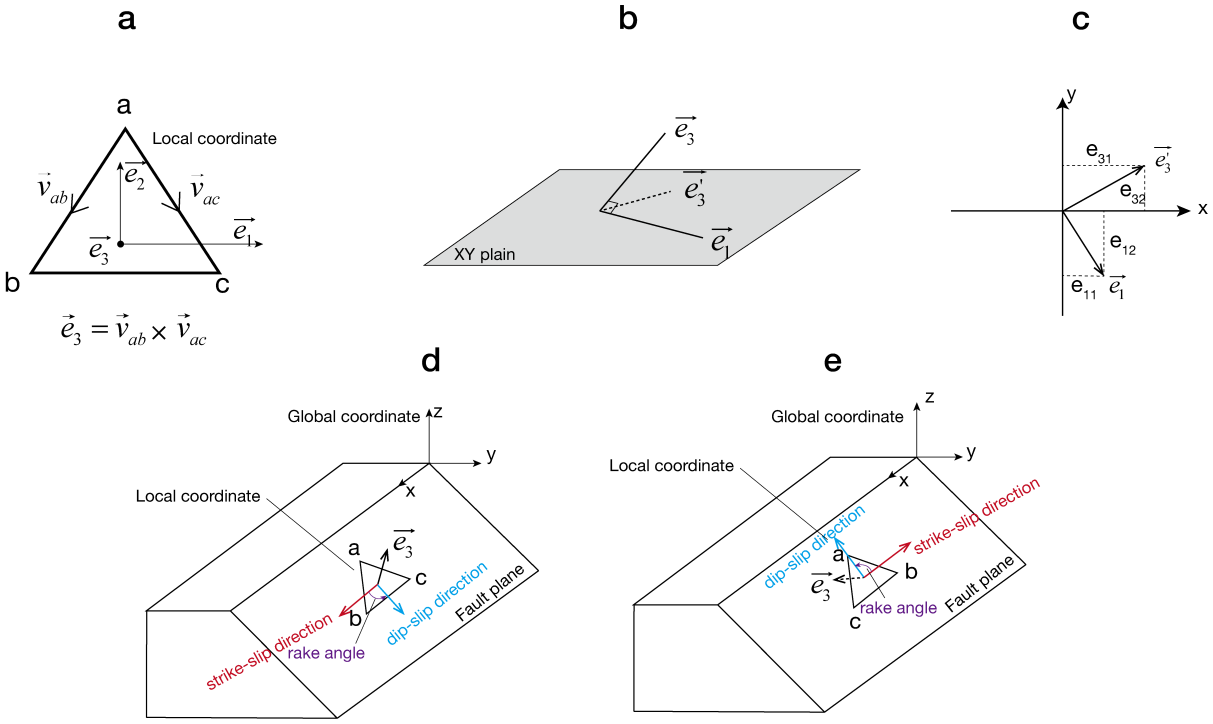


Figure 2: Definition of strike-slip and dip-slip direction. (**a**) Calculation of normal vector or z-vector in element local coordinate system $\vec{e_3}$. (b) Projection of $\vec{e_3}$ on XY plane $\vec{e_3}'$. (c) Conversion between $\vec{e_3}'$ and $\vec{e_1}$. (d) Looking from the negative direction of the z-axis, the unit nodes are counterclockwise, and the element normal is outwards. The strike-slip direction is defined as the positive x direction, and the y direction can be obtained by cross-producting the z and x unit vectors. (e) Looking from the negative direction of the z-axis, the unit nodes are clockwise, and the element normal is inwards. The strike-slip direction is defined as the negative x direction.

3.8 Coordinate system

In post-processing, we need to do transforms between local and global coordinates for some vector results, such as slip direction. Base on the relationship in Figure 2d and e, the transform from local coordinates to global coordinates can be obtained by coordinate rotation.

$$\begin{bmatrix} e_{11} & e_{21} & e_{31} \\ e_{12} & e_{22} & e_{32} \\ e_{13} & e_{23} & e_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix} \tag{34}$$

Where x is a vector of local coordinates and X is a vector of global coordinates.

4 Code Availability and Installation

The **PyQuake3D** code is openly available at https://github.com/Computational-Geophysics/PyQuake3D, and we recommend installing it within a conda environment. If you are not familier with conda, we pro-

vide a step-by-step guide to installing at https://github.com/Computational-Geophysics/PyQuake3D/blob/main/tutorials
QD_low_resolution/tutorial_BP5.ipynb, which cover installation, compilation, setup, and running a simple simulation. We also describe the details in Section 5.5.

4.1 python Requirements

`PyQuake3D` is implemented in Python and requires the following packages for successful installation and execution:

Package
`python`≥ 3.8
`numPy`
`psutil`
`matplotlib`
`sciPy`
`joblib`
`mpi4py`
`imageio`
`h5py`
`joblib`
`pyvista`

Table 1: Required dependencies for running `PyQuake3D`.

Notes: PyQuake3D supports Python 3.8 and above, so there is no need to specify a version when installing the library. To install all required dependencies, simply run:

```
pip install -r requirements.txt
```

Or use conda to install at root directory:

```
conda env update -f environment.yml
```

Ensure that the correct Python environment is activated before installation, especially when using virtual environments or managing dependencies with tools such as `conda`.

Install cupy if you want to use GPU acceleration, we recommened to use conda in prompt terminal (e.g. CUDA 11.8):

```
conda install -c conda-forge cupy cudatoolkit=11.8
```

Notes: CuPy relies on NVIDIA CUDA for GPU computing, so the following components are required:

1. NVIDIA GPU Driver: Ensure your system has a CUDA-capable NVIDIA GPU (check compatibility at https://developer.nvidia.com/cuda-gpus). The driver version must support the CUDA Toolkit version used by CuPy (e.g., CUDA 10.2–12.x for CuPy v13.x).

2. CUDA Toolkit: CuPy supports specific CUDA versions (check the latest compatibility at https://docs.cupy.dev/en/st
Common versions: CUDA 11.8 or 12.2 are widely used.

4.2 C++ Requirements

The TDstressFS_C.cpp in folder src is a C++ source file that computes full-space and half-space Green's functions, translated from the Python script TDstressFS.py and TDstressHS.py to leverage C++'s per-

formance for efficient numerical calculations. It is compiled into a dynamic library, TDstressFS_C.so, using a provided Makefile, which must be executed with the make command before running the code to ensure compatibility across different computing environments. The generated library is called by the Python script Hmatrix.py via dynamic loading (e.g., using ctypes). To use it, navigate to the code directory src, run make to build TDstressFS_C.so.

To make .so file and implement MPI in Linux system, install g++ compiler (including gcc and other related tools) are needed. Common options are:

- sudo apt update
- sudo apt install g++
- sudo apt install openmpi-bin libopenmpi-dev

5 How to run PyQuake3D

To get started, clone the PyQuake3D repository from GitHub:

```
https://github.com/Computational-Geophysics/PyQuake3D
```

5.1 Usage Modes

PyQuake3D supports two execution backends:

- **single CPU/GPU-based backend**: Constructs dense stiffness matrices using direct evaluation of all source–receiver interactions. GPU acceleration is implemented via `CuPy`, and parallelism is handled using Python's `ProcessPoolExecutor` for kernel calculations.
- **MPI-based CPU backend**: Implements a memory-efficient H-matrix representation of the stiffness matrix, distributed across multiple processors using `mpi4py`. This version is well-suited for simulations with >40,000 elements and optimized for HPC systems.

This structure allows users to scale from fast exploratory models on local machines to high-resolution, physics-rich earthquake simulations on supercomputing clusters. The modular design also facilitates extension of the framework to include additional rheologies, boundary conditions, or coupling with geodynamic models.

PyQuake3D can be executed either in single CPU/GPU mode (`main_gpu.py`) or in MPI-parallel mode (`main_mpi.py`), depending on the size of your model. All simulations are launched from the project's root directory using the `main_gpu.py` and `main_mpi.py` script located in the `src` folder.

5.2 Standard Execution (single GPU/CPU)

To run a simulation using the standard execution mode (with optional GPU acceleration), use the following command in the PyQuake3D root directory:

```
python src/main.py -g <input_geometry_file> -p <input_parameter_file>
```

For example, to execute the benchmark simulation `BP5-QD`:

```
python src/main_gpu.py -g examples/BP5-QD/bp5t.msh -p examples/BP5-QD/parameter.txt
```

Ensure you modify the input parameter (`parameter.txt`) as follows:

- `InputHetoparamter`: `True`
- `Inputparamter file`: bp5tparam.dat

5.3 MPI-Based Execution (High-Resolution Models)

For large-scale simulations using the MPI-parallel H-matrix version, use the following command:

```
mpirun -np <N> python src/main.py -g <input_geometry_file> -p <input_parameter_file>
```

For example, using 10 parallel processes:

```
mpirun -np 10 python src/main_mpi.py -g examples/BP5-QD/bp5t.msh -p examples/BP5-QD/parameter.txt
```

Note: On Windows systems, replace `mpirun` with `mpiexec`.In the MPI version, the stress Green's function is implemented in C++ and invoked from Python. On Linux systems, the shared library `TDstressFS_C.so` has already been compiled. On Windows systems, however, it is necessary to compile `src/TDstressFS_C.cpp` into a DLL before it can be imported in Hmatrix.py (see line 21). For the MPI version, it is recommended to run the program in a Linux environment.

5.4 Parameter Configuration

Each example folder contains a `parameter.txt` file, which defines the model settings, material properties, and solver options. Ensure this file is correctly configured before launching a simulation. A detailed explanation of the parameter settings is provided in Section 7.

5.5 Step by step to run a simple case

We provides a step-by-step guide to use **PyQuake3D**. We'll cover installation, compilation, setup, and running a simple simulation. ‘

5.5.1 Standard Execution single GPU/CPU version on Windows

Step 1: Set Up Conda environment

1. Download from https://www.anaconda.com/docs/getting-started/miniconda/install
2. Then install it using exe program.
3. Open miniconda Prompt and navigate to the working directory(e.g. E:\work) using: E: then cd E:\work
4. After installation, create a new environment: conda create -n PyQuake3D python=3.12
5. Activate the environment: conda activate PyQuake3D
6. Install Jupyter notebook using: conda install jupyter notebook

Step 2: Install Python Dependencies

1. Download from https://github.com/Computational-Geophysics/PyQuake3D
2. Install PyQuake3D in conda Prompt using pip:
   pip install -r requirements.txt
   or
   python -m pip install -r requirements.txt
   or
   conda env update -f environment.yml
   (make sure PyQuake3D environment is activated)

Notes:PyQuake3D supports Python 3.8 and above, so there is no need to specify a specific version when installing the library.

Step 3: Runging a simple case

The easiest way to run pyquake3d is to run the single CPU version of Pyquake3D, without installing cupy. **Here we provide a BP5-QD simulation with low resolution, with initial paremeters being set up in parameter.txt.** The procedures are follows:

1. Read model and mesh parameter(Please refer to mannul for specific meanning).
2. Calculating Green functions.
3. Calculating Qusi-Dynamic model.
4. Visualize the results.

You can also run the BP5-QD example case in the terminal:

python src/main_mpi.py -g examples/BP5-QD/bp5t.msh -p examples/BP5-QD/parameter.txt

Initial: Import all sub-functions and libraries

```
#Let runing PyQuake3D
#First Import all sub-functions

import readmsh
import numpy as np
import sys
import matplotlib.pyplot as plt
import QDsim_gpu
from math import *
import time
import argparse
import os
import psutil
from datetime import datetime
process = psutil.Process(os.getpid())
```

Listing 1: Import all sub-functions and libraries

Step 1: read the model file, parameter file, and create a basic class

Create mesh model through Gmesh

You can create your own mesh model through Gmesh, download from https://gmsh.info/ . You can download the latest version, but remember to use version 2 when exporting mesh model. First you need

to prepare .geo file, and open it using Gmesh and it automatically mesh the domain.Then expot .msh file by $File->export->Inputbp5t.msh->Version2ASCII$).

A simple asperity_circle.geo format and notes are as follows:

```
// Parameters
angle = 25;    // Dip angle of the fault in degrees
Len = 100;     // Fault length along x-axis
width = 40;    // Fault width along z-axis
lc = 1.5;      // Characteristic length for mesh cell size

// Define points for the rectangular fault plane
Point(1) = {-Len/2, 0, 0, lc};    // Bottom-left corner (A)
Point(2) = {Len/2, 0, 0, lc};     // Bottom-right corner (B)
Point(3) = {Len/2, 0, -width, lc}; // Top-right corner (C)
Point(4) = {-Len/2, 0, -width, lc}; // Top-left corner (D)

// Alternative points for a dipping fault (commented out)
//Point(3) = {Len/2, width * Cos(angle * Pi / 180), -width * Sin(angle * Pi / 180), lc};
//Point(4) = {-Len/2, width * Cos(angle * Pi / 180), -width * Sin(angle * Pi / 180), lc
    };

// Define lines connecting the points
Line(1) = {1, 2}; // Line from A to B (bottom edge)
Line(2) = {2, 3}; // Line from B to C (right edge)
Line(3) = {3, 4}; // Line from C to D (top edge)
Line(4) = {4, 1}; // Line from D to A (left edge)

// Define the surface by creating a line loop and plane
Line Loop(1) = {1, 2, 3, 4}; // Closed loop of lines A-B-C-D
Plane Surface(1) = {1};      // Create a planar surface from the line loop

// Mesh generation settings
//Mesh.Algorithm = 1;       // Delaunay triangulation
//Mesh.Algorithm = 8;       //Structured quadrilateral mesh
//Mesh.ElementOrder = 1;    // Use first-order
//Mesh.RecombineAll = 1;    // Recombine triangles

// Generate the 2D mesh
Mesh 2;

// Save the mesh to a file
Mesh.Format = 2;             // Output format: 1 = MSH1 format (Gmsh legacy format)
//Save "fault_mesh.msh";       // Save the mesh as fault_mesh.msh
```

Listing 2: A simple asperity_circle.geo file

Create parameter and external input file

The parameter.txt file serves as a configuration file for the simulation, defining all necessary parameters to control the execution of the model. It organizes input data into distinct sections for clarity and modularity, including:

1. **Files and Cells**: Specifies file paths and mesh cell size settings.
2. **Property:** Defines material properties of the fault, such as elastic moduli or density.
3. **Fluid:** Configures fluid-related parameters, like Dilatancy coefficient or pore pressure.
4. **Stress:** Sets stress conditions applied to the fault, such as normal or shear stress.
5. **Friction:** Describes frictional properties, including friction coefficients or laws.
6. **Nucleation:** Defines parameters for initiating fault slip, such as nucleation zone size or critical stress.
7. **Output:** Configures output settings, including data formats and variables to save.

If external data is imported as the initial model, just set **InputHetoparamter: True**, and stress and friction parameters can be ignored.

Note: Please refer to section 7 for a detailed explanation of each parameter. The order of all parameters in parameter.txt can be arbitrarily disrupted, but the parameter names must remain strictly unchanged.

Visualize mesh model using PyQuake3D

All initial parameters are easily accessible through class member variables.

```
#Establish the initial model and generate simulation class
sim0=QDsim_gpu.QDsim(elelst,nodelst,fnamePara,calc_greenfunc=False)
#Ouput initial state
fname='Init.vtk'
sim0.ouputVTK(fname)

import pyvista as pv
plotter = pv.Plotter(off_screen=True)
mesh = pv.read(fname)
print(mesh)
print(mesh.cell_data)
variab = 'a-b' #choose the arrays to display

scalar_bar_args={
        'title': variab,
        'position_x': 0.22,
        'position_y': 0.85,
    }
plotter.add_mesh(mesh, scalars=variab, show_edges=True,scalar_bar_args=scalar_bar_args)
     # show shear stress

plotter.show(cpos='xz')
```

Listing 3: Visualize mesh model

output:

Step 3: Calculating Qusi-Dynamic model by loop

```
start_time=time.time()

totaloutputsteps=int(sim0.Para0['totaloutputsteps'])
directory='outvtk'
if not os.path.exists(directory):
    os.mkdir(directory)

f=open('state.txt','w')
f.write('iteration time_step(s) maximum_slip1_rate(m/s) maximum_slip2_rate(m/s) time(s)
    time(h)\n')

if(sim0.useGPU==False):  #CPU case
    for i in range(totaloutputsteps):


        if(i==0):
            dttry=sim0.htry
        else:
            dttry=dtnext
        dttry,dtnext=sim0.simu_forward(dttry)
        year=sim0.time/3600/24/365
        if(i%10==0):
            print('iteration:',i)
            print('dt:',dttry,' max_vel:',np.max(np.abs(sim0.slipv)),' Seconds:',sim0.
                time,'  Days:',sim0.time/3600/24,
                'year',year)
            memory_info = process.memory_info()
            print(f"Memory usage: {memory_info.rss / (1024 ** 2)} MB")
        f.write('%d %f %.16e %.16e %f %f\n' %(i,dttry,np.max(np.abs(sim0.slipv1)),np.max
            (np.abs(sim0.slipv2)),sim0.time,sim0.time/3600.0/24.0))


```

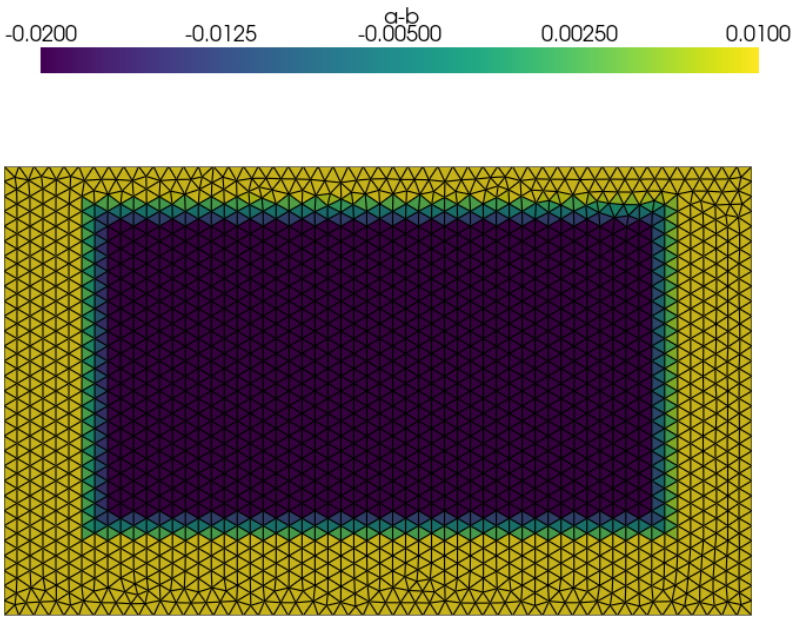


Figure 3: mesh model and spatial distribution of a-b.

```
30
31          if(sim0.Para0['outputvtk']=='True'):
32              outsteps=int(sim0.Para0['outsteps']) #output steps
33              if(i%outsteps==0):
34                  fname=directory+'/step'+str(i)+'.vtk'
35                  sim0.ouputVTK(fname)
36
37  end_time = time.time()
38  timetake=end_time-start_time
39  f.write('Program end time: %s\n'%str(datetime.now()))
40  f.write("Time taken: %.2f seconds\n"%timetake)
41
42  print('Program end time: %s\n'%str(datetime.now()))
43  print("Time taken: %.2f seconds\n"%timetake)
```

Listing 4: Calculating Qusi-Dynamic model by loop

Step 4: Visualize Results

The simulation generates VTK files, visualize them using PyVista.You can also output all results in your own format by accessing the member variables of sim0 (e.g. sim0.slip).

```
1  import pyvista as pv
2  import glob
3  import os
4
5  # set vtk file path
6  folder_path = "outvtk/"
7
```

```
vtk_files = glob.glob(os.path.join(folder_path, "*.vtk")) + glob.glob(os.path.join(folder_path, "*.vtu"))


#read state file for time show
def is_number(s):
    try:
        float(s)
        return True
    except ValueError:
        return False

def readstate(fname):
    f=open(fname,'r')
    K=0
    data0=[]
    for line in f:
        tem=line.split()

        if(len(tem)==6 and is_number(tem[0])==True):
            data0.append(np.array(tem).astype(float))
    return np.array(data0)
state=readstate('state.txt')
print(state.shape)

# # show one figure
plotter = pv.Plotter(off_screen=True)
mesh = pv.read(vtk_files[10])
slip_velocity = mesh['Slipv[m/s]']
# Apply logarithmic transformation (e.g., base 10)
# Add small constant (e.g., 1e-10) to avoid log(0) issues
log_slip_velocity = np.log10(slip_velocity)

# Assign the transformed scalars back to the mesh
mesh['Log_Slipv[m/s]'] = log_slip_velocity
scalar_range = (-12, 0)
scalar_bar_args={
        'title': 'log10(slip rate)[m/s]',
        'position_x': 0.22,
        'position_y': 0.85,
    }
plotter.add_mesh(mesh, scalars='Log_Slipv[m/s]', cmap="plasma", show_edges=True,clim=scalar_range,scalar_bar_args=scalar_bar_args)  # show shear stress

#plotter.camera_position =[(0, -5, 0), (0, 0, 0), (0, 0, 1)]
# plotter.add_scalar_bar(
#         title='log10(Slip rate)[m/s]',  # Label for the colorbar
#         position_x=0.25,         # Horizontal position (0 to 1, 0.25 centers it with width=0.5)
#         position_y=0.05,         # Vertical position (close to bottom)
#         width=0.5,               # Width of the colorbar (0 to 1, relative to plot)
#         height=0.1,              # Height of the colorbar
#         vertical=False           # Horizontal orientation
#     )
plotter.show(cpos='xz')

# create animation
outsteps=int(sim0.Para0['outsteps'])
plotter = pv.Plotter(off_screen=True)
plotter.open_gif('animation.gif')  # Initialize GIF output

for i, vtk_file in enumerate(vtk_files):
    mesh = pv.read(folder_path+'step'+str(i*outsteps)+'.vtk')
    slip_velocity = mesh['Slipv[m/s]']
    # Apply logarithmic transformation (e.g., base 10)
    # Add small constant (e.g., 1e-10) to avoid log(0) issues
    log_slip_velocity = np.log10(slip_velocity)

    # Assign the transformed scalars back to the mesh
    mesh['Log_Slipv[m/s]'] = log_slip_velocity
    plotter.clear()  # Clear previous mesh
    plotter.add_mesh(
        mesh,
```

```
        scalars='Log_Slipv[m/s]',
        cmap="plasma",
        show_edges=False,
        clim=scalar_range,
        scalar_bar_args=scalar_bar_args
    )

    timeyear=state[i*outsteps,-1]/365
    plotter.add_text(f"Time: {timeyear:.8f} yr", position="upper_left", font_size=14,
        color="white", shadow=True)
    plotter.camera_position = 'xz'
    plotter.write_frame()  # Write frame to GIF

plotter.close()
print('Video has been saved.')
```

Listing 5: Visualize Results

output:

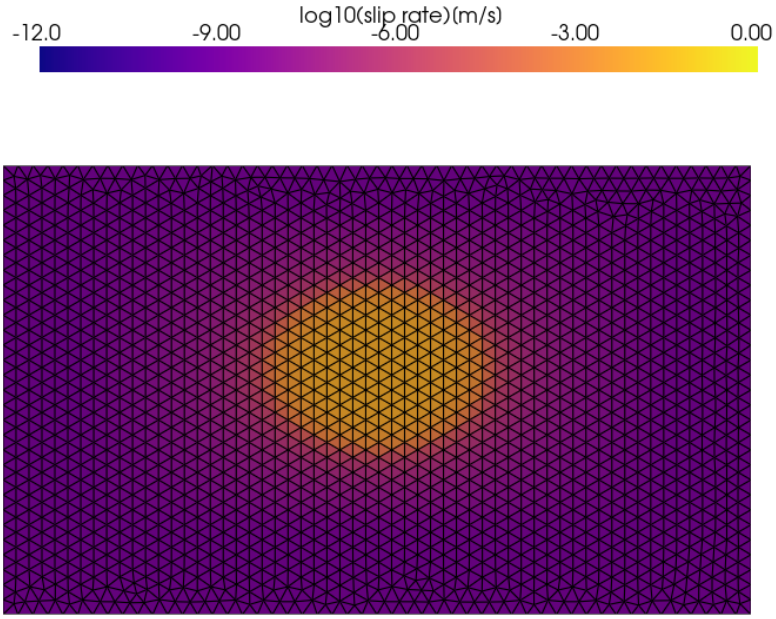


Figure 4: Snapshot of slip rate

GPU acceleration Install cupy if you want to use GPU acceleration, we recommened to use conda in prompt terminal (e.g. CUDA 11.8):conda install -c conda-forge cupy cudatoolkit=11.8

Notes: CuPy relies on NVIDIA CUDA for GPU computing, so the following components are required:

1. **NVIDIA GPU Driver**: Ensure your system has a CUDA-capable NVIDIA GPU (check compatibility at https://developer.nvidia.com/cuda-gpus). The driver version must support the CUDA

Toolkit version used by CuPy (e.g., CUDA 10.2–12.x for CuPy v13.x).

2. **CUDA Toolkit**: CuPy supports specific CUDA versions (check the latest compatibility at https://docs.cupy.dev/en Common versions: CUDA 11.8 or 12.2 are widely used.

The GPU operates in a similar way to the CPU, only the output is converted to numpy each time. Reading model and parameters is the same as CPU. **Note**: Make sure to set GPU:True in the parameter.txt

```
1
2  start_time=time.time()
3
4  totaloutputsteps=int(sim0.Para0['totaloutputsteps'])
5  directory='outvtk'
6  if not os.path.exists(directory):
7      os.mkdir(directory)
8
9  f=open('state.txt','w')
10 f.write('iteration time_step(s) maximum_slip1_rate(m/s) maximum_slip2_rate(m/s) time(s)
       time(h)\n')
11
12
13 if(sim0.useGPU==True): #GPU case
14     sim0.initGPUvariable()
15
16     for i in range(totaloutputsteps):
17         print('iteration:',i)
18         if(i==0):
19             dttry=sim0.htry
20         else:
21             dttry=dtnext
22         dttry,dtnext=sim0.simu_forwardGPU(dttry)
23
24         #transform form cupy to numpy data
25         sim0.slipv1=sim0.slipv1_gpu.get()
26         sim0.slipv2=sim0.slipv2_gpu.get()
27         sim0.slipv=sim0.slipv_gpu.get()
28         sim0.slip=sim0.slip_gpu.get()
29         sim0.Tt1o=sim0.Tt1o_gpu.get()
30         sim0.Tt2o=sim0.Tt2o_gpu.get()
31         sim0.Tt=sim0.Tt_gpu.get()
32         #sim0.state1=sim0.state1_gpu.get()
33         #sim0.state2=sim0.state2_gpu.get()
34         sim0.rake=sim0.rake_gpu.get()
35         year=sim0.time/3600/24/365
36         if(i%10==0):
37             print('dt:',dttry,' Seconds:',sim0.time,'  Days:',sim0.time/3600/24,'year',
                   year)
38             print(' max_vel1:',np.max(np.abs(sim0.slipv1)),' max_vel2:',np.max(np.abs(
                   sim0.slipv2)),' min_Tt1:',np.min(sim0.Tt1o),' min_Tt2:',np.min(sim0.Tt2o
                   ))
39         f.write('%d %f %.16e %.16e %f %f\n' %(i,dttry,np.max(np.abs(sim0.slipv1)),np.max
               (np.abs(sim0.slipv2)),sim0.time,sim0.time/3600.0/24.0))
40
41         memory_info = process.memory_info()
42         print(f"Memory usage: {memory_info.rss / (1024 ** 2)} MB")
43
44         if(sim0.Para0['outputvtk']=='True'):
45             outsteps=int(sim0.Para0['outsteps'])#output steps
46             if(i%outsteps==0):
47                 fname=directory+'/step'+str(i)+'.vtk'
48                 sim0.ouputVTK(fname)
```

Listing 6: conde to run GPU case

5.5.2 MPI-Based Execution on Linux

Step 1: Set Up Conda environment

1. Download latest Miniconda3, type: *wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh -O miniconda.sh*

2. Install Miniconda with the following command: *bash miniconda.sh -b -u -p ~/miniconda3*

Parameters:

-b: Batch mode, automatically accepts the license agreement. -u: Updates the installation if the directory already exists. -p ~/miniconda3: Specifies the installation directory (here, miniconda3 in the home directory).

3. After installation, initialize Conda for terminal use: *~/miniconda3/bin/conda init*

4. Close and reopen the terminal, or run: *source ~/.bashrc*

5. Verify the Installation,Check the Conda version: *conda –version*

6. After installation, create a new environment: *conda create -n PyQuake3D python=3.12*

7. Activate the environment: *conda activate PyQuake3D*

Notes:Dependencies: Ensure wget is installed. If not, install it:

sudo apt install wget # For Ubuntu/Debian

sudo yum install wget # For CentOS/RHEL

Step 2: Install Python and C++ Dependencies

1. Download from https://github.com/Computational-Geophysics/PyQuake3D
2. Install an MPI implementation and g++ compiler(including gcc and other related tools). Common options are:

- sudo apt update
- sudo apt install g++
- sudo apt install openmpi-bin libopenmpi-dev

1. Install PyQuake3D using pip: pip install -r requirements (make sure PyQuake3D environment is activated).
2. Navigate to the src directory, run **make** to build TDstressFS_C.so.The generated library is called by the Python script Hmatrix.py via dynamic loading (e.g., using ctypes).

Notes1:MPI-CPU version is recommended for use on Linux because it requires compiling a C dynamic library, which used for green's function calculation acceleration. Compiling C dynamic libraries (DLLs) on Windows can be less convenient compared to Linux or other Unix-like systems for several reasons, stemming from differences in tooling, ecosystem, and system design.

Notes2:PyQuake3D supports Python 3.8 and above, so there is no need to specify a specific version when installing the library.

Step 3: Examples for MPI-based backend

```
1
2 import readmsh
3 import numpy as np
4 import sys
5 import matplotlib.pyplot as plt
```

```
import QDsim
from math import *
import time
import argparse
import os
import psutil
from datetime import datetime
from mpi4py import MPI
import config
from config import comm, rank, size

import sys
#old_stdout = sys.stdout
#log_file = open("message.log", "w")
#sys.stdout = log_file

if __name__ == "__main__":

    sim0=None
    fnamePara=None
    jud_coredir=None
    blocks_to_process=[] #Save block submatirx from Hmatrix
    if(rank==0):
        try:

            parser = argparse.ArgumentParser(description="Process some files and enter interactive mode.")
            parser.add_argument('-g', '--inputgeo', required=True, help='Input msh geometry file to execute')
            parser.add_argument('-p', '--inputpara', required=True, help='Input parameter file to process')

            args = parser.parse_args()

            fnamegeo = args.inputgeo
            fnamePara = args.inputpara

        except:

            fnamegeo='examples/BP5-QD/bp5t.msh'
            fnamePara='examples/BP5-QD/parameter.txt'




        print('Input msh geometry file:',fnamegeo, flush=True)
        print('Input parameter file:',fnamePara, flush=True)

        nodelst,elelst=readmsh.read_mshV2(fnamegeo)
        config.readPara0(fnamePara)
        sim0=QDsim.QDsim(elelst,nodelst,fnamePara)


        #Determine whether Hmatrix has been calculated. If it has been calculated, read it directly
        jud_coredir,blocks_to_process=sim0.get_block_core()
        print('jud_coredir',jud_coredir) #if saved corefunc
        if(jud_coredir==False):
            print('Start to calculate Hmatrix...')
        else:
            print('Hmatrix reading...')



    print('rank:',rank)
    # bcast parameters to all ranks
    fnamePara = comm.bcast(fnamePara, root=0)
    config.readPara0(fnamePara)

    sim0 = comm.bcast(sim0, root=0)
    jud_coredir = comm.bcast(jud_coredir, root=0)

```

```
75
76
77      if(jud_coredir==False):#Calculate green functions and compress in Hmatrix
78          #sim0.local_blocks=sim0.tree_block.parallel_traverse_SVD(sim0.Para0['Corefunc
                directory'],plotHmatrix=sim0.Para0['Hmatrix_mpi_plot'])
79          if(rank==0):
80              #Assign tasks for calculating green functions
81             sim0.tree_block.master(sim0.Para0['Corefunc directory'],blocks_to_process,
                   size-1,save_corefunc=sim0.save_corefunc)
82          else:
83              #Calculat green functions
84             sim0.tree_block.worker()
85          #sim0.tree_block.master_scatter(sim0.Para0['Corefunc directory'],
                blocks_to_process,size)
86          '''Assign forward modelling missions for each rank with completed blocks
                submatrice'''
87          sim0.local_blocks=sim0.tree_block.parallel_block_scatter_send(sim0.tree_block.
                blocks_to_process,plotHmatrix=sim0.Para0['Hmatrix_mpi_plot'])
88      else:
89          '''Assign forward modelling missions for each rank with completed blocks
                submatrice'''
90          sim0.local_blocks=sim0.tree_block.parallel_block_scatter_send(blocks_to_process,
                plotHmatrix=sim0.Para0['Hmatrix_mpi_plot'])
91
92      if(sim0.Ifdila==True):
93          sim0.local_index=sim0.tree_block.parallel_cells_scatter_send()
94
95
96
97
98      start_time = MPI.Wtime()
99      totaloutputsteps=int(sim0.Para0['totaloutputsteps']) #total time steps
100     for i in range(totaloutputsteps):
101     #for i in range(0):
102         sim0.step=i
103         if(i==0):#inital step length
104             dttry=sim0.htry
105         else:
106             dttry=dtnext
107         dttry,dtnext=sim0.simu_forward_mpi(dttry) #Forward modeling
108         #sim0.simu_forward(dttry)
109         if(rank==0):
110             year=sim0.time/3600/24/365
111             #if(i%10==0):
112             print('iteration:',i, flush=True)
113             print('dt:',dttry,' max_vel:',np.max(np.abs(sim0.slipv)),' min_vel:',np.min(
                    np.abs(sim0.slipv)),' Porepressure max:',np.max(sim0.P),' Porepressure
                    min:',np.min(sim0.P),' dpdt_max:',np.max((sim0.dPdt0)),' dpdt_min:',np.
                    min((sim0.dPdt0)),' Seconds:',sim0.time,'  Days:',sim0.time/3600/24,
                'year',year, flush=True)
114
115
116             #Save slip rate and shear stress for each iteration
117             outsteps=int(sim0.Para0['outsteps'])
118             directory='out_vtk'
119             if not os.path.exists(directory):
120                 os.mkdir(directory)
121
122                 #output vtk
123                 if(sim0.Para0['outputvtk']=='True'):
124                     #print('!!!!!!!!!!!!!!!!!!!!!!!!!!!')
125                     fname=directory+'/step'+str(i)+'.vtk'
126                     sim0.ouputVTK(fname)
127
128
129
130     end_time = MPI.Wtime()
131
132     if rank == 0:
133
134         print(f"Program run time: {end_time - start_time:.6f} sec")
135
```

Listing 7: Examples for MPI-based backend

Step 4: The visualization of the MPI-based backend is consistent with that of a single CPU/GPU

6 Code Structure and File Description

The PyQuake3D source code is organized to reflect its modular architecture and hybrid parallel computing design (Figure 5). All core Python modules are located in the `src` directory, while model configurations and parameter files are provided in the `examples` folder. The `examples` directory contains predefined models, including the `BP5-QD` benchmark based on the Southern California Earthquake Center's SEAS community validation project[1].

The PyQuake3D codebase is structured around distinct computational tasks:

- `main.py`: Entry point for running the quasi-dynamic simulations. It handles the model setup, time integration, and output generation. Users may customize this script for advanced diagnostics or automated post-processing.
- `QDsim.py` and `QDsim_gpu.py` : Defines the `QDsim` class, which encapsulates the governing equations, numerical integrator, and interface to the selected Green's function backend (dense in `QDsim_gpu.py` or hierarchical in `QDsim.py`).
  - `QDsim.Init_condition()`: Setting initial condition of fault model.
  - `QDsim.simu_forward()`: Model forward calculation.
- `DH_Greenfunction.py` and `SH_Greenfunction.py`: Compute the displacement and stress Green's functions for homogeneous elastic half-space and full-space media, respectively.
- `TDstressFS_C.cpp`: Compute the stress Green's functions for homogeneous elastic half-space and full-space media using C++ language.
- `Hmatrix.py`: Constructs and applies the hierarchical matrix representation based on Adaptive Cross Approximation (ACA), optimized for distributed-memory parallelism using `mpi4py`.
  - `QDsim.create_recursive_blocks()`: Recursively build the blocktree.
  - `QDsim.createHmatrix()`: Build the Hmatrix.
  - `tree_block.master()`: Dynamically allocate MPI tasks
  - `tree_block.worker()`: Excute MPI tasks of calculating stress green's functions.
  - `tree_block.tree_block.parallel_block_scatter_send()`:Distribute sub-matrices of the fully built Hmatrix to each process.
- `Readmsh.py`: Parses unstructured mesh files (in `.msh` format) and imports model geometry, fault segmentation, and material parameters.

7 Parameters setting

7.1 Table parameters setting

The simulation parameters are implemented by modifying the *parameter.txt* file, rather than by changing the source code. The input variable list is in *Table* 2. If $InputHetoparamter$ in *Table* 2 is True, heterogeneous stress and friction parameters are imported from external files. The external filename is defined in *parameter.txt* and must remain in the same directory as *parameter.txt*. In this case, you only need to appropriately set parameter of *Table* 2 *Table* and *Table* 6. Otherwise, you need to appropriately set the parameters of stress and frictional initial condition shown in *Table* 4 and *Table* 5. **??** must be set up if fluid induced dilatancy and compaction is considered.

[1] `https://strike.scec.org/cvws/seas/download/`

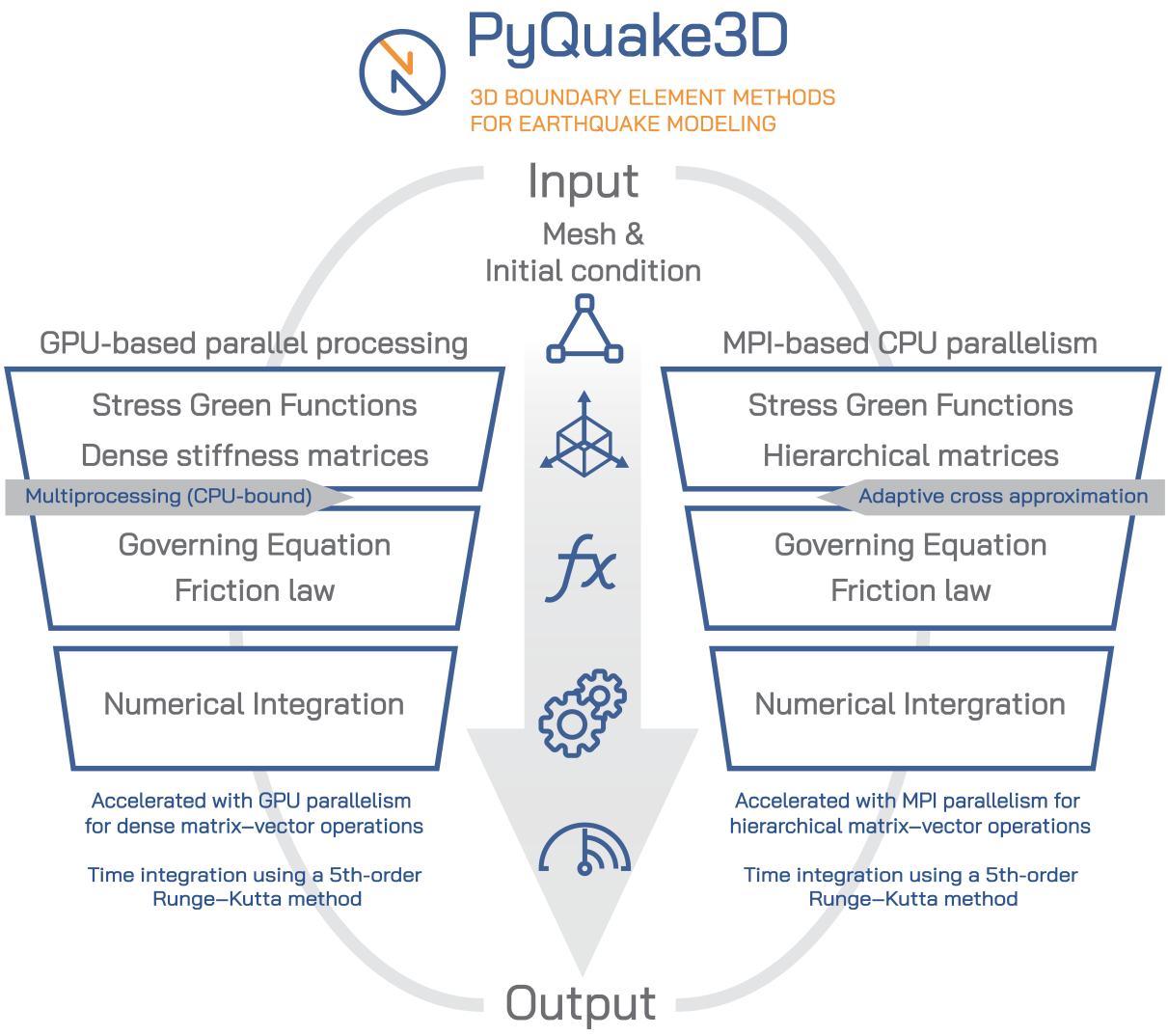


Figure 5: Overview of the PyQuake3D computational framework. The workflow includes input setup, solver implementation on GPU or CPU architectures, and output diagnostics. The GPU version uses dense matrix–vector operations accelerated with CUDA, while the CPU version applies hierarchical matrix compression and MPI-based parallelism.

External file format

In addition to *parameter.txt*, the program requires external the .msh file for geometry, as well as $Inputparameterfiles$ if `Input Hetoparamter` is set to True. The filename for $Inputparameterfiles$ is defined within *parameter.txt*.

The *.msh* file contains necessary mesh data such as node coordinates, element numbers, and other relevant information. It is exported from *Gmsh* software. Please ensure selecting the $Version2/ASCII$ format to match the code's reading program.

When `InputHetoparameter` == `True`, the initial condition can be imported externally (via the Input-parameter file):

```
rake(0) a(0) b(0) dc(0) f0(0) tau(0) sigma(0) vel(0) taudot(0) sigdot(0) InitP(0)
rake(1) a(1) b(1) dc(1) f0(1) tau(1) sigma(1) vel(1) taudot(1) sigdot(1) InitP(1)
...
```

Table 2: General Parameters

Parameter	Default	Description
`Corefunc directory`		The storage path for the kernel function matrix composed of stress Green's functions
`Node_order`	False	If 'True', the node order of the triangular element is clockwise
`save Corefunc`	False	If 'True', save corefuns Save the kernel function so that it does not need to be recalculated for the next time
`Scale_km`	True	If 'True', the coordinates will be scaled up by a factor of 1000, meaning they are modeled in kilometers, which is applicable to natural earthquakes; otherwise, the coordinates remain unchanged, meaning they are modeled in meters, which is applicable to laboratory earthquakes.
`Hmatrix_mpi_plot`	False (Only availible for MPI verison)	If 'True', draw the Hmatirx structure diagram, with different colors representing the sub-matrices calculated by different processes.
`Using C++ green function`	True (Only availible for MPI verison)	If 'True'. Using C++ to calculate Green functions, else using python.Only availible for MPI verison.
`GPU`	False (Only availible for CPU/GPU verison.)	If 'True', Using GPU parallel acceleration (cupy)
`Max thread workers`	50 (Only availible for CPU/GPU verison.)	The number of processors in ProcessPoolExecutor to parallelize Green's function calculations.
`Batch_size`	1000 (Only availible for CPU/GPU verison.)	The number of batches in ProcessPoolExecutor to parallelize Green's function calculations.
`Input Hetoparamter`	False	If 'True', the heterogeneous stress and friction parameters are imported from external files.
`Inputparamter file`		The file name of imported heterogeneous stress and friction parameters
`Lame constants`	0.32×10^{11} Pa	The first Lame constant
`Shear modulus`	0.32×10^{11} Pa	Shear modulus
`Rock density`	2670 kg/m^3	Rock mass density
`Reference slip rate`	1×10^{-6}	Reference slip rate
`Reference friction coefficient`	0.6	Reference friction coefficient
`Plate loading rate`	1×10^{-9}	Plate loading rate

The total number of rows in the `InputHetoparameter` file is equal to the number of cells. The first column represents the rake angle, which varies in the calculation process. Columns 2, 3, 4, and 5 represent parameters related to the rate-state friction law: frictional parameters a, frictional parameters a, characteristic slip dc, reference friction coefficient. Columns 6 and 7 denote shear and normal tractions, respectively. Column 8 is the initial slip rate, and the column 9 and 10 represent the loading rates for

Table 3: Fluid Parameters

Parameter	Default	Description
`If Dilatancy`	False	If 'True', Coupled solve the fluid diffusion equation and quasi-dynamic model
`Dilatancy coefficient`	3×10^{-4}	Quantifies the volume change (dilation or contraction) in a porous medium due to shear deformation
`Hydraulic diffusivity`	1×10^{-5} m^2/s	The rate at which pore fluid diffuses cross the fault
`Actively shearing zone thickness`	3×10^{-3} m	The thickness of the actively shearing zone represents the width of the region undergoing significant shear deformation
`Effective compressibility`	8×10^{-11} Pa^{-1}	The volumetric strain response of the porous medium to changes in pore pressure
`Constant porepressure`	2 MPa	Constant porepressure
`Initial porepressure`	2 MPa	Initial porepressure

shear and normal tractions. Note that this loading refers to additional loading other than the slip deficit rate loading, with a default value of 0. Column 11 is the initial pore pressure, which is optional.

8 Frictional heterogeneous model examples

Let's consider the heterogeneous frictional model (HF) to compute the earthquake cycle of strike-slip fault using MPI_basd PyQuake3D (Figure 6). Listing 8 shows the entire input file required to describe the simulation. Note how the code is called, with the command

```
mpirun -np 12 python src/main.py -g examples/HF-model/HFmodel.msh -p examples/HF-model/parameter.txt
```

In this case, the program is called with 12 CPUs for MPI parallel computing. After the program starts, it outputs screen information as shown in Listing 9. The screen output consists of three main parts:

1. **Basic Information**: This section provides a summary of the key model settings, allowing users to verify the correctness of the configuration.
2. **MPI-Based Dynamic Process Allocation**: This section displays the status of each process during the parallel computation of Green's functions, offering insight into how tasks are distributed across different MPI ranks.
3. **Numerical Integration of the Governing Equations**: Upon completion of the Green's function computation, the program proceeds to solve the differential equations using the Runge-Kutta method. It outputs the maximum slip rate, current time step size, and related parameters, while also beginning to save the simulation results.

The file output including cell information, the slip and slip rate distribution, stress and state etc are exported in the .vtk format in `outputvtk` directory , which is a standard 3-D geometry format that can be read with such visualization tools as Paraview (http://www.paraview.org). We performed a 200-year simulation of a frictionally heterogeneous fault model using over 350,000 time steps (Figure 7). Computations were carried out in parallel using MPI on 14 cores of a 12th Gen Intel® Core™ i9-12900K processor, requiring approximately 10 days. The model contains more than 240,000 boundary

Table 4: Stress and Frition Settings

Parameter	**Default**	**Description**
`Half space`	False	If 'True', calculating half-space green's functions
`Fix_Tn`	True	If 'True', fixed the normal stress
`Vertical principal stress (ssv)`	1.0	The vertical principal stress scale: the real vertical principal stress is obtained by multiplying the scale and the value
`Maximum horizontal principal stress (ssh1)`	1.6	Maximum horizontal principal stress scale.
`Minimum horizontal principal stress(ssh2)`	0.6	Minimum horizontal principal stress scale
`Angle between ssh1 and X-axis`	30°	Angle between maximum horizontal principal stress and X-axis.
`Vertical principal stress value`	50 MPa	Vertical principal stress value
`Vertical principal stress value varies with depth`	True	If True, Vertical principal stress value varies with depthand the horizontal principal stress value also changes with depth simultaneously
Turnning depth	5000 m	If Vertical principal stress value varies with depth is true, starting at this depth, the stress no longer changes with depth
`Shear traction solved from stress tensor`	False	If 'True', the non-uniform shear stress is projected onto the non-planar fault surface by the stress tensor
`Rake solved from stress tensor`	False	If 'True', the non-uniform rakes are solved from the stress tensor.
`Fix_rake`	30°	If 'True', Set fixed rakes if 'Rake solved from stress tensor' is 'False'.
`Widths of VS region`	5000 m	The width of the VS region near boundary.
`Widths of surface VS region`	2000 m	Widths of VS region near free surface
`Transition region from VS to VW region`	3000 m	Transition region width from VS to VW region

elements and consumed over 60 GB of memory during execution. With a uniform cell size of 120 m, the critical nucleation size and cohesive zone length within the VW patches are approximately 1773 m and 374 m, respectively, allowing accurate simulation of events down to M_w 4.6 (Figure 6b). PyQuake3D successfully captures a broad range of fault behaviors, including the progressive buildup and abrupt release of shear stress (Figure 7h); aseismic creep and afterslip; episodic slow-slip events without seismic radiation; and recurring microseismic activity surrounding major events. These results highlight the ability of PyQuake3D to model multi-scale interactions between seismic and aseismic slip in geologically motivated settings.

Table 5: Nucleation and Friction Setting

Parameter	Default	Description
`Set_nucleation`	False	If True, sets a patch whose shear stress and sliding rate are significantly greater than the surrounding area to meet the nucleation requirements.
`Radius of nucleation`	8000 m	The radius of the nucleation region
`Nuclea_posx`	34000 m	Posx of Nucleation
`Nuclea_posy`	15000 m	Posy of Nucleation
`Nuclea_posz`	-15000 m	Posz of Nucleation
`Rate-and-state parameters a in VS region`	0.04	Rate-and-state parameters a in VS region
`Rate-and-state parameters b in VS region`	0.03	Rate-and-state parameters a in VS region
`Characteristic slip distance in VS region`	0.13 m	Characteristic slip distance in VS region
`Rate-and-state parameters a in VW region`	0.004	Rate-and-state parameters a in VW region
`Rate-and-state parameters a in VW region`	0.03	Rate-and-state parameters a in VW region
`Characteristic slip distance in VW region`	0.13 m	Characteristic slip distance in VW regioN
`Rate-and-state parameters a in nucleation region`	0.004	Rate-and-state parameters a in nucleation region
`Rate-and-state parameters a in nucleation region`	0.03	Rate-and-state parameters a in nucleation region
`Characteristic slip distance in nucleation region`	0.14 m	Characteristic slip distance in nucleation regioN
`Initial slip rate in nucleation region`	3e-2	Initial slip rate in nucleation region
`ChangefriA`	False	If True, a changes gradually b remains unchanged, vice versa
`Initlab`	True	If True, setting random non-uniform normal stress

```
#Files and cells
Corefunc directory: core_HF
save Corefunc: True
Node_order: False
Scale_km: True

#Only availible for mpi version
```

Table 6: Output Setting

Parameter	Default	Description
`totaloutputsteps`	2000	The number of calculating time steps.
`outsteps`	50	The time step interval for outputting the VTK files.
`outputSLIPV`	False	If True, output slip rate for each step.
`outputTt`	False	If True, output shear stress for each step.
`outputstv`	True	If True, the VTK files will be saved in outvtk directroy.
`outputmatrix`	False	If True, the matrix format txt files will be saved in out directroy.

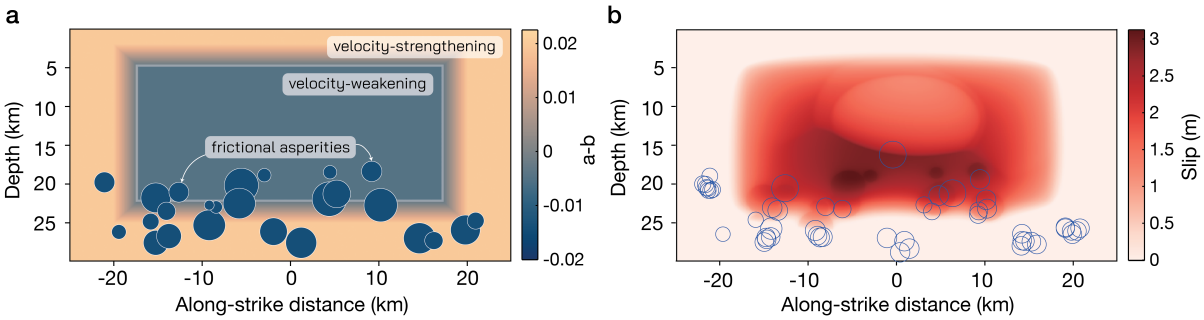


Figure 6: Frictional heterogeneity and rupture behavior in the fault model. (**a**) Initial distribution of the friction parameter $a-b$, with a central velocity-weakening region bounded by a velocity-strengthening margin. Dark blue circles mark strongly velocity-weakening patches representing small-scale asperities. (**b**) Coseismic slip distribution during the first dynamic rupture, followed by the first slow slip event. Blue circles indicate all simulated microseismic events over the 200-year period.

```
Hmatrix_mpi_plot:False
Using C++ green function:True

#Only availible for single GPU/CPU version
Max_thread workers: 50
Batch_size: 1000
GPU:False


#Property
Lame constants: 32038120320.0
Shear modulus: 32038120320.0
Rock density: 2670
InputHetoparamter: True
Inputparamter file: initcondion.txt

#Fluid
If Dilatancy:False
Dilatancy coefficient: 3e-4
Hydraulic diffusivity: 1e-5
Actively shearing zone thickness: 3e-3
Effective compressibility: 8e-11
Constant porepressure: 2
Initial porepressure: 2

#Stress
Half space: True
Fix_Tn:True
Vertical principal stress: 1.0
Maximum horizontal principal stress: 1.6
Minimum horizontal principal stress: 0.6
Angle between ssh1 and X-axis: 90
```

```
Vertical principal stress value: 50
Vertical principal stress value varies with depth: True
Turnning depth: 5000
Normal traction solved from stress tensor: False
Shear traction solved from stress tensor: False
Rake solved from stress tensor: False
Fix_rake: 0
Widths of VS region: 5000
Widths of surface VS region: 2000
Transition region from VS to VW region: 3000


#Friction
Reference slip rate: 1e-6
Reference friction coefficient: 0.6
Rate-and-state parameters a in VS region: 0.0185
Rate-and-state parameters b in VS region: 0.001
Characteristic slip distance in VS region: 0.015
Rate-and-state parameters a in VW region: 0.0185
Rate-and-state parameters b in VW region: 0.024
Characteristic slip distance in VW region: 0.015
Rate-and-state parameters a in nucleation region: 0.01
Rate-and-state parameters b in nucleation region: 0.0185
Characteristic slip distance in nucleation region: 0.004
Initial slip rate in nucleation region: 3e-2
Plate loading rate: 1e-9
ChangefriA:False
Initlab: False

#nucleartion
Set_nucleation: False
Nuclea_posx: 34000
Nuclea_posy: 15000
Nuclea_posz: -15000
Radius of nucleation: 8000

#output
outputSLIPV: False
outputTt: False
totaloutputsteps: 1000
outsteps: 200
outputvtk: True
outputmatrix: False
```

Listing 8: Input of Heterogeneous frictional model

```
# ---------------------------------------------------------------------------
# PyQuake3D: Boundary Element Method to simulate sequences of earthquakes and aseismic
    slips
# * 3D non-planar quasi-dynamic earthquake cycle simulations
# * Support for Hierarchical matrix storage and calculation
# * Parallelized with MPI (12 cpus)
# * Support for rate-and-state aging friction laws
# * Supports output to VTK formats
# * ---------------------------------------------------------------------
Cs 3464.0
First Lame constants 32038120320.0
Shear Modulus 32038120320.0
Youngs Modulus 80095300800.0
Poissons ratio 0.25
a,b,L: 0.01 0.025 0.001
maximum element size 161.5330463540536
average elesize 120.13229265066029
Critical nucleation size 118.21362089765466
Cohesive zone: 23.938258231775073
Start calculating Hmatrix...
Worker 0 receive the task result 1407, size = (32, 30)
Master: Worker 3 finish task 1407 , size = (32, 30)
Master: complete 1397/ 213580
Master: assign task 1408 to Worker 3, size = (25, 28)
[Worker 3] waiting for task...
rank 3,start fullmatrix calc...
rank 3,Full matrix calculation task success,size(25, 28),Time takes 0.0179865360 seconds
```

```
Worker 3 send the the task result 1408, size = (25, 28)
[Worker 3] waiting for task...
Worker 0 receive the task result 1408, size = (25, 28)
Master: Worker 3 finish task 1408 , size = (25, 28)
Master: complete 1398/ 213580
Master: assign task 1409 to Worker 3, size = (25, 30)
rank 3,start fullmatrix calc...
rank 3,Full matrix calculation task success,size(25, 30),Time takes 0.0158853531 seconds
Worker 3 send the the task result 1409, size = (25, 30)
Worker 0 receive the task result 1409, size = (25, 30)
Master: Worker 3 finish task 1409 , size = (25, 30)
Master: complete 1399/ 213580
Master: assign task 1410 to Worker 3, size = (112, 114)
[Worker 3] waiting for task...
rank 3,start ACA...
rank 8,ACA of task success,size(476, 465),Time takes 2.8612349033 seconds
Worker 8 send the the task result 1375, size = (476, 465)
[Worker 8] waiting for task...
iteration: 0
dt: 0.001  max_vel: 9.999999717199876e-10  Seconds: 0.001   Days: 1.1574074074074074e-08
     year 3.1709791983764586e-11
iteration: 20
dt: 3.3252567300796505  max_vel: 9.999999715603074e-10  Seconds: 9.973770190238952
    Days: 0.00011543715497961749 year 3.162661780263493e-07
iteration: 40
dt: 11057.332320940008  max_vel: 9.999994415927274e-10  Seconds: 33171.99496282003
    Days: 0.38393512688449105 year 0.0010518770599575098
iteration: 60
dt: 903248.7103340095  max_vel: 9.999207208431887e-10  Seconds: 8690287.646132415   Days
    : 100.58203294134739 year 0.27556721353793806
iteration: 80
dt: 607203.8217271981  max_vel: 1.0574466703090563e-09  Seconds: 30915432.66035066
    Days: 357.8175076429475 year 0.9803219387478013
```

Listing 9: State output of HF-model during program execution

9 Stop Control

The simulation stops when any of the following is satisfied.

- The time-step reaches the *totaloutputsteps*.
- RungeKutta_solve_Dormand_Prince iteration attempted 20 times without meeting the error tolerance requirements.

10 Visualization

The current program supports output in VTK format and numpy,and the vtk can be used for 3D visulization in ParaView. We encourage users to develop their own code within the main function to flexibly output data using sim0 object from QDsim class. The sim0 object contains all variables required for the simulation, and you can access them by using sim0.variables.

11 License

This project is licensed under the MIT License.

12 Acknowledgments

We acknowledge the support and feedback provided by the broader scientific community and all contributors to this work. Development of the Python-based BEM algorithm was informed by the HBI code introduced in:

Ozawa, S., Ida, A., Hoshino, T., & Ando, R. (2023). Large-scale earthquake sequence simulations on

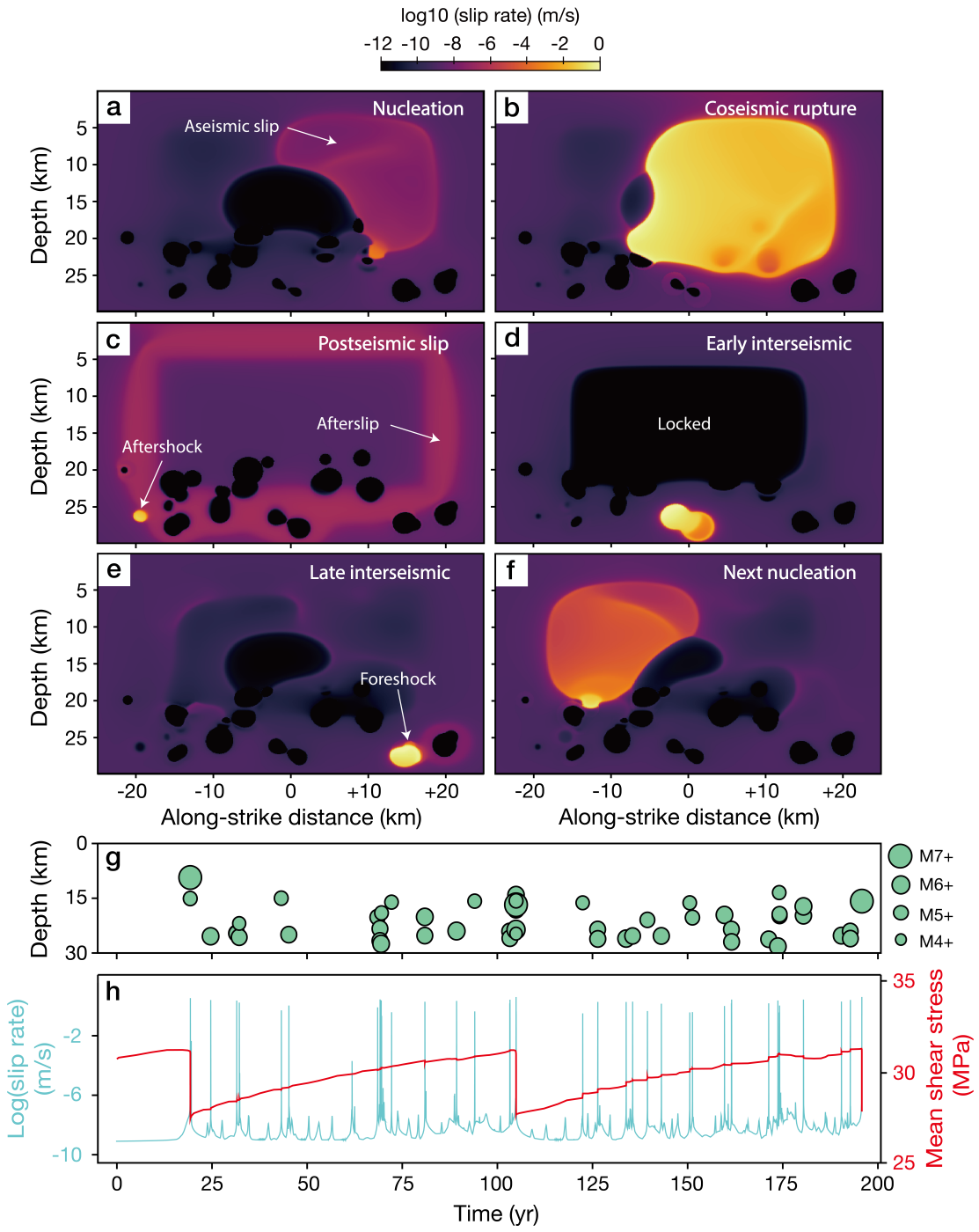


Figure 7: Spatiotemporal variations in slip rate and seismicity in the frictionally heterogeneous model. Panels (a–f) show slip rate evolution during key phases of the seismic cycle. (g) Spatiotemporal variations in event locations and magnitudes. (h) Time evolution of maximum slip rate and average shear stress.

3-D non-planar faults using the boundary element method accelerated by lattice H-matrices. *Geophysical Journal International*, 232(3), 1471–1481.

The implementation of the stress Green's functions builds on the MATLAB routines from:
Nikkhoo, M., & Walter, T. R. (2015). Triangular dislocation: an analytical, artefact-free solution. *Geophysical Journal International*, 201(2), 1119–1141.

We sincerely thank Ryosuke Ando and So Ozawa for their valuable guidance in the development of the code. We also thanks Steffen Börm for his assistance with H-matrix implementation and T. Ben Thompson for his assistance with the H-matrix compression via Adaptive Cross Approximation (ACA).

References

Mario Bebendorf. Approximation of boundary element matrices. *Numerische Mathematik*, 86:565–589, 2000.

Steffen Börm, Lars Grasedyck, and Wolfgang Hackbusch. Introduction to hierarchical matrices with applications. *Engineering analysis with boundary elements*, 27(5):405–422, 2003.

James H Dieterich. Modeling of rock friction: 2. simulation of preseismic slip. *Journal of Geophysical Research: Solid Earth*, 84(B5):2169–2175, 1979.

Lars Grasedyck. Adaptive recompression of-matrices for bem. *Computing*, 74:205–223, 2005.

Elías Rafn Heimisson. Crack to pulse transition and magnitude statistics during earthquake cycles on a self-similar rough fault. *Earth and Planetary Science Letters*, 537:116202, 2020.

Mehdi Nikkhoo and Thomas R Walter. Triangular dislocation: an analytical, artefact-free solution. *Geophysical Journal International*, 201(2):1119–1141, 2015.

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, Cambridge, 3rd edition, 2007.

James R Rice. Spatio-temporal complexity of slip on a fault. *Journal of Geophysical Research: Solid Earth*, 98(B6):9885–9907, 1993.

Sergej Rjasanow and Olaf Steinbach. Approximation of boundary element matrices. *The Fast Solution of Boundary Integral Equations*, pages 101–130, 2007.

Andy Ruina. Slip instability and state variable friction laws. *Journal of Geophysical Research: Solid Earth*, 88(B12):10359–10370, 1983.