Article

Meta-MR: The meta-methodology and the web platform for Mendelian Randomization analysis

WenJun Zhang

School of Life Sciences, Sun Yat-sen University, Guangzhou 510275, China; International Academy of Ecology and Environmental Sciences, Hong Kong, China

E-mail: zhwj@mail.sysu.edu.cn, wjzhang@iaees.org

Received 26 Feruary 2025; *Accepted 1 June 2026*; *Published online 23 June 2026*; *Published 1 June 2027*

Abstract

Mendelian randomization (MR) has emerged as a powerful epidemiological strategy for inferring causal relationships from observational data by leveraging genetic variants as instrumental variables. The fixed nature of genotypes renders them largely immune to reverse causation and confounding, making MR particularly valuable when randomized controlled trials are infeasible. However, the growing diversity of MR methodologies and the often-inconsistent findings across different analytical approaches pose significant challenges for researchers seeking robust causal evidence. This study presents a comprehensive mathematical formalization of the major MR algorithms and introduces Meta-MR, an integrated web-based computational platform for systematic MR analysis. I detail the mathematical foundations of twelve single-variable MR methods, including the Inverse-Variance Weighted method, Weighted Median, MR-Egger, Maximum Likelihood, Mode-Based Estimation, Contamination Mixture Model, Heterogeneity Penalization, Lasso-based selection, Debiased IVW, Penalized IVW, Constrained Maximum Likelihood, and Leave-One-Out diagnostics. Additionally, I formalize seven multivariable MR approaches: Multivariable IVW, Multivariable Median, Multivariable Egger, Multivariable Lasso, Multivariable Constrained Maximum Likelihood, Multivariable Generalized Method of Moments, and Multivariable Principal Component GMM. The Meta-MR web platform, implemented entirely in JavaScript with HTML and CSS, addresses a critical gap in existing MR software by incorporating meta-analytic pooling across algorithm-specific estimates. The platform enables researchers to: (1) generate harmonized MR input data through direct JSON input, file upload, or AI-assisted GWAS data fetching; (2) execute user-selected MR methods dynamically; (3) assess between-algorithm heterogeneity using Cochran's Q test; (4) obtain pooled causal estimates with proper uncertainty quantification under both homogeneous and heterogeneous models; (5) generate interactive forest plots for visualizing SNP-level and method-level estimates; and (6) export comprehensive results to Word-compatible documents. The meta-pooling algorithm employs inverse-variance weighting to compute combined estimates, with the DerSimonian-Laird estimator for between-algorithm variance when significant heterogeneity is detected. This approach mitigates the impact of potentially biased or inefficient individual algorithms, providing more robust and reliable causal inferences. The significance of this work lies in its dual contribution: rigorous mathematical exposition of MR algorithms and a practical, accessible implementation that promotes evidence synthesis in precision medicine. By formalizing the underlying algorithms and enabling systematic pooling, Meta-MR facilitates more reliable causal insights for disease etiology and drug discovery.

Keywords Mendelian randomization; causal inference; instrumental variables; SNPs; algorithms; algorithm pooling; meta-analysis; AI; web platform; computational tool; JavaScript.

Network Biology
ISSN 2220-8879
URL: http://www.iaees.org/publications/journals/nb/online-version.asp
RSS: http://www.iaees.org/publications/journals/nb/rss.xml
E-mail: networkbiology@iaees.org
Editor-in-Chief: WenJun Zhang
Publisher: International Academy of Ecology and Environmental Sciences

1 Introduction

1.1 Causal inference

Causal inference represents a cornerstone of scientific research, enabling the identification of cause-and-effect relationships amid complex data. Its importance stems from the need to distinguish true causal links from mere associations, informing interventions in fields like epidemiology, economics, and biology. Due to confounding factors, reverse causation, and measurement errors, numerous methods have been developed to robustly infer causality. These range from randomized controlled trials to observational techniques like instrumental variable analysis.

In my earlier studies, I proposed three tailored methods for causal inference across variable types: Boolean, nominal (categorical), and linearly correlated (scale or interval) variables (Zhang, 2021a-c). For linearly correlated variables, a statistical simulation and regression approach was developed to generate artificial datasets with predefined causality. Simulations involved creating paired variables with imposed directional effects, followed by regression modeling to extract patterns. Rules derived from these analyses—such as directional coefficient stability under perturbation—were then applied to real data for causality determination (Zhang, 2021b). For nominal variables, a simulation-based method generated synthetic categorical data with known causal directions. Analysis revealed probabilistic dependencies, leading to a law of asymmetric association strength. Causality was inferred by applying randomization tests to assess the statistical significance of correlation measures (e.g., chi-square or Cramér's V), followed by simulations to confirm direction and significance (Zhang, 2021c). In the case of Boolean variables, simulations produced datasets of binary pairs with designated independent and dependent roles. Patterns in joint distributions informed a causality law based on conditional probabilities. For observed data, randomization tested the significance of Boolean correlations (e.g., point, quartile, or Jaccard indices). Causal direction was assigned per the law, with simulations validating overall significance (Zhang, 2021a).

These methods emphasize simulation-driven validation, offering flexible tools for natural data while addressing limitations of traditional approaches.

1.2 Principles, methods, and applications of Mendelian Randomization

Mendelian Randomization (MR) leverages genetic variants as instrumental variables to infer causal effects, mimicking randomization in observational studies. Its significance lies in overcoming confounding and reverse causation, providing robust evidence for lifestyle, genetic, and pharmacological impacts on diseases. Widely applied in epidemiology, pharmacology, and public health, MR evaluates exposures like nutrients or behaviors on outcomes such as cardiovascular disease, cancer, and metabolic disorders, identifying novel risk factors and intervention targets.

The core principle relies on three assumptions: (1) the instrument (e.g., genetic variant) strongly associates with the exposure; (2) it remains independent of confounders; and (3) it affects the outcome solely via the

exposure (no pleiotropy). Methods include one-sample (individual-level data) and two-sample MR (summary statistics from separate studies), with estimators like inverse-variance weighted (IVW), MR-Egger (pleiotropy-robust), weighted median, and MR-PRESSO for outlier detection. Advances incorporate multivariable MR for multiple exposures and bidirectional designs to probe directions.

In nutrition and oncology, single-nucleotide polymorphisms (SNPs) linked to micronutrient levels serve as instruments. Kim et al. (2023) used two-sample MR with SNPs for 14 micronutrients (vitamins and minerals) to assess effects on 22 cancers from UK Biobank and FinnGen meta-analyses. Findings indicated higher genetically predicted magnesium levels increased breast cancer risk, while vitamin B12 elevated colorectal cancer risk. This aligns with prior MR studies linking micronutrients to cancer risks (Fu et al., 2021; Papadimitriou et al., 2021; Yuan et al., 2022).

In mental health, bidirectional two-sample MR explores psychosocial factors. Zhu et al. (2024) used genetic instruments from genome-wide association studies (GWAS) for social isolation phenotypes and depression, revealing that genetically predicted loneliness and social isolation were causally linked to higher depression incidence.

Cardiac research benefits from proteomics-integrated MR. Kim et al. (2021) found no causal tie between caffeine metabolism variants and arrhythmia risk. Henry et al. (2022) applied an MR model accounting for linkage disequilibrium, using multiverse sensitivity (up to 120 instrument-model combinations) on plasma proteins. They identified eight circulating proteins causally tied to heart failure (HF), assessing druggability, mechanisms, and side effects via cross-trait MR. Rasooly et al. (2023) combined MR proteomics with cis-only colocalization, pinpointing 10 druggable genes causally implicated in HF.

Vascular and neurological applications include DePaolo et al. (2023), who employed pleiotropy-robust two-sample MR to link blood pressure causally to ascending thoracic aortic diameter. In neurodegeneration, Deng et al. (2023) used univariable and multivariable MR (adjusting for BMI and physical activity) to show frailty causally raises Parkinson's disease risk.

These advances highlight MR's evolution toward robustness and integration with omics data, enhancing precision medicine (Burgess et al., 2013; Bowden et al., 2015; Zheng et al., 2017; Grant and Burgess, 2020).

1.3 GWAS data used in MR analysis and AI for acquiring GWAS data

At its statistical core, MR employs instrumental variables to probe causality, where the instrument correlates with the exposure but not with confounders or outcomes (GWASLab, 2024; Zhang, 2025b). In genetics, single-nucleotide polymorphisms (SNPs)—single-base DNA variations—commonly serve as instruments. Genome-Wide Association Study (GWAS) data aggregates SNP-phenotype associations, forming the bedrock of MR by identifying SNPs significantly tied to exposures (e.g., biomarkers or diseases).

Key GWAS databases for MR include: (1) OpenGWAS, a user-friendly, open-access repository aggregating GWAS summaries for diverse phenotypes (diseases, traits, behaviors), enabling SNP lookups for exposures. (2) GWAS Catalog, a curated, searchable database of SNP-trait links from published studies, with metadata extraction, weekly updates, and API/FTP access; it accepts unpublished submissions post-2020. (3) GWAS Atlas, a harmonized platform for summary statistics, supporting cross-trait analyses, genetic correlations, and PheWAS via web queries, RESTful APIs, or bulk downloads, with ongoing updates from biobanks. (4) eQTLGen Consortium, the largest blood eQTL meta-analysis (>31,000 Europeans), mapping cis- and trans-eQTL for ~19,000 genes; it aids colocalization and transcriptome-wide MR through portals, downloads, or pipelines. (5) FinnGen, integrating genomics with Finnish health records (>1.4 million participants), offering free GWAS summaries for broad diseases (cardiovascular, cancer, mental health).

In an earlier study, I created a web tool for multi-source GWAS retrieval, ensuring reliability despite access restrictions (Zhang, 2016; Zhang, 2026b). AI-driven approaches are gaining traction for data acquisition

(Extance, 2025). I developed an AI-powered GWAS fetcher as a web tool, using models like DeepSeek, Google Gemini, and OpenAI GPT to simulate realistic genetic data for univariate/multivariable MR (Zhang, 2026a). This bypasses database restrictions, allowing experimentation with exposure-outcome pairs and tool validation without real-data access.

1.4 Software tools for MR Analysis and their Pros and Cons

MR analysis relies on specialized software for harmonization, estimation, and sensitivity testing. Key R-based tools include TwoSampleMR, MendelianRandomization, MR-Base, and others, with Python alternatives emerging.

TwoSampleMR (Hemani et al., 2018) is a comprehensive R package for two-sample MR, automating data extraction from OpenGWAS, harmonization (e.g., allele alignment), and analysis with IVW, MR-Egger, weighted median, and MR-PRESSO. Pros: User-friendly pipeline, integrates with IEU OpenGWAS, supports Steiger filtering for directionality, and visualizes results (e.g., scatter/funnel plots). Cons: Relies on public databases, potentially limiting non-European ancestries; computationally intensive for large SNP sets; assumes summary statistics availability.

MendelianRandomization (Burgess et al., 2023) focuses on core MR methods, including IVW, MR-Egger, weighted median, and robust variants like MR-Lasso. It handles individual- or summary-level data, with functions for pleiotropy tests (e.g., Egger intercept) and power calculations. Pros: Flexible for custom analyses, strong on statistical rigor (e.g., bias correction), and educational via vignettes. Cons: Less automated than TwoSampleMR for data fetching; requires manual input, increasing error risk for novices; limited built-in visualization.

MR-Base (Hemani et al., 2018) is a platform extending TwoSampleMR, offering a web interface for querying >10,000 traits and running MR interactively. Pros: Seamless integration of phenome-wide scans, colocalization, and App-based workflows; supports multivariable MR. Cons: Web dependency may face access issues; less customizable for advanced scripting; potential privacy concerns with cloud processing.

Other tools: MRrobust (Wang et al., 2020) excels in bias-robust estimation under weak instruments or pleiotropy, using median-based methods. Pros: High accuracy in simulations; efficient for invalid instruments. Cons: Narrow scope, lacking full pipelines. ivonesamplemr handles one-sample MR with individual data. Pros: Tailored for cohort studies. Cons: Less versatile for two-sample designs. Python's MendelianRandomization (via PyMR) mirrors R tools but is nascent. Pros: Integrates with pandas/scikit-learn. Cons: Fewer validated methods; community smaller than R ecosystem.

Overall, R tools dominate due to maturity and GWAS integration, but selection depends on data type and expertise. Limitations include ancestry biases and computational demands; future tools may incorporate AI for automation (Zhang, 2025a).

1.5 Limitations of current MR application studies

Current MR studies face notable limitations. Reliance on European-ancestry GWAS introduces bias, limiting generalizability to diverse populations. Pleiotropy remains challenging, even with robust methods like MR-Egger, as horizontal effects can bias estimates. Weak instrument bias arises from low SNP-exposure associations, inflating type I errors. Sample overlap in two-sample MR may induce confounding, and summary statistics often lack individual-level detail for multivariable adjustments. Small effect sizes in complex traits demand large samples, yet data scarcity persists for rare diseases. Inconsistent findings across algorithms/estimators and studies—e.g., contradictory causal estimates for the same exposure-outcome pair—stem from varying instruments and assumptions (Huang, 2023; Zhang, 2024a).

1.6 Meta-analyses and their advantages

Meta-analyses address these by quantitatively pooling MR results, enhancing precision and resolving

discrepancies. A systematic review systematically collects, appraises, and synthesizes literature with rigorous methods, often incorporating meta-analysis for quantitative integration (Tang and Yang, 2015). "Systematic" ensures comprehensive literature searches, standardized protocols, and statistical pooling. Meta-analysis, emerging in the 1970s amid literature explosion (Adair and Vohra, 2003; Bangert-Drowns, 1986), statistically combines commensurable effects (e.g., odds ratios from multiple MRs) to yield averaged estimates, overcoming qualitative review subjectivity (Zhang, 2024a).

Advantages include increased statistical power, reduced bias via heterogeneity tests (e.g., I^2), and subgroup analyses for robustness. Huang (2023) and Zhang (2024a) showed averaged estimators outperform individuals in bias and efficiency. In evidence-based medicine, meta-analyzing high-quality randomized trials (or MR equivalents) provides top-tier evidence, vital for rare diseases lacking large studies (European Medicines Agency, 2006; Korn et al., 2013; Gagne et al., 2014; Röver et al., 2015; Zhang, 2024a). Despite occasional inconsistencies, meta-analyses standardize MR synthesis, guiding policy and research.

1.7 The present study

Mendelian Randomization (MR) encompasses diverse methods for causal inference using genetic instruments (Hansen, 1982; Burgess et al., 2013; Burgess and Bowden, 2015; Bowden et al., 2015; del Greco et al., 2015; Bowden et al., 2016; Burgess et al., 2015; Burgess et al., 2016; Hartwig et al., 2017; Zheng et al., 2017; Grant and Burgess, 2020; Lin et al., 2023; Wang, 2023; Xu et al., 2023; Yavorska and Staley, 2023; GWASLab, 2024; XM, 2024). In a prior article, I provided a descriptive overview of select MR methods (Zhang, 2025). This study advances that by detailing the mathematical algorithms underlying key MR approaches, including IVW (weighted least squares minimization), MR-Egger (regression with intercept for pleiotropy), and median-based robust estimators (e.g., optimization under directional constraints). In present study, I proposed a pooled algorithm for meta-analysis of MR estimates, integrating inverse-variance weighting across studies to compute overall causal effects, with heterogeneity assessment via Cochran's Q.

Additionally, I developed Meta-MR, a web-based computational tool for MR analysis. It supports data harmonization, multi-method estimation, and meta-pooling, with web-based interfaces for accessibility. This tool addresses gaps in existing software by enabling meta-pooling of estimators, AI-assisted data fetching and cross-database integration (Zhang, 2026a; Zhang, 2026b). The significance lies in enhancing MR's rigor and usability, facilitating reliable causal insights for disease etiology and drug discovery. By formalizing algorithms and introducing meta-pooling, this work mitigates inconsistencies in MR applications, promoting evidence synthesis in precision medicine.

2 Mathematical Algorithms of Mendelian Randomization

2.1 The Weighted Median Method (WM)

The Weighted Median method in Mendelian Randomization (MR) is a robust method used to estimate causal effects when some of the genetic instruments (IVs) may be invalid. It provides a consistent estimate of the causal effect even if up to 50% of the weight comes from invalid instruments.

The following is a detailed description of background and algorithm involved in the Weighted Median method.

(I) Overview of the Weighted Median Method

The weighted median method aggregates the causal effect estimates from multiple genetic instruments (IVs) by calculating a weighted median. This method is less sensitive to outliers and invalid instruments compared to the inverse-variance weighted (IVW) method.

(II) Key Steps and Procedures

Step 1: Obtain SNP-Outcome and SNP-Exposure Associations

For each genetic variant (SNP), obtain:

(i) The SNP-exposure association (β_{X_j}) and its standard error (SE_{X_j}).

(ii) The SNP-outcome association (β_{Y_j}) and its standard error (SE_{Y_j}).

These are typically derived from genome-wide association studies (GWAS).

Step 2: Calculate the Ratio Estimates

For each SNP *j*, compute the ratio estimate of the causal effect:

$$\hat{\theta}_j = \frac{\beta_{Y_j}}{\beta_{X_j}}$$

where

β_{Y_j} is the SNP-outcome association,

β_{X_j} is the SNP-exposure association.

Step 3: Compute the Weights

The weight for each SNP *j* is calculated as:

$$w_j = \frac{\beta_{X_j}^2}{SE_{Y_j}^2}$$

where

SE_{Y_j} is the standard error of the SNP-outcome association.

These weights reflect the precision of the ratio estimates, with more weight given to SNPs with stronger associations with the exposure and smaller standard errors.

Step 4: Sort the Ratio Estimates

Sort the ratio estimates $\hat{\theta}_j$ in ascending order, along with their corresponding weights w_j.

Step 5: Calculate the Cumulative Weights

Compute the cumulative sum of the weights for the sorted ratio estimates. The cumulative weight for the *k*-th SNP is:

$$w_k = \sum_{j=1}^{k} w_j$$

Step 6: Determine the Weighted Median

The weighted median is the value of $\hat{\theta}_j$ for which the cumulative weight reaches 50% of the total weight. Mathematically, find the smallest *k* such that:

$$w_k \geq 0.5 \times w_{\text{total}}$$

where $w_{\text{total}} = \sum_{j=1}^{n} w_j$ is the total weight across all SNPs.

The corresponding $\hat{\theta}_j$ is the weighted median estimate of the causal effect.

(III) Formula for the Weighted Median Estimate

The weighted median estimate $\hat{\theta}_{\text{WM}}$ is given by:

$$\hat{\theta}_{\text{WM}} = \hat{\theta}_k$$

where

$$w_k \geq 0.5 \times w_{\text{total}}$$

(IV) Confidence Interval Calculation

To compute the confidence interval for the weighted median estimate, use a bootstrapping procedure:

(i) Resample the SNPs with replacement.

(ii) Recalculate the weighted median for each bootstrap sample.

(iii) Use the percentiles of the bootstrap distribution (e.g., 2.5% and 97.5%) to construct the confidence interval.

(V) Assumptions

(i) Validity of Instruments: Up to 50% of the weight can come from invalid instruments (those that violate the exclusion restriction or have pleiotropic effects).

(ii) Independence: SNPs should be independent (no linkage disequilibrium).

(iii) Monotonicity: The effect of the SNP on the exposure should be monotonic.

(VI) Advantages

(i) Robust to invalid instruments, provided less than 50% of the weight comes from invalid SNPs.

(ii) Less sensitive to outliers compared to the IVW method.

(VII) Limitations

(i) Requires a large number of SNPs to ensure robustness.

(ii) Less efficient than the IVW method when all SNPs are valid.

2.2 The Inverse-Variance Weighted Method (IVW)

The Inverse-Variance Weighted (IVW) method (Zhang, 2024a) is a common method used in Mendelian Randomization (MR) to estimate the causal effect of an exposure on an outcome using genetic variants as instrumental variables (IVs). The method combines the ratio estimates from multiple genetic variants, weighting each by the inverse of its variance to provide an overall causal estimate.

The following is a detailed description of background and algorithm involved in the IVW method.

(I) Assumptions for Mendelian Randomization

Before applying the IVW method, ensure the genetic variants satisfy the MR assumptions:

(i) Relevance: The genetic variants are strongly associated with the exposure.

(ii) Independence: The genetic variants are not associated with confounders.

(iii) Exclusion Restriction: The genetic variants affect the outcome only through the exposure.

(II) Data Requirements

(i) Genetic associations with the exposure: For each genetic variant *j*, obtain the estimated effect size ($\hat{\beta}_{X_j}$) and its standard error ($SE_{\hat{\beta}_{X_j}}$).

(ii) Genetic associations with the outcome: For each genetic variant *j*, obtain the estimated effect size ($\hat{\beta}_{Y_j}$) and its standard error ($SE_{\hat{\beta}_{Y_j}}$).

(III) Calculate Ratio Estimates

For each genetic variant *j*, compute the ratio estimate ($\hat{\theta}_j$) of the causal effect of the exposure on the outcome:

$$\hat{\theta}_j = \frac{\hat{\beta}_{Y_j}}{\hat{\beta}_{X_j}}$$

The standard error of the ratio estimate is approximated using the delta method:

$$SE_{\hat{\theta}_j} = \frac{SE_{\hat{\beta}_{Y_j}}}{|\hat{\beta}_{X_j}|}$$

(IV) Weighting by Inverse Variance

Each ratio estimate is weighted by the inverse of its variance (w_j):

$$w_j = \frac{1}{SE^2_{\hat{\theta}_j}}$$

(V) Combine Ratio Estimates Using IVW

The overall causal effect ($\hat{\theta}_{\text{IVW}}$) is calculated as a weighted average of the individual ratio estimates:

$$\hat{\theta}_{\text{IVW}} = \frac{\sum_{j=1}^{J} w_j \hat{\theta}_j}{\sum_{j=1}^{J} w_j}$$

The standard error of the IVW estimate is:

$$SE_{\hat{\theta}_{\text{IVW}}} = \sqrt{\frac{1}{\sum_{j=1}^{L} w_j}}$$

(VI) Fixed-Effect *vs*. Random-Effects Models

(i) Fixed-Effect IVW: Assumes all genetic variants estimate the same causal effect. The formula above applies directly.

(ii) Random-Effects IVW: Accounts for heterogeneity (variability) among the ratio estimates. The weights are adjusted by incorporating an additional variance term (τ^2)

$$w_j^* = \frac{1}{SE^2_{\hat{\theta}_j} + \tau^2}$$

The heterogeneity statistic (Q) is calculated as:

$$Q = \sum_{j=1}^{J} w_j (\hat{\theta}_j - \hat{\theta}_{\text{IVW}})^2$$

If Q exceeds its degrees of freedom ($J-1$), τ^2 is estimated and used to adjust the weights.

(VII) Interpretation

(i) The IVW estimate ($\hat{\theta}_{\text{IVW}}$) represents the causal effect of the exposure on the outcome.

(ii) A confidence interval can be constructed using:

$$\hat{\theta}_{\text{IVW}} \pm Z_{1-\alpha/2} * SE_{\hat{\theta}_{\text{IVW}}}$$

The α-value should follow a stricter standard (0.001, 0.0001, etc. Zhang, 2022a-c, 2024a-c).

(VIII) Sensitivity Analyses

Perform sensitivity analyses to assess robustness, such as Egger regression or weighted median methods, to detect pleiotropy or violations of MR assumptions.

2.3 The Debiased Inverse-Variance Weighted Method (DIVW)

The Debiased Inverse-Variance Weighted (DIVW) method is an extension of the traditional Inverse-Variance Weighted method used in Mendelian Randomization (MR) analysis. The Debiased IVW method aims to reduce bias in the causal effect estimate, particularly when there is weak instrument bias or pleiotropy (where genetic variants affect the outcome through pathways other than the exposure).

The following is a detailed description of background and algorithm involved in the Debiased IVW method.

(I) Key Assumptions of Mendelian Randomization

Before applying the Debiased IVW method, ensure the following MR assumptions are met:

(i) Relevance: The genetic variants are strongly associated with the exposure.

(ii) Exclusion Restriction: The genetic variants affect the outcome only through the exposure (no horizontal pleiotropy).

(iii) Independence: The genetic variants are not confounded by other variables.

(II) Traditional Inverse-Variance Weighted (IVW) Method

The traditional IVW method estimates the causal effect (β) by performing a weighted regression of the genetic variant-outcome associations ($\hat{\beta}_{Y_j}$) on the genetic variant-exposure associations ($\hat{\beta}_{X_j}$), with weights proportional to the inverse of the variance of the outcome associations. The formula for the IVW estimate is:

$$\hat{\beta}_{\text{IVW}} = \frac{\sum_{j=1}^{L} \hat{\beta}_{Y_j} \hat{\beta}_{X_j} / \sigma_{Y_j}^2}{\sum_{j=1}^{L} \hat{\beta}_{X_j}^2 / \sigma_{Y_j}^2}$$

where

L is the number of genetic variants.

$\hat{\beta}_{X_j}$ is the estimated association between the j-th genetic variant and the exposure.

$\hat{\beta}_{Y_j}$ is the estimated association between the j-th genetic variant and the outcome.

$\sigma_{Y_j}^2$ is the variance of $\hat{\beta}_{Y_j}$.

(III) Weak Instrument Bias in IVW

The traditional IVW method can produce biased estimates when the genetic variants are weak instruments (i.e., $\hat{\beta}_{X_j}$ is small). This bias arises because the denominator in the IVW formula is inflated by measurement error in $\hat{\beta}_{X_j}$.

(IV) Debiased IVW Method

The Debiased IVW method adjusts for weak instrument bias by modifying the denominator of the IVW formula. The adjustment accounts for the uncertainty in $\hat{\beta}_{X_j}$.

The formula for the Debiased IVW estimate is:

$$\hat{\beta}_{\text{Debised IVW}} = \frac{\sum_{j=1}^{L} \hat{\beta}_{Y_j} \hat{\beta}_{X_j} / \sigma_{Y_j}^2}{\sum_{j=1}^{L} (\hat{\beta}_{X_j}^2 - \sigma_{X_j}^2) / \sigma_{Y_j}^2}$$

where

$\sigma_{X_j}^2$ is the variance of $\hat{\beta}_{X_j}$.

The term $\sigma_{X_j}^2$ is subtracted from $\hat{\beta}_{X_j}^2$ to correct for the bias introduced by weak instruments.

(V) Steps to Implement the Debiased IVW Method

(i) Obtain Genetic Associations:

Collect summary statistics for the genetic variant-exposure associations ($\hat{\beta}_{X_j}$) and their variances ($\sigma_{X_j}^2$).

Collect summary statistics for the genetic variant-outcome associations ($\hat{\beta}_{Y_j}$) and their variances ($\sigma_{Y_j}^2$).

(ii) Calculate the Debiased IVW Estimate:

Use the formula above to compute $\hat{\beta}_{\text{Debised IVW}}$.

(iii) Estimate the Standard Error:

The standard error of the Debiased IVW estimate can be calculated using:

$$SE_{\hat{\beta}_{\text{Debised IVW}}} = \sqrt{\frac{1}{\sum_{j=1}^{L}(\hat{\beta}_{X_j}^2 - \sigma_{X_j}^2)/\sigma_{Y_j}^2}}$$

(iv) Test for Significance:

Compute the 95% confidence interval and *p*-value for the causal effect estimate using the standard error.

(VI) Advantages of the Debiased IVW Method

(i) Reduces bias in the causal effect estimate when genetic variants are weak instruments.

(ii) Provides a more reliable estimate in the presence of measurement error in $\hat{\beta}_{X_j}$.

(VII) Limitations

(i) The Debiased IVW method assumes that all genetic variants are valid instruments (no pleiotropy).

(ii) If pleiotropy is present, additional methods (e.g., Egger or Weighted Median) may be required.

2.4 The Egger Method (Egger)

The Egger method in Mendelian Randomization (MR) is a statistical technique used to assess and correct for pleiotropy, which occurs when a genetic variant influences multiple traits, potentially biasing the causal estimate. The Egger method is an extension of the inverse-variance weighted (IVW) method and is particularly useful when there is directional pleiotropy (i.e., the pleiotropic effects are not balanced around zero).

The following is a detailed description of background and algorithm involved in the Egger method.

(I) Data Preparation:

(i) Collect summary statistics for the genetic variants (SNPs) from a genome-wide association study (GWAS) for both the exposure (*X*) and the outcome (*Y*).

(ii) Ensure that the SNPs are valid instrumental variables (IVs), meaning they are strongly associated with the exposure and satisfy the exclusion restriction (i.e., they affect the outcome only through the exposure).

(II) Estimation of SNP-Exposure and SNP-Outcome Associations:

(i) For each SNP *i*, obtain the estimated effect size ($\hat{\beta}_{X_i}$) on the exposure (*X*) and its standard error ($SE_{\hat{\beta}_{X_i}}$).

(ii) Similarly, obtain the estimated effect size ($\hat{\beta}_{Y_i}$) on the outcome (*Y*) and its standard error ($SE_{\hat{\beta}_{Y_i}}$).

(III) Weighted Regression:

(i) Perform a weighted linear regression of the SNP-outcome effects ($\hat{\beta}_{Y_i}$) on the SNP-exposure effects ($\hat{\beta}_{X_i}$), using the inverse of the variance of the SNP-outcome effects as weights.

(ii) The regression model is:

$$\hat{\beta}_{Y_i} = \beta_0 + \beta_1 \hat{\beta}_{X_i} + \varepsilon_i$$

where

β_0 is the intercept, which captures the average pleiotropic effect.

β_1 is the slope, which represents the causal effect of the exposure on the outcome.

ε_i is the error term.

(IV) Estimation of Causal Effect:

(i) The causal effect of the exposure on the outcome is given by the slope β_1 from the regression model.

(ii) The intercept β_0 provides an estimate of the average pleiotropic effect. If β_0 is significantly different from zero, it suggests the presence of directional pleiotropy.

(V) Assessment of Pleiotropy:

(i) The significance of the intercept β_0 can be tested using a *t*-test. A non-zero intercept indicates that the genetic variants have pleiotropic effects that are not balanced around zero.

(ii) If the intercept is not significantly different from zero, the Egger method reduces to the standard IVW method.

(VI) Robustness Checks:

(i) Perform sensitivity analyses to assess the robustness of the results, such as leave-one-out analysis or using different sets of genetic instruments.

(ii) Compare the results from the Egger method with those from other MR methods (e.g., IVW, weighted median) to ensure consistency.

(VII) Formulae for the Egger Method

(i) Weighted Regression Model:

$$\hat{\beta}_{Y_i} = \beta_0 + \beta_1 \hat{\beta}_{X_i} + \varepsilon_i$$

where

$\hat{\beta}_{Y_i}$ is the effect size of SNP *i* on the outcome.

$\hat{\beta}_{X_i}$ is the effect size of SNP *i* on the exposure.

β_0 is the intercept (average pleiotropic effect).

β_1 is the slope (causal effect of the exposure on the outcome).

ε_i is the error term.

(ii) Weights:

The weights w_i are the inverse of the variance of the SNP-outcome effects:

$$w_i = \frac{1}{SE_{\hat{\beta}_{Y_i}}^2}$$

(iii) Estimation of Parameters:

The parameters β_0 and β_1 are estimated using weighted least squares (WLS) regression:

$$\hat{\beta}_1 = \frac{\sum w_i (\hat{\beta}_{X_i} - \bar{\beta}_X)(\hat{\beta}_{Y_i} - \bar{\beta}_Y)}{\sum w_i (\hat{\beta}_{X_i} - \bar{\beta}_X)^2}$$

$$\hat{\beta}_0 = \bar{\beta}_Y - \hat{\beta}_1 \bar{\beta}_X$$

where $\bar{\beta}_X$ and $\bar{\beta}_Y$ are the weighted means of $\hat{\beta}_{X_i}$ and $\hat{\beta}_{Y_i}$, respectively.

(iv) Standard Errors:

The standard errors of $\hat{\beta}_0$ and $\hat{\beta}_1$ are obtained from the WLS regression output, which can be used to construct confidence intervals and perform hypothesis tests.

(VIII) Interpretation

(i) Causal Effect β_1: Represents the estimated causal effect of the exposure on the outcome. A significant β_1 suggests a causal relationship.

(ii) Intercept β_0: Captures the average pleiotropic effect. A significant β_0 indicates the presence of directional pleiotropy, suggesting that some genetic variants may influence the outcome through pathways other than the exposure.

2.5 The Maximum Likelihood Method (MaxLik)

The Maximum Likelihood Method (MaxLik) in Mendelian Randomization (MR) is a statistical method used to estimate the causal effect of an exposure (X) on an outcome (Y) using genetic variants (G) as instrumental variables (IVs). The method relies on the assumption that the genetic variants are associated with the exposure but not with any confounding factors that might affect the outcome.

The following is a detailed description of background and algorithm involved in the Maximum Likelihood method.

(I) Key Assumptions

Before applying the Maximum Likelihood Method, the following assumptions must hold:

(i) Relevance: The genetic variants (G) are strongly associated with the exposure (X).

(ii) Exclusion Restriction: The genetic variants (G) affect the outcome (Y) only through the exposure (X).

(iii) Independence: The genetic variants (G) are independent of confounders (U) that affect both the exposure (X) and the outcome (Y).

(II) Model Specification

The relationships between the genetic variants (G), exposure (X), and outcome (Y) can be modeled using linear regression equations:

(i) First-Stage Model (Exposure Model):

$$X = \alpha_0 + \alpha_G G + \varepsilon_X$$

where

X: Exposure.

G: Genetic variant(s).

α_G: Effect of the genetic variant(s) on the exposure.

ε_X: Error term, assumed to be normally distributed with mean 0 and variance σ_X^2.

(ii) Second-Stage Model (Outcome Model):

$$Y = \beta_0 + \beta_X X + \varepsilon_Y$$

where

Y: Outcome.

β_X: Causal effect of the exposure on the outcome (parameter of interest).

ε_Y: Error term, assumed to be normally distributed with mean 0 and variance σ_Y^2.

(iii) Reduced-Form Model (Combined Model):

Substitute the first-stage model into the second-stage model to obtain:

$$Y = (\beta_0 + \beta_X \alpha_0) + (\beta_X \alpha_G) G + (\beta_X \varepsilon_X + \varepsilon_Y)$$

This can be rewritten as:

$$Y = \gamma_0 + \gamma_G G + \varepsilon$$

where

$\gamma_0 = (\beta_0 + \beta_X \alpha_0)$: Intercept.

$\gamma_G = \beta_X \alpha_G$: Reduced-form effect of the genetic variant(s) on the outcome.

$\varepsilon = \beta_X \varepsilon_X + \varepsilon_Y$: Combined error term.

(III) Likelihood Function

The Maximum Likelihood Method estimates the parameters by maximizing the likelihood function. The likelihood function is derived from the joint distribution of the exposure (X) and outcome (Y) given the genetic variant(s) (G).

(i) Joint Distribution of X and Y:

Assuming normality of the error terms, the joint distribution of X and Y given G is:

$$\begin{bmatrix} X \\ Y \end{bmatrix} \sim N(\begin{bmatrix} \alpha_0 + \alpha_G G \\ \gamma_0 + \gamma_G G \end{bmatrix}, \begin{bmatrix} \sigma_X^2 & \beta_X \sigma_X^2 \\ \beta_X \sigma_X^2 & \beta_X^2 \sigma_X^2 + \sigma_Y^2 \end{bmatrix})$$

(ii) Likelihood Function:

The likelihood function L is the product of the probability density functions (PDFs) of X and Y given G:

$$L(\alpha_0, \alpha_G, \beta_0, \beta_X, \sigma_X^2, \sigma_Y^2 | X, Y, G) = \prod_{i=1}^{n} f(X_i, Y_i | G_i)$$

where $f(X_i, Y_i | G_i)$ is the bivariate normal PDF for the i-th observation.

(IV) Log-Likelihood Function

To simplify calculations, the log-likelihood function is used:

$$\ell(\alpha_0, \alpha_G, \beta_0, \beta_X, \sigma_X^2, \sigma_Y^2 | X, Y, G) = \sum_{i=1}^{n} \ln f(X_i, Y_i | G_i)$$

The log-likelihood function is maximized with respect to the parameters $\alpha_0, \alpha_G, \beta_0, \beta_X, \sigma_X^2, \sigma_Y^2$.

(V) Parameter Estimation

The parameters are estimated by solving the following system of equations derived from the first-order conditions of the log-likelihood function:

$$\frac{\partial \ell}{\partial \alpha_0} = 0, \frac{\partial \ell}{\partial \alpha_G} = 0, \frac{\partial \ell}{\partial \beta_0} = 0, \frac{\partial \ell}{\partial \beta_X} = 0, \frac{\partial \ell}{\partial \sigma_X^2} = 0, \frac{\partial \ell}{\partial \sigma_Y^2} = 0$$

Estimate of the Causal Effect (β_X):

The causal effect of the exposure on the outcome is estimated as:

$$\hat{\beta}_X = \frac{\hat{\gamma}_G}{\hat{\alpha}_G}$$

where $\hat{\gamma}_G$ and $\hat{\alpha}_G$ are the maximum likelihood estimates of γ_G and α_G, respectively.

(VI) Standard Errors and Confidence Intervals

The standard errors of the parameter estimates are obtained from the inverse of the Fisher information matrix. Confidence intervals for β_X can be constructed using the standard error and the normal approximation:

$$CI = \hat{\beta}_X \pm z_{1-\alpha/2} SE_{\hat{\beta}_X}$$

where $z_{1-\alpha/2}$ is the critical value from the standard normal distribution.

(VII) Advantages and Limitations

(1) Advantages:

(i) Provides efficient and consistent estimates under the assumptions.

(ii) Can handle multiple genetic variants and covariates.

(2) Limitations:

(i) Sensitive to violations of the IV assumptions (e.g., pleiotropy).

(ii) Requires large sample sizes for reliable estimates.

2.6 The Multivariable Inverse-Variance Weighted Method (MVIVW)

The Multivariable Inverse-Variance Weighted Method is an extension of the standard Inverse-Variance Weighted (IVW) method used in Mendelian Randomization (MR) to estimate causal effects when multiple exposures or risk factors are involved. It accounts for potential pleiotropy (where genetic variants influence multiple traits) and provides a more robust estimate of causal effects in the presence of correlated exposures.

The following is a detailed description of background and algorithm involved in the Multivariable Inverse-Variance Weighted method.

(I) Key Assumptions

The MVIVW method relies on the following assumptions:

(i) Relevance: Genetic variants are strongly associated with the exposures.

(ii) Independence: Genetic variants are independent of confounders.

(iii) Exclusion Restriction: Genetic variants affect the outcome only through the exposures (no horizontal pleiotropy).

(II) Data Requirements

(i) Genetic associations with exposures: For each genetic variant j, obtain the estimated effects ($\hat{\beta}_{X_{1j}}$, $\hat{\beta}_{X_{2j}}$,..., $\hat{\beta}_{X_{pj}}$) on p exposures, along with their standard errors ($SE_{\hat{\beta}_{X_{1j}}}$, $SE_{\hat{\beta}_{X_{2j}}}$, ..., $SE_{\hat{\beta}_{X_{pj}}}$).

(ii) Genetic associations with the outcome: For each genetic variant j, obtain the estimated effect ($\hat{\beta}_{Y_j}$) on the outcome and its standard error ($SE_{\hat{\beta}_{Y_j}}$).

(iii) Correlation matrix: A matrix R describing the correlations between the genetic associations with the exposures.

(III) Model Specification

The MVIVW method models the relationship between the genetic associations with the exposures and the genetic associations with the outcome as a linear regression:

$$\hat{\beta}_{Y_j} = \theta_1\hat{\beta}_{X_{1j}} + \theta_2\hat{\beta}_{X_{2j}} + ... + \theta_p\hat{\beta}_{X_{pj}} + \varepsilon_j$$

where

$\theta_1, \theta_2, ..., \theta_p$ are the causal effects of the p exposures on the outcome.

ε_j is the error term, assumed to follow a normal distribution with mean 0 and variance σ_j^2 .

(IV) Weighting by Inverse Variance

The MVIVW method uses inverse-variance weighting to account for the precision of the genetic associations. The weight for each genetic variant j is:

$$w_j = \frac{1}{SE_{\hat{\beta}_{Y_j}}^2}$$

(V) Estimation of Causal Effects

The causal effects $\theta_1, \theta_2, ..., \theta_p$ are estimated using weighted least squares regression. The formula for the estimator is:

$$\hat{\theta} = (X^T W X)^{-1} X^T W Y$$

where

X is the $n \times p$ matrix of genetic associations with the exposures ($\hat{\beta}_{X_{1j}}$, $\hat{\beta}_{X_{2j}}$,..., $\hat{\beta}_{X_{pj}}$).

Y is the $n \times 1$ vector of genetic associations with the outcome ($\hat{\beta}_{Y_j}$).

W is the $n \times n$ diagonal weight matrix with w_j on the diagonal.

(VI) Variance-Covariance Matrix

The variance-covariance matrix of the estimated causal effects is given by:

$$VAR(\hat{\theta}) = (X^T W X)^{-1}$$

This matrix provides the standard errors and covariances of the causal effect estimates, which are used to calculate confidence intervals and perform hypothesis testing.

(VII) Handling Correlated Exposures

If the exposures are correlated, the correlation matrix R is incorporated into the weighting scheme. The weighted least squares regression is adjusted to account for the correlations, ensuring unbiased estimates.

(VIII) Interpretation

(i) The estimated θ_1, θ_2, ..., θ_p represent the causal effects of each exposure on the outcome, conditional on the other exposures.

(ii) The method assumes that all genetic variants are valid instruments (no horizontal pleiotropy). If this assumption is violated, sensitivity analyses (e.g., Egger or MVEgger) may be required.

(VIIII) An Example

Suppose you have two exposures (X_1 and X_2) and one outcome (Y). The steps are:

(i) Collect genetic associations for X_1, X_2, and Y.

(ii) Construct the matrices X, Y, and W.

(iii) Use the formula $\hat{\theta} = (X^T W X)^{-1} X^T W Y$ to estimate the causal effects.

(iv) Interpret θ_1 and θ_2 as the causal effects of X_1 and X_2, on Y, respectively.

2.7 The Multivariable Egger Method (MVEgger)

The Multivariable Egger Method is an extension of the standard Mendelian Randomization (MR) framework, which is used to estimate causal effects in the presence of unmeasured confounding. The method is particularly useful when multiple exposures or risk factors are being analyzed simultaneously.

The following is a detailed description of background and algorithm involved in the Multivariable Egger method.

(I) Key Concepts

(i) Egger Regression: A method that extends MR to account for potential pleiotropy (where a genetic variant influences the outcome through pathways other than the exposure).

(ii) Multivariable Egger: Extends Egger to handle multiple exposures simultaneously, allowing for the estimation of direct causal effects of each exposure while adjusting for the others.

(II) Assumptions

The Multivariable Egger Method relies on the following assumptions:

(i) Relevance: The genetic variants are strongly associated with the exposures.

(ii) Independence: The genetic variants are independent of confounders.

(iii) Exclusion Restriction: The genetic variants affect the outcome only through the exposures (no horizontal pleiotropy).

(iv) Instrument Strength Independent of Direct Effect (InSIDE): The strength of the genetic instruments is independent of their direct effects on the outcome (required for Egger).

(III) Procedures

Step 1: Data Preparation

(i) Collect summary statistics for genetic associations with:

Exposures (X_1, X_2, ...): Associations between SNPs and each exposure.

Outcome (Y): Associations between SNPs and the outcome.

(ii) Ensure that the same SNPs are used for all exposures and the outcome.

Step 2: Multivariable Egger Regression

The Multivariable Egger regression extends the standard Egger method to multiple exposures. The model is specified as follows:

$$\hat{\beta}_{Y_j} = \theta_0 + \sum_{i=1}^{k} \theta_i \hat{\beta}_{X_{ij}} + \varepsilon_j$$

where

$\hat{\beta}_{Y_j}$: The estimated association of SNP j with the outcome.

$\hat{\beta}_{X_{ij}}$: The estimated association of SNP *j* with exposure *i*.

θ_0: The intercept, which captures the average pleiotropic effect (bias due to horizontal pleiotropy).

θ_i: The causal effect of exposure i*i* on the outcome, adjusted for other exposures.

ε_j: The error term for SNP *j*.

Step 3: Estimation

(i) Use weighted regression to account for the uncertainty in the SNP-exposure and SNP-outcome associations.

(ii) The weights are typically the inverse of the variance of the SNP-outcome associations.

Step 4: Interpretation

(i) The intercept (θ_0) tests for directional pleiotropy. If $\theta_0 = 0$, there is no evidence of pleiotropy.

(ii) The slope coefficients (θ_i) represent the direct causal effects of each exposure on the outcome, adjusted for the other exposures.

(IV) Formulae

(i) Weighted Multivariable Egger Regression

The weighted regression minimizes the following objective function:

$$\sum_{j=1}^{m} w_j (\hat{\beta}_{Y_j} - \theta_0 - \sum_{i=1}^{k} \theta_i \hat{\beta}_{X_{ij}})^2$$

where

$w_j = 1/ VAR(\hat{\beta}_{Y_j})$: The weight for SNP *j*.

m: The number of SNPs.

(ii) Causal Effect Estimates

The causal effect estimates (θ_i) are obtained by solving the weighted least squares problem. The standard errors of the estimates are derived from the regression output and can be used to compute confidence intervals.

(iii) Test for Pleiotropy

The intercept (θ_0) is tested for significance using a *t*-test. A significant intercept suggests the presence of directional pleiotropy.

(V) Advantages

(i) Accounts for pleiotropy by allowing for an intercept.

(ii) Handles multiple exposures simultaneously, providing direct causal estimates for each exposure.

(iii) Robust to violations of the InSIDE assumption under certain conditions.

(VI) Limitations

(i) Requires strong genetic instruments for all exposures.

(ii) Sensitive to weak instrument bias.

(iii) Assumes linearity and homogeneity of causal effects.

2.8 The Mode-Based Estimation Method (MBE)

The Mode-Based Estimation Method (MBE) in Mendelian Randomism (MR) is a robust method used to estimate causal effects in the presence of pleiotropy, where genetic variants influence the outcome through multiple pathways. The method leverages the assumption that the most common (modal) causal effect estimate across multiple genetic variants is likely to represent the true causal effect, while outliers are due to pleiotropy.

The following is a detailed description of background and algorithm involved in the Mode-Based Estimation method.

(I) Prerequisites

- (I) Data Requirements:
  - (i) A set of genetic variants (SNPs) that are strongly associated with the exposure (instrumental variables).
  - (ii) Summary statistics or individual-level data for the exposure (*X*) and outcome (*Y*).
  - (iii) Estimates of the SNP-exposure associations ($\hat{\beta}_{X_j}$) and SNP-outcome associations ($\hat{\beta}_{Y_j}$) for each SNP *j*, along with their standard errors.
- (II) Assumptions:
  - (i) The genetic variants are valid instrumental variables (satisfy the three MR assumptions: relevance, independence, and exclusion restriction).
  - (ii) Pleiotropic effects are balanced or follow a specific distribution.

(II) Procedure for Mode-Based Estimation

Step 1: Calculate SNP-Specific Causal Estimates

For each SNP *j*, compute the Wald ratio estimate of the causal effect ($\hat{\theta}_j$):

$$\hat{\theta}_j = \frac{\hat{\beta}_{Y_j}}{\hat{\beta}_{X_j}}$$

where

$\hat{\beta}_{Y_j}$ is the estimated effect of SNP *j* on the outcome.

$\hat{\beta}_{X_j}$ is the estimated effect of SNP *j* on the exposure.

Step 2: Compute Standard Errors for Causal Estimates

Calculate the standard error ($SE_{\hat{\theta}_j}$) for each $\hat{\theta}_j$ using the delta method:

$$SE_{\hat{\theta}_j} = \sqrt{\frac{SE_{\hat{\beta}_{Y_j}}^2}{\hat{\beta}_{X_j}{}^2} + \frac{\hat{\beta}_{Y_j}{}^2 SE_{\hat{\beta}_{X_j}}^2}{\hat{\beta}_{X_j}{}^4}}$$

where

$SE_{\hat{\beta}_{Y_j}}$ and $SE_{\hat{\beta}_{X_j}}$ are the standard errors of $\hat{\beta}_{Y_j}$ and $\hat{\beta}_{X_j}$, respectively.

Step 3: Apply a Weighting Scheme

Weight each causal estimate by the inverse of its variance to account for precision:

$$w_j = \frac{1}{SE_{\hat{\theta}_j}^2}$$

Step 4: Estimate the Mode

Identify the mode of the distribution of causal estimates ($\hat{\theta}_j$). This can be done using:

(i) Kernel Density Estimation (KDE): Smooth the distribution of $\hat{\theta}_j$ and identify the peak.

(ii) Weighted Mode Estimation: Use a weighted version of the mode to account for the precision of each estimate.

The mode ($\hat{\theta}_{\mathrm{MBE}}$) is the point where the density of causal estimates is highest.

Step 5: Calculate Confidence Intervals

Use bootstrapping or a similar resampling method to estimate confidence intervals for the mode. This involves:

(i) Resampling the SNPs with replacement.

(ii) Recalculating the mode for each bootstrap sample.

(iii) Determining the 2.5th and 97.5th percentiles of the bootstrap distribution to obtain the 95% confidence interval.

(III) Advantages of MBE

- (i) Robust to invalid instruments (e.g., SNPs with pleiotropic effects).
- (ii) Does not require strong assumptions about the distribution of pleiotropic effects.

(iii) Provides a consistent estimate of the causal effect when the majority of SNPs are valid instruments.

(IV) Limitations

(i) Requires a sufficient number of genetic variants to accurately estimate the mode.

(ii) Performance may degrade if the distribution of causal estimates is multimodal or poorly defined.

(iii) Sensitivity to the choice of bandwidth in kernel density estimation.

2.9 The Heterogeneity Penalization Model (Hetpen)

The Heterogeneity Penalization Model in Mendelian Randomization (MR) is a method used to address heterogeneity in the causal effect estimates derived from multiple genetic instruments. Heterogeneity can arise due to various reasons, such as pleiotropy (where a genetic variant influences the outcome through pathways other than the exposure), population stratification, or other biases. The goal of the Heterogeneity Penalization Model is to provide a robust estimate of the causal effect by down-weighting or penalizing the contributions of genetic variants that exhibit heterogeneity.

The following is a detailed description of background and algorithm involved in the Heterogeneity Penalization Model method.

(I) Detailed Procedures

(1) Data Preparation:

(i) Collect summary statistics from Genome-Wide Association Studies (GWAS) for the exposure (X) and the outcome (Y).

(ii) Identify a set of genetic variants (SNPs) that are strongly associated with the exposure (X) and can serve as instrumental variables (IVs).

(2) Initial Estimation:

Estimate the causal effect of the exposure (X) on the outcome (Y) for each individual SNP using the Wald ratio method:

$$\hat{\beta}_j = \frac{\hat{\gamma}_j}{\hat{\alpha}_j}$$

where $\hat{\gamma}_j$ is the estimated effect of the j-th SNP on the outcome (Y), and $\hat{\alpha}_j$ is the estimated effect of the j-th SNP on the exposure (X).

(3) Heterogeneity Assessment:

Calculate the heterogeneity statistic (e.g., Cochran's Q statistic) to assess the variability in the causal effect estimates across the SNPs:

$$Q = \sum_{j=1}^{J} \left(\frac{\hat{\beta}_j - \hat{\beta}_{\mathrm{IVW}}}{SE_{\hat{\beta}_j}}\right)^2$$

where $\hat{\beta}_{\text{IVW}}$ is the inverse-variance weighted (IVW) estimate of the causal effect, and $SE_{\hat{\beta}_j}$ is the standard error of the *j*-th SNP's causal effect estimate.

(4) Penalization:

Apply a penalization approach to down-weight the contributions of SNPs that exhibit heterogeneity. One common approach is to use a random-effects model where the weights are adjusted based on the extent of heterogeneity:

$$w_j = \frac{1}{SE_{\hat{\beta}_j}^2 + \hat{\tau}^2}$$

where $\hat{\tau}^2$ is the estimated heterogeneity variance.

(5) Re-estimation:

Re-estimate the causal effect using the penalized weights:

$$\hat{\beta}_{\text{penalized}} = \frac{\sum_{j=1}^{J} w_j \hat{\beta}_j}{\sum_{j=1}^{J} w_j}$$

(6) Inference:

Calculate the standard error of the penalized estimate and construct confidence intervals to assess the statistical significance of the causal effect.

(II) Formulae

(1) Wald Ratio Estimate:

$$\hat{\beta}_j = \frac{\hat{\gamma}_j}{\hat{\alpha}_j}$$

(2) Cochran's Q Statistic:

$$Q = \sum_{j=1}^{J} \left(\frac{\hat{\beta}_j - \hat{\beta}_{\text{IVW}}}{SE_{\hat{\beta}_j}}\right)^2$$

(3) Penalized Weights:

$$w_j = \frac{1}{SE_{\hat{\beta}_j}^2 + \hat{\tau}^2}$$

(4) Penalized Causal Effect Estimate:

$$\hat{\beta}_{\text{penalized}} = \frac{\sum_{j=1}^{J} w_j \hat{\beta}_j}{\sum_{j=1}^{J} w_j}$$

(III) Summary

The Heterogeneity Penalization Model in Mendelian Randomization involves identifying genetic instruments, estimating initial causal effects, assessing heterogeneity, applying penalization to down-weight heterogeneous SNPs, and re-estimating the causal effect. It combines likelihood-based methods with mixture modeling to provide robust causal inference. The key formulae include the Wald ratio estimate, Cochran's Q statistic, penalized weights, and the penalized causal effect estimate. This method helps to provide a more robust estimate of the causal effect by accounting for heterogeneity among the genetic instruments.

2.10 The Contamination Mixture Model (Conmix)

The Contamination Mixture Model (Conmix) in Mendelian Randomization (MR) is a statistical approach used to address the issue of invalid instrumental variables (IVs) that may bias causal effect estimates. The model assumes that some proportion of the genetic variants used as IVs may be invalid due to pleiotropy (i.e., they affect the outcome through pathways other than the exposure).

The following is a detailed description of background and algorithm involved in the Contamination Mixture Model method.

(I) Key Assumptions

(1) Valid IVs: A subset of genetic variants are valid IVs, meaning they satisfy the three key assumptions of MR:

(i) Relevance: Associated with the exposure.

(ii) Exclusion restriction: Affect the outcome only through the exposure.

(iii) Independence: Not associated with confounders.

(2) Invalid IVs: The remaining genetic variants are invalid due to pleiotropy, violating the exclusion restriction.

(3) Contamination Proportion: A proportion π of the genetic variants are invalid, while 1 - π are valid.

(II) Model Setup

Let:

- (i) β_X be the causal effect of the exposure X on the outcome Y.
- (ii) β_{G_j} be the effect of the j-th genetic variant G_j on the exposure X.
- (iii) α_j be the direct effect of the j-th genetic variant G_j on the outcome Y (representing pleiotropy).
- (iv) ϵ_{X_j} and ϵ_{Y_j} be random errors.

The exposure and outcome models are:

(i) Exposure Model:

$$X_j = \beta_{G_j} G_j + \epsilon_{X_j}$$

(ii) Outcome Model:

$$Y_j = \beta_X X_j + \alpha_j G_j + \epsilon_{Y_j}$$

For valid IVs, $\alpha_j = 0$; for invalid IVs, $\alpha_j \neq 0$.

(III) Contamination Mixture Model

The CMM assumes that the genetic variants are a mixture of valid and invalid IVs. The likelihood function is constructed as follows:

(i) Likelihood for Valid IVs:

For valid IVs ($\alpha_j = 0$):

$$Y_j = \beta_X X_j + \epsilon_{Y_j}$$

The likelihood contribution for valid IVs is:

$$L_j^{\text{valid}} = N(Y_j | \beta_X X_j, \sigma^2)$$

where N denotes the normal distribution and σ^2 is the variance of the error term.

(ii) Likelihood for Invalid IVs:

For invalid IVs ($\alpha_j \neq 0$):

$$Y_j = \beta_X X_j + \alpha_j G_j + \epsilon_{Y_j}$$

The likelihood contribution for invalid IVs is:

$$L_j^{\text{valid}} = N(Y_j | \beta_X X_j + \alpha_j G_j, \sigma^2)$$

(iii) Mixture Likelihood:

The overall likelihood for the j-th genetic variant is a weighted mixture of the valid and invalid likelihoods:

$$L_j = (1 - \pi) L_j^{\text{valid}} + \pi L_j^{\text{invalid}}$$

where π is the contamination proportion (probability that a variant is invalid).

(iv) Full Likelihood:

Assuming independence across genetic variants, the full likelihood for all J variants is:

$$L = \prod_{j=1}^{J} (1-\pi) L_j^{\text{valid}} + \pi\, L_j^{\text{invalid}}$$

(IV) Estimation

The parameters β_X, π, and α_j (for invalid IVs) are estimated using maximum likelihood estimation or Bayesian methods. The steps are:

(i) Specify priors for β_X, π, and α_j (if using Bayesian methods).

(ii) Maximize the log-likelihood function or sample from the posterior distribution.

(iii) Obtain point estimates and confidence intervals for β_X.

(V) Robustness

The CMM is robust to invalid IVs because it explicitly models the presence of pleiotropy. By estimating the contamination proportion π, it downweights the influence of invalid IVs on the causal effect estimate.

(VI) Advantages and Limitations

(1) Advantages:

(i) Accounts for pleiotropy without requiring prior knowledge of which IVs are invalid.

(ii) Provides a flexible framework for modeling heterogeneity in IV validity.

(2) Limitations:

(i) Requires strong assumptions about the distribution of pleiotropic effects.

(ii) Computationally intensive, especially for large numbers of genetic variants.

2.11 The Multivariable Median Method (MVMedian)

The Multivariable Median Method (MVMedian) is a robust approach in Mendelian Randomization (MR) used to estimate causal effects in the presence of multiple genetic instruments and potential pleiotropy. Pleiotropy occurs when a genetic variant influences multiple traits, which can bias MR estimates. The MVMedian extends the standard median-based methods to handle multiple exposures and instruments, providing more reliable causal estimates. By leveraging the median of ratio estimates, it offers a reliable way to infer causal relationships even when some genetic instruments may be invalid.

The following is a detailed description of background and algorithm involved in the Multivariable Median method.

(I) Key Concepts:

(i) Mendelian Randomization (MR): A method that uses genetic variants as instrumental variables (IVs) to estimate causal effects of exposures on outcomes.

(ii) Pleiotropy: When a genetic variant affects the outcome through pathways other than the exposure of interest, leading to biased estimates.

(iii) Multivariable Median Method (MVMedian): A method that combines multiple genetic instruments and exposures to provide robust causal estimates, even in the presence of pleiotropy.

(II) Procedures for MVMedian:

(1) Data Preparation:

(i) Collect summary statistics from genome-wide association studies (GWAS) for the exposure(s) and outcome.

(ii) Identify genetic variants (SNPs) that are strongly associated with the exposure(s) and satisfy the IV assumptions (relevance, independence, and exclusion restriction).

(2) Model Specification:

(i) Define the multivariable model where multiple exposures are considered simultaneously.

(ii) Let *X* be the matrix of exposures, *Y* the outcome, and *G* the matrix of genetic instruments.

(3) Estimation of Causal Effects:

(i) Use the median of the ratio estimates from individual genetic instruments to estimate the causal effect.

(ii) For each genetic instrument *j*, compute the ratio estimate

$$\hat{\beta}_j = \frac{\hat{\gamma}_j}{\hat{\alpha}_j}$$

where $\hat{\gamma}_j$ is the association of the genetic variant with the outcome, and $\hat{\alpha}_j$ is the association of the genetic variant with the exposure.

(iii) The causal effect $\hat{\beta}$ is the median of these ratio estimates.

(4) Handling Multiple Exposures:

(i) Extend the univariable median method to multiple exposures by considering the joint distribution of the ratio estimates.

(ii) Use a weighted median approach to account for the strength of each genetic instrument and its association with multiple exposures.

(5) Robustness Checks:

(i) Perform sensitivity analyses to assess the robustness of the estimates to pleiotropy.

(ii) Use methods like the Egger regression or the weighted median method to compare results and check for consistency.

(III) Formulae:

(i) Ratio Estimate for Each Instrument:

$$\hat{\beta}_j = \frac{\hat{\gamma}_j}{\hat{\alpha}_j}$$

where $\hat{\gamma}_j$ is the estimated effect of the genetic variant *j* on the outcome, and $\hat{\alpha}_j$ is the estimated effect of the genetic variant *j* on the exposure.

(ii) Multivariable Median Estimate:

$$\hat{\beta}_{\text{median}} = \text{median}(\hat{\beta}_1, \hat{\beta}_2, \ldots, \hat{\beta}_J)$$

where *J* is the number of genetic instruments.

(iii) Weighted Median Estimate:

$$\hat{\beta}_{\text{weighted median}} = \text{weighted median}(\hat{\beta}_1, \hat{\beta}_2, \ldots, \hat{\beta}_J; w_1, w_2, \ldots, w_J)$$

where w_j is the weight for each genetic instrument, typically based on the inverse of the variance of the ratio estimate.

(iv) Multivariable Extension:

For multiple exposures, the ratio estimates are extended to a vector of estimates for each exposure. The median is then taken across the vector of estimates for each exposure.

(IV) Advantages of MVMedian:

(1) Robustness to Pleiotropy: By using the median of ratio estimates, MVMedian is less sensitive to outliers and invalid instruments.

(2) Handling Multiple Exposures: MVMedian can simultaneously estimate causal effects for multiple exposures, providing a more comprehensive understanding of the causal relationships.

(V) Limitations:

(1) Assumption of Valid Instruments: MVMedian relies on the assumption that at least 50% of the genetic instruments are valid (i.e., not pleiotropic).

(2) Complexity: The method can be computationally intensive, especially with a large number of genetic instruments and exposures.

2.12 The Multivariable Lasso Method (MVLasso)

The Multivariable Lasso (Least Absolute Shrinkage and Selection Operator) method in Mendelian Randomization (MR) is a technique used to estimate causal effects in the presence of multiple potential exposures and potential pleiotropy (where genetic variants influence the outcome through pathways other than the exposure of interest). By combining the strengths of MR and Lasso regularization, the Multivariable Lasso method provides a powerful tool for causal inference in the presence of multiple exposures and pleiotropy.

The following is a detailed description of background and algorithm involved in the Multivariable Lasso method.

(I) Overview of Multivariable Mendelian Randomization (MR)

MR is a method that uses genetic variants as instrumental variables (IVs) to estimate the causal effect of an exposure on an outcome. The key assumption is that the genetic variants are associated with the exposure but not with the outcome through any confounding pathways. In multivariable MR, multiple exposures are considered simultaneously, and the goal is to estimate the direct causal effect of each exposure on the outcome.

(II) Multivariable Lasso in MR

The Multivariable Lasso method extends MR to handle multiple exposures and addresses pleiotropy by performing variable selection and regularization. It penalizes the coefficients of irrelevant or weak exposures, shrinking them toward zero.

(III) Procedures for Multivariable Lasso in MR

Step 1: Data Preparation

(i) Collect summary statistics for genetic variants, exposures, and outcomes from genome-wide association studies (GWAS).

(ii) Ensure that the genetic variants are valid instrumental variables (IVs) for the exposures (i.e., they satisfy the MR assumptions).

Step 2: Model Specification

(i) Let X be the matrix of genetic variants (IVs), Y be the outcome, and E be the matrix of exposures.

(ii) The goal is to estimate the causal effects

$$\beta = (\beta_1, \beta_2, \dots, \beta_p)$$

of the p exposures on the outcome Y.

Step 3: Multivariable Regression with Lasso Penalty

The Multivariable Lasso method solves the following optimization problem:

$$\hat{\beta} = \arg\min_{\beta} \{\frac{1}{2n}\left||Y - E\beta|\right|_2^2 + \lambda||\beta||_1\}$$

where

$\left||Y - E\beta|\right|_2^2$ is the residual sum of squares.

$||\beta||_1 = \sum_{j=1}^{p} |\beta_j|$ is the L1-norm penalty, which encourages sparsity.

λ is the regularization parameter that controls the strength of the penalty.

Step 4: Selection of Regularization Parameter λ

Use cross-validation or information criteria (e.g., AIC, BIC) to select the optimal λ that balances model fit and sparsity.

Step 5: Estimation of Causal Effects

(i) Solve the Lasso optimization problem to obtain the estimated causal effects $\hat{\beta}$.

(ii) Exposures with non-zero coefficients in $\hat{\beta}$ are considered to have a direct causal effect on the outcome.

Step 6: Inference and Interpretation

(i) Perform inference on the estimated causal effects (e.g., using bootstrap methods to estimate confidence intervals).

(ii) Interpret the results in the context of the MR assumptions and potential pleiotropy.

(IV) Key Formulae

(i) Lasso Objective Function:

$$\hat{\beta} = \arg\min_{\beta} \{\frac{1}{2n}\left|\left|Y - E\beta\right|\right|_2^2 + \lambda||\beta||_1\}$$

(ii) Residual Sum of Squares:

$$\left|\left|Y - E\beta\right|\right|_2^2 = \sum\nolimits_{i=1}^{n} (Y_i - \sum_{j=1}^{p} E_{ij}\beta_j)^2$$

(iii) L1-Norm Penalty:

$$||\beta||_1 = \sum_{j=1}^{p} |\beta_j|$$

(iv) Cross-Validation for λ:

Split the data into *K* folds.

For each λ, compute the average prediction error across folds.

Choose λ that minimizes the prediction error.

(V) Advantages of Multivariable Lasso in MR

(i) Handles multiple exposures simultaneously.

(ii) Reduces the impact of pleiotropy by shrinking weak or irrelevant exposures to zero.

(iii) Provides a sparse solution, making it easier to interpret the results.

(VI) Limitations

(i) The choice of λ can influence the results.

(ii) Assumes that the Lasso penalty adequately addresses pleiotropy.

(iii) Requires careful validation of the genetic variants as IVs.

2.13 The Single-Variable Lasso Method (Lasso)

The Single-Variable Lasso Method in Mendelian Randomization (MR) is a method used to select genetic variants (instruments) that are strongly associated with the exposure of interest while minimizing the influence of weak or invalid instruments. This method is particularly useful in MR studies to reduce bias from pleiotropy (where a genetic variant influences multiple traits) and to improve the robustness of causal inference.

The following is a detailed description of background and algorithm involved in the Single-Variable Lasso method.

(I) Single-Variable Lasso Method

The Single-Variable Lasso Method applies the Lasso (Least Absolute Shrinkage and Selection Operator) regression to select genetic variants that are strongly associated with the exposure. Lasso is a regularization technique that performs both variable selection and shrinkage by imposing an L1 penalty on the regression coefficients.

(II) Procedures for Single-Variable Lasso in MR

Step 1: Data Preparation

(i) Collect data on genetic variants (e.g., SNPs), the exposure, and the outcome.

(ii) Ensure that the genetic variants are pre-screened for relevance to the exposure (e.g., genome-wide association study (GWAS) data).

Step 2: Model the Exposure

(i) Let X be the exposure, G be the matrix of genetic variants (SNPs), and β be the vector of coefficients for the genetic variants.

(ii) The relationship between the exposure and genetic variants can be modeled as:

$$X = G\beta + \epsilon$$

where ϵ is the error term.

Step 3: Apply Lasso Regression

The Lasso estimator is obtained by minimizing the following objective function:

$$\hat{\beta} = \arg\min_{\beta} \{\frac{1}{2n}\left||X - G\beta|\right|_2^2 + \lambda||\beta||_1\}$$

where

$\left||X - G\beta|\right|_2^2$ is the residual sum of squares.

$||\beta||_1$ is the L1 norm of the coefficient vector (sum of absolute values).

λ is the regularization parameter that controls the strength of the penalty.

Step 4: Select Genetic Variants

(i) The Lasso regression shrinks some coefficients to zero, effectively selecting a subset of genetic variants that are most strongly associated with the exposure.

(ii) The selected variants are used as instruments in the MR analysis.

Step 5: Perform MR Analysis

(i) Use the selected genetic variants as instruments to estimate the causal effect of the exposure on the outcome.

(ii) The causal effect θ can be estimated using methods such as the Inverse-Variance Weighted (IVW) estimator:

$$\hat{\theta} = \frac{\sum_{j=1}^{p} \hat{\beta}_{Y_j}\hat{\beta}_{X_j}/\sigma_{Y_j}^2}{\sum_{j=1}^{p} \hat{\beta}_{X_j}^2/\sigma_{Y_j}^2}$$

where

$\hat{\beta}_{Y_j}$ is the estimated association between the j-th genetic variant and the outcome.

$\hat{\beta}_{X_j}$ is the estimated association between the j-th genetic variant and the exposure.

$\sigma_{Y_j}^2$ is the variance of the association between the j-th genetic variant and the outcome.

(III) Key Considerations

- (i) Choice of λ: The regularization parameter λ can be selected using cross-validation
- or other criteria (e.g., Bayesian Information Criterion).

(ii) Pleiotropy: The Single-Variable Lasso Method helps reduce bias from pleiotropic variants by selecting only those variants strongly associated with the exposure.

(iii) Weak Instruments: Lasso tends to shrink weak instruments to zero, improving the robustness of the MR analysis.

(IV) Advantages of Single-Variable Lasso in MR

(i) Reduces the influence of weak or invalid instruments.

(ii) Improves the interpretability of the selected genetic variants.

(iii) Provides a data-driven approach to instrument selection.

(V) Limitations

(i) The method assumes that the selected genetic variants are valid instruments (no horizontal pleiotropy).

(ii) The choice of λ can influence the results, and improper selection may lead to biased estimates.

2.14 The Constrained Maximum Likelihood Method (cML)

The Constrained Maximum Likelihood (cML) method in Mendelian Randomization (MR) is a statistical method used to estimate causal effects between an exposure and an outcome using genetic variants as instrumental variables (IVs). The method accounts for potential pleiotropy (where a genetic variant influences the outcome through pathways other than the exposure) by imposing constraints on the likelihood function. Below is a detailed description of the procedures and formulae involved in the cML method.

The following is a detailed description of background and algorithm involved in the Constrained Maximum Likelihood method.

(I) Key Assumptions

The cML method relies on the standard MR assumptions:

(i) Relevance: The genetic variants are strongly associated with the exposure.

(ii) Exclusion Restriction: The genetic variants affect the outcome only through the exposure.

(iii) Independence: The genetic variants are not associated with confounders of the exposure-outcome relationship.

In the presence of pleiotropy, the exclusion restriction may be violated. The cML method addresses this by incorporating constraints into the likelihood function.

(II) Model Setup

Let:

- X be the exposure.
- Y be the outcome.
- $G = (G_1, G_2, \ldots, G_p)$ be a set of p genetic variants used as IVs.
- U represents unmeasured confounders.

The relationships between the variables can be modeled as:

(1) Exposure model:

$$X = \alpha_0 + \sum_{j=1}^{p} \alpha_j G_j + \varepsilon_X \qquad \varepsilon_X \sim N(0, \sigma_X^2)$$

where α_j represents the effect of the j-th genetic variant on the exposure.

(2) Outcome model:

$$Y = \beta_0 + \beta_X X + \sum_{j=1}^{p} \beta_j G_j + \varepsilon_Y \qquad \varepsilon_Y \sim N(0, \sigma_Y^2)$$

where β_X is the causal effect of the exposure on the outcome, and β_j represents the direct effect of the j-th genetic variant on the outcome (pleiotropic effect).

(III) Constrained Maximum Likelihood Estimation

The goal is to estimate the causal effect β_X while accounting for pleiotropy. The cML method achieves this by imposing constraints on the likelihood function.

(1) Likelihood Function

The joint likelihood of the exposure and outcome models is:

$$L(\alpha, \beta_X, \beta, \sigma_X^2, \sigma_Y^2) = \prod_{i=1}^{n} f(X_i | G_i; \alpha, \sigma_X^2)\, f(Y_i | X_i, G_i; \beta_X, \beta, \sigma_Y^2)$$

where $f(\cdot)$ denotes the probability density function of the normal distribution.

(2) Constraints

To address pleiotropy, constraints are imposed on the direct effects β_j. For example:

(i) Zero-mean constraint: Assume that the pleiotropic effects have a mean of zero:

$$\sum_{j=1}^{p} \beta_j = 0$$

(ii) Variance constraint: Assume that the pleiotropic effects follow a distribution with a known variance.

These constraints are incorporated into the likelihood function using Lagrange multipliers or penalty terms.

(3) Optimization

The constrained likelihood function is maximized to obtain estimates of β_X and other parameters. This involves solving:

$$\hat{\beta}_X, \hat{\alpha}, \hat{\beta} = \arg\min_{\beta_X, \alpha, \beta} L(\alpha, \beta_X, \beta, \sigma_X^2, \sigma_Y^2) \quad \text{subject to constraints.}$$

(IV) Implementation Steps

(1) Data Preparation: Collect data on the genetic variants *G*, exposure *X*, and outcome *Y*.

(2) Model Specification: Define the exposure and outcome models, and specify the constraints.

(3) Likelihood Maximization: Use numerical optimization techniques (e.g., Newton-Raphson, gradient descent) to maximize the constrained likelihood function.

(4) Inference: Compute standard errors and confidence intervals for the causal effect estimate β_X using the observed Fisher information or bootstrap methods.

(V) Advantages and Limitations

(1) Advantages:

(i) Accounts for pleiotropy by incorporating constraints.

(ii) Provides robust causal effect estimates under valid assumptions.

(2) Limitations:

(i) Relies on the correctness of the constraints.

(ii) Computationally intensive for large numbers of genetic variants.

(VI) Extensions

The cML method can be extended to:

(i) Incorporate additional covariates.

(ii) Handle binary outcomes using logistic regression.

(iii) Use Bayesian frameworks to incorporate prior information.

2.15 The Penalized Inverse-Variance Weighted Method (PIVW)

The Penalized Inverse-Variance Weighted (PIVW) Method is an extension of the standard Inverse-Variance Weighted method used in Mendelian Randomization (MR) analysis. It is designed to mitigate the influence of weak instrument bias and pleiotropy, which can distort causal effect estimates. By incorporating a penalty term, the Penalized IVW method provides a more robust method to causal inference in Mendelian Randomization, particularly in the presence of weak instruments or pleiotropy.

The following is a detailed description of background and algorithm involved in the Penalized Inverse-Variance Weighted method.

(I) Overview of IVW Method in Mendelian Randomization (MR)

Mendelian Randomization uses genetic variants (typically single nucleotide polymorphisms, SNPs) as instrumental variables (IVs) to estimate the causal effect of an exposure (*X*) on an outcome (*Y*). The IVW method is a common approach that combines the ratio estimates from individual SNPs, weighting them by their precision (inverse variance).

(II) Standard Inverse-Variance Weighted (IVW) Method

The standard IVW estimator for the causal effect (β) is given by:

$$\beta_{\text{IVW}} = \frac{\sum_i w_i \hat{\beta}_{Y,i} \hat{\beta}_{X,i} p_i}{\sum_i w_i \hat{\beta}_{X,i}^2 p_i}$$

where

$\hat{\beta}_{Y,i}$: Estimated effect of SNP *i* on the outcome (*Y*).

$\hat{\beta}_{X,i}$: Estimated effect of SNP *i* on the exposure (*X*).

w_i: Weight for SNP *i*, typically the inverse of the variance of $\hat{\beta}_{Y,i}$:

$$w_i = \frac{1}{SE_{\hat{\beta}_{Y,i}}^2}$$

(III) Penalized IVW Method

The Penalized IVW method modifies the standard IVW approach by introducing a penalty term to downweight the contribution of weak instruments or SNPs with potential pleiotropic effects. The penalty reduces the influence of SNPs with small $\hat{\beta}_{X,i}$ values, which are more likely to introduce bias.

(1) Formulae for Penalized IVW

The penalized IVW estimator is:

$$\beta_{\text{penalized}} = \frac{\sum_i w_i \hat{\beta}_{Y,i} \hat{\beta}_{X,i} p_i}{\sum_i w_i \hat{\beta}_{X,i}^2 p_i}$$

where

p_i: Penalty factor for SNP *i*, defined as:

$$p_i = \frac{\hat{\beta}_{X,i}^2}{\hat{\beta}_{X,i}^2 + \lambda}$$

λ: A tuning parameter that controls the strength of the penalty. Larger values of λ impose a stronger penalty on weak instruments.

(2) Interpretation of the Penalty Factor (p_i)

(i) For strong instruments (large $\hat{\beta}_{X,i}$), $p_i \approx 1$, and the SNP contributes fully to the estimate.

(ii) For weak instruments (small $\hat{\beta}_{X,i}$), $p_i \approx 0$, and the SNP's contribution is downweighted.

(IV) Steps for Implementing the Penalized IVW Method

(1) Obtain SNP-Exposure and SNP-Outcome Estimates:

Extract the effect sizes ($\hat{\beta}_{X,i}$, $\hat{\beta}_{Y,i}$) and standard errors ($SE_{\hat{\beta}_{X,i}}$, $SE_{\hat{\beta}_{Y,i}}$) for each SNP from GWAS summary statistics.

(2) Calculate Weights (w_i):

Compute the inverse-variance weights for each SNP:

$$w_i = \frac{1}{SE_{\hat{\beta}_{Y,i}}^2}$$

(3) Choose the Tuning Parameter (λ):

Select a value for λ. Common choices include:

λ = median ($\hat{\beta}_{X,i}^2$): A data-driven approach.

λ = 0.01 or λ = 0.1: Predefined small values.

(4) Compute Penalty Factors (p_i):

For each SNP, calculate the penalty factor:

$$p_i = \frac{\hat{\beta}_{X,i}^2}{\hat{\beta}_{X,i}^2 + \lambda}$$

(5) Calculate the Penalized IVW Estimate:

Use the formula for $\beta_{\text{penalized IVW}}$ to compute the causal effect estimate.

(6) Assess Uncertainty:

Estimate the standard error of $\beta_{\text{penalized IVW}}$ using bootstrapping or analytical methods.

(V) Advantages of the Penalized IVW Method

(i) Reduces bias from weak instruments by downweighting their contribution.

(ii) Robust to pleiotropy when the pleiotropic effects are independent of instrument strength.

(iii) Simple to implement and computationally efficient.

(VI) Limitations

(i) The choice of λ can influence results, and there is no universally optimal value.

(ii) Assumes that pleiotropic effects are balanced or random, which may not always hold.

(iii) May lose efficiency if too many SNPs are penalized.

(VII) Practical Considerations

(i) Sensitivity analyses should be performed to assess the robustness of results to different λ values.

(ii) Compare results with other MR methods (e.g., Egger, Weighted Median) to evaluate consistency.

2.16 The Multivariable Constrained Maximum Likelihood Method (MVcML)

The Multivariable Constrained Maximum Likelihood Method (MVcML) in Mendelian Randomization (MR) is a statistical method used to estimate causal effects between an exposure and an outcome using genetic variants as instrumental variables (IVs). This method extends the standard maximum likelihood estimation (MLE) to handle multiple genetic instruments and potential confounders while incorporating constraints to ensure valid causal inference. It involves specifying a likelihood function, imposing constraints to ensure valid inference, and using numerical optimization to estimate the causal effect. It is a robust method for estimating causal effects by leveraging genetic variants as instrumental variables. It is particularly useful in the presence of multiple genetic instruments and potential confounders.

The following is a detailed description of background and algorithm involved in the Multivariable Constrained Maximum Likelihood method.

(I) Key Assumptions

(i) Relevance: The genetic variants are strongly associated with the exposure.

(ii) Exclusion Restriction: The genetic variants affect the outcome only through the exposure.

(iii) Independence: The genetic variants are independent of confounders and the outcome given the exposure.

(II) Procedures

(1) Model Specification:

(i) Define the exposure X and outcome Y.

(ii) Identify K genetic variants $G_1, G_2, \ldots, G_K$ as instrumental variables.

(iii) Specify the structural equations for the exposure and outcome:

$$X = \alpha_0 + \sum_{k=1}^{K} \alpha_k G_k + \varepsilon_X$$

$$Y = \beta_0 + \beta_X X + \sum_{k=1}^{K} \alpha_k G_k + \varepsilon_Y$$

Here, α_k represents the effect of the k-th genetic variant on the exposure, and β_X is the causal effect of the exposure on the outcome.

(2) Likelihood Function:

(i) Assume $\varepsilon_X \sim N(0, \sigma_X^2)$, $\varepsilon_Y \sim N(0, \sigma_Y^2)$.

(ii) The joint likelihood function for *X* and *Y* given the genetic variants *G* is:

$$L(\theta; X, Y, G) = \prod_{i=1}^{N} f_X(X_i|G_i; \alpha, \sigma_X^2)\, f_Y(Y_i|X_i, G_i; \beta, \sigma_Y^2)$$

Here, $\theta = (\alpha_0, \alpha_1, \ldots, \alpha_K, \beta_1, \beta_2, \ldots, \beta_K, \sigma_X^2, \sigma_Y^2)$ is the vector of parameters to be estimated.

(3) Constraints:

(i) To ensure the exclusion restriction, impose constraints on the direct effects of the genetic variants on the outcome:

$$\beta_k = 0;\ k=1,2,\ldots,K$$

(ii) These constraints are incorporated into the likelihood function.

(4) Maximization:

(i) Maximize the constrained likelihood function with respect to θ:

$$\theta = \arg\max_{\theta}\ L(\theta; X, Y, G) \qquad \beta_k = 0;\ k=1,2,\ldots,K$$

(ii) This can be done using numerical optimization techniques such as the Lagrange multiplier method or iterative algorithms like the Expectation-Maximization (EM) algorithm.

(5) Estimation of Causal Effect:

The causal effect of the exposure on the outcome is given by β_X, which is estimated from the maximized likelihood function.

(6) Inference:

(i) Calculate standard errors and confidence intervals for β_X using the Fisher information matrix or bootstrap methods.

(ii) Perform hypothesis testing to assess the significance of the causal effect.

(III) Formulae

(1) Likelihood Function:

$$L(\theta; X, Y, G) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma_X^2}} \exp\left(-\frac{(X_i - \alpha_0 - \sum_{j=1}^{K} \alpha_j G_{ji})^2}{2\sigma_X^2}\right) \frac{1}{\sqrt{2\pi\sigma_Y^2}} \exp\left(-\frac{(Y_i - \beta_0 - \beta_X X_i - \sum_{j=1}^{K} \beta_j G_{ji})^2}{2\sigma_Y^2}\right)$$

(2) Constrained Likelihood:

$$L_{\text{constrained}}\ (\theta; X, Y, G) = L(\theta; X, Y, G) \qquad \beta_k = 0;\ k=1,2,\ldots,K$$

(3) Causal Effect:

$$\hat{\beta}_X = \arg\max_{\beta_X}\ L_{\text{constrained}}\ (\theta; X, Y, G)$$

2.17 The Principal Component Generalized Method of Moments (PCGMM)

The Principal Component Generalized Method of Moments (PCGMM) in Mendelian Randomization (MR) is a statistical technique used to estimate causal effects in the presence of pleiotropy (where genetic variants influence the outcome through multiple pathways). By combining PCA and GMM, PCGMM provides a powerful framework for Mendelian Randomization analysis, particularly in the presence of complex genetic architectures.

The following is a detailed description of background and algorithm involved in the Principal Component Generalized Method of Moments.

(I) Challenges Addressed by PCGMM

(1) Pleiotropy: Some genetic variants may affect the outcome through pathways other than the exposure, violating the exclusion restriction assumption.

(2) Weak Instruments: Genetic variants may only weakly predict the exposure, leading to biased estimates.

(3) Correlated Instruments: Genetic variants may be correlated due to linkage disequilibrium (LD).

PCGMM addresses these issues by combining Principal Component Analysis (PCA) with the Generalized Method of Moments (GMM).

(II) Principal Component Analysis (PCA)

PCA is used to reduce the dimensionality of the genetic data and address multicollinearity among instruments. The steps are:

(1) Standardize the Genetic Data: Let Z be an $n \times p$ matrix of p genetic variants for n individuals. Standardize each column of Z to have mean 0 and variance 1.

(2) Compute the Covariance Matrix: Calculate the covariance matrix $\sum = \frac{1}{n} Z^T Z$.

(3) Perform Eigenvalue Decomposition: Decompose Σ eigenvalues and eigenvectors: $\Sigma = V \Lambda V^T$, where Λ is a diagonal matrix of eigenvalues and V is a matrix of eigenvectors.

(4) Select Principal Components (PCs): Retain the top k PCs that explain most of the variance in Z. Let P be the $n \times k$ matrix of PCs.

(III) Generalized Method of Moments (GMM)

GMM is used to estimate the causal effect β of the exposure X on the outcome Y. The key idea is to use moment conditions derived from the assumptions of MR.

(1) Moment Conditions

The moment conditions are based on the assumption that the genetic variants (or their PCs) are valid instruments:

(i) Relevance: The PCs P are associated with the exposure X.

(ii) Exclusion Restriction: The PCs P affect the outcome Y only through the exposure X.

The moment conditions can be written as:

$$E[P^T(Y - X\beta)] = 0$$

This states that the instruments (PCs) are uncorrelated with the residuals ($Y - X\beta$).

(2) GMM Estimator

The GMM estimator minimizes the following quadratic form:

$$\hat{\beta} = \arg\min_{\beta} \left[\left(Y - X\beta\right)^T P W P^T (Y - X\beta)\right]$$

where W is a weighting matrix. The optimal weighting matrix is the inverse of the covariance matrix of the moment conditions:

$$W = \left(\frac{1}{n} P^T (Y - X\beta)\left(Y - X\beta\right)^T P\right)^{-1}$$

(IV) Combining PCA and GMM

(1) Reduce Dimensionality: Use PCA to transform the genetic variants Z into PCs P.

(2) Estimate Causal Effect: Use GMM with the PCs P as instruments to estimate β.

(V) Steps for PCGMM in MR

(1) Data Preparation:

(i) Collect data on genetic variants Z, exposure X, and outcome Y.

(ii) Standardize Z.

(2) Perform PCA:

(i) Compute the covariance matrix of Z.

(ii) Perform eigenvalue decomposition and select the top k PCs.

(3) Set Up Moment Conditions:

Define the moment conditions $E[P^T(Y - X\beta)] = 0$.

(4) Estimate β Using GMM:

Minimize the GMM objective function to obtain $\hat{\beta}$.

(5) Inference:

Compute standard errors and confidence intervals for $\hat{\beta}$ using the asymptotic distribution of the GMM estimator.

(VI) Advantages of PCGMM

(i) Addresses pleiotropy by reducing the influence of invalid instruments.

(ii) Handles weak instruments and multicollinearity.

(iii) Provides efficient and robust causal estimates.

(VII) Limitations

(i) Requires large sample sizes for reliable PCA and GMM estimation.

(ii) Assumes linearity in the relationship between instruments, exposure, and outcome.

(iii) Sensitivity to the choice of the number of PCs.

2.18 The Multivariable Principal Component Generalized Method of Moments (MVPCGMM)

The Multivariable Principal Component Generalized Method of Moments (MVPCGMM) in Mendelian Randomization (MR) is an advanced statistical technique used to estimate causal effects in the presence of multiple exposures and potential pleiotropy. The MVPCGMM method combines PCA for dimensionality reduction and GMM for efficient estimation, making it a powerful tool for multivariable MR analyses. It addresses challenges such as multicollinearity and pleiotropy while providing robust causal effect estimates.

The following is a detailed description of background and algorithm involved in the Multivariable Principal Component Generalized Method of Moments.

(I) Background and Assumptions

Mendelian Randomization (MR) uses genetic variants as instrumental variables (IVs) to estimate causal effects of exposures on outcomes. The MVPCGMM extends this framework to handle:

(i) Multiple exposures: Simultaneously modeling multiple correlated exposures.

(ii) Pleiotropy: Accounting for genetic variants that may affect the outcome through pathways other than the exposures of interest.

The key assumptions for MVPCGMM are:

(i) Relevance: Genetic variants are strongly associated with the exposures.

(ii) Exclusion restriction: Genetic variants affect the outcome only through the exposures.

(iii) Independence: Genetic variants are independent of confounders.

(II) Data Preparation

(i) Let X be the matrix of p exposures (each exposure is a column vector).

(ii) Let G be the matrix of q genetic variants (each variant is a column vector).

(iii) Let Y be the vector of the outcome.

(III) Principal Component Analysis (PCA) on Exposures

To reduce dimensionality and handle multicollinearity among exposures, PCA is applied to X:

(i) Standardize X to have zero mean and unit variance.

(ii) Compute the covariance matrix $C = \frac{1}{n} X^T X$, where n is the sample size.

(iii) Perform eigenvalue decomposition: $C = V D V^T$, where D is the diagonal matrix of eigenvalues and V contains the eigenvectors.

(iv) Select the top k principal components (PCs) that explain most of the variance in X. Let $P = X V_k$, where V_k contains the first k eigenvectors.

(IV). Generalized Method of Moments (GMM)

GMM is used to estimate the causal effects of the PCs on the outcome Y. The steps are as follows:

(1) Moment Conditions

Define the moment conditions based on the instrumental variables G:

$$G(\beta) = \frac{1}{n} G^T (Y - P\beta) = 0$$

where β is the vector of causal effects of the PCs on Y.

(2) Weighting Matrix

Construct a weighting matrix W to account for the covariance structure of the moment conditions:

$$W = (\frac{1}{n} G^T G)^{-1}$$

(3) GMM Objective Function

Minimize the GMM objective function:

$$Q(\beta) = \frac{1}{n} G(\beta)^T W G(\beta)$$

(4) Estimation

Solve for β by minimizing $Q(\beta)$:

$$\hat{\beta} = \arg\ \min_{\beta}\ Q(\beta)$$

(V) Transformation Back to Original Exposures

Once $\hat{\beta}$ is estimated, transform the causal effects back to the original exposures using the PCA loadings:

$$\hat{\theta} = V_k \hat{\beta}$$

where $\hat{\theta}$ represents the causal effects of the original exposures on Y.

(VI) Inference and Testing

(1) Standard Errors: Compute the standard errors of $\hat{\theta}$ using the sandwich estimator:

$$VAR(\hat{\theta}) = V_k\ VAR(\hat{\beta})\ V_k^T$$

(2) Hypothesis Testing: Test the null hypothesis H_0: $\theta_j = 0$ for each exposure using a Wald test:

$$Z_j = \frac{\hat{\theta}_j}{SE_{\hat{\theta}_j}}$$

(VII) Handling Pleiotropy

To account for pleiotropy, use robust GMM or include additional moment conditions to test for violations of the exclusion restriction assumption.

2.19 The Multivariable Generalized Method of Moments (MVGMM)

The Multivariable Generalized Method of Moments (MVGMM) in Mendelian Randomization (MR) is a statistical approach used to estimate causal effects in the presence of unmeasured confounding and pleiotropy. MR leverages genetic variants as instrumental variables (IVs) to infer causal relationships between exposures and outcomes. The MVGMM framework is particularly useful when there are multiple genetic instruments and potential violations of the exclusion restriction (e.g., pleiotropy).

The following is a detailed description of background and algorithm involved in the Multivariable Generalized Method of Moments method.

(I) Key Assumptions in Mendelian Randomization

Before applying MVGMM, the following assumptions must hold:

(i) Relevance: Genetic variants are strongly associated with the exposure(s).

(ii) Exclusion Restriction: Genetic variants affect the outcome only through the exposure(s).

(iii) Independence: Genetic variants are independent of confounders.

(iv) No Pleiotropy (or balanced pleiotropy): Genetic variants do not directly affect the outcome through pathways other than the exposure(s).

(II) Data Structure

(i) Let X be the exposure(s) (a vector if multiple exposures are considered).

(ii) Let Y be the outcome.

(iii) Let $Z = (Z_1, Z_2, \ldots, Z_p)$ be the set of genetic instruments (SNPs).

(iv) Let U represent unmeasured confounders.

(III) Model Specification

The relationships between the variables can be modeled as:

(1) First-Stage Regression (Exposure model):

$$X = Z\alpha + \varepsilon_X$$

where α is the vector of SNP effects on the exposure, and ε_X is the error term.

(2) Second-Stage Regression (Outcome model):

$$Y = X\beta + \varepsilon_Y$$

where β is the causal effect of the exposure on the outcome, and ε_Y is the error term.

(3) Pleiotropy Adjustment:

If pleiotropy is present, the outcome model can be extended to:

$$Y = X\beta + Z\gamma + \varepsilon_Y$$

where γ represents direct effects of the SNPs on the outcome (violating the exclusion restriction).

(IV) Generalized Method of Moments (GMM)

GMM is used to estimate the causal effect β by minimizing the discrepancy between the observed data and the model-implied moment conditions.

(1) Moment Conditions

The moment conditions are derived from the assumption that the genetic instruments ZZ are uncorrelated with the error term ε_Y:

$$E(Z^T(Y - X\beta) = 0$$

If pleiotropy is present, the moment conditions are adjusted to:

$$E(Z^T(Y - X\beta - Z\gamma) = 0$$

(2) Objective Function

The GMM objective function minimizes the weighted sum of squared deviations from the moment conditions:

$$Q(\beta, \gamma) = [\frac{1}{n}\sum_{i=1}^{n} Z_i^T(Y_i - X_i\beta - Z_i\gamma)]^T W[\frac{1}{n}\sum_{i=1}^{n} Z_i^T(Y_i - X_i\beta - Z_i\gamma)]$$

where W is a weighting matrix (e.g., the inverse of the covariance matrix of the moment conditions).

(3) Estimation

(i) Two-Step GMM:

Step 1: Use an initial weighting matrix (e.g., identity matrix) to obtain preliminary estimates of β and γ.

Step 2: Update the weighting matrix using the residuals from the first step and re-estimate β and γ.

(ii) Iterative GMM:

Iteratively update the weighting matrix and parameter estimates until convergence.

(V) Handling Pleiotropy

To account for pleiotropy, the GMM framework can incorporate additional constraints or penalties:

(1) Heterogeneity Penalty: Penalize deviations from the assumption of no pleiotropy.

(2) Robust Weighting: Use a robust weighting matrix to downweight SNPs with large pleiotropic effects.

(VI) Inference

(1) Standard Errors: Compute standard errors for β using the sandwich estimator:

$$VAR(\hat{\beta}) = (G^T W G)^{-1} G^T W \, \Omega \, W G (G^T W G)^{-1}$$

where G is the gradient of the moment conditions, and Ω is the covariance matrix of the moment conditions.

(2) Hypothesis Testing: Test the null hypothesis H_0: $\beta = 0$ using a Wald test or likelihood ratio test.

(VII) Example Workflow

(1) Data Preparation: Collect summary statistics or individual-level data for *X*, *Y*, and *Z*.

(2) First-Stage Regression: Estimate SNP effects on the exposure(s).

(3) Second-Stage Regression: Apply GMM to estimate the causal effect β, adjusting for pleiotropy.

(4) Validation: Perform sensitivity analyses (e.g., Egger, Weighted Median) to assess robustness.

2 Meta-MR

2.1 Heterogeneity test of algorithms

If the between-algorithm variation is entirely caused by probabilistic factors, the size of the variation is limited. Therefore, with a certain number of algorithms, there is sufficient confidence (such as 99.9%) that the observed overall variation should be less than a certain upper limit. If the actual variation observed is greater than this upper limit, it indicates that important algorithmic heterogeneity may exist (Zhang, 2024a). Otherwise, there is no sufficient reason to consider algorithmic heterogeneity exists, and it is believed that the variation is mainly caused by probabilistic factors. It would be reasonable to use meta-analyses to pool data. The significance testing used to measure the size of the heterogeneity of a set of algorithms and to estimate whether it is entirely due to probability or not are called heterogeneity testing, including *Q*-Test (Zhang, 2024a).

Let

k: Number of algorithms (treatments).

n: Number of datasets (blocks/subjects).

A performance matrix P_{ij} of size *n* x *k*, where P_{ij} is the performance score (e.g., accuracy) of the *j*-th algorithm on the *i*-th dataset.

The algorithm of Cochrane's *Q*-Test for heterogeneity testing of algorithms is:

(1) State the hypotheses

Null Hypothesis (H_0): All algorithms have identical performance. Any differences are due to random variation.

H_0: $\mu_1 = \mu_2 = \ldots = \mu_k$

Alternative Hypothesis (H_1): At least one algorithm performs differently from the others.

H_1: At least one μ_i is different.

(2) Rank the performance within each dataset

This is a crucial step that makes the test non-parametric and robust. For each dataset (each row of the performance matrix *P*), you assign ranks to the algorithms based on their performance.

The best-performing algorithm on that dataset gets the highest rank (e.g., *k* if using ranks 1 to *k*, or 1 if using average ranks for ties where 1 is best).

In case of ties, assign the average rank to the tied algorithms. For example, for 3 algorithms (A, B, C) on one dataset with accuracies (0.95, 0.88, 0.95): A: 0.95 -> Tied for 1st -> Average rank of (1+2)/2 = 1.5; B:

0.88 -> 3rd place -> 3; C: 0.95 -> Tied for 1st -> Average rank of (1+2)/2 = 1.5;

End up with a rank matrix R of the same size n x k.

(3) Calculate the sum of ranks for each algorithm

For each algorithm j (each column of the rank matrix R), compute its total rank sum, R_j.

$$R_j = \Sigma_{i=1}^{n} r_{ij}$$

where r_{ij} is the rank of algorithm j on dataset i.

(4) Compute Cochrane's Q statistic

Use the following formula:

$$Q = ((k\text{-}1) [k \Sigma R_j^2 - (\Sigma R_j)^2]) / (k \Sigma T - \Sigma S)$$

where

k: Number of algorithms.

R_j: The total rank sum for algorithm j (from step (3)).

ΣR_j^2: Sum of the squares of each algorithm's total rank sum, $= R_1^2 + R_2^2 + ... + R_k^2$

$(\Sigma R_j)^2$: Square of the sum of all total rank sums. $= (R_1 + R_2 + ... + R_k)^2$

ΣT: Sum of the squares of the ranks in the entire rank matrix R. $= \Sigma\Sigma(r_{ij}^2)$ for all i and j.

ΣS: Sum of the squares of the sums of ranks for each dataset. For each dataset i, calculate the sum of its ranks $S_i = \Sigma r_{ij}$, then square it S_i^2, and sum these squares across all datasets. $= \Sigma S_i^2$

(5) Determine the significance

The Q statistic follows a Chi-square (χ^2) distribution with k-1 degrees of freedom.

Compare your calculated Q value to the critical value from the χ^2 distribution table with k-1 degrees of freedom at your chosen significance level (e.g., $\alpha = 0.05$).

Alternatively, compute the p-value: $p = P(\chi^2_{k-1} > Q)$.

(6) Make a decision

If p-value $< \alpha$ (e.g., < 0.05), reject the null hypothesis (H_0). You conclude that there is statistically significant heterogeneity in performance, meaning at least one algorithm is different.

If p-value $\geq \alpha$, fail to reject H_0, we do not have sufficient evidence to say that the algorithms' performances are different.

If the Q-Test is significant, something is different, sometimes we need a post-hoc test to perform pairwise comparisons. A common choice is the Nemenyi test. The steps are:

(a) Calculate the critical difference (CD): $CD = q_\alpha$ sqrt((k*(k+1)) / (6n)), where q_α is the critical value from the Studentized range statistic divided by √2.

(b) Compute the average ranks for each algorithm: $AR_j = R_j / n$.

(c) Compare the difference in average ranks between every pair of algorithms. If the difference $|AR_x - AR_y| > CD$, then that pair is considered significantly different.

2.2 Meta-MR: pooled algorithm

The choice of different algorithms will certainly affect the final conclusion (Huang, 2023; Zhang, 2024a). Antonelli and Cefalu (2020) suggested that using as many algorithms as possible is the best choice for reducing the impact that some bad algorithms can have and increasing the efficiency of the pooled algorithm. Algorithms may be averaged to solve the algorithm selection problem, which is aimed to provide a pooled method that linearly combined of a set of individual algorithms in order to provide a better estimate (Lavancier

and Rochet, 2015, 2017; Mitra et al., 2019; Antonelli and Cefalu, 2020; Huang, 2023; Zhang, 2014a). Huang (2023) has proved that the algorithm average performed better than individual algorithms in terms of bias and efficiency. In MR study, we can use algorithm averaging to pool the causal effects from individual algorithms, where the biases of individual algorithms are assumed to be uniformly distributed about zero. By using pooled algorithm, the estimated causal effect and conclusion of MR will be more robust and reliable.

The pooled causal effect is the weighted average of *N* individual algorithms described above, which is given by

$$\theta_A = \sum_{i=1}^{N} w_i \theta^{(i)} / \sum_{i=1}^{N} w_i$$

where w_i is the weight associated with the *i*th algorithm $\theta^{(i)}$. The weights are given by

$$w_i = 1/s_{\theta^{(i)}}^2$$

where $s_{\theta^{(i)}}$ is the standard error for *i*th algorithm $\theta^{(i)}$.

If the algorithms are homogeneous, the standard error of pooled causal effect is given by (Huang, 2023; Zhang, 2024a)

$$s_{\theta_A} = \sum_{i=1}^{N} (1/s_{\theta^{(i)}}) / \sum_{i=1}^{N} (1/s_{\theta^{(i)}}^2)$$

The 100(1-α)% confidence (coverage) interval is given by

$$\theta_A \pm z_\alpha s_{\theta_A}$$

If the algorithms are heterogeneous, the between-algorithm variance is given by

$$\tau_A^2 = \sum_{i=1}^{N} (1/s_{\theta^{(i)}}^2) \tau_i^2 / \sum_{i=1}^{N} (1/s_{\theta^{(i)}}^2)$$

and τ_i^2 is the between-algorithm variance of *i*th algorithm. And the 100(1-α)% confidence (coverage) interval is given by

$$\theta_A \pm z_\alpha s_A$$

where

$$s_A = \sqrt{\tau_A^2 + s_{\theta_A}^2}$$

However, the between-algorithm variance of *i*th algorithm, τ_i^2, are unavailable at present. In present study, the algorithms are supposed to be homogeneous.

3 JavaScript Codes of Mendelian Randomization Algorithms

I have developed the JavaScript codes of Mendelian randomization algorithms. Mathematical and statistical libraries used in present study are mainly from Burgess et al. (2023), Cloudflare (2025), Zhang (2026c), etc.

3.1 Data preparation

The JavaScript+HTML codes for acquisition and preparation of MR data can be found in Zhang (2025a, 2026a-b, 2026d), Zhang and Qi (2026).

3.2 JavaScript Codes for Various Algorithms

Full JavaScript codes for all algorithms of Mendelian randomization (http://www.iaees.org/publications/journals/nb/articles/2027-17(2)/1-Zhang-Abstract.asp) are listed bellow.

3.2.1 Algorithms for single variable

(1) The Inverse-Variance Weighted Method (IVW)

The JavaScript codes for The Inverse-Variance Weighted Method (IVW) are as follows:

```
/*
Method name and objective
Method: The Inverse-Variance Weighted Method (IVW) Mendelian Randomization
Objective: Estimate the causal effect of an exposure on an outcome using inverse-variance weighted regression, with support for robust regression, penalized weights, correlated instruments, and different confidence interval/distribution options.

Main classes and functions (brief descriptions)
MRInput
Purpose: Encapsulates the input data for the MR analysis.
Key fields: betaX, betaY, betaXse, betaYse (Arrays of SNP effects and standard errors); correlation (optional matrix); exposure, outcome (descriptions).
IVW
Purpose: Store and present the IVW analysis results.
Key fields: Model, Exposure, Outcome, Robust, Penalized, Correlation, Estimate, StdError, CILower, CIUpper, SNPs, Pvalue, Alpha, RSE, HeterStat, Fstat.
Method: display() formats and prints the results.
ci_normal, ci_t
Purpose: Compute confidence interval bounds using normal or t-distribution, respectively.
penalisedWeights, penalisedWeightsDelta
Purpose: Compute weights with penalization; the Delta version accounts for sample overlap via a psi term.
rWeights, rWeightsDelta
Purpose: Convert base weights into final regression weights, with and without Delta (overlap) adjustments.
mr_ivw
Purpose: Core analysis function that performs the IVW MR procedure.
Input: an MRInput object plus an optional configuration object (model, robust, penalized, weights, psi, correl, distribution, alpha).
Output: an IVW instance containing the estimate, standard error, confidence interval, p-value, F-statistic, heterogeneity stats, and flags about correlation/model usage.
Key capabilities: supports standard IVW, robust regression, penalized weights, handling of correlated instruments, single vs multiple SNP scenarios, and switching between normal or t-distribution-based inference.

Required inputs and expected outputs

Inputs
MRInput data: betaX, betaY, betaXse, betaYse (all same length, one per SNP).
```

Optional: correlation (matrix of correlations between instruments) or null (no correlation).
Description fields: exposure, outcome.
Optional settings for mr_ivw: model (e.g., "default", "random", "fixed"), robust (true/false), penalized (true/false), weights ("simple" or "delta"), psi (overlap parameter, default 0), correl (include instrument correlations), distribution ("normal" or "t-dist"), alpha (significance level, default 0.05).

Outputs
An IVW object that includes: Model, Exposure, Outcome, Robust, Penalized, Correlation, Estimate (causal effect), StdError, CILower, CIUpper, SNPs, Pvalue, Alpha, RSE, HeterStat, Fstat.
A display() method on the IVW object to print a human-readable summary.
Depending on options, results may reflect standard IVW, robust regression, penalized weighting, and/or correlated instruments, with confidence intervals computed via normal or t-distribution.
*/

// --- Helper Functions (ci_normal, ci_t) ---

```
/**
 * Calculates confidence interval bound using normal distribution.
 * @param {'l'|'u'} type - 'l' for lower bound, 'u' for upper bound.
 * @param {number} estimate - The point estimate.
 * @param {number} se - The standard error of the estimate.
 * @param {number} alpha - The significance level (e.g., 0.05 for 95% CI).
 * @returns {number} The confidence interval bound.
 */
function ci_normal(type, estimate, se, alpha) {
    const q = jStat.normal.inv(1 - alpha / 2, 0, 1);
    if (type === "l") {
        return estimate - q * se;
    } else if (type === "u") {
        return estimate + q * se;
    }
    return NaN;
}

/**
 * Calculates confidence interval bound using t-distribution.
 * @param {'l'|'u'} type - 'l' for lower bound, 'u' for upper bound.
 * @param {number} estimate - The point estimate.
 * @param {number} se - The standard error of the estimate.
 * @param {number} df - Degrees of freedom.
 * @param {number} alpha - The significance level (e.g., 0.05 for 95% CI).
 * @returns {number} The confidence interval bound.
 */
function ci_t(type, estimate, se, df, alpha) {
    if (df <= 0) return NaN;
```

```
    const q = jStat.studentt.inv(1 - alpha / 2, df);
    if (type === "l") {
        return estimate - q * se;
    } else if (type === "u") {
        return estimate + q * se;
    }
    return NaN;
}

// --- Penalized Weights Functions ---

/**
 * Calculates p-values for penalization of weights (simple).
 * @param {number[]} Bx - Beta-coefficient for genetic association with the risk factor.
 * @param {number[]} Bxse - Standard error of genetic association with the risk factor.
 * @param {number[]} By - Beta-coefficient for genetic association with the outcome.
 * @param {number[]} Byse - Standard error of genetic association with the outcome.
 * @returns {number[]} P-values corresponding to heterogeneity test for each genetic variant.
 */
function penalisedWeights(Bx, Bxse, By, Byse) {
    if (Bx.length === 0) return [];
    const ratio_By_Bx = By.map((y, i) => y / Bx[i]);
    const theta = jStat.median(ratio_By_Bx);

    const penWeights = Bx.map((x, i) => {
        const term1 = (x * x) / (Byse[i] * Byse[i]);
        const term2 = (By[i] / Bx[i] - theta);
        const chiSqStat = term1 * (term2 * term2);
        return 1 - jStat.chisquare.cdf(chiSqStat, 1);
    });
    return penWeights;
}

/**
 * Calculates p-values for penalization of weights with second-order (delta) weights.
 * @param {number[]} Bx - Beta-coefficient for genetic association with the risk factor.
 * @param {number[]} Bxse - Standard error of genetic association with the risk factor.
 * @param {number[]} By - Beta-coefficient for genetic association with the outcome.
 * @param {number[]} Byse - Standard error of genetic association with the outcome.
 * @param {number} psi - Correlation between genetic associations (sample overlap).
 * @returns {number[]} P-values corresponding to heterogeneity test for each genetic variant.
 */
function penalisedWeightsDelta(Bx, Bxse, By, Byse, psi) {
    if (Bx.length === 0) return [];
    const ratio_By_Bx = By.map((y, i) => y / Bx[i]);
```

```
    const theta = jStat.median(ratio_By_Bx);

    const penWeights = Bx.map((x, i) => {
        const varDeltaDenom = (Byse[i] * Byse[i]) / (x * x) +
                              (By[i] * By[i] * Bxse[i] * Bxse[i]) / (x * x * x * x) -
                              (2 * psi * By[i] * Bxse[i] * Byse[i]) / (x * x * x);
        const weightFactor = (varDeltaDenom === 0 || isNaN(varDeltaDenom) || !isFinite(varDeltaDenom)) ? 0 : 1 / varDeltaDenom;

        const termDiffSquared = (By[i] / Bx[i] - theta);
        const chiSqStat = weightFactor * (termDiffSquared * termDiffSquared);
        return 1 - jStat.chisquare.cdf(chiSqStat, 1);
    });
    return penWeights;
}

/**
 * Calculates penalized weights (simple).
 * @param {number[]} Byse - Standard error of genetic association with the outcome.
 * @param {number[]} penWeights - Factors for penalizing weights from `penalisedWeights`.
 * @returns {number[]} Penalized weights.
 */
function rWeights(Byse, penWeights) {
    return Byse.map((se, i) => (1 / (se * se)) * Math.min(1, penWeights[i] * 100));
}

/**
 * Calculates penalized weights (delta).
 * @param {number[]} Bx - Beta-coefficient for genetic association with the risk factor.
 * @param {number[]} Bxse - Standard error of genetic association with the risk factor.
 * @param {number[]} By - Beta-coefficient for genetic association with the outcome.
 * @param {number[]} Byse - Standard error of genetic association with the outcome.
 * @param {number} psi - Correlation between genetic associations (sample overlap).
 * @param {number[]} penWeights - Factors for penalizing weights from `penalisedWeightsDelta`.
 * @returns {number[]} Penalized weights.
 */
function rWeightsDelta(Bx, Bxse, By, Byse, psi, penWeights) {
    return Bx.map((x, i) => {
        const varDeltaDenom = (Byse[i] * Byse[i]) +
                              (By[i] * By[i] * Bxse[i] * Bxse[i]) / (x * x) -
                              (2 * psi * By[i] * Bxse[i] * Byse[i]) / x;
        const weightFactor = (varDeltaDenom === 0 || isNaN(varDeltaDenom) || !isFinite(varDeltaDenom)) ? 0 : 1 / varDeltaDenom;
        return weightFactor * Math.min(1, penWeights[i] * 20);
    });
```

```
}

// --- MRInput and IVW Result Classes ---

/**
 * Represents the input data for Mendelian Randomization analysis.
 */
class MRInput {
    constructor({betaX, betaY, betaXse, betaYse, correlation = null, exposure = "Exposure", outcome = "Outcome"}) {
        this.betaX = betaX || [];
        this.betaY = betaY || [];
        this.betaXse = betaXse || [];
        this.betaYse = betaYse || [];
        this.correlation = correlation;
        this.exposure = exposure;
        this.outcome = outcome;

        const lengths = new Set([this.betaX.length, this.betaY.length, this.betaXse.length, this.betaYse.length]);
        if (lengths.size !== 1) {
            throw new Error("MRInput: All input beta and SE arrays must have the same length.");
        }
    }
}

/**
 * Represents the results of an Inverse-Variance Weighted (IVW) MR analysis.
 */
class IVW {
    constructor({
        Model, Exposure, Outcome, Robust, Penalized, Correlation,
        Estimate, StdError, CILower, CIUpper,
        SNPs, Pvalue, Alpha, RSE, HeterStat, Fstat
    }) {
        this.Model = Model;
        this.Exposure = Exposure;
        this.Outcome = Outcome;
        this.Robust = Robust;
        this.Penalized = Penalized;
        this.Correlation = Correlation;
        this.Estimate = Estimate;
        this.StdError = StdError;
        this.CILower = CILower;
        this.CIUpper = CIUpper;
        this.SNPs = SNPs;
        this.Pvalue = Pvalue;
```

```
        this.Alpha = Alpha;
        this.RSE = RSE;
        this.HeterStat = HeterStat;
        this.Fstat = Fstat;
    }

    // Method to display results
    display() {
        console.log(`\n=== IVW Results ===`);
        console.log(`Model: ${this.Model}`);
        console.log(`Exposure: ${this.Exposure} → Outcome: ${this.Outcome}`);
        console.log(`Robust: ${this.Robust}, Penalized: ${this.Penalized}`);
        console.log(`Estimate: ${this.Estimate.toFixed(6)}`);
        console.log(`Std Error: ${this.StdError.toFixed(6)}`);
        console.log(`95% CI: [${this.CILower.toFixed(6)}, ${this.CIUpper.toFixed(6)}]`);
        console.log(`P-value: ${this.Pvalue.toExponential(4)}`);
        console.log(`SNPs: ${this.SNPs}`);
        console.log(`F-statistic: ${this.Fstat.toFixed(4)}`);
        console.log(`RSE: ${this.RSE.toFixed(6)}`);
        if (!isNaN(this.HeterStat[0])) {
            console.log(`Heterogeneity Stat: ${this.HeterStat[0].toFixed(4)} (p=${this.HeterStat[1].toFixed(6)})`);
        }
    }
}

// --- Main mr_ivw Function ---

/**
 * Performs Inverse-Variance Weighted (IVW) Mendelian Randomization analysis.
 *
 * @param {MRInput} object - MRInput object containing genetic data.
 * @param {object} options - Configuration options for the analysis.
 * @param {string} [options.model="default"] - Model type: "default", "fixed", "random".
 * @param {boolean} [options.robust=false] - Whether to use robust regression.
 * @param {boolean} [options.penalized=false] - Whether to use penalized weights.
 * @param {string} [options.weights="simple"] - Weighting scheme: "simple", "delta".
 * @param {number} [options.psi=0] - Psi parameter for delta weights (sample overlap correlation).
 * @param {boolean} [options.correl=false] - Whether to account for correlated instruments.
 * @param {string} [options.distribution="normal"] - Distribution for CI and p-value: "normal", "t-dist".
 * @param {number} [options.alpha=0.05] - Significance level for confidence intervals.
 * @returns {IVW} An IVW object containing the analysis results.
 */
function mr_ivw(object, options = {}) {
    const {
        model: initialModel = "default",
```

```
        robust = false,
        penalized = false,
        weights = "simple",
        psi = 0,
        correl = false,
        distribution = "normal",
        alpha = 0.05
    } = options;

    const Bx = object.betaX;
    const By = object.betaY;
    const Bxse = object.betaXse;
    const Byse = object.betaYse;
    const rho = object.correlation;
    const nsnps = Bx.length;

    let currentModel = initialModel;
    if (currentModel === "default") {
        currentModel = (nsnps < 4) ? "fixed" : "random";
    }

    // Input validation
    const validModels = ["default", "random", "fixed"];
    const validDistributions = ["normal", "t-dist"];
    const validWeights = ["simple", "delta"];

    if (!validModels.includes(currentModel) && initialModel !== "default") {
        throw new Error(`Invalid model type: '${initialModel}'. Must be one of: ${validModels.join(', ')}.`);
    }
    if (!validDistributions.includes(distribution)) {
        throw new Error(`Invalid distribution type: '${distribution}'. Must be one of: ${validDistributions.join(', ')}.`);
    }
    if (!validWeights.includes(weights)) {
        throw new Error(`Invalid weights type: '${weights}'. Must be one of: ${validWeights.join(', ')}.`);
    }

    let thetaIVW, thetaIVWse, rse, pvalue, ciLower, ciUpper, heterStat, pvalueHeterStat, fstat;
    let effectiveRobust = robust;
    let effectivePenalized = penalized;
    let effectiveCorrelation = correl;

    // Check if correlation matrix is provided and not entirely null/NaN
    if (rho !== null && math.sum(rho) !== null && !isNaN(math.sum(rho))) {
        effectiveCorrelation = true;
    }
```

```
if (effectiveCorrelation) {
    if (rho === null) {
        console.warn("Correlation matrix not given, but 'correl' was true or inferred. Proceeding without correlation.");
        effectiveCorrelation = false;
    } else {
        const omega_base = math.multiply(math.transpose(math.matrix(Byse)), math.matrix([Byse]));
        const Omega_matrix = math.dotMultiply(omega_base, rho);

        let Omega_inv;
        try {
            Omega_inv = math.inv(Omega_matrix);
        } catch (e) {
            throw new Error("Failed to invert Omega matrix for correlated instruments. Error: " + e.message);
        }

        const Bx_col_vec = math.transpose(math.matrix(Bx));
        const Bx_row_vec = math.matrix([Bx]);

        const term_denom_matrix = math.multiply(Bx_row_vec, Omega_inv, Bx_col_vec);
        const term_denom = math.subset(term_denom_matrix, math.index(0, 0));
        if (term_denom === 0 || !isFinite(term_denom)) {
            throw new Error("Singular matrix encountered in correlated IVW calculation.");
        }
        const inv_term_denom = 1 / term_denom;

        const term_num_matrix = math.multiply(Bx_row_vec, Omega_inv, math.transpose(math.matrix(By)));
        const term_num = math.subset(term_num_matrix, math.index(0, 0));

        thetaIVW = inv_term_denom * term_num;

        const residuals = By.map((y, i) => y - thetaIVW * Bx[i]);
        const residuals_col_vec = math.transpose(math.matrix(residuals));

        let rse_calc_term_matrix;
        if (nsnps > 1) {
            rse_calc_term_matrix = math.multiply(math.transpose(residuals_col_vec), Omega_inv, residuals_col_vec);
            rse = Math.sqrt(math.subset(rse_calc_term_matrix, math.index(0, 0)) / (nsnps - 1));
        } else {
            rse = NaN;
        }

        if (currentModel === "random") {
            thetaIVWse = Math.sqrt(inv_term_denom) * Math.max(rse, 1);
        } else if (currentModel === "fixed") {
```

```
                thetaIVWse = Math.sqrt(inv_term_denom);
            }

            effectiveRobust = false;
            effectivePenalized = false;

            if (nsnps > 1 && rse_calc_term_matrix) {
                heterStat = (nsnps - 1) * (math.subset(rse_calc_term_matrix, math.index(0, 0)) / (nsnps - 1));
                pvalueHeterStat = 1 - jStat.chisquare.cdf(heterStat, nsnps - 1);
            } else {
                heterStat = NaN;
                pvalueHeterStat = NaN;
            }

            try {
                const Bx_div_Bxse = Bx.map((b, i) => b / Bxse[i]);
                const chol_rho = math.cholesky(rho);
                const inv_chol_rho = math.inv(chol_rho);

                const transformed_Bx_Bxse_row = math.multiply(math.matrix([Bx_div_Bxse]), inv_chol_rho);
                const transformed_Bx_Bxse_array = transformed_Bx_Bxse_row.toArray()[0];
                const sum_sq_transformed = transformed_Bx_Bxse_array.reduce((sum, val) => sum + val * val, 0);
                fstat = sum_sq_transformed / nsnps;
            } catch (e) {
                console.warn(`Could not calculate F-stat with correlation: ${e.message}`);
                fstat = NaN;
            }

        }
    } else {
        if (nsnps === 0) {
            throw new Error("No SNPs detected for IVW analysis.");
        } else if (nsnps === 1) {
            thetaIVW = By[0] / Bx[0];
            if (weights === "simple") {
                thetaIVWse = Math.abs(Byse[0] / Bx[0]);
            } else if (weights === "delta") {
                thetaIVWse = Math.sqrt(
                    (Byse[0] * Byse[0]) / (Bx[0] * Bx[0]) +
                    (By[0] * By[0] * Bxse[0] * Bxse[0]) / (Bx[0] * Bx[0] * Bx[0] * Bx[0]) -
                    (2 * psi * By[0] * Bxse[0] * Byse[0]) / (Bx[0] * Bx[0] * Bx[0])
                );
            }
            rse = 1;
            heterStat = NaN;
```

```
            pvalueHeterStat = NaN;
            fstat = (Bx[0] / Bxse[0]) * (Bx[0] / Bxse[0]);

        } else {
            let currentWeights;
            if (robust || penalized) {
                console.warn("Robust regression (lmrob equivalent) is not fully implemented. Falling back to weighted least squares.");

                if (penalized) {
                    if (weights === "simple") {
                        const penWeights = penalisedWeights(Bx, Bxse, By, Byse);
                        currentWeights = rWeights(Byse, penWeights);
                    } else if (weights === "delta") {
                        const penWeights = penalisedWeightsDelta(Bx, Bxse, By, Byse, psi);
                        currentWeights = rWeightsDelta(Bx, Bxse, By, Byse, psi, penWeights);
                    }
                } else {
                    if (weights === "simple") {
                        currentWeights = Byse.map(se => 1 / (se * se));
                    } else if (weights === "delta") {
                        currentWeights = Bx.map((x, i) => {
                            const denom = (Byse[i] * Byse[i]) +
                                (By[i] * By[i] * Bxse[i] * Bxse[i]) / (x * x) -
                                (2 * psi * By[i] * Bxse[i] * Byse[i]) / x;
                            return (denom === 0 || isNaN(denom) || !isFinite(denom)) ? 0 : 1 / denom;
                        });
                    }
                }
            } else {
                if (weights === "simple") {
                    currentWeights = Byse.map(se => 1 / (se * se));
                } else if (weights === "delta") {
                    currentWeights = Bx.map((x, i) => {
                        const denom = (Byse[i] * Byse[i]) +
                            (By[i] * By[i] * Bxse[i] * Bxse[i]) / (x * x) -
                            (2 * psi * By[i] * Bxse[i] * Byse[i]) / x;
                        return (denom === 0 || isNaN(denom) || !isFinite(denom)) ? 0 : 1 / denom;
                    });
                }
            }

            const weightedBxSumSq = Bx.reduce((sum, x, i) => sum + currentWeights[i] * x * x, 0);
            const weightedBxBySum = Bx.reduce((sum, x, i) => sum + currentWeights[i] * x * By[i], 0);
```

```
            if (weightedBxSumSq === 0 || !isFinite(weightedBxSumSq)) {
                throw new Error("Weighted sum of squares of Bx is zero or non-finite.");
            }
            thetaIVW = weightedBxBySum / weightedBxSumSq;

            const residuals = By.map((y, i) => y - thetaIVW * Bx[i]);
            const weightedResidualSumSq = residuals.reduce((sum, r, i) => sum + currentWeights[i] * r * r, 0);

            let sigma_hat_squared;
            if (nsnps > 1) {
                sigma_hat_squared = weightedResidualSumSq / (nsnps - 1);
            } else {
                sigma_hat_squared = NaN;
            }

            rse = Math.sqrt(sigma_hat_squared);

            const base_thetaIVWse = Math.sqrt((1 / weightedBxSumSq) * sigma_hat_squared);

            if (currentModel === "random") {
                thetaIVWse = base_thetaIVWse / Math.min(rse, 1);
            } else {
                thetaIVWse = base_thetaIVWse / rse;
            }

            if (robust || penalized) {
                heterStat = NaN;
                pvalueHeterStat = NaN;
            } else {
                if (nsnps > 1) {
                    heterStat = (nsnps - 1) * (rse * rse);
                    pvalueHeterStat = 1 - jStat.chisquare.cdf(heterStat, nsnps - 1);
                } else {
                    heterStat = NaN;
                    pvalueHeterStat = NaN;
                }
            }

            fstat = Bx.reduce((sum, x, i) => sum + (x / Bxse[i]) * (x / Bxse[i]), 0) / nsnps;
        }
    }

    // Calculate p-value and confidence intervals
    if (distribution === "normal") {
        pvalue = 2 * jStat.normal.cdf(-Math.abs(thetaIVW / thetaIVWse), 0, 1);
```

```
            ciLower = ci_normal("l", thetaIVW, thetaIVWse, alpha);
            ciUpper = ci_normal("u", thetaIVW, thetaIVWse, alpha);
        } else if (distribution === "t-dist") {
            const df = Math.max(1, nsnps - 1);
            pvalue = 2 * jStat.studentt.cdf(-Math.abs(thetaIVW / thetaIVWse), df);
            ciLower = ci_t("l", thetaIVW, thetaIVWse, df, alpha);
            ciUpper = ci_t("u", thetaIVW, thetaIVWse, df, alpha);
        }

        return new IVW({
            Model: currentModel,
            Exposure: object.exposure,
            Outcome: object.outcome,
            Robust: effectiveRobust,
            Penalized: effectivePenalized,
            Correlation: effectiveCorrelation ? object.correlation : null,
            Estimate: thetaIVW,
            StdError: thetaIVWse,
            CILower: ciLower,
            CIUpper: ciUpper,
            SNPs: nsnps,
            Pvalue: pvalue,
            Alpha: alpha,
            RSE: rse,
            HeterStat: [heterStat, pvalueHeterStat],
            Fstat: fstat
        });
}
```

(2) Debiased Inverse Variance Weighted Method (dIVW)

The JavaScript codes for Debiased Inverse Variance Weighted Method (dIVW) are as follows:

```
/*
Method name and objective
Method: Debiased Inverse Variance Weighted Method (dIVW)
Objective: Compute Debiased Inverse Variance Weighted Method (dIVW) Mendelian Randomization estimate of the causal effect of an exposure on an outcome, with optional over-dispersion adjustment and diagnostics.

Main classes and functions
MRInput
Purpose: Encapsulates the MR input data.
Key fields: betaX, betaY (effect estimates for exposure and outcome per SNP), betaXse, betaYse (standard errors), correlation (optional matrix), exposure, outcome.
Behavior: Validates that the input arrays have the same length.
DIVW
```

Purpose: Container for the analysis results of dIVW.
Key fields: OverDispersion, Exposure, Outcome, Estimate, StdError, CILower, CIUpper, Alpha, Pvalue, SNPs, Condition.
Behavior: toString() provides a formatted summary of results.
mr_divw (main function)
Purpose: Perform the dIVW calculation given an MRInput and options.
Inputs:
object: MRInput instance containing betaX, betaY, betaXse, betaYse, etc.
options: Configuration with overDispersion (default true), alpha (default 0.05), diagnostics (default false).
Output: a DIVW instance with the estimate, standard error, confidence interval, p-value, number of SNPs, and a computed condition value. If no SNPs are present, returns NaN for statistics. It may throw errors if a zero standard error is encountered.

Required inputs and expected outputs

Required inputs
An MRInput instance with:
betaX, betaY: arrays of per-SNP exposure/outcome effect estimates.
betaXse, betaYse: arrays of corresponding standard errors.
correlation (optional): SNP correlation matrix.
exposure, outcome: strings naming the variables.
Optional options object:
overDispersion: boolean to enable over-dispersion adjustment.
alpha: significance level for CI and p-value.
diagnostics: boolean to enable diagnostic outputs.

Expected outputs
A DIVW object containing:
OverDispersion, Exposure, Outcome, Alpha, SNPs, Condition.
Estimate: point estimate of the causal effect (beta_dIVW).
StdError: standard error of the estimate (se_dIVW).
CILower, CIUpper: confidence interval bounds.
Pvalue: two-sided p-value.
SNPs: number of SNPs used, or 0 if none.
Behavior in edge cases:
If no SNPs, all statistics are NaN (with a warning).
If inputs lead to invalid computations (e.g., zero denominators), the function may throw or return NaN for certain outputs, depending on the situation.
*/

// --- MRInput Class ---

```
/**
  * Represents the input data for Mendelian Randomization analysis.
  */
class MRInput {
      constructor({betaX, betaY, betaXse, betaYse, correlation = null, exposure = "Exposure", outcome = "Outcome"}) {
```

```
            this.betaX = betaX || [];
            this.betaY = betaY || [];
            this.betaXse = betaXse || [];
            this.betaYse = betaYse || [];
            this.correlation = correlation; // Can be a math.js matrix or 2D array
            this.exposure = exposure;
            this.outcome = outcome;
            // Basic validation: all arrays must have the same length
            const lengths = new Set([this.betaX.length, this.betaY.length, this.betaXse.length, this.betaYse.length]);
            if (lengths.size !== 1) {
                throw new Error("MRInput: All input beta and SE arrays must have the same length.");
            }
        }
}

// --- DIVW Class ---

/**
 * Represents the results of a Debiased Inverse-Variance Weighted (dIVW) MR analysis.
 */
class DIVW {
        constructor({
            OverDispersion, Exposure, Outcome, Estimate, StdError, CILower, CIUpper, Alpha, Pvalue, SNPs, Condition
        }) {
            this.OverDispersion = OverDispersion;
            this.Exposure = Exposure;
            this.Outcome = Outcome;
            this.Estimate = Estimate;
            this.StdError = StdError;
            this.CILower = CILower;
            this.CIUpper = CIUpper;
            this.Alpha = Alpha;
            this.Pvalue = Pvalue;
            this.SNPs = SNPs;
            this.Condition = Condition;
        }

        // Pretty print method
        toString() {
            return `
=== Debiased IVW Results ===
Exposure: ${this.Exposure}
Outcome: ${this.Outcome}
SNPs: ${this.SNPs}
Over-dispersion: ${this.OverDispersion}
```

```
Alpha: ${this.Alpha}

Estimate: ${this.Estimate.toFixed(6)}
Std Error: ${this.StdError.toFixed(6)}
95% CI: [${this.CILower.toFixed(6)}, ${this.CIUpper.toFixed(6)}]
P-value: ${this.Pvalue.toExponential(4)}
Condition: ${this.Condition.toFixed(4)}
============================
            `;
        }
}

// --- Main mr_divw Function ---

/**
 * Calculates the debiased inverse-variance weighted estimate (dIVW).
 *
 * @param {MRInput} object - MRInput object containing genetic data.
 * @param {object} options - Configuration options for the analysis.
 * @param {boolean} [options.overDispersion=true] - Whether to account for over-dispersion.
 * @param {number} [options.alpha=0.05] - Significance level for confidence intervals.
 * @param {boolean} [options.diagnostics=false] - Whether to perform diagnostics (QQ plot equivalent).
 * @returns {DIVW} A DIVW object containing the analysis results.
 */
function mr_divw(object, options = {}) {
    const {
        overDispersion = true,
        alpha = 0.05,
        diagnostics = false
    } = options;

    const Bx = object.betaX;
    const By = object.betaY;
    const Bxse = object.betaXse;
    const Byse = object.betaYse;
    const nsnps = Bx.length;

    if (nsnps === 0) {
        // Handle case with no SNPs gracefully
        console.warn("No SNPs detected. Returning NaN for all statistics.");
        return new DIVW({
            OverDispersion: overDispersion,
            Exposure: object.exposure,
            Outcome: object.outcome,
            Estimate: NaN,
```

```
            StdError: NaN,
            CILower: NaN,
            CIUpper: NaN,
            Alpha: alpha,
            Pvalue: NaN,
            SNPs: 0,
            Condition: NaN
        });
    }

    // --- Input Validation ---
    // Check for zero standard errors early to prevent division by zero errors later
    for (let i = 0; i < nsnps; i++) {
        if (Bxse[i] === 0 || Byse[i] === 0) {
            throw new Error(`Standard error is zero at index ${i} (Bxse[${i}]=${Bxse[i]}, Byse[${i}]=${Byse[i]}). Cannot compute dIVW.`);
        }
    }

    // --- Calculations ---

    // SE.ratio <- object@betaXse / object@betaYse
    const SE_ratio = Bxse.map((bxse, i) => bxse / Byse[i]);

    // beta_dIVW <- sum(By*Bx/Byse^2) / sum((Bx^2-Bxse^2)/Byse^2)
    let sum_num_beta = 0;
    let sum_den_beta = 0;
    for (let i = 0; i < nsnps; i++) {
        const byse_sq = Byse[i] * Byse[i];
        sum_num_beta += (By[i] * Bx[i]) / byse_sq;
        sum_den_beta += (Bx[i] * Bx[i] - Bxse[i] * Bxse[i]) / byse_sq;
    }

    let beta_dIVW;
    if (sum_den_beta === 0) {
        console.warn("Denominator for beta_dIVW is zero. Setting beta_dIVW to NaN.");
        beta_dIVW = NaN;
    } else {
        beta_dIVW = sum_num_beta / sum_den_beta;
    }

    // mu <- Bx/Bxse
    const mu = Bx.map((bx, i) => bx / Bxse[i]);

    // condition <- (mean(mu^2)-1)*sqrt(length(Bx))
```

```
    const mu_sq = mu.map(val => val * val);
    const mean_mu_sq = jStat.mean(mu_sq);
    const condition = (mean_mu_sq - 1) * Math.sqrt(nsnps);

    // tau.square calculation (conditional on overDispersion)
    let tau_square;
    if (overDispersion === false) {
        tau_square = 0;
    } else {
        let sum_num_tau = 0;
        let sum_den_tau = 0;
        for (let i = 0; i < nsnps; i++) {
            const byse_sq = Byse[i] * Byse[i];
            const bxse_sq = Bxse[i] * Bxse[i];

            // Numerator of the fraction inside max(): (By-beta_dIVW*Bx)^2 - Byse^2 - beta_dIVW^2*Bxse^2
            const term_num_val = (By[i] - beta_dIVW * Bx[i]) * (By[i] - beta_dIVW * Bx[i]) - byse_sq - (beta_dIVW *
beta_dIVW * bxse_sq);
            sum_num_tau += term_num_val / byse_sq;

            // Denominator of the fraction inside max(): Byse^(-2)
            sum_den_tau += 1 / byse_sq;
        }

        if (sum_den_tau === 0) {
            console.warn("Denominator for tau.square calculation is zero. Setting tau.square to 0.");
            tau_square = 0;
        } else {
            tau_square = Math.max(0, sum_num_tau / sum_den_tau);
        }
    }

    // V1 calculation
    let V1 = 0;
    for (let i = 0; i < nsnps; i++) {
        const se_ratio_sq = SE_ratio[i] * SE_ratio[i];
        const se_ratio_pow4 = se_ratio_sq * se_ratio_sq; // SE.ratio^4
        const mu_i_sq = mu[i] * mu[i];
        const byse_sq = Byse[i] * Byse[i];

        V1 += (se_ratio_sq * mu_i_sq) +
            (beta_dIVW * beta_dIVW * se_ratio_pow4 * (mu_i_sq + 1)) +
            (tau_square * se_ratio_sq / byse_sq * mu_i_sq);
    }
```

```
// V2 calculation
let V2 = 0;
for (let i = 0; i < nsnps; i++) {
    const se_ratio_sq = SE_ratio[i] * SE_ratio[i];
    const mu_i_sq = mu[i] * mu[i];
    V2 += se_ratio_sq * (mu_i_sq - 1);
}

// se_dIVW <- sqrt(V1/V2^2)
let se_dIVW;
if (V2 === 0) {
    console.warn("V2 is zero. Cannot compute standard error. Setting se_dIVW to NaN.");
    se_dIVW = NaN;
} else {
    se_dIVW = Math.sqrt(V1 / (V2 * V2));
}

// CI and p-value calculation
let c_alpha, ciLower, ciUpper, pval;
if (isNaN(se_dIVW) || !isFinite(se_dIVW) || isNaN(beta_dIVW) || !isFinite(beta_dIVW)) {
    // If estimate or SE is invalid, set all related outputs to NaN
    c_alpha = NaN;
    ciLower = NaN;
    ciUpper = NaN;
    pval = NaN;
} else {
    c_alpha = jStat.normal.inv(1 - alpha / 2, 0, 1);
    ciLower = beta_dIVW - c_alpha * se_dIVW;
    ciUpper = beta_dIVW + c_alpha * se_dIVW;
    pval = 2 * jStat.normal.cdf(-Math.abs(beta_dIVW / se_dIVW), 0, 1);
}

// --- Diagnostics ---
if (diagnostics) {
    // In a non-visual JavaScript environment (like Node.js or a simple script without a browser DOM/canvas),
    // we cannot directly replicate these plotting functions.
    // We will compute the `t` values (standardized residuals) and log them,
    // suggesting that they can be used with an external plotting library.
    const t_values = [];
    for (let i = 0; i < nsnps; i++) {
        const byse_sq = Byse[i] * Byse[i];
        const bxse_sq = Bxse[i] * Bxse[i];
        const sqrt_denom = Math.sqrt(byse_sq + tau_square + beta_dIVW * beta_dIVW * bxse_sq);
        if (sqrt_denom === 0) {
            t_values.push(NaN); // Avoid division by zero
```

```
                } else {
                    t_values.push((By[i] - beta_dIVW * Bx[i]) / sqrt_denom);
                }
            }
            console.log("=== Diagnostics ===");
            console.log("Standardized residuals (t-values):", t_values);
            console.log("QQ-plot of standardized residuals cannot be drawn in this non-visual environment.");
            console.log("You can plot these 't-values' against theoretical normal quantiles using a plotting library if desired.");
            console.log("==================");
        }

        // --- Return Result ---
        return new DIVW({
            OverDispersion: overDispersion,
            Exposure: object.exposure,
            Outcome: object.outcome,
            Estimate: beta_dIVW,
            StdError: se_dIVW,
            CILower: ciLower,
            CIUpper: ciUpper,
            Alpha: alpha,
            Pvalue: pval,
            SNPs: nsnps,
            Condition: condition
        });
}
```

(3) The Constrained Maximum Likelihood Method (cML)

The JavaScript codes for The Constrained Maximum Likelihood Method (cML) are as follows:

```
/*
Method name and aim
Method: The Constrained Maximum Likelihood Method (cML)
Objective: Estimate causal effects in multivariate Mendelian Randomization under constraints. Supports extensions such as cML-BIC and cML-MA-BIC, data perturbation (DP), and model averaging (MA) to improve robustness. Provides goodness-of-fit statistics and confidence intervals.

Main components and functions (high-level)
Seeded RNG and normal RNG
mulberry32(seed): seeded pseudo-random number generator
makeNormalGenerator(seed): Box-Muller-based normal RNG
Utilities
Basic math and array helpers (mean, variance, transpose, deepClone, etc.)
Statistical distributions (pnorm, qnorm via a library like jStat)
Core interface
```

mr_cML(object, options): external entry point; accepts data and configuration, returns results
Internal estimation routines
cML_estimate_random(params): multiple random starts for a given K, selects best solution
cML_estimate(params): single-start iterative estimator; updates theta, b_vec, r_vec, etc.
cML_SdTheta(params): computes standard error of theta
Extensions and paths
Data Perturbation (DP): perturbs data across multiple K values, computes GOF-like metrics, combines results
Model Averaging (MA): weights results across K by BIC (or similar) to produce a pooled estimate and SE
Output construction
Builds a result object containing Exposure, Outcome, Estimate, StdError, Pvalue, CILower, CIUpper, SNPs, Alpha, plus optional DP/MA fields (GOF_p, BIC_invalid, etc.)

Required inputs and expected outputs

Required inputs (object)
betaX: array of exposure effects [βX1, βX2, ..., βXp]
betaY: array of outcome effects [βY1, βY2, ..., βYp]
betaXse: standard errors of betaX
betaYse: standard errors of betaY
exposure: string naming the exposure (e.g., "BMI")
outcome: string naming the outcome (e.g., "CHD")
Options (object)
MA: boolean, enable model averaging (default may be true)
DP: boolean, enable data perturbation path (default may be true)
K_vec: array of candidate K values (e.g., [0, 1, ..., p-1])
random_start: integer, extra random starts
num_pert: perturbation repetitions for DP
random_start_pert: random starts for perturbations
maxit: maximum iterations for estimators
random_seed: seed for reproducibility
n: sample size (for information criteria)
Alpha: significance level for CIs and p-values (default 0.05)

Expected outputs (object)
Exposure, Outcome: strings matching inputs
Estimate: estimated causal effect (theta) or MA/DP combined estimate
StdError: standard error of the estimate
Pvalue: two-sided p-value
CILower, CIUpper: confidence interval bounds
SNPs: number of SNPs (length of the input arrays)
-Alpha: used significance level
Optional DP/MA fields: GOF1_p, GOF2_p (goodness-of-fit p-values), BIC_invalid (indices with invalid BIC), flags for MA/DP status
Any additional fields for intermediate results or diagnostics as implemented
*/

```
// Utility: seeded PRNG (mulberry32) and normal RNG (Box-Muller using PRNG)
function mulberry32(seed) {
  let t = seed >>> 0;
  return function() {
    t += 0x6D2B79F5;
    let r = Math.imul(t ^ (t >>> 15), 1 | t);
    r ^= r + Math.imul(r ^ (r >>> 7), 61 | r);
    return ((r ^ (r >>> 14)) >>> 0) / 4294967296;
  };
}
function makeNormalGenerator(seed) {
  const u = seed == null ? Math.random : mulberry32(seed);
  let spare = null;
  return function(mean = 0, sd = 1) {
    if (spare !== null) {
      const val = spare;
      spare = null;
      return mean + sd * val;
    }
    let u1 = 0, u2 = 0;
    do {
      u1 = u();
      u2 = u();
    } while (u1 <= Number.EPSILON);
    const mag = Math.sqrt(-2.0 * Math.log(u1));
    const z0 = mag * Math.cos(2 * Math.PI * u2);
    const z1 = mag * Math.sin(2 * Math.PI * u2);
    spare = z1;
    return mean + sd * z0;
  };
}

// Helpers (using math.js and jStat when helpful)
const EPS = 1e-12;
function mean(arr) {
  if (!arr || arr.length === 0) return NaN;
  return arr.reduce((a, b) => a + b, 0) / arr.length;
}
function variance(arr) {
  const m = mean(arr);
  return arr.reduce((s, v) => s + (v - m) * (v - m), 0) / (arr.length - 1);
}
function rowMeans(matrix) {
  // matrix as array of arrays, each column is a column (R style used cbind) — we'll assume matrix is 2D: rows x cols
```

```
    // For our usage we will pass arrays of shape rows x cols (JS nested arrays).
    const rows = matrix.length;
    if (rows === 0) return [];
    const cols = matrix[0].length;
    const res = Array(rows).fill(0);
    for (let i = 0; i < rows; i++) {
        res[i] = mean(matrix[i]);
    }
    return res;
}
function colMeans(matrix) {
    // matrix as rows x cols
    const rows = matrix.length;
    if (rows === 0) return [];
    const cols = matrix[0].length;
    const res = Array(cols).fill(0);
    for (let j = 0; j < cols; j++) {
        let s = 0;
        for (let i = 0; i < rows; i++) s += matrix[i][j];
        res[j] = s / rows;
    }
    return res;
}
function transpose(matrix) {
    if (matrix.length === 0) return [];
    const rows = matrix.length, cols = matrix[0].length;
    const res = Array.from({ length: cols }, () => Array(rows));
    for (let i = 0; i < rows; i++) for (let j = 0; j < cols; j++) res[j][i] = matrix[i][j];
    return res;
}
function deepClone(x) { return JSON.parse(JSON.stringify(x)); }

// Normal CDF and inverse using jStat
function pnorm(x) { return jStat.normal.cdf(x, 0, 1); }
function qnorm(p) { return jStat.normal.inv(p, 0, 1); }

// Main exported function: mr_cML
// object must have: betaX (array), betaY (array), betaXse (array), betaYse (array), exposure (string), outcome (string)
function mr_cML(object, {
    MA = true,
    DP = true,
    K_vec = null,           // array of K values; default 0:(p-2)
    random_start = 0,
    num_pert = 200,
    random_start_pert = 0,
```

```
    maxit = 100,
    random_seed = 314,
    n,
    Alpha = 0.001
} = {}) {
    // Validate inputs
    const b_exp = object.betaX.slice();
    const b_out = object.betaY.slice();
    const se_exp = object.betaXse.slice();
    const se_out = object.betaYse.slice();
    const p = b_exp.length;
    if (!K_vec) K_vec = Array.from({ length: Math.max(1, p - 1) }, (_, i) => i);

    // Setup RNG
    const normalGen = makeNormalGenerator(random_seed == null ? null : random_seed);

    // Collect random-start results across K_vec
    const theta_v = [];
    const sd_v = [];
    const l_v = [];
    const invalid_mat = []; // each row per K

    for (let Ki = 0; Ki < K_vec.length; Ki++) {
        const K = K_vec[Ki];
        const rand_res = cML_estimate_random({
            b_exp, b_out, se_exp, se_out, K, random_start, maxit, rngSeed: random_seed
        });
        theta_v.push(rand_res.theta);
        sd_v.push(rand_res.se);
        l_v.push(rand_res.l);
        invalid_mat.push(rand_res.r_est.slice());
    }

    // Non-DP branch (no data perturbation)
    if (!DP) {
        if (!MA) {
            // cML-BIC
            const BIC_vec = l_v.map((l, idx) => Math.log(n) * K_vec[idx] + 2 * l);
            const minVal = Math.min(...BIC_vec);
            const BIC_adj = BIC_vec.map(v => v - minVal);
            const min_ind = BIC_adj.indexOf(Math.min(...BIC_adj));
            const BIC_theta = theta_v[min_ind];
            const BIC_se = sd_v[min_ind];
            const BIC_p = pnorm(-Math.abs(BIC_theta / BIC_se)) * 2;
            const BIC_invalid = invalid_mat[min_ind].map((v, i) => (v !== 0 ? i : -1)).filter(i => i >= 0);
```

```
        return {
          Exposure: object.exposure,
          Outcome: object.outcome,
          Estimate: BIC_theta,
          StdError: BIC_se,
          Pvalue: BIC_p,
          BIC_invalid,
          MA, DP,
          SNPs: p,
          Alpha,
          CILower: BIC_theta - qnorm(1 - Alpha / 2) * BIC_se,
          CIUpper: BIC_theta + qnorm(1 - Alpha / 2) * BIC_se
        };
      } else {
        // cML-MA-BIC
        const BIC_vec = l_v.map((l, idx) => Math.log(n) * K_vec[idx] + 2 * l);
        const minVal = Math.min(...BIC_vec);
        const BIC_adj = BIC_vec.map(v => v - minVal);
        const weight_vec = BIC_adj.map(v => Math.exp(-0.5 * v));
        const s = weight_vec.reduce((a, b) => a + b, 0);
        for (let i = 0; i < weight_vec.length; i++) weight_vec[i] /= s;
        const MA_BIC_theta = theta_v.reduce((acc, t, i) => acc + t * weight_vec[i], 0);
        const MA_BIC_se = weight_vec.reduce((acc, w, i) => acc + w * Math.sqrt(sd_v[i] * sd_v[i] + Math.pow(theta_v[i] -
MA_BIC_theta, 2)), 0);
        const MA_BIC_p = pnorm(-Math.abs(MA_BIC_theta / MA_BIC_se)) * 2;
        return {
          Exposure: object.exposure,
          Outcome: object.outcome,
          Estimate: MA_BIC_theta,
          StdError: MA_BIC_se,
          Pvalue: MA_BIC_p,
          MA, DP,
          SNPs: p,
          Alpha,
          CILower: MA_BIC_theta - qnorm(1 - Alpha / 2) * MA_BIC_se,
          CIUpper: MA_BIC_theta + qnorm(1 - Alpha / 2) * MA_BIC_se
        };
      }
    }

    // DP branch: data perturbation
    // Generate perturbations
    // We'll simulate num_pert perturbed datasets and for each K compute MLE via cML_estimate
    const rand_pert_theta = []; // rows: K index; columns: perturbation index
    const rand_pert_sd = [];
```

```
  const rand_pert_l = [];

  // initialize arrays of length K_vec.length, each containing arrays for each perturbation
  for (let i = 0; i < K_vec.length; i++) {
    rand_pert_theta.push([]);
    rand_pert_sd.push([]);
    rand_pert_l.push([]);
  }

  for (let DPind = 0; DPind < num_pert; DPind++) {
    // perturb b_exp and b_out
    const b_exp_new = new Array(p);
    const b_out_new = new Array(p);
    for (let i = 0; i < p; i++) {
      b_exp_new[i] = b_exp[i] + normalGen(0, 1) * se_exp[i];
      b_out_new[i] = b_out[i] + normalGen(0, 1) * se_out[i];
    }
    // for each K, compute MLE
    for (let Ki = 0; Ki < K_vec.length; Ki++) {
      const K = K_vec[Ki];
      const MLE_result = cML_estimate({
        b_exp: b_exp_new, b_out: b_out_new, se_exp, se_out, K,
        initial_theta: 0, initial_mu: b_exp_new.slice(), maxit, rngSeed: random_seed == null ? null : random_seed + DPind
      });
      rand_pert_theta[Ki].push(MLE_result.theta);
      rand_pert_sd[Ki].push(cML_SdTheta({
        b_exp: b_exp_new, b_out: b_out_new, se_exp, se_out,
        theta: MLE_result.theta, b_vec: MLE_result.b_vec, r_vec: MLE_result.r_vec
      }));
      // negative log-likelihood for perturbed data
      const neg_l = b_exp_new.reduce((acc, be, i) => acc + Math.pow(be - MLE_result.b_vec[i], 2) / (2 * Math.pow(se_exp[i],
2)), 0)
        + b_out_new.reduce((acc, bo, i) => acc + Math.pow(bo - MLE_result.theta * MLE_result.b_vec[i] -
MLE_result.r_vec[i], 2) / (2 * Math.pow(se_out[i], 2)), 0);
      rand_pert_l[Ki].push(neg_l);
    }
  }

  // Convert collected arrays into summary vectors across perturbations
  // For each K compute row means etc.
  const theta_pt_v = rand_pert_theta.map(arr => mean(arr));
  const sd_pt_v = rand_pert_sd.map(arr => {
    // standard deviation of perturbation estimates (sd across perturbations)
    if (arr.length <= 1) return NaN;
    return Math.sqrt(variance(arr));
```

```
  });
  const l_pt_v = rand_pert_l.map(arr => mean(arr));
  const var_mat = rand_pert_sd.map(arr => arr.map(v => v * v)); // per-K, array of variance across perturbs

  // GOF1 and GOF2: need BIC from original runs to find selected K (min BIC)
  const BIC_vec_orig = l_v.map((l, idx) => Math.log(n) * K_vec[idx] + 2 * l);
  const minValOrig = Math.min(...BIC_vec_orig);
  const BIC_adj_orig = BIC_vec_orig.map(v => v - minValOrig);
  const min_ind = BIC_adj_orig.indexOf(Math.min(...BIC_adj_orig));

  const pt_sd_v = sd_pt_v[min_ind];
  const origin_sd_v = sd_v[min_ind];
  // more_var = variance of var_mat[min_ind,]
  const more_var = variance(var_mat[min_ind]);
  const x = rand_pert_theta[min_ind].slice();
  const numer_perturb = x.length;
  const sd_x = Math.sqrt(
    (mean(x.map(v => Math.pow(v - mean(x), 4))) - (numer_perturb - 3) / (numer_perturb - 1) * Math.pow(variance(x), 2)) /
numer_perturb
    + more_var
  );
  const Tstat1 = (origin_sd_v * origin_sd_v - pt_sd_v * pt_sd_v) / sd_x;
  const GOF1_p = pnorm(-Math.abs(Tstat1)) * 2;

  const Tstat2 = (origin_sd_v * origin_sd_v - pt_sd_v * pt_sd_v) / Math.sqrt(2 / (numer_perturb - 1) * Math.pow(pt_sd_v, 4) +
more_var);
  const GOF2_p = pnorm(-Math.abs(Tstat2)) * 2;

  // Final outputs depend on MA flag
  if (!MA) {
    // cML-BIC-DP
    const BIC_vec_DP = l_pt_v.map((l, idx) => Math.log(n) * K_vec[idx] + 2 * l);
    const minValDP = Math.min(...BIC_vec_DP);
    const BIC_adj_DP = BIC_vec_DP.map(v => v - minValDP);
    const min_ind_DP = BIC_adj_DP.indexOf(Math.min(...BIC_adj_DP));
    const BIC_DP_theta = theta_pt_v[min_ind_DP];
    const BIC_DP_se = sd_pt_v[min_ind_DP];
    const BIC_DP_p = pnorm(-Math.abs(BIC_DP_theta / BIC_DP_se)) * 2;
    return {
      Exposure: object.exposure,
      Outcome: object.outcome,
      Estimate: BIC_DP_theta,
      StdError: BIC_DP_se,
      Pvalue: BIC_DP_p,
      MA, DP,
```

```
        GOF1_p, GOF2_p,
        SNPs: p,
        Alpha,
        CILower: BIC_DP_theta - qnorm(1 - Alpha / 2) * BIC_DP_se,
        CIUpper: BIC_DP_theta + qnorm(1 - Alpha / 2) * BIC_DP_se
      };
    } else {
      // cML-MA-BIC-DP
      const BIC_vec_DP = l_pt_v.map((l, idx) => Math.log(n) * K_vec[idx] + 2 * l);
      const minValDP = Math.min(...BIC_vec_DP);
      const BIC_adj_DP = BIC_vec_DP.map(v => v - minValDP);
      const weight_vec = BIC_adj_DP.map(v => Math.exp(-0.5 * v));
      const sW = weight_vec.reduce((a, b) => a + b, 0);
      for (let i = 0; i < weight_vec.length; i++) weight_vec[i] /= sW;
      const MA_BIC_DP_theta = theta_pt_v.reduce((acc, t, i) => acc + t * weight_vec[i], 0);
      const MA_BIC_DP_se = weight_vec.reduce((acc, w, i) => acc + w * Math.sqrt(Math.pow(sd_pt_v[i], 2) +
Math.pow(theta_pt_v[i] - MA_BIC_DP_theta, 2)), 0);
      const MA_BIC_DP_p = pnorm(-Math.abs(MA_BIC_DP_theta / MA_BIC_DP_se)) * 2;
      return {
        Exposure: object.exposure,
        Outcome: object.outcome,
        Estimate: MA_BIC_DP_theta,
        StdError: MA_BIC_DP_se,
        Pvalue: MA_BIC_DP_p,
        MA, DP,
        GOF1_p, GOF2_p,
        SNPs: p,
        Alpha,
        CILower: MA_BIC_DP_theta - qnorm(1 - Alpha / 2) * MA_BIC_DP_se,
        CIUpper: MA_BIC_DP_theta + qnorm(1 - Alpha / 2) * MA_BIC_DP_se
      };
    }
}

// cML_estimate_random: multiple random starts, pick best negative log-likelihood
// Input object with fields b_exp, b_out, se_exp, se_out, K, random_start, maxit, rngSeed
function cML_estimate_random({ b_exp, b_out, se_exp, se_out, K, random_start = 0, maxit = 100, rngSeed = 314 } = {}) {
    const p = b_exp.length;
    const min_theta_range = Math.min(...b_out.map((v, i) => v / b_exp[i]));
    const max_theta_range = Math.max(...b_out.map((v, i) => v / b_exp[i]));

    const theta_candidates = [];
    const sd_candidates = [];
    const l_candidates = [];
    const invalid_candidates = [];
```

```
  const normalGen = makeNormalGenerator(rngSeed);

  for (let random_ind = 0; random_ind < (1 + random_start); random_ind++) {
    let initial_theta, initial_mu;
    if (random_ind === 0) {
      initial_theta = 0;
      initial_mu = Array(p).fill(0);
    } else {
      initial_theta = min_theta_range + (max_theta_range - min_theta_range) * Math.random();
      initial_mu = new Array(p);
      for (let i = 0; i < p; i++) initial_mu[i] = normalGen(b_exp[i], se_exp[i]);
    }
    const MLE_result = cML_estimate({
      b_exp, b_out, se_exp, se_out,
      K, initial_theta, initial_mu, maxit
    });


const Neg_l = b_exp.reduce((acc, be, i) => acc + Math.pow(be - MLE_result.b_vec[i], 2) / (2 * Math.pow(se_exp[i], 2)), 0)
  + b_out.reduce((acc, bo, i) => acc + Math.pow(bo - MLE_result.theta * MLE_result.b_vec[i] - MLE_result.r_vec[i], 2) / (2 *
Math.pow(se_out[i], 2)), 0);

const sd_theta = cML_SdTheta({
  b_exp, b_out, se_exp, se_out,
  theta: MLE_result.theta, b_vec: MLE_result.b_vec, r_vec: MLE_result.r_vec
});

theta_candidates.push(MLE_result.theta);
sd_candidates.push(sd_theta);
l_candidates.push(Neg_l);
invalid_candidates.push(MLE_result.r_vec.slice());

  }

  if (sd_candidates.some(v => Number.isNaN(v))) {
    console.warn("May not converge to minimums with some given start points and maximum number of iteration, lead to
Fisher Information matrices not positive definite. Could try increasing number of iterations (maxit) or try different start points.
Note: If multiple random start points are used, this warning does not likely affect result.");
  }

  let min_neg_l_index = 0;
  for (let i = 1; i < l_candidates.length; i++) if (l_candidates[i] < l_candidates[min_neg_l_index]) min_neg_l_index = i;

  return {
```

```
      theta: theta_candidates[min_neg_l_index],
      se: sd_candidates[min_neg_l_index],
      l: l_candidates[min_neg_l_index],
      r_est: invalid_candidates[min_neg_l_index]
    };
}

// cML_estimate: iterate to convergence updating v_bg (r_vec), mu_vec (b_vec), theta
// Input: object with b_exp, b_out, se_exp, se_out, K, initial_theta, initial_mu, maxit
function cML_estimate({ b_exp, b_out, se_exp, se_out, K, initial_theta = 0, initial_mu = null, maxit = 100 } = {}) {
    const p = b_exp.length;
    let theta = initial_theta;
    let theta_old = theta - 1;
    let mu_vec = initial_mu ? initial_mu.slice() : Array(p).fill(0);
    if (!initial_mu) for (let i = 0; i < p; i++) mu_vec[i] = 0;

    let ite_ind = 0;
    let v_bg = Array(p).fill(0);

    while (Math.abs(theta_old - theta) > 1e-7 && ite_ind < maxit) {
        theta_old = theta;
        ite_ind += 1;



if (K > 0) {
    const v_importance = new Array(p);
    for (let i = 0; i < p; i++) v_importance[i] = Math.pow((b_out[i] - theta * mu_vec[i]) / se_out[i], 2);
    // pick top K indices by v_importance
    const idxs = Array.from({ length: p }, (_, i) => i);
    idxs.sort((a, b) => v_importance[b] - v_importance[a]);
    const nonzero_bg_ind = idxs.slice(0, Math.min(K, p)).sort((a, b) => a - b);
    v_bg = Array(p).fill(0);
    for (const ii of nonzero_bg_ind) v_bg[ii] = b_out[ii] - theta * mu_vec[ii];
} else {
    v_bg = Array(p).fill(0);
}

// update mu_vec
for (let i = 0; i < p; i++) {
    const num = b_exp[i] / (se_exp[i] * se_exp[i]) + theta * (b_out[i] - v_bg[i]) / (se_out[i] * se_out[i]);
    const den = 1 / (se_exp[i] * se_exp[i]) + (theta * theta) / (se_out[i] * se_out[i]);
    mu_vec[i] = num / den;
}

// update theta
```

```
let numTheta = 0, denTheta = 0;
for (let i = 0; i < p; i++) {
   numTheta += (b_out[i] - v_bg[i]) * mu_vec[i] / (se_out[i] * se_out[i]);
   denTheta += (mu_vec[i] * mu_vec[i]) / (se_out[i] * se_out[i]);
}
theta = numTheta / denTheta;

   }

   // one more step for v_bg and mu
   if (K > 0) {
      const nonzero_ind = [];
      for (let i = 0; i < p; i++) if (Math.abs(v_bg[i]) > 0) nonzero_ind.push(i);
      for (const idx of nonzero_ind) mu_vec[idx] = b_exp[idx];
      for (const idx of nonzero_ind) v_bg[idx] = b_out[idx] - theta * mu_vec[idx];
   }

   return { theta, b_vec: mu_vec, r_vec: v_bg };
}

// cML_SdTheta: compute standard error of theta.
// Input object: b_exp,b_out,se_exp,se_out,theta,b_vec,r_vec
function cML_SdTheta({ b_exp, b_out, se_exp, se_out, theta, b_vec, r_vec } = {}) {
   const p = b_exp.length;
   const nonzero_ind = [];
   const zero_ind = [];
   for (let i = 0; i < p; i++) (Math.abs(r_vec[i]) > 0 ? nonzero_ind.push(i) : zero_ind.push(i));

   let sum1 = 0;
   for (const i of zero_ind) sum1 += Math.pow(b_vec[i], 2) / Math.pow(se_out[i], 2);

   let sum2 = 0;
   for (const i of zero_ind) {
      const term = Math.pow(2 * b_vec[i] * theta - b_out[i], 2) / Math.pow(se_out[i], 4)
         * 1 / (1 / Math.pow(se_exp[i], 2) + Math.pow(theta, 2) / Math.pow(se_out[i], 2));
      sum2 += term;
   }

   const VarTheta = 1 / (sum1 - sum2);

   if (VarTheta <= 0) return NaN;
   return Math.sqrt(VarTheta);
}
```

(4) The Heterogeneity Penalization Model (Hetpen)

The JavaScript codes for The Heterogeneity Penalization Model (Hetpen) are as follows:

```
/*
Method name and objective
Name: The Heterogeneity Penalization Model (Hetpen)
Objective: A Mendelian Randomization method that uses multiple SNPs and heterogeneity-penalized weights to estimate a causal effect of an exposure on an outcome, aiming for robust and stable inference.

Main classes and functions
MRInput
Purpose: Encapsulates input data for MR analysis.
Fields: betaX, betaY, betaXse, betaYse, exposure, outcome, correlation (optional).
MRHetPen
Purpose: Container for the analysis results.
Fields: Exposure, Outcome, Prior, Estimate, CIRange, CILower, CIUpper, CIMin, CIMax, CIStep, SNPs, Alpha.
modelPrior(modelSize, nObs, probValidInst)
Purpose: Compute a prior (model) weight for a subset size.
combinations(arr, k)
Purpose: Generate all k-element combinations from an array (used to form SNP subsets).
hetWeight(probValidInst, bx, by, byse)
Purpose: Core computation.
Behavior: For every non-empty SNP subset, compute:
IVW estimate and standard error
Heterogeneity exponent
Log theta se sum
Heterogeneity-penalized weight
Returns: three arrays — weights, estimates, and standard errors for each subset.
calculateCIs(CIRange, CIStep)
Purpose: Derive confidence-interval bounds from a range of grid points.
mrHetPen(mrInputObject, options)
Purpose: Main entry point to run the analysis.
Options: prior, CIMin, CIMax, CIStep, alpha (default 0.05).
Behavior: Validates SNP count (max ~30), runs hetWeight, normalizes weights, forms a grid over CIMin..CIMax, computes a likelihood-like score across the grid, identifies the modal estimate and the confidence interval, returns an MRHetPen object.

Required inputs and expected outputs

Required inputs
MRInput object containing:
betaX (SNP effects on exposure)
betaY (SNP effects on outcome)
betaXse (SEs for betaX)
betaYse (SEs for betaY)
exposure (string label)
```

outcome (string label)
optional correlation
Options for mrHetPen: prior, CIMin, CIMax, CIStep, alpha

Expected outputs
An MRHetPen object with:
Exposure, Outcome
Prior, Estimate (the MR-HetPen point estimate)
CIRange (grid of plausible estimates)
CILower, CIUpper (confidence-interval bounds)
CIMin, CIMax, CIStep, SNPs (count of SNPs used), Alpha

Notes
The method performs exhaustive enumeration of SNP subsets (up to practical limits, max around 30 SNPs in this implementation).
Runtime increases with the number of SNPs; a heads-up warning is emitted for larger sets.
*/

```
// --- MRInput Class ---
/**
 * Represents the input data for Mendelian Randomization analysis.
 */
class MRInput {
    constructor(betaX, betaY, betaXse, betaYse, exposure, outcome, correlation = null) {
        const nsnps = betaX.length;
        if (nsnps === 0 || betaY.length !== nsnps || betaXse.length !== nsnps || betaYse.length !== nsnps) {
            throw new Error("Input arrays (betaX, betaY, betaXse, betaYse) must have the same non-zero length.");
        }
        this.betaX = betaX;
        this.betaY = betaY;
        this.betaXse = betaXse;
        this.betaYse = betaYse;
        this.exposure = exposure;
        this.outcome = outcome;
        this.correlation = correlation;
    }
}

// --- MRHetPen Class ---
/**
 * Represents the results of the MR-HetPen analysis.
 */
class MRHetPen {
    constructor(exposure, outcome, prior, estimate, ciRange, ciLower, ciUpper,
                ciMin, ciMax, ciStep, snps, alpha) {
```

```
            this.Exposure = exposure;
            this.Outcome = outcome;
            this.Prior = prior;
            this.Estimate = estimate;
            this.CIRange = ciRange;
            this.CILower = ciLower;
            this.CIUpper = ciUpper;
            this.CIMin = ciMin;
            this.CIMax = ciMax;
            this.CIStep = ciStep;
            this.SNPs = snps;
            this.Alpha = alpha;
        }
}

// --- Helper Functions ---

/**
 * Prior weight function.
 */
const modelPrior = (modelSize, nObs, probValidInst) => {
    return Math.pow(probValidInst, modelSize) * Math.pow((1 - probValidInst), (nObs - modelSize));
};

/**
 * Generate all combinations of k elements from an array.
 */
const combinations = (arr, k) => {
    const result = [];
    if (k < 0 || k > arr.length) return result;
    if (k === 0) return [[]];
    if (k === arr.length) return [arr];

    const recurse = (currentCombo, startIndex) => {
        if (currentCombo.length === k) {
            result.push([...currentCombo]);
            return;
        }
        for (let i = startIndex; i < arr.length; i++) {
            currentCombo.push(arr[i]);
            recurse(currentCombo, i + 1);
            currentCombo.pop();
        }
    };
```

```
    recurse([], 0);
    return result;
};

/**
 * Heterogeneity-penalized weight function.
 */
const hetWeight = (probValidInst, bx, by, byse) => {
    const J = by.length;

    // Pre-calculate common terms
    const thetaEst = by.map((y, i) => bx[i] === 0 ? (y === 0 ? 0 : Infinity) : y / bx[i]);
    const thetaSe = byse.map((y_se, i) => bx[i] === 0 ? Infinity : Math.abs(y_se / bx[i]));
    const thetaSeSq = thetaSe.map(se => se * se);
    const logThetaSe = thetaSe.map(se => Math.log(se));

    const tmp1 = by.map((y, i) => byse[i] === 0 ? Infinity : y / byse[i]);
    const tmp2 = bx.map((x, i) => byse[i] === 0 ? Infinity : x / byse[i]);

    const allModelsHetWeight = [];
    const allModelsEst = [];
    const allModelsSeest = [];

    const snpIndices = Array.from({ length: J }, (_, i) => i);

    // Iterate through all possible subset sizes
    for (let n = 1; n <= J; n++) {
        const perms = combinations(snpIndices, n);

        for (const subset of perms) {
            const subsetThetaEst = subset.map(idx => thetaEst[idx]);
            const subsetThetaSe = subset.map(idx => thetaSe[idx]);
            const subsetThetaSeSq = subset.map(idx => thetaSeSq[idx]);
            const subsetLogThetaSe = subset.map(idx => logThetaSe[idx]);
            const subsetTmp1 = subset.map(idx => tmp1[idx]);
            const subsetTmp2 = subset.map(idx => tmp2[idx]);

            // Calculate IVW Estimate
            const ivwSumNum = subset.reduce((sum, idx, i) => sum + subsetThetaEst[i] / subsetThetaSeSq[i], 0);
            const ivwSumDenom = subset.reduce((sum, idx, i) => sum + 1 / subsetThetaSeSq[i], 0);
            const estIvw = ivwSumDenom === 0 ? NaN : ivwSumNum / ivwSumDenom;

            // Calculate Standard Error
            let seestVal;
            if (n === 1) {
```

```
                seestVal = subsetThetaSe[0];
            } else {
                const psiHatNumeratorSum = subset.reduce((sum, idx, i) => {
                    const term = subsetTmp1[i] - estIvw * subsetTmp2[i];
                    return sum + term * term;
                }, 0);

                const psiHatSq = (1 / (n - 1)) * psiHatNumeratorSum;
                const psiHatVal = Math.sqrt(Math.max(1, psiHatSq));
                seestVal = ivwSumDenom === 0 ? NaN : psiHatVal / Math.sqrt(ivwSumDenom);
            }

            // Calculate Heterogeneity Exponent
            const hetExponentSum = subset.reduce((sum, idx, i) => {
                const term = subsetThetaEst[i] - estIvw;
                return sum + term * term / subsetThetaSeSq[i];
            }, 0);

            // Calculate Log Theta Se Sum
            const logThetaSeSum = subset.reduce((sum, idx, i) => sum + subsetLogThetaSe[i], 0);

            // Calculate Heterogeneity-Penalized Weight
            const prVal = modelPrior(n, J, probValidInst);
            const hetWeightVal = Math.exp(-(logThetaSeSum + 0.5 * hetExponentSum)) * prVal;

            allModelsEst.push(estIvw);
            allModelsSeest.push(seestVal);
            allModelsHetWeight.push(hetWeightVal);
        }
    }

    return [allModelsHetWeight, allModelsEst, allModelsSeest];
};

/**
 * Calculate confidence intervals from a range of values.
 */
const calculateCIs = (CIRange, CIStep) => {
    if (CIRange.length === 0) {
        return { CILower: [], CIUpper: [] };
    }

    const CILower = [];
    const CIUpper = [];
```

```
    let currentSegmentStart = CIRange[0];
    let currentSegmentEnd = CIRange[0];

    for (let i = 1; i < CIRange.length; i++) {
        if (CIRange[i] - CIRange[i-1] > 1.01 * CIStep) {
            CILower.push(currentSegmentStart);
            CIUpper.push(currentSegmentEnd);
            currentSegmentStart = CIRange[i];
        }
        currentSegmentEnd = CIRange[i];
    }

    CILower.push(currentSegmentStart);
    CIUpper.push(currentSegmentEnd);

    return { CILower, CIUpper };
};

/**
 * Main MR-HetPen function.
 */
const mrHetPen = (mrInputObject, options = {}) => {
    const {
        prior = 0.5,
        CIMin = -1,
        CIMax = 1,
        CIStep = 0.001,
        alpha = 0.05
    } = options;

    const Bx = mrInputObject.betaX;
    const By = mrInputObject.betaY;
    const Bxse = mrInputObject.betaXse;
    const Byse = mrInputObject.betaYse;

    const nsnps = Bx.length;

    if (nsnps === 0) {
        console.error("No genetic variants provided. Cannot run mr_hetpen.");
        return null;
    }
    if (nsnps > 30) {
        console.error("Too many genetic variants for this version of the method to run. Max 30 SNPs allowed.");
        return null;
    }
```

```
    if (nsnps > 25) {
        console.warn("Method is likely to take a long time to run for >25 SNPs. Please be patient.");
    }

    const results = hetWeight(prior, Bx, By, Byse);
    const hetWeightRaw = results[0];
    const est = results[1];
    const seest = results[2];

    const sumHetWeight = hetWeightRaw.reduce((sum, w) => sum + w, 0);
    const hetWeightNorm = hetWeightRaw.map(w => sumHetWeight === 0 ? 0 : w / sumHetWeight);

    // Generate the grid
    const point = Array.from({ length: Math.floor((CIMax - CIMin) / CIStep) + 1 }, (_, i) => CIMin + i * CIStep);

    // Calculate sumlik for each point
    const sumlik = point.map(p => {
        let currentSum = 0;
        for (let i = 0; i < hetWeightNorm.length; i++) {
            if (!isNaN(est[i]) && !isNaN(seest[i]) && seest[i] > 0) {
                currentSum += hetWeightNorm[i] * jStat.normal.pdf(p, est[i], seest[i]);
            }
        }
        return currentSum;
    });

    // Find the confidence interval
    const logSumlik = sumlik.map(s => s > 0 ? Math.log(s) : -Infinity);
    const maxLogSumlik = Math.max(...logSumlik.filter(isFinite));

    const chiSqQuantile = jStat.chisquare.inv(1 - alpha, 1);

    const whichin = logSumlik.map((l, i) => (l !== -Infinity && 2 * l > (2 * maxLogSumlik - chiSqQuantile)) ? i : -1)
                             .filter(idx => idx !== -1);

    let betaHetPen = NaN;
    if (logSumlik.some(isFinite)) {
        const maxLogSumlikIndex = logSumlik.indexOf(maxLogSumlik);
        betaHetPen = CIMin + CIStep * maxLogSumlikIndex;
    } else {
        console.warn("All likelihoods were zero or effectively zero. Cannot determine modal estimate.");
    }

    let CIRange = [];
    if (whichin.length > 0) {
```

```
        CIRange = whichin.map(idx => point[idx]);
    }

    const { CILower, CIUpper } = calculateCIs(CIRange, CIStep);

    return new MRHetPen(
        mrInputObject.exposure, mrInputObject.outcome, prior, betaHetPen,
        CIRange, CILower, CIUpper, CIMin, CIMax, CIStep, nsnps, alpha
    );
};
```

(5) The Single-Variable Lasso Method (Lasso)

The JavaScript codes for The Single-Variable Lasso Method (Lasso) are as follows:

```
/*
Method name and objective
The Single-Variable Lasso Method (Lasso)
Objective: A Mendelian randomization analysis that uses LASSO (L1) regularization to select valid genetic instruments and estimate the causal effect of an exposure on an outcome.

Main classes and functions
Distribution helpers: pnorm, qnorm, pt, qt, pchisq, qchisq (interfaces to normal, t, and chi-square CDFs/Inverses via jStat).
Confidence-interval helpers: ci_normal and ci_t to compute lower/upper bounds and handle normal or t-based intervals.
LASSO solvers:
lassoCoordinateDescent: standard coordinate-descent solver for LASSO (one-pass updates with soft-thresholding).
lassoCoordinateDescentWarmStart: similar solver that starts from a provided coefficient vector.
lassoPath: builds a sequence of LASSO fits across a lambda path.
Main MR-LASSO routine (mr_lasso):
Takes SNP-exposure and SNP-outcome effects (and their standard errors) and optional SNP identifiers.
Transforms inputs into standardized forms (Bx, By, Byse, Bxse) and constructs a reduced-form matrix (xlas, ylas) that accounts for overlap/collinearity among instruments.
If a lambda is provided, fits a single LASSO model; otherwise builds a lambda path and selects a lambda using a data-driven heuristic based on residual standard errors and a simple “heterogeneity” rule.
After LASSO selection, computes a post-LASSO causal estimate (post_est) and its standard error (post_se) using a weighted regression on the selected instruments.
Computes confidence intervals and p-values according to the chosen distribution (normal or t with appropriate degrees of freedom).
Produces a results object that also reports which SNPs were kept as valid instruments and the LASSO-related regression coefficients.

Required inputs and expected outputs

Required inputs (object to mr_lasso):
betaX: array of SNP effects on the exposure (size n SNPs)
betaY: array of SNP effects on the outcome (size n)
```

betaXse: standard errors of betaX (size n)
betaYse: standard errors of betaY (size n)
snps: optional array of SNP identifiers (length n)
exposure: string name of the exposure
outcome: string name of the outcome
Optional inputs (options object):
distribution: "normal" or "t-dist" (default "normal")
alpha: significance level for CIs and p-values (default 0.05)
lambda: array of one or more lambda values for explicit LASSO fitting; if empty/missing, an automatic lambda path is used

Output (object):
Exposure: name of the exposure
Outcome: name of the outcome
Estimate: the post-LASSO causal effect estimate
StdError: standard error of the estimate
CILower / CIUpper: confidence interval bounds
Alpha: alpha used for CIs
Pvalue: p-value for the null of no causal effect (based on chosen distribution)
SNPs: total number of SNPs supplied
RegEstimate: the interception/overall regression estimate from the LASSO step
RegIntercept: array of regression intercepts (length equal to number of SNPs)
Valid: number of instruments deemed valid after selection
ValidSNPs: identifiers of the valid SNPs (or indices if identifiers missing)
Lambda: the lambda value actually used (selected by the procedure)

In short, this code provides a small pipeline to perform MR analysis with LASSO-driven instrument selection, including data preparation, regularized fitting, post-selection causal estimation, and a final results object with both the instrument selection details and the causal effect estimates.
*/

```
// ==================== Dependencies ====================
// Assumes jStat, Math.js, and seedrandom are loaded
// <script src="jstat.min.js"></script>
// <script src="math.min.js"></script>
// <script src="seedrandom.min.js"></script>

// ==================== Helper Functions ====================

function pnorm(x) { return jStat.normal.cdf(x, 0, 1); }
function qnorm(p) { return jStat.normal.inv(p, 0, 1); }
function pt(x, df) { return jStat.studentt.cdf(x, df); }
function qt(p, df) { return jStat.studentt.inv(p, df); }
function pchisq(x, df, lowerTail = true) {
  var c = jStat.chisquare.cdf(x, df);
  return lowerTail ? c : 1 - c;
```

```
}
function qchisq(p, df) { return jStat.chisquare.inv(p, df); }

function sum(arr) { return arr.reduce((s, v) => s + v, 0); }
function mean(arr) { return arr.length ? sum(arr) / arr.length : NaN; }
function cloneArray(a) { return a.slice(); }
function isNumber(x) { return typeof x === "number" && isFinite(x); }

function diagFromArray(a) {
   var n = a.length;
   var M = Array(n);
   for (var i = 0; i < n; i++) {
      M[i] = Array(n).fill(0);
      M[i][i] = a[i];
   }
   return M;
}

function ci_normal(type, theta, thetaSe, alpha) {
   if (!isNumber(theta) || !isNumber(thetaSe)) return NaN;
   var z = qnorm(1 - alpha / 2);
   if (type === "l") return theta - z * thetaSe;
   if (type === "u") return theta + z * thetaSe;
   return NaN;
}

function ci_t(type, theta, thetaSe, df, alpha) {
   if (!isNumber(theta) || !isNumber(thetaSe)) return NaN;
   var tval = qt(1 - alpha / 2, df);
   if (type === "l") return theta - tval * thetaSe;
   if (type === "u") return theta + tval * thetaSe;
   return NaN;
}

// ==================== LASSO Solver ====================

function lassoCoordinateDescent(X, y, lambda, tol = 1e-6, maxIter = 10000) {
   var n = X.length;
   if (n === 0) return [];
   var p = X[0].length;

   var colNorm2 = Array(p).fill(0);
   var Xty = Array(p).fill(0);
   for (var j = 0; j < p; j++) {
      var s = 0;
```

```
    var sty = 0;
    for (var i = 0; i < n; i++) {
      var xij = X[i][j];
      s += xij * xij;
      sty += xij * y[i];
    }
    colNorm2[j] = s;
    Xty[j] = sty;
  }

  var beta = Array(p).fill(0);
  var residual = cloneArray(y);
  var iter = 0;
  var converged = false;

  while (!converged && iter < maxIter) {
    iter++;
    var maxChange = 0;
    for (var j = 0; j < p; j++) {
      var xj = [];
      for (var i = 0; i < n; i++) xj.push(X[i][j]);

      var rho = 0;
      for (var i = 0; i < n; i++) rho += xj[i] * residual[i];
      rho += colNorm2[j] * beta[j];

      var z = colNorm2[j];
      var newBeta = 0;
      if (z === 0) {
        newBeta = 0;
      } else {
        var thresh = lambda;
        if (rho > thresh) newBeta = (rho - thresh) / z;
        else if (rho < -thresh) newBeta = (rho + thresh) / z;
        else newBeta = 0;
      }

      var delta = newBeta - beta[j];
      if (delta !== 0) {
        for (var i = 0; i < n; i++) residual[i] -= xj[i] * delta;
        maxChange = Math.max(maxChange, Math.abs(delta));
        beta[j] = newBeta;
      }
    }
    if (maxChange < tol) converged = true;
```

```
    }
    return beta;
}

function lassoCoordinateDescentWarmStart(X, y, lambda, betaInit, tol = 1e-6, maxIter = 10000) {
    var n = X.length;
    var p = X[0].length;
    var beta = betaInit.slice();

    var residual = cloneArray(y);
    for (var j = 0; j < p; j++) {
        var bj = beta[j];
        if (bj !== 0) {
            for (var i = 0; i < n; i++) residual[i] -= X[i][j] * bj;
        }
    }

    var colNorm2 = Array(p).fill(0);
    for (var j = 0; j < p; j++) {
        var s = 0;
        for (var i = 0; i < n; i++) {
            var xij = X[i][j];
            s += xij * xij;
        }
        colNorm2[j] = s;
    }

    var iter = 0;
    var converged = false;
    while (!converged && iter < maxIter) {
        iter++;
        var maxChange = 0;
        for (var j = 0; j < p; j++) {
            var rho = 0;
            for (var i = 0; i < n; i++) rho += X[i][j] * residual[i];
            rho += colNorm2[j] * beta[j];

            var z = colNorm2[j];
            var newBeta = 0;
            if (z === 0) {
                newBeta = 0;
            } else {
                if (rho > lambda) newBeta = (rho - lambda) / z;
                else if (rho < -lambda) newBeta = (rho + lambda) / z;
                else newBeta = 0;
```

```
            }

            var delta = newBeta - beta[j];
            if (delta !== 0) {
                for (var i = 0; i < n; i++) residual[i] -= X[i][j] * delta;
                maxChange = Math.max(maxChange, Math.abs(delta));
                beta[j] = newBeta;
            }
        }
        if (maxChange < tol) converged = true;
    }
    return beta;
}

function lassoPath(X, y, lambdaSeq, tol = 1e-6, maxIter = 10000) {
    var L = lambdaSeq.length;
    var p = X[0].length;
    var coefs = Array(L);
    var betaPrev = Array(p).fill(0);

    for (var k = 0; k < L; k++) {
        var lam = lambdaSeq[k];
        betaPrev = lassoCoordinateDescentWarmStart(X, y, lam, betaPrev, tol, maxIter);
        coefs[k] = betaPrev.slice();
    }
    return coefs;
}

// ==================== Main MR Lasso Function ====================

function mr_lasso(object, options = {}) {
    var distribution = options.distribution || "normal";
    var alpha = (typeof options.alpha === "number") ? options.alpha : 0.05;
    var lambda = Array.isArray(options.lambda) ? options.lambda.slice() : [];

    var betaX = object.betaX.slice();
    var betaY = object.betaY.slice();
    var betaXse = object.betaXse.slice();
    var betaYse = object.betaYse.slice();
    var snps = Array.isArray(object.snps) ? object.snps.slice() : [];
    var exposure = object.exposure || null;
    var outcome = object.outcome || null;

    var nsnps = betaX.length;
    if (nsnps !== betaY.length || nsnps !== betaXse.length || nsnps !== betaYse.length) {
```

```
    throw new Error("Input vectors must have the same length.");
  }

  if (["normal", "t-dist"].indexOf(distribution) === -1) {
    throw new Error("Distribution must be one of: normal, t-dist.");
  }

  var Bx = betaX.map(v => Math.abs(v));
  var By = betaY.map((v, i) => Math.sign(betaX[i]) * v);
  var Bxse = betaXse.slice();
  var Byse = betaYse.slice();

  // S = diag(Byse^-2)
  var Sdiag = Byse.map(s => 1 / (s * s));
  // S^(1/2) is diagonal of Byse^-1
  var SsqrtDiag = Byse.map(s => 1 / s);

  // b = S^(1/2) %*% Bx
  var b = Array(nsnps);
  for (var i = 0; i < nsnps; i++) b[i] = SsqrtDiag[i] * Bx[i];

  // Pb = b %*% solve(t(b) %*% b, t(b))
  var denom = 0;
  for (var i = 0; i < nsnps; i++) denom += b[i] * b[i];
  var Pb = Array(nsnps);
  if (denom === 0) {
    for (var i = 0; i < nsnps; i++) {
      Pb[i] = Array(nsnps).fill(0);
    }
  } else {
    for (var i = 0; i < nsnps; i++) {
      Pb[i] = Array(nsnps);
      for (var j = 0; j < nsnps; j++) {
        Pb[i][j] = (b[i] * b[j]) / denom;
      }
    }
  }

  // Identity matrix
  var I = Array(nsnps);
  for (var i = 0; i < nsnps; i++) {
    I[i] = Array(nsnps).fill(0);
    I[i][i] = 1;
  }
```

```
  // xlas = (I - Pb) %*% S^(1/2)
  var xlas = Array(nsnps);
  for (var i = 0; i < nsnps; i++) {
    xlas[i] = Array(nsnps);
    for (var j = 0; j < nsnps; j++) {
      var mij = (i === j ? 1 : 0) - Pb[i][j];
      xlas[i][j] = mij * SsqrtDiag[j];
    }
  }

  // ylas = (I - Pb) %*% S^(1/2) %*% By
  var tvec = Array(nsnps);
  for (var i = 0; i < nsnps; i++) tvec[i] = SsqrtDiag[i] * By[i];
  var ylas = Array(nsnps).fill(0);
  for (var i = 0; i < nsnps; i++) {
    var s = 0;
    for (var j = 0; j < nsnps; j++) {
      var mij = (i === j ? 1 : 0) - Pb[i][j];
      s += mij * tvec[j];
    }
    ylas[i] = s;
  }

  // LASSO fitting
  var las_mod = null;
  if (lambda.length !== 0) {
    var lam = lambda[0];
    var coef = lassoCoordinateDescent(xlas, ylas, lam);
    las_mod = { fit: coef, lambda: lam };
  } else {
    // Automatic lambda sequence
    var p = xlas[0].length;
    var Xt_y = Array(p).fill(0);
    for (var j = 0; j < p; j++) {
      var s = 0;
      for (var i = 0; i < nsnps; i++) s += xlas[i][j] * ylas[i];
      Xt_y[j] = s;
    }
    var lambdaMax = Math.max.apply(null, Xt_y.map(v => Math.abs(v)));
    if (!isNumber(lambdaMax) || lambdaMax <= 0) lambdaMax = 1.0;

    var lamlen = 100;
    var lamseq = Array(lamlen);
    for (var k = 0; k < lamlen; k++) {
      lamseq[k] = lambdaMax * Math.pow(1e-4, k / (lamlen - 1));
```

```
}

var coefsPath = lassoPath(xlas, ylas, lamseq);

// RSE calculation for lambda selection
var rse_mat = [];
for (var idx = 0; idx < lamlen; idx++) {
   var beta_col = coefsPath[lamlen - idx - 1];
   var av = [];
   for (var ii = 0; ii < p; ii++) {
      if (Math.abs(beta_col[ii]) < 1e-12) av.push(ii);
   }
   var numValid = av.length;
   if (numValid < 2) {
      rse_mat.push([NaN, numValid]);
      continue;
   }

   // Weighted regression
   var Wvals = av.map(i => 1 / (Byse[i] * Byse[i]));
   var BX = av.map(i => Bx[i]);
   var BY = av.map(i => By[i]);
   var WXBX = 0;
   var WXB = 0;
   for (var kk = 0; kk < av.length; kk++) {
      WXBX += Wvals[kk] * BX[kk] * BX[kk];
      WXB += Wvals[kk] * BX[kk] * BY[kk];
   }
   var est = WXBX === 0 ? 0 : WXB / WXBX;

   var residuals = [];
   for (var kk = 0; kk < av.length; kk++) {
      residuals.push(BY[kk] - est * BX[kk]);
   }

   var ssr = 0;
   for (var kk = 0; kk < av.length; kk++) {
      ssr += residuals[kk] * residuals[kk];
   }

   var dfRes = Math.max(1, av.length - 1);
   var rseVal = Math.sqrt(ssr / dfRes);
   rse_mat.push([rseVal, numValid]);
}
```

```
    var rse_vals = rse_mat.map(x => x[0]);
    var num_valid = rse_mat.map(x => x[1]);

    var rse_inc = [];
    for (var j = 1; j < rse_vals.length; j++) {
      var a = rse_vals[j], b = rse_vals[j - 1];
      rse_inc.push((isNumber(a) ? a : 0) - (isNumber(b) ? b : 0));
    }

    var het = [];
    var chi1alpha = qchisq(0.95, 1);
    for (var j = 1; j < rse_vals.length; j++) {
      var rse_j = rse_vals[j];
      var inc_j = rse_inc[j - 1];
      var denom_j = rse_mat[j][1];
      var threshold = NaN;
      if (isNumber(denom_j) && denom_j !== 0) {
        threshold = chi1alpha / denom_j;
      }
      if (isNumber(rse_j) && rse_j > 1 && isNumber(inc_j) && inc_j > threshold) {
        het.push(j + 1);
      }
    }

    var lam_pos;
    if (het.length === 0) lam_pos = lamlen;
    else lam_pos = Math.min.apply(null, het);

    var num_valid_arr = num_valid;
    var min_lam_pos = -1;
    for (var t = 0; t < num_valid_arr.length; t++) {
      if (num_valid_arr[t] > 1) {
        min_lam_pos = t + 1;
        break;
      }
    }
    if (min_lam_pos === -1) min_lam_pos = lamlen;
    if (lam_pos < min_lam_pos) lam_pos = min_lam_pos;

    var chosenIndex = lamlen - lam_pos;
    if (chosenIndex < 0) chosenIndex = 0;
    if (chosenIndex >= lamlen) chosenIndex = lamlen - 1;
    var chosenCoef = coefsPath[chosenIndex];
    las_mod = { fit: chosenCoef, lambda: lamseq[chosenIndex] };
  }
```

```
  var a = las_mod.fit;
  if (!Array.isArray(a) || a.length !== nsnps) {
    var newa = Array(nsnps).fill(0);
    for (var i = 0; i < Math.min(nsnps, a.length); i++) newa[i] = a[i];
    a = newa;
  }

  // e = By - a
  var e = Array(nsnps);
  for (var i = 0; i < nsnps; i++) e[i] = By[i] - a[i];

  // est = solve(t(Bx) %*% S %*% Bx, t(Bx) %*% S %*% e)
  var BXSBX = 0;
  var BXS_e = 0;
  for (var i = 0; i < nsnps; i++) {
    var w = Sdiag[i];
    BXSBX += Bx[i] * w * Bx[i];
    BXS_e += Bx[i] * w * e[i];
  }
  var est = BXSBX === 0 ? NaN : BXS_e / BXSBX;

  // v = which(a == 0)
  var v = [];
  for (var i = 0; i < a.length; i++) {
    if (Math.abs(a[i]) < 1e-12) v.push(i);
  }

  var post_est = NaN, post_se = NaN;
  if (v.length > 1) {
    var WXBX = 0, WXB = 0;
    var Wvals = [];
    var BXs = [], BYs = [];
    for (var k = 0; k < v.length; k++) {
      var idx = v[k];
      var w = 1 / (Byse[idx] * Byse[idx]);
      Wvals.push(w);
      BXs.push(Bx[idx]);
      BYs.push(By[idx]);
      WXBX += w * Bx[idx] * Bx[idx];
      WXB += w * Bx[idx] * By[idx];
    }
    post_est = WXBX === 0 ? NaN : WXB / WXBX;

    var residuals = [];
```

```
        for (var k = 0; k < v.length; k++) {
            residuals.push(BYs[k] - post_est * BXs[k]);
        }

        var ssr = 0;
        for (var k = 0; k < residuals.length; k++) {
            ssr += residuals[k] * residuals[k];
        }

        var dfRes = Math.max(1, residuals.length - 1);
        var sigma_post = Math.sqrt(ssr / dfRes);
        var se_raw = WXBX === 0 ? NaN : Math.sqrt(1 / WXBX);
        post_se = se_raw / Math.min(sigma_post, 1);
    } else {
        post_est = NaN;
        post_se = NaN;
        console.warn("Fewer than two valid instruments. Post-lasso method cannot be performed.");
    }

    // Confidence intervals and p-value
    var ciLower = NaN, ciUpper = NaN, pvalue = NaN;
    if (distribution === "normal") {
        ciLower = ci_normal("l", post_est, post_se, alpha);
        ciUpper = ci_normal("u", post_est, post_se, alpha);
        if (isNumber(post_est) && isNumber(post_se)) {
            pvalue = 2 * (1 - pnorm(Math.abs(post_est / post_se)));
        }
    } else {
        var df = Math.max(1, v.length - 1);
        ciLower = ci_t("l", post_est, post_se, df, alpha);
        ciUpper = ci_t("u", post_est, post_se, df, alpha);
        if (isNumber(post_est) && isNumber(post_se)) {
            pvalue = 2 * (1 - pt(Math.abs(post_est / post_se), df));
        }
    }

    var validSNPs = v.map(idx => (snps[idx] !== undefined ? String(snps[idx]) : String(idx)));

    var result = {
        Exposure: exposure,
        Outcome: outcome,
        Estimate: post_est,
        StdError: post_se,
        CILower: ciLower,
        CIUpper: ciUpper,
```

```
    Alpha: alpha,
    Pvalue: pvalue,
    SNPs: nsnps,
    RegEstimate: est,
    RegIntercept: a,
    Valid: v.length,
    ValidSNPs: validSNPs,
    Lambda: las_mod.lambda
  };

  return result;
}
```

(6) The Weighted Median Method (WM)

The JavaScript codes for The Weighted Median Method (WM) are as follows:

```
/*
Method name and aim
Name: The Weighted Median Method (WM) for Mendelian Randomization
Aim: Estimate a causal effect using a weighted median approach with three weighting schemes (simple, weighted, penalized) and provide statistical inference (CI and P-values) under either a normal or t-distribution. Supports seeded bootstrap for standard error estimation.

Main components and functions
Utility: seeded RNG (mulberry32) and normal variate generator (rnorm_one) to enable reproducible bootstrap resampling.
CI helpers: ci_normal and ci_t to compute confidence intervals under normal or t-distribution assumptions.
Core estimator: weightedMedian(theta, weights) computes the weighted median of the per-variant causal estimates theta with corresponding weights.
Bootstrap SE: weightedMedianBootSE(Bx, By, Bxse, Byse, weights, iter, seed) generates bootstrap samples of the ratio estimates and returns the bootstrap standard error of the median.
Main driver: mr_median(object, args)
Builds per-variant causal estimates theta = By / Bx.
Constructs three weighting schemes:
simple: equal weights
weighted: weights proportional to (Bx / Byse)^2
penalized: down-weights outliers using a penalty based on a chi-squared tail probability
Chooses theWM estimate (thetaWM) and bootstrap SE (seBoot) accordingly.
Computes confidence intervals and p-values using either normal or t-distribution.
Returns a result object with the estimated effect and inference metrics.

Required inputs and expected outputs

Required inputs:
object: {
betaX: array of SNP-exposure effects,
```

betaY: array of SNP-outcome effects,
betaXse: array of SEs for betaX,
betaYse: array of SEs for betaY,
exposure: string label for exposure,
outcome: string label for outcome
}
args (optional): {
weighting: "simple" | "weighted" | "penalized",
distribution: "normal" | "t-dist",
alpha: significance level for CIs (default if unspecified),
iterations: number of bootstrap iterations,
seed: integer seed for reproducibility
}

Expected outputs:
An object with fields:
Type: the chosen weighting ("simple", "weighted", or "penalized")
Exposure: object.exposure
Outcome: object.outcome
Estimate: the weighted-median causal estimate (thetaWM)
StdError: bootstrap standard error (seBoot)
CILower: lower bound of the confidence interval
CIUpper: upper bound of the confidence interval
Alpha: significance level used
Pvalue: p-value for the causal estimate
SNPs: number of variants (n)

Notes
If the number of variants is less than 3, the function logs a message and returns null.
The penalized weighting approach uses outlier penalties derived from a chi-square tail, scaled by a factor to adjust down-weighting.
*/

```
// ---------- Utility: seeded RNG (mulberry32) ----------
function mulberry32(seed) {
  // seed should be a 32-bit integer
  let t = seed >>> 0;
  return function() {
    t += 0x6D2B79F5;
    let r = Math.imul(t ^ (t >>> 15), 1 | t);
    r ^= r + Math.imul(r ^ (r >>> 7), 61 | r);
    return ((r ^ (r >>> 14)) >>> 0) / 4294967296;
  };
}
```

```
// ---------- Helper: generate a single normal random variate with optional seeded RNG ----------
function rnorm_one(mean = 0, sd = 1, rng) {
  // Use inverse CDF with a uniform random number
  const u = (typeof rng === "function") ? rng() : Math.random();
  return jStat.normal.inv(u, mean, sd);
}

// ---------- CI helpers (same as in other code) ----------
function ci_normal(side, estimate, se, alpha) {
  const z = Math.abs(jStat.normal.inv(alpha / 2, 0, 1));
  return side === "l" ? estimate - z * se : estimate + z * se;
}
function ci_t(side, estimate, se, df, alpha) {
  const tcrit = Math.abs(jStat.studentt.inv(alpha / 2, df));
  return side === "l" ? estimate - tcrit * se : estimate + tcrit * se;
}

// ---------- Weighted median ----------
function weightedMedian(theta, weights) {
  if (!Array.isArray(theta) || !Array.isArray(weights)) {
    throw new Error("theta and weights must be arrays");
  }
  const n = theta.length;
  if (weights.length !== n) throw new Error("theta and weights must have same length");

  // Create index array and sort by theta
  const idx = Array.from({ length: n }, (_, i) => i);
  idx.sort((a, b) => {
    if (theta[a] < theta[b]) return -1;
    if (theta[a] > theta[b]) return 1;
    return 0;
  });

  // Sorted theta and weights
  const thetaSorted = idx.map(i => theta[i]);
  const wSorted = idx.map(i => weights[i]);

  const wSum = wSorted.reduce((s, v) => s + v, 0);
  if (wSum === 0) {
    // if all weights are zero, fallback to simple median of theta
    const copy = theta.slice().sort((a, b) => a - b);
    const mid = Math.floor(copy.length / 2);
    return (copy.length % 2 === 1) ? copy[mid] : (copy[mid - 1] + copy[mid]) / 2;
  }
```

```
    const cumsum = [];
    let running = 0;
    for (let i = 0; i < n; i++) {
        running += wSorted[i];
        cumsum.push((running - 0.5 * wSorted[i]) / wSum);
    }

    // find k = length(which(cumsum < 0.5)) i.e. last index where cumsum < 0.5
    let k = -1;
    for (let i = 0; i < n; i++) {
        if (cumsum[i] < 0.5) k = i;
        else break;
    }

    // Handle edge cases: if k = -1 (all cumsum >= 0.5) -> median is thetaSorted[0]
    // if k = n-1 (all cumsum < 0.5) -> median is thetaSorted[n-1]
    if (k === -1) return thetaSorted[0];
    if (k === n - 1) return thetaSorted[n - 1];

    // ratio interpolation between thetaSorted[k] and thetaSorted[k+1]
    const denom = cumsum[k + 1] - cumsum[k];
    if (denom === 0) return thetaSorted[k]; // avoid division by zero
    const ratio = (0.5 - cumsum[k]) / denom;
    const weightedEstimate = thetaSorted[k] + (thetaSorted[k + 1] - thetaSorted[k]) * ratio;
    return weightedEstimate;
}

// ---------- Weighted median bootstrap SE (weighted.median.boot.se) ----------
function weightedMedianBootSE(Bx, By, Bxse, Byse, weights, iter = 100, seed = null) {
    // Input validation
    const n = By.length;
    if (Bx.length !== n || Bxse.length !== n || Byse.length !== n || weights.length !== n) {
        throw new Error("All input vectors must have same length");
    }
    // Setup RNG: if seed is null or undefined => use Math.random, else create seeded rng
    let rng = null;
    if (seed !== null && seed !== undefined && !Number.isNaN(seed)) {
        // Use a 32-bit integer from seed (coerce)
        const s = Math.floor(seed) >>> 0;
        rng = mulberry32(s);
    }

    const med = new Array(iter);
    for (let it = 0; it < iter; it++) {
        const BxBoot = new Array(n);
```

```
      const ByBoot = new Array(n);
      for (let i = 0; i < n; i++) {
        BxBoot[i] = rnorm_one(Bx[i], Bxse[i], rng);
        ByBoot[i] = rnorm_one(By[i], Byse[i], rng);
      }
      const thetaBoot = new Array(n);
      for (let i = 0; i < n; i++) thetaBoot[i] = ByBoot[i] / BxBoot[i];
      med[it] = weightedMedian(thetaBoot, weights);
    }

    // compute standard deviation of med
    const meanMed = med.reduce((s, v) => s + v, 0) / med.length;
    const sd = Math.sqrt(med.reduce((s, v) => s + (v - meanMed) * (v - meanMed), 0) / (med.length - 1 || 1));
    return sd;
}

// ---------- mr_median implementation ----------
function mr_median(object, args) {
    // object must be MRInput-like: fields betaX, betaY, betaXse, betaYse, exposure, outcome
    args = args || {};
    const weighting = args.weighting || "weighted"; // "simple", "weighted", "penalized"
    const distribution = args.distribution || "normal"; // "normal" or "t-dist"
    const alpha = (args.alpha === undefined ? 0.001 : args.alpha);
    const iterations = (args.iterations === undefined ? 10000 : args.iterations);
    const seed = (args.seed === undefined ? 314159265 : args.seed);

    const Bx = object.betaX.slice();
    const By = object.betaY.slice();
    const Bxse = object.betaXse.slice();
    const Byse = object.betaYse.slice();

    const n = By.length;
    if (Bx.length !== n || Bxse.length !== n || Byse.length !== n) {
      throw new Error("Input vectors betaX, betaY, betaXse, betaYse must have the same length");
    }
    if (n < 3) {
      console.log("Method requires data on >2 variants.");
      return null;
    }

    // Theta per SNP
    const Theta = new Array(n);
    for (let i = 0; i < n; i++) Theta[i] = By[i] / Bx[i];

    const Simple = new Array(n).fill(1 / n);
```

```
    const Weighted = new Array(n);
    for (let i = 0; i < n; i++) {
      // avoid division by zero
      const denom = Byse[i] === 0 ? Number.EPSILON : Byse[i];
      Weighted[i] = Math.pow(Bx[i] / denom, 2);
    }

    if (!["simple", "weighted", "penalized"].includes(weighting) || !["normal", "t-dist"].includes(distribution)) {
      throw new Error("Weighting must be one of: simple, weighted, penalized. Distribution must be one of: normal, t-dist.");
    }

    let thetaWM, seBoot, ciLower, ciUpper, pval;
    if (weighting === "simple") {
      thetaWM = weightedMedian(Theta, Simple);
      seBoot = weightedMedianBootSE(Bx, By, Bxse, Byse, Simple, iterations, seed);
    } else if (weighting === "weighted") {
      thetaWM = weightedMedian(Theta, Weighted);
      seBoot = weightedMedianBootSE(Bx, By, Bxse, Byse, Weighted, iterations, seed);
    } else { // penalized
      // penalty <- pchisq(Weighted*(Theta - weighted.median(Theta, Weighted))^2, df = 1, lower.tail = FALSE)
      const wmWeighted = weightedMedian(Theta, Weighted);
      const penalty = new Array(n);
      for (let i = 0; i < n; i++) {
        const chisq = Weighted[i] * Math.pow(Theta[i] - wmWeighted, 2);
        // pchisq(..., lower.tail = FALSE) = 1 - CDF(chisq)
        const c = 1 - jStat.chisquare.cdf(chisq, 1);
        penalty[i] = c;
      }
      const penWeights = new Array(n);
      for (let i = 0; i < n; i++) penWeights[i] = Weighted[i] * Math.min(1, penalty[i] * 20);

thetaWM = weightedMedian(Theta, penWeights);
seBoot = weightedMedianBootSE(Bx, By, Bxse, Byse, penWeights, iterations, seed);

    }

    // Confidence intervals and p-value according to distribution
    if (distribution === "normal") {
      ciLower = ci_normal("l", thetaWM, seBoot, alpha);
      ciUpper = ci_normal("u", thetaWM, seBoot, alpha);
      // pval: 2 * pnorm(-abs(thetaWM/seBoot))
      const z = Math.abs(thetaWM / seBoot);
      pval = 2 * (1 - jStat.normal.cdf(z, 0, 1));
    } else {
      const df = n - 1;
```

```
    ciLower = ci_t("l", thetaWM, seBoot, df, alpha);
    ciUpper = ci_t("u", thetaWM, seBoot, df, alpha);
    const tstat = Math.abs(thetaWM / seBoot);
    pval = 2 * (1 - jStat.studentt.cdf(tstat, df));
  }

  // Construct return object
  const result = {
    Type: weighting,
    Exposure: object.exposure || null,
    Outcome: object.outcome || null,
    Estimate: thetaWM,
    StdError: seBoot,
    CILower: ciLower,
    CIUpper: ciUpper,
    Alpha: alpha,
    Pvalue: pval,
    SNPs: n
  };

  return result;
}
```

(7) The Contamination Mixture Model (Conmix)

The JavaScript codes for The Contamination Mixture Model (Conmix) are as follows:

```
/*
Method name and aim
Name: Contamination Mixture Model (Conmix) for Mendelian Randomization
Aim: Estimate a causal effect while downweighting outlier instruments using a contamination mixture framework. Identify which SNPs are valid instruments at the optimal estimate and provide a confidence interval (which may be disjoint) around the causal effect.

Main classes and functions
MRInput class
Purpose: Container for input data used in the analysis.
Keys: betaX, betaY, betaXse, betaYse, optional correlation, exposure, outcome, snps.
Behavior: Validates that all input arrays have the same length; assigns SNP names if not provided.
MRConMix class
Purpose: Data structure for returning results.
Keys: Exposure, Outcome, Psi, Estimate, CIRange, CILower, CIUpper, CIMin, CIMax, CIStep, Valid, ValidSNPs, Pvalue, SNPs, Alpha.
Helper functions
weightedMean(values, weights): computes weighted mean.
seq(from, to, by): generates a numeric sequence.
```

getIndices(array, conditionFn): returns indices where a condition holds.
diff(arr): differences between adjacent elements.
calculateSd(arr): sample standard deviation (uses jStat).
mr_conmix function
Purpose: Core analysis routine. Filters to valid SNPs, computes ratio statistics and standard errors, constructs a grid of possible causal effects (theta), evaluates a likelihood for each theta under inclusive/exclusive models, selects the maximum-likelihood theta, derives the set of valid SNPs at the optimum, computes phi, builds the null model likelihood, and outputs a MRConMix object.
Inputs: an object with betaX, betaY, betaXse, betaYse, snps, exposure, outcome; options including psi, CIMin, CIMax, CIStep, alpha.
Outputs: an MRConMix instance with the estimated causal effect, confidence-interval range (possibly disjoint), information about which SNPs are considered valid, and p-value.

Required inputs and expected outputs

Required inputs
An object containing:
betaX: array of SNP-exposure effects
betaY: array of SNP-outcome effects
betaXse: standard errors for betaX
betaYse: standard errors for betaY
snps: array of SNP names
exposure: string name of exposure
outcome: string name of outcome
Optional: correlation (matrix), though not mandatory
options: psi (contamination parameter), CIMin, CIMax, CIStep (grid for causal effect), alpha (significance level)

Expected outputs
An MRConMix object containing:
Exposure, Outcome
Psi (calibration parameter used)
Estimate (causal effect at the chosen theta)
CIRange (array of CI values, may be disjoint)
CILower, CIUpper (arrays corresponding to CI ranges)
CIMin, CIMax, CIStep (grid parameters used)
Valid (indices of SNPs deemed valid at the optimum)
ValidSNPs (names of valid SNPs)
Pvalue (statistical significance)
SNPs (original total number of SNPs)
Alpha (significance level)

Notes
The function filters out SNPs with invalid data (non-finite SEs, zero betaX, etc.) before analysis.
The CI can be a disjoint range, reflecting the contamination-mixture logic.
The implementation relies on numerical routines (e.g., jStat) for statistical calculations.

```
*/

// --- MRInput Class (re-used from previous problem) ---
class MRInput {
    constructor({betaX, betaY, betaXse, betaYse, correlation = null, exposure = "Exposure", outcome = "Outcome", snps = null}) {
        this.betaX = betaX || [];
        this.betaY = betaY || [];
        this.betaXse = betaXse || [];
        this.betaYse = betaYse || [];
        this.correlation = correlation; // Can be a math.js matrix or 2D array
        this.exposure = exposure;
        this.outcome = outcome;
        // snps: Array of SNP names.
        this.snps = snps || Array.from({length: betaX.length}, (_, i) => `SNP_${i + 1}`);

        const lengths = new Set([this.betaX.length, this.betaY.length, this.betaXse.length, this.betaYse.length, this.snps.length]);
        if (lengths.size !== 1) {
            throw new Error("MRInput: All input beta, SE, and SNP arrays must have the same length.");
        }
    }
}

// --- MRConMix Class ---
class MRConMix {
    constructor({
        Exposure, Outcome, Psi, Estimate, CIRange, CILower, CIUpper, CIMin, CIMax, CIStep, Valid, ValidSNPs, Pvalue, SNPs, Alpha
    }) {
        this.Exposure = Exposure;
        this.Outcome = Outcome;
        this.Psi = Psi;
        this.Estimate = Estimate;
        this.CIRange = CIRange; // Array of estimates forming the CI range (can be disjoint)
        this.CILower = CILower; // Array of lower bounds for CI (can be multiple if disjoint)
        this.CIUpper = CIUpper; // Array of upper bounds for CI (can be multiple if disjoint)
        this.CIMin = CIMin;
        this.CIMax = CIMax;
        this.CIStep = CIStep;
        this.Valid = Valid; // Indices (0-based) of valid SNPs (those selected for likelihood_inc)
        this.ValidSNPs = ValidSNPs; // Names of valid SNPs
        this.Pvalue = Pvalue;
        this.SNPs = SNPs; // Original total number of SNPs
        this.Alpha = Alpha;
```

```
    }
}

// --- Helper Functions ---

/**
 * Calculates a weighted mean.
 * @param {number[]} values - Array of values.
 * @param {number[]} weights - Array of weights.
 * @returns {number} The weighted mean.
 */
function weightedMean(values, weights) {
    if (values.length === 0) return NaN;
    let sum = 0;
    let weightSum = 0;
    for (let i = 0; i < values.length; i++) {
        if (isFinite(values[i]) && isFinite(weights[i])) {
            sum += values[i] * weights[i];
            weightSum += weights[i];
        }
    }
    return weightSum === 0 ? NaN : sum / weightSum;
}

/**
 * Generates a sequence of numbers.
 * @param {number} from - Starting value.
 * @param {number} to - Ending value.
 * @param {number} by - Step size.
 * @returns {number[]} The generated sequence.
 */
function seq(from, to, by) {
    const result = [];
    // Adjust for floating point precision issues, add a small epsilon
    const epsilon = 1e-9;
    for (let i = from; i <= to + epsilon; i += by) {
        result.push(i);
    }
    return result;
}

/**
 * Finds indices of elements in an array that satisfy a condition.
 * @param {Array} array - The array to search.
 * @param {Function} conditionFn - A function that returns true for elements to include.
```

```
 * @returns {number[]} An array of 0-based indices.
 */
function getIndices(array, conditionFn) {
    const indices = [];
    for (let i = 0; i < array.length; i++) {
        if (conditionFn(array[i], i, array)) {
            indices.push(i);
        }
    }
    return indices;
}

/**
 * Calculates the differences between adjacent elements in an array.
 * @param {number[]} arr - The input array.
 * @returns {number[]} An array of differences.
 */
function diff(arr) {
    if (arr.length < 2) return [];
    const result = [];
    for (let i = 1; i < arr.length; i++) {
        result.push(arr[i] - arr[i - 1]);
    }
    return result;
}

/**
 * Calculates the sample standard deviation of an array).
 * @param {number[]} arr - The array of numbers.
 * @returns {number} The sample standard deviation.
 */
function calculateSd(arr) {
    return jStat.stdev(arr, false); // false for sample standard deviation
}

// --- mr_conmix Function ---
/**
 * Performs Mendelian Randomization analysis using the Contamination Mixture model.
 *
 * @param {MRInput} object - MRInput object containing genetic data.
 * @param {object} [options={}] - Configuration options for the analysis.
 * @param {number} [options.psi=0] - Parameter for the contamination mixture model. If 0 or negative, it's set to 1.5 * sd(ratio).
 * @param {number} [options.CIMin=NaN] - Minimum value for the search grid of the causal estimate. If NaN, it's derived from data.
 * @param {number} [options.CIMax=NaN] - Maximum value for the search grid of the causal estimate. If NaN, it's derived
```

```
from data.
 * @param {number} [options.CIStep=0.01] - Step size for the search grid.
 * @param {number} [options.alpha=0.001] - Significance level for confidence intervals.
 * @returns {MRConMix} An MRConMix object containing the analysis results.
 */
function mr_conmix(object, options = {}) {
    let {
        psi = 0,
        CIMin = NaN,
        CIMax = NaN,
        CIStep = 0.01,
        alpha = 0.001
    } = options;

    const original_nsnps = object.betaX.length;

    // --- Data Filtering and Preparation ---
    const validSnpsData = [];
    for (let i = 0; i < original_nsnps; i++) {
        const betaX = object.betaX[i];
        const betaY = object.betaY[i];
        const betaXse = object.betaXse[i];
        const betaYse = object.betaYse[i];
        const snp_name = object.snps[i];

        // Filter out SNPs with non-finite or non-positive SEs, or zero betaX (which would cause division by zero for ratio)
        if (isFinite(betaX) && isFinite(betaY) && isFinite(betaXse) && isFinite(betaYse) &&
            betaXse > 0 && betaYse > 0 && betaX !== 0) {
            validSnpsData.push({ betaX, betaY, betaXse, betaYse, snp_name });
        } else {
            console.warn(`SNP '${snp_name}' (index ${i}) excluded from MR-ConMix analysis due to invalid data (non-finite/non-positive SEs or zero betaX): ` +
                         `betaX=${betaX}, betaY=${betaY}, betaXse=${betaXse}, betaYse=${betaYse}.`);
        }
    }

    if (validSnpsData.length === 0) {
        throw new Error("No valid SNPs remaining after initial filtering for MR-ConMix analysis.");
    }

    const Bx = validSnpsData.map(d => d.betaX);
    const By = validSnpsData.map(d => d.betaY);
    const Bxse = validSnpsData.map(d => d.betaXse);
    const Byse = validSnpsData.map(d => d.betaYse);
    const snp_names_filtered = validSnpsData.map(d => d.snp_name);
```

```
    const nsnps_filtered = Bx.length;

    const ratio = By.map((y, i) => y / Bx[i]);
    const ratio_se = Byse.map((se, i) => Math.abs(se / Bx[i]));

    // Initialize CIMin and CIMax if not provided
    if (isNaN(CIMin) || isNaN(CIMax)) {
        const lower_bound_candidates = By.map((y, i) => (y - 2 * Byse[i]) / Bx[i]);
        const upper_bound_candidates = By.map((y, i) => (y + 2 * Byse[i]) / Bx[i]);

        const all_candidates = [...lower_bound_candidates, ...upper_bound_candidates].filter(isFinite);

        if (all_candidates.length === 0) {
            throw new Error("Could not derive CIMin/CIMax from data; all calculated candidate values are non-finite. Please provide explicit CIMin and CIMax.");
        }
        if (isNaN(CIMin)) { CIMin = Math.min(...all_candidates); }
        if (isNaN(CIMax)) { CIMax = Math.max(...all_candidates); }
    }

    // Adjust CIMin/CIMax if CIMin >= CIMax
    if (CIMin >= CIMax) {
        console.warn(`CIMin (${CIMin}) is greater than or equal to CIMax (${CIMax}). Adjusting CIMax to CIMin + CIStep * 2.`);
        CIMax = CIMin + CIStep * 2;
        if (CIMax <= CIMin) CIMax = CIMin + 0.1; // Ensure it's strictly greater to form a sequence
    }

    // Initialize psi
    if (psi <= 0) {
        if (nsnps_filtered < 2) { // Need at least 2 points to calculate standard deviation
            throw new Error("Cannot calculate standard deviation for psi with fewer than 2 valid SNPs. Please provide a positive psi value.");
        }
        const sd_ratio = calculateSd(ratio);
        if (!isFinite(sd_ratio) || sd_ratio === 0) {
            console.warn("Standard deviation of ratio is non-finite or zero. Defaulting psi to 0.1 to allow calculation.");
            psi = 0.1; // Fallback to a small positive value
        } else {
            psi = 1.5 * sd_ratio;
        }
    }

    // Create theta sequence (search grid for causal estimate)
    const theta = seq(CIMin, CIMax, CIStep);
```

```
    const iters = theta.length;

    if (iters === 0) {
        throw new Error("Theta sequence is empty. Check CIMin, CIMax, and CIStep values. Ensure CIMax is sufficiently greater than CIMin.");
    }

    const lik = new Array(iters); // Stores total likelihood for each theta value
    const all_valid_arrays = new Array(iters); // Stores the 'valid' (0/1) array for each theta value

    for (let j1 = 0; j1 < iters; j1++) {
        const current_theta = theta[j1];

        let current_lik_sum = 0;
        const current_valid = new Array(nsnps_filtered);

        for (let i = 0; i < nsnps_filtered; i++) {
            const r = ratio[i];
            const r_se = ratio_se[i];

            // Likelihood for inclusive model (SNP is valid instrument)
            // FIX: Add parentheses around the squared term
            const lik_inc_term1 = -((current_theta - r)**2) / (2 * r_se**2);
            const lik_inc_term2 = -Math.log(Math.sqrt(2 * Math.PI * r_se**2));
            const lik_inc = lik_inc_term1 + lik_inc_term2;

            // Likelihood for exclusive model (SNP is an outlier/invalid instrument)
            const lik_exc_denominator = (psi**2 + r_se**2);
            if (lik_exc_denominator <= 0 || !isFinite(lik_exc_denominator)) {
                current_lik_sum = NaN; // Propagate NaN if denominator is invalid
                break; // Exits the inner loop for this j1
            }
            // FIX: Add parentheses around the squared term
            const lik_exc_term1 = -(r**2) / (2 * lik_exc_denominator);
            const lik_exc_term2 = -Math.log(Math.sqrt(2 * Math.PI * lik_exc_denominator));
            const lik_exc = lik_exc_term1 + lik_exc_term2;

            if (isFinite(lik_inc) && isFinite(lik_exc)) {
                if (lik_inc > lik_exc) {
                    current_valid[i] = 1;
                    current_lik_sum += lik_inc;
                } else {
                    current_valid[i] = 0;
                    current_lik_sum += lik_exc;
                }
```

```
            } else {
                current_lik_sum = NaN; // If any likelihood component is not finite, sum is NaN
                break;
            }
        }
        lik[j1] = current_lik_sum;
        all_valid_arrays[j1] = current_valid;
    }

    // Find the maximum likelihood and corresponding index
    const finite_lik = lik.filter(isFinite);
    const max_lik_val = finite_lik.length > 0 ? Math.max(...finite_lik) : NaN;

    if (!isFinite(max_lik_val)) {
        throw new Error("Maximum likelihood value is non-finite. This can indicate issues with input data or model parameters (psi, CI range) leading to non-finite likelihoods across the entire grid.");
    }
    const max_lik_idx = lik.indexOf(max_lik_val); // Get the index from the original 'lik' array

    // Determine 'valid_best' (which SNPs were considered valid at the optimal estimate)
    const valid_best = all_valid_arrays[max_lik_idx];
    const sum_valid_best = valid_best.reduce((sum, val) => sum + val, 0);

    let phi;
    if (sum_valid_best < 1.5) { // If 0 or 1 SNP selected as valid, phi defaults to 1
        phi = 1;
    } else {
        const valid_ratio_subset = ratio.filter((_, i) => valid_best[i] === 1);
        const valid_ratio_se_sq_inv_subset = ratio_se.filter((_, i) => valid_best[i] === 1).map(se => 1 / (se * se));

        const weighted_m = weightedMean(valid_ratio_subset, valid_ratio_se_sq_inv_subset);

        let sum_sq_diff_weighted = 0;
        for (let i = 0; i < valid_ratio_subset.length; i++) {
            sum_sq_diff_weighted += (valid_ratio_subset[i] - weighted_m)**2 * valid_ratio_se_sq_inv_subset[i];
        }

        const denominator_phi = sum_valid_best - 1;
        if (denominator_phi <= 0) { // Avoid division by zero or negative
            phi = 1; // Fallback, though sum_valid_best < 1.5 should already handle this for common cases
            console.warn("Denominator for phi calculation is non-positive for sum_valid_best=" + sum_valid_best + ", defaulting phi to 1.");
        } else {
            phi = Math.max(Math.sqrt(sum_sq_diff_weighted / denominator_phi), 1);
        }
```

```
}

// Null model likelihood (effectively with theta = 0)
let loglik0_sum = 0;
const valid0_dummy_array = new Array(nsnps_filtered); // Not strictly needed, but useful for structure
for (let i = 0; i < nsnps_filtered; i++) {
    const r = ratio[i];
    const r_se = ratio_se[i];

    // FIX: Add parentheses around the squared term
    const lik_inc0_term1 = -(r**2) / (2 * r_se**2);
    const lik_inc0_term2 = -Math.log(Math.sqrt(2 * Math.PI * r_se**2));
    const lik_inc0 = lik_inc0_term1 + lik_inc0_term2;

    const lik_exc0_denominator = (psi**2 + r_se**2);
    if (lik_exc0_denominator <= 0 || !isFinite(lik_exc0_denominator)) {
        loglik0_sum = NaN;
        break;
    }
    // FIX: Add parentheses around the squared term
    const lik_exc0_term1 = -(r**2) / (2 * lik_exc0_denominator);
    const lik_exc0_term2 = -Math.log(Math.sqrt(2 * Math.PI * lik_exc0_denominator));
    const lik_exc0 = lik_exc0_term1 + lik_exc0_term2;

    if (isFinite(lik_inc0) && isFinite(lik_exc0)) {
        if (lik_inc0 > lik_exc0) {
            valid0_dummy_array[i] = 1; // dummy, not returned
            loglik0_sum += lik_inc0;
        } else {
            valid0_dummy_array[i] = 0; // dummy, not returned
            loglik0_sum += lik_exc0;
        }
    } else {
        loglik0_sum = NaN;
        break;
    }
}
const loglik0 = loglik0_sum;

const betaConMix = theta[max_lik_idx];

// Confidence interval calculation
const qchisq_val = jStat.chisquare.inv(1 - alpha, 1);
const threshold = 2 * max_lik_val - qchisq_val * phi**2;
```

```
// Indices in `theta` where the condition is met (lik > threshold)
const whichin_indices = getIndices(lik, val => isFinite(val) && (2 * val > threshold));
let CIRange_values = whichin_indices.map(idx => theta[idx]);
CIRange_values.sort((a,b) => a - b); // Ensure sorted for diff calculation

const CILower = [];
const CIUpper = [];

if (CIRange_values.length > 0) {
    CILower.push(CIRange_values[0]);
    for (let i = 0; i < CIRange_values.length - 1; i++) {
        // Check for gaps (discontinuities) in the CI range
        if (CIRange_values[i+1] - CIRange_values[i] > 1.01 * CIStep) {
            CIUpper.push(CIRange_values[i]);
            CILower.push(CIRange_values[i+1]);
        }
    }
    CIUpper.push(CIRange_values[CIRange_values.length - 1]);
} else {
    // If no values met the CI criteria, set to NaN
    CILower.push(NaN);
    CIUpper.push(NaN);
}

let pvalue = NaN;
const chi_sq_statistic = 2 * (max_lik_val - loglik0) * phi**2;
if (isFinite(chi_sq_statistic) && chi_sq_statistic >= 0) {
    pvalue = 1 - jStat.chisquare.cdf(chi_sq_statistic, 1);
} else {
    console.warn("Chi-squared statistic for p-value is non-finite or negative, setting Pvalue to NaN.");
}

// Get names and indices of valid SNPs at the optimal estimate
const valid_indices_at_best = getIndices(valid_best, val => val === 1);
const valid_snps_names = valid_indices_at_best.map(idx => snp_names_filtered[idx]);

// Return the result object
return new MRConMix({
    Exposure: object.exposure,
    Outcome: object.outcome,
    Psi: psi,
    Estimate: betaConMix,
    CIRange: CIRange_values,
    CILower: CILower,
    CIUpper: CIUpper,
```

```
            CIMin: CIMin,
            CIMax: CIMax,
            CIStep: CIStep,
            Valid: valid_indices_at_best, // Indices of SNPs where lik.inc > lik.exc at the optimal theta
            ValidSNPs: valid_snps_names,
            Pvalue: pvalue,
            SNPs: original_nsnps, // Total original SNPs provided to the function
            Alpha: alpha
        });
}
```

(8) The Mode-Based Estimation Method (MBE)

The JavaScript codes for The Mode-Based Estimation Method (MBE) are as follows:

```
/*
Method name and aim
Name: Mode-Based Estimation Method (MBE), used in Mendelian randomization.
Objective: To robustly estimate a causal effect using multiple genetic instruments by modeling the distribution of ratio estimates and selecting its mode (via kernel density estimation), with bootstrap/MAD-based standard errors and confidence intervals.

Main classes and functions
MRInput class
Purpose: Wraps raw data into a single object, including betaX, betaY, betaXse, betaYse, optional correlation, exposure/outcome names, and SNPs.
MRMBE class (result container)
Purpose: Stores analysis configuration and results (exposure, outcome, weighting, std. error method, phi, point estimate, standard error, CIs, p-value, SNP count, etc.).
Core functions (high level)
calculateSd, calculateMedian, calculateMad: helper statistics for dispersion.
weightedKDE: kernel density estimation with weights to locate the distribution mode.
mbe_est: core mode-estimation algorithm (computes the mode of the BetaIV distribution).
mbe_boot: bootstrap routine to generate a distribution of mode estimates for SE.
ci_normal, ci_t: compute confidence intervals under normal or t-distributions.
mr_mbe: main analysis driver that orchestrates input validation, computes BetaIV = By/Bx, selects weighting, runs mbe_est and mbe_boot, and returns an MRMBE result object.

Required inputs and expected outputs

Required inputs (for mr_mbe):
betaX: effect of SNP on exposure (array)
betaY: effect of SNP on outcome (array)
betaXse: standard error of betaX (array)
betaYse: standard error of betaY (array)
snps: optional SNP identifiers (array)
exposure, outcome: names (strings)
```

Optional inputs (mr_mbe options):
weighting: "weighted" or "unweighted" (default: weighted)
stderror: "simple" or "delta" (default: delta)
phi: bandwidth multiplier (default: 1)
seed: random seed (for reproducibility)
iterations: bootstrap iterations (default: large, e.g., 1000+)
distribution: "normal" or "t-dist" (default: normal)
alpha: significance level for CIs (default often 0.05 or 0.001 as set)

Expected outputs (MRMBE instance)
Exposure, Outcome, Weighting, StdErr, Phi, Alpha (from config)
Estimate: the MBE causal estimate (mode of BetaIV distribution)
StdError: bootstrap/MAD-based standard error
CILower, CIUpper: confidence interval bounds
SNPs: number or list of SNPs used
Pvalue: two-sided P-value (based on the chosen distribution)
Other metadata reflecting inputs and computation

Notes
The analysis filters out invalid instruments (e.g., non-positive SE, NaNs, zero betaX) and requires at least two valid SNPs for t-distribution CIs.
Output is an MRMBE object containing both the configuration and the results for downstream use.
*/

```
// --- MRInput Class ---
class MRInput {
    constructor({betaX, betaY, betaXse, betaYse, correlation = null, exposure = "Exposure", outcome = "Outcome", snps = null}) {
        this.betaX = betaX || [];
        this.betaY = betaY || [];
        this.betaXse = betaXse || [];
        this.betaYse = betaYse || [];
        this.correlation = correlation; // Can be a math.js matrix or 2D array
        this.exposure = exposure;
        this.outcome = outcome;
        // snps: Array of SNP names.
        this.snps = snps || Array.from({length: betaX.length}, (_, i) => `SNP_${i + 1}`);

        const lengths = new Set([this.betaX.length, this.betaY.length, this.betaXse.length, this.betaYse.length, this.snps.length]);
        if (lengths.size !== 1) {
            throw new Error("MRInput: All input beta, SE, and SNP arrays must have the same length.");
        }
    }
}
```

```
// --- MRMBE Class ---
class MRMBE {
    constructor({
        Exposure, Outcome, Weighting, StdErr, Phi, Estimate, StdError, CILower, CIUpper, SNPs, Pvalue, Alpha
    }) {
        this.Exposure = Exposure;
        this.Outcome = Outcome;
        this.Weighting = Weighting;
        this.StdErr = StdErr;
        this.Phi = Phi;
        this.Estimate = Estimate;
        this.StdError = StdError;
        this.CILower = CILower;
        this.CIUpper = CIUpper;
        this.SNPs = SNPs;
        this.Pvalue = Pvalue;
        this.Alpha = Alpha;
    }
}

// --- Helper Functions ---

/**
 * Calculates the sample standard deviation).
 * @param {number[]} arr - The array of numbers.
 * @returns {number} The sample standard deviation.
 */
function calculateSd(arr) {
    if (arr.length < 2) return 0; // sd undefined for less than 2 points
    return jStat.stdev(arr, false); // false for sample standard deviation
}

/**
 * Calculates the median of an array.
 * @param {number[]} arr - The array of numbers.
 * @returns {number} The median.
 */
function calculateMedian(arr) {
    if (arr.length === 0) return NaN;
    const sortedArr = [...arr].sort((a, b) => a - b);
    const mid = Math.floor(sortedArr.length / 2);
    if (sortedArr.length % 2 === 0) {
        return (sortedArr[mid - 1] + sortedArr[mid]) / 2;
    } else {
```

```
            return sortedArr[mid];
        }
}

/**
 * Calculates the Median Absolute Deviation (MAD), manually implemented.
 * @param {number[]} arr - The array of numbers.
 * @returns {number} The MAD.
 */
function calculateMad(arr) {
    if (arr.length === 0) return 0;

    // 1. Calculate the median of the data
    const median = calculateMedian(arr);

    // 2. Calculate the absolute deviations from the median
    const absoluteDeviations = arr.map(val => Math.abs(val - median));

    // 3. Calculate the median of these absolute deviations
    const mad_value = calculateMedian(absoluteDeviations);

    // 4. Multiply by a constant (1.4826 for consistency with normal distribution std dev)
    const constant = 1.4826;
    return mad_value * constant;
}


/**
 * Implements a weighted Gaussian Kernel Density Estimate.
 * @param {number[]} data - The data points.
 * @param {number[]} weights - The weights corresponding to each data point.
 * @param {number} bandwidth - The bandwidth (h).
 * @param {number} [nPoints=512] - Number of points for the density grid.
 * @returns {{x: number[], y: number[]}} An object with x (grid points) and y (density values).
 */
function weightedKDE(data, weights, bandwidth, nPoints = 512) {
    if (data.length === 0 || bandwidth <= 0 || !isFinite(bandwidth)) {
        return { x: [], y: [] };
    }

    const dataMin = Math.min(...data);
    const dataMax = Math.max(...data);

    // Handle case of single data point or all data points being identical
    let stdDev = calculateSd(data);
```

```
    if (stdDev === 0) { // All data points are the same
        const x_single = [data[0]];
        const y_single = [1]; // Represent as a spike at that point
        return { x: x_single, y: y_single };
    }

    // Define evaluation points (x-grid)
    const extendFactor = 4; // Extend range by 4 std deviations on each side
    const xMin = dataMin - extendFactor * stdDev;
    const xMax = dataMax + extendFactor * stdDev;

    const x = [];
    for (let i = 0; i < nPoints; i++) {
        x.push(xMin + (xMax - xMin) * i / (nPoints - 1));
    }

    const y = new Array(nPoints).fill(0);
    const totalWeights = weights.reduce((sum, w) => sum + w, 0);

    if (totalWeights === 0) {
        return { x: x, y: new Array(nPoints).fill(0) };
    }

    const invBandwidth = 1 / bandwidth;
    const invSqrt2Pi = 1 / Math.sqrt(2 * Math.PI);

    for (let i = 0; i < nPoints; i++) {
        const currentX = x[i];
        let densitySum = 0;
        for (let j = 0; j < data.length; j++) {
            const diff = (currentX - data[j]) * invBandwidth;
            densitySum += weights[j] * invSqrt2Pi * Math.exp(-0.5 * diff * diff);
        }
        y[i] = densitySum * invBandwidth / totalWeights;
    }

    return { x: x, y: y };
}

/**
 * Calculates the normal distribution confidence interval.
 */
function ci_normal(type, estimate, se, alpha) {
    const z = jStat.normal.inv(1 - alpha / 2, 0, 1);
    if (type === "l") return estimate - z * se;
```

```
    if (type === "u") return estimate + z * se;
    return NaN;
}

/**
 * Calculates the t-distribution confidence interval.
 */
function ci_t(type, estimate, se, df, alpha) {
    if (df <= 0) return NaN; // Degrees of freedom must be positive
    const t = jStat.studentt.inv(1 - alpha / 2, df);
    if (type === "l") return estimate - t * se;
    if (type === "u") return estimate + t * se;
    return NaN;
}


// --- mbe_est Function ---
/**
 * Mode-based estimate (Hartwig) estimation function.
 * @param {number[]} BetaIV_in - Ratio causal estimates for each genetic variant.
 * @param {number[]} seBetaIV_in - Standard errors of ratio causal estimates.
 * @param {number} phi_in - Bandwidth multiplication factor.
 * @returns {number} Mode-based estimate.
 */
function mbe_est(BetaIV_in, seBetaIV_in, phi_in) {
    // Robustness check for input data
    const n = BetaIV_in.length;
    if (n === 0) return NaN;

    const sd_BetaIV = calculateSd(BetaIV_in);
    const mad_BetaIV = calculateMad(BetaIV_in);

    // s <- 0.9*(min(sd(BetaIV.in), mad(BetaIV.in)))/length(BetaIV.in)^(1/5)

    // If all BetaIV_in values are identical, s_val (and thus h) would be 0, which breaks KDE.
    // In this specific case, the mode is simply that value.
    if (new Set(BetaIV_in).size === 1) {
        return BetaIV_in[0];
    }

    let s_val = Math.min(sd_BetaIV, mad_BetaIV);
    // If sd_BetaIV or mad_BetaIV is zero (but not all BetaIV are identical, perhaps due to tiny numerical differences),
    // provide a small positive default for s_val to prevent h from becoming zero.
    if (s_val === 0) {
        s_val = 1e-6; // Small positive value
```

```
    }
    const s = 0.9 * s_val / Math.pow(n, 1/5);

    // Standardised weights: seBetaIV.in^-2 / sum(seBetaIV.in^-2)
    const inv_se_sq = seBetaIV_in.map(se => {
        // Handle potential zero SEs. If an SE is 0, its weight should be infinite, dominating.
        // However, BetaIVs from such SNPs are problematic themselves.
        // The calling `mr_mbe` function already filters for `seBetaIV[i] > 0`.
        return 1 / (se * se);
    });

    const sum_inv_se_sq = inv_se_sq.reduce((sum, val) => sum + val, 0);

    let s_weights;
    if (sum_inv_se_sq === 0) {
        // This should ideally not happen if seBetaIV_in are all > 0.
        // If it does, all weights would be zero. Fallback to uniform weights.
        s_weights = new Array(n).fill(1 / n);
        console.warn("Sum of inverse squared standard errors is zero. Defaulting to unweighted mode estimation.");
    } else {
        s_weights = inv_se_sq.map(val => val / sum_inv_se_sq);
    }

    // Define the actual bandwidth
    let h = s * phi_in;
    if (h <= 0 || !isFinite(h)) {
        // If calculated bandwidth is problematic, use a robust fallback (e.g., Scott's rule)
        const fallback_h = 1.06 * sd_BetaIV * Math.pow(n, -1/5);
        console.warn(`Calculated bandwidth (h=${h}) is non-positive or non-finite. Using fallback bandwidth:
${fallback_h}`);
        if (fallback_h <= 0 || !isFinite(fallback_h)) {
            return NaN; // Cannot proceed if bandwidth is truly unusable
        }
        h = fallback_h;
    }

    // Compute the smoothed empirical density function
    const densityIV = weightedKDE(BetaIV_in, s_weights, h);

    if (densityIV.x.length === 0 || densityIV.y.length === 0 || !densityIV.y.some(y_val => y_val > 0)) {
        console.warn("KDE returned empty or invalid density. Cannot determine mode-based estimate.");
        return NaN;
    }

    // Extract the point with the highest density as the point estimate
```

```
    let max_density = -Infinity;
    let beta_est = NaN;
    for (let i = 0; i < densityIV.y.length; i++) {
        if (densityIV.y[i] > max_density) {
            max_density = densityIV.y[i];
            beta_est = densityIV.x[i];
        }
    }
    return beta_est;
}

// --- mbe_boot Function ---
/**
 * Mode-based estimate (Hartwig) bootstrap function.
 * @param {number[]} BetaIV_in - Ratio causal estimates for each genetic variant.
 * @param {number[]} seBetaIV_in - Standard errors of ratio causal estimates.
 * @param {string} weighting_in - "weighted" or "unweighted".
 * @param {number} iterations_in - Number of bootstrap iterations.
 * @param {number} phi_in - Bandwidth multiplication factor.
 * @returns {number[]} Bootstrapped mode-based estimates.
 */
function mbe_boot(BetaIV_in, seBetaIV_in, weighting_in, iterations_in, phi_in) {
    if (iterations_in <= 0 || !isFinite(iterations_in)) {
        console.warn("Bootstrap iterations must be a positive finite number. Returning empty array.");
        return [];
    }

    const beta_boot_results = [];
    const n_snps = BetaIV_in.length;

    for (let i = 0; i < iterations_in; i++) {
        const BetaIV_boot = new Array(n_snps);
        // Sample from normal distribution for each SNP's BetaIV
        for (let j = 0; j < n_snps; j++) {
            // Ensure mean and sd are finite for jStat.normal.sample
            if (isFinite(BetaIV_in[j]) && isFinite(seBetaIV_in[j]) && seBetaIV_in[j] >= 0) {
                BetaIV_boot[j] = jStat.normal.sample(BetaIV_in[j], seBetaIV_in[j]);
            } else {
                // If a SNP has invalid mean/sd, its bootstrapped value might be problematic.
                // For simplicity, for this SNP, use its original value or mark as NaN.
                // Since mr_mbe filters for finite BetaIV and positive seBetaIV, this path should be rare.
                BetaIV_boot[j] = BetaIV_in[j]; // Use original value as fallback
                console.warn(`Invalid mean/sd for SNP ${j} in bootstrap sampling. Using original value.`);
            }
        }
```

```
            let current_seBetaIV;
            if (weighting_in === "weighted") {
                current_seBetaIV = seBetaIV_in;
            } else if (weighting_in === "unweighted") {
                current_seBetaIV = new Array(n_snps).fill(1);
            } else {
                console.error(`Invalid weighting.in: ${weighting_in}. Must be "weighted" or "unweighted".`);
                continue; // Skip this iteration
            }

            const est = mbe_est(BetaIV_boot, current_seBetaIV, phi_in);
            if (isFinite(est)) {
                beta_boot_results.push(est);
            }
        }

        return beta_boot_results;
}

// --- mr_mbe Function ---
/**
 * Performs Mendelian Randomization analysis using the Mode-Based Estimate (MBE) method.
 *
 * @param {MRInput} object - MRInput object containing genetic data.
 * @param {object} [options={}] - Configuration options for the analysis.
 * @param {string} [options.weighting="weighted"] - Weighting method: "weighted" or "unweighted".
 * @param {string} [options.stderror="delta"] - Standard error calculation method: "simple" or "delta".
 * @param {number} [options.phi=1] - Bandwidth multiplication factor.
 * @param {number} [options.seed=314159265] - Seed for random number generation (Note: JavaScript Math.random is not directly seedable in a standard way).
 * @param {number} [options.iterations=10000] - Number of bootstrap iterations.
 * @param {string} [options.distribution="normal"] - Distribution for P-value and CI: "normal" or "t-dist".
 * @param {number} [options.alpha=0.001] - Significance level for confidence intervals.
 * @returns {MRMBE} An MRMBE object containing the analysis results.
 */
function mr_mbe(object, options = {}) {
    let {
        weighting = "weighted",
        stderror = "delta",
        phi = 1,
        seed = 314159265, // Note: Seed has limited impact in standard JS Math.random, used by jStat.
        iterations = 10000,
        distribution = "normal",
        alpha = 0.001
```

```
} = options;

// For strict reproducibility, a dedicated seeded PRNG library would be needed.
// console.log(`Seed parameter (${seed}) is provided but may not ensure exact reproducibility across JS environments/runs due to Math.random limitations.`);

const Bx = object.betaX;
const By = object.betaY;
const Bxse = object.betaXse;
const Byse = object.betaYse;
const nsnps_original = Bx.length; // Keep track of original number of SNPs

// --- Input Validation ---
if (!["weighted", "unweighted"].includes(weighting)) {
    throw new Error("Weighting must be 'weighted' or 'unweighted'.");
}
if (!["simple", "delta"].includes(stderror)) {
    throw new Error("Stderror must be 'simple' or 'delta'.");
}
if (!["normal", "t-dist"].includes(distribution)) {
    throw new Error("Distribution must be 'normal' or 't-dist'.");
}
if (nsnps_original === 0) {
    throw new Error("No SNPs provided for MBE analysis.");
}

// Filter out invalid SNPs early to prevent NaN/Infinity issues
const filteredBx = [];
const filteredBy = [];
const filteredBxse = [];
const filteredByse = [];
const filteredSnpNames = [];

for (let i = 0; i < nsnps_original; i++) {
    const currentBx = Bx[i];
    const currentBy = By[i];
    const currentBxse = Bxse[i];
    const currentByse = Byse[i];
    const snpName = object.snps[i];

    // Ensure all are finite, SEs are positive, and Bx is non-zero for ratio
    if (isFinite(currentBx) && isFinite(currentBy) && isFinite(currentBxse) && isFinite(currentByse) &&
        currentBxse > 0 && currentByse > 0 && currentBx !== 0) {
        filteredBx.push(currentBx);
        filteredBy.push(currentBy);
```

```
                filteredBxse.push(currentBxse);
                filteredByse.push(currentByse);
                filteredSnpNames.push(snpName);
            } else {
                console.warn(`SNP '${snpName}' (index ${i}) excluded from MR-MBE analysis due to invalid initial data (non-finite values, non-positive SEs, or zero betaX): ` +
                             `betaX=${currentBx}, betaY=${currentBy}, betaXse=${currentBxse}, betaYse=${currentByse}.`);
            }
        }

        const nsnps_filtered = filteredBx.length;
        if (nsnps_filtered === 0) {
            throw new Error("No valid SNPs remaining after initial filtering for MR-MBE analysis.");
        }

        const BetaIV = filteredBy.map((y, i) => y / filteredBx[i]);

        let seBetaIV;
        if (stderror === "simple") {
            seBetaIV = filteredByse.map((se, i) => se / Math.abs(filteredBx[i]));
        } else { // "delta"
            seBetaIV = filteredByse.map((seY, i) => {
                const y = filteredBy[i];
                const x = filteredBx[i];
                const seX = filteredBxse[i];
                // sqrt(Byse^2/abs(Bx)^2 + By^2*Bxse^2/Bx^4)
                return Math.sqrt( (seY**2 / (x**2)) + (y**2 * seX**2 / (x**4)) );
            });
        }

        // Ensure seBetaIV elements are all finite and positive for mbe_est and mbe_boot
        const finalBetaIV = [];
        const finalSeBetaIV = [];
        for (let i = 0; i < BetaIV.length; i++) {
            if (isFinite(BetaIV[i]) && isFinite(seBetaIV[i]) && seBetaIV[i] > 0) {
                finalBetaIV.push(BetaIV[i]);
                finalSeBetaIV.push(seBetaIV[i]);
            } else {
                console.warn(`SNP originally '${filteredSnpNames[i]}' (after initial filtering) excluded from MBE calculation due to invalid ratio estimate or SE: BetaIV=${BetaIV[i]}, seBetaIV=${seBetaIV[i]}.`);
            }
        }

        const nsnps_final = finalBetaIV.length;
```

```
if (nsnps_final === 0) {
    throw new Error("No valid SNPs remaining after ratio and SE calculation filtering for MR-MBE analysis.");
}

// Adjust distribution for p-value/CI if too few SNPs for t-distribution
if (nsnps_final < 2 && distribution === "t-dist") { // t-distribution requires df = n-1 >= 1
     console.warn("Fewer than 2 SNPs remaining for t-distribution (df < 1). Switching to 'normal' distribution.");
     distribution = "normal";
}

let betaMBE;
if (weighting === "weighted") {
    betaMBE = mbe_est(finalBetaIV, finalSeBetaIV, phi);
} else { // "unweighted"
    betaMBE = mbe_est(finalBetaIV, new Array(nsnps_final).fill(1), phi);
}

// Handle potential NaN from mbe_est if data is problematic (e.g., all identical and s_val=0 handled badly)
if (!isFinite(betaMBE)) {
    throw new Error("Failed to calculate a finite mode-based estimate. Check input data and phi parameter.");
}

const betaMBE_boot_results = mbe_boot(finalBetaIV, finalSeBetaIV, weighting, iterations, phi);

// Filter out NaN/Infinity from bootstrap results before MAD
const finite_betaMBE_boot = betaMBE_boot_results.filter(isFinite);

let seMBE;
if (finite_betaMBE_boot.length > 1) { // MAD requires at least 2 points
    seMBE = calculateMad(finite_betaMBE_boot);
} else {
    seMBE = NaN; // Cannot calculate valid SE from bootstrap
    console.warn("Too few finite bootstrap estimates to calculate standard error (MAD). StdError will be NaN.");
}

let pvalue = NaN;
if (isFinite(betaMBE) && isFinite(seMBE) && seMBE > 0) {
    if (distribution === "normal") {
        pvalue = 2 * jStat.normal.cdf(-Math.abs(betaMBE / seMBE), 0, 1);
    } else { // "t-dist"
        pvalue = 2 * jStat.studentt.cdf(-Math.abs(betaMBE / seMBE), nsnps_final - 1);
    }
} else {
    console.warn("Cannot calculate P-value due to non-finite estimate or standard error, or zero standard error.");
}
```

```
    let ciLower = NaN, ciUpper = NaN;
    if (isFinite(betaMBE) && isFinite(seMBE) && seMBE > 0) {
        if (distribution === "normal") {
            ciLower = ci_normal("l", betaMBE, seMBE, alpha);
            ciUpper = ci_normal("u", betaMBE, seMBE, alpha);
        } else { // "t-dist"
            ciLower = ci_t("l", betaMBE, seMBE, nsnps_final - 1, alpha);
            ciUpper = ci_t("u", betaMBE, seMBE, nsnps_final - 1, alpha);
        }
    } else {
        console.warn("Cannot calculate confidence interval due to non-finite estimate or standard error, or zero standard error.");
    }

    return new MRMBE({
        Exposure: object.exposure,
        Outcome: object.outcome,
        Weighting: weighting,
        StdErr: stderror,
        Phi: phi,
        Estimate: betaMBE,
        StdError: seMBE,
        CILower: ciLower,
        CIUpper: ciUpper,
        SNPs: nsnps_original, // Total original SNPs provided to the function
        Pvalue: pvalue,
        Alpha: alpha
    });
}
```

(9) The Penalized Inverse-Variance Weighted Method (PIVW)

The JavaScript codes for The Penalized Inverse-Variance Weighted Method are as follows:

```
/*
Method name and aim
Name: Penalized Inverse-Variance Weighted Method (PIVW) for Mendelian Randomization.
Objective: To estimate a causal effect using multiple genetic instruments with a penalization mechanism that downweights weaker instruments, while optionally accounting for over-dispersion and using bootstrapped Fieller-style confidence intervals.

Main functions and components
rnorm_array: generates random numbers from a normal distribution (supports scalar or per-snp means and sds).
array helpers (array_op, array_scalar_op, array_sum, array_mean, array_pow): utilities for element-wise and aggregate array computations.
BF_dist: internal bootstrap routine that generates Fieller-style bootstrap samples for confidence intervals.
```

mr_pivw: the main function that computes the penalized IVW estimate, optionally performs instrument selection (delta and sel_pval), handles over-dispersion, and returns CIs (normal or Fieller), p-value, and other diagnostics.

Required inputs and expected outputs

Inputs (required):
betaX: array of SNP-exposure effects.
betaY: array of SNP-outcome effects.
betaXse: array of standard errors for betaX.
betaYse: array of standard errors for betaY.
exposure: name of the exposure (string).
outcome: name of the outcome (string).
Optional controls: lambda (penalty), over_dispersion (boolean), delta (IV-selection tuning), sel_pval (array of p-values for selection), Boot_Fieller (boolean), alpha (significance level for CIs).

Outputs (expected):
An object with: Estimate, StdError, Pvalue, CILower, CIUpper, CI (full CI structure), Confidence/Fieller settings, Tau2 (heterogeneity), SNPs (number used), Exposure, Outcome, Over_dispersion, Delta, Lambda, Boot_Fieller, Alpha, and Condition (optional diagnostic).

Behavior notes:
The function may subset instruments if IV selection or over-dispersion settings require it.
Confidence intervals can be either normal-based or Fieller-based (bootstrapped) depending on settings.
*/

```
/**
 * Generates an array of random numbers from a normal distribution.
 * Requires jStat to be loaded.
 * @param {number} n The number of samples to generate.
 * @param {number|Array<number>} mean The mean of the distribution(s). Can be a single value or an array.
 * @param {number|Array<number>} sd The standard deviation of the distribution(s). Can be a single value or an array.
 * @returns {Array<number>} An array of random numbers.
 */
function rnorm_array(n, mean, sd) {
    const result = [];
    if (Array.isArray(mean) && Array.isArray(sd) && mean.length === n && sd.length === n) {
        for (let i = 0; i < n; i++) {
            result.push(jStat.normal.sample(mean[i], sd[i]));
        }
    } else if (typeof mean === 'number' && typeof sd === 'number') {
        for (let i = 0; i < n; i++) {
            result.push(jStat.normal.sample(mean, sd));
        }
    } else {
        throw new Error("rnorm_array: mean and sd must be numbers or arrays of the same length as n, or single numbers.");
```

```
    }
    return result;
}

/**
 * Helper to perform element-wise operation on two arrays.
 * @param {Array<number>} arr1 First array.
 * @param {Array<number>} arr2 Second array.
 * @param {function} operation The function to apply (e.g., (a, b) => a + b).
 * @returns {Array<number>} New array with results.
 */
function array_op(arr1, arr2, operation) {
    if (arr1.length !== arr2.length) {
        throw new Error("Arrays must have the same length for element-wise operation.");
    }
    return arr1.map((val, idx) => operation(val, arr2[idx]));
}

/**
 * Helper to perform element-wise operation on an array with a scalar.
 * @param {Array<number>} arr Array.
 * @param {number} scalar Scalar value.
 * @param {function} operation The function to apply (e.g., (a, b) => a * b).
 * @returns {Array<number>} New array with results.
 */
function array_scalar_op(arr, scalar, operation) {
    return arr.map(val => operation(val, scalar));
}

/**
 * Helper for sum of an array.
 * @param {Array<number>} arr The array to sum.
 * @returns {number} The sum.
 */
function array_sum(arr) {
    return arr.reduce((acc, val) => acc + val, 0);
}

/**
 * Helper for mean of an array.
 * @param {Array<number>} arr The array to average.
 * @returns {number} The mean.
 */
function array_mean(arr) {
    if (arr.length === 0) return 0; // Handle empty array
```

```
    return arr.reduce((acc, val) => acc + val, 0) / arr.length;
}

/**
 * Helper for element-wise power.
 * @param {Array<number>} arr Array.
 * @param {number} p Power.
 * @returns {Array<number>} New array with results.
 */
function array_pow(arr, p) {
    return arr.map(val => Math.pow(val, p));
}

// Function to handle jStat seeding. Assumes jStat.setSeed might be available
// or relies on Math.random. For reliable seeding, a library like 'seedrandom'
// patching Math.random is recommended, or passing a PRNG to jStat methods if supported.
function setJStatSeed(seed) {
    if (typeof jStat.setSeed === 'function') {
        jStat.setSeed(seed);
    } else {
        console.warn("jStat.setSeed is not available in this jStat version. Random numbers may not be reproducible without a custom PRNG (e.g., 'seedrandom').");
        // Fallback for demonstration, not robust for all jStat versions:
        // If 'seedrandom' is loaded: `Math.seedrandom(seed);`
    }
}

// --- BF_dist Function ---

/**
 * Internal function of the penalized inverse-variance weighted (pIVW) method,
 * which generates bootstrap samples for the bootstrapping Fieller's confidence interval.
 *
 * @param {object} object An MRInput object (JS plain object with properties).
 * @param {Array<number>} object.betaX Exposure effect estimates.
 * @param {Array<number>} object.betaY Outcome effect estimates.
 * @param {Array<number>} object.betaXse Standard errors of exposure effect estimates.
 * @param {Array<number>} object.betaYse Standard errors of outcome effect estimates.
 * @param {number} [beta_hat=0] The causal effect estimate.
 * @param {number} [tau2=0] The estimated variance of the horizontal pleiotropy.
 * @param {number} [lambda=1] The penalty parameter in the pIVW estimator.
 * @param {number} [n_boot=1000] The sample size of the bootstrap samples.
 * @param {number} [seed_boot=1] The seed for random sampling in the bootstrap method.
 * @returns {Array<number>} A vector containing the bootstrap samples for the bootstrapping Fieller's confidence interval.
 */
```

```
function BF_dist(object, beta_hat = 0, tau2 = 0, lambda = 1, n_boot = 1000, seed_boot = 1) {
    setJStatSeed(seed_boot);

    let i = 0;
    const z_b = [];

    while (i < n_boot) {
        const Bx = object.betaX;
        const Byse = object.betaYse;
        const Bxse = object.betaXse;
        const n_snps = Bx.length;

        // Ensure tau2 is non-negative for sqrt
        const sqrt_tau2 = Math.sqrt(Math.max(0, tau2));

        const alpha = rnorm_array(n_snps, 0, sqrt_tau2);

        const By_hat_means = Bx.map((val, idx) => val * beta_hat + alpha[idx]);
        const By_hat = rnorm_array(n_snps, By_hat_means, Byse);
        const Bx_hat = rnorm_array(n_snps, Bx, Bxse);

        // Pre-calculate common terms for efficiency
        const Byse_neg4 = array_pow(Byse, -4);
        const Byse_neg2 = array_pow(Byse, -2);
        const By_hat_sq = array_pow(By_hat, 2);
        const Bx_hat_sq = array_pow(Bx_hat, 2);
        const Byse_sq = array_pow(Byse, 2);
        const Bxse_sq = array_pow(Bxse, 2);

        const v1_terms = Bx.map((_, idx) => {
            const term1 = By_hat_sq[idx] * Bx_hat_sq[idx];
            const term2 = (By_hat_sq[idx] - Byse_sq[idx] - tau2) * (Bx_hat_sq[idx] - Bxse_sq[idx]);
            return Byse_neg4[idx] * (term1 - term2);
        });
        const v1 = array_sum(v1_terms);

        const v2_terms = Bx.map((_, idx) => {
            const term1 = 4 * (Bx_hat_sq[idx] - Bxse_sq[idx]) * Bxse_sq[idx];
            const term2 = 2 * Math.pow(Bxse_sq[idx], 2); // Bxse^4
            return Byse_neg4[idx] * (term1 + term2);
        });
        const v2 = array_sum(v2_terms);

        const v12_terms = Bx.map((_, idx) => {
            return 2 * Byse_neg4[idx] * By_hat[idx] * Bx_hat[idx] * Bxse_sq[idx];
```

```
});
const v12 = array_sum(v12_terms);

const mu1_terms = Bx.map((_, idx) => {
    return Byse_neg2[idx] * By_hat[idx] * Bx_hat[idx];
});
const mu1 = array_sum(mu1_terms);

const mu2_terms = Bx.map((_, idx) => {
    return Byse_neg2[idx] * (Bx_hat_sq[idx] - Bxse_sq[idx]);
});
const mu2 = array_sum(mu2_terms);

const mu2p = mu2 / 2 + Math.sign(mu2) * Math.sqrt(Math.max(0, mu2 * mu2 / 4 + lambda * v2));
// Handle potential division by zero for mu1p if v2 is zero.
// If v2 is 0, v12/v2 would be Inf/NaN unless v12 is also 0.
 // Assuming v2 is non-zero, as per the calculation for v2.
let mu1p;
if (v2 === 0) {
    mu1p = mu1; // If v2 is zero, (v12/v2)*(mu2p-mu2) becomes 0, so mu1p = mu1. This is a common limit.
} else {
    mu1p = mu1 + (v12 / v2) * (mu2p - mu2);
}

let w;
if ((2 * mu2p - mu2) === 0) {
    // This case would imply mu2p = mu2/2. If v2 > 0, this should not happen.
    // If v2 = 0, then mu2p = mu2/2 + sign(mu2)|mu2|/2.
    // If mu2 > 0, mu2p = mu2, then 2*mu2p-mu2 = mu2.
    // If mu2 < 0, mu2p = 0, then 2*mu2p-mu2 = -mu2.
    // If mu2 = 0, mu2p = 0, then 2*mu2p-mu2 = 0. This is the only division by zero case.
    // If mu2 is 0 and v2 is 0, mu2p is 0, leading to 0/0 for w.
    // In these degenerate cases, we skip the sample.
    w = NaN;
} else {
    w = mu2p / (2 * mu2p - mu2);
}

const v1p = v1;
const v2p = w * w * v2;
const v12p = w * v12;

const denominator = (v1p - 2 * beta_hat * v12p + beta_hat * beta_hat * v2p);

// Check for invalid values before calculation
```

```
            if (denominator === 0 || isNaN(denominator) || !isFinite(denominator)) {
                continue; // Skip this iteration
            }

            const temp = Math.pow(mu1p - beta_hat * mu2p, 2) / denominator;

            if (temp < 0 || isNaN(temp) || !isFinite(temp)) {
                continue;
            } else {
                z_b.push(temp);
                i = i + 1;
            }
        }
        z_b.sort((a, b) => a - b); // Sort in ascending order
        return z_b;
}


// --- mr_pivw Function ---

/**
 * Calculates the penalized inverse-variance weighted estimate (and relevant statistics).
 * Requires jStat to be loaded.
 *
 * @param {object} object An MRInput object (JS plain object with properties).
 * @param {Array<number>} object.betaX Exposure effect estimates.
 * @param {Array<number>} object.betaY Outcome effect estimates.
 * @param {Array<number>} object.betaXse Standard errors of exposure effect estimates.
 * @param {Array<number>} object.betaYse Standard errors of outcome effect estimates.
 * @param {string} [object.exposure=""] Name of the exposure.
 * @param {string} [object.outcome=""] Name of the outcome.
 * @param {number} [lambda=1] The penalty parameter.
 * @param {boolean} [over_dispersion=true] Whether to estimate and account for over-dispersion.
 * @param {number} [delta=0] Tuning parameter for instrumental variable selection.
 * @param {Array<number>} [sel_pval=null] P-values for instrumental variable selection.
 * @param {boolean} [Boot_Fieller=true] Whether to use bootstrapping Fieller's method for confidence intervals.
 * @param {number} [alpha=0.001] Significance level for confidence intervals.
 * @returns {object|null} An object containing the pIVW results or null if an error occurs.
 */
function mr_pivw(object, lambda = 1, over_dispersion = true, delta = 0, sel_pval = null, Boot_Fieller = true, alpha = 0.001) {

    // Make copies of original data to allow subsetting later
    const Bx_orig = [...object.betaX];
    const By_orig = [...object.betaY];
    const Bxse_orig = [...object.betaXse];
```

```
    const Byse_orig = [...object.betaYse];

    // Working copies that might get subsetted
    let Bx = [...Bx_orig];
    let By = [...By_orig];
    let Bxse = [...Bxse_orig];
    let Byse = [...Byse_orig];

    let p = Bx.length; // Current number of SNPs

    // --- Parameter Validation ---
    if (lambda < 0) {
        console.error("'lambda' cannot be smaller than zero.");
        return null;
    }
    if (delta < 0) {
        console.error("'delta' cannot be smaller than zero.");
        return null;
    }
    if (alpha <= 0) {
        alpha = 0.001;
        console.warn("'alpha' provided is less than or equal to zero. 'alpha' is set to be 0.001.");
    }

    let kappa;
    let eta;

    // --- Instrumental Variable Selection (if delta > 0) ---
    if (delta > 0) {
        if (!sel_pval || sel_pval.length !== p) {
            delta = 0;
            console.warn("'sel_pval' is not provided or its length doesn't match the number of snps in the MRInput object. 'delta' is set to be zero and no IV selection conducted.");
        } else {
            // Determine selected SNPs based on delta and sel_pval
            const sel_indices = sel_pval.map((pval, idx) => {
                                    return pval < 2 * (1 - jStat.normal.cdf(delta, 0, 1)) ? idx : -1;
                                }).filter(idx => idx !== -1);

            if (sel_indices.length === 0) {
                console.error("No snps is kept in the analysis after IV selection. Please try a smaller 'delta' value.");
                return null;
            }

            // Calculate kappa and eta using selected SNPs (from original data)
```

```
            const Bx_sel_for_kappa = sel_indices.map(idx => Bx_orig[idx]);
            const Bxse_sel_for_kappa = sel_indices.map(idx => Bxse_orig[idx]);

            const Bx_div_Bxse_sq = array_op(Bx_sel_for_kappa, Bxse_sel_for_kappa, (b, bse) => (b / bse) * (b / bse));
            kappa = array_mean(Bx_div_Bxse_sq) - 1;
            eta = kappa * Math.sqrt(sel_indices.length);

            // Calculation for psi2 and adjusting eta
            const sel_z = sel_pval.map(pval => jStat.normal.inv(1 - pval / 2, 0, 1));

            const q = sel_z.map(z => jStat.normal.cdf(z - delta, 0, 1) + jStat.normal.cdf(-z - delta, 0, 1));

            const psi2_terms = Bx_orig.map((b, idx) => {
                const term1 = Math.pow(b / Bxse_orig[idx], 4);
                const term2 = 6 * Math.pow(b / Bxse_orig[idx], 2);
                const term3 = 3;
                return (term1 - term2 + term3) * q[idx] * (1 - q[idx]);
            });
            const psi2 = array_sum(psi2_terms) / sel_indices.length;
            eta = eta / Math.max(1, Math.sqrt(psi2));
        }
    } else {
        // No IV selection (delta = 0)
        const Bx_div_Bxse_sq = array_op(Bx, Bxse, (b, bse) => (b / bse) * (b / bse));
        kappa = array_mean(Bx_div_Bxse_sq) - 1;
        eta = kappa * Math.sqrt(p);
    }

    // --- Conditional Subsetting (if delta != 0 and over_dispersion is FALSE) ---
    if (delta !== 0 && !over_dispersion) {
        const sel_indices = sel_pval.map((pval, idx) => pval < 2 * (1 - jStat.normal.cdf(delta, 0, 1)) ? idx : -1)
                                    .filter(idx => idx !== -1);

        Bx = sel_indices.map(idx => Bx_orig[idx]);
        By = sel_indices.map(idx => By_orig[idx]);
        Bxse = sel_indices.map(idx => Bxse_orig[idx]);
        Byse = sel_indices.map(idx => Byse_orig[idx]);
        p = Bx.length; // Update p to reflect the new number of SNPs
    }

    // --- Core pIVW Estimate Calculation ---
    // These calculations are repeated if delta != 0 and over_dispersion == TRUE in R.
    // We'll calculate them here, and potentially re-calculate after subsetting below.

    let v2_terms_denom = array_pow(Byse, -4);
```

```
    let v2_inner_term1 = array_op(array_pow(Bx, 2), array_pow(Bxse, 2), (bx_sq, bxse_sq) => 4 * (bx_sq - bxse_sq) * bxse_sq);
    let v2_inner_term2 = array_pow(Bxse, 4).map(val => 2 * val);
    let v2_terms = array_op(v2_inner_term1, v2_inner_term2, (t1, t2) => t1 + t2);
    let v2 = array_sum(array_op(v2_terms_denom, v2_terms, (d, t) => d * t));

    let v12_terms = array_op(v2_terms_denom, By, (d, by) => d * by);
    v12_terms = array_op(v12_terms, Bx, (t, bx) => t * bx);
    v12_terms = array_op(v12_terms, array_pow(Bxse, 2), (t, bxse_sq) => 2 * t * bxse_sq);
    let v12 = array_sum(v12_terms);

    let mu_denom = array_pow(Byse, -2);
    let mu1_terms = array_op(mu_denom, By, (d, by) => d * by);
    mu1_terms = array_op(mu1_terms, Bx, (t, bx) => t * bx);
    let mu1 = array_sum(mu1_terms);

    let mu2_terms_inner = array_op(array_pow(Bx, 2), array_pow(Bxse, 2), (bx_sq, bxse_sq) => bx_sq - bxse_sq);
    let mu2_terms = array_op(mu_denom, mu2_terms_inner, (d, t) => d * t);
    let mu2 = array_sum(mu2_terms);

    let mu2p = mu2 / 2 + Math.sign(mu2) * Math.sqrt(Math.max(0, mu2 * mu2 / 4 + lambda * v2));
    let mu1p;
    if (v2 === 0) { // Handle potential division by zero for mu1p if v2 is zero
        mu1p = mu1;
    } else {
        mu1p = mu1 + (v12 / v2) * (mu2p - mu2);
    }
    let beta_pIVW = mu1p / mu2p; // Can be Inf or NaN if mu2p is 0.

    let tau2;
    if (over_dispersion) {
        const By_minus_betaBx_sq = array_op(By, array_scalar_op(Bx, beta_pIVW, (val, s) => val * s), (by, beta_bx) => Math.pow(by - beta_bx, 2));
        const Byse_sq = array_pow(Byse, 2);
        const beta_sq_Bxse_sq = array_scalar_op(array_pow(Bxse, 2), beta_pIVW * beta_pIVW, (val, s) => val * s);

        const numerator_terms = array_op(By_minus_betaBx_sq, Byse_sq, (t1, t2) => t1 - t2);
        const final_numerator_terms = array_op(numerator_terms, beta_sq_Bxse_sq, (t1, t2) => t1 - t2);
        const weighted_numerator = array_op(final_numerator_terms, array_pow(Byse, -2), (val, inv_se_sq) => val * inv_se_sq);

        const denominator_tau2 = array_sum(array_pow(Byse, -2));

        tau2 = array_sum(weighted_numerator) / denominator_tau2;
        if (tau2 < 0) {
```

```
            tau2 = 0;
        }
        if (isNaN(tau2) || !isFinite(tau2)) tau2 = 0; // Handle NaN/Inf results gracefully
    } else {
        tau2 = 0;
    }

    // --- Re-calculation if delta != 0 AND over_dispersion == TRUE ---
    if (delta !== 0 && over_dispersion) {
        const sel_indices = sel_pval.map((pval, idx) => pval < 2 * (1 - jStat.normal.cdf(delta, 0, 1)) ? idx : -1)
                                    .filter(idx => idx !== -1);

        Bx = sel_indices.map(idx => Bx_orig[idx]);
        By = sel_indices.map(idx => By_orig[idx]);
        Bxse = sel_indices.map(idx => Bxse_orig[idx]);
        Byse = sel_indices.map(idx => Byse_orig[idx]);
        p = Bx.length; // Update p to reflect the new number of SNPs

        // Re-calculate based on subsetted data
        v2_terms_denom = array_pow(Byse, -4);
        v2_inner_term1 = array_op(array_pow(Bx, 2), array_pow(Bxse, 2), (bx_sq, bxse_sq) => 4 * (bx_sq - bxse_sq) *
bxse_sq);
        v2_inner_term2 = array_pow(Bxse, 4).map(val => 2 * val);
        v2_terms = array_op(v2_inner_term1, v2_inner_term2, (t1, t2) => t1 + t2);
        v2 = array_sum(array_op(v2_terms_denom, v2_terms, (d, t) => d * t));

        v12_terms = array_op(v2_terms_denom, By, (d, by) => d * by);
        v12_terms = array_op(v12_terms, Bx, (t, bx) => t * bx);
        v12_terms = array_op(v12_terms, array_pow(Bxse, 2), (t, bxse_sq) => 2 * t * bxse_sq);
        v12 = array_sum(v12_terms);

        mu_denom = array_pow(Byse, -2);
        mu1_terms = array_op(mu_denom, By, (d, by) => d * by);
        mu1_terms = array_op(mu1_terms, Bx, (t, bx) => t * bx);
        mu1 = array_sum(mu1_terms);

        mu2_terms_inner = array_op(array_pow(Bx, 2), array_pow(Bxse, 2), (bx_sq, bxse_sq) => bx_sq - bxse_sq);
        mu2_terms = array_op(mu_denom, mu2_terms_inner, (d, t) => d * t);
        mu2 = array_sum(mu2_terms);

        mu2p = mu2 / 2 + Math.sign(mu2) * Math.sqrt(Math.max(0, mu2 * mu2 / 4 + lambda * v2));
        if (mu2p === 0 && mu1p !== 0) { // Division by zero for beta_pIVW
            beta_pIVW = (mu1p > 0) ? Infinity : -Infinity;
        } else if (mu2p === 0 && mu1p === 0) { // 0/0 case
            beta_pIVW = 0; // Or NaN, depending on desired behavior.
```

```
        } else {
            mu1p = mu1 + (v12 / v2) * (mu2p - mu2);
            beta_pIVW = mu1p / mu2p;
        }
    }

    // --- Standard Error Calculation ---
    const se_pIVW_term1_numerator = array_op(array_pow(Bx, 2), array_pow(Byse, 2), (bx_sq, byse_sq) => bx_sq / byse_sq);
    const se_pIVW_term1_multiplier = array_scalar_op(array_pow(Byse, -2), tau2, (inv_byse_sq, t2) => 1 + t2 * inv_byse_sq);
    const se_pIVW_term1 = array_op(se_pIVW_term1_numerator, se_pIVW_term1_multiplier, (n, m) => n * m);

    const se_pIVW_term2_numerator_left = array_scalar_op(array_pow(Bxse, 2), beta_pIVW * beta_pIVW, (bxse_sq, bpivw_sq) => bpivw_sq * bxse_sq);
    const se_pIVW_term2_numerator_right = array_op(array_pow(Bx, 2), array_pow(Bxse, 2), (bx_sq, bxse_sq) => bx_sq + bxse_sq);
    const se_pIVW_term2_numerator = array_op(se_pIVW_term2_numerator_left, se_pIVW_term2_numerator_right, (l, r) => l * r);
    const se_pIVW_term2_denominator = array_pow(Byse, 4);
    const se_pIVW_term2 = array_op(se_pIVW_term2_numerator, se_pIVW_term2_denominator, (n, d) => n / d);

    const se_pIVW_sum_terms = array_op(se_pIVW_term1, se_pIVW_term2, (t1, t2) => t1 + t2);
    let se_pIVW = (1 / Math.abs(mu2p)) * Math.sqrt(array_sum(se_pIVW_sum_terms));
    if (isNaN(se_pIVW) || !isFinite(se_pIVW)) se_pIVW = Infinity; // Handle NaN/Inf gracefully

    let pval;
    let CI_ll;
    let CI_ul;
    let CI = []; // To store full CI structure (can be disjoint for Fieller's)

    // --- Confidence Interval Calculation ---
    if (Boot_Fieller) {
        // Pass the currently active (potentially subsetted) data to BF_dist
        const current_object_for_bf = {
            betaX: Bx,
            betaY: By,
            betaXse: Bxse,
            betaYse: Byse
        };

        const z_b = BF_dist(current_object_for_bf, beta_pIVW, tau2, lambda);

        const By_sq = array_pow(By, 2);
        const Bx_sq = array_pow(Bx, 2);
        const Byse_sq = array_pow(Byse, 2);
```

```
const Bxse_sq = array_pow(Bxse, 2);
const Byse_neg4 = array_pow(Byse, -4);

const v1_terms = Byse_neg4.map((val, idx) => {
    const term1 = By_sq[idx] * Bx_sq[idx];
    const term2 = (By_sq[idx] - Byse_sq[idx] - tau2) * (Bx_sq[idx] - Bxse_sq[idx]);
    return val * (term1 - term2);
});
const v1p = array_sum(v1_terms);

let w_val;
if ((2 * mu2p - mu2) === 0) { w_val = NaN; }
else { w_val = mu2p / (2 * mu2p - mu2); }

const v2p = w_val * w_val * v2;
const v12p = w_val * v12;

const z0 = Math.pow(mu1p, 2) / v1p;
// pval = sum(z0 < z_b) / length(z_b)
pval = z_b.filter(val => z0 < val).length / z_b.length;

// Calculate confidence interval using Fieller's method
const qt_index = Math.round(z_b.length * (1 - alpha)) - 1; // -1 for 0-indexed array
const qt = z_b[Math.min(qt_index, z_b.length - 1)]; // Ensure index is within bounds

const A = Math.pow(mu2p, 2) - qt * v2p;
const B = 2 * (qt * v12p - mu1p * mu2p);
const C = Math.pow(mu1p, 2) - qt * v1p;
const D = B * B - 4 * A * C;

if (D < 0 || isNaN(D)) { // No real roots or complex, CI is (-Inf, Inf)
    CI_ll = -Infinity;
    CI_ul = Infinity;
    CI = [[-Infinity, Infinity]];
} else if (Math.abs(A) < 1e-9) { // A is effectively zero (linear case or degenerate)
    // If A is zero, it's a linear equation B*beta + C = 0.
    // If B is also zero, it's either 0=0 (all real numbers) or C=0 (no solution).
    CI_ll = -Infinity;
    CI_ul = Infinity;
    CI = [[-Infinity, Infinity]];
} else { // Standard quadratic equation
    const sqrt_D = Math.sqrt(D);
    const r1 = (-B - sqrt_D) / (2 * A);
    const r2 = (-B + sqrt_D) / (2 * A);
```

```
            if (A > 0) {
                // If A > 0, the interval is outside the roots: (-Infinity, min(r1,r2)] U [max(r1,r2), Infinity)
                // However, semantically, if A>0, the region where the quadratic is <= qt is *between* the roots.
                // The problem is often defined as f(beta) <= quantile. If A>0, the parabola opens upwards, so
                // f(beta) <= quantile holds between the roots if the roots exist.
                // For safety, assuming it wants the enclosed interval if D>=0 and A!=0.
                CI_ll = Math.min(r1, r2);
                CI_ul = Math.max(r1, r2);
                CI = [[CI_ll, CI_ul]];
            } else { // A < 0, parabola opens downwards
                // If A < 0, the interval is outside the roots: (-Infinity, min(r1,r2)] U [max(r1,r2), Infinity)
                CI_ll = [-Infinity, Math.min(r1, r2)];
                CI_ul = [Math.max(r1, r2), Infinity];
                CI = [[-Infinity, Math.min(r1, r2)], [Math.max(r1, r2), Infinity]];
            }
        }
    } else {
        // Normal approximation for p-value and CI
        // jStat.normal.cdf(q, mean, sd, lower_tail=true)
        pval = 2 * (1 - jStat.normal.cdf(Math.abs(beta_pIVW), 0, se_pIVW));

        CI_ll = beta_pIVW + jStat.normal.inv(alpha / 2, 0, se_pIVW);
        CI_ul = beta_pIVW + jStat.normal.inv(1 - alpha / 2, 0, se_pIVW);
        CI = [[CI_ll, CI_ul]];
    }

    // --- Return Result Object ---
    const result = {
        Over_dispersion: over_dispersion,
        Boot_Fieller: Boot_Fieller,
        Lambda: lambda,
        Delta: delta,
        Exposure: object.exposure || "Unknown Exposure",
        Outcome: object.outcome || "Unknown Outcome",

        Estimate: beta_pIVW,
        StdError: se_pIVW,
        CILower: (Array.isArray(CI_ll) && CI_ll.length > 1) ? CI_ll[0] : CI_ll, // Store first bound if disjoint
        CIUpper: (Array.isArray(CI_ul) && CI_ul.length > 1) ? CI_ul[1] : CI_ul, // Store second bound if disjoint
        CI: CI, // Store the full CI structure (array of arrays for disjoint)
        Alpha: alpha,

        Pvalue: pval,
        Tau2: tau2,
        SNPs: p, // Number of SNPs used in the final calculation
```

```
        Condition: eta
    };

    return result;
}
```

(10) Leave-One-Out (LOO) Method for The Inverse Variance Weighted (IVW)

The JavaScript codes for Leave-One-Out (LOO) Method for The Inverse Variance Weighted (IVW) are as follows:

```
/*
Method name and objective
Name: The Leave-One-Out (LOO) Method Method for The Inverse Variance Weighted (IVW) Mendelian Randomization.
Objective: To assess the causal effect by repeatedly omitting each SNP (instrument) to identify influential instruments, obtain country-confidence intervals for each leave-one-out estimate, and report the overall IVW estimate. It also provides a forest-plot style data structure for visualization and supports optional robust/penalized weighting and correlated instruments.

Main components and functions
ci_normal(type, estimate, se, alpha): Helper that computes normal-distribution-based confidence interval bounds (lower or upper) given an estimate, standard error, and alpha.
mr_input(betaX, betaXse, betaY, betaYse): Simple constructor that packages the four arrays into an object suitable for IVW calculation.
mr_ivw(object, options): Core IVW computation. Supports:
Models: default/random/fixed
Robust and penalized options
Weights: simple or delta
Correlation between instruments (correl) with a provided rho
Distribution for CIs: normal or t-dist
Alpha for confidence intervals
Returns the IVW estimate and SE, plus optional heterogeneity statistics (depending on settings)
mr_loo(object, alpha): Main Leave-One-Out analysis. For each SNP:
Recomputes the IVW estimate with that SNP removed
Stores the leave-one-out estimate and its confidence interval
Builds a data frame (dframe) with SNP name, estimate, lower CI, and upper CI
Computes the overall IVW result and appends it as a final row labeled “IVW estimate”
Prints a forest-plot style summary to the console and returns the dframe for plotting

Required inputs and expected outputs

Required inputs:
object: An object with arrays:
betaX: array of SNP-exposure associations
betaY: array of SNP-outcome associations
betaXse: standard errors for betaX
betaYse: standard errors for betaY
```

snps (optional): array of SNP names
alpha (optional): significance level for CIs (default 0.001 in the code)
Additional options for mr_ivw (model, robust, penalized, weights, psi, correl, distribution)

Expected outputs:
An array (dframe) of objects, each with:
snps: SNP name (or "IVW estimate" for the overall)
estimates: the leave-one-out IVW estimate for that row
CI_lower: lower bound of the confidence interval
CI_upper: upper bound of the confidence interval
The function returns this dframe, and the console also prints a textual forest plot-like summary and a JSON representation of the data for plotting.

Notes
The code relies on numeric libraries (e.g., jStat for quantiles and math.js for matrix operations).
If snp names are not provided, default names like "snp-1", "snp-2", etc., are generated.
*/

```
function ci_normal(type, estimate, se, alpha) {
      // jStat.normal.inv(p, mean, std) is the quantile function (inverse CDF)
      // For 1-alpha/2, we get the critical Z-score.
      const q = jStat.normal.inv(1 - alpha / 2, 0, 1);
      if (type === "l") {
            return estimate - q * se;
      } else if (type === "u") {
            return estimate + q * se;
      }
      return NaN; // Should not be reached
}

function mr_input(betaX, betaXse, betaY, betaYse) {
   return {
      betaX: betaX,
      betaXse: betaXse,
      betaY: betaY,
      betaYse: betaYse
   };
}


function mr_ivw(object, options = {}) {
      const {
            model: initialModel = "default",
            robust = false,
            penalized = false,
```

```
        weights = "simple",
        psi = 0,
        correl = false,
        distribution = "normal",
        alpha = 0.001
    } = options;

    const Bx = object.betaX;
    const By = object.betaY;
    const Bxse = object.betaXse;
    const Byse = object.betaYse;
    const rho = null; // Expected to be a math.js matrix or 2D array
    const nsnps = Bx.length;

    let currentModel = initialModel;
    if (currentModel === "default") {
        currentModel = (nsnps < 4) ? "fixed" : "random";
    }

    // Input validation
    const validModels = ["default", "random", "fixed"];
    const validDistributions = ["normal", "t-dist"];
    const validWeights = ["simple", "delta"];

    if (!validModels.includes(currentModel) && initialModel !== "default") {
        throw new Error(`Invalid model type: '${initialModel}'. Must be one of: ${validModels.join(', ')}.`);
    }
    if (!validDistributions.includes(distribution)) {
        throw new Error(`Invalid distribution type: '${distribution}'. Must be one of: ${validDistributions.join(', ')}.`);
    }
    if (!validWeights.includes(weights)) {
        throw new Error(`Invalid weights type: '${weights}'. Must be one of: ${validWeights.join(', ')}.`);
    }

    let thetaIVW, thetaIVWse, rse, pvalue, ciLower, ciUpper, heterStat, pvalueHeterStat, fstat;
    let effectiveRobust = robust;
    let effectivePenalized = penalized;
    let effectiveCorrelation = correl;

    // Check if correlation matrix is provided and not entirely null/NaN
    if (rho !== null && math.sum(rho) !== null && !isNaN(math.sum(rho))) {
        effectiveCorrelation = true;
    }

    if (effectiveCorrelation) {
```

```
        if (rho === null) {
            console.warn("Correlation matrix not given, but 'correl' was true or inferred. Proceeding without correlation.");
            effectiveCorrelation = false;
        } else {
            const omega_base = math.multiply(math.transpose(math.matrix(Byse)), math.matrix([Byse])); // Outer product
            const Omega_matrix = math.dotMultiply(omega_base, rho); // Element-wise multiplication with rho

            // Ensure Omega is invertible
            let Omega_inv;
            try {
                Omega_inv = math.inv(Omega_matrix);
            } catch (e) {
                throw new Error("Failed to invert Omega matrix for correlated instruments. Check correlation data for singularity or non-positive definiteness. Error: " + e.message);
            }

            const Bx_col_vec = math.transpose(math.matrix(Bx)); // (nsnps x 1)
            const Bx_row_vec = math.matrix([Bx]); // (1 x nsnps)

            // (Bx' * Omega^-1 * Bx)
            const term_denom_matrix = math.multiply(Bx_row_vec, Omega_inv, Bx_col_vec);
            const term_denom = math.subset(term_denom_matrix, math.index(0, 0)); // Extract scalar
            if (term_denom === 0 || !isFinite(term_denom)) {
                throw new Error("Singular matrix encountered in correlated IVW calculation (Bx' * Omega^-1 * Bx is zero or non-finite).");
            }
            const inv_term_denom = 1 / term_denom;

            // (Bx' * Omega^-1 * By)
            const term_num_matrix = math.multiply(Bx_row_vec, Omega_inv, math.transpose(math.matrix(By)));
            const term_num = math.subset(term_num_matrix, math.index(0, 0)); // Extract scalar

            thetaIVW = inv_term_denom * term_num;

            const residuals = By.map((y, i) => y - thetaIVW * Bx[i]);
            const residuals_col_vec = math.transpose(math.matrix(residuals));

            let rse_calc_term_matrix;
            if (nsnps > 1) {
                rse_calc_term_matrix = math.multiply(math.transpose(residuals_col_vec), Omega_inv, residuals_col_vec);
                rse = Math.sqrt(math.subset(rse_calc_term_matrix, math.index(0, 0)) / (nsnps - 1));
            } else {
                rse = NaN; // Cannot estimate RSE with 1 SNP
            }
```

```
            if (currentModel === "random") {
                thetaIVWse = Math.sqrt(inv_term_denom) * Math.max(rse, 1);
            } else if (currentModel === "fixed") {
                thetaIVWse = Math.sqrt(inv_term_denom);
            }

            effectiveRobust = false;
            effectivePenalized = false;

            if (nsnps > 1 && rse_calc_term_matrix) {
                heterStat = (nsnps - 1) * (math.subset(rse_calc_term_matrix, math.index(0, 0)) / (nsnps - 1));
                pvalueHeterStat = 1 - jStat.chisquare.cdf(heterStat, nsnps - 1);
            } else {
                heterStat = NaN;
                pvalueHeterStat = NaN;
            }

            try {
                const Bx_div_Bxse = Bx.map((b, i) => b / Bxse[i]);
                const chol_rho = math.cholesky(rho); // Cholesky decomposition of correlation matrix
                const inv_chol_rho = math.inv(chol_rho);

                // Math.js handles `multiply(vector, matrix)` as `1xN * NxN = 1xN`
                const transformed_Bx_Bxse_row = math.multiply(math.matrix([Bx_div_Bxse]), inv_chol_rho);
                // Convert back to simple array to map for sum of squares
                const transformed_Bx_Bxse_array = transformed_Bx_Bxse_row.toArray()[0]; // Get the 1D array from the 1xN matrix
                const sum_sq_transformed = transformed_Bx_Bxse_array.reduce((sum, val) => sum + val * val, 0);
                fstat = sum_sq_transformed / nsnps;
            } catch (e) {
                console.warn(`Could not calculate F-stat with correlation (e.g., non-positive definite rho or other matrix issue): ${e.message}`);
                fstat = NaN;
            }

        } // End of correlated processing
    } else { // No correlation
        if (nsnps === 0) {
            throw new Error("No SNPs detected for IVW analysis.");
        } else if (nsnps === 1) {
            thetaIVW = By[0] / Bx[0];
            if (weights === "simple") {
                thetaIVWse = Math.abs(Byse[0] / Bx[0]);
            } else if (weights === "delta") {
```

```
                thetaIVWse = Math.sqrt(
                    (Byse[0] * Byse[0]) / (Bx[0] * Bx[0]) +
                    (By[0] * By[0] * Bxse[0] * Bxse[0]) / (Bx[0] * Bx[0] * Bx[0] * Bx[0]) -
                    (2 * psi * By[0] * Bxse[0] * Byse[0]) / (Bx[0] * Bx[0] * Bx[0])
                );
            }
            rse = 1;
            heterStat = NaN;
            pvalueHeterStat = NaN;
            fstat = (Bx[0] / Bxse[0]) * (Bx[0] / Bxse[0]);

        } else { // nsnps > 1, no correlation
            let currentWeights;
            if (robust || penalized) {
                console.warn("Robust regression (lmrob equivalent) is not fully implemented. Falling back to weighted least squares.");

                if (penalized) { // If penalized, specific r.weights are used
                    if (weights === "simple") {
                        const penWeights = penalisedWeights(Bx, Bxse, By, Byse);
                        currentWeights = rWeights(Byse, penWeights);
                    } else if (weights === "delta") {
                        const penWeights = penalisedWeightsDelta(Bx, Bxse, By, Byse, psi);
                        currentWeights = rWeightsDelta(Bx, Bxse, By, Byse, psi, penWeights);
                    }
                } else { // If robust but not penalized, use inverse variance (simple) or delta inverse variance
                    if (weights === "simple") {
                        currentWeights = Byse.map(se => 1 / (se * se));
                    } else if (weights === "delta") {
                        currentWeights = Bx.map((x, i) => {
                            const denom = (Byse[i] * Byse[i]) +
                                (By[i] * By[i] * Bxse[i] * Bxse[i]) / (x * x) -
                                (2 * psi * By[i] * Bxse[i] * Byse[i]) / x;
                            return (denom === 0 || isNaN(denom) || !isFinite(denom)) ? 0 : 1 / denom;
                        });
                    }
                }
            } else { // Not robust or penalized
                if (weights === "simple") {
                    currentWeights = Byse.map(se => 1 / (se * se));
                } else if (weights === "delta") {
                    currentWeights = Bx.map((x, i) => {
                        const denom = (Byse[i] * Byse[i]) +
                            (By[i] * By[i] * Bxse[i] * Bxse[i]) / (x * x) -
                            (2 * psi * By[i] * Bxse[i] * Byse[i]) / x;
```

```
                return (denom === 0 || isNaN(denom) || !isFinite(denom)) ? 0 : 1 / denom;
            });
        }
    }

    // --- Weighted Least Squares Calculation ---
    // Model: By ~ Bx - 1 (regression through origin)
    // beta_hat = (X' W X)^-1 X' W Y
    // Here X is the Bx vector. X'WX is sum(w_i * Bx_i^2). X'WY is sum(w_i * Bx_i * By_i).

    const weightedBxSumSq = Bx.reduce((sum, x, i) => sum + currentWeights[i] * x * x, 0);
    const weightedBxBySum = Bx.reduce((sum, x, i) => sum + currentWeights[i] * x * By[i], 0);

    if (weightedBxSumSq === 0 || !isFinite(weightedBxSumSq)) {
        throw new Error("Weighted sum of squares of Bx is zero or non-finite. Cannot perform regression.");
    }
    thetaIVW = weightedBxBySum / weightedBxSumSq;

    // Calculate residuals and residual sum of squares
    const residuals = By.map((y, i) => y - thetaIVW * Bx[i]);
    const weightedResidualSumSq = residuals.reduce((sum, r, i) => sum + currentWeights[i] * r * r, 0);

    let sigma_hat_squared;
    if (nsnps > 1) { // df = nsnps - 1 (for 1 estimated parameter: slope)
        sigma_hat_squared = weightedResidualSumSq / (nsnps - 1);
    } else {
        sigma_hat_squared = NaN; // Cannot estimate sigma with 1 SNP
    }

    rse = Math.sqrt(sigma_hat_squared);

    // Standard error of the coefficient (thetaIVWse)
    // se(beta_hat) = sqrt( (X' W X)^-1 * sigma_hat^2 )
    const base_thetaIVWse = Math.sqrt((1 / weightedBxSumSq) * sigma_hat_squared);

    if (currentModel === "random") {
        thetaIVWse = base_thetaIVWse / Math.min(rse, 1);
    } else { // fixed model
        thetaIVWse = base_thetaIVWse / rse;
    }

    if (robust || penalized) {
        heterStat = NaN;
        pvalueHeterStat = NaN;
    } else {
```

```
                    // For non-robust/non-penalized, calculate heterogeneity stats (Q statistic)
                    if (nsnps > 1) {
                        heterStat = (nsnps - 1) * (rse * rse); // Equivalent to weightedResidualSumSq
                        pvalueHeterStat = 1 - jStat.chisquare.cdf(heterStat, nsnps - 1);
                    } else {
                        heterStat = NaN;
                        pvalueHeterStat = NaN;
                    }
                }

                // F-statistic for multiple SNPs without correlation
                fstat = Bx.reduce((sum, x, i) => sum + (x / Bxse[i]) * (x / Bxse[i]), 0) / nsnps;
            } // End of nsnps > 1 (no correlation)
        } // End of no correlation block

        // Calculate p-value and confidence intervals based on chosen distribution
        if (distribution === "normal") {
            pvalue = 2 * jStat.normal.cdf(-Math.abs(thetaIVW / thetaIVWse), 0, 1);
            ciLower = ci_normal("l", thetaIVW, thetaIVWse, alpha);
            ciUpper = ci_normal("u", thetaIVW, thetaIVWse, alpha);
        } else if (distribution === "t-dist") {
            const df = Math.max(1, nsnps - 1); // Ensure df is at least 1
            pvalue = 2 * jStat.studentt.cdf(-Math.abs(thetaIVW / thetaIVWse), df);
            ciLower = ci_t("l", thetaIVW, thetaIVWse, df, alpha);
            ciUpper = ci_t("u", thetaIVW, thetaIVWse, df, alpha);
        }

        return {
            Estimate: thetaIVW,
            CILower: ciLower,
            CIUpper: ciUpper,
            PValue: pvalue
        };
}

/**
 * Performs a Leave-One-Out (LOO) analysis for Mendelian Randomization.
 *
 * @param {object} object - An object of type MRInput containing genetic association data.
 *     Expected properties: betaX, betaY, betaXse, betaYse, snps.
 * @param {number} [alpha=0.001] - The significance level for confidence intervals.
 * @returns {Array<object>} An array of objects, where each object represents a LOO estimate
 *     and includes the SNP name, estimate, lower CI, and upper CI.
 *     Also prints a forest plot to the console (or renders it in an environment that supports it).
 */
```

```
function mr_loo(object, alpha = 0.001) {
  const Bx = object.betaX;
  const By = object.betaY;
  const Bxse = object.betaXse;
  const Byse = object.betaYse;
  let snps = object.snps;

  const n = Bx.length;

  // In case of named variants, below code
  if (snps && snps.includes("snp") && !snps.every((snp, index) => snp === `snp-${index + 1}`)) {
    // A more robust check might be needed depending on how 'snp' is used.
    // This assumes if 'snp' is in the array, it might be a placeholder for default naming.
    snps = Array.from({ length: n }, (_, i) => `snp-${i + 1}`);
  } else if (!snps || snps.length !== n) {
    // If snps array is missing or doesn't match the length, generate default names.
    snps = Array.from({ length: n }, (_, i) => `snp-${i + 1}`);
  }

  const estimates = [];
  const CI_lower = [];
  const CI_upper = [];

  for (let j = 0; j < n; j++) {
    // Create new arrays excluding the j-th SNP
    const Bx_loo = Bx.filter((_, index) => index !== j);
    const By_loo = By.filter((_, index) => index !== j);
    const Bxse_loo = Bxse.filter((_, index) => index !== j);
    const Byse_loo = Byse.filter((_, index) => index !== j);

    const mrInput_loo = mr_input(Bx_loo, Bxse_loo, By_loo, Byse_loo);
    const ivw_result_loo = mr_ivw(mrInput_loo, alpha);

    estimates.push(ivw_result_loo.Estimate);
    CI_lower.push(ivw_result_loo.CILower);
    CI_upper.push(ivw_result_loo.CIUpper);
  }

  const dframe = [];
  for (let i = 0; i < n; i++) {
    dframe.push({
      snps: snps[i],
      estimates: estimates[i],
      CI_lower: CI_lower[i],
      CI_upper: CI_upper[i]
```

```
    });
}

// Calculate overall IVW estimate
const mrInput_ivw = mr_input(Bx, Bxse, By, Byse);
const ivw_output = mr_ivw(mrInput_ivw, alpha);
const ivw_estimate = ivw_output.Estimate;
const ivw_lower = ivw_output.CILower;
const ivw_upper = ivw_output.CIUpper;

const ivw_row = {
    snps: "IVW estimate",
    estimates: ivw_estimate,
    CI_lower: ivw_lower,
    CI_upper: ivw_upper
};

dframe.push(ivw_row);

// Sort dframe for plotting (reversing order of SNPs for typical forest plot)
// We'll create the order first, then sort the array.
const snpOrder = [...snps, "IVW estimate"].reverse();
dframe.sort((a, b) => snpOrder.indexOf(a.snps) - snpOrder.indexOf(b.snps));

const Interval_type = `${(1 - alpha) * 100}% CI)`;
const Y_label = `Leave-one-out causal estimate (${Interval_type}`;

// --- Forest Plot Generation (Conceptual) ---
// In JavaScript, creating complex plots usually requires a plotting library
// like Chart.js, Plotly.js, or D3.js.
// This section provides a conceptual representation of how you *might*
// generate a forest plot and print the data.

console.log("Leave-One-Out Analysis Results:");
console.log("---------------------------------");
console.log("SNP | Estimate | CI Lower | CI Upper");
console.log("---------------------------------");
dframe.forEach(row => {
    console.log(`${row.snps.padEnd(10)} | ${row.estimates.toFixed(4).padEnd(10)} | ${row.CI_lower.toFixed(4).padEnd(10)}
| ${row.CI_upper.toFixed(4)}`);
});
console.log("---------------------------------");

console.log("\nConceptual Forest Plot Data (for visualization with a plotting library):");
console.log(JSON.stringify(dframe, null, 2));
```

```
    console.log(`\nX-axis label: ${Y_label}`);

    // Here's a hypothetical example using a library that might accept data in this format:

    /*
    // Example using a hypothetical plotting library (not real code):
    const plotData = dframe.map(row => ({
        y: row.snps,
        x: row.estimates,
        xmin: row.CI_lower,
        xmax: row.CI_upper,
        isIvw: row.snps === "IVW estimate"
    }));

    // Assuming a function 'renderForestPlot' exists that takes data and labels
    renderForestPlot({
        data: plotData,
        yLabel: "Variants",
        xLabel: Y_label,
        highlightIvw: true, // Indicator for styling the IVW point
        zeroLine: true
    });
    */

    return dframe; // Return the data frame
}
```

(11) The Maximum Likelihood Method (MaxLik)

The JavaScript codes for The Maximum Likelihood Method (MaxLik) are as follows:

```
/*
Name: The Maximum Likelihood Method (MaxLik) for Mendelian Randomization.
Objective: To estimate the causal effect (theta) by maximizing the likelihood of the observed SNP-exposure and SNP-outcome associations, allowing for either uncorrelated or correlated instruments, and offering fixed/random-effect modeling with normal or t-distributed confidence intervals.

Main components and functions
loglikelihood(param, x, sigmax, y, sigmay): Log-likelihood for uncorrelated variants (two-sample, no overlap). param contains latent exposure effects and theta.
loglikelihoodcorrel(param, x, Taux, y, Tauy): Log-likelihood for correlated variants (separate precision matrices for X and Y).
loglikelihoodrhocorrel(param, x, y, Tauxy): Log-likelihood with correlation from sample overlap.
invertMatrix(mat): Utility to invert a matrix (via math.js).
numericHessian(func, x0, eps): Numerical Hessian approximation (central differences).
nelderMead(func, x0, options): Nelder–Mead optimizer to minimize the objective function.
ci_normal(side, estimate, se, alpha) / ci_t(side, estimate, se, df, alpha): CI helpers using normal or t-distribution quantiles (via
```

jStat).

mr_maxlik(object, args): Core function. Builds the appropriate likelihood, handles uncorrelated/correlated/overlap cases, initializes parameters, runs Nelder–Mead optimization, computes Hessian and standard errors (with fixed vs. random-effects scaling), constructs CIs, p-values, and a result object with rich metadata.

Required inputs and expected outputs

Required inputs (via object)

betaX: array of SNP-exposure associations

betaY: array of SNP-outcome associations

betaXse: standard errors for betaX

betaYse: standard errors for betaY

correlation: optional matrix (as 2D array or math.js matrix) describing SNP correlations

exposure: name of exposure

outcome: name of outcome

Optional/mode inputs (via args)

model: "default", "fixed", or "random"

correl: boolean override for treating covariance as correlated

psi: overlap correlation parameter (0 means no overlap)

distribution: "normal" or "t-dist" for CIs

alpha: significance level for CIs

options: optimization options (maxIter, tol)

Expected outputs

An object with fields including:

Model: chosen model ("default"/"fixed"/"random")

Exposure, Outcome, Correlation, Psi

Estimate: theta estimate

StdError: standard error of theta

CILower, CIUpper: confidence interval bounds

SNPs: number of SNPs

Pvalue: p-value for theta

Alpha: significance level

RSE: relative standard error

Heter: heterogeneity statistics (and p-value)

optimization: convergence status, iterations, objective value, and parameter estimates

hessian, inverseHessian: diagnostic numerical derivatives

Notes

The method supports both uncorrelated and correlated instrument scenarios, including a special case for overlapping samples (rhocorrel).

If the Hessian is not positive definite, a small ridge is added as a fallback.

The output is designed to be rich enough for downstream reporting and plotting (e.g., forest plots of SNP-wise vs. overall estimates).

*/

```
// Utility: deep clone array
function cloneArray(a) {
  return JSON.parse(JSON.stringify(a));
}

// Log-likelihood for uncorrelated variants (two-sample, no overlap)
// param: array length = nsnps + 1: first nsnps = latent true betas for exposure, last = theta
// x: array of observed betaX
// sigmax: array of standard errors for betaX
// y: array of observed betaY
// sigmay: array of standard errors for betaY
function loglikelihood(param, x, sigmax, y, sigmay) {
  const nsnps = x.length;
  const betas = param.slice(0, nsnps);
  const theta = param[nsnps];

  let sum1 = 0;
  for (let i = 0; i < nsnps; i++) {
    const d = x[i] - betas[i];
    sum1 += (d * d) / (sigmax[i] * sigmax[i]);
  }
  let sum2 = 0;
  for (let i = 0; i < nsnps; i++) {
    const d = y[i] - theta * betas[i];
    sum2 += (d * d) / (sigmay[i] * sigmay[i]);
  }
  return 0.5 * (sum1 + sum2);
}

// Log-likelihood for correlated variants (two-sample, no overlap)
// param: as above
// x: array of observed betaX
// Taux: precision matrix (math.js matrix) for x (inverse of covariance)
// y: array of observed betaY
// Tauy: precision matrix for y
function loglikelihoodcorrel(param, x, Taux, y, Tauy) {
  const nsnps = x.length;
  const betas = param.slice(0, nsnps);
  const theta = param[nsnps];

  // compute dx and dy as plain arrays
  const dxArr = new Array(nsnps);
  const dyArr = new Array(nsnps);
  for (let i = 0; i < nsnps; i++) {
```

```
        dxArr[i] = x[i] - betas[i];
        dyArr[i] = y[i] - theta * betas[i];
    }

    // convert to explicit column vectors (nsnps x 1) so math.multiply behaves predictably
    const dxCol = math.matrix(math.reshape(dxArr, [nsnps, 1]));
    const dyCol = math.matrix(math.reshape(dyArr, [nsnps, 1]));

    // compute dx^T * Taux * dx
    const leftDx = math.multiply(math.transpose(dxCol), Taux); // 1 x nsnps
    const v1Mat = math.multiply(leftDx, dxCol); // 1 x 1 matrix

    // compute dy^T * Tauy * dy
    const leftDy = math.multiply(math.transpose(dyCol), Tauy);
    const v2Mat = math.multiply(leftDy, dyCol);

    // extract scalar safely using math.subset or valueOf
    const v1 = (typeof v1Mat.valueOf === 'function') ? v1Mat.valueOf() : v1Mat;
    const v2 = (typeof v2Mat.valueOf === 'function') ? v2Mat.valueOf() : v2Mat;

    // v1 and v2 may be [[num]] or a math.matrix — handle both
    const scalar1 = Array.isArray(v1) ? v1[0][0] : (v1._data ? v1._data[0][0] : v1);
    const scalar2 = Array.isArray(v2) ? v2[0][0] : (v2._data ? v2._data[0][0] : v2);

    return 0.5 * (scalar1 + scalar2);
}

// Log-likelihood with correlation from sample overlap
// param: as above
// x: array of observed betaX
// y: array of observed betaY
// Tauxy: precision matrix (math.js) for the stacked vector [x; y]
function loglikelihoodrhocorrel(param, x, y, Tauxy) {
    const nsnps = x.length;
    const betas = param.slice(0, nsnps);
    const theta = param[nsnps];

    // Build dx and dy as plain JS arrays
    const dxArr = new Array(nsnps);
    const dyArr = new Array(nsnps);
    for (let i = 0; i < nsnps; i++) {
        dxArr[i] = x[i] - betas[i];
        dyArr[i] = y[i] - theta * betas[i];
    }
```

```
    // Stack dx and dy into a column vector of length 2*nsnps
    const stackedArr = dxArr.concat(dyArr);
    // Ensure explicit column vector (2n x 1)
    const sm = math.matrix(math.reshape(stackedArr, [stackedArr.length, 1]));

    // Compute scalar: 0.5 * sm^T * Tauxy * sm
    const left = math.multiply(math.transpose(sm), Tauxy); // 1 x (2n)
    const prod = math.multiply(left, sm); // 1 x 1

    // Safely extract the scalar value regardless of internal representation
    let scalar;
    if (prod && typeof prod.valueOf === 'function') {
        const val = prod.valueOf(); // could be [[num]] or number
        if (Array.isArray(val)) {
            // val might be [[num]]
            if (Array.isArray(val[0])) scalar = val[0][0];
            else scalar = val[0];
        } else {
            scalar = val;
        }
    } else if (prod && prod._data) {
        // older internal representation
        const d = prod._data;
        if (Array.isArray(d[0])) scalar = d[0][0]; else scalar = d[0];
    } else {
        // fallback: try subset
        try {
            scalar = math.subset(prod, math.index(0, 0));
        } catch (e) {
            // ultimate fallback: convert to number
            scalar = Number(prod);
        }
    }

    return 0.5 * scalar;
}

// Inverse of a matrix using math.js
function invertMatrix(mat) {
    return math.inv(mat);
}

// Numerical Hessian approximation (central finite differences)
// func: function taking param array and returning scalar
// x0: parameter vector (array)
```

```
// eps: step size (optional)
function numericHessian(func, x0, eps) {
    const n = x0.length;
    const hessian = Array.from({ length: n }, () => Array(n).fill(0));
    const fx0 = func(x0);
    const step = eps || Math.pow(Number.EPSILON, 1 / 3) * Math.max(1, math.norm(x0));
    for (let i = 0; i < n; i++) {
        for (let j = i; j < n; j++) {
            const xi = cloneArray(x0);
            const xj = cloneArray(x0);
            const xij = cloneArray(x0);
            xi[i] += step;
            xj[j] += step;
            xij[i] += step;
            xij[j] += step;
            const f1 = func(xij);
            const f2 = func(xi);
            const f3 = func(xj);
            const f4 = fx0;
            // Mixed partial using central difference formula:
            const value = (f1 - f2 - f3 + f4) / (step * step);
            hessian[i][j] = value;
            hessian[j][i] = value;
        }
    }
    return hessian;
}

// Nelder-Mead optimizer (minimization)
// Returns object {x: bestParams, fx: bestValue, iterations}
// Implementation adapted for clarity and compactness
function nelderMead(func, x0, options) {
    options = options || {};
    const maxIter = options.maxIter || 10000;
    const tol = options.tol || 1e-8;
    const alpha = options.alpha || 1;
    const gamma = options.gamma || 2;
    const rho = options.rho || 0.5;
    const sigma = options.sigma || 0.5;

    const n = x0.length;
    // initial simplex: x0 and x0 + e_i where e_i scaled
    const simplex = [];
    simplex.push({ x: cloneArray(x0), fx: func(cloneArray(x0)) });
    for (let i = 0; i < n; i++) {
```

```
      const xi = cloneArray(x0);
      xi[i] = xi[i] !== 0 ? xi[i] * (1 + 0.05) : 0.00025;
      simplex.push({ x: xi, fx: func(cloneArray(xi)) });
    }

    let iter = 0;
    while (iter < maxIter) {
      simplex.sort((a, b) => a.fx - b.fx);
      const best = simplex[0];
      const worst = simplex[n];

// convergence check: standard deviation of fx
const fvals = simplex.map(s => s.fx);
const meanF = fvals.reduce((s, v) => s + v, 0) / fvals.length;
const sd = Math.sqrt(fvals.reduce((s, v) => s + (v - meanF) * (v - meanF), 0) / fvals.length);
if (sd < tol) break;

// centroid of all but worst
const centroid = Array(n).fill(0);
for (let i = 0; i < n; i++) {
  for (let j = 0; j < n; j++) {
    centroid[j] += simplex[i].x[j];
  }
}
for (let j = 0; j < n; j++) centroid[j] /= n;

// reflection
const xr = centroid.map((c, j) => c + alpha * (c - worst.x[j]));
const fxr = func(xr);
if (fxr < simplex[0].fx) {
  // expansion
  const xe = centroid.map((c, j) => c + gamma * (xr[j] - c));
  const fxe = func(xe);
  if (fxe < fxr) {
    simplex[n] = { x: xe, fx: fxe };
  } else {
    simplex[n] = { x: xr, fx: fxr };
  }
} else if (fxr < simplex[n - 1].fx) {
  simplex[n] = { x: xr, fx: fxr };
} else {
  // contraction
  let xc, fxc;
  if (fxr < worst.fx) {
    // outside contraction
```

```
        xc = centroid.map((c, j) => c + rho * (xr[j] - c));
        fxc = func(xc);
    } else {
        // inside contraction
        xc = centroid.map((c, j) => c + rho * (worst.x[j] - c));
        fxc = func(xc);
    }
    if (fxc < worst.fx) {
        simplex[n] = { x: xc, fx: fxc };
    } else {
        // shrink
        for (let i = 1; i < simplex.length; i++) {
            simplex[i].x = simplex[0].x.map((b, j) => b + sigma * (simplex[i].x[j] - b));
            simplex[i].fx = func(simplex[i].x);
        }
    }
}
iter++;

    }

    simplex.sort((a, b) => a.fx - b.fx);
    return { x: simplex[0].x, fx: simplex[0].fx, iterations: iter };
}

// CI helpers using jStat
function ci_normal(side, estimate, se, alpha) {
    // side: "l" or "u"
    const z = Math.abs(jStat.normal.inv(alpha / 2, 0, 1));
    if (side === "l") return estimate - z * se;
    return estimate + z * se;
}
function ci_t(side, estimate, se, df, alpha) {
    const tcrit = Math.abs(jStat.studentt.inv(alpha / 2, df));
    if (side === "l") return estimate - tcrit * se;
    return estimate + tcrit * se;
}

// Main function: mr_maxlik
// object must have fields: betaX (array), betaY (array), betaXse (array), betaYse (array), correlation (matrix as 2D array or null),
exposure, outcome
// model: "default", "fixed", or "random"
// correl: boolean override whether to treat correlated variant covariance provided
// psi: scalar for overlap correlation between X and Y (0 means none)
// distribution: "normal" or "t-dist"
```

```
// alpha: significance level for CIs
// options: optional control parameters for optimizer: {maxIter, tol}
function mr_maxlik(object, args) {
  args = args || {};
  let model = args.model || "default";
  let correl = args.correl || false;
  const psi = (args.psi === undefined ? 0 : args.psi);
  const distribution = args.distribution || "normal";
  const alpha = (args.alpha === undefined ? 0.001 : args.alpha);
  const options = args.options || {};

  // Extract
  const Bx = object.betaX.slice();
  const By = object.betaY.slice();
  const Bxse = object.betaXse.slice();
  const Byse = object.betaYse.slice();
  const rho = object.correlation || null;
  const nsnps = Bx.length;

  if (model === "default") {
    if (nsnps < 4) model = "fixed"; else model = "random";
  }

  if (rho !== null) correl = true;

  if (!(["random", "fixed"].includes(model) && ["normal", "t-dist"].includes(distribution))) {
    throw new Error("Model must be one of: default, random, fixed. Distribution must be one of: normal, t-dist.");
  }

  // Starting param vector: c(Bx, theta0) where theta0 = sum(Bx*By/Byse^2)/sum(Bx^2/Byse^2)
  let denom = 0;
  let numer = 0;
  for (let i = 0; i < nsnps; i++) {
    const w = 1 / (Byse[i] * Byse[i]);
    numer += Bx[i] * By[i] * w;
    denom += Bx[i] * Bx[i] * w;
  }
  const theta0 = denom === 0 ? 0 : numer / denom;
  const start = Bx.concat([theta0]);

  let objectiveFunction;
  let usedHessianApproximation = true; // we'll compute numeric Hessian after optimization

  if (psi === 0) {
    if (!correl) {
```

```
        // uncorrelated
        objectiveFunction = function (param) {
            return loglikelihood(param, Bx, Bxse, By, Byse);
        };
    } else {
        // correlated with separate precision matrices for x and y
        // Build covariance matrices: Sigmax = Bxse %o% Bxse * rho
        // Here rho can be provided as 2D array; ensure it's matrix
        const R = math.matrix(rho);
        // Build Sigmax and Sigmay as nsnps x nsnps
        const Sigmax = math.zeros(nsnps, nsnps);
        const Sigmay = math.zeros(nsnps, nsnps);
        for (let i = 0; i < nsnps; i++) {
            for (let j = 0; j < nsnps; j++) {
                const r = R._data[i][j];
                Sigmax._data[i][j] = Bxse[i] * Bxse[j] * r;
                Sigmay._data[i][j] = Byse[i] * Byse[j] * r;
            }
        }
        const Taux = invertMatrix(Sigmax);
        const Tauy = invertMatrix(Sigmay);
        objectiveFunction = function (param) {
            return loglikelihoodcorrel(param, Bx, Taux, By, Tauy);
        };
    }
} else {
    // psi != 0: overlap correlation -> build full covariance for stacked vector
    // If no rho provided, assume identity
    const R = correl ? math.matrix(rho) : math.identity(nsnps);
    // Sigmax, Sigmay, Sigmaxy
    const Sigmax = math.zeros(nsnps, nsnps);
    const Sigmay = math.zeros(nsnps, nsnps);
    const Sigmaxy = math.zeros(nsnps, nsnps);
    for (let i = 0; i < nsnps; i++) {
        for (let j = 0; j < nsnps; j++) {
            const r = R._data ? R._data[i][j] : R[i][j];
            Sigmax._data[i][j] = Bxse[i] * Bxse[j] * r;
            Sigmay._data[i][j] = Byse[i] * Byse[j] * r;
            Sigmaxy._data[i][j] = Bxse[i] * Byse[j] * r * psi;
        }
    }
    // Build block matrix
    // [Sigmax, Sigmaxy; Sigmaxy, Sigmay]
    const top = [];
    for (let i = 0; i < nsnps; i++) {
```

```
        top.push(Sigmax._data[i].concat(Sigmaxy._data[i]));
    }
    const bottom = [];
    for (let i = 0; i < nsnps; i++) {
        bottom.push(Sigmaxy._data[i].concat(Sigmay._data[i]));
    }
    const Sigmaxyall = math.matrix(top.concat(bottom));
    const Tauxy = invertMatrix(Sigmaxyall);
    objectiveFunction = function (param) {
        return loglikelihoodrhocorrel(param, Bx, By, Tauxy);
    };
}

// Minimization using Nelder-Mead
const nmOpts = { maxIter: options.maxIter || 25000, tol: options.tol || 1e-10 };
const opt = nelderMead(objectiveFunction, start, nmOpts);

const optPar = opt.x;
const optValue = opt.fx;

// Numerical Hessian at optimum
const hess = numericHessian(objectiveFunction, optPar);
// invert Hessian
// convert to math matrix and invert; handle potential non-positive-definite by pseudo-inverse
let hessMat = math.matrix(hess);
let invHess;
try {
    invHess = invertMatrix(hessMat);
} catch (e) {
    // fallback: add small ridge to diagonal and invert
    const eps = 1e-8;
    const h2 = hess.map((row, i) => row.map((v, j) => v + (i === j ? eps : 0)));
    invHess = invertMatrix(math.matrix(h2));
}

const thetaML = optPar[nsnps];
// Standard error
let thetaMLse;
const invHessData = invHess._data || invHess;
const varTheta = invHessData[nsnps][nsnps];
if (model === "fixed") {
    thetaMLse = Math.sqrt(Math.abs(varTheta));
} else {
    // random: inflate by max(2*opt$value/(nsnps-1), 1)
    const scale = Math.max(2 * optValue / (nsnps - 1), 1);
```

```
    thetaMLse = Math.sqrt(Math.abs(varTheta) * scale);
  }

  const rse = Math.sqrt(2 * optValue / (nsnps - 1));

  // CIs
  let ciLower, ciUpper;
  if (distribution === "normal") {
    ciLower = ci_normal("l", thetaML, thetaMLse, alpha);
    ciUpper = ci_normal("u", thetaML, thetaMLse, alpha);
  } else {
    ciLower = ci_t("l", thetaML, thetaMLse, nsnps - 1, alpha);
    ciUpper = ci_t("u", thetaML, thetaMLse, nsnps - 1, alpha);
  }

  const heterStat = 2 * optValue;
  const pvalueHeter = 1 - jStat.chisquare.cdf(heterStat, nsnps - 1);

  // p-value for estimate
  let pvalue;
  if (distribution === "normal") {
    pvalue = 2 * (1 - jStat.normal.cdf(Math.abs(thetaML / thetaMLse), 0, 1));
  } else {
    pvalue = 2 * (1 - jStat.studentt.cdf(Math.abs(thetaML / thetaMLse), nsnps - 1));
  }

  // Construct result object
  const result = {
    Model: model,
    Exposure: object.exposure || null,
    Outcome: object.outcome || null,
    Correlation: object.correlation || null,
    Psi: psi,
    Estimate: thetaML,
    StdError: thetaMLse,
    CILower: ciLower,
    CIUpper: ciUpper,
    SNPs: nsnps,
    Pvalue: pvalue,
    Alpha: alpha,
    RSE: rse,
    Heter: {
      Stat: heterStat,
      Pvalue: pvalueHeter
    },
```

```
    // internal info
    optimization: {
      converged: opt.iterations < (options.maxIter || 25000),
      iterations: opt.iterations,
      objectiveValue: optValue,
      parameters: optPar
    },
    hessian: hess,
    inverseHessian: invHessData
  };

  return result;
}
```

(12) The Egger Method (Egger)

The JavaScript codes for The Egger Method (Egger) are as follows:

```
/*
Method name and objective
Name: Egger Method (Egger) for Mendelian Randomization.
Objective: To estimate a causal effect using Egger regression while also testing for directional pleiotropy via an intercept term. The implementation supports both uncorrelated and correlated instruments, and offers options for robust or penalized fitting, as well as normal or t-distribution based confidence intervals.

Main components and functions
Utility helpers: isNumber, sum, mean, clone, zeros, and edge-of-library helpers for normal and t-distribution quantiles (qnorm, pnorm, qt, pt, qchisq, pchisq).
Matrix and regression helpers: weightedMean, matrix operations (matMul, matTranspose, matInv, matDiag) and vecDot.
Confidence interval helper: eggerBounds(type, dist, theta, thetaSe, df, rse, alpha) to compute lower/upper bounds with either normal or t-distribution assumptions.
Core modeling helpers:
weightedLeastSquares(X, y, w): standard weighted least squares.
huberIRLS(X, y, w_init, ...): robust (Huber) IRLS for robust regression.
Main method: mr_egger(object, options)
Supports correlation matrices (adjusts design and omega accordingly) or uncorrelated instruments (uses Byse-based weights).
Options include robust, penalized, correl, distribution (normal or t-dist), and alpha.
Outputs a result object with estimates for the slope (causal effect) and intercept, their standard errors and CIs, pleiotropy and causal p-values, model metadata, and heterogeneity statistics (Q, I-squared).

Required inputs and expected outputs

Required inputs:
object: {
betaX, betaY, betaXse, betaYse, correlation (optional), exposure, outcome, snps
}
```

```
options: {
robust (boolean), penalized (boolean), correl (boolean), distribution ("normal" or "t-dist"), alpha (significance level)
}

Expected outputs:
An object describing the fitted model, e.g.:
Model, Exposure, Outcome, Robust, Penalized, Correlation
Estimate (causal effect), StdError_Est, CILower_Est, CIUpper_Est, Pvalue_Est
Intercept (pleiotropy estimate), StdError_Int, CILower_Int, CIUpper_Int, Pvalue_Int
Pleio_pval, Causal_pval, Alpha, SNPs
RSE (residual standard error), Heter_Stat (statistic and p-value), I_sq
The function validates input (e.g., requires at least 3 SNPs) and returns the model results suitable for reporting or plotting (e.g., forest plots, the Egger regression line).
*/

// ---------- Utility helpers ----------

function isNumber(x) { return typeof x === "number" && isFinite(x); }
function sum(arr) { return arr.reduce((a, b) => a + b, 0); }
function mean(arr) { return arr.length ? sum(arr) / arr.length : NaN; }
function clone(a) { return a.slice(); }
function zeros(n) { return Array(n).fill(0); }

function qnorm(p) { return jStat.normal.inv(p, 0, 1); }
function pnorm(x) { return jStat.normal.cdf(x, 0, 1); }
function qt(p, df) { return jStat.studentt.inv(p, df); }
function pt(x, df) { return jStat.studentt.cdf(x, df); }
function qchisq(p, df) { return jStat.chisquare.inv(p, df); }
function pchisq(x, df) { return jStat.chisquare.cdf(x, df); }

// Weighted mean
function weightedMean(x, w) {
  var num = 0, den = 0;
  for (var i = 0; i < x.length; i++) { num += x[i] * w[i]; den += w[i]; }
  return den === 0 ? NaN : num / den;
}

// Matrix helpers via math.js for convenience
function matMul(A, B) { return math.multiply(A, B); }
function matTranspose(A) { return math.transpose(A); }
function matInv(A) { return math.inv(A); }
function matDiag(v) { return math.diag(v); }
function vecDot(a, b) { var s = 0; for (var i = 0; i < a.length; i++) s += a[i] * b[i]; return s; }

// ---------- egger.bounds ----------
```

```
//
// type: "l" or "u"
// dist: "normal" or "t-dist"
// .theta: estimate
// .thetase: standard error
// df: degrees of freedom (for t)
// .rse: residual standard error (RSE)
// .alpha: significance level
//
function eggerBounds(type, dist, theta, thetaSe, df = 0, rse = 1, alpha = 0.001) {
   var x = 1 - alpha / 2;
   if (type !== "l" && type !== "u") return NaN;

   if (dist === "normal") {
      var z = qnorm(x);
      return type === "u" ? theta + z * thetaSe : theta - z * thetaSe;
   } else if (dist === "t-dist") {
      var tval = qt(x, df);
      if (rse < 1) {
         // compute two intervals and return wider one per description
         var normalBound = (type === "u") ? theta + qnorm(x) * thetaSe : theta - qnorm(x) * thetaSe;
         var tBound = (type === "u") ? theta + tval * thetaSe * rse : theta - tval * thetaSe * rse;
         // For upper: take max; for lower: take min (wider)
         return (type === "u") ? Math.max(normalBound, tBound) : Math.min(normalBound, tBound);
      } else {
         return (type === "u") ? theta + tval * thetaSe : theta - tval * thetaSe;
      }
   } else {
      throw new Error("Distribution must be 'normal' or 't-dist'.");
   }
}

// ----------------------- Weighted Least Squares -----------------------
//
// Solves weighted regression y ~ X (with intercept column included in X if desired).
// X: matrix n x p (array of arrays), y: vector length n, w: weights vector length n
// Returns object {coef: vector length p, sigma: residual sd, vcov: p x p matrix, df_residual}
//
// For intercept-free case include column of 1s explicitly in X if needed.
//
function weightedLeastSquares(X, y, w) {
   var n = y.length;
   var p = X[0].length;
   // Build W^(1/2) * X and W^(1/2) * y
   var WX = [];
```

```
    var Wy = [];
    for (var i = 0; i < n; i++) {
        var wi = Math.sqrt(w[i]);
        Wy[i] = y[i] * wi;
        WX[i] = Array(p);
        for (var j = 0; j < p; j++) WX[i][j] = X[i][j] * wi;
    }
    // compute XtWX and XtWy
    var XtWX = math.multiply(math.transpose(WX), WX); // p x p
    var XtWy = math.multiply(math.transpose(WX), Wy); // p
    var coef, vcov;
    try {
        coef = math.multiply(math.inv(XtWX), XtWy); // p-vector
        vcov = math.inv(XtWX);
    } catch (err) {
        // singular; fallback to pseudoinverse
        coef = math.multiply(math.pinv(XtWX), XtWy);
        vcov = math.pinv(XtWX);
    }
    // residuals y - X beta
    var residuals = [];
    for (var i = 0; i < n; i++) {
        var pred = 0;
        for (var j = 0; j < p; j++) pred += X[i][j] * coef[j];
        residuals[i] = y[i] - pred;
    }
    // weighted sum squares
    var ssr = 0;
    for (var i = 0; i < n; i++) ssr += w[i] * residuals[i] * residuals[i];
    var dfRes = Math.max(1, n - p);
    var sigma = Math.sqrt(ssr / dfRes);
    return {
        coef: coef,
        sigma: sigma,
        vcov: vcov,
        residuals: residuals,
        df_residual: dfRes
    };
}

// ---------------------- Huber IRLS (approximate lmrob) ----------------------
//
// Implements Huber regression via IRLS to approximate robust regression.
// X: n x p, y: n, w_init: initial weights (optional) (we will multiply by these in each update)
// k: Huber tuning constant (default 1.345 => approx 95% efficiency)
```

```
// maxIter, tol
//
// Returns similar object as weightedLeastSquares with robust weights applied.
//
function huberIRLS(X, y, w_init, k = 1.345, maxIter = 200, tol = 1e-8) {
    var n = y.length;
    var p = X[0].length;
    var w_init_local = (w_init && w_init.length === n) ? w_init.slice() : Array(n).fill(1);
    // Start with weighted least squares with w_init
    var model = weightedLeastSquares(X, y, w_init_local);
    var beta = model.coef.map(v => +v); // array
    var sigma = model.sigma;
    var iter = 0;
    while (iter < maxIter) {
        iter++;
        // residuals
        var r = [];
        for (var i = 0; i < n; i++) {
            var pred = 0;
            for (var j = 0; j < p; j++) pred += X[i][j] * beta[j];
            r[i] = y[i] - pred;
        }
        // scale estimate: use MAD-like / 0.6745 or model sigma
        var absr = r.map(Math.abs);
        var med = jStat.median(absr);
        var s = med / 0.6745;
        if (!isNumber(s) || s === 0) s = sigma || 1;
        // compute Huber weights
        var huberW = [];
        for (var i = 0; i < n; i++) {
            var u = Math.abs(r[i]) / (s * k);
            // weight = 1 if u <=1, else 1/u
            huberW[i] = (u <= 1) ? 1 : (1 / u);
            // incorporate initial weights
            huberW[i] *= Math.sqrt(w_init_local[i] || 1);
            // square for weightedLeastSquares which expects weights (not sqrt) -> we'll pass huberW^2 as weights
        }
        var wSquare = huberW.map(x => x * x);
        var newModel = weightedLeastSquares(X, y, wSquare);
        var newBeta = newModel.coef.map(v => +v);
        var maxChange = 0;
        for (var j = 0; j < p; j++) maxChange = Math.max(maxChange, Math.abs(newBeta[j] - beta[j]));
        beta = newBeta;
        sigma = newModel.sigma;
        if (maxChange < tol) break;
```

```
    }
    // Compute final vcov (using final weights)
    var finalResiduals = [];
    for (var i = 0; i < n; i++) {
        var pred = 0;
        for (var j = 0; j < p; j++) pred += X[i][j] * beta[j];
        finalResiduals[i] = y[i] - pred;
    }
    var df_res = Math.max(1, n - p);
    var ssr = sum(finalResiduals.map((ri, i) => (ri * ri)));
    var finalSigma = Math.sqrt(ssr / df_res);
    // For vcov we reuse weightedLeastSquares machinery with final weights approximated as 1/(sigma^2)
    // Simpler: compute XtX inverse from X (unweighted) scaled by sigma^2:
    var XtX = math.multiply(math.transpose(X), X);
    var vcov;
    try { vcov = math.multiply(math.inv(XtX), (finalSigma * finalSigma)); }
    catch (err) { vcov = math.multiply(math.pinv(XtX), (finalSigma * finalSigma)); }
    return {
        coef: beta,
        sigma: finalSigma,
        vcov: vcov,
        residuals: finalResiduals,
        df_residual: df_res
    };
}

// ---------------------- mr_egger ----------------------
//
// Returns an object.
//
// Behavior notes:
// - If correlation matrix provided and correl flag is true (or correlation present in input),
//      performs generalized least squares on [1, Bx] design using omega (Byse outer Byse * rho).
// - If no correlation, performs weighted least squares with Byse^-2 weights.
// - Robust option uses Huber IRLS approximation.
//
// Parameters:
// object: { betaX, betaY, betaXse, betaYse, correlation (matrix or null), exposure, outcome, snps }
// options: { robust, penalized, correl, distribution, alpha }
//
function mr_egger(object, options) {
    if (!object || !object.betaX) throw new Error("object with betaX required.");
    var robust = !!options.robust;
    var penalized = !!options.penalized;
    var correlFlag = !!options.correl;
```

```
var distribution = options.distribution || "normal";
var alpha = (typeof options.alpha === "number") ? options.alpha : 0.001;

var betaX = object.betaX.slice();
var betaY = object.betaY.slice();
var betaXse = object.betaXse.slice();
var betaYse = object.betaYse.slice();
var correlation = object.correlation ? object.correlation : null;
var exposure = object.exposure || "";
var outcome = object.outcome || "";
var snps = object.snps ? object.snps.slice() : [];

var nsnps = betaX.length;
if (nsnps < 3) {
   throw new Error("Method requires data on >2 variants.");
}
if (["normal", "t-dist"].indexOf(distribution) === -1) {
   throw new Error("Distribution must be one of : normal, t-dist.");
}

// Ensure betaX estimate has positive values
var By = [], Bx = [];
for (var i = 0; i < nsnps; i++) {
   var s = (betaX[i] >= 0) ? 1 : -1;
   By[i] = s * betaY[i];
   Bx[i] = Math.abs(betaX[i]);
}

// If correlation present and not all NA, set correlFlag true
if (correlation && correlation.length === nsnps && correlation[0].length === nsnps) {
   // consider correlation present if not all NaN
   var anyFinite = false;
   for (var i = 0; i < nsnps && !anyFinite; i++) {
      for (var j = 0; j < nsnps && !anyFinite; j++) {
         if (isNumber(correlation[i][j])) anyFinite = true;
      }
   }
   if (anyFinite) correlFlag = true;
}

// Non-naive branching
if (correlFlag) {
   if (!correlation || correlation.length !== nsnps || correlation[0].length !== nsnps) {
      throw new Error("Correlation matrix not provided or wrong dimensions.");
   }
```

```
    // modify rho by sign outer product
    var signVec = betaX.map(x => (x >= 0 ? 1 : -1));
    var rho = math.clone(correlation);
    for (var i = 0; i < nsnps; i++) {
       for (var j = 0; j < nsnps; j++) rho[i][j] = rho[i][j] * (signVec[i] * signVec[j]);
    }
    // omega = Byse %o% Byse * rho    => outer product of Byse times rho componentwise
    var omega = math.zeros(nsnps, nsnps)._data;
    for (var i = 0; i < nsnps; i++) {
       for (var j = 0; j < nsnps; j++) {
          omega[i][j] = betaYse[i] * betaYse[j] * rho[i][j];
       }
    }
    // Design matrix [1, Bx]
    var X = [];
    for (var i = 0; i < nsnps; i++) X.push([1, Bx[i]]);
    // GLS estimate: theta.vals = (X^T Omega^-1 X)^-1 X^T Omega^-1 By
    var OmegaInv = matInv(omega);
    var Xt = matTranspose(X);
    var Xt_Oinv = matMul(Xt, OmegaInv);
    var XtOx = matMul(Xt_Oinv, X); // 2x2
    var XtOy = matMul(Xt_Oinv, By); // 2-vector
    var thetaVals = matMul(matInv(XtOx), XtOy); // 2-vector
    var thetaInter = thetaVals[0];
    var thetaE = thetaVals[1];
    // residuals r = By - (intercept + slope * Bx)
    var r = [];
    for (var i = 0; i < nsnps; i++) r[i] = By[i] - thetaInter - thetaE * Bx[i];
    // rse.corr = sqrt(t(r) Omega^-1 r / (nsnps - 2))
    var rMat = math.matrix([r]); // 1 x n
    var rOmegaInv = matMul(matMul(rMat, OmegaInv), math.transpose(rMat))._data[0][0];
    var rseCorr = Math.sqrt(rOmegaInv / Math.max(1, nsnps - 2));
    // sigma matrix = (X^T Omega^-1 X)^-1
    var sigma = matInv(XtOx);
    var thetaEse = Math.sqrt(sigma[1][1]) * Math.max(1, rseCorr);
    var thetaInterse = Math.sqrt(sigma[0][0]) * Math.max(1, rseCorr);

var ciUpper = eggerBounds("u", distribution, thetaE, thetaEse, nsnps - 2, rseCorr, alpha);
var ciLower = eggerBounds("l", distribution, thetaE, thetaEse, nsnps - 2, rseCorr, alpha);
var ciUpperInter = eggerBounds("u", distribution, thetaInter, thetaInterse, nsnps - 2, rseCorr, alpha);
var ciLowerInter = eggerBounds("l", distribution, thetaInter, thetaInterse, nsnps - 2, rseCorr, alpha);

var heter_stat = 0;
for (var i = 0; i < nsnps; i++) heter_stat += Math.pow((1 / betaYse[i]) * (By[i] - thetaInter - thetaE * Bx[i]), 2);
var pvalue_heter_stat = 1 - pchisq(heter_stat, nsnps - 2);
```

```
// pleiotropic & causal p-values depending on distribution + RSE logic
var pleiotropic_pvalue, causal_pvalue;
var tstatInter = thetaInter / thetaInterse;
var tstatE = thetaE / thetaEse;
if (distribution === "normal") {
   pleiotropic_pvalue = 2 * (1 - pnorm(Math.abs(tstatInter)));
   causal_pvalue = 2 * (1 - pnorm(Math.abs(tstatE)));
} else {
   // t-dist
   if (rseCorr < 1) {
      var pv1Inter = 2 * (1 - pnorm(Math.abs(tstatInter)));
      var pv2Inter = 2 * (1 - pt(Math.abs(tstatInter / rseCorr), nsnps - 2));
      pleiotropic_pvalue = Math.max(pv1Inter, pv2Inter);

      var pv1E = 2 * (1 - pnorm(Math.abs(tstatE)));
      var pv2E = 2 * (1 - pt(Math.abs(tstatE / rseCorr), nsnps - 2));
      causal_pvalue = Math.max(pv1E, pv2E);
   } else {
      pleiotropic_pvalue = 2 * (1 - pt(Math.abs(tstatInter), nsnps - 2));
      causal_pvalue = 2 * (1 - pt(Math.abs(tstatE), nsnps - 2));
   }
}

// Return result object
return {
   Model: "random",
   Exposure: exposure,
   Outcome: outcome,
   Robust: robust,
   Penalized: penalized,
   Correlation: rho,
   Estimate: thetaE,
   StdError_Est: thetaEse,
   CILower_Est: ciLower,
   CIUpper_Est: ciUpper,
   Pvalue_Est: causal_pvalue,
   Intercept: thetaInter,
   StdError_Int: thetaInterse,
   CILower_Int: ciLowerInter,
   CIUpper_Int: ciUpperInter,
   Pvalue_Int: pleiotropic_pvalue,
   Pleio_pval: pleiotropic_pvalue,
   Causal_pval: causal_pvalue,
   Alpha: alpha,
```

```
    SNPs: nsnps,
    RSE: rseCorr,
    Heter_Stat: [heter_stat, pvalue_heter_stat],
    I_sq: NaN
};

    } else {
        // No correlation case: use weighted regression (By ~ Bx)
        // Build design matrix X = [1, Bx]
        var X = [];
        for (var i = 0; i < nsnps; i++) X.push([1, Bx[i]]);
        // weights default w = 1 / Byse^2
        var w = betaYse.map(s => 1 / (s * s));

// Branch handling for robust/penalized combos
var modelSummary;
var rse, thetaE, thetaEse, thetaInter, thetaInterse, ciUpper, ciLower, ciUpperInter, ciLowerInter;
if (robust) {
    if (penalized) {
        var olsInit = weightedLeastSquares(X, By, w);
        var intercept0 = olsInit.coef[0];
        var slope0 = olsInit.coef[1];
        // pen.weights = pchisq((1/Byse^2)*(By - intercept - slope*Bx)^2, df=1, lower.tail=F)
        var pen_weights = [];
        for (var i = 0; i < nsnps; i++) {
            var val = (1 / (betaYse[i] * betaYse[i])) * Math.pow(By[i] - intercept0 - slope0 * Bx[i], 2);
            pen_weights[i] = 1 - pchisq(val, 1);
        }
        // r.weights = Byse^-2 * pmin(1, pen.weights*100)
        var r_weights = [];
        for (var i = 0; i < nsnps; i++) r_weights[i] = w[i] * Math.min(1, pen_weights[i] * 100);
        // use huberIRLS with these weights
        modelSummary = huberIRLS(X, By, r_weights);
        rse = modelSummary.sigma;
        thetaInter = modelSummary.coef[0];
        thetaE = modelSummary.coef[1];
        // approximate se from vcov diagonal if available
        // modelSummary.vcov for huberIRLS used approximated version
        var seInter = Math.sqrt(Math.abs(modelSummary.vcov ? modelSummary.vcov[0][0] : (1 / Math.max(1e-12, math.multiply(math.transpose(X), X)[0][0])) ));
        var seE = Math.sqrt(Math.abs(modelSummary.vcov ? modelSummary.vcov[1][1] : (1 / Math.max(1e-12, math.multiply(math.transpose(X), X)[1][1])) ));
        thetaInterse = seInter / Math.min(rse, 1);
        thetaEse = seE / Math.min(rse, 1);
        ciUpper = eggerBounds("u", distribution, thetaE, thetaEse, nsnps - 2, rse, alpha);
```

```
        ciLower = eggerBounds("l", distribution, thetaE, thetaEse, nsnps - 2, rse, alpha);
        ciUpperInter = eggerBounds("u", distribution, thetaInter, thetaInterse, nsnps - 2, rse, alpha);
        ciLowerInter = eggerBounds("l", distribution, thetaInter, thetaInterse, nsnps - 2, rse, alpha);
    } else {
        // Robust but not penalized: apply Huber IRLS with initial weights w
        modelSummary = huberIRLS(X, By, w);
        rse = modelSummary.sigma;
        thetaInter = modelSummary.coef[0];
        thetaE = modelSummary.coef[1];
        // approximate se from modelSummary.vcov
        var seInter = Math.sqrt(Math.abs(modelSummary.vcov ? modelSummary.vcov[0][0] : 0));
        var seE = Math.sqrt(Math.abs(modelSummary.vcov ? modelSummary.vcov[1][1] : 0));
        thetaInterse = seInter / Math.min(rse, 1);
        thetaEse = seE / Math.min(rse, 1);
        ciUpper = eggerBounds("u", distribution, thetaE, thetaEse, nsnps - 2, rse, alpha);
        ciLower = eggerBounds("l", distribution, thetaE, thetaEse, nsnps - 2, rse, alpha);
        ciUpperInter = eggerBounds("u", distribution, thetaInter, thetaInterse, nsnps - 2, rse, alpha);
        ciLowerInter = eggerBounds("l", distribution, thetaInter, thetaInterse, nsnps - 2, rse, alpha);
    }
} else if (penalized && !robust) {
    // penalized only: compute pen-weights from initial OLS, then do weighted least squares with r.weights
    var olsInit = weightedLeastSquares(X, By, w);
    var intercept0 = olsInit.coef[0];
    var slope0 = olsInit.coef[1];
    var pen_weights = [];
    for (var i = 0; i < nsnps; i++) {
        var val = (1 / (betaYse[i] * betaYse[i])) * Math.pow(By[i] - intercept0 - slope0 * Bx[i], 2);
        pen_weights[i] = 1 - pchisq(val, 1);
    }
    var r_weights = [];
    for (var i = 0; i < nsnps; i++) r_weights[i] = w[i] * Math.min(1, pen_weights[i] * 100);
    modelSummary = weightedLeastSquares(X, By, r_weights);
    rse = modelSummary.sigma;
    thetaInter = modelSummary.coef[0];
    thetaE = modelSummary.coef[1];
    // se from vcov
    var seInter = Math.sqrt(Math.abs(modelSummary.vcov[0][0]));
    var seE = Math.sqrt(Math.abs(modelSummary.vcov[1][1]));
    thetaInterse = seInter / Math.min(rse, 1);
    thetaEse = seE / Math.min(rse, 1);
    ciUpper = eggerBounds("u", distribution, thetaE, thetaEse, nsnps - 2, rse, alpha);
    ciLower = eggerBounds("l", distribution, thetaE, thetaEse, nsnps - 2, rse, alpha);
    ciUpperInter = eggerBounds("u", distribution, thetaInter, thetaInterse, nsnps - 2, rse, alpha);
    ciLowerInter = eggerBounds("l", distribution, thetaInter, thetaInterse, nsnps - 2, rse, alpha);
} else {
```

```
    // Standard weighted least squares
    modelSummary = weightedLeastSquares(X, By, w);
    rse = modelSummary.sigma;
    thetaInter = modelSummary.coef[0];
    thetaE = modelSummary.coef[1];
    var seInter = Math.sqrt(Math.abs(modelSummary.vcov[0][0]));
    var seE = Math.sqrt(Math.abs(modelSummary.vcov[1][1]));
    thetaInterse = seInter / Math.min(rse, 1);
    thetaEse = seE / Math.min(rse, 1);
    ciUpper = eggerBounds("u", distribution, thetaE, thetaEse, nsnps - 2, rse, alpha);
    ciLower = eggerBounds("l", distribution, thetaE, thetaEse, nsnps - 2, rse, alpha);
    ciUpperInter = eggerBounds("u", distribution, thetaInter, thetaInterse, nsnps - 2, rse, alpha);
    ciLowerInter = eggerBounds("l", distribution, thetaInter, thetaInterse, nsnps - 2, rse, alpha);
}

var heter_stat = NaN, pvalue_heter_stat = NaN;
if (!penalized || (penalized && !isNaN(thetaInter))) {
    // For branches where heter.stat defined: sum(((1/Byse)*(By - intercept - slope*Bx))^2)
    heter_stat = 0;
    for (var i = 0; i < nsnps; i++) heter_stat += Math.pow((1 / betaYse[i]) * (By[i] - thetaInter - thetaE * Bx[i]), 2);
    pvalue_heter_stat = 1 - pchisq(heter_stat, nsnps - 2);
}

// pleiotropic & causal p-values with RSE logic
var pleiotropic_pvalue, causal_pvalue;
var tstatInter = thetaInter / thetaInterse;
var tstatE = thetaE / thetaEse;
if (distribution === "normal") {
    pleiotropic_pvalue = 2 * (1 - pnorm(Math.abs(tstatInter)));
    causal_pvalue = 2 * (1 - pnorm(Math.abs(tstatE)));
} else {
    if (rse < 1) {
        pleiotropic_pvalue = Math.max(2 * (1 - pnorm(Math.abs(tstatInter))),
            2 * (1 - pt(Math.abs(tstatInter / rse), nsnps - 2)));
        causal_pvalue = Math.max(2 * (1 - pnorm(Math.abs(tstatE))),
            2 * (1 - pt(Math.abs(tstatE / rse), nsnps - 2)));
    } else {
        pleiotropic_pvalue = 2 * (1 - pt(Math.abs(tstatInter), nsnps - 2));
        causal_pvalue = 2 * (1 - pt(Math.abs(tstatE), nsnps - 2));
    }
}

// Q/I-sq:
// Q = sum((Bxse/Byse)^-2*(Bx/Byse - weighted.mean(Bx/Byse, w=(Bxse/Byse)^-2))^2)
var ratio = [], wQ = [];
```

```
for (var i = 0; i < nsnps; i++) {
   var denom = (betaXse[i] / betaYse[i]);
   if (denom === 0) { ratio[i] = 0; wQ[i] = 0; }
   else {
      ratio[i] = (Bx[i] / betaYse[i]);
      wQ[i] = 1 / (denom * denom);
   }
}
var meanRatio = weightedMean(ratio, wQ);
var Q = 0;
for (var i = 0; i < nsnps; i++) Q += (wQ[i] * Math.pow(ratio[i] - meanRatio, 2));
var Isq = (Q === 0) ? 0 : Math.max(0, (Q - (nsnps - 1)) / Q);

return {
   Model: "random",
   Exposure: exposure,
   Outcome: outcome,
   Robust: robust,
   Penalized: penalized,
   Correlation: [],
   Estimate: thetaE,
   StdError_Est: thetaEse,
   CILower_Est: ciLower,
   CIUpper_Est: ciUpper,
   Pvalue_Est: causal_pvalue,
   Intercept: thetaInter,
   StdError_Int: thetaInterse,
   CILower_Int: ciLowerInter,
   CIUpper_Int: ciUpperInter,
   Pvalue_Int: pleiotropic_pvalue,
   Pleio_pval: pleiotropic_pvalue,
   Causal_pval: causal_pvalue,
   Alpha: alpha,
   SNPs: nsnps,
   RSE: rse,
   Heter_Stat: [heter_stat, pvalue_heter_stat],
   I_sq: Isq
};

   }
}
```

3.2.2 Algorithms for multivariables

(1) The Multivariable Egger Method (MVEgger)

The JavaScript codes for The Multivariable Egger Method (MVEgger) are as follows:

/*

Method name and objective

Name: The Multivariable Egger Method (MVEgger) for Mendelian Randomization.

Objective: To estimate causal effects for multiple exposures on an outcome while allowing for directional pleiotropy via an intercept term. Supports both uncorrelated and correlated instruments, and provides CIs using either a normal or t-distribution-based approach.

Main classes and functions

Core function: mr_mvegger(object, orientate = 1, correl = false, distribution = "normal", alpha = 0.001)

orchestrates the Multivariable Egger analysis.

Helper utilities:

isMatrixNaN(mat): checks for NaN or missing entries in a matrix.

to2D(a): ensures inputs are treated as 2D arrays.

flatten1D(a): flattens 2D single-column arrays to 1D.

ci_normal(side, est, se, alphaVal): computes normal-based confidence intervals.

ci_t(side, est, se, df, alphaVal): computes t-distribution-based confidence intervals.

transpose, multiply, inv: basic linear algebra operations (via libraries like math.js/jStat).

diagOfMatrixToArray, diagSqrt: extract diagonal elements and their square roots (standard errors).

ensure2D(mat): utility to normalize input shapes.

Computation branches:

Correlated instruments (correl = true): builds a weighting matrix omega from Byse and a correlation adjustment (rhoAdjusted), then fits a generalized least-squares Egger model with an intercept and p exposure effects.

Uncorrelated instruments (correl = false): performs weighted linear regression with weights 1/Byse^2, estimates exposure effects and an intercept, and computes residual-based standard errors.

Output structure:

An object with fields such as Model, Orientate, Exposure, Outcome, Correlation, Estimate, StdErrorEst, CILowerEst, CIUpperEst, PvalueEst, Intercept, StdErrorInt, CILowerInt, CIUpperInt, PvalueInt, Alpha, SNPs, RSE, HeterStat.

Estimates correspond to each exposure; Intercept captures pleiotropy; includes p-values and confidence intervals for both estimates and intercept.

Required inputs and expected outputs

Required inputs:

object: an object containing:

betaX: nsnps x p matrix of SNP–exposure associations.

betaY: length-nsnps vector of SNP–outcome associations.

betaXse: nsnps x p matrix of standard errors for betaX.

betaYse: length-nsnps vector of standard errors for betaY.

exposure: label for the exposure (optional).

outcome: label for the outcome (optional).

correlation (optional): nsnps x nsnps matrix if instruments are correlated.

orientate: integer indicating which exposure column to use for orientation/sign alignment (default 1).

correl: boolean indicating whether to use the correlated-instrument branch (default false).

distribution: "normal" or "t-dist" for confidence-interval calculation (default "normal").
alpha: significance level for CIs (default 0.001).

Expected outputs:
A JavaScript object with:
Model, Orientate, Exposure, Outcome, Correlation
Estimate: array of p exposure effects
StdErrorEst, CILowerEst, CIUpperEst, PvalueEst for the exposure effects
Intercept, StdErrorInt, CILowerInt, CIUpperInt, PvalueInt
Alpha, SNPs, RSE, HeterStat
If inputs are invalid (e.g., missing correlation when correl = true), the function throws an error with a descriptive message.

Notes
The implementation relies on linear algebra helpers (and likely math.js/jStat) for matrix operations and distribution quantiles.
The orientation step flips signs of the SNP effects to align with the chosen orientation, which affects both the estimates and the intercept.
The correlated branch computes a GLS-type estimator with a weighted covariance matrix (omega); the uncorrelated branch uses weighted least squares and a residual-based standard error.
*/

```
function mr_mvegger(object, orientate = 1, correl = false, distribution = "normal", alpha = 0.001) {
  // Helper functions
  function isMatrixNaN(mat) {
    if (mat === null || mat === undefined) return true;
    // assume 2D array
    for (let i = 0; i < mat.length; i++) {
      for (let j = 0; j < mat[0].length; j++) {
        if (mat[i][j] === null || mat[i][j] === undefined || Number.isNaN(mat[i][j])) return true;
      }
    }
    return false;
  }

  function to2D(a) {
    // convert 1D to 2D column
    if (!Array.isArray(a)) return [[a]];
    if (Array.isArray(a[0])) return a; // already 2D
    return a.map(v => [v]);
  }

  function flatten1D(a) {
    // if 2D single-col, convert to 1D
    if (!Array.isArray(a)) return [a];
    if (Array.isArray(a[0]) && a[0].length === 1) return a.map(r => r[0]);
    if (Array.isArray(a[0]) && a[0].length > 1) {
```

```
        // flatten row-major if it's a single row
        if (a.length === 1) return a[0].slice();
    }
    return a.slice();
}

function ci_normal(side, est, se, alphaVal) {
    // est and se may be arrays
    const z = Math.abs(jStat.normal.inv(alphaVal / 2, 0, 1)); // two-sided critical
    // For lower ("l") we return est - z * se
    // For upper ("u") we return est + z * se
    const estA = Array.isArray(est) ? est : flatten1D(est);
    const seA = Array.isArray(se) ? se : flatten1D(se);
    const out = estA.map((e, i) => {
        if (side === "l") return e - z * seA[i];
        return e + z * seA[i];
    });
    return out;
}

function ci_t(side, est, se, df, alphaVal) {
    // t critical value for two-sided alpha
    const tcrit = Math.abs(jStat.studentt.inv(alphaVal / 2, df)); // positive
    const estA = Array.isArray(est) ? est : flatten1D(est);
    const seA = Array.isArray(se) ? se : flatten1D(se);
    const out = estA.map((e, i) => {
        if (side === "l") return e - tcrit * seA[i];
        return e + tcrit * seA[i];
    });
    return out;
}

function transpose(mat) {
    return math.transpose(mat);
}

function multiply(A, B) {
    return math.multiply(A, B);
}

function inv(A) {
    return math.inv(A);
}

function diagOfMatrixToArray(M) {
```

```
    const n = Math.min(M.length, M[0].length);
    const out = [];
    for (let i = 0; i < n; i++) out.push(M[i][i]);
    return out;
  }

  function diagSqrt(arr) {
    return arr.map(x => Math.sqrt(Math.max(0, x)));
  }

  function ensure2D(mat) {
    // convert arrays to proper 2D arrays for math.js operations
    if (!Array.isArray(mat)) return [[mat]];
    if (!Array.isArray(mat[0])) return mat.map(v => [v]);
    return mat;
}

  // end helpers

  // Validate distribution
  if (distribution !== "normal" && distribution !== "t-dist") {
    console.error("Distribution must be one of : normal, t-dist.");
    throw new Error("Distribution must be one of : normal, t-dist.");
  }

  // Extract fields & coerce to matrices/arrays
  if (!object || typeof object !== "object") throw new Error("object must be provided");

  let Bx = ensure2D(object.betaX); // nsnps x p
  let By = flatten1D(object.betaY); // length nsnps
  let Bxse = ensure2D(object.betaXse); // nsnps x p
  let Byse = flatten1D(object.betaYse); // length nsnps
  const rho = object.correlation === undefined ? null : object.correlation;

  const nsnps = Bx.length;
  const p = Bx[0].length; // number of exposures

  // orientate checking (R code: if orientate in 1:dim(betaX)[2])
  let orientAte = 1;
  if (Number.isInteger(orientate) && orientate >= 1 && orientate <= p) orientAte = orientate;
  // orient vector: sign(object@betaX)[,orientAte]
  // sign by column orientAte: get column values and get sign for each SNP
  const signCol = [];
  for (let i = 0; i < nsnps; i++) {
    const val = Bx[i][orientAte - 1];
```

```
    signCol.push(val > 0 ? 1 : val < 0 ? -1 : 0);
  }
  // orientate all columns: Bx = object@betaX * orient (elementwise multiply each row by sign)
  const Bx_oriented = [];
  const By_oriented = [];
  for (let i = 0; i < nsnps; i++) {
    const rowBx = [];
    for (let j = 0; j < p; j++) {
      rowBx.push(Bx[i][j] * signCol[i]);
    }
    Bx_oriented.push(rowBx);
    By_oriented.push(By[i] * signCol[i]);
  }
  Bx = Bx_oriented;
  By = By_oriented;
  Bxse = ensure2D(Bxse);
  Byse = flatten1D(Byse);

  // if correlation provided, set correl true
  if (rho !== null && rho !== undefined) correl = true;

  // check correl branch
  if (distribution === "normal" || distribution === "t-dist") {
    if (correl === true) {
      if (!rho || isMatrixNaN(rho)) {
        console.error("Correlation matrix not given.");
        throw new Error("Correlation matrix not given.");
      } else {
        // compute adjusted rho = object.correlation * (orient %o% orient)
        // orient outer product: signCol %o% signCol
        const orientOuter = [];
        for (let i = 0; i < nsnps; i++) {
          const row = [];
          for (let j = 0; j < nsnps; j++) {
            row.push(signCol[i] * signCol[j]);
          }
          orientOuter.push(row);
        }
        // rho is nsnps x nsnps
        const rhoAdjusted = [];
        for (let i = 0; i < nsnps; i++) {
          const row = [];
          for (let j = 0; j < nsnps; j++) {
            row.push(rho[i][j] * orientOuter[i][j]);
          }
```

```
            rhoAdjusted.push(row);
        }

    // omega <- Byse %o% Byse * rhoAdjusted
    const outerByse = [];
    for (let i = 0; i < nsnps; i++) {
      const row = [];
      for (let j = 0; j < nsnps; j++) {
        row.push(Byse[i] * Byse[j]);
      }
      outerByse.push(row);
    }
    const omega = [];
    for (let i = 0; i < nsnps; i++) {
      const row = [];
      for (let j = 0; j < nsnps; j++) {
        row.push(outerByse[i][j] * rhoAdjusted[i][j]);
      }
      omega.push(row);
    }

    // design matrix X = cbind(Bx, rep(1, nsnps))
    const ones = Array(nsnps).fill(1);
    const X = Bx.map((row, i) => row.concat([1])); // nsnps x (p+1)

    // compute thetaEgger = inv(X^T inv(omega) X) X^T inv(omega) By
    const Xt = transpose(X);
    const invOmega = inv(omega);
    const Xt_invOmega = multiply(Xt, invOmega);
    const Xt_invOmega_X = multiply(Xt_invOmega, X);
    const Xt_invOmega_X_inv = inv(Xt_invOmega_X);
    const Xt_invOmega_By = multiply(Xt_invOmega, to2D(By)); // nsnpsx1 -> returns (p+1)x1 after multi
    let thetaEggerMat = multiply(Xt_invOmega_X_inv, Xt_invOmega_By); // (p+1)x1
    // flatten
    const thetaEgger = flatten1D(thetaEggerMat);

    // residuals rse = By - X %*% thetaEgger
    const X_theta = multiply(X, thetaEggerMat); // nsnps x 1
    const rse = [];
    for (let i = 0; i < nsnps; i++) {
      const XiTh = X_theta[i][0];
      rse.push(By[i] - XiTh);
    }
    // thetaEggerse = sqrt(diag(inv(X^T inv(omega) X))) * max(sqrt(t(rse) %*% inv(omega) %*% rse / (nsnps - p -1)), 1)
    const covTheta = Xt_invOmega_X_inv;
```

```
const covDiag = diagOfMatrixToArray(covTheta);
const covStd = diagSqrt(covDiag);
// compute rse^T invOmega rse
const rseCol = to2D(rse);
const tmp = multiply(transpose(rseCol), multiply(invOmega, rseCol)); // 1x1
const rse_corr = Math.sqrt(tmp[0][0] / (nsnps - p - 1));
const scaleFactor = Math.max(rse_corr, 1);
const thetaEggerse = covStd.map(v => v * scaleFactor);

const correlation = true;

// Compute CI
let ciLower, ciUpper;
if (distribution === "normal") {
   ciLower = ci_normal("l", thetaEgger, thetaEggerse, alpha);
   ciUpper = ci_normal("u", thetaEgger, thetaEggerse, alpha);
} else {
   ciLower = ci_t("l", thetaEgger, thetaEggerse, nsnps - p - 1, alpha);
   ciUpper = ci_t("u", thetaEgger, thetaEggerse, nsnps - p - 1, alpha);
}

// heter.stat and pvalue
const rseCorr = rse_corr;
const heter_stat = (nsnps - p - 1) * (rseCorr * rseCorr);
const pvalue_heter_stat = 1 - jStat.chisquare.cdf(heter_stat, nsnps - p - 1);

// p-values for estimates and intercept
const tvals = thetaEgger.map((th, i) => th / thetaEggerse[i]);
let pvalue;
if (distribution === "normal") {
   pvalue = tvals.map(t => 2 * jStat.normal.cdf(-Math.abs(t), 0, 1));
} else {
   pvalue = tvals.map(t => 2 * (1 - jStat.studentt.cdf(Math.abs(t), nsnps - p - 1)));
}

// Build return object similar to new("MVEgger", ...)
const out = {
   Model: "random",
   Orientate: orientAte,
   Exposure: object.exposure || null,
   Outcome: object.outcome || null,
   Correlation: rho,
   Estimate: thetaEgger.slice(0, p), // first p entries are estimates for exposures
   StdErrorEst: thetaEggerse.slice(0, p),
   CILowerEst: ciLower.slice(0, p),
```

```
        CIUpperEst: ciUpper.slice(0, p),
        PvalueEst: pvalue.slice(0, p),
        Intercept: thetaEgger[p],
        StdErrorInt: thetaEggerse[p],
        CILowerInt: ciLower[p],
        CIUpperInt: ciUpper[p],
        PvalueInt: pvalue[p],
        Alpha: alpha,
        SNPs: nsnps,
        RSE: rseCorr,
        HeterStat: [heter_stat, pvalue_heter_stat]
      };
      return out;
    }
  } else {
    // not correlated branch
    // Weighted linear regression: lm(By ~ Bx, weights = Byse^(-2))
    // Build design matrix X = [column1..p, 1] => nsnps x (p+1)
    const X = Bx.map((row, i) => row.concat([1]));
    const y = to2D(By); // nsnps x 1
    // Weights vector w = 1 / (Byse^2)
    const w = Byse.map(v => 1 / (v * v));
    // Form W = diagonal matrix of weights
    const W = math.diag(w);

    const Xt = transpose(X); // (p+1) x nsnps
    // Xt W X
    const Xt_W = multiply(Xt, W); // (p+1) x nsnps
    const Xt_W_X = multiply(Xt_W, X); // (p+1) x (p+1)
    const Xt_W_X_inv = inv(Xt_W_X);
    const Xt_W_y = multiply(Xt_W, y); // (p+1) x 1
    const thetaEggerMat = multiply(Xt_W_X_inv, Xt_W_y); // (p+1) x 1
    const thetaEgger = flatten1D(thetaEggerMat);

    // residuals r = y - X %*% thetaEgger
    const X_theta = multiply(X, thetaEggerMat); // nsnps x 1
    const rVec = [];
    for (let i = 0; i < nsnps; i++) {
      rVec.push(y[i][0] - X_theta[i][0]);
    }

    const df_resid = nsnps - p - 1;
    const numerator = rVec.reduce((acc, ri, i) => acc + w[i] * ri * ri, 0);
    const sigma = Math.sqrt(Math.max(0, numerator / df_resid));
    const sigma_for_se = Math.min(sigma, 1);
```

```
// thetaEggerse <- summary$coef[,2] / min(summary$sigma, 1)
// In matrix world, var(theta) = inv(Xt W X)
const covTheta = Xt_W_X_inv;
const covDiag = diagOfMatrixToArray(covTheta);
const seTheta = covDiag.map(v => Math.sqrt(Math.max(0, v)));
const thetaEggerse = seTheta.map(v => v / sigma_for_se);

// p-values
let pvalue;
if (distribution === "normal") {
  pvalue = thetaEgger.map((th, i) => 2 * jStat.normal.cdf(-Math.abs(th / thetaEggerse[i]), 0, 1));
} else {
  pvalue = thetaEgger.map((th, i) => 2 * (1 - jStat.studentt.cdf(Math.abs(th / thetaEggerse[i]), nsnps - p)));
}

const rse_val = sigma;
const heter_stat = (nsnps - p - 1) * (rse_val * rse_val);
const pvalue_heter_stat = 1 - jStat.chisquare.cdf(heter_stat, nsnps - p - 1);

// Compute CIs
let ciLower, ciUpper;
if (distribution === "normal") {
  ciLower = ci_normal("l", thetaEgger, thetaEggerse, alpha);
  ciUpper = ci_normal("u", thetaEgger, thetaEggerse, alpha);
} else {
  ciLower = ci_t("l", thetaEgger, thetaEggerse, nsnps - p - 1, alpha);
  ciUpper = ci_t("u", thetaEgger, thetaEggerse, nsnps - p - 1, alpha);
}

const Estimate = thetaEgger.slice(0, p).slice(0); // careful ordering:
// In our thetaEgger, intercept is last element. So produce:
const Estimate_Rstyle = thetaEgger.slice(0, p); // these correspond to exposures
const StdErrorEst_Rstyle = thetaEggerse.slice(0, p);
const CILowerEst_Rstyle = ciLower.slice(0, p);
const CIUpperEst_Rstyle = ciUpper.slice(0, p);
const PvalueEst_Rstyle = pvalue.slice(0, p);

const Intercept = thetaEgger[p];
const StdErrorInt = thetaEggerse[p];
const CILowerInt = ciLower[p];
const CIUpperInt = ciUpper[p];
const PvalueInt = pvalue[p];

const out = {
```

```
      Model: "random",
      Orientate: orientAte,
      Exposure: object.exposure || null,
      Outcome: object.outcome || null,
      Correlation: rho,
      Estimate: Estimate_Rstyle,
      StdErrorEst: StdErrorEst_Rstyle,
      CILowerEst: CILowerEst_Rstyle,
      CIUpperEst: CIUpperEst_Rstyle,
      PvalueEst: PvalueEst_Rstyle,
      Intercept: Intercept,
      StdErrorInt: StdErrorInt,
      CILowerInt: CILowerInt,
      CIUpperInt: CIUpperInt,
      PvalueInt: PvalueInt,
      Alpha: alpha,
      SNPs: nsnps,
      RSE: rse_val,
      HeterStat: [heter_stat, pvalue_heter_stat]
    };
    return out;
}

    } else {
      console.error("Distribution must be one of : normal, t-dist.");
      throw new Error("Distribution must be one of : normal, t-dist.");
    }
}

// Export for node environments (optional)
if (typeof module !== "undefined" && module.exports) {
    module.exports = { mr_mvegger };
}
```

(2) The Multivariable Inverse-Variance Weighted Method (MVIVW)

The JavaScript codes for The Multivariable Inverse-Variance Weighted Method (MVIVW) are as follows:

```
/*
Method name and objective
The Multivariable Inverse-Variance Weighted Method (MVIVW)
Objective: estimate the causal effects of multiple exposures on an outcome using multiple genetic instruments (SNPs), while accounting for LD among SNPs and correlations among exposures. Produces one effect estimate per exposure.

Main classes and functions (high-level)
Utility and matrix helpers
```

isMatrix, colVec, flattenCol, t (transpose), inv (inverse), mm (matrix multiply), I (identity), diagFrom, kron (Kronecker product)
toArray (normalize inputs to arrays)
Linear algebra and numerics
eigenDecompose, matrixSqrt (for symmetric matrices)
solve (solve linear systems)
Optimization
nelderMead (a small Nelder–Mead optimizer used in condFstat computations)
Conditional F-statistics
condFstat(Bx, Bxse, nx, ld, corx): computes ambient conditional F-statistics for each exposure, accounting for LD (rho) and exposure correlation (corx)
Handles both K > 2 and K = 2 cases, with internal steps for building projectors and solving subproblems
Main method
mr_mvivw(object, opts): orchestrates the MVIVW calculation
Inputs: object containing betaX (nsnp x K), betaY (nsnp), betaXse, betaYse, optional correlation (ld, nsnps x nsnps), exposure/outcome labels
Options: model (default/fixed/random), robust, correl, correl_x (K x K exposure correlation), nx (sample sizes per exposure), distribution ("normal" or "t-dist"), alpha
Behavior:
Prepares nx (handling single value or NaNs) and correl_x/ld
Computes condFstat if nx is provided
Chooses model (default: fixed for few SNPs, random otherwise)
If exposures are correlated (correl = true), builds omega from Byse and rho
Computes IVW estimates (thetaIVW) for correlated exposures; otherwise uses a weighted least-squares approach for uncorrelated case
Computes standard errors (thetaIVWse), CIs (normal or t-distribution), and p-values
Returns a result object with comprehensive summary

Required inputs and expected outputs

Required inputs
object
betaX: nsnps x K matrix of SNP-exposure effects
betaY: length nsnps vector of SNP-outcome effects
betaXse: nsnps x K matrix of standard errors for betaX
betaYse: length nsnps vector of standard errors for betaY
correlation (rho): nsnps x nsnps matrix of LD between SNPs (optional; defaults to identity)
exposure: name(s) of exposure(s)
outcome: name of outcome
opts
nx: array of sample sizes per exposure (length K) or a single value to replicate
correl: whether to account for exposure correlations (boolean)
correl_x: K x K exposure correlation matrix (if correl is true)
ld: alternative way to supply SNP-SNP LD (if not using correlation structure)
model: "default" | "fixed" | "random"
distribution: "normal" | "t-dist"

alpha: significance level for CIs/p-values (e.g., 0.05)

Outputs
An object containing:
Model: "fixed" | "random" (or whatever was chosen)
Exposure: exposure name(s)
Outcome: outcome name
Robust: whether a robust (not used in some branches) approach was applied
Correlation: the rho matrix supplied (or the default)
Estimate: array of K MVIVW estimates (one per exposure)
StdError: array of standard errors (one per exposure)
CILower / CIUpper: arrays of lower/upper confidence interval bounds
SNPs: number of SNPs used
Pvalue: array of p-values (one per exposure)
Alpha: significance level used
RSE: residual standard error (or related metric)
HeterStat: [statistic, p-value] for heterogeneity (per some implementations)
CondFstat: array of conditional F statistics per exposure (or NaN if not computed)

Notes
The method handles two main paths:
When exposures are correlated (correl = true): MVIVW uses a weighted/omega-adjusted formula that accounts for LD and exposure correlation, delivering multivariate IVW estimates with CIs and p-values.
When exposures are uncorrelated: a simpler weighted least squares approach is used with appropriate SE and CI calculations.
The code relies on matrix operations (via math.js) and statistical functions (via jStat) and may require those libraries to be present.
If nx contains NaN, condFstat yields NaNs, and the MVIVW results proceed under uninformative strength assumptions for the exposure(s).
*/

```
// Helper: check if value is a 2D matrix (Array of Arrays)
function isMatrix(A) {
  return Array.isArray(A) && Array.isArray(A[0]);
}

// Convert Array/TypedArray to column vector (n x 1) array of arrays
function colVec(a) {
  return a.map(v => [v]);
}

// Flatten column vector matrix to 1D array
function flattenCol(m) {
  if (!isMatrix(m)) return m;
  return m.map(r => r[0]);
}
```

```
// Transpose
function t(A) {
  return math.transpose(A);
}

// Matrix inverse
function inv(A) {
  return math.inv(A);
}

// Matrix multiply
function mm(A, B) {
  return math.multiply(A, B);
}

// Identity matrix n x n
function I(n) {
  return math.identity(n)._data || math.identity(n);
}

// Diagonal matrix from 1D array
function diagFrom(arr) {
  return math.diag(arr);
}

// Kronecker product (A kron B)
function kron(A, B) {
  // A and B are arrays (2D)
  const aRows = A.length, aCols = A[0].length;
  const bRows = B.length, bCols = B[0].length;
  const out = new Array(aRows * bRows);
  for (let i = 0; i < aRows * bRows; i++) {
    out[i] = new Array(aCols * bCols);
  }
  for (let i = 0; i < aRows; i++) {
    for (let j = 0; j < aCols; j++) {
      const aij = A[i][j];
      for (let p = 0; p < bRows; p++) {
        for (let q = 0; q < bCols; q++) {
          out[i * bRows + p][j * bCols + q] = aij * B[p][q];
        }
      }
    }
  }
```

```
  return out;
}

// Eigen-decomposition using math.eigs if available. Returns {values:[], vectors:matrix}
function eigenDecompose(M) {
  if (!math.eigs) {
    throw new Error("math.eigs is required for eigen-decomposition. Ensure math.js build supports eigs.");
  }
  // math.eigs expects a matrix object; many builds return an object with values and vectors
  const eig = math.eigs(M);
  // eig.values may be complex; we assume symmetric real matrices (R code uses eigen for symmetric matrices)
  const vals = eig.values.map(v => (typeof v === 'object' && 're' in v) ? v.re : v);
  const vecs = eig.vectors;
  return { values: vals, vectors: vecs };
}

// Matrix square root via eigen-decomposition (symmetric positive semidefinite assumed)
function matrixSqrt(M) {
  const ed = eigenDecompose(M);
  const vals = ed.values;
  const vecs = ed.vectors;
  // build diag(sqrt(vals))
  const sqrtVals = vals.map(v => {
    if (v < 0 && Math.abs(v) < 1e-12) return 0; // numerical tolerance
    if (v < 0) throw new Error("Matrix has negative eigenvalues; cannot take sqrt");
    return Math.sqrt(v);
  });
  const D = math.diag(sqrtVals);
  return mm(mm(vecs, D), t(vecs));
}

// Solve linear system using math.lusolve or inv * multiply for small matrices
function solve(A, B) {
  // A: matrix NxN, B: NxM
  // Prefer math.lusolve if available
  if (math.lusolve) {
    // convert B to column if B is vector
    try {
      return math.lusolve(A, B);
    } catch (e) {
      // fallback
    }
  }
  return mm(inv(A), B);
}
```

```
// Vector dot product
function dot(a, b) {
  // a and b are arrays (1D)
  let s = 0;
  for (let i = 0; i < a.length; i++) s += a[i] * b[i];
  return s;
}

// Convert matrix to JS nested arrays if mathjs typed objects
function toArray(M) {
  if (Array.isArray(M)) return M;
  if (M && M._data) return M._data;
  return M;
}

// Nelder-Mead optimizer (small implementation)
// - Minimizes objective funct f(x) for x vector of length n
// - Returns {x: best solution array, f: function value at solution}
// - Parameters tuned for small dimensions; this is sufficient for use in condFstat where dimension = K-1 (K small)
// Reference: standard Nelder-Mead heuristic

function nelderMead(f, x0, options = {}) {
  const alpha = options.alpha || 1;
  const gamma = options.gamma || 2;
  const rho = options.rho || 0.5;
  const sigma = options.sigma || 0.5;
  const maxIter = options.maxIter || 500;
  const tol = options.tol || 1e-8;

  const n = x0.length;
  // initialize simplex: n+1 points
  let simplex = [];
  simplex.push({ x: x0.slice(), fx: f(x0) });
  for (let i = 0; i < n; i++) {
    const xi = x0.slice();
    const val = xi[i];
    const step = (Math.abs(val) > 1e-6) ? 0.05 * val : 0.00025;
    xi[i] = xi[i] + step;
    simplex.push({ x: xi, fx: f(xi) });
  }

  let iter = 0;
  while (iter < maxIter) {
    // sort simplex by fx ascending
```

```
        simplex.sort((a, b) => a.fx - b.fx);
        const best = simplex[0], worst = simplex[n], secondWorst = simplex[n - 1];


// compute centroid of all points except worst
const centroid = new Array(n).fill(0);
for (let i = 0; i < n; i++) {
   const xi = simplex[i].x;
   for (let j = 0; j < n; j++) centroid[j] += xi[j];
}
for (let j = 0; j < n; j++) centroid[j] /= n;

// reflection
const xr = centroid.map((c, j) => c + alpha * (c - worst.x[j]));
const fxr = f(xr);
if (fxr < best.fx) {
   // expansion
   const xe = centroid.map((c, j) => c + gamma * (xr[j] - c));
   const fxe = f(xe);
   if (fxe < fxr) {
      simplex[n] = { x: xe, fx: fxe };
   } else {
      simplex[n] = { x: xr, fx: fxr };
   }
} else if (fxr < secondWorst.fx) {
   simplex[n] = { x: xr, fx: fxr };
} else {
   // contraction
   if (fxr < worst.fx) {
      // outside contraction
      const xc = centroid.map((c, j) => c + rho * (xr[j] - c));
      const fxc = f(xc);
      if (fxc <= fxr) simplex[n] = { x: xc, fx: fxc };
      else {
         // shrink
         for (let i = 1; i < simplex.length; i++) {
            const xi = simplex[i].x;
            simplex[i].x = simplex[0].x.map((b, j) => b + sigma * (xi[j] - b));
            simplex[i].fx = f(simplex[i].x);
         }
      }
   } else {
      // inside contraction
      const xc = centroid.map((c, j) => c + rho * (worst.x[j] - c));
      const fxc = f(xc);
```

```
      if (fxc < worst.fx) simplex[n] = { x: xc, fx: fxc };
      else {
        // shrink
        for (let i = 1; i < simplex.length; i++) {
          const xi = simplex[i].x;
          simplex[i].x = simplex[0].x.map((b, j) => b + sigma * (xi[j] - b));
          simplex[i].fx = f(simplex[i].x);
        }
      }
    }
}

// convergence check
const fvals = simplex.map(s => s.fx);
const fmean = fvals.reduce((a, b) => a + b, 0) / fvals.length;
let ss = 0;
for (let k = 0; k < fvals.length; k++) ss += Math.pow(fvals[k] - fmean, 2);
const std = Math.sqrt(ss / fvals.length);
if (std < tol) break;

iter++;

    }

    simplex.sort((a, b) => a.fx - b.fx);
    return { x: simplex[0].x, f: simplex[0].fx };
}

// Confidence interval helpers (normal and t)

function ci_normal(side, estimate, se, alpha) {
    // estimate and se can be arrays or numbers
    const z = Math.abs(jStat.normal.inv(alpha / 2, 0, 1));
    if (Array.isArray(estimate)) {
      return estimate.map((est, i) => (side === 'l' ? est - z * se[i] : est + z * se[i]));
    } else {
      return side === 'l' ? estimate - z * se : estimate + z * se;
    }
}

function ci_t(side, estimate, se, df, alpha) {
    const tcrit = Math.abs(jStat.studentt.inv(alpha / 2, df));
    if (Array.isArray(estimate)) {
      return estimate.map((est, i) => (side === 'l' ? est - tcrit * se[i] : est + tcrit * se[i]));
    } else {
```

```
        return side === 'l' ? estimate - tcrit * se : estimate + tcrit * se;
    }
}

// condFstat implementation

function condFstat(Bx, Bxse, nx, ld, corx) {
    // Inputs:
    // - Bx: J x K matrix (Array of arrays)
    // - Bxse: J x K matrix
    // - nx: array length K (sample sizes for each exposure)
    // - ld: J x J matrix (variant correlation)
    // - corx: K x K matrix (exposure correlation)
    //
    // Returns: array length K with conditional F-statistics
    Bx = toArray(Bx);
    Bxse = toArray(Bxse);
    ld = toArray(ld);
    corx = toArray(corx);

    const J = Bx.length;
    const K = Bx[0].length;

    // ax matrix J x K
    const ax = new Array(J);
    for (let i = 0; i < J; i++) {
        ax[i] = new Array(K);
        for (let k = 0; k < K; k++) {
            const sx = Bxse[i][k];
            ax[i][k] = 1 / ((nx[k] * sx * sx) + (Bx[i][k] * Bx[i][k]));
        }
    }

    // Ax list: each element is JxJ matrix: (sqrt(ax[,k]) %*% t(sqrt(ax[,k]))) * ld
    const Ax = [];
    for (let k = 0; k < K; k++) {
        // vector sqrt(ax[,k])
        const svec = ax.map(row => Math.sqrt(row[k]));
        // outer product
        const outer = new Array(J);
        for (let i = 0; i < J; i++) {
            outer[i] = new Array(J);
            for (let j = 0; j < J; j++) {
                outer[i][j] = svec[i] * svec[j];
            }
```

```
      }
      // element-wise multiply by ld
      const mat = new Array(J);
      for (let i = 0; i < J; i++) {
        mat[i] = new Array(J);
          for (let j = 0; j < J; j++) {
              mat[i][j] = outer[i][j] * ld[i][j];
          }
      }
      Ax.push(mat);
   }

   // BxVec = function(k) ax[,k] * bx[,k]
   const BxVec = new Array(K);
   for (let k = 0; k < K; k++) {
      BxVec[k] = new Array(J);
      for (let i = 0; i < J; i++) {
         BxVec[k][i] = ax[i][k] * Bx[i][k];
      }
   }
   // convert to J x K matrix (columns are exposures)
   const BxMat = new Array(J);
   for (let i = 0; i < J; i++) {
      BxMat[i] = new Array(K);
      for (let k = 0; k < K; k++) BxMat[i][k] = BxVec[k][i];
   }

   // sqrt.Ax: list of JxJ matrix sqrt via eigen
   const sqrtAx = Ax.map(A => matrixSqrt(A));

   // SigX function returning JxJ block between exposures k and l using expression:
   // solve(sqrtAx[k] %% t(sqrtAx[l])) * (corx[k][l] - (t(Bx[,k]) %% solve(sqrtAx[k] %% t(sqrtAx[l])) %% Bx[,l])) *
(1/(sqrt(nx[k]-J+1)sqrt(nx[l]-J+1)))
   function SigXkl(k, l) {
      const S = mm(sqrtAx[k], t(sqrtAx[l])); // JxJ
      const Sinv = inv(S);
      // compute scalar term s2 = corx[k][l] - t(Bx[,k]) %% Sinv %*% Bx[,l]
      const Bk = BxMat.map(r => r[k]); // J vector
      const Bl = BxMat.map(r => r[l]);
      const SinvBl = mm(Sinv, colVec(Bl)); // Jx1
      const tBk_SinvBl = dot(Bk, flattenCol(SinvBl));
      const s2 = corx[k][l] - tBk_SinvBl;
      const factor = 1 / (Math.sqrt(nx[k] - J + 1) * Math.sqrt(nx[l] - J + 1));
      // final matrix
      return math.multiply(Sinv, math.multiply(s2 * factor, math.identity(J)._data));
```

```
}

// SigX2(m): rows of blocks for fixed m: returns (K*J) x J matrix built by stacking SigX(m1,m) for m1 in 1:K
function SigX2(m) {
  const blocks = [];
  for (let m1 = 0; m1 < K; m1++) {
    blocks.push(SigXkl(m1, m)); // each block is JxJ
  }
  // stack vertically
  return blocks.reduce((acc, b, idx) => {
    if (!acc) return b;
    return acc.concat(b);
  }, null);
}

// Build SigX full matrix as KxK blocks each JxJ, flattened to (KJ)x(KJ)
const SigXblocks = [];
for (let m1 = 0; m1 < K; m1++) {
  const rowBlocks = [];
  for (let m2 = 0; m2 < K; m2++) {
    rowBlocks.push(SigXkl(m1, m2));
  }
  // horizontally concatenate rowBlocks (each JxJ) to produce Jx(K*J)
  const rowConc = new Array(J);
  for (let i = 0; i < J; i++) {
    rowConc[i] = [];
    for (let b = 0; b < rowBlocks.length; b++) {
      rowConc[i] = rowConc[i].concat(rowBlocks[b][i]);
    }
  }
  SigXblocks.push(rowConc);
}
// vertically concatenate SigXblocks to final SigX matrix
const SigX = SigXblocks.reduce((acc, r) => (acc ? acc.concat(r) : r), null);

// gamX_est(k) = solve(Ax[[k]]) %*% Bx[,k]    (returns length J vector)
const gamX_est_cols = [];
for (let k = 0; k < K; k++) {
  const Axk = Ax[k];
  const Bxk = BxMat.map(r => r[k]);
  const sol = solve(Axk, colVec(Bxk));
  gamX_est_cols.push(flattenCol(sol));
}
// convert to J x K matrix (columns exposures)
const gamX_est = new Array(J);
```

```
for (let i = 0; i < J; i++) {
   gamX_est[i] = new Array(K);
   for (let k = 0; k < K; k++) gamX_est[i][k] = gamX_est_cols[k][i];
}

// Now compute conditional F-statistics for each exposure j
const condF = new Array(K);

if (K > 2) {
   for (let j = 0; j < K; j++) {
      // Build SigXX by reordering SigX so that block j (J rows) is first
      const idxRows = [];
      const allIdx = Array.from({ length: K * J }, (_, i) => i);
      const blockStart = j * J;
      const blockIdx = Array.from({ length: J }, (_, r) => blockStart + r);
      const remaining = allIdx.filter(i => !blockIdx.includes(i));
      const newOrder = blockIdx.concat(remaining);
      // reorder rows and columns
      const SigXX = newOrder.map(r => newOrder.map(c => SigX[r][c]));



// g(tet) = gamX_est[:,j] - gamX_est[,-j] %*% tet
const gam_j = gamX_est.map(row => row[j]); // length J
// matrix gamX_est[,-j]: J x (K-1)
const gam_omit = gamX_est.map(row => row.filter((_, idx) => idx !== j));

function gOfTet(tet) {
   // tet is array length K-1
   const prod = mm(gam_omit, colVec(tet)); // J x 1
   return gam_j.map((v, i) => v - prod[i][0]);
}

function Qgg(tet) {
   const g = gOfTet(tet);
   // t(g) %*% g
   return dot(g, g);
}

const tet0 = new Array(K - 1).fill(0);
const nm1 = nelderMead(Qgg, tet0, { maxIter: 200, tol: 1e-9 });

function OmNr(tet) {
   // Om.nr = (cbind(I_J, kronecker(t(-tet), I_J)) %*% SigXX %*% t(cbind(I_J, kronecker(t(-tet), I_J))))
   // Build M = cbind(I_J, kronecker(t(-tet), I_J)) : J x (J + (K-1)*J)
   // But we want square final matrix of size J + (K-1)*J
```

```
      const tetNeg = tet.map(x => -x);
      const kronPart = kron([tetNeg], I(J));
      // We'll build kronecker manually: produce J x ((K-1)*J) where block columns are for each element of tetNeg multiplied by I_J
      const kronMat = new Array(J);
      for (let i = 0; i < J; i++) {
        kronMat[i] = [];
        for (let s = 0; s < tetNeg.length; s++) {
          for (let c = 0; c < J; c++) {
            // element = tetNeg[s] * (I_J)[i,c] => tetNeg[s] if i===c else 0
            kronMat[i].push(i === c ? tetNeg[s] : 0);
          }
        }
      }
      // Build cbind(I_J, kronMat)
      const M = new Array(J);
      for (let i = 0; i < J; i++) {
        M[i] = I(J)._data ? I(J)._data[i].concat(kronMat[i]) : math.identity(J)._data[i].concat(kronMat[i]);
      }
      // Now compute M %*% SigXX %*% t(M)
      const left = mm(M, SigXX);
      return mm(left, t(M));
    }

    function Qnr(tet) {
      const g = gOfTet(tet);
      const Om = OmNr(tet);
      const OmInv = inv(Om);
      const tmp = mm(t(colVec(g)), OmInv);
      const res = mm(tmp, colVec(g));
      return res[0][0];
    }

    // Gradient function; we use Nelder-Mead.
    const nm2 = nelderMead(Qnr, nm1.x, { maxIter: 500, tol: 1e-8 });
    condF[j] = nm2.f / (J - K + 1);
  }

    } else if (K === 2) {
      for (let j = 0; j < K; j++) {
        const idxRows = [];
        const allIdx = Array.from({ length: K * J }, (_, i) => i);
        const blockStart = j * J;
        const blockIdx = Array.from({ length: J }, (_, r) => blockStart + r);
        const remaining = allIdx.filter(i => !blockIdx.includes(i));
```

```
const newOrder = blockIdx.concat(remaining);
const SigXX = newOrder.map(r => newOrder.map(c => SigX[r][c]));
const gam_j = gamX_est.map(row => row[j]);
function gOfTet(tet) {
  // tet is array length 1
  const tet0 = Array.isArray(tet) ? tet[0] : tet;
  const prod = gamX_est.map(row => row.filter((_, idx) => idx !== j)[0] * tet0);
  return gam_j.map((v, i) => v - prod[i]);
}
function Qgg(tet) {
  const g = gOfTet(tet);
  return dot(g, g);
}
const tet0 = [0];
const nm1 = nelderMead(Qgg, tet0, { maxIter: 200, tol: 1e-9 });
function OmNr(tet) {
  const tet0 = tet[0];
  const tetNeg = [-tet0];
  // kron(t(-tet), I_J)
  const kronMat = new Array(J);
  for (let i = 0; i < J; i++) {
    kronMat[i] = [];
    for (let s = 0; s < tetNeg.length; s++) {
      for (let c = 0; c < J; c++) {
        kronMat[i].push(i === c ? tetNeg[s] : 0);
      }
    }
  }
  const M = new Array(J);
  for (let i = 0; i < J; i++) {
    M[i] = I(J)._data ? I(J)._data[i].concat(kronMat[i]) : math.identity(J)._data[i].concat(kronMat[i]);
  }
  const left = mm(M, SigXX);
  return mm(left, t(M));
}
function Qnr(tet) {
  const g = gOfTet(tet);
  const Om = OmNr(tet);
  const OmInv = inv(Om);
  const tmp = mm(t(colVec(g)), OmInv);
  const res = mm(tmp, colVec(g));
  return res[0][0];
}
const nm2 = nelderMead(Qnr, nm1.x,{ maxIter: 500, tol: 1e-8 });
condF[j] = nm2.f / (J - K + 1);
```

```
        }
    } else {
        throw new Error("K must be at least 1");
    }

    return condF;
}

// mr_mvivw function

function mr_mvivw(object, opts = {}) {
    // opts: model ("default"/"random"/"fixed"), robust (bool), correl (bool), correl_x (KxK), nx (array or single), distribution ("normal" or "t-dist"), alpha
    const {
        model: modelIn = "default",
        robust: robustIn = false,
        correl: correlIn = false,
        correl_x = null,
        nx = NaN,
        distribution = "normal",
        alpha = 0.05
    } = opts;

    // unpack object
    const Bx = toArray(object.betaX);
    const ByArr = object.betaY.slice ? object.betaY.slice() : object.betaY;
    const Bxse = toArray(object.betaXse);
    const ByseArr = object.betaYse.slice ? object.betaYse.slice() : object.betaYse;
    const rho = toArray(object.correlation);
    const exposure = object.exposure || [];
    const outcome = object.outcome || null;

    const nsnps = Bx.length;
    const K = Bx[0].length;

    // normalize nx: if NaN or missing => array of NaN length K; if single => replicate
    let nxArr;
    if (!nx || (Array.isArray(nx) && nx.every(v => isNaN(v)))) {
        nxArr = new Array(K).fill(NaN);
    } else if (!Array.isArray(nx)) {
        nxArr = new Array(K).fill(nx);
    } else if (nx.length === 1) {
        nxArr = new Array(K).fill(nx[0]);
    } else {
        nxArr = nx.slice();
```

```
    }

    // correl.x fallback to identity
    const corx = correl_x ? toArray(correl_x) : math.identity(K)._data;

    // ld (genetic correlations) fallback to identity
    const ld = (rho && !isNaN(rho.reduce ? rho.reduce((a, b) => a + (Array.isArray(b) ? b.reduce((x, y) => x + y, 0) : b), 0) :
NaN)) ? rho : math.identity(nsnps)._data;

    // compute conditional F-statistics if nx provided
    let condF;
    if (nxArr.some(v => isNaN(v))) {
        condF = new Array(K).fill(NaN);
    } else {
        condF = condFstat(Bx, Bxse, nxArr, ld, corx);
        if (condF.some(v => v < 0)) {
            condF = new Array(K).fill(NaN);
            console.warn("Conditional F statistics did not converge to positive values - should the sample sizes be larger?");
        }
    }

    // set model default
    let model = modelIn;
    if (model === "default") {
        model = (nsnps < 4) ? "fixed" : "random";
    }

    // if rho is present (not identity) set correl true
    let correl = correlIn;
    // detect if rho is all zeros/NaN by checking sum
    try {
        const sumRho = rho.flat ? rho.flat().reduce((a, b) => a + (b || 0), 0) : NaN;
        if (!isNaN(sumRho)) correl = true;
    } catch (e) {
        // ignore
    }

    // Main branches: correlated (using omega) or not correlated (lm weighted)
    if (["random", "fixed"].includes(model) && ["normal", "t-dist"].includes(distribution)) {
        if (correl) {
            // ensure rho provided
            if (!rho) {
                console.warn("Genetic variant correlation matrix not given.");
            } else {
                // turn off robust (R code)
```

```
        const robust = false;
        // omega = Byse %o% Byse * rho    (outer product scaled by rho)
        // Build Byse column and row
        const Byse = ByseArr.slice();
        // Build outer product matrix of Byse (JxJ) where omega[i,j] = Byse[i] * Byse[j] * rho[i][j]
        const omega = new Array(nsnps);
        for (let i = 0; i < nsnps; i++) {
          omega[i] = new Array(nsnps);
          for (let j = 0; j < nsnps; j++) {
            omega[i][j] = Byse[i] * Byse[j] * rho[i][j];
          }
        }


    // compute thetaIVW = solve(t(Bx) %*% solve(omega) %*% Bx) %*% t(Bx) %*% solve(omega) %*% By
    const tBx = t(Bx);
    const omegaInv = inv(omega);
    const left = solve(mm(mm(tBx, omegaInv), Bx), I(K)); // solve(A, I) to get inverse of A
    // but easier: compute A = tBx %*% omegaInv %*% Bx; then theta = inv(A) %*% tBx %*% omegaInv %*% By
    const A = mm(mm(tBx, omegaInv), Bx);
    const rhs = mm(mm(tBx, omegaInv), colVec(ByArr));
    const thetaIVWcol = solve(A, rhs);
    const thetaIVW = flattenCol(thetaIVWcol);

    // residuals rse = By - Bx %*% thetaIVW
    const Bx_theta = mm(Bx, colVec(thetaIVW));
    const rseVec = ByArr.map((v, i) => v - Bx_theta[i][0]);

    // compute thetaIVWse
    let thetaIVWse;
    if (model === "random") {
      // thetaIVWse = sqrt(diag(inv(A))) * max(sqrt(t(rse) %*% solve(omega) %*% rse /(nsnps-K)), 1)
      const invA = inv(A);
      const diagInvA = invA.map((row, i) => row[i]);
      const sqrtDiag = diagInvA.map(v => Math.sqrt(v));
      const rse_col = colVec(rseVec);
      const tmp = mm(mm(t(rse_col), omegaInv), rse_col)[0][0];
      const scale = Math.max(Math.sqrt(tmp / (nsnps - K)), 1);
      thetaIVWse = sqrtDiag.map(v => v * scale);
    } else {
      const invA = inv(A);
      const diagInvA = invA.map((row, i) => row[i]);
      const sqrtDiag = diagInvA.map(v => Math.sqrt(v));
      thetaIVWse = sqrtDiag;
    }
```

```
      const correlation = true;
      // compute CI and p-values
      let ciLower, ciUpper;
      if (distribution === "normal") {
        ciLower = ci_normal("l", thetaIVW, thetaIVWse, alpha);
        ciUpper = ci_normal("u", thetaIVW, thetaIVWse, alpha);
      } else {
        ciLower = ci_t("l", thetaIVW, thetaIVWse, nsnps - K, alpha);
        ciUpper = ci_t("u", thetaIVW, thetaIVWse, nsnps - K, alpha);
      }

      const rse_corr = Math.sqrt(mm(mm(t(colVec(rseVec)), omegaInv), colVec(rseVec))[0][0] / (nsnps - K));
      const heter_stat = (nsnps - K) * (rse_corr * rse_corr);
      const pvalue_heter = 1 - jStat.chisquare.cdf(heter_stat, nsnps - K);

      let pvalue;
      if (distribution === "normal") {
        pvalue = thetaIVW.map((est, i) => 2 * (1 - jStat.normal.cdf(Math.abs(est / thetaIVWse[i]), 0, 1)));
      } else {
        pvalue = thetaIVW.map((est, i) => 2 * (1 - jStat.studentt.cdf(Math.abs(est / thetaIVWse[i]), nsnps - K)));
      }

      return {
        Model: model,
        Exposure: exposure,
        Outcome: outcome,
        Robust: robust,
        Correlation: rho,
        Estimate: thetaIVW,
        StdError: thetaIVWse,
        CILower: ciLower,
        CIUpper: ciUpper,
        SNPs: nsnps,
        Pvalue: pvalue,
        Alpha: alpha,
        RSE: rse_corr,
        HeterStat: [heter_stat, pvalue_heter],
        CondFstat: condF
      };
    }
  } else {
    // not correlated: perform weighted linear regression By ~ Bx - 1 with weights = 1 / Byse^2
    // We implement WLS directly: theta = inv(t(Bx) W Bx) t(Bx) W By
    const w = ByseArr.map(s => 1 / (s * s));
```

```
// form W as diagonal matrix
const W = math.diag(w);
const tBx = t(Bx);
const A = mm(mm(tBx, W._data || W), Bx); // KxK
const rhs = mm(mm(tBx, W._data || W), colVec(ByArr));
let thetaIVWcol = solve(A, rhs);
let thetaIVW = flattenCol(thetaIVWcol);

// Estimate residual variance sigma^2 from residuals
const Bx_theta = mm(Bx, colVec(thetaIVW));
const residuals = ByArr.map((y, i) => y - Bx_theta[i][0]);
// compute weighted residual sum of squares
let wrss = 0;
for (let i = 0; i < residuals.length; i++) wrss += w[i] * residuals[i] * residuals[i];
// sigma^2 estimate is wrss/(nsnps-K)
const sigma2 = wrss / (nsnps - K);
// Here, treat "random" model differently: scale by min(sigma,1) vs 1?
let thetaIVWse;
const invA = inv(A);
const diagInvA = invA.map((row, i) => row[i]);
const baseSE = diagInvA.map(v => Math.sqrt(v));
if (model === "random") {
   const sigma = Math.sqrt(sigma2);
   const scale = Math.min(sigma, 1);
   thetaIVWse = baseSE.map(v => v / scale);
} else {
   const sigma = Math.sqrt(sigma2);
   thetaIVWse = baseSE.map(v => v * sigma);
}

// p-values
let pvalue;
if (distribution === "normal") {
   pvalue = thetaIVW.map((est, i) => 2 * (1 - jStat.normal.cdf(Math.abs(est / thetaIVWse[i]), 0, 1)));
} else {
   pvalue = thetaIVW.map((est, i) => 2 * (1 - jStat.studentt.cdf(Math.abs(est / thetaIVWse[i]), nsnps - K)));
}

const rse = Math.sqrt(sigma2);
const heter_stat = (nsnps - K) * (rse * rse);
const pvalue_heter = 1 - jStat.chisquare.cdf(heter_stat, nsnps - K);

let ciLower, ciUpper;
if (distribution === "normal") {
   ciLower = ci_normal("l", thetaIVW, thetaIVWse, alpha);
```

```
      ciUpper = ci_normal("u", thetaIVW, thetaIVWse, alpha);
    } else {
      ciLower = ci_t("l", thetaIVW, thetaIVWse, nsnps - K, alpha);
      ciUpper = ci_t("u", thetaIVW, thetaIVWse, nsnps - K, alpha);
    }

    return {
      Model: model,
      Exposure: exposure,
      Outcome: outcome,
      Robust: robustIn,
      Correlation: object.correlation,
      Estimate: thetaIVW,
      StdError: thetaIVWse,
      CILower: ciLower,
      CIUpper: ciUpper,
      SNPs: nsnps,
      Pvalue: pvalue,
      Alpha: alpha,
      RSE: rse,
      HeterStat: [heter_stat, pvalue_heter],
      CondFstat: condF
    };
}

    } else {
      throw new Error("Model type must be one of: default, random, fixed. Distribution must be one of: normal, t-dist.");
    }
}
```

(3) The Multivariable Median Method (MVMedian)

The JavaScript codes for The Multivariable Median Method (MVMedian) are as follows:

```
/*
Method name and objective
The Multivariable Median Method (MVMedian)
Objective: estimate the causal effects of multiple exposures on an outcome using summary statistics from many SNP instruments. It uses a weighted regression framework and bootstrap to obtain robust estimates, standard errors, confidence intervals, and P-values for each exposure.

Main classes and functions (high-level)
ci_normal(type, estimate, se, alpha): computes confidence interval bounds under a normal distribution (lower or upper bound).
ci_t(type, estimate, se, df, alpha): computes confidence interval bounds under a t-distribution (uses degrees of freedom; falls back to normal if df is non-positive).
createSeededPRNG(seed): simple seeded pseudo-random number generator for repeatable bootstrap sampling.
```

MRMVInput: class that holds input data for MVMedian (betaX, betaY, betaXse, betaYse, snps, exposure, outcome).
MVMedian: class that holds the results (Exposure, Outcome, Estimate, StdError, CILower, CIUpper, Alpha, Pvalue, SNPs).
rq_substitute_wls(Y, X, weights): weighted least squares solver used as a substitute estimation step (used to obtain MVMedian coefficients from the input data).
mr_mvmedian(object, distribution, alpha, iterations, seed): main function that
computes a point estimate via WLS substitution,
uses bootstrap (iterations) to estimate standard errors,
constructs CIs and P-values based on the chosen distribution (normal or t-dist),
returns an MVMedian result object.

Required inputs and expected outputs

Inputs
object: an MRMVInput instance containing:
betaX: nsnps x K matrix of SNP-to-exposure effects
betaY: length-nsnps vector of SNP-to-outcome effects
betaXse: nsnps x K matrix of SEs for betaX
betaYse: length-nsnps vector of SEs for betaY
snps: array of SNP identifiers
exposure: exposure name
outcome: outcome name
distribution: "normal" or "t-dist" (controls CI/p-value calculation)
alpha: significance level for CIs (e.g., 0.05)
iterations: number of bootstrap iterations (positive to estimate SEs; 0 to skip bootstrapping)
seed: seed for the bootstrap RNG (optional; if omitted, a non-deterministic seed is used)

Outputs
An MVMedian object containing:
Exposure: the exposure name
Outcome: the outcome name
Estimate: array of one coefficient per exposure (length K)
StdError: array of standard errors per exposure (null or NaN if bootstrapping is not performed)
CILower: array of lower CI bounds per exposure
CIUpper: array of upper CI bounds per exposure
Alpha: the significance level used
Pvalue: array of P-values per exposure
SNPs: number of SNPs used (nsnps)

Notes
When iterations > 0, standard errors are obtained by bootstrap: resample betaX, betaY (using their SEs) for many bootstrap runs, re-run the WLS substitute, and compute SEs from the distribution of bootstrap estimates.
If distribution is "t-dist" and degrees of freedom are not positive, CIs/P-values fall back to the normal approximation.
If bootstrapping is disabled (iterations = 0), StdError, CILower, CIUpper, and Pvalue may be null/NaN.
*/

```
// --- Helper Functions (Shared or re-used from other conversions) ---

/**
 * Calculates confidence interval bound for normal distribution.
 * @param {string} type - 'l' for lower, 'u' for upper.
 * @param {number[]} estimate - The estimated coefficients (array).
 * @param {number[]} se - The standard errors of the estimates (array).
 * @param {number} alpha - The significance level (e.g., 0.001 for 99.9% CI).
 * @returns {number[]} An array of CI bounds, corresponding to each estimate.
 */
function ci_normal(type, estimate, se, alpha) {
    const q = jStat.normal.inv(1 - alpha / 2, 0, 1); // Z-score for alpha/2
    return estimate.map((val, i) => {
        if (type === 'l') {
            return val - q * se[i];
        } else if (type === 'u') {
            return val + q * se[i];
        }
        throw new Error("Invalid type for ci_normal. Must be 'l' or 'u'.");
    });
}

/**
 * Calculates confidence interval bound for t-distribution.
 * @param {string} type - 'l' for lower, 'u' for upper.
 * @param {number[]} estimate - The estimated coefficients (array).
 * @param {number[]} se - The standard errors of the estimates (array).
 * @param {number} df - Degrees of freedom.
 * @param {number} alpha - The significance level (e.g., 0.001 for 99.9% CI).
 * @returns {number[]} An array of CI bounds, corresponding to each estimate.
 */
function ci_t(type, estimate, se, df, alpha) {
    if (df <= 0) {
        console.warn("Degrees of freedom for t-distribution is non-positive. Using normal approximation.");
        return ci_normal(type, estimate, se, alpha);
    }
    const q = jStat.studentt.inv(1 - alpha / 2, df); // t-score for alpha/2
    return estimate.map((val, i) => {
        if (type === 'l') {
            return val - q * se[i];
        } else if (type === 'u') {
            return val + q * se[i];
        }
        throw new Error("Invalid type for ci_t. Must be 'l' or 'u'.");
    });
```

```
}

/**
 * Custom PRNG (Pseudo-Random Number Generator) for seeding `jStat.normal.sample`.
 * This is a simple xorshift32 implementation.
 * @param {number} seed - The seed value.
 * @returns {function(): number} A function that returns a random number between 0 (inclusive) and 1 (exclusive).
 */
function createSeededPRNG(seed) {
    let x = seed;
    return function() {
        x ^= x << 13;
        x ^= x >> 17;
        x ^= x << 5;
        // Ensure non-negative result and scale to [0, 1) range
        return ((x < 0 ? (~x + 1) : x) >>> 0) / 0x100000000;
    };
}

/**
 * @class MRMVInput
 * Represents the input data for MRMVMedian.
 * @property {number[][]} betaX - Matrix of exposure effects (nsnps x K).
 * @property {number[]} betaY - Vector of outcome effects (nsnps).
 * @property {number[][]} betaXse - Matrix of standard errors for exposure effects (nsnps x K).
 * @property {number[]} betaYse - Vector of standard errors for outcome effects (nsnps).
 * @property {string[]} snps - Array of SNP names.
 * @property {string} exposure - Exposure name.
 * @property {string} outcome - Outcome name.
 */
class MRMVInput {
    constructor(betaX, betaY, betaXse, betaYse, snps, exposure, outcome) {
        this.betaX = betaX;
        this.betaY = betaY;
        this.betaXse = betaXse;
        this.betaYse = betaYse;
        this.snps = snps;
        this.exposure = exposure;
        this.outcome = outcome;
    }
}

/**
 * @class MVMedian
 * Represents the output of the MRMVMedian analysis.
```

```
 * @property {string} Exposure
 * @property {string} Outcome
 * @property {number[]} Estimate - Estimated coefficients.
 * @property {number[]} StdError - Standard errors of estimates.
 * @property {number[]} CILower - Lower bounds of confidence intervals.
 * @property {number[]} CIUpper - Upper bounds of confidence intervals.
 * @property {number} Alpha - Significance level used.
 * @property {number[]} Pvalue - P-values for estimates.
 * @property {number} SNPs - Number of SNPs used.
 */
class MVMedian {
    constructor(Exposure, Outcome, Estimate, StdError, CILower, CIUpper, Alpha, Pvalue, SNPs) {
        this.Exposure = Exposure;
        this.Outcome = Outcome;
        this.Estimate = Estimate;
        this.StdError = StdError;
        this.CILower = CILower;
        this.CIUpper = CIUpper;
        this.Alpha = Alpha;
        this.Pvalue = Pvalue;
        this.SNPs = SNPs;
    }
}

/**
 *
 *
 * @param {number[]} Y - The dependent variable (vector).
 * @param {number[][]} X - The independent variables (matrix), without intercept.
 * @param {number[]} weights - The weights for each observation.
 * @returns {{coefficients: number[]}} An object containing the estimated coefficients.
 */
function rq_substitute_wls(Y, X, weights) {
    // This implements Weighted Least Squares (WLS) for the formula Y ~ X - 1
    // The formula for WLS coefficients is: beta_hat = (X_w^T * X_w)^-1 * X_w^T * Y_w
    // where X_w = W^(1/2) * X and Y_w = W^(1/2) * Y
    // and W is a diagonal matrix of weights (W is a diagonal matrix where W_ii = weights[i]).

    const n = Y.length; // Number of observations (SNPs)
    const k = X[0].length; // Number of predictors (exposures)

    // Calculate sqrt(weights)
    const sqrt_weights = weights.map(w => Math.sqrt(w));

    // Create weighted Y and X
```

```
    const Y_w = Y.map((val, i) => val * sqrt_weights[i]);
    const X_w = X.map((row, i) => row.map(val => val * sqrt_weights[i]));

    const X_w_T = math.transpose(X_w);
    const X_w_T_X_w = math.multiply(X_w_T, X_w);

    let coefficients;

    try {
        const inv_X_w_T_X_w = math.inv(X_w_T_X_w);
        const X_w_T_Y_w = math.multiply(X_w_T, math.matrix(Y_w).reshape([n, 1]));

        coefficients = math.multiply(inv_X_w_T_X_w, X_w_T_Y_w).valueOf().flat();

    } catch (e) {
        console.error("Error in rq_substitute_wls: Matrix inversion failed. This may indicate multicollinearity or too few observations.", e);
        coefficients = Array(k).fill(NaN);
    }

    return { coefficients: coefficients };
}


// --- Main mr_mvmedian function (JS equivalent) ---

/**
 * Implements the MR MVMedian method for Mendelian Randomization.
 *
 * @param {MRMVInput} object - An MRMVInput object containing betaX, betaY, betaXse, betaYse.
 * @param {string} distribution - The distribution to use for CIs and p-values ("normal" or "t-dist").
 * @param {number} alpha - Significance level for confidence intervals (e.g., 0.001).
 * @param {number} iterations - Number of bootstrap iterations for standard error estimation.
 * @param {number|null} seed - Seed for random number generation. If null/undefined, system random is used.
 * @returns {MVMedian} An MVMedian object containing the results.
 */
function mr_mvmedian(object, distribution = "normal", alpha = 0.001, iterations = 10000, seed = 314159265) {

    // --- Random Seed Management ---
    let randomFunc = Math.random;
    if (seed !== null && !isNaN(seed)) {
        randomFunc = createSeededPRNG(seed);
    }

    // --- Extract Data ---
```

```
const Bx = object.betaX;
const By = object.betaY;
const Bxse = object.betaXse; // Assuming Bxse is a 2D array (nsnps x K).
const Byse = object.betaYse;

const nsnps = Bx.length; // Number of SNPs (rows)
const K = Bx[0].length; // Number of exposures (columns in Bx)

if (!["normal", "t-dist"].includes(distribution)) {
    console.error("Distribution must be one of: 'normal', 't-dist'. See documentation for details.");
    throw new Error("Invalid distribution specified.");
}

// --- Initial Regression (rq equivalent using WLS substitute) ---
const weights = Byse.map(se => 1 / (se * se));
const qr_mod = rq_substitute_wls(By, Bx, weights); // Using WLS as substitute

const thetaMed = qr_mod.coefficients; // This will be the WLS coefficients

// --- Bootstrap for Standard Errors ---
let thetaMedse;
if (iterations > 0) {
    const est_boot_results = [];

    for (let i = 0; i < iterations; i++) {
        const Bxboot = Array(nsnps).fill(0).map((_, snpIdx) => {
            // For each SNP, sample K exposure effects
            return Array(K).fill(0).map((_, expIdx) =>
                jStat.normal.sample(Bx[snpIdx][expIdx], Bxse[snpIdx][expIdx], randomFunc)
            );
        });

        const Byboot = By.map((meanVal, snpIdx) =>
            jStat.normal.sample(meanVal, Byse[snpIdx], randomFunc)
        );

        // Re-run the regression on bootstrapped data using the WLS substitute
        const boot_qr_mod = rq_substitute_wls(Byboot, Bxboot, weights);
        est_boot_results.push(boot_qr_mod.coefficients);
    }

    // Calculate standard deviation of bootstrapped coefficients
    // Convert array of coefficient arrays (iterations x K) to array of arrays of coefficients (K x iterations)
    const transposed_est_boot = math.transpose(est_boot_results); // K x iterations matrix
    thetaMedse = transposed_est_boot.map(row => jStat.stdev(row)); // Calculate sd for each coefficient (row)
```

```
    } else {
        thetaMedse = Array(K).fill(NaN); // No bootstrap, standard errors are not estimated
    }

    // --- Confidence Intervals and P-values ---
    let ciLower = Array(K).fill(NaN);
    let ciUpper = Array(K).fill(NaN);
    let pvalue = Array(K).fill(NaN);

    for (let i = 0; i < K; i++) {
        // Ensure estimate and SE are valid numbers and SE is positive to avoid division by zero
        if (!isNaN(thetaMed[i]) && !isNaN(thetaMedse[i]) && thetaMedse[i] > 0) {
            if (distribution === "normal") {
                ciLower[i] = ci_normal("l", [thetaMed[i]], [thetaMedse[i]], alpha)[0];
                ciUpper[i] = ci_normal("u", [thetaMed[i]], [thetaMedse[i]], alpha)[0];
                // P-value for two-sided test: 2 * P(Z < -|test_stat|)
                pvalue[i] = 2 * jStat.normal.cdf(-Math.abs(thetaMed[i] / thetaMedse[i]), 0, 1);
            } else if (distribution === "t-dist") {
                const df = nsnps - K; // Degrees of freedom for t-distribution
                if (df > 0) {
                    ciLower[i] = ci_t("l", [thetaMed[i]], [thetaMedse[i]], df, alpha)[0];
                    ciUpper[i] = ci_t("u", [thetaMed[i]], [thetaMedse[i]], df, alpha)[0];
                    // P-value for two-sided test: 2 * P(T < -|test_stat|)
                    pvalue[i] = 2 * jStat.studentt.cdf(-Math.abs(thetaMed[i] / thetaMedse[i]), df);
                } else {
                    console.warn(`Degrees of freedom for t-distribution (df=${df}) is not positive for coefficient ${i}.
P-value and CI will be NaN.`);
                }
            }
        }
    }

    // --- Return MVMedian object ---
    return new MVMedian(
        object.exposure,
        object.outcome,
        thetaMed,
        thetaMedse,
        ciLower,
        ciUpper,
        alpha,
        pvalue,
        nsnps
    );
}
```

(4) The Multivariable Lasso Method (MVLasso)

The JavaScript codes for The Multivariable Lasso Method (MVLasso) are as follows:

/*

Method name and objective

Name: The Multivariable Lasso Method (MVLasso)

Objective: Estimate the causal effects of multiple exposures on an outcome using genetic instruments (SNPs). It orients SNP effects, selects valid instruments with a LASSO-based approach, and provides post-selection estimates with confidence intervals and p-values. It is designed to be robust to pleiotropy and heterogeneity across instruments.

Main classes and functions

MRMVInput (class)

A data container that holds the input data needed for analysis:

betaX: SNP effects on exposures (n SNPs × K exposures)

betaY: SNP effects on the outcome

betaXse: standard errors of betaX

betaYse: standard errors of betaY

exposure: list of exposure names (length K)

outcome: name of the outcome

snps: SNP identifiers (length n)

MVLasso (class)

Wrapper object used to store and return the analysis results.

Helper functions (supporting math and statistics)

diag(n), diagFromVector(vec): create diagonal matrices

elementwiseMultiply(matrix, vector): multiply each row by a corresponding scalar

matrixPower(matrix, power): raise each element to a power (used for simple exponent ops)

ci_normal(type, est, se, alpha): compute normal-based confidence intervals

ci_t(type, est, se, df, alpha): compute t-distribution-based confidence intervals

simpleLasso (function)

A simple, coordinate-descent implementation of LASSO (a simplified glmnet-like approach).

Fits a linear model with L1 regularization (lambda). Can run along a lambda path if lambda is not provided.

Returns estimated coefficients for the given design matrix and response.

mr_mvlasso (function)

The main workflow:

Accepts an MRMVInput instance and optional settings (orientation, distribution, alpha, lambda).

Determines the orientation (which exposure to align to) and builds oriented SNP effects.

Constructs a weight matrix S from the standard errors and forms transformed matrices (b, Pb, xlas, ylas) used for LASSO.

Runs LASSO to select instruments (SNPs) that predict the exposures while accounting for a possibly large number of instruments.

Performs post-LASSO estimation (weighted regression on the selected instruments) to obtain causal effect estimates for each exposure.

Computes standard errors, confidence intervals (normal or t-distribution, depending on settings), and p-values.

Identifies the valid instruments actually used in the post-selection step (and reports which SNPs were valid).

Returns an MVLasso object containing: orientation, exposure/outcome info, estimated effects, standard errors, confidence

intervals, p-values, the set of valid SNPs, the full list of SNPs, the LASSO lambda used, and other diagnostic/details.

Required inputs and expected outputs

Required inputs:
An MRMVInput object with:
betaX: array of SNP effects on each exposure (n SNPs by K exposures)
betaY: array of SNP effects on the outcome (length n)
betaXse: standard errors for betaX (n by K)
betaYse: standard errors for betaY (length n)
exposure: list of exposure names (length K)
outcome: outcome name
snps: SNP identifiers (length n)
Optional settings in mr_mvlasso(options):
orientate: which exposure to use as reference for orientation (default 1)
distribution: 'normal' or 't-dist' for CI/p-value calculation
alpha: significance level for CIs (default 0.05)
lambda: optional fixed lambda for LASSO (if not provided, a path is used)

Expected outputs:
An MVLasso object containing:
Orientate, Exposure, Outcome
Estimate: post-LASSO causal effect estimates for each exposure
StdError: standard errors of the estimates
CILower / CIUpper: confidence intervals for each exposure
Pvalue: p-values for the estimates
SNPs: total number of SNPs
Valid: number of valid/instrument SNPs used in the final estimation
ValidSNPs: identifiers of the valid SNPs
RegEstimate / RegIntercept: auxiliary regression estimates used in the post-selection step
Lambda: the lambda used by LASSO (or the path) for instrument selection

In short: you provide summary-level SNP-to-exposure and SNP-to-outcome data, the function selects valid instruments with LASSO, then delivers causal effect estimates with accompanying uncertainty measures and instrument validity information.
*/

```
class MRMVInput {
    constructor(betaX, betaY, betaXse, betaYse, exposure, outcome, snps) {
        this.betaX = betaX;
        this.betaY = betaY;
        this.betaXse = betaXse;
        this.betaYse = betaYse;
        this.exposure = exposure;
        this.outcome = outcome;
        this.snps = snps;
```

```
    }
}

class MVLasso {
    constructor(data) {
        Object.assign(this, data);
    }
}

// Helper functions
function diag(n) {
    return math.identity(n)._data;
}

function diagFromVector(vec) {
    const n = vec.length;
    const result = math.zeros(n, n)._data;
    for (let i = 0; i < n; i++) {
        result[i][i] = vec[i];
    }
    return result;
}

function elementwiseMultiply(matrix, vector) {
    return matrix.map((row, i) => row.map(val => val * vector[i]));
}

function matrixPower(matrix, power) {
    return matrix.map(row => row.map(val => Math.pow(val, power)));
}

function ci_normal(type, est, se, alpha) {
    const z = jStat.normal.inv(1 - alpha / 2, 0, 1);
    return est.map((e, i) => {
        if (isNaN(e) || isNaN(se[i])) return NaN;
        return type === 'l' ? e - z * se[i] : e + z * se[i];
    });
}

function ci_t(type, est, se, df, alpha) {
    const t = jStat.studentt.inv(1 - alpha / 2, df);
    return est.map((e, i) => {
        if (isNaN(e) || isNaN(se[i])) return NaN;
        return type === 'l' ? e - t * se[i] : e + t * se[i];
    });
```

```
}

// Simple LASSO implementation (simplified glmnet)
function simpleLasso(X, y, lambda = null, maxIter = 1000, tol = 1e-7) {
    const n = X.length;
    const p = X[0].length;

    // Standardize X
    const means = Array(p).fill(0).map((_, j) =>
        X.reduce((sum, row) => sum + row[j], 0) / n
    );
    const sds = Array(p).fill(0).map((_, j) => {
        const mean = means[j];
        const variance = X.reduce((sum, row) => sum + Math.pow(row[j] - mean, 2), 0) / n;
        return Math.sqrt(variance);
    });

    const Xs = X.map(row => row.map((val, j) => (val - means[j]) / (sds[j] + 1e-10)));

    // Auto-select lambda if not provided
    let lambdaSeq = [];
    if (lambda === null) {
        const maxLambda = Math.max(...Array(p).fill(0).map((_, j) =>
            Math.abs(Xs.reduce((sum, row, i) => sum + row[j] * y[i], 0)) / n
        ));
        lambdaSeq = Array(100).fill(0).map((_, i) =>
            maxLambda * Math.exp(-i * Math.log(maxLambda / 0.01) / 99)
        );
    } else {
        lambdaSeq = [lambda];
    }

    const results = lambdaSeq.map(lam => {
        let beta = Array(p).fill(0);

        for (let iter = 0; iter < maxIter; iter++) {
            const betaOld = [...beta];

            for (let j = 0; j < p; j++) {
                const residual = y.map((yi, i) =>
                    yi - beta.reduce((sum, b, k) => k !== j ? sum + b * Xs[i][k] : sum, 0)
                );

                const rho = Xs.reduce((sum, row, i) => sum + row[j] * residual[i], 0) / n;
```

```
                if (Math.abs(rho) <= lam) {
                    beta[j] = 0;
                } else {
                    beta[j] = rho > 0 ? rho - lam : rho + lam;
                }
            }

            const diff = Math.sqrt(beta.reduce((sum, b, j) =>
                sum + Math.pow(b - betaOld[j], 2), 0
            ));

            if (diff < tol) break;
        }

        return { beta, lambda: lam };
    });

    return {
        beta: results.map(r => r.beta),
        lambda: lambdaSeq
    };
}

// Main MV-LASSO function
function mr_mvlasso(object, options = {}) {
    const {
        orientate = 1,
        distribution = 'normal',
        alpha = 0.05,
        lambda = null
    } = options;

    const K = object.betaX[0].length;
    const orientAte = (orientate >= 1 && orientate <= K) ? orientate : 1;

    // Orient based on specified exposure
    const orient = object.betaX.map(row => Math.sign(row[orientAte - 1]));
    const Bx = elementwiseMultiply(object.betaX, orient);
    const By = object.betaY.map((val, i) => val * orient[i]);
    const Bxse = object.betaXse;
    const Byse = object.betaYse;
    const nsnps = Bx.length;

    if (!['normal', 't-dist'].includes(distribution)) {
        console.log('Distribution must be one of: normal, t-dist.');
```

```
        return null;
    }

    // Create S matrix (diagonal weight matrix)
    const S = diagFromVector(Byse.map(se => Math.pow(se, -2)));

    // Calculate sqrt(S)
    const Ssqrt = matrixPower(S, 0.5);

    // b = S^(1/2) %*% Bx
    const b = math.multiply(Ssqrt, Bx);

    // Pb = b %*% solve(t(b) %*% b, t(b))
    const btb = math.multiply(math.transpose(b), b);
    const btbInv = math.inv(btb);
    const Pb = math.multiply(b, math.multiply(btbInv, math.transpose(b)));

    // xlas = (I - Pb) %*% S^(1/2)
    const I = diag(nsnps);
    const IminusPb = math.subtract(I, Pb);
    const xlas = math.multiply(IminusPb, Ssqrt);

    // ylas = (I - Pb) %*% S^(1/2) %*% By
    const SsqrtBy = Ssqrt.map((row, i) => row[i] * By[i]);
    const ylas = math.multiply(IminusPb, SsqrtBy);

    // Fit LASSO
    let las_mod;
    if (lambda !== null) {
        const las_fit = simpleLasso(xlas, ylas, lambda);
        las_mod = {
            fit: las_fit.beta[0],
            lambda: lambda
        };
    } else {
        const las_fit = simpleLasso(xlas, ylas);
        const lamseq = las_fit.lambda.slice().sort((a, b) => a - b);
        const lamlen = lamseq.length;

        // Calculate residual standard errors
        const rse = las_fit.beta.slice().reverse().map((beta, j) => {
            const av = beta.map((b, idx) => b === 0 ? idx : -1).filter(idx => idx >= 0);

            if (av.length > K) {
                // Weighted least squares on valid instruments
```

```
                const Bx_valid = av.map(idx => Bx[idx]);
                const By_valid = av.map(idx => By[idx]);
                const weights = av.map(idx => Math.pow(Byse[idx], -2));

                // Simplified regression
                const residuals = By_valid.map((y, i) => {
                    const fitted = Bx_valid[i].reduce((sum, x, k) => sum + x * 0, 0);
                    return y - fitted;
                });

                const rss = residuals.reduce((sum, r, i) => sum + r * r * weights[i], 0);
                const df = av.length - K;
                const rse_val = Math.sqrt(rss / df);

                return [rse_val, av.length];
            }
            return [1, av.length];
        });

        // Select lambda based on heterogeneity test
        const rse_inc = rse.slice(1).map((r, i) => r[0] - rse[i][0]);
        const het = rse_inc.map((inc, i) => {
            const threshold = Math.sqrt(jStat.chisquare.inv(0.95, 1) / rse[i + 1][1]);
            return rse[i + 1][0] > 1 && inc > threshold ? i : -1;
        }).filter(idx => idx >= 0);

        let lam_pos = het.length === 0 ? lamlen - 1 : Math.min(...het);

        const num_valid = rse.map(r => r[1]);
        const min_lam_pos = num_valid.findIndex(nv => nv > K);
        if (lam_pos < min_lam_pos) lam_pos = min_lam_pos;

        las_mod = {
            fit: las_fit.beta[lamlen - lam_pos - 1],
            lambda: lamseq[lam_pos]
        };
    }

    // Post-LASSO estimation
    const a = las_mod.fit;
    const e = By.map((y, i) => y - a[i]);

    // est = solve(t(Bx) %*% S %*% Bx, t(Bx) %*% S %*% e)
    const BxT = math.transpose(Bx);
    const BxTSBx = math.multiply(math.multiply(BxT, S), Bx);
```

```
const BxTSe = math.multiply(math.multiply(BxT, S), e);
const est = math.multiply(math.inv(BxTSBx), BxTSe);

// Find valid instruments
const v = a.map((val, idx) => val === 0 ? idx : -1).filter(idx => idx >= 0);

let post_est, post_se, ciLower, ciUpper, pvalue;

if (v.length > K) {
    // Weighted regression on valid instruments
    const Bx_v = v.map(idx => Bx[idx]);
    const By_v = v.map(idx => By[idx]);
    const weights = v.map(idx => Math.pow(Byse[idx], -2));

    // Simplified weighted least squares
    const XTW = math.transpose(Bx_v).map(row =>
        row.map((val, i) => val * weights[i])
    );
    const XTWX = math.multiply(XTW, Bx_v);
    const XTWy = math.multiply(XTW, By_v);

    post_est = math.multiply(math.inv(XTWX), XTWy);

    // Calculate standard errors
    const fitted = Bx_v.map(row =>
        row.reduce((sum, x, k) => sum + x * post_est[k], 0)
    );
    const residuals = By_v.map((y, i) => y - fitted[i]);
    const rss = residuals.reduce((sum, r, i) => sum + r * r * weights[i], 0);
    const df = v.length - K;
    const sigma = Math.sqrt(rss / df);
    const sigma_constrained = Math.min(sigma, 1);

    const XTWXinv = math.inv(XTWX);
    post_se = Array(K).fill(0).map((_, k) =>
        Math.sqrt(XTWXinv[k][k]) * sigma_constrained
    );

    // Confidence intervals
    if (distribution === 'normal') {
        ciLower = ci_normal('l', post_est, post_se, alpha);
        ciUpper = ci_normal('u', post_est, post_se, alpha);
        pvalue = post_est.map((e, i) =>
            2 * jStat.normal.cdf(-Math.abs(e / post_se[i]), 0, 1)
        );
```

```
        } else {
            ciLower = ci_t('l', post_est, post_se, df, alpha);
            ciUpper = ci_t('u', post_est, post_se, df, alpha);
            pvalue = post_est.map((e, i) =>
                2 * jStat.studentt.cdf(-Math.abs(e / post_se[i]), df)
            );
        }
    } else {
        post_est = Array(K).fill(NaN);
        post_se = Array(K).fill(NaN);
        ciLower = Array(K).fill(NaN);
        ciUpper = Array(K).fill(NaN);
        pvalue = Array(K).fill(NaN);

        const msg = v.length === K
            ? 'Same number of valid instruments as risk factors.'
            : 'Fewer valid instruments than risk factors.';
        console.log(`${msg} Post-lasso method cannot be performed.`);
    }

    return new MVLasso({
        Orientate: orientate,
        Exposure: object.exposure,
        Outcome: object.outcome,
        Estimate: post_est,
        StdError: post_se,
        CILower: ciLower,
        CIUpper: ciUpper,
        Alpha: alpha,
        Pvalue: pvalue,
        SNPs: nsnps,
        RegEstimate: est,
        RegIntercept: a,
        Valid: v.length,
        ValidSNPs: v.map(idx => object.snps[idx]),
        Lambda: las_mod.lambda
    });
}
```

(5) The Multivariable Constrained Maximum Likelihood Method (MVcML)

The JavaScript codes for The Multivariable Constrained Maximum Likelihood Method (MVcML) are as follows:

```
/*
Method name and objective
```

The Multivariable Constrained Maximum Likelihood Method (MVcML)

Objective: estimate the causal effects of multiple exposures on an outcome using summary statistics from many SNPs, while allowing for a few SNPs to be invalid (constrained by a model and validated via information criteria). It can optionally use data perturbation (DP) to obtain more robust uncertainty estimates.

Main classes and functions (overview)

numericHessian(f, x0, eps): computes the Hessian matrix of a scalar function f at x0 using central finite differences.

pl(x, b_exp_v, b_out_v, Sig_inv_v): the profile likelihood (up to a constant) for a candidate theta given per-SNP exposure/outcome effects and their inverse-covariance matrices.

invcov_mvmr(se_bx, se_by, rho_mat): builds, for each SNP, the (L+1)x(L+1) inverse covariance matrix using per-SNP standard errors and a correlation matrix rho_mat.

MVcML_SdTheta(b_exp, b_out, Sig_inv_l, theta, zero_ind, r_vec): estimates standard errors for theta by evaluating the Hessian of the profile likelihood on the set of valid SNPs and inverting it.

MVmr_cML(opts): a simplified constrained maximum-likelihood estimator that uses a BIC-like criterion to select a set of valid SNPs (and estimate theta) for a given K (number of allowed invalid instruments).

MVmr_cML_DP(opts): data-perturbation wrapper that repeatedly perturbs the data, runs MVmr_cML, and aggregates results to produce DP-based estimates and uncertainties.

mr_mvcML(object, n, opts): the main wrapper that ties everything together. It builds inverse-covariance matrices, runs MVmr_cML (non-DP) to get theta and valid SNPs, computes standard errors via MVcML_SdTheta, then derives confidence intervals and P-values. If DP is enabled, it runs MVmr_cML_DP and reports DP-based estimates and uncertainty (with additional DP-specific outputs).

Required inputs and expected outputs

Required inputs (via the wrapper):

object: an MRMV-like input with

betaX: m x L matrix of SNP effects on exposures

betaY: m-length vector of SNP effects on outcome

betaXse: m x L matrix of SEs for betaX

betaYse: m-length vector of SEs for betaY

exposure: (optional) name of exposure

outcome: (optional) name of outcome

n: sample size (scalar)

opts (optional):

DP: boolean, enable data perturbation (default true)

rho_mat: (L+1)x(L+1) correlation matrix among exposures + outcome (default identity)

K_vec: array of candidate numbers of invalid instruments (default 0..ceil(m/2))

random_start: number of random starts (default 0)

num_pert: number of perturbations for DP (default 200)

min_theta_range, max_theta_range: range for random starts

maxit: max iterations for underlying optimizers

alpha: significance level for CIs/P-values

seed: optional seed for reproducibility

Expected outputs:

An object describing MVcML results, including:
Exposure: name or null
Outcome: name or null
DP: boolean indicating if DP was used
Estimate: theta, an L-length vector of causal effects
StdError: L-length vector of standard errors
CILower: L-length lower confidence bounds
CIUpper: L-length upper confidence bounds
Alpha: significance level used
Pvalue: L-length vector of P-values for each theta
BIC_invalid: array of invalid-SNP indices (1-based) selected by the non-DP path
K_hat: selected number of invalid SNPs (non-DP) or DP_ninvalid (DP path)
SNPs: total number of SNPs used
If DP is enabled: additional fields like BIC_DP_theta, BIC_DP_se, BIC_DP_invalid, DP_ninvalid, eff_DP_B, etc., summarizing perturbation results

In short, MVcML provides a constrained multivariable maximum-likelihood framework with optional data perturbation, delivering point estimates, uncertainty, and instrument validity choices for multiple exposures.
*/

```
// ---------- Utility: numeric Hessian (central finite differences) ----------
function numericHessian(f, x0, eps = 1e-6) {
   // f: function mapping an array to scalar
   // x0: array
   const n = x0.length;
   const H = math.zeros(n, n)._data;
   const fx0 = f(x0);
   for (let i = 0; i < n; i++) {
      for (let j = i; j < n; j++) {
         const xi = x0.slice();
         const xj = x0.slice();
         const xij_p = x0.slice();
         const xij_m = x0.slice();


   // step sizes scaled by magnitude
   const hi = eps * Math.max(1, Math.abs(x0[i]));
   const hj = eps * Math.max(1, Math.abs(x0[j]));

   xi[i] += hi;
   xj[j] += hj;

   xij_p[i] += hi;
   xij_p[j] += hj;
```

```
    xij_m[i] -= hi;
    xij_m[j] -= hj;

    const f_pp = f(xij_p);
    const f_pm = f(xi.slice().map((v, idx) => (idx === j ? v - hj : v)));
    const f_mp = f(xj.slice().map((v, idx) => (idx === i ? v - hi : v)));
    const f_mm = f(xij_m);

    // second mixed partial (central)
    const val = (f_pp - f_pm - f_mp + f_mm) / (4 * hi * hj);
    H[i][j] = val;
    H[j][i] = val;
}

    }
    return math.matrix(H);
}

// ---------- pl: profile likelihood (up to constant) ----------
// Inputs:
// - x: array (length L) candidate theta vector (column vector as JS array)
// - b_exp_v: m x L matrix (JS array of arrays) of SNP effects on L exposures for the valid SNPs only
// - b_out_v: m-length array of SNP effects on outcome for valid SNPs only
// - Sig_inv_v: array of m inverse covariance matrices (each (L+1)x(L+1), math.js matrix or compatible)
// Returns numeric scalar: -pll (R code returns -pll)
function pl(x, b_exp_v, b_out_v, Sig_inv_v) {
    // Convert x to column math.js matrix
    const L = x.length;
    const m_valid = b_out_v.length;
    let pll = 0;
    for (let i = 0; i < m_valid; i++) {
        // W is inverse covariance for SNP i: (L+1) x (L+1)
        const W = math.matrix(Sig_inv_v[i]);


// b_exp_i is length L (array), b_out_i is scalar
const b_exp_i = b_exp_v[i]; // array length L
const b_out_i = b_out_v[i];

// beta = c(b_exp_i, b_out_i) length L+1 as column vector
const beta = math.matrix(math.reshape(b_exp_i.concat([b_out_i]), [L + 1, 1])); // (L+1)x1

// Build B = W[-(k+1),] %*% beta + (W[(k+1),] %*% beta) * x
// In JS: indices 0..L-1 correspond to exposures, index L corresponds to outcome
const W_arr = W; // math.matrix
```

```
const W_data = W_arr.valueOf();
// Extract subblocks:
// W_11: rows 0..L-1, cols 0..L-1    => (L x L)
const W_11 = math.matrix(W_data.slice(0, L).map(row => row.slice(0, L)));
// W_12: rows 0..L-1, col L => (L x 1)
const W_12 = math.matrix(W_data.slice(0, L).map(row => [row[L]]));
// W_21: row L, cols 0..L-1 => (1 x L)
const W_21 = math.matrix([W_data[L].slice(0, L)]);
// W_22: scalar
const W_22 = W_data[L][L];

// beta as column vector (L+1 x 1); we need to compute W * beta
const Wbeta = math.multiply(W, beta); // (L+1)x1
// Extract Wbeta_1 = first L entries, Wbeta_2 = last entry
const Wbeta_data = Wbeta.valueOf().map(r => (Array.isArray(r) ? r[0] : r));
const Wbeta_1 = math.matrix(math.reshape(Wbeta_data.slice(0, L), [L, 1])); // Lx1
const Wbeta_2 = Wbeta_data[L]; // scalar

// B = Wbeta_1 + Wbeta_2 * x (x as column)
const x_col = math.matrix(math.reshape(x.slice(), [L, 1]));
const B = math.add(Wbeta_1, math.multiply(Wbeta_2, x_col)); // Lx1

// A = W_11 + W_12 * t(x) + x * t(W_21) + W_22 * x * t(x)
// Note: W_12 is Lx1, W_21 is 1xL, x is Lx1
const term1 = W_11;
const term2 = math.multiply(W_12, math.transpose(x_col)); // LxL
const term3 = math.multiply(x_col, W_21); // LxL
const term4 = math.multiply(W_22, math.multiply(x_col, math.transpose(x_col))); // LxL
const A = math.add(math.add(term1, term2), math.add(term3, term4)); // LxL

// Solve bhat_xi = solve(A) %*% B
// Use math.lusolve or pseudo-inverse
let bhat_xi;
try {
  bhat_xi = math.lusolve(A, B); // Lx1
} catch (e) {
  // fallback to pseudo-inverse
  const A_pinv = math.pinv(A);
  bhat_xi = math.multiply(A_pinv, B);
}

// b_i = c(bhat_xi, t(bhat_xi) %*% x)
const bhat_vec = bhat_xi.valueOf().map(r => (Array.isArray(r) ? r[0] : r)); // length L
const bhat_xi_col = math.matrix(math.reshape(bhat_vec, [L, 1]));
const bhat_xi_t_x = math.multiply(math.transpose(bhat_xi_col), x_col).valueOf();
```

```
const bhat_xi_t_x_scalar = Array.isArray(bhat_xi_t_x) ? (Array.isArray(bhat_xi_t_x[0]) ? bhat_xi_t_x[0][0] : bhat_xi_t_x[0]) : bhat_xi_t_x;
const b_i = math.matrix(math.reshape(bhat_vec.concat([bhat_xi_t_x_scalar]), [L + 1, 1])); // (L+1)x1

// pll = pll    -1/2 * t(b_i) %*% W %*% b_i + t(beta) %*% W %*% b_i
const termA = math.multiply(math.transpose(b_i), math.multiply(W, b_i)).valueOf();
const termB = math.multiply(math.transpose(beta), math.multiply(W, b_i)).valueOf();

// Extract scalars safely
function extractScalar(m) {
   if (typeof m === "number") return m;
   if (Array.isArray(m)) {
      // could be [[num]] or [num]
      if (Array.isArray(m[0])) return m[0][0];
      return m[0];
   }
   if (m && typeof m.valueOf === "function") {
      const v = m.valueOf();
      if (Array.isArray(v)) return Array.isArray(v[0]) ? v[0][0] : v[0];
   }
   return Number(m);
}
const t1 = -0.5 * extractScalar(termA);
const t2 = extractScalar(termB);
pll += t1 + t2;

   }

   return -pll;
}

// ---------- invcov_mvmr: build list of inverse covariance matrices per SNP ----------
// Inputs:
// - se_bx: m x L array (JS array of arrays) where each row i is vector of standard errors for exposures
// - se_by: m-length array of standard errors for outcome (one per SNP)
// - rho_mat: (L+1) x (L+1) correlation matrix (math.js matrix or 2d array)
// Returns: array of length m where each element is (L+1)x(L+1) inverse covariance (math.matrix)
function invcov_mvmr(se_bx, se_by, rho_mat) {
   const m = se_bx.length;
   if (!Array.isArray(se_by) || se_by.length !== m) {
      throw new Error("se_by must be an array of length m (rows of se_bx)");
   }

   // ensure rho_mat is a numeric 2d array of size (L+1)x(L+1)
   const rho = math.matrix(rho_mat).valueOf();
```

```
    const Lplus1 = rho.length;
    const L = Lplus1 - 1;

    const Sig_inv_l = new Array(m);
    for (let i = 0; i < m; i++) {
        const row_se_bx = se_bx[i];
        if (row_se_bx.length !== L) throw new Error("Each row of se_bx must have length L");
        // build vector s = c(se_bx[i,], se_by[i])
        const s = row_se_bx.concat([se_by[i]]); // length L+1
        // construct Sig = rho * crossprod(t(s))
        // compute outer product s s^T
        const outer = [];
        for (let r = 0; r < Lplus1; r++) {
            const row = [];
            for (let c = 0; c < Lplus1; c++) {
                row.push(s[r] * s[c]);
            }
            outer.push(row);
        }
        // elementwise multiply rho and outer
        const Sig = [];
        for (let r = 0; r < Lplus1; r++) {
            const row = [];
            for (let c = 0; c < Lplus1; c++) {
                row.push(rho[r][c] * outer[r][c]);
            }
            Sig.push(row);
        }
        // invert Sig (use math.inv)
        let Sig_inv;
        try {
            Sig_inv = math.inv(math.matrix(Sig));
        } catch (e) {
            // fallback: pseudo-inverse
            Sig_inv = math.pinv(math.matrix(Sig));
        }
        Sig_inv_l[i] = Sig_inv;
    }
    return Sig_inv_l;
}

// ---------- MVcML_SdTheta: SE estimate via Hessian of profile likelihood ----------
// Inputs:
// - b_exp: m x L array of exposures for all SNPs
// - b_out: m-length array of outcome betas
```

```
// - Sig_inv_l: array length m of inverse covariance matrices (L+1 x L+1 math.matrix)
// - theta: L-length array (final estimate of causal effects)
// - zero_ind: array of indices of valid IVs -> convert to 0-based JS indices
// - r_vec: optional vector where zeros indicate valid ivs (not used if zero_ind provided)
function MVcML_SdTheta(b_exp, b_out, Sig_inv_l, theta, zero_ind, r_vec = null) {
  // zero_ind might be 1-based indices. We will accept either 0-based or 1-based.
  let valid_idx = [];
  if (r_vec !== null) {
    // derive zero_ind from r_vec
    for (let i = 0; i < r_vec.length; i++) if (r_vec[i] === 0) valid_idx.push(i);
  } else {
    // zero_ind provided: convert to zero-based if > 0
    zero_ind.forEach(v => {
      if (v === null || v === undefined) return;
      const idx = (v > 0 ? v - 1 : v); // if 1-based convert
      valid_idx.push(idx);
    });
  }

  const m_valid = valid_idx.length;
  const L = theta.length;
  if (m_valid === 0) throw new Error("No valid IVs provided to MVcML_SdTheta");

  // build b_exp_v (m_valid x L), b_out_v (m_valid) and Sig_inv_v (m_valid)
  const b_exp_v = [];
  const b_out_v = [];
  const Sig_inv_v = [];
  for (let ii = 0; ii < m_valid; ii++) {
    const i = valid_idx[ii];
    b_exp_v.push(b_exp[i]);
    b_out_v.push(b_out[i]);
    Sig_inv_v.push(Sig_inv_l[i]);
  }

  // define wrapper function for pl
  const f = function(xArr) {
    // ensure xArr is plain JS array
    return pl(xArr, b_exp_v, b_out_v, Sig_inv_v);
  };

  // compute Hessian numerically at theta
  const Hmat = numericHessian(f, theta, 1e-6); // math.matrix
  // compute inverse of H, then sqrt(diagonal)
  let H_inv;
  try {
```

```
        H_inv = math.inv(Hmat);
    } catch (e) {
        H_inv = math.pinv(Hmat);
    }
    const diag = [];
    const Hinv_data = H_inv.valueOf();
    for (let i = 0; i < L; i++) diag.push(Math.sqrt(Math.max(0, Hinv_data[i][i])));
    return diag;
}

// ---------- A simplified MVmr_cML implementation (heuristic) ----------
// Purpose: find theta (L-length) and invalid SNPs via BIC selection over K_vec
// Inputs:
// - b_exp: m x L array
// - b_out: m x 1 array (or m-length)
// - se_bx: m x L array (not used in complex ways here)
// - Sig_inv_l: list of m inverse covariance matrices (L+1 x L+1)
// - n: sample size (scalar). This function will use it for likelihood scaling and BIC
// - K_vec: array of candidate K values (numbers of invalid instruments to allow)
// - random_start: integer number of random starts (unused in this simplified version except attempt multiple tries)
// - min_theta_range, max_theta_range: ranges for initial theta sampling (not heavily used here)
// - maxit, thres: optimization controls (ignored, but present for compatibility)
// Returns object containing: BIC_theta (L-array), BIC_invalid (array of 1-based indices), Khat, Converge (0 ok)
function MVmr_cML(opts) {
    const {
        b_exp, b_out, se_bx, Sig_inv_l, n, K_vec = [0], random_start = 1,
        min_theta_range = -0.5, max_theta_range = 0.5, maxit = 100, thres = 1e-4
    } = opts;

    const m = b_out.length;
    const L = b_exp[0].length;

    // Helper: compute profile likelihood at given theta across all SNPs (sum of per-SNP contributions)
    function profileLikAll(thetaArr) {
        const b_exp_v = b_exp.slice();
        const b_out_v = b_out.slice();
        const Sig_inv_v = Sig_inv_l.slice();
        return pl(thetaArr, b_exp_v, b_out_v, Sig_inv_v);
    }

    // Heuristic to select invalid SNPs for a given K:
    // For a candidate theta, compute per-SNP "profile residual" defined by squared error between observed (b_out) and predicted given exposures & theta.
    // Rank SNPs by magnitude and pick top K as invalid.
    function selectInvalidsByResiduals(thetaArr, K) {
```

```
    const pred = [];
    for (let i = 0; i < m; i++) {
      // predicted outcome effect = b_exp_i %*% theta
      const be = b_exp[i];
      let dot = 0;
      for (let j = 0; j < L; j++) dot += be[j] * thetaArr[j];
      pred.push(dot);
    }
    const residuals = [];
    for (let i = 0; i < m; i++) {
      const r = Math.abs(b_out[i] - pred[i]);
      residuals.push({ idx: i, r });
    }
    residuals.sort((a, b) => b.r - a.r); // descending largest residual first
    const invalid = residuals.slice(0, Math.min(K, m)).map(o => o.idx);
    return invalid;
  }

  // Estimation of theta given a set of valid SNPs: maximize profile likelihood w.r.t theta
  // We'll use simple numerical optimization (Nelder-Mead via math.js is not provided; implement very small NM)
  function estimateThetaGivenValid(valid_idx, theta_init = null) {
    const b_exp_v = valid_idx.map(i => b_exp[i]);
    const b_out_v = valid_idx.map(i => b_out[i]);
    const Sig_inv_v = valid_idx.map(i => Sig_inv_l[i]);


// define objective = pl(theta, b_exp_v, b_out_v, Sig_inv_v)
function obj(thetaArr) {
  return pl(thetaArr, b_exp_v, b_out_v, Sig_inv_v);
}

// Use a simple gradient-free local minimizer: Nelder-Mead (small implementation)
function nelderMead(f, x0, opts = {}) {
  const maxIter = opts.maxIter || 200;
  const alpha = 1, gamma = 2, rho = 0.5, sigma = 0.5;
  const n = x0.length;
  // initialize simplex
  let simplex = [x0.slice()];
  const step = opts.step || 1e-2;
  for (let i = 0; i < n; i++) {
    const xi = x0.slice();
    xi[i] += step;
    simplex.push(xi);
  }
  const fvals = simplex.map(v => f(v));
```

```
  for (let iter = 0; iter < maxIter; iter++) {
    // sort simplex by f value
    const order = Array.from({ length: simplex.length }, (_, i) => i);
    order.sort((a, b) => fvals[a] - fvals[b]);
    simplex = order.map(i => simplex[i]);
    const fs = order.map(i => fvals[i]);
    const x_best = simplex[0], f_best = fs[0];
    const x_worst = simplex[n], f_worst = fs[n];
    const x_second = simplex[n - 1];

    // centroid of all but worst
    let x_cent = new Array(n).fill(0);
    for (let i = 0; i < n; i++) {
      const xi = simplex[i];
      for (let j = 0; j < n; j++) x_cent[j] += xi[j];
    }
    for (let j = 0; j < n; j++) x_cent[j] /= n;

    // reflection
    const xr = x_cent.map((c, j) => c + alpha * (c - x_worst[j]));
    const fr = f(xr);
    if (fr < f_best) {
      // expansion
      const xe = x_cent.map((c, j) => c + gamma * (xr[j] - c));
      const fe = f(xe);
      if (fe < fr) {
        simplex[n] = xe;
        fvals[n] = fe;
      } else {
        simplex[n] = xr;
        fvals[n] = fr;
      }
    } else if (fr < fs[n - 1]) {
      simplex[n] = xr;
      fvals[n] = fr;
    } else {
      // contraction
      let xc;
      if (fr < f_worst) {
        // outside contraction
        xc = x_cent.map((c, j) => c + rho * (xr[j] - c));
      } else {
        // inside contraction
        xc = x_cent.map((c, j) => c + rho * (x_worst[j] - c));
      }
```

```
            const fc = f(xc);
            if (fc < f_worst) {
                simplex[n] = xc;
                fvals[n] = fc;
            } else {
                // shrink
                for (let i = 1; i < simplex.length; i++) {
                    simplex[i] = simplex[0].map((b, j) => b + sigma * (simplex[i][j] - b));
                    fvals[i] = f(simplex[i]);
                }
            }
        }
        // check convergence by spread
        const fMean = fvals.reduce((s, v) => s + v, 0) / fvals.length;
        const ss = fvals.reduce((s, v) => s + Math.pow(v - fMean, 2), 0) / fvals.length;
        if (Math.sqrt(ss) < 1e-6) break;
    }
    // return best
    const bestIdx = fvals.indexOf(Math.min(...fvals));
    return { x: simplex[bestIdx], fx: fvals[bestIdx], convergence: 0 };
}

// initial theta guess:
let theta0 = theta_init;
if (!theta0) {
    // use simple linear regression (least squares) of b_out ~ b_exp (no intercept)
    // Solve (X^T X) theta = X^T y, where X: m_valid x L
    const X = math.matrix(b_exp_v);
    const Xt = math.transpose(X);
    const XtX = math.multiply(Xt, X);
    const Xty = math.multiply(Xt, math.matrix(math.reshape(b_out_v.slice(), [b_out_v.length, 1])));
    try {
        const theta_sol = math.lusolve(XtX, Xty); // Lx1
        theta0 = theta_sol.valueOf().map(r => (Array.isArray(r) ? r[0] : r));
    } catch (e) {
        // fallback small vector
        theta0 = new Array(L).fill(0.01);
    }
}

// run Nelder-Mead to minimize pl
const resNM = nelderMead(obj, theta0, { maxIter: 200, step: 1e-3 });
return { theta: resNM.x, obj: resNM.fx, convergence: resNM.convergence };

    }
```

```
// Iterate over candidate K values, find best via BIC
let bestBIC = Number.POSITIVE_INFINITY;
let bestResult = null;
let Khat = 0;
for (let k_i = 0; k_i < K_vec.length; k_i++) {
   const K = K_vec[k_i];
   // initialize theta guess randomly or zero
   let theta_init = new Array(L).fill(0);
   for (let r = 0; r < random_start; r++) {
      for (let j = 0; j < L; j++) theta_init[j] = min_theta_range + Math.random() * (max_theta_range - min_theta_range);


// select invalids with heuristic
const invalid_idx = selectInvalidsByResiduals(theta_init, K);
const valid_idx = [];
for (let i = 0; i < m; i++) if (!invalid_idx.includes(i)) valid_idx.push(i);

if (valid_idx.length < L) {
   // Insufficient valid SNPs to estimate L parameters -> skip
   continue;
}

// estimate theta given valid
const est = estimateThetaGivenValid(valid_idx, theta_init);
const theta_hat = est.theta;

// compute model fit (pl) using valid SNPs
const b_exp_v2 = valid_idx.map(i => b_exp[i]);
const b_out_v2 = valid_idx.map(i => b_out[i]);
const Sig_inv_v2 = valid_idx.map(i => Sig_inv_l[i]);
const lik = pl(theta_hat, b_exp_v2, b_out_v2, Sig_inv_v2); // scalar

// compute BIC: here treat -2*loglike + p*log(m) where p = L + K (parameters for theta and invalid biases)
// lik returned is -pll, use it as deviance approximation
const p = L + K;
const BIC = 2 * lik + p * Math.log(m);

if (BIC < bestBIC) {
   bestBIC = BIC;
   bestResult = {
      BIC_theta: theta_hat,
      BIC_invalid: invalid_idx.map(ii => ii + 1), // return 1-based indices
      Khat: K,
      Converge: est.convergence
```

```
      };
      Khat = K;
    }
}

    }

    if (!bestResult) {
      // fallback: return zeros
      return {
        BIC_theta: new Array(L).fill(0),
        BIC_invalid: [],
        Khat: 0,
        Converge: 1
      };
    }
    return bestResult;
}

// ---------- MVmr_cML_DP: data perturbation wrapper (simplified) ----------
// Inputs similar to MVmr_cML; additional num_pert for number of perturbations.
// Approach:
// - For each perturbation b (1..num_pert):
//       * perturb b_exp and b_out by drawing from normal with sd from se_bx and se_by
//       * run MVmr_cML on perturbed data
//       * collect BIC_theta and BIC_invalid
// - Aggregate: compute median of theta components across perturbations that converged;
// - Compute se as empirical sd across perturbed thetas
function MVmr_cML_DP(opts) {
    const {
      b_exp, b_out, se_bx, Sig_inv_l, n,
      K_vec = [0], random_start = 1, num_pert = 200,
      min_theta_range = -0.5, max_theta_range = 0.5, maxit = 100, thres = 1e-4
    } = opts;

    const m = b_out.length;
    const L = b_exp[0].length;

    // Helper to draw normal N(mean, sd) using jStat
    function rnorm_scalar(mean, sd) {
      return jStat.normal.sample(mean, sd);
    }

    const thetas = [];
    const invalids_list = [];
```

```
    let eff_DP_B = 0;
    let conv_count = 0;

    for (let b = 0; b < num_pert; b++) {
      // perturb each SNP's b_exp and b_out by sampling normal(mean = observed, sd from the se derived from Sig_inv_l)
      // We don't have direct se arrays here (se_bx etc), but opts includes se_bx in original call; if not present we use small noise
      const pert_b_exp = [];
      const pert_b_out = [];
      // If se_bx and se_by not directly passed, we derive per-SNP diagonal elements from Sig_inv_l (approx)
      for (let i = 0; i < m; i++) {
        // Estimate per-parameter variances by inverting Sig_inv_l[i]
        let Sig_i;
        try {
          Sig_i = math.inv(Sig_inv_l[i]);
        } catch (e) {
          // if inversion fails, attempt pseudo-inverse or use small variances
          try {
            Sig_i = math.pinv(Sig_inv_l[i]);
          } catch (e2) {
            Sig_i = math.identity(L + 1);
          }
        }
        const Sig_data = Sig_i.valueOf();
        // Extract diagonal variances: first L for exposures, last for outcome
        const sd_exp = [];
        for (let j = 0; j < L; j++) sd_exp.push(Math.sqrt(Math.abs(Sig_data[j][j]) || 1e-8));
        const sd_out = Math.sqrt(Math.abs(Sig_data[L][L]) || 1e-8);


      // sample perturbed SNP effects
      const row_be = [];
      for (let j = 0; j < L; j++) row_be.push(jStat.normal.sample(b_exp[i][j], sd_exp[j]));
      pert_b_exp.push(row_be);
      pert_b_out.push(jStat.normal.sample(b_out[i], sd_out));
    }

    // compute Sig_inv_l for perturbed data: we reuse original Sig_inv_l since correlation and sds are approximated
    // Call MVmr_cML on perturbed data
    const res = MVmr_cML({
      b_exp: pert_b_exp,
      b_out: pert_b_out,
      se_bx: se_bx,
      Sig_inv_l: Sig_inv_l,
      n: n,
      K_vec: K_vec,
```

```
    random_start: random_start,
    min_theta_range: min_theta_range,
    max_theta_range: max_theta_range,
    maxit: maxit,
    thres: thres
});

if (res && res.Converge === 0) {
    eff_DP_B++;
    conv_count++;
    thetas.push(res.BIC_theta);
    invalids_list.push(res.BIC_invalid);
}

    }

    // Aggregate results:
    if (thetas.length === 0) {
        // No converged perturbations
        // Return the base-case MVmr_cML on original data
        const baseRes = MVmr_cML({
            b_exp: b_exp,
            b_out: b_out,
            se_bx: se_bx,
            Sig_inv_l: Sig_inv_l,
            n: n,
            K_vec: K_vec,
            random_start: random_start,
            min_theta_range: min_theta_range,
            max_theta_range: max_theta_range,
            maxit: maxit,
            thres: thres
        });
        return {
            BIC_DP_theta: baseRes.BIC_theta,
            BIC_DP_se: new Array(L).fill(NaN),
            BIC_invalid: baseRes.BIC_invalid,
            DP_ninvalid: baseRes.Khat,
            eff_DP_B: eff_DP_B
        };
    }

    // compute componentwise median and sd across thetas
    const thetas_matrix = thetas; // array of arrays
    const B = thetas_matrix.length;
```

```
    const theta_med = new Array(L).fill(0);
    const theta_sd = new Array(L).fill(0);
    for (let j = 0; j < L; j++) {
      const col = thetas_matrix.map(row => row[j]).sort((a, b) => a - b);
      const mid = Math.floor(col.length / 2);
      theta_med[j] = (col.length % 2 === 1) ? col[mid] : (col[mid - 1] + col[mid]) / 2;
      // sd
      const meanCol = col.reduce((s, v) => s + v, 0) / col.length;
      theta_sd[j] = Math.sqrt(col.reduce((s, v) => s + (v - meanCol) * (v - meanCol), 0) / Math.max(1, col.length - 1));
    }

    // For BIC_invalid and DP_ninvalid produce aggregated results: choose most frequent invalid set or median K
    // Here we pick the invalid set from a majority of perturbations (mode)
    const invalid_map = {};
    invalids_list.forEach(inv => {
      const key = inv.join(",");
      invalid_map[key] = (invalid_map[key] || 0) + 1;
    });
    const mode_key = Object.keys(invalid_map).reduce((a, b) => invalid_map[a] > invalid_map[b] ? a : b);
    const BIC_invalid = mode_key === "" ? [] : mode_key.split(",").map(s => s === "" ? [] : Number(s));

    // DP_ninvalid: take the median number of invalid SNPs observed
    const K_vals = invalids_list.map(inv => inv.length).sort((a, b) => a - b);
    const Kmid = Math.floor(K_vals.length / 2);
    const DP_ninvalid = K_vals.length % 2 === 1 ? K_vals[Kmid] : (K_vals[Kmid - 1] + K_vals[Kmid]) / 2;

    return {
      BIC_DP_theta: theta_med,
      BIC_DP_se: theta_sd,
      BIC_invalid: BIC_invalid,
      DP_ninvalid: DP_ninvalid,
      eff_DP_B: eff_DP_B
    };
}

// ---------- mr_mvcML wrapper ----------
// Parameters:
// - object: MRMVInput-like object with properties:
//         betaX: m x L array, betaY: m-length array, betaXse: m x L array, betaYse: m-length array,
//         exposure: optional, outcome: optional
// - n: sample size (scalar)    [required for some BIC-like computations]
// - DP: boolean to enable data perturbation (default TRUE)
// - rho_mat: (L+1)x(L+1) correlation matrix among exposures + outcome
// - K_vec: array of candidate K values (defaults to 0..ceil(m/2))
// - random_start: integer (0 means single default start)
```

```
// - num_pert: number of perturbations (for DP mode)
// - min_theta_range, max_theta_range: range for random starts
// - maxit: iterations (passed to underlying functions)
// - alpha: significance level
// - seed: integer seed for reproducibility (optional)
//
// Returns an object describing MVMRcML.
function mr_mvcML(object, n, opts = {}) {
  // Default options
  const DP = opts.DP === undefined ? true : opts.DP;
  const betaX = object.betaX;
  const betaY = object.betaY;
  const betaXse = object.betaXse;
  const betaYse = object.betaYse;
  const m = betaX.length;
  const L = betaX[0].length;
  const default_rho = math.identity(L + 1).valueOf();
  const rho_mat = opts.rho_mat === undefined ? default_rho : opts.rho_mat;
  const K_vec = opts.K_vec === undefined ? Array.from({ length: Math.ceil(m / 2) + 1 }, (_, i) => i) : opts.K_vec;
  const random_start = opts.random_start === undefined ? 0 : opts.random_start;
  const num_pert = opts.num_pert === undefined ? 200 : opts.num_pert;
  const min_theta_range = opts.min_theta_range === undefined ? -0.5 : opts.min_theta_range;
  const max_theta_range = opts.max_theta_range === undefined ? 0.5 : opts.max_theta_range;
  const maxit = opts.maxit === undefined ? 100 : opts.maxit;
  const alpha = opts.alpha === undefined ? 0.001 : opts.alpha;
  const seed = opts.seed === undefined ? null : opts.seed;

  // Optional seeding for reproducibility: patch Math.random if seed provided
  let rngOld = null;
  if (seed !== null && seed !== undefined && !Number.isNaN(seed)) {
    // create a seeded RNG using mulberry32
    function mulberry32(seedInt) {
      let t = seedInt >>> 0;
      return function() {
        t += 0x6D2B79F5;
  let r = Math.imul(t ^ (t >>> 15), 1 | t);
        r ^= r + Math.imul(r ^ (r >>> 7), 61 | r);
        return ((r ^ (r >>> 14)) >>> 0) / 4294967296;
      };
    }
    rngOld = Math.random;
    const rng = mulberry32(Math.floor(seed) >>> 0);
    Math.random = rng;
    // jStat.normal.sample uses its own RNG, but jStat uses Math.random internally by default in many builds; this helps.
  }
```

```
    try {
        // Build Sig_inv_l
        const Sig_inv_l = invcov_mvmr(betaXse, betaYse, rho_mat);


// If no data perturbation: call MVmr_cML and compute SE via MVcML_SdTheta
if (!DP) {
    const res = MVmr_cML({
        b_exp: betaX,
        b_out: betaY,
        se_bx: betaXse,
        Sig_inv_l: Sig_inv_l,
        n: n,
        K_vec: K_vec,
        random_start: random_start + 1,
        min_theta_range: min_theta_range,
        max_theta_range: max_theta_range,
        maxit: maxit,
        thres: 1e-4
    });
    const theta = res.BIC_theta;
    const nsnps = m;
    // zero indices are complement of invalids
    const invalid_zero_based = res.BIC_invalid.map(v => v - 1);
    const valid_idx = [];
    for (let i = 0; i < nsnps; i++) if (!invalid_zero_based.includes(i)) valid_idx.push(i);

    const se_theta = MVcML_SdTheta(betaX, betaY, Sig_inv_l, theta, valid_idx.map(v => v + 1)); // pass 1-based indices

    // pvalue and CI
    // For L>1, theta is vector. We compute componentwise z and p-values and CI
    const pvalue = theta.map((t, j) => 2 * (1 - jStat.normal.cdf(Math.abs(t / se_theta[j]), 0, 1)));
    const ciLower = theta.map((t, j) => t - jStat.normal.inv(1 - alpha / 2, 0, 1) * se_theta[j]);
    const ciUpper = theta.map((t, j) => t + jStat.normal.inv(1 - alpha / 2, 0, 1) * se_theta[j]);

    if (res.Converge === 1) {
        throw new Error("MVMRcML-BIC failed to converge, please try using multiple random starting points (random_start) or
increasing number of iterations (maxit).");
    }

    return {
        Exposure: object.exposure || null,
        Outcome: object.outcome || null,
        DP: false,
```

```
      Estimate: theta,
      StdError: se_theta,
      CILower: ciLower,
      CIUpper: ciUpper,
      Alpha: alpha,
      Pvalue: pvalue,
      BIC_invalid: res.BIC_invalid,
      K_hat: res.Khat,
      SNPs: nsnps
    };
}

// With data perturbation
const resDP = MVmr_cML_DP({
    b_exp: betaX,
    b_out: betaY,
    se_bx: betaXse,
    Sig_inv_l: Sig_inv_l,
    n: n,
    K_vec: K_vec,
    random_start: random_start + 1,
    num_pert: num_pert,
    min_theta_range: min_theta_range,
    max_theta_range: max_theta_range,
    maxit: maxit,
    thres: 1e-4
});

const theta = resDP.BIC_DP_theta;
const se_theta = resDP.BIC_DP_se;
const pvalue = theta.map((t, j) => 2 * (1 - jStat.normal.cdf(Math.abs(t / se_theta[j] || 0), 0, 1)));
const ciLower = theta.map((t, j) => t - jStat.normal.inv(1 - alpha / 2, 0, 1) * se_theta[j]);
const ciUpper = theta.map((t, j) => t + jStat.normal.inv(1 - alpha / 2, 0, 1) * se_theta[j]);

if (resDP.eff_DP_B < 100) {
    console.warn("The number of data perturbation is too small, results may not be stable. num_pert >= 100 is recommended.");
}
if (resDP.eff_DP_B < num_pert / 2) {
    console.warn("Less than half of the perturbations converged, results may not be stable. You may try increasing the number of
random starting points (random_start).");
}

return {
    Exposure: object.exposure || null,
    Outcome: object.outcome || null,
```

```
    DP: true,
    Estimate: theta,
    StdError: se_theta,
    CILower: ciLower,
    CIUpper: ciUpper,
    Alpha: alpha,
    Pvalue: pvalue,
    BIC_invalid: resDP.BIC_invalid,
    K_hat: resDP.DP_ninvalid,
    eff_DP_B: resDP.eff_DP_B,
    SNPs: m
};

    } finally {
        // restore Math.random
        if (rngOld !== null) Math.random = rngOld;
    }
}
```

(6) The Multivariable Generalized Method of Moments (MVGMM)

The JavaScript codes for The Multivariable Generalized Method of Moments (MVGMM) are as follows:

```
/*
Method name and objective
Method: The Multivariable Generalized Method of Moments (MVGMM)
Objective: Perform a multivariable generalized method of moments Mendelian randomization (MR) analysis to estimate the causal effect of an exposure on an outcome using multiple genetic variants (SNPs). It supports two-sample or one-sample designs and can incorporate linkage disequilibrium (LD) between SNPs. It can also provide robust variance estimates.

Main components and what they do
Deep copy utility
deepCopy(obj): makes a deep copy of an object (to avoid mutating inputs).
Core function
mr_mvgmm(params): the main analysis function. It:
Validates inputs and sets defaults (e.g., robustness, alpha).
Prepares required data from inputs (by, bx, sy, sx, and optional LD matrix Rxy).
Computes outcome-related quantities (ay, Ay, By, gamY_est, SigY) and exposure-related quantities (ax, Ax, Bx, gamX_est).
Builds an initial causal estimate (theta) and then refines it with a Newton-Raphson optimization to solve for the causal parameter (theta).
Computes the variance of the estimate (seTheta) and a Q-statistic for heterogeneity.
Optionally performs a robust variance adjustment:
Estimates a dispersion parameter (kappa), redefines the variance structure, and re-optimizes theta.
Recomputes a robust standard error and Q-statistic.
Produces final results including point estimate and uncertainty, confidence intervals, p-values, heterogeneity metrics, and overdispersion.
```

Output structure

The function returns an object with key results (see Inputs/Outputs below) and, in some environments, may be exported for use as a module.

Example blocks

The code includes multiple example usages (Examples 1–8) demonstrating:

Basic two-sample MR with and without LD

One-sample MR

Large numbers of SNPs

Non-robust vs robust variance

Custom confidence levels

Weak instruments

Complex LD structures

A helper at the end displays detailed results for one example.

Environment support

Thecode is wrapped in a modular wrapper:

Exports via module.exports (CommonJS) for Node

Defines an AMD module if define is present

Otherwise exposes a browser-global function this.mr_mvgmm

It relies on external libraries available globally:

math.js (for matrix operations)

jStat (for statistical functions)

Required inputs and expected outputs

Required inputs (params object)

by: Array of outcome associations (length p)

bx: Array of exposure associations (length p)

sy: Array of outcome standard errors (length p)

sx: Array of exposure standard errors (length p)

Either:

n: single-sample size (one-sample MR), or

nx and ny: exposure/outcome sample sizes (two-sample MR)

Optional:

Rxy: p x p LD matrix (if not provided, identity is used)

robust: boolean (default true) to use robust variance

alpha: significance level (default 0.05)

Outputs (object with results)

Estimate: causal effect estimate (theta)

StdError: standard error of the estimate

CILower, CIUpper: confidence interval bounds

Alpha: significance level used

Pvalue: p-value for the causal estimate

Qstat: heterogeneity Q-statistic

Heter: heterogeneity p-value

Fstat: conditional F-statistic (instrument strength)
Overdispersion: dispersion parameter (kappa if robust, otherwise phi)
(Additional fields may appear, such as Left/Right CI format in some outputs)

In short, this is a JavaScript implementation of a Mendelian randomization analysis using a multivariable generalized method of moments approach, with optional LD, two-sample or one-sample designs, and optional robust variance. It validates inputs, computes intermediate matrices, estimates a causal effect, quantifies uncertainty and heterogeneity, and can output detailed results for multiple example scenarios.
*/

```
(function () {
  "use strict";

  if (typeof math === "undefined") throw new Error("math.js is required (global math).");
  if (typeof jStat === "undefined") throw new Error("jStat is required (global jStat).");

  // deep copy
  function deepCopy(obj) {
    return JSON.parse(JSON.stringify(obj));
  }

  /**
   * mr_mvgmm: Multivariable Generalized Method of Moments for Mendelian Randomization
   * @param {Object} params - Parameters object
   * @param {Array} params.by - Outcome associations
   * @param {Array} params.bx - Exposure associations
   * @param {Array} params.sy - Outcome standard errors
   * @param {Array} params.sx - Exposure standard errors
   * @param {number} [params.ny] - Outcome sample size (two-sample)
   * @param {number} [params.nx] - Exposure sample size (two-sample)
   * @param {number} [params.n] - Sample size (one-sample)
   * @param {Array} [params.Rxy] - LD matrix
   * @param {boolean} [params.robust=true] - Use robust variance
   * @param {number} [params.alpha=0.05] - Significance level
   */
  function mr_mvgmm(params) {
    // Validate params object
    if (!params || typeof params !== 'object') {
      throw new Error("mr_mvgmm requires a parameters object");
    }

    // Extract parameters with validation
    var by = params.by;
    var bx = params.bx;
    var sy = params.sy;
```

```
    var sx = params.sx;
    var ny = params.ny;
    var nx = params.nx;
    var n = params.n;
    var Rxy = params.Rxy;
    var robust = params.robust !== undefined ? params.robust : true;
    var alpha = params.alpha !== undefined ? params.alpha : 0.05;

    // Validate required parameters
    if (!by || !Array.isArray(by)) {
        throw new Error("by (outcome associations) must be an array");
    }
    if (!bx || !Array.isArray(bx)) {
        throw new Error("bx (exposure associations) must be an array");
    }
    if (!sy || !Array.isArray(sy)) {
        throw new Error("sy (outcome standard errors) must be an array");
    }
    if (!sx || !Array.isArray(sx)) {
        throw new Error("sx (exposure standard errors) must be an array");
    }

    var p = bx.length;

    // Validate array lengths
    if (by.length !== p || sy.length !== p || sx.length !== p) {
        throw new Error("by, bx, sy, sx must have the same length");
    }

    // Validate sample size input
    if (n !== undefined) {
        if (nx !== undefined || ny !== undefined) {
            throw new Error("Provide either n (one-sample) or nx/ny (two-sample), not both");
        }
        nx = n;
        ny = n;
    } else {
        if (nx === undefined || ny === undefined) {
            throw new Error("Must provide either n or both nx and ny");
        }
    }

    // Default Rxy to identity if not provided
    if (!Rxy) {
        Rxy = math.identity(p)._data;
```

```
} else {
   Rxy = deepCopy(Rxy);
   if (Rxy.length !== p || Rxy[0].length !== p) {
      throw new Error("Rxy must be p x p matrix");
   }
}

// ========================================================================
// OUTCOME QUANTITIES OF INTEREST (following R code exactly)
// ========================================================================

// ay <- 1/((ny*sy^2)+by^2)
var ay = Array(p);
for (var i = 0; i < p; i++) {
   ay[i] = 1 / ((ny * sy[i] * sy[i]) + (by[i] * by[i]));
}

// Ay <- (sqrt(ay)%*%t(sqrt(ay)))*ld
var sqrtAy = ay.map(function(v) { return Math.sqrt(v); });
var Ay = Array(p);
for (var i = 0; i < p; i++) {
   Ay[i] = Array(p);
   for (var j = 0; j < p; j++) {
      Ay[i][j] = sqrtAy[i] * sqrtAy[j] * Rxy[i][j];
   }
}

// By <- ay*by
var By = Array(p);
for (var i = 0; i < p; i++) {
   By[i] = ay[i] * by[i];
}

var AyInv = math.inv(Ay);

// gamY_est <- solve(Ay)%*%By
var gamY_est = math.multiply(AyInv, By);
if (gamY_est._data) gamY_est = gamY_est._data;

// SigY <- solve(Ay)*(1-t(By)%*%solve(Ay)%*%By); SigY <- SigY*(1/(ny-p+1))
var ByT_AyInv_By = 0;
for (var i = 0; i < p; i++) {
   for (var j = 0; j < p; j++) {
      ByT_AyInv_By += By[i] * AyInv[j][i] * By[j];
   }
```

```
}
var SigY = math.multiply(AyInv, (1 - ByT_AyInv_By));
SigY = math.multiply(SigY, 1 / (ny - p + 1));
if (SigY._data) SigY = SigY._data;

// ========================================================================
// EXPOSURE QUANTITIES OF INTEREST
// ========================================================================

// ax <- 1/((nx*sx^2)+bx^2)
var ax = Array(p);
for (var i = 0; i < p; i++) {
   ax[i] = 1 / ((nx * sx[i] * sx[i]) + (bx[i] * bx[i]));
}

// Ax <- (sqrt(ax)%*%t(sqrt(ax)))*ld
var sqrtAx = ax.map(function(v) { return Math.sqrt(v); });
var Ax = Array(p);
for (var i = 0; i < p; i++) {
   Ax[i] = Array(p);
   for (var j = 0; j < p; j++) {
      Ax[i][j] = sqrtAx[i] * sqrtAx[j] * Rxy[i][j];
   }
}

// Bx <- ax*bx
var Bx = Array(p);
for (var i = 0; i < p; i++) {
   Bx[i] = ax[i] * bx[i];
}

var AxInv = math.inv(Ax);

// gamX_est <- solve(Ax)%*%Bx
var gamX_est = math.multiply(AxInv, Bx);
if (gamX_est._data) gamX_est = gamX_est._data;

// ========================================================================
// CONDITIONAL F-STATISTIC (simplified - full version requires optimization)
// ========================================================================

// BxT_AxInv_Bx
var condF = 0;
for (var i = 0; i < p; i++) {
   for (var j = 0; j < p; j++) {
```

```
        condF += Bx[i] * AxInv[j][i] * Bx[j];
    }
}
condF = condF / (p - 1);

// ============================================================================
// NON-ROBUST LIML ESTIMATE (initial estimate)
// ============================================================================

// g(tet) = gamY_est - gamX_est*tet
// Initial estimate using simple ratio
var thetaInit = 0;
var sumNum = 0, sumDen = 0;
for (var i = 0; i < p; i++) {
    sumNum += gamY_est[i] * gamX_est[i];
    sumDen += gamX_est[i] * gamX_est[i];
}
thetaInit = sumNum / sumDen;

// Optimization using simple Newton-Raphson
var theta = thetaInit;
var maxIter = 100;
var tol = 1e-8;

for (var iter = 0; iter < maxIter; iter++) {
    // g = gamY_est - gamX_est*theta
    var g = Array(p);
    for (var i = 0; i < p; i++) {
        g[i] = gamY_est[i] - gamX_est[i] * theta;
    }

    // Om = SigY + theta^2 * SigX (simplified - should use full SigX)
    // For single exposure, SigX approximation
    var Om = Array(p);
    for (var i = 0; i < p; i++) {
        Om[i] = Array(p);
        for (var j = 0; j < p; j++) {
            var sigXij = AxInv[i][j] * (1 - Bx[i] * AxInv[i][j] * Bx[j]) / (nx - p + 1);
            Om[i][j] = SigY[i][j] + theta * theta * sigXij;
        }
    }

    var OmInv = math.inv(Om);

    // Q = t(g) %*% solve(Om) %*% g
```

```
    var OmInv_g = math.multiply(OmInv, g);
    if (OmInv_g._data) OmInv_g = OmInv_g._data;

    var Q = 0;
    for (var i = 0; i < p; i++) {
      Q += g[i] * OmInv_g[i];
    }

    // Gradient: DQ = -2 * t(gamX_est) %*% solve(Om) %*% g
    var grad = 0;
    for (var i = 0; i < p; i++) {
      grad += -2 * gamX_est[i] * OmInv_g[i];
    }

    // Hessian approximation
    var hess = 0;
    for (var i = 0; i < p; i++) {
      for (var j = 0; j < p; j++) {
        hess += 2 * gamX_est[i] * OmInv[i][j] * gamX_est[j];
      }
    }

    var step = grad / hess;
    theta = theta - step;

    if (Math.abs(step) < tol) break;
  }

  // ============================================================================
  // VARIANCE CALCULATION
  // ============================================================================

  // Recalculate at final theta
  var g = Array(p);
  for (var i = 0; i < p; i++) {
    g[i] = gamY_est[i] - gamX_est[i] * theta;
  }

  var Om = Array(p);
  for (var i = 0; i < p; i++) {
    Om[i] = Array(p);
    for (var j = 0; j < p; j++) {
      var sigXij = AxInv[i][j] * (1 - Bx[i] * AxInv[i][j] * Bx[j]) / (nx - p + 1);
      Om[i][j] = SigY[i][j] + theta * theta * sigXij;
    }
```

```
}

var OmInv = math.inv(Om);

// var.liml <- solve(t(gamX_est) %*% solve(Om) %*% gamX_est)
var varTheta = 0;
for (var i = 0; i < p; i++) {
   for (var j = 0; j < p; j++) {
      varTheta += gamX_est[i] * OmInv[i][j] * gamX_est[j];
   }
}
varTheta = 1 / varTheta;

var seTheta = Math.sqrt(varTheta);

// Q statistic
var OmInv_g = math.multiply(OmInv, g);
if (OmInv_g._data) OmInv_g = OmInv_g._data;

var Qstat = 0;
for (var i = 0; i < p; i++) {
   Qstat += g[i] * OmInv_g[i];
}

// ============================================================================
// ROBUST ESTIMATION (if requested)
// ============================================================================

var kappaEst = 0;
var thetaRobust = theta;
var seThetaRobust = seTheta;

if (robust && p > 1) {
   // Estimate overdispersion parameter kappa
   // Q.kap <- function(kappa){t(g)%*%solve(Om(theta,kappa))%*%g - (p-1)}

   // Search for kappa that makes Q = p-1
   var targetQ = p - 1;
   var kappaSearch = [];
   var qValues = [];

   for (var kap = -10; kap <= 50; kap += 0.5) {
      var OmKappa = Array(p);
      for (var i = 0; i < p; i++) {
         OmKappa[i] = Array(p);
```

```
      for (var j = 0; j < p; j++) {
        var sigXij = AxInv[i][j] * (1 - Bx[i] * AxInv[i][j] * Bx[j]) / (nx - p + 1);
        var kappaAdj = (i === j) ? kap / ny : 0;
        OmKappa[i][j] = SigY[i][j] + kappaAdj + theta * theta * sigXij;
      }
    }

    try {
      var OmKappaInv = math.inv(OmKappa);
      var OmKappaInv_g = math.multiply(OmKappaInv, g);
      if (OmKappaInv_g._data) OmKappaInv_g = OmKappaInv_g._data;

      var qKap = 0;
      for (var i = 0; i < p; i++) {
        qKap += g[i] * OmKappaInv_g[i];
      }

      kappaSearch.push(kap);
      qValues.push(qKap - targetQ);
    } catch (e) {
      // Singular matrix, skip
    }
  }

  // Find root
  kappaEst = 0;
  for (var i = 0; i < qValues.length - 1; i++) {
    if (qValues[i] * qValues[i + 1] < 0) {
      // Sign change - linear interpolation
      kappaEst = kappaSearch[i] - qValues[i] * (kappaSearch[i + 1] - kappaSearch[i]) / (qValues[i + 1] - qValues[i]);
      break;
    }
  }

  kappaEst = Math.max(0, kappaEst);

  // Re-estimate with robust variance
  var OmRobust = Array(p);
  for (var i = 0; i < p; i++) {
    OmRobust[i] = Array(p);
    for (var j = 0; j < p; j++) {
      var sigXij = AxInv[i][j] * (1 - Bx[i] * AxInv[i][j] * Bx[j]) / (nx - p + 1);
      var kappaAdj = (i === j) ? kappaEst / ny : 0;
      OmRobust[i][j] = SigY[i][j] + kappaAdj + theta * theta * sigXij;
    }
```

```
}

var OmRobustInv = math.inv(OmRobust);

// Re-optimize theta with robust variance
for (var iter = 0; iter < maxIter; iter++) {
  var g = Array(p);
  for (var i = 0; i < p; i++) {
    g[i] = gamY_est[i] - gamX_est[i] * thetaRobust;
  }

  var OmRobustInv_g = math.multiply(OmRobustInv, g);
  if (OmRobustInv_g._data) OmRobustInv_g = OmRobustInv_g._data;

  var grad = 0;
  for (var i = 0; i < p; i++) {
    grad += -2 * gamX_est[i] * OmRobustInv_g[i];
  }

  var hess = 0;
  for (var i = 0; i < p; i++) {
    for (var j = 0; j < p; j++) {
      hess += 2 * gamX_est[i] * OmRobustInv[i][j] * gamX_est[j];
    }
  }

  var step = grad / hess;
  thetaRobust = thetaRobust - step;

  if (Math.abs(step) < tol) break;
}

// Robust variance
var varThetaRobust = 0;
for (var i = 0; i < p; i++) {
  for (var j = 0; j < p; j++) {
    varThetaRobust += gamX_est[i] * OmRobustInv[i][j] * gamX_est[j];
  }
}
varThetaRobust = 1 / varThetaRobust;
seThetaRobust = Math.sqrt(varThetaRobust);

// Update Q statistic
var gRobust = Array(p);
for (var i = 0; i < p; i++) {
```

```
            gRobust[i] = gamY_est[i] - gamX_est[i] * thetaRobust;
        }
        var OmRobustInv_gRobust = math.multiply(OmRobustInv, gRobust);
        if (OmRobustInv_gRobust._data) OmRobustInv_gRobust = OmRobustInv_gRobust._data;

        Qstat = 0;
        for (var i = 0; i < p; i++) {
            Qstat += gRobust[i] * OmRobustInv_gRobust[i];
        }
    }

    // ============================================================================
    // FINAL RESULTS
    // ============================================================================

    var thetaFinal = robust ? thetaRobust : theta;
    var seFinal = robust ? seThetaRobust : seTheta;

    // Confidence intervals
    var z = jStat.normal.inv(1 - alpha / 2, 0, 1);
    var ciLower = thetaFinal - z * seFinal;
    var ciUpper = thetaFinal + z * seFinal;

    // P-value
    var zStat = thetaFinal / seFinal;
    var pvalue = 2 * (1 - jStat.normal.cdf(Math.abs(zStat), 0, 1));

    // Heterogeneity test
    var dfQ = p - 1;
    var heter = dfQ > 0 ? (1 - jStat.chisquare.cdf(Qstat, dfQ)) : null;

    // Overdispersion
    var phi = dfQ > 0 ? Math.max(1, Qstat / dfQ) : null;

    return {
        Estimate: thetaFinal,
        StdError: seFinal,
        CILower: ciLower,
        CIUpper: ciUpper,
        Alpha: alpha,
        Pvalue: pvalue,
        Qstat: Qstat,
        Heter: heter,
        Fstat: condF,
        Overdispersion: robust ? kappaEst : phi
```

```
    };
  }

  // Export for different environments
  if (typeof module !== "undefined" && module.exports) {
    module.exports = { mr_mvgmm: mr_mvgmm };
  } else if (typeof define === "function" && define.amd) {
    define([], function () {
      return { mr_mvgmm: mr_mvgmm };
    });
  } else {
    // Browser global
    this.mr_mvgmm = mr_mvgmm;
  }
}.call(this));
```

(7) The Multivariable Principal Component Generalized Method of Moments (MVPCGMM)

The JavaScript codes for The Multivariable Principal Component Generalized Method of Moments (MVPCGMM) are as follows:

```
/*
Method name and objective
Name: The Multivariable Principal Component Generalized Method of Moments (MVPCGMM)
Objective: It performs a multivariate Mendelian randomization analysis with multiple exposures. It estimates causal effects of several exposures on an outcome, with options for robustness, principal components (PCs) handling, and optional exposure correlation structures. It also computes various statistics like standard errors, confidence intervals, and heterogeneity.

Main classes and functions
Matrix utilities (a collection of small functions):
zeros, eye, diagFromVector: create basic matrices (all zeros, identity, diagonal from a vector).
cloneMatrix, transpose, diagFromVector, dotVec: support matrix/vector operations.
matVecMultiply, matMul, matInv, matAdd, matScale: basic matrix algebra (multiply, inverse, add, scale).
subMatrix, cbind, rbind: extract submatrices and combine matrices by columns or rows.
kronecker: compute the Kronecker product of two matrices.
vecToMatrixCol, colMatToVec, flattenMatrixCol: convert between vectors and single-column matrices.
flatten and specific utilities like norm2Vec, quadrant helpers used in the math steps.
Linear algebra helpers:
eigenJacobi: computes eigenvalues and eigenvectors of a symmetric matrix (Jacobi method).
prcompSymmetric: performs a simple eigen-decomposition-based PCA on a symmetric matrix and returns standard deviations, rotation, and eigenvalues.
Simple math helpers:
dot product, vector addition/subtraction, scalar-vector multiplication, norm.
uniroot: a numeric root-finding method (bisection-like) to find zeros.
Statistical helpers:
pnorm, qnorm: normal CDF and inverse CDF (via a library like jStat).
```

pchisq, zPvalue, ci_normal: chi-square CDF, z-score p-value, and normal-based confidence intervals.
Main analysis function (mr_mvpcgmm):
Accepts a structured object with summary statistics per SNP/exposure, LD matrix, sample sizes, and options like the number of PCs, robustness flag, and alpha level.
Builds a Phi matrix from exposure effects and the LD matrix, performs PCA to select the number of principal components, and then carries out a series of matrix operations to estimate exposure-outcome effects (gamY_est, gamX_est), variances, and confidence intervals.
Supports two modes for robustness (robust true/false) and optional PC-based reduction (pc_check) with conditional F-statistics.
Produces a packed result with fields like estimates, standard errors, confidence intervals, P-values, PCs used, and optional heterogeneity/overdispersion statistics, depending on the chosen mode.

Required inputs and expected outputs

Required inputs (via the mr_mvpcgmm call):
object: A data object containing:
betaX: matrix of SNP-exposure effects (rows = SNPs, cols = exposures)
betaY: array of SNP-outcome effects
betaXse: matrix of standard errors for betaX
betaYse: array of standard errors for betaY
correlation: LD-like matrix (SNP-SNP correlations)
exposure: optional labels for exposures
outcome: name/label for the outcome
nx: array of exposure sample sizes
ny: sample size for the outcome
corx: optional exposure correlation matrix (or null to auto-create)
r (optional): number of principal components to use (or null to auto-select)
thres (optional): threshold for variance explained to determine r
robust (optional): boolean to perform robust estimation
alpha (optional): significance level for CIs

Expected outputs:
An object with summary results, typically including:
Estimate: estimated causal effects of exposures on the outcome
StdError: standard errors of the estimates
Pvalue: p-values for the estimates
PCs: number of principal components used
CondFstat: conditional F-statistics (when applicable)
CILower, CIUpper: confidence interval bounds for the estimates
Exposure/Outcome/Correlation fields from inputs
Overdispersion: estimated overdispersion parameter (if robust mode computes it)
HeterStat: heterogeneity statistics (for non-robust mode)
ExpCorrelation: whether exposure correlation was used, and its value
Pvalue (per exposure) for some configurations
Alpha: the alpha level used for CIs
*/

```
function zeros(rows, cols) {
   var A = new Array(rows);
   for (var i = 0; i < rows; i++) {
      A[i] = new Array(cols);
      for (var j = 0; j < cols; j++) { A[i][j] = 0; }
   }
   return A;
}

function cloneMatrix(A) {
   var r = A.length;
   var c = (r === 0 ? 0 : A[0].length);
   var B = new Array(r);
   for (var i = 0; i < r; i++) {
      B[i] = new Array(c);
      for (var j = 0; j < c; j++) B[i][j] = A[i][j];
   }
   return B;
}

function transpose(A) {
   var r = A.length;
   var c = (r === 0 ? 0 : A[0].length);
   var T = zeros(c, r);
   for (var i = 0; i < r; i++) for (var j = 0; j < c; j++) T[j][i] = A[i][j];
   return T;
}

function eye(n) {
   var I = zeros(n, n);
   for (var i = 0; i < n; i++) I[i][i] = 1;
   return I;
}

function diagFromVector(v) {
   var n = v.length;
   var D = zeros(n, n);
   for (var i = 0; i < n; i++) D[i][i] = v[i];
   return D;
}

function dotVec(a, b) {
   var n = a.length;
   var s = 0;
```

```
    for (var i = 0; i < n; i++) s += a[i] * b[i];
    return s;
}

function matVecMultiply(A, v) {
    var r = A.length;
    var c = (r === 0 ? 0 : A[0].length);
    var out = new Array(r);
    for (var i = 0; i < r; i++) {
        var s = 0;
        for (var j = 0; j < c; j++) s += A[i][j] * v[j];
        out[i] = s;
    }
    return out;
}

function vecToMatrixCol(v) {
    var r = v.length;
    var M = zeros(r, 1);
    for (var i = 0; i < r; i++) M[i][0] = v[i];
    return M;
}

function flattenMatrixCol(M) {
    var r = M.length; var c = (r === 0 ? 0 : M[0].length);
    if (c !== 1) throw new Error('Expected single-column matrix');
    var v = new Array(r);
    for (var i = 0; i < r; i++) v[i] = M[i][0];
    return v;
}

function addVectors(a, b) {
    var n = a.length; var out = new Array(n);
    for (var i = 0; i < n; i++) out[i] = a[i] + b[i];
    return out;
}

function subtractVectors(a, b) {
    var n = a.length; var out = new Array(n);
    for (var i = 0; i < n; i++) out[i] = a[i] - b[i];
    return out;
}

function scalarMultiplyVec(a, s) {
    var n = a.length; var out = new Array(n);
```

```
    for (var i = 0; i < n; i++) out[i] = a[i] * s;
    return out;
}

function norm2Vec(a) {
    var s = 0;
    for (var i = 0; i < a.length; i++) s += a[i] * a[i];
    return Math.sqrt(s);
}

function subMatrix(A, rowIdx, colIdx) {
    var R = rowIdx.length;
    var C = colIdx.length;
    var M = zeros(R, C);
    for (var i = 0; i < R; i++) {
        for (var j = 0; j < C; j++) {
            M[i][j] = A[rowIdx[i]][colIdx[j]];
        }
    }
    return M;
}

function cbind(matrices) {
    if (matrices.length === 0) return [];
    var rows = matrices[0].length;
    var totalCols = 0;
    for (var k = 0; k < matrices.length; k++) totalCols += matrices[k][0].length;
    var out = zeros(rows, totalCols);
    var colStart = 0;
    for (var m = 0; m < matrices.length; m++) {
        var A = matrices[m];
        var cols = A[0].length;
        for (var i = 0; i < rows; i++) {
            for (var j = 0; j < cols; j++) {
                out[i][colStart + j] = A[i][j];
            }
        }
        colStart += cols;
    }
    return out;
}

function rbind(matrices) {
    if (matrices.length === 0) return [];
    var cols = matrices[0][0].length;
```

```
    var totalRows = 0;
    for (var k = 0; k < matrices.length; k++) totalRows += matrices[k].length;
    var out = zeros(totalRows, cols);
    var rowStart = 0;
    for (var m = 0; m < matrices.length; m++) {
        var A = matrices[m];
        var rows = A.length;
        for (var i = 0; i < rows; i++) {
            for (var j = 0; j < cols; j++) {
                out[rowStart + i][j] = A[i][j];
            }
        }
        rowStart += rows;
    }
    return out;
}

function kronecker(A, B) {
    var ar = A.length; var ac = (ar === 0 ? 0 : A[0].length);
    var br = B.length; var bc = (br === 0 ? 0 : B[0].length);
    var out = zeros(ar * br, ac * bc);
    for (var i = 0; i < ar; i++) {
        for (var j = 0; j < ac; j++) {
            var aij = A[i][j];
            for (var p = 0; p < br; p++) {
                for (var q = 0; q < bc; q++) {
                    out[i * br + p][j * bc + q] = aij * B[p][q];
                }
            }
        }
    }
    return out;
}

function matMul(A, B) {
    // Manual matrix multiplication to avoid math.js dependency issues
    var aRows = A.length;
    var aCols = A[0].length;
    var bRows = B.length;
    var bCols = B[0].length;

    if (aCols !== bRows) {
        throw new Error('Matrix dimension mismatch');
    }
```

```
    var result = zeros(aRows, bCols);
    for (var i = 0; i < aRows; i++) {
        for (var j = 0; j < bCols; j++) {
            var sum = 0;
            for (var k = 0; k < aCols; k++) {
                sum += A[i][k] * B[k][j];
            }
            result[i][j] = sum;
        }
    }
    return result;
}

function matInv(A) {
    return math.inv(A);
}

function matAdd(A, B) {
    var r = A.length;
    var c = A[0].length;
    var result = zeros(r, c);
    for (var i = 0; i < r; i++) {
        for (var j = 0; j < c; j++) {
            result[i][j] = A[i][j] + B[i][j];
        }
    }
    return result;
}

function matScale(A, scalar) {
    var r = A.length;
    var c = A[0].length;
    var result = zeros(r, c);
    for (var i = 0; i < r; i++) {
        for (var j = 0; j < c; j++) {
            result[i][j] = A[i][j] * scalar;
        }
    }
    return result;
}

function toColMat(v) {
    var n = v.length;
    var M = zeros(n, 1);
    for (var i = 0; i < n; i++) M[i][0] = v[i];
```

```
    return M;
}

function colMatToVec(M) {
    var n = M.length;
    var v = new Array(n);
    for (var i = 0; i < n; i++) v[i] = M[i][0];
    return v;
}

function quadFormVec(v, M) {
    var tmp = matVecMultiply(M, v);
    return dotVec(v, tmp);
}

function eigenJacobi(A, tol, maxIter) {
    var n = A.length;
    if (n === 0) return { values: [], vectors: [] };
    tol = (typeof tol === 'undefined') ? 1e-12 : tol;
    maxIter = (typeof maxIter === 'undefined') ? 100 : maxIter;

    var a = cloneMatrix(A);
    var V = eye(n);

    function maxOffDiag(a) {
        var max = 0;
        var p = 0, q = 1;
        for (var i = 0; i < n; i++) {
            for (var j = i + 1; j < n; j++) {
                var val = Math.abs(a[i][j]);
                if (val > max) { max = val; p = i; q = j; }
            }
        }
        return { max: max, p: p, q: q };
    }

    for (var iter = 0; iter < maxIter; iter++) {
        var mo = maxOffDiag(a);
        if (mo.max < tol) break;
        var p = mo.p; var q = mo.q;
        var app = a[p][p], aqq = a[q][q], apq = a[p][q];
        var phi = 0.5 * Math.atan2(2 * apq, (aqq - app));
        var c = Math.cos(phi), s = Math.sin(phi);

        for (var i = 0; i < n; i++) {
```

```
            if (i !== p && i !== q) {
                var aip = a[i][p], aiq = a[i][q];
                a[i][p] = c * aip - s * aiq;
                a[p][i] = a[i][p];
                a[i][q] = s * aip + c * aiq;
                a[q][i] = a[i][q];
            }
        }

        var new_app = c * c * app - 2 * s * c * apq + s * s * aqq;
        var new_aqq = s * s * app + 2 * s * c * apq + c * c * aqq;
        a[p][p] = new_app;
        a[q][q] = new_aqq;
        a[p][q] = 0;
        a[q][p] = 0;

        for (var i = 0; i < n; i++) {
            var vip = V[i][p], viq = V[i][q];
            V[i][p] = c * vip - s * viq;
            V[i][q] = s * vip + c * viq;
        }
    }

    var vals = new Array(n);
    for (var i = 0; i < n; i++) vals[i] = a[i][i];

    var idx = new Array(n);
    for (var i = 0; i < n; i++) idx[i] = i;
    idx.sort(function(i, j) { return Math.abs(vals[j]) - Math.abs(vals[i]); });

    var sortedVals = new Array(n);
    var sortedVecs = zeros(n, n);
    for (var ii = 0; ii < n; ii++) {
        sortedVals[ii] = vals[idx[ii]];
        for (var r = 0; r < n; r++) sortedVecs[r][ii] = V[r][idx[ii]];
    }

    return { values: sortedVals, vectors: sortedVecs };
}

function nelderMead(f, x0, options) {
    options = options || {};
    var maxIter = options.maxIter || 2000;
    var tol = (typeof options.tol === 'undefined') ? 1e-8 : options.tol;
    var alpha = 1; var gamma = 2; var rho = 0.5; var sigma = 0.5;
```

```
  var n = x0.length;
  var simplex = new Array(n + 1);
  simplex[0] = { x: x0.slice(0), f: f(x0) };
  var scale = options.scale || 0.05;
  for (var i = 1; i <= n; i++) {
    var xi = x0.slice(0);
    xi[i - 1] = xi[i - 1] + (xi[i - 1] === 0 ? scale : scale * xi[i - 1]);
    simplex[i] = { x: xi, f: f(xi) };
  }

  function sortSimplex() {
    simplex.sort(function(a, b) { return a.f - b.f; });
  }

  sortSimplex();
  var iter = 0;
  while (iter < maxIter) {
    sortSimplex();
    var best = simplex[0], worst = simplex[n];

    var centroid = new Array(n);
    for (var j = 0; j < n; j++) {
      var s = 0;
      for (var i = 0; i < n; i++) s += simplex[i].x[j];
      centroid[j] = s / n;
    }

    var xr = new Array(n);
    for (var j = 0; j < n; j++) xr[j] = centroid[j] + alpha * (centroid[j] - worst.x[j]);
    var fr = f(xr);

    if (fr < simplex[0].f && fr >= simplex[n - 1].f) {
      simplex[n] = { x: xr, f: fr };
    } else if (fr < simplex[0].f) {
      var xe = new Array(n);
      for (var j = 0; j < n; j++) xe[j] = centroid[j] + gamma * (xr[j] - centroid[j]);
      var fe = f(xe);
      if (fe < fr) simplex[n] = { x: xe, f: fe }; else simplex[n] = { x: xr, f: fr };
    } else {
      var xc = new Array(n);
      for (var j = 0; j < n; j++) xc[j] = centroid[j] + rho * (worst.x[j] - centroid[j]);
      var fc = f(xc);
      if (fc < worst.f) simplex[n] = { x: xc, f: fc };
      else {
```

```
            for (var i = 1; i <= n; i++) {
              var xi = simplex[i].x;
              var x0best = simplex[0].x;
              for (var j = 0; j < n; j++) xi[j] = x0best[j] + sigma * (xi[j] - x0best[j]);
              simplex[i].f = f(xi);
            }
          }
        }
        iter++;

        var fmean = 0;
        for (var i = 0; i <= n; i++) fmean += simplex[i].f;
        fmean /= (n + 1);
        var sst = 0;
        for (var i = 0; i <= n; i++) sst += Math.pow(simplex[i].f - fmean, 2);
        if (Math.sqrt(sst / (n + 1)) < tol) break;
    }
    sortSimplex();
    return { par: simplex[0].x, fval: simplex[0].f };
}

function uniroot(f, a, b, options) {
    options = options || {};
    var tol = (typeof options.tol === 'undefined') ? 1e-8 : options.tol;
    var maxIter = options.maxIter || 200;
    var fa = f(a), fb = f(b);
    var expand = (typeof options.expand === 'undefined') ? true : options.expand;
    var step = 1;

    if (fa === 0) return a;
    if (fb === 0) return b;
    if (fa * fb > 0) {
        if (!expand) throw new Error('Root is not bracketed');
        var maxExpand = options.maxExpand || 50;
        for (var i = 0; i < maxExpand; i++) {
            var left = a - step * (b - a);
            fa = f(left);
            if (fa === 0) return left;
            if (fa * fb < 0) { a = left; break; }
            var right = b + step * (b - a);
            fb = f(right);
            if (fb === 0) return right;
            if (fa * fb < 0) { b = right; break; }
            step *= 2;
        }
```

```
    if (fa * fb > 0) throw new Error('Unable to bracket root in uniroot');
  }

  var left = a, right = b;
  var fl = fa, fr = fb;
  for (var iter = 0; iter < maxIter; iter++) {
    var mid = 0.5 * (left + right);
    var fm = f(mid);
    if (Math.abs(fm) < tol || (right - left) / 2 < tol) return mid;
    if (fl * fm <= 0) {
      right = mid; fr = fm;
    } else {
      left = mid; fl = fm;
    }
  }
  throw new Error('uniroot did not converge');
}

function pnorm(x) {
  return jStat.normal.cdf(x, 0, 1);
}

function qnorm(p) {
  return jStat.normal.inv(p, 0, 1);
}

function pchisq(q, df, lowerTail) {
  var cdf = jStat.chisquare.cdf(q, df);
  return lowerTail ? cdf : 1 - cdf;
}

function zPvalue(z) {
  var p = 2 * (1 - jStat.normal.cdf(Math.abs(z), 0, 1));
  return p;
}

function ci_normal(side, est, se, alpha) {
  var z = qnorm(1 - alpha / 2);
  var out = new Array(est.length);
  if (side === 'l') {
    for (var i = 0; i < est.length; i++) out[i] = est[i] - z * se[i];
  } else {
    for (var i = 0; i < est.length; i++) out[i] = est[i] + z * se[i];
  }
  return out;
```

```
}

function prcompSymmetric(Phi) {
   var eig = eigenJacobi(Phi);
   var sdev = new Array(eig.values.length);
   for (var i = 0; i < eig.values.length; i++) {
      sdev[i] = (eig.values[i] < 0 ? 0 : Math.sqrt(eig.values[i]));
   }
   return { sdev: sdev, rotation: eig.vectors, eigenvalues: eig.values };
}

function mr_mvpcgmm(object, nx, ny, corx, r, thres, robust, alpha) {
   thres = (typeof thres === 'undefined' || thres === null) ? 0.99 : thres;
   robust = (typeof robust === 'undefined') ? true : robust;
   alpha = (typeof alpha === 'undefined') ? 0.05 : alpha;

   var bx = object.betaX;
   var by = object.betaY;
   var sx = object.betaXse;
   var sy = object.betaYse;
   var ld = object.correlation;

   if (!ld || ld.length === 0) {
      throw new Error('LD matrix required in object.correlation');
   }

   var p = bx.length;
   var K = bx[0].length;

   var mis_cor_x = false;
   if (!corx) {
      corx = eye(K);
      mis_cor_x = false;
   } else {
      mis_cor_x = true;
   }

   // Build Phi
   var rowSumsAbsBxDivSy = new Array(p);
   for (var i = 0; i < p; i++) {
      var s = 0;
      for (var j = 0; j < K; j++) s += Math.abs(bx[i][j]);
      rowSumsAbsBxDivSy[i] = s / sy[i];
   }
```

```
var Phi = zeros(p, p);
for (var i = 0; i < p; i++) {
    for (var j = 0; j < p; j++) {
        Phi[i][j] = rowSumsAbsBxDivSy[i] * rowSumsAbsBxDivSy[j] * ld[i][j];
    }
}

// PCA
var pca = prcompSymmetric(Phi);
if (typeof r === 'undefined' || r === null) {
    var sdev2 = new Array(p);
    var total = 0;
    for (var i = 0; i < p; i++) {
        sdev2[i] = pca.sdev[i] * pca.sdev[i];
        total += sdev2[i];
    }
    var cum = 0;
    r = 1;
    while (r <= p) {
        cum += sdev2[r - 1];
        if (cum / total > thres) break;
        r++;
    }
    if (r > p) r = p;
}

var pc_check = (r < (K + 1)) ? 1 : 0;
if (pc_check === 1) {
    console.warn('More principal components needed to perform robust analysis.');
}

// lambda = sqrt(p) * rotation[,1:r]
var lambda = zeros(p, r);
for (var i = 0; i < p; i++) {
    for (var j = 0; j < r; j++) {
        lambda[i][j] = Math.sqrt(p) * pca.rotation[i][j];
    }
}

// Normalize lambda
var tLambda = transpose(lambda);
var tlam_lam = matMul(tLambda, lambda);
var eig2 = eigenJacobi(tlam_lam);
var evec = eig2.vectors;
var evals = eig2.values;
```

```
var sqrtEval = new Array(evals.length);
for (var i = 0; i < evals.length; i++) {
   sqrtEval[i] = Math.sqrt(Math.abs(evals[i]));
}
var S = diagFromVector(sqrtEval);
var evecS = matMul(evec, matMul(S, transpose(evec)));
var inv_evecS = matInv(evecS);
lambda = matMul(lambda, inv_evecS);

// Recalculate tLambda after normalization
tLambda = transpose(lambda);

// Outcome quantities
var ay = new Array(p);
for (var i = 0; i < p; i++) {
   ay[i] = 1 / ((ny * sy[i] * sy[i]) + by[i] * by[i]);
}

var sqrt_ay = new Array(p);
for (var i = 0; i < p; i++) sqrt_ay[i] = Math.sqrt(ay[i]);

var Ay = zeros(p, p);
for (var i = 0; i < p; i++) {
   for (var j = 0; j < p; j++) {
      Ay[i][j] = sqrt_ay[i] * sqrt_ay[j] * ld[i][j];
   }
}

var By = new Array(p);
for (var i = 0; i < p; i++) By[i] = ay[i] * by[i];

var Ayf = matMul(tLambda, matMul(Ay, lambda));
var Byf_col = matMul(tLambda, vecToMatrixCol(By));
var Byf = colMatToVec(Byf_col);

var invAyf = matInv(Ayf);
var temp_vec = matVecMultiply(invAyf, Byf);
var scalar = 1 - dotVec(Byf, temp_vec);
var SigY = matScale(invAyf, scalar * (1 / (ny - r + 1)));
var gamY_est = matVecMultiply(invAyf, Byf);

// Exposures quantities
var ax = zeros(p, K);
for (var k = 0; k < K; k++) {
```

```
    for (var i = 0; i < p; i++) {
      ax[i][k] = 1 / ((nx[k] * sx[i][k] * sx[i][k]) + bx[i][k] * bx[i][k]);
    }
  }

  var Ax_list = new Array(K);
  for (var k = 0; k < K; k++) {
    var sqrt_axk = new Array(p);
    for (var i = 0; i < p; i++) sqrt_axk[i] = Math.sqrt(ax[i][k]);
    var Axk = zeros(p, p);
    for (var i = 0; i < p; i++) {
      for (var j = 0; j < p; j++) {
        Axk[i][j] = sqrt_axk[i] * sqrt_axk[j] * ld[i][j];
      }
    }
    Ax_list[k] = Axk;
  }

  var Bx = zeros(p, K);
  for (var k = 0; k < K; k++) {
    for (var i = 0; i < p; i++) {
      Bx[i][k] = ax[i][k] * bx[i][k];
    }
  }

  var Axf_list = new Array(K);
  for (var k = 0; k < K; k++) {
    Axf_list[k] = matMul(tLambda, matMul(Ax_list[k], lambda));
  }

  var Bx_f = matMul(tLambda, Bx);

  var sqrtAxf_list = new Array(K);
  for (var k = 0; k < K; k++) {
    var eigAx = eigenJacobi(Axf_list[k]);
    var evalAx = eigAx.values;
    var vecAx = eigAx.vectors;
    var sqrtEvalAx = new Array(evalAx.length);
    for (var i = 0; i < evalAx.length; i++) {
      sqrtEvalAx[i] = Math.sqrt(Math.abs(evalAx[i]));
    }
    var S_mat = diagFromVector(sqrtEvalAx);
    sqrtAxf_list[k] = matMul(vecAx, matMul(S_mat, transpose(vecAx)));
  }
```

```
function SigX_kl(k, l) {
  var Skl = matMul(sqrtAxf_list[k], transpose(sqrtAxf_list[l]));
  var invSkl = matInv(Skl);
  var Bk = new Array(r);
  for (var ii = 0; ii < r; ii++) Bk[ii] = Bx_f[ii][k];
  var Bl = new Array(r);
  for (var ii = 0; ii < r; ii++) Bl[ii] = Bx_f[ii][l];
  var temp = matVecMultiply(invSkl, Bl);
  var inner = dotVec(Bk, temp);
  var scalar = corx[k][l] - inner;
  var coeff = scalar * (1 / (Math.sqrt(nx[k] - r + 1) * Math.sqrt(nx[l] - r + 1)));
  return matScale(invSkl, coeff);
}

var SigX = zeros(r * K, r * K);
for (var k = 0; k < K; k++) {
  for (var l = 0; l < K; l++) {
    var block = SigX_kl(k, l);
    for (var i = 0; i < r; i++) {
      for (var j = 0; j < r; j++) {
        SigX[k * r + i][l * r + j] = block[i][j];
      }
    }
  }
}

var gamX_est = zeros(r, K);
for (var k = 0; k < K; k++) {
  var invAxf = matInv(Axf_list[k]);
  var Bk = new Array(r);
  for (var i = 0; i < r; i++) Bk[i] = Bx_f[i][k];
  var vec = matVecMultiply(invAxf, Bk);
  for (var i = 0; i < r; i++) gamX_est[i][k] = vec[i];
}

function gamX_est_col(k) {
  var out = new Array(r);
  for (var i = 0; i < r; i++) out[i] = gamX_est[i][k];
  return out;
}

// Conditional F-test
var condF = new Array(K);
if (K > 1) {
  for (var j = 0; j < K; j++) {
```

```
var idx_block_j = [];
for (var i = 0; i < r; i++) idx_block_j.push(j * r + i);
var idx_rest = [];
for (var t = 0; t < r * K; t++) {
  var included = false;
  for (var q = 0; q < idx_block_j.length; q++) {
    if (t === idx_block_j[q]) included = true;
  }
  if (!included) idx_rest.push(t);
}
var rowsCols = idx_block_j.concat(idx_rest);
var SigXX = subMatrix(SigX, rowsCols, rowsCols);

function g_tet(tet) {
  var left = gamX_est_col(j);
  var Gmat = zeros(r, K - 1);
  var cidx = 0;
  for (var col = 0; col < K; col++) {
    if (col === j) continue;
    for (var rr = 0; rr < r; rr++) Gmat[rr][cidx] = gamX_est[rr][col];
    cidx++;
  }
  var prod = new Array(r);
  for (var ii = 0; ii < r; ii++) {
    var s = 0;
    for (var tt = 0; tt < (K - 1); tt++) s += Gmat[ii][tt] * tet[tt];
    prod[ii] = s;
  }
  var out = new Array(r);
  for (var ii = 0; ii < r; ii++) out[ii] = left[ii] - prod[ii];
  return out;
}

function Qgg(tet) {
  var gvec = g_tet(tet);
  return dotVec(gvec, gvec);
}

var initTet = new Array(K - 1);
for (var ii = 0; ii < K - 1; ii++) initTet[ii] = 0;
var nm = nelderMead(Qgg, initTet, { maxIter: 500 });
var tetgg = nm.par;

function Om_nr(tet) {
  var Id = eye(r);
```

```
            var leftMat = zeros(r, r * K);
            for (var ii = 0; ii < r; ii++) {
              for (var jj = 0; jj < r; jj++) {
                leftMat[ii][jj] = Id[ii][jj];
              }
            }
            for (var tt = 0; tt < K - 1; tt++) {
              var scalar = -tet[tt];
              for (var ii = 0; ii < r; ii++) {
                for (var jj = 0; jj < r; jj++) {
                  leftMat[ii][r + tt * r + jj] = scalar * (ii === jj ? 1 : 0);
                }
              }
            }
            var leftMatT = transpose(leftMat);
            var prod = matMul(leftMat, matMul(SigXX, leftMatT));
            return prod;
          }

          function Qnr(tet) {
            var gvec = g_tet(tet);
            var Om = Om_nr(tet);
            var invOm = matInv(Om);
            var tgm = matVecMultiply(invOm, gvec);
            return dotVec(gvec, tgm);
          }

          var nm2 = nelderMead(Qnr, tetgg, { maxIter: 1000 });
          var condFval = nm2.fval / (r - K + 1);
          condF[j] = condFval;
        }
      } else {
        for (var j = 0; j < K; j++) condF[j] = NaN;
      }

      function packResult(params) {
        var res = {
          robust: params.robust,
          Exposure: object.exposure || null,
          Outcome: object.outcome || null,
          Correlation: object.correlation,
          ExpCorrelation: mis_cor_x,
          CondFstat: params.CondFstat,
          Estimate: params.Estimate,
          StdError: params.StdError,
```

```
        CILower: params.CILower,
        CIUpper: params.CIUpper,
        Overdispersion: params.Overdispersion || null,
        PCs: params.PCs,
        Pvalue: params.Pvalue,
        Alpha: params.Alpha,
        HeterStat: params.HeterStat
    };
    return res;
}

function Om_full_nr(tet) {
    var T = zeros(r, r * K);
    for (var kk = 0; kk < K; kk++) {
        var scalar = tet[kk];
        for (var i = 0; i < r; i++) {
            for (var j = 0; j < r; j++) {
                T[i][kk * r + j] = scalar * (i === j ? 1 : 0);
            }
        }
    }
    var Tt = transpose(T);
    var prod = matMul(T, matMul(SigX, Tt));
    var out = matAdd(SigY, prod);
    return out;
}

if (robust === false && pc_check === 0) {
    function g_full(tet) {
        var out = new Array(r);
        for (var i = 0; i < r; i++) {
            var s = gamY_est[i];
            for (var kk = 0; kk < K; kk++) s -= gamX_est[i][kk] * tet[kk];
            out[i] = s;
        }
        return out;
    }

    function Qgg_full(tet) {
        var gvec = g_full(tet);
        return dotVec(gvec, gvec);
    }

    var tet0 = new Array(K);
    for (var i = 0; i < K; i++) tet0[i] = 0;
```

```
var nm0 = nelderMead(Qgg_full, tet0, { maxIter: 1000 });
var tet_gg = nm0.par;

function Om_nr_full(tet) {
   return Om_full_nr(tet);
}

function Qnr_full(tet) {
   var gvec = g_full(tet);
   var Om = Om_nr_full(tet);
   var invOm = matInv(Om);
   var tgm = matVecMultiply(invOm, gvec);
   return dotVec(gvec, tgm);
}

var Gmat = zeros(r, K);
for (var i = 0; i < r; i++) {
   for (var k = 0; k < K; k++) {
      Gmat[i][k] = -gamX_est[i][k];
   }
}

var nm1 = nelderMead(Qnr_full, tet_gg, { maxIter: 2000 });
var liml_nr = nm1.par;
var Om_at = Om_nr_full(liml_nr);
var invOmAt = matInv(Om_at);

var Gt = transpose(Gmat);
var Atemp = matMul(Gt, matMul(invOmAt, Gmat));
var var_liml_nr = matInv(Atemp);
var Qnr_val = Qnr_full(liml_nr);

var se = new Array(K);
for (var i = 0; i < K; i++) {
   se[i] = Math.sqrt(Math.abs(var_liml_nr[i][i]));
}
var ciL = ci_normal('l', liml_nr, se, alpha);
var ciU = ci_normal('u', liml_nr, se, alpha);
var pvalue_heter = pchisq(Qnr_val, r - K, false);
var pvals = new Array(K);
for (var i = 0; i < K; i++) {
   pvals[i] = 2 * (1 - jStat.normal.cdf(Math.abs(liml_nr[i] / se[i]), 0, 1));
}

return packResult({
```

```
      robust: false,
      CondFstat: condF,
      Estimate: liml_nr,
      StdError: se,
      CILower: ciL,
      CIUpper: ciU,
      PCs: r,
      Pvalue: pvals,
      Alpha: alpha,
      HeterStat: [Qnr_val, pvalue_heter]
    });
  }

  if (robust === true && pc_check === 0) {
    function g_full(tet) {
      var out = new Array(r);
      for (var i = 0; i < r; i++) {
        var s = gamY_est[i];
        for (var kk = 0; kk < K; kk++) s -= gamX_est[i][kk] * tet[kk];
        out[i] = s;
      }
      return out;
    }

    function Qgg_full(tet) {
      var gvec = g_full(tet);
      return dotVec(gvec, gvec);
    }

    var tet0 = new Array(K);
    for (var i = 0; i < K; i++) tet0[i] = 0;
    var nm0 = nelderMead(Qgg_full, tet0, { maxIter: 1000 });
    var tet_gg = nm0.par;

    function Om_nr_full(tet) {
      return Om_full_nr(tet);
    }

    function Qnr_full(tet) {
      var gvec = g_full(tet);
      var Om = Om_nr_full(tet);
      var invOm = matInv(Om);
      var tgm = matVecMultiply(invOm, gvec);
      return dotVec(gvec, tgm);
    }
```

```
var Gmat = zeros(r, K);
for (var i = 0; i < r; i++) {
   for (var k = 0; k < K; k++) {
      Gmat[i][k] = -gamX_est[i][k];
   }
}

var nm1 = nelderMead(Qnr_full, tet_gg, { maxIter: 2000 });
var liml_nr = nm1.par;
var Om_at = Om_nr_full(liml_nr);
var invOmAt = matInv(Om_at);
var Gt = transpose(Gmat);
var Atemp = matMul(Gt, matMul(invOmAt, Gmat));
var var_liml_nr = matInv(Atemp);
var Qnr_val = Qnr_full(liml_nr);

function Om_with_kappa(tet, kappa) {
   var diagk = zeros(r, r);
   for (var i = 0; i < r; i++) diagk[i][i] = kappa / ny;
   var T = zeros(r, r * K);
   for (var kk = 0; kk < K; kk++) {
      var scalar = tet[kk];
      for (var i = 0; i < r; i++) {
         for (var j = 0; j < r; j++) {
            T[i][kk * r + j] = scalar * (i === j ? 1 : 0);
         }
      }
   }
   var prod = matMul(T, matMul(SigX, transpose(T)));
   var sumMat = matAdd(SigY, prod);
   var out = matAdd(sumMat, diagk);
   return out;
}

function Q_kap(kappa) {
   var Omk = Om_with_kappa(liml_nr, kappa);
   var invOmk = matInv(Omk);
   var gvec = g_full(liml_nr);
   var tgm = matVecMultiply(invOmk, gvec);
   return dotVec(gvec, tgm) - (r - K);
}

var kappas = [];
for (var kv = -50; kv <= 50; kv += 0.1) kappas.push(kv);
```

```
var Qvals = new Array(kappas.length);
for (var i = 0; i < kappas.length; i++) Qvals[i] = Q_kap(kappas[i]);
var posCount = 0;
for (var i = 0; i < Qvals.length; i++) if (Qvals[i] > 0) posCount++;
var kappa_est = 0;
var defaultFlag = 0;

if (posCount === 0) {
   kappa_est = 0;
   defaultFlag = 1;
   console.warn('Overdispersion parameter could not be estimated. Default value 0 selected.');
} else {
   var idxMaxPos = -1;
   for (var i = 0; i < Qvals.length; i++) {
      if (Qvals[i] > 0) idxMaxPos = i;
   }
   var a = kappas[idxMaxPos];
   var b = 20;
   try {
      var root = uniroot(Q_kap, a, b, { expand: true });
      kappa_est = root;
      defaultFlag = 0;
   } catch (e) {
      kappa_est = 0;
      defaultFlag = 1;
      console.warn('uniroot failed to find overdispersion root; default 0 selected.');
   }
}

function Om_r(tet) {
   var diagk = zeros(r, r);
   for (var i = 0; i < r; i++) diagk[i][i] = Math.max(0, kappa_est) / ny;
   var T = zeros(r, r * K);
   for (var kk = 0; kk < K; kk++) {
      var scalar = tet[kk];
      for (var i = 0; i < r; i++) {
         for (var j = 0; j < r; j++) {
            T[i][kk * r + j] = scalar * (i === j ? 1 : 0);
         }
      }
   }
   var prod = matMul(T, matMul(SigX, transpose(T)));
   var sumMat = matAdd(SigY, prod);
   var out = matAdd(sumMat, diagk);
   return out;
```

```
    }

    function Q_r(tet) {
      var gvec = g_full(tet);
      var Omr = Om_r(tet);
      var invOmr = matInv(Omr);
      var tgm = matVecMultiply(invOmr, gvec);
      return dotVec(gvec, tgm);
    }

    var nmRob = nelderMead(Q_r, liml_nr, { maxIter: 2000 });
    var liml = nmRob.par;
    var OmrAt = Om_r(liml);
    var invOmrAt = matInv(OmrAt);
    var AtempRob = matMul(Gt, matMul(invOmrAt, Gmat));
    var var_liml = matInv(AtempRob);

    var seRob = new Array(K);
    for (var i = 0; i < K; i++) {
      seRob[i] = Math.sqrt(Math.abs(var_liml[i][i]));
    }
    var ciL = ci_normal('l', liml, seRob, alpha);
    var ciU = ci_normal('u', liml, seRob, alpha);
    var pvals = new Array(K);
    for (var i = 0; i < K; i++) {
      pvals[i] = 2 * (1 - jStat.normal.cdf(Math.abs(liml[i] / seRob[i]), 0, 1));
    }

    var heterStat = Q_r(liml);

    return packResult({
      robust: true,
      CondFstat: condF,
      Estimate: liml,
      StdError: seRob,
      CILower: ciL,
      CIUpper: ciU,
      Overdispersion: kappa_est,
      PCs: r,
      Pvalue: pvals,
      Alpha: alpha,
      HeterStat: heterStat
    });
  }
```

```
    throw new Error('The robust argument must be TRUE or FALSE, or PC check failed.');
}
```

3.3 *Q*-Test

The following Javascrip codes are for *Q*-test of among-algorithmic heterogeneity.

```
/*
Method Name and Objective
Method Name: Cochrane's Q-Test for Algorithm Heterogeneity Assessment
Objective: To determine whether the variation observed among different algorithms' causal effect estimates is simply due to random chance (probabilistic factors) or reflects genuine algorithmic heterogeneity. This test answers the critical question: "Are these algorithms telling us different stories, or are their differences just statistical noise?" The answer determines whether it's scientifically valid to combine (pool) the algorithms' results into a single averaged estimate.

Main Classes and Functions
Main Function: Qtest(x, p)
This function performs a statistical heterogeneity test through the following steps:
Step 1: Data Extraction - Retrieves causal effect estimates and their precision (inverse variance, calculated from standard errors) from each algorithm's results
Step 2: Weighted Average Calculation - Computes a pooled estimate (theta) by averaging all algorithms' causal effects, giving more weight to more precise (lower variance) algorithms
Step 3: Q Statistic Computation - Calculates how much the individual algorithms deviate from this pooled estimate, weighted by their precision. This measures the total observed variation
Step 4: Statistical Comparison - Converts the Q statistic to a p-value using chi-square distribution with k-1 degrees of freedom (where k = number of algorithms). This determines the probability that the observed variation could occur by chance alone
Step 5: Interpretation - Compares the p-value against a threshold (default 0.1):
p-value < 0.1: The variation exceeds what we'd expect from random chance alone → significant algorithmic heterogeneity exists
p-value ≥ 0.1: The variation is within expected random limits → no evidence of important heterogeneity, safe to pool algorithms

Required Inputs and Expected Outputs

Required Inputs:
x: An array of arrays, where each inner array contains one algorithm's results:
Index 0: Algorithm identifier (used for reference, not in calculations)
Index 1: Causal effect estimate from that algorithm
Index 2: Standard error of the causal effect (measures uncertainty)
p: (Optional) Significance threshold for heterogeneity detection, default is 0.1 (10% significance level)

Expected Outputs:
Returns an object with two values:

Q: The Q statistic (a measure of total variation; larger values suggest more heterogeneity)
pValue: The probability that the observed variation is due to chance alone

Practical Interpretation:
```

```
High p-value (≥ 0.1): "The algorithms agree reasonably well. Their differences can be explained by normal statistical variation. Proceed with averaging/pooling the algorithms to get a robust combined estimate."
Low p-value (< 0.1): "The algorithms disagree more than chance would predict. Important algorithmic heterogeneity exists. Investigate why algorithms differ before deciding whether to pool them."

Example Results:
Example 1 (similar algorithms): Q = 0.083, p = 0.999 → No heterogeneity, safe to pool
Example 2 (divergent algorithms): Q = 41.07, p = 0.00000003 → Significant heterogeneity, investigate before pooling
*/

/**
 * Calculates the Cochrane's Q statistic and its associated p-value to assess heterogeneity.
 *
 * @param {Array<Array<number>>} x - An array of algorithms' data. Each inner array should contain:
 *     - index 0: (Common in meta-analysis data)
 *     - index 1: The causal effect for the algorithm.
 *     - index 2: The inverse variance of the causal effect.
 * @returns {number} The pvalue of the Q-statistic. A low pvalue (e.g., < 0.1) indicates significant heterogeneity.
 */
/**
 * Calculates the Cochrane's Q statistic and its associated p-value to assess heterogeneity.
 *
 * @param {Array<Array<number>>} x - An array of algorithms' data. Each inner array should contain:
 *     - index 0: (Common in meta-analysis data)
 *     - index 1: The causal effect for the algorithm.
 *     - index 2: The inverse variance of the causal effect.
 * @returns {number} The pvalue of the Q-statistic. A low pvalue (e.g., < 0.1) indicates significant heterogeneity.
 */
/**
 * Calculates the Cochrane's Q statistic and its associated p-value to assess heterogeneity.
 *
 * @param {Array<Array<number>>} x - An array of algorithms' data. Each inner array should contain:
 *     - index 0: (Common in meta-analysis data)
 *     - index 1: The causal effect for the algorithm.
 *     - index 2: The inverse variance of the causal effect.
 * @returns {number} The pvalue of the Q-statistic. A low pvalue (e.g., < 0.1) indicates significant heterogeneity.
 */
function Qtest(x, p = 0.1) {
  // Ensure we have valid input
  if (!Array.isArray(x) || x.length === 0) {
    throw new Error("Input 'x' must be a non-empty array of algoritms' data.");
  }

  // Extract causal effects and inverse variances
  const causalEffects = x.map((algorithm) => algorithm[1]);
```

```
    const inverseVariances = x.map((algorithm) => 1 / Math.pow(algorithm[2], 2));

    const k = x.length; // Number of algorithms

    // Calculate the weighted mean of causal effects (theta)
    const weightedSumOfCausalEffects = math.dot(causalEffects, inverseVariances);
    const sumOfInverseVariances = math.sum(inverseVariances);
    const theta = weightedSumOfCausalEffects / sumOfInverseVariances;

    // Calculate the Q statistic
    // Sum of w_i * (x_i - theta)^2
    const squaredDifferences = causalEffects.map((causalEffect) => Math.pow(causalEffect - theta, 2));
    const Q = math.dot(inverseVariances, squaredDifferences);

    // Calculate the p-value. The degrees of freedom for Q statistic is k - 1
    const degreesOfFreedom = k - 1;
    const pValue = 1 - jStat.chisquare.cdf(Q, degreesOfFreedom);

    if (pValue < p) {
        console.log("There is a possible clinical and/or algorithmic heterogeneity.");
    } else {
        console.log("There may not be (or there is not enough evidence to show that there is) important clinical and/or algorithmic
heterogeneity.");
    }

    return {Q, pValue};
}
```

3.4 Meta-MR

The following JavaScript codes are for Meta-MR.

```
/*
Method Name and Objective
Method Name: Meta-MR: Algorithm Averaging for Mendelian Randomization

Objective: This method combines results from multiple Mendelian Randomization (MR) algorithms to produce single or multiple
causal effect estimates using the formulae from Huang (2023) and Zhang (2024a). It supports both single-variable MR (scalar
estimates) and multi-variable MR (vector estimates for multiple exposures/outcomes). The method pools estimates using
inverse-variance weighting with proper homogeneous and heterogeneous standard error calculations.

Main Classes and Functions

Main Function: MetaMR(studiesData, alpha, heterogeneityThreshold)
This core function performs all calculations and handles both:
- Single-variable MR: Each algorithm provides scalar estimates
```

- Multi-variable MR: Each algorithm provides vector estimates (same length for all algorithms)

The function:

- Validates input data structure and consistency
- Determines if data contains scalars or vectors
- Calculates weights based on precision (inverse variance): $w_i = 1/s_\theta_i^2$
- Computes pooled causal effects: $\theta_A = \Sigma(w_i * \theta_i) / \Sigma w_i$
- Calculates homogeneous SE: $s_\theta_A = 1 / \sqrt{(\Sigma w_i)}$
- Tests for heterogeneity using Cochran's Q statistic
- Calculates heterogeneous SE if significant heterogeneity detected
- Provides model selection recommendation

Required Inputs:

studiesData - An array containing results from multiple MR algorithms. Each algorithm's data is:

For single-variable MR (scalars, 3 values):

[ID, causalEffect, standardError]

For multi-variable MR (vectors, 3 values where 2nd-3rd are arrays):

[ID, [causalEffects], [standardErrors]]

alpha - Significance level for confidence intervals (default = 0.05 for 95% CI)

heterogeneityThreshold - p-value threshold for heterogeneity test (default = 0.10)

Expected Outputs:

- pooledEffect: Combined causal effects (scalar or vector)
- pooledSE_homo: Homogeneous standard errors
- pooledCI_homo: Homogeneous confidence intervals
- heterogeneityTest: Q statistic, df, p-value for heterogeneity test
- recommendedModel: "homogeneous" or "heterogeneous" based on test
- pooledSE_final: Final SE (heterogeneous if significant, homogeneous otherwise)
- pooledCI_final: Final confidence intervals
- If heterogeneous model selected:
  - tauSquared_A: Between-algorithm variance τ_A^2
  - pooledSE_heter: Heterogeneous standard errors
  - pooledCI_heter: Heterogeneous confidence intervals

Interpretation:

- $Q > \chi^2$ critical value ($p < 0.10$) indicates significant heterogeneity
- Use heterogeneous model when $p <$ heterogeneityThreshold
- $\tau_A^2 > 0$ indicates between-algorithm variability
- Narrower CIs = higher precision and algorithm agreement

*/

```
/**
 * Meta-MR: Algorithm Averaging for Mendelian Randomization
 *
 * @param {Array<Array>} studiesData - Array of algorithm data. Each inner array:
 *      Single-variable: [ID, causalEffect, standardError]
```

```
 *      Multi-variable:    [ID, [causalEffects], [standardErrors]]
 * @param {number} alpha - Significance level for confidence intervals (default: 0.05)
 * @param {number} heterogeneityThreshold - p-value threshold for heterogeneity test (default: 0.10)
 * @returns {Object} Results object with pooled estimates, SEs, CIs, and heterogeneity measures
 */

function MetaMR(studiesData, alpha = 0.05, heterogeneityThreshold = 0.10) {
   // Input validation
   if (!Array.isArray(studiesData) || studiesData.length === 0) {
      throw new Error("Input data must be a non-empty array of algorithm results.");
   }

   const N = studiesData.length; // Number of algorithms
   if (N < 2) {
      throw new Error("At least 2 algorithms are required for meta-analysis.");
   }

   const firstStudy = studiesData[0];
   const cols = firstStudy.length;

   if (cols !== 3) {
      throw new Error("Each algorithm must have exactly 3 columns: [ID, effect(s), SE(s)]");
   }

   // Determine if we're dealing with single-variable (scalars) or multi-variable (vectors)
   const isMultiVariable = Array.isArray(firstStudy[1]);
   const vectorLength = isMultiVariable ? firstStudy[1].length : 1;

   // Validate all studies have consistent structure
   for (let i = 0; i < N; i++) {
      const study = studiesData[i];

      if (!Array.isArray(study) || study.length !== 3 || typeof study[0] !== 'number') {
         throw new Error(`Invalid structure for algorithm ${i + 1}. Expected 3 elements starting with numeric ID.`);
      }

      // Validate effects and SEs
      const effects = isMultiVariable ? study[1] : [study[1]];
      const ses = isMultiVariable ? study[2] : [study[2]];

      if (!Array.isArray(effects) || effects.length !== vectorLength ||
            !Array.isArray(ses) || ses.length !== vectorLength) {
         throw new Error(`Algorithm ${i + 1} has inconsistent vector lengths. Expected length ${vectorLength}.`);
      }
```

```
        // Check each element
        effects.forEach((effect, j) => {
            if (typeof effect !== 'number') {
                throw new Error(`Algorithm ${i + 1}, effect ${j + 1} is not a number.`);
            }
        });

        ses.forEach((se, j) => {
            if (typeof se !== 'number' || se <= 0) {
                throw new Error(`Algorithm ${i + 1}, SE ${j + 1} must be a positive number. Got: ${se}`);
            }
        });
    }

    // Extract data - handle both scalar and vector cases
    const ids = studiesData.map(study => study[0]);
    const causalEffects = studiesData.map(study => isMultiVariable ? study[1] : [study[1]]);
    const standardErrors = studiesData.map(study => isMultiVariable ? study[2] : [study[2]]);

    // Calculate z-score for confidence intervals
    const z_alpha_half = jStat.normal.inv(1 - alpha / 2, 0, 1);

    // Main computation - vectorized for multi-variable case
    const results = computePooledEstimates(
        causalEffects, standardErrors,
        vectorLength, z_alpha_half, alpha, heterogeneityThreshold, N
    );

    return {
        isMultiVariable,
        vectorLength,
        numberOfAlgorithms: N,
        alpha,
        confidenceLevel: (1 - alpha) * 100,
        heterogeneityThreshold,
        ...results
    };
}

/**
 * Core computation function
 * Uses exact formulae from PoolAlgorithm.docx
 */
function computePooledEstimates(causalEffects, standardErrors, vectorLength, z_alpha_half, alpha, heterogeneityThreshold, N)
{
```

```
  const results = {};

  // For multi-variable, compute each element separately
  if (vectorLength > 1) {
    const pooledEffects = [];
    const pooledSEs_homo = [];
    const pooledCIs_homo = [];
    const heterogeneityTests = [];
    let pooledSEs_final = [];
    let pooledCIs_final = [];
    let pooledSEs_heter = null;
    let pooledCIs_heter = null;
    let tauSquared_A = null;

    for (let k = 0; k < vectorLength; k++) {
      // Extract k-th element from all algorithms
      const effects_k = causalEffects.map(row => row[k]);
      const ses_k = standardErrors.map(row => row[k]);

      // Compute for this element using formulae
      const elementResults = computeSingleElement(effects_k, ses_k, z_alpha_half, alpha, heterogeneityThreshold, N);

      pooledEffects.push(elementResults.pooledEffect);
      pooledSEs_homo.push(elementResults.pooledSE_homo);
      pooledCIs_homo.push(elementResults.pooledCI_homo);
      heterogeneityTests.push(elementResults.heterogeneityTest);

      pooledSEs_final.push(elementResults.pooledSE_final);
      pooledCIs_final.push(elementResults.pooledCI_final);

      if (elementResults.isHeterogeneous) {
        if (!pooledSEs_heter) pooledSEs_heter = [];
        if (!pooledCIs_heter) pooledCIs_heter = [];
        if (!tauSquared_A) tauSquared_A = [];

        pooledSEs_heter.push(elementResults.pooledSE_heter);
        pooledCIs_heter.push(elementResults.pooledCI_heter);
        tauSquared_A.push(elementResults.tauSquared_A);
      }
    }

    results.pooledEffect = pooledEffects;
    results.pooledSE_homo = pooledSEs_homo;
    results.pooledCI_homo = pooledCIs_homo;
    results.heterogeneityTests = heterogeneityTests;
```

```
        results.pooledSE_final = pooledSEs_final;
        results.pooledCI_final = pooledCIs_final;

        if (pooledSEs_heter) {
            results.pooledSE_heter = pooledSEs_heter;
            results.pooledCI_heter = pooledCIs_heter;
            results.tauSquared_A = tauSquared_A;
            results.recommendedModel = "heterogeneous";
        } else {
            results.recommendedModel = "homogeneous";
        }

    } else {
        // Single-variable case
        const elementResults = computeSingleElement(
            causalEffects.map(row => row[0]),
            standardErrors.map(row => row[0]),
            z_alpha_half, alpha, heterogeneityThreshold, N
        );

        results.pooledEffect = elementResults.pooledEffect;
        results.pooledSE_homo = elementResults.pooledSE_homo;
        results.pooledCI_homo = elementResults.pooledCI_homo;
        results.heterogeneityTest = elementResults.heterogeneityTest;
        results.pooledSE_final = elementResults.pooledSE_final;
        results.pooledCI_final = elementResults.pooledCI_final;

        if (elementResults.isHeterogeneous) {
            results.pooledSE_heter = elementResults.pooledSE_heter;
            results.pooledCI_heter = elementResults.pooledCI_heter;
            results.tauSquared_A = elementResults.tauSquared_A;
            results.recommendedModel = "heterogeneous";
        } else {
            results.recommendedModel = "homogeneous";
        }
    }

    return results;
}

/**
 * Compute pooled estimates for a single element
 * Uses EXACT formulae from PoolAlgorithm.docx (Huang, 2023; Zhang, 2024a)
 */
function computeSingleElement(effects, ses, z_alpha_half, alpha, heterogeneityThreshold, N) {
```

```
// 1. Calculate weights: w_i = 1 / s_θ_i²
const weights = ses.map(se => 1 / (se * se));

// 2. Calculate pooled effect: θ_A = Σ(w_i * θ_i) / Σw_i
const weightedSum = effects.reduce((sum, effect, i) => sum + effect * weights[i], 0);
const sumWeights = weights.reduce((sum, w) => sum + w, 0);
const pooledEffect = weightedSum / sumWeights;

// 3. Calculate homogeneous standard error: s_θ_A = 1 / √(Σw_i)
const pooledSE_homo = 1 / Math.sqrt(sumWeights);

// 4. Calculate confidence interval for homogeneous model
const pooledCI_homo = [
  pooledEffect - z_alpha_half * pooledSE_homo,
  pooledEffect + z_alpha_half * pooledSE_homo
];

// 5. Test for heterogeneity using Cochran's Q statistic
// Q = Σw_i(θ_i - θ_A)²
const Q = weights.reduce((sum, w, i) => {
  const diff = effects[i] - pooledEffect;
  return sum + w * diff * diff;
}, 0);

const df = N - 1; // Degrees of freedom
const pValue = 1 - jStat.chisquare.cdf(Q, df); // p-value for heterogeneity test

const isHeterogeneous = pValue < heterogeneityThreshold;

const heterogeneityTest = {
  Q: Q,
  df: df,
  pValue: pValue,
  significant: isHeterogeneous,
  threshold: heterogeneityThreshold
};

const results = {
  pooledEffect,
  pooledSE_homo,
  pooledCI_homo,
  heterogeneityTest,
  isHeterogeneous,
  pooledSE_final: pooledSE_homo,
  pooledCI_final: pooledCI_homo
```

```
    };

    // 6. If significant heterogeneity, calculate heterogeneous model
    if (isHeterogeneous) {
        // τ_A² = (Q - (N-1)) / Σw_i    [DerSimonian-Laird estimator]
        const tauSquared_A = Math.max(0, (Q - df) / sumWeights);

        // s_A = √(τ_A² + s_θ_A²)
        const pooledSE_heter = Math.sqrt(tauSquared_A + pooledSE_homo * pooledSE_homo);

        // Confidence interval for heterogeneous model
        const pooledCI_heter = [
            pooledEffect - z_alpha_half * pooledSE_heter,
            pooledEffect + z_alpha_half * pooledSE_heter
        ];

        results.pooledSE_heter = pooledSE_heter;
        results.pooledCI_heter = pooledCI_heter;
        results.tauSquared_A = tauSquared_A;
        results.pooledSE_final = pooledSE_heter;
        results.pooledCI_final = pooledCI_heter;
    }

    return results;
}

// ==================== ENHANCED DISPLAY FUNCTION ====================

function displayResults(title, results, inputData, alpha, heterogeneityThreshold) {
    console.log(`\n${'='.repeat(90)}`);
    console.log(`${title}`);
    console.log(`${'='.repeat(90)}`);

    console.log(`Analysis Type: ${results.isMultiVariable ? 'Multi-variable' : 'Single-variable'} MR`);
    console.log(`Number of Algorithms: ${results.numberOfAlgorithms}`);
    console.log(`Confidence Level: ${results.confidenceLevel}% (α = ${alpha})`);
    console.log(`Heterogeneity Threshold: p < ${heterogeneityThreshold}`);
    console.log(`Vector Length: ${results.vectorLength}`);
    console.log(`Recommended Model: ${results.recommendedModel}`);

    console.log(`\n--- FORMULAE USED (per Huang 2023, Zhang 2024a) ---`);
    console.log("1. Weights: w_i = 1 / s_θ_i²");
    console.log("2. Pooled Effect: θ_A = Σ(w_i × θ_i) / Σw_i");
    console.log("3. Homogeneous SE: s_θ_A = 1 / √(Σw_i)");
    console.log("4. Heterogeneity: Q = Σw_i(θ_i - θ_A)² ~ χ²(N-1)");
```

```
    console.log("5. τ_A² = max(0, (Q - (N-1)) / Σw_i)    [DerSimonian-Laird]");
    console.log("6. Heterogeneous SE: s_A = √(τ_A² + s_θ_A²)");

    if (!results.isMultiVariable) {
        console.log(`\n--- RESULTS (Single-Variable) ---`);
        console.log(`Pooled Causal Effect (θ_A): ${results.pooledEffect.toFixed(4)}`);
        console.log(`Homogeneous SE (s_θ_A): ${results.pooledSE_homo.toFixed(4)}`);
        console.log(`${results.confidenceLevel}% CI (homogeneous): [${results.pooledCI_homo[0].toFixed(4)}, ${results.pooledCI_homo[1].toFixed(4)}]`);

        const test = results.heterogeneityTest;
        console.log(`\n--- Heterogeneity Test (Cochran's Q) ---`);
        console.log(`Q = ${test.Q.toFixed(4)}, df = ${test.df}, p = ${test.pValue.toFixed(6)}`);
        console.log(`Significant heterogeneity: ${test.significant ? 'YES' : 'NO'} (p ${test.pValue < test.threshold ? '<' : '>='} ${test.threshold})`);

        if (test.significant) {
            console.log(`\n--- HETEROGENEOUS MODEL (Recommended) ---`);
            console.log(`Between-Algorithm Variance (τ_A²): ${results.tauSquared_A.toFixed(6)}`);
            console.log(`Heterogeneous SE (s_A): ${results.pooledSE_heter.toFixed(4)}`);
            console.log(`${results.confidenceLevel}% CI (heterogeneous): [${results.pooledCI_heter[0].toFixed(4)}, ${results.pooledCI_heter[1].toFixed(4)}]`);
            console.log(`FINAL ESTIMATE: θ_A = ${results.pooledEffect.toFixed(4)} ± ${results.pooledSE_final.toFixed(4)}`);
        } else {
            console.log(`\n--- HOMOGENEOUS MODEL (Recommended) ---`);
            console.log(`FINAL ESTIMATE: θ_A = ${results.pooledEffect.toFixed(4)} ± ${results.pooledSE_homo.toFixed(4)}`);
        }
    } else {
        console.log(`\n--- RESULTS (Multi-Variable) ---`);
        console.log("Index | θ_A | s_θ_A | Homo CI Lwr | Homo CI Upr | Q | p-value | Model | τ_A² | s_A | Heter CI Lwr | Heter CI Upr");
        console.log("-".repeat(120));

        results.pooledEffect.forEach((effect, i) => {
            const se_homo = results.pooledSE_homo[i];
            const ci_homo = results.pooledCI_homo[i];
            const test = results.heterogeneityTests[i];
            const model = test.significant ? "Heter" : "Homo";
            let tau2 = "", se_heter = "", ci_heter_lwr = "", ci_heter_upr = "";

            if (test.significant) {
                tau2 = results.tauSquared_A[i].toFixed(6);
                se_heter = results.pooledSE_heter[i].toFixed(4);
                const ci_heter = results.pooledCI_heter[i];
                ci_heter_lwr = ci_heter[0].toFixed(4);
```

```
            ci_heter_upr = ci_heter[1].toFixed(4);
        }

        console.log(`${i.toString().padStart(5)} | ${effect.toFixed(4).padStart(6)} | ${se_homo.toFixed(4).padStart(6)} | ${ci_homo[0].toFixed(4).padStart(9)} | ${ci_homo[1].toFixed(4).padStart(9)} | ${test.Q.toFixed(4).padStart(6)} | ${test.pValue.toFixed(6).padStart(8)} | ${model.padStart(5)} | ${tau2.padStart(8)} | ${se_heter.padStart(6)} | ${ci_heter_lwr.padStart(9)} | ${ci_heter_upr.padStart(9)}`);
    });
  }

  console.log(`\n--- Input Data Summary ---`);
  console.log("Algorithm | Effect(s)" + (results.isMultiVariable ? " (vector)" : ""));
  console.log("----------" + (results.isMultiVariable ? "----------" : "----------"));

  inputData.forEach((algo, i) => {
    const id = algo[0];
    const effects = results.isMultiVariable ? algo[1] : [algo[1]];
    const effectStr = results.isMultiVariable ?
      `[${effects.map(e => e.toFixed(3)).join(', ')}]` :
      `${effects[0].toFixed(3)}`;
    console.log(`    ${id}      | ${effectStr}`);
  });

  console.log(`\n--- INTERPRETATION ---`);
  console.log("- Uses EXACT formulae from Huang (2023) and Zhang (2024a)");
  console.log("- Q-test p < 0.10 indicates significant between-algorithm heterogeneity");
  console.log("- τ_A² > 0 shows between-algorithm variability (heterogeneous model)");
  console.log("- Final estimates use heterogeneous model when heterogeneity is significant");
  console.log("- Narrower CIs = higher precision and algorithm agreement");
  console.log("- Multi-variable: Each column represents a different causal pathway");
  console.log(`${'='.repeat(90)}\n`);
}

// ==================== EXAMPLES ====================

// Helper function to generate synthetic data with known properties
function generateExampleData(type, N, vectorLength = 1, addHeterogeneity = false) {
  const data = [];
  const trueEffects = vectorLength > 1 ? Array.from({length: vectorLength}, () => Math.random() * 0.5) : [0.25];

  for (let i = 1; i <= N; i++) {
    const id = i;
    const effects = vectorLength > 1 ?
      trueEffects.map(effect => effect + (addHeterogeneity ? (Math.random() - 0.5) * 0.3 : (Math.random() - 0.5) * 0.05)) :
      [trueEffects[0] + (addHeterogeneity ? (Math.random() - 0.5) * 0.3 : (Math.random() - 0.5) * 0.05)];
```

```
      const ses = vectorLength > 1 ?
        Array.from({length: vectorLength}, () => 0.08 + Math.random() * 0.04) :
        [0.08 + Math.random() * 0.04];

      if (vectorLength > 1) {
        data.push([id, effects, ses]);
      } else {
        data.push([id, effects[0], ses[0]]);
      }
    }

    return data;
}
```

3.5 Plotting MR results

3.5.1 Plotting genetic association data

The following JavaScript codes are for plotting genetic association data.

```
/**
The provided JavaScript file, mr_plot-methods.js, contains code for plotting genetic association data using the Plotly library.

Key Functions:

mrPlotMRMVInput(object, options = {}):
This function is designed to plot genetic association data.
It takes an object containing genetic data and an options object to customize the plot.
The options object includes:
error: Whether to display error bars (default: true).
line: Whether to display a regression line (default: true).
orientate: Whether to orient the data based on the sign of the fitted values (default: false).
interactive: Whether the plot should be interactive (default: true).
labels: Whether to display labels (default: false).

linearRegression(x, y, weights):
This function is intended to perform linear regression.
It is supposed to take x and y data points along with weights to compute the regression line.

Data Processing:
The function extracts betaX, betaY, betaXse, betaYse and alpha from the input object.
It calculates fitted values (BxPlot) using the linearRegression function.
Error bounds (BxErrorLower and BxErrorUpper) are calculated using the standard error (Bxse) and the normal distribution quantile.

Plot Customization:
```

The plot is customized using Plotly's trace and layout objects.
The trace object defines the scatter plot with markers, and optionally includes error bars.
The layout object defines the axes, titles, and additional shapes like reference lines.
Interactive vs. Non-Interactive Plot:
If interactive is true, the plot is rendered using Plotly.newPlot.
If interactive is false, the function logs a message indicating that non-interactive plots are not yet implemented.

Example Usage:
To use this function, you would need to provide an object with the required properties (betaX, betaY, betaXse, betaYse, outcome, snps) and call mrPlotMRMVInput with the desired options.

```
const data = {
    betaX: [1, 2, 3, 4],
    betaY: [2, 4, 6, 8],
    betaXse: [0.1, 0.2, 0.3, 0.4],
    betaYse: [0.1, 0.2, 0.3, 0.4],
    outcome: 'Trait',
    snps: ['SNP1', 'SNP2', 'SNP3', 'SNP4'],
    alpha: 0.001
};
mrPlotMRMVInput(data, { error: true, line: true, orientate: false, interactive: true });
```

Next Steps:
Add support for non-interactive plots if needed.
This code is a good starting point for visualizing genetic association data, but it requires some additional work to be fully functional.
*/

```
function mrPlotMRMVInput(object, options = {}) {
    const { error = true, line = true, orientate = false, interactive = true, labels = false } = options;

    let Bx = object.betaX;
    let By = object.betaY;
    let Bxse = object.betaXse;
    let Byse = object.betaYse;
    let alpha = object.alpha;
    const BxPlot = linearRegression(Bx, By, Byse.map(se => 1 / (se * se))).fittedValues;
    const BxErrorLower = Bx.map((bx, i) => BxPlot[i] - normalQuantile(1-alpha/2) * Bxse[i]);
    const BxErrorUpper = Bx.map((bx, i) => BxPlot[i] + normalQuantile(1-alpha/2) * Bxse[i]);
    if (orientate) {
        By = By.map((val, i) => val * Math.sign(BxPlot[i]));
        BxPlot = BxPlot.map(Math.abs);
    }
    const trace = {
        x: BxPlot,
```

```
            y: By,
            mode: 'markers',
            type: 'scatter',
            marker: { color: 'black', size: 8, opacity: 0.5 },
            text: object.snps
        };
        const layout = {
            xaxis: {
                title: `Fitted value of genetic association with ${object.outcome}`,
                tickfont: { size: 13 },
                titlefont: { size: 15 }
            },
            yaxis: {
                title: `Estimated genetic association with ${object.outcome}`,
                tickfont: { size: 13 },
                titlefont: { size: 15 }
            },
            shapes: [
                { type: 'line', x0: 0, x1: 0, y0: Math.min(...By), y1: Math.max(...By), line: { color: 'red', width: 1, dash: 'dot' } },
                { type: 'line', y0: 0, y1: 0, x0: Math.min(...BxPlot), x1: Math.max(...BxPlot), line: { color: 'red', width: 1, dash:
'dot' } }
            ]
        };

        if (error) {
            trace.error_x = { type: 'data', array: BxErrorUpper.map((upper, i) => upper - BxPlot[i]), arrayminus: BxPlot.map((val, i)
=> val - BxErrorLower[i]), color: 'blue', opacity: 0.3 };
            trace.error_y = { type: 'data', array: Byse.map(se => normalQuantile(1-alpha/2) * se), color: 'blue', opacity: 0.3 };
        }

        if (line) {
            layout.shapes.push({
                type: 'line',
                x0: Math.min(...BxPlot),
                x1: Math.max(...BxPlot),
                y0: 0,
                y1: 0,
                line: { color: 'blue', width: 2 }
            });
        }
        if (interactive) {
            Plotly.newPlot('plotDiv', [trace], layout);
        } else {
            console.log("Non-interactive plot not implemented yet.");
        }
```

```
}

function linearRegression(x, y, weights) {
    if (x.length !== y.length || x.length !== weights.length) {
        throw new Error("Input arrays must have the same length");
    }
    let sumWeights = 0;
    let sumX = 0;
    let sumY = 0;
    let sumXY = 0;
    let sumXX = 0;
    for (let i = 0; i < x.length; i++) {
        const w = weights[i];
        sumWeights += w;
        sumX += w * x[i];
        sumY += w * y[i];
        sumXY += w * x[i] * y[i];
        sumXX += w * x[i] * x[i];
    }
    const meanX = sumX / sumWeights;
    const meanY = sumY / sumWeights;
    const slope = (sumXY - sumWeights * meanX * meanY) / (sumXX - sumWeights * meanX * meanX);
    const intercept = meanY - slope * meanX;
    // Calculate fitted values
    const fittedValues = x.map(xi => intercept + slope * xi);
    return {
        fittedValues: fittedValues,
        slope: slope,
    };
}
```

3.5.2 Plotting funnel plot

The codes are designed to generate a funnel plot, which is commonly used in meta-analyses and causal inference studies to visualize the precision of causal estimates and their confidence intervals.

```
/*
The codes are a JavaScript designed to generate a funnel plot, which is commonly used in meta-analyses and causal inference studies to visualize the precision of causal estimates and their confidence intervals.

1. MRInput Class
Purpose: This class is used to store input data required for generating the funnel plot.
Attributes:
betaX: The effect size of the exposure variable.
betaY: The effect size of the outcome variable.
betaXse: The standard error of the exposure variable's effect size.
```

betaYse: The standard error of the outcome variable's effect size.

2. mrFunnel Function

Purpose: This function generates the funnel plot using the Plotly.js library.

Parameters:

object: An instance of MRInput containing the necessary data.

r: The causal estimate, calculated as By / Bx.

precision: The precision of the estimate, calculated as Math.abs(Bx / Byse).

CI_lower and CI_upper: The lower and upper bounds of the, e.g., 99.9% confidence interval, calculated as (By – 3.29 * Byse) / Bx and (By + 3.29 * Byse) / Bx, respectively.

CI: A boolean flag indicating whether to display confidence intervals on the plot (default is true).

Plot Elements:

Data Points: Scatter plot of r (causal estimate) against precision.

Confidence Intervals: Error bars representing the 99.9% confidence interval for each data point.

Vertical Lines:

A solid line at x=0 representing no effect.

A dashed line at the IVW (Inverse Variance Weighted) estimate, representing the overall causal estimate.

Plotly.js: The function uses Plotly.js to render the plot. If CI is true, it includes error bars; otherwise, it plots the data points without confidence intervals.

3. mrIVW Function

4. Example Usage

An example MRInput object is created with sample data for betaX, betaY, betaXse, and betaYse.

The mrFunnel function is called with this object to generate the funnel plot, displaying both data points and confidence intervals.

```
const mrInput = new MRInput([0.5, 0.8, 1.2], [0.3, 0.4, 0.6], [0.1, 0.15, 0.2], [0.05, 0.08, 0.1], 0.001);
mrFunnel(mrInput, true);
```

5. Code Structure

The code is well-structured, with clear separation between data input, metric calculation, and plot generation.

The mrFunnel function is flexible, allowing users to control whether confidence intervals are displayed.

6. Error Handling: The code could include error handling to manage cases where the input data is invalid or missing.

7. Application Scenarios

This script is particularly useful in causal inference studies, where funnel plots are used to detect biases such as publication bias.

It can also be used in meta-analyses to visualize the precision and distribution of effect sizes across different studies.

8. Example Output

The generated funnel plot will display the causal estimates (r) on the x-axis and their precision on the y-axis.

Confidence intervals will be shown as error bars around each data point.

The vertical lines at x=0 and the IVW estimate provide reference points for interpreting the plot.

Summary

The codes provide a JavaScript implementation for generating funnel plots, which are useful for visualizing causal estimates and their precision. The code is modular, with clear separation between data handling, metric calculation, and visualization. It relies on Plotly.js for rendering the plot and can be extended to include more advanced features such as custom IVW calculations and additional plot customization options.

*/

```
class MRInput {
    constructor(betaX, betaY, betaXse, betaYse, alpha) {
        this.betaX = betaX;
        this.betaY = betaY;
        this.betaXse = betaXse;
        this.betaYse = betaYse;
        this.alpha = alpha;
    }
}

// mrIVW function
function mrIVW(object) {
  // Extracting input data
  const betaX = object.betaX;
  const betaY = object.betaY;
  const seX = object.betaXse;
  const seY = object.betaYse;
  const alpha = object.alpha;
  // Calculate weights (inverse variance)
  const weights = betaX.map((bx, i) => {
    const bx2 = bx * bx;
    const bx4 = bx2 * bx2;
    const seY2 = seY[i] * seY[i];
    const seX2 = seX[i] * seX[i];
    const by2 = betaY[i] * betaY[i];
    return 1 / (seY2 / bx2 + seX2 * by2 / bx4);
  });
  // Calculate the weighted average effect
  const weightedEffects = weights.map((w, i) => w * (betaY[i] / betaX[i]));
  const sumWeights = weights.reduce((acc, w) => acc + w, 0);
  const sumWeightedEffects = weightedEffects.reduce((acc, w) => acc + w, 0);
  const beta_IVW = sumWeightedEffects / sumWeights;
  // Return estimate
  return {
    estimate: beta_IVW
  };
}

// Funnel plot function
```

```
function mrFunnel(object, CI = true) {
    const Bx = object.betaX;
    const By = object.betaY;
    const Bxse = object.betaXse;
    const Byse = object.betaYse;
    const alpha = object.alpha;
    // Calculate ratios and precision
    const r = By.map((by, i) => by / Bx[i]);
    const precision = Bx.map((bx, i) => Math.abs(bx / Byse[i]));

    // Estimate of the IVW method
    const estimate = mrIVW(object).estimate;

    // Calculate confidence intervals
    const CI_lower = By.map((by, i) => (by - normalQuantile(1-alpha/2) * Byse[i]) / Bx[i]);
    const CI_upper = By.map((by, i) => (by + normalQuantile(1-alpha/2) * Byse[i]) / Bx[i]);
    // Create data frame equivalent
    const dframe = r.map((val, i) => ({
        r: val,
        precision: precision[i],
        CI_lower: CI_lower[i],
        CI_upper: CI_upper[i]
    }));
    const X_label = "Causal estimate " +    `(${100 * (1 - alpha)}% CI)`;
    const Y_label = "Precision";
    if (CI) {
        console.log("Creating funnel plot with confidence intervals");
        const trace = {
            x: r,
            y: precision,
            type: 'scatter',
            mode: 'markers',
            error_x: {
                type: 'data',
                array: CI_upper.map((upper, i) => upper - r[i]),
                arrayminus: r.map((val, i) => val - CI_lower[i])
            },
            name: 'Data points'
        };
        const layout = {
            title: 'MR Funnel Plot',
            xaxis: { title: X_label },
            yaxis: { title: Y_label },
            shapes: [
                {
```

```
                type: 'line',
                x0: 0,
                x1: 0,
                y0: Math.min(...precision),
                y1: Math.max(...precision),
                line: { color: 'black' }
            },
            {
                type: 'line',
                x0: estimate,
                x1: estimate,
                y0: Math.min(...precision),
                y1: Math.max(...precision),
                line: { color: 'black', dash: 'dash' }
            }
        ]
    };

    Plotly.newPlot('funnel-plot', [trace], layout);

} else {
    console.log("Plotting without confidence intervals");
    const trace = {
        x: r,
        y: precision,
        type: 'scatter',
        mode: 'markers',
        name: 'Data points'
    };
    const layout = {
        title: 'MR Funnel Plot',
        xaxis: { title: X_label },
        yaxis: { title: Y_label },
        shapes: [
            {
                type: 'line',
                x0: 0,
                x1: 0,
                y0: Math.min(...precision),
                y1: Math.max(...precision),
                line: { color: 'black' }
            },
            {
                type: 'line',
                x0: estimate,
```

```
                        x1: estimate,
                        y0: Math.min(...precision),
                        y1: Math.max(...precision),
                        line: { color: 'black', dash: 'dash' }
                    }
                ]
            };

            Plotly.newPlot('funnel-plot', [trace], layout);        }

        // Return the data that would be used for plotting
        return {
            data: dframe,
            estimate: estimate,
            X_label: X_label,
            Y_label: Y_label,
            hasCI: CI
        };

            Plotly.newPlot('funnel-plot', [trace], layout);
}
```

3.5.3 Plotting forest plot

The codes are designed for generating forest plots.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>MR Forest Plot (Univariable & Multivariable)</title>
    <script src="https://cdn.plot.ly/plotly-latest.min.js"></script>
    <style>
        body {
            font-family: Arial, sans-serif;
            margin: 20px;
        }
        #forest-plot {
            width: 100%;
            height: 600px;
            margin-top: 20px;
        }
        .controls {
            margin-bottom: 20px;
        }
```

```
        button {
            margin-right: 10px;
            padding: 8px 16px;
            background-color: #007bff;
            color: white;
            border: none;
            border-radius: 4px;
            cursor: pointer;
        }
        button:hover {
            background-color: #0056b3;
        }
        .section-title {
            margin-top: 30px;
            margin-bottom: 10px;
            font-weight: bold;
            color: #333;
        }
        select {
            padding: 8px 16px;
            margin-right: 20px;
            border: 2px solid #007bff;
            border-radius: 4px;
            background-color: white;
            color: #333;
            font-size: 14px;
            cursor: pointer;
        }
        select:hover {
            background-color: #f0f8ff;
        }
        .control-row {
            display: flex;
            align-items: center;
            margin-bottom: 20px;
        }
    </style>
</head>
<body>
    <h1>Mendelian Randomization Forest Plot</h1>
    <div class="section-title">Select MR Type and Example:</div>
    <div class="control-row">
        <select id="mrTypeSelector">
            <option value="univariable">Univariable MR</option>
            <option value="multivariable">Multivariable MR</option>
```

```
            </select>
            <button onclick="generateExample(1)">Example 1: SNPs + Methods</button>
            <button onclick="generateExample(2)">Example 2: Methods Only</button>
            <button onclick="generateExample(3)">Example 3: SNPs Only</button>
            <button onclick="generateExample(4)">Example 4: Sorted</button>
        </div>
        <div id="forest-plot"></div>
<script src="http://www.iaees.org/publications/software/JavaScript/probfunctions.js"></script>
        <script>
/*
MR Forest Plot Generator

Objective:
This tool generates interactive forest plots for visualizing Mendelian Randomization (MR) results. It can display causal effect estimates at the SNP level and/or method level (e.g., IVW, MR-Egger) with confidence intervals, standard errors, and sorting options. The script supports Univariable MR and Multivariable MR (MVMR), allowing users to compare effects across multiple exposures.

Parameters:
object – Input data (MRInput for univariable, MVMRInput for multivariable)
alpha – Confidence interval significance level (e.g., 0.05)
snp_estimates – Show SNP-level estimates (true/false)
snp_methods – Show method-level estimates (true/false)
estmatrix – Optional method-level results array
ordered – Sort output by effect size (true / false)
isMVMR – Enable multivariable mode (true / false)

Required Inputs:

Univariable MR (MRInput):
betaX – SNP–exposure effect sizes
betaXse – Standard errors for betaX
betaY – SNP–outcome effect sizes
betaYse – Standard errors for betaY
snps – SNP identifiers
(Optional) estmatrix – Method names, estimates, CI lower, CI upper, SE

Multivariable MR (MVMRInput):
betaX – 2D array: SNP–exposure effects for multiple exposures
betaXse – 2D array of standard errors matching betaX
betaY – SNP–outcome effect sizes
betaYse – Standard errors for betaY
snps – SNP identifiers
exposures – Exposure variable names
(Optional) estmatrix – Method-level estimates per exposure
```

Outputs:

Interactive Forest Plot

Plots effect sizes with error bars (confidence intervals)

Displays SNP-level and/or method-level estimates

Can sort results by effect size

Renders in HTML (#forest-plot div) using Plotly.js for zoom, tooltips, and export

Console Messages

Data validation and transformation logs

Warnings if input data is inconsistent or incomplete

Processing steps for transparency

Use Case:

Designed for researchers performing MR analyses who want a clear, interactive visual summary of causal estimates across multiple instruments or statistical methods. It enables side-by-side comparison, quick identification of significant effects, and transparent reporting in publications or presentations.

*/

```
class MRInput {
    constructor(betaX, betaXse, betaY, betaYse, snps) {
        this.betaX = betaX || [];
        this.betaXse = betaXse || [];
        this.betaY = betaY || [];
        this.betaYse = betaYse || [];
        this.snps = snps || [];
    }
    /**
     * 验证输入数据
     * @returns {boolean} 是否有效
     */
    isValid() {
        const n = this.betaX.length;
        return this.betaX.length === n &&
               this.betaXse.length === n &&
               this.betaY.length === n &&
               this.betaYse.length === n;
    }
}
/**
 * 多变量MR输入类
 */
class MVMRInput {
    constructor(betaX, betaXse, betaY, betaYse, snps, exposures) {
        // betaX: 二维数组 [SNPs x Exposures]
        // betaXse: 二维数组 [SNPs x Exposures]
```

```
            this.betaX = betaX || [];
            this.betaXse = betaXse || [];
            this.betaY = betaY || [];
            this.betaYse = betaYse || [];
            this.snps = snps || [];
            this.exposures = exposures || [];
        }
        isValid() {
            const nSNPs = this.betaX.length;
            if (nSNPs === 0) return false;
            const nExposures = this.betaX[0].length;
            return this.betaXse.length === nSNPs &&
                    this.betaY.length === nSNPs &&
                    this.betaYse.length === nSNPs &&
                    this.betaXse.every(row => row.length === nExposures) &&
                    this.betaX.every(row => row.length === nExposures);
        }
        getNumExposures() {
            return this.betaX.length > 0 ? this.betaX[0].length : 0;
        }
        getNumSNPs() {
            return this.betaX.length;
        }
    }
     /**
     * 生成MR森林图（支持单变量和多变量MR）
     * @param {MRInput|MVMRInput|Object} object - 输入数据对象
     * @param {number} alpha - 显著性水平，默认0.001
     * @param {boolean} snp_estimates - 是否显示SNP估计值，默认true
     * @param {boolean} snp_methods - 是否显示方法估计值，默认true
     * @param {Array} estmatrix - 方法估计值矩阵，可选
     * @param {boolean} ordered - 是否排序，默认false
     * @param {boolean} isMVMR - 是否为多变量MR，默认false
     */
    function mrForest(object, alpha = 0.001, snp_estimates = true, snp_methods = true, estmatrix = null, ordered = false,
isMVMR = false) {
        console.log('mrForest called with:', { alpha, snp_estimates, snp_methods, ordered, isMVMR });
        // 确定是否为MVMR
        const isMVMRMode = isMVMR || (object instanceof MVMRInput);
        if (isMVMRMode) {
            // 多变量MR处理
            if (!(object instanceof MVMRInput)) {
                console.error('Invalid input object for MVMR. Expected MVMRInput instance.');
                return;
            }
```

```
            if (!object.isValid()) {
                console.error('Invalid MVMR input data');
                return;
            }
            generateMVMRForestPlot(object, alpha, snp_estimates, snp_methods, estmatrix, ordered);
        } else {
            // 单变量MR处理
            if (!(object instanceof MRInput)) {
                console.error('Invalid input object. Expected MRInput instance.');
                return;
            }
            if (!object.isValid()) {
                console.error('Invalid input data: arrays must have equal length');
                return;
            }
            generateUnivariableMRForestPlot(object, alpha, snp_estimates, snp_methods, estmatrix, ordered);
        }
    }
    /**
     * 生成单变量MR森林图
     */
    function generateUnivariableMRForestPlot(object, alpha, snp_estimates, snp_methods, estmatrix, ordered) {
        const { betaX, betaXse, betaY, betaYse, snps } = object;
        // 计算SNP级别的估计值
        let dframe = [];
        let calculated = [];
        // 1. 处理方法估计值（estmatrix）
        if (snp_methods && estmatrix) {
            const numCols = estmatrix[0].length;
            console.log('Processing estmatrix with', numCols, 'columns');
            for (let i = 0; i < numCols; i++) {
                const rowCount = estmatrix.length;
                if (rowCount < 5) {
                    console.warn(`estmatrix row ${i} has insufficient data`);
                    continue;
                }
                const methodName = estmatrix[0] && estmatrix[0][i] ?
                    String(estmatrix[0][i]) : `Method_${i + 1}`;
                const estimate = estmatrix[1] && estmatrix[1][i] ?
                    parseFloat(estmatrix[1][i]) : 0;
                const ciLower = estmatrix[2] && estmatrix[2][i] ?
                    parseFloat(estmatrix[2][i]) : estimate - 0.1;
                const ciUpper = estmatrix[3] && estmatrix[3][i] ?
                    parseFloat(estmatrix[3][i]) : estimate + 0.1;
                const se = estmatrix[4] && estmatrix[4][i] ?
```

```
                parseFloat(estmatrix[4][i]) : 0.08;
            calculated.push({
                snp: methodName,
                estimate: estimate,
                CI_lower: ciLower,
                CI_upper: ciUpper,
                effect: estimate,
                se: se
            });
        }
    }
    // 2. 处理SNP估计值
    if (snp_estimates) {
        const n = betaX.length;
        const CI_range = normalQuantile(1 - alpha / 2);
        for (let i = 0; i < n; i++) {
            const snpName = snps[i] || `snp_${i + 1}`;
            const estimate = betaY[i] / betaX[i];
            const ciLower = estimate - (CI_range * betaYse[i]) / Math.abs(betaX[i]);
            const ciUpper = estimate + (CI_range * betaYse[i]) / Math.abs(betaX[i]);
            const se = (ciUpper - ciLower) / (2 * CI_range);
            dframe.push({
                snp: snpName,
                estimate: estimate,
                CI_lower: ciLower,
                CI_upper: ciUpper,
                effect: estimate,
                se: se
            });
        }
    }
    // 3. 合并数据
    if (snp_estimates && snp_methods) {
        dframe = dframe.concat(calculated);
    } else if (!snp_estimates && snp_methods) {
        dframe = calculated;
    }
    if (dframe.length === 0) {
        console.error('No data to plot');
        return;
    }
    console.log('Generated dframe with', dframe.length, 'rows:', dframe);
    // 4. 排序（如果需要）
    if (ordered) {
        dframe.sort((a, b) => a.estimate - b.estimate);
```

```
    }
    // 5. 绘制图表
    plotForestChart(dframe, alpha, 'Mendelian Randomization Forest Plot');
}
/**
 * 生成多变量MR森林图
 */
function generateMVMRForestPlot(object, alpha, snp_estimates, snp_methods, estmatrix, ordered) {
    const { betaX, betaXse, betaY, betaYse, snps, exposures } = object;
    const nSNPs = object.getNumSNPs();
    const nExposures = object.getNumExposures();
    console.log(`Processing MVMR with ${nSNPs} SNPs and ${nExposures} exposures`);
    let dframe = [];
    let calculated = [];
    // 1. 处理方法估计值（estmatrix）- MVMR格式
    if (snp_methods && estmatrix) {
        // estmatrix格式：
        // [[Method1_Exp1, Method1_Exp2, ...], [estimates], [CI_lower], [CI_upper], [se]]
        const numCols = estmatrix[0].length;
        console.log('Processing MVMR estmatrix with', numCols, 'columns');
        for (let i = 0; i < numCols; i++) {
            const rowCount = estmatrix.length;
            if (rowCount < 5) {
                console.warn(`estmatrix row ${i} has insufficient data`);
                continue;
            }
            const methodName = estmatrix[0] && estmatrix[0][i] ?
                String(estmatrix[0][i]) : `Method_${i + 1}`;
            const estimate = estmatrix[1] && estmatrix[1][i] ?
                parseFloat(estmatrix[1][i]) : 0;
            const ciLower = estmatrix[2] && estmatrix[2][i] ?
                parseFloat(estmatrix[2][i]) : estimate - 0.1;
            const ciUpper = estmatrix[3] && estmatrix[3][i] ?
                parseFloat(estmatrix[3][i]) : estimate + 0.1;
            const se = estmatrix[4] && estmatrix[4][i] ?
                parseFloat(estmatrix[4][i]) : 0.08;
            calculated.push({
                snp: methodName,
                estimate: estimate,
                CI_lower: ciLower,
                CI_upper: ciUpper,
                effect: estimate,
                se: se
            });
        }
```

```
        }
        // 2. 处理SNP估计值 - 对每个暴露分别计算
        if (snp_estimates) {
            const CI_range = normalQuantile(1 - alpha / 2);
            for (let expIdx = 0; expIdx < nExposures; expIdx++) {
                const exposureName = exposures[expIdx] || `Exposure_${expIdx + 1}`;
                for (let i = 0; i < nSNPs; i++) {
                    const snpName = snps[i] || `snp_${i + 1}`;
                    const fullName = `${snpName} (${exposureName})`;
                    // 计算该SNP对该暴露的效应
                    const estimate = betaY[i] / betaX[i][expIdx];
                    const ciLower = estimate - (CI_range * betaYse[i]) / Math.abs(betaX[i][expIdx]);
                    const ciUpper = estimate + (CI_range * betaYse[i]) / Math.abs(betaX[i][expIdx]);
                    const se = (ciUpper - ciLower) / (2 * CI_range);
                    dframe.push({
                        snp: fullName,
                        estimate: estimate,
                        CI_lower: ciLower,
                        CI_upper: ciUpper,
                        effect: estimate,
                        se: se,
                        exposure: exposureName,
                        snpOnly: snpName
                    });
                }
            }
        }
        // 3. 合并数据
        if (snp_estimates && snp_methods) {
            dframe = dframe.concat(calculated);
        } else if (!snp_estimates && snp_methods) {
            dframe = calculated;
        }
        if (dframe.length === 0) {
            console.error('No data to plot');
            return;
        }
        console.log('Generated MVMR dframe with', dframe.length, 'rows:', dframe);
        // 4. 排序（如果需要）
        if (ordered) {
            dframe.sort((a, b) => a.estimate - b.estimate);
        }
        // 5. 绘制图表
        plotForestChart(dframe, alpha, 'Multivariable Mendelian Randomization Forest Plot');
    }
```

```
/**
 * 通用森林图绘制函数
 */
function plotForestChart(dframe, alpha, title) {
    // 1. 创建绘图容器
    let plotDiv = document.getElementById('forest-plot');
    if (!plotDiv) {
        plotDiv = document.createElement('div');
        plotDiv.id = 'forest-plot';
        document.body.appendChild(plotDiv);
    }
    // 设置容器高度
    plotDiv.style.height = Math.max(400, dframe.length * 30) + 'px';
    // 2. 准备绘图数据
    const xMax = Math.max(...dframe.map(d => d.CI_upper));
    const xMin = Math.min(...dframe.map(d => d.CI_lower));
    const xRange = xMax - xMin;
    const trace = {
        x: dframe.map(d => d.estimate),
        y: dframe.map((_, i) => i),
        error_x: {
            type: 'data',
            array: dframe.map(d => d.CI_upper - d.estimate),
            arrayminus: dframe.map(d => d.estimate - d.CI_lower),
            color: 'rgba(0,0,255,0.5)',
            thickness: 2,
            width: 4
        },
        mode: 'markers',
        type: 'scatter',
        marker: {
            size: 6,
            color: 'rgb(31, 19, 255)',
            line: {
                color: 'rgb(31, 19, 255)',
                width: 2
            }
        },
        name: 'Estimates',
        text: dframe.map(d => `${d.snp}<br>Estimate: ${d.effect.toFixed(4)}<br>SE: ${d.se.toFixed(4)}`),
        hovertemplate: '%{text}<extra></extra>',
        showlegend: false
    };
    // 3. 布局配置
    const layout = {
```

```
title: {
    text: title,
    x: 0.5,
    font: { size: 16 }
},
xaxis: {
    title: `Causal Estimate (${(1 - alpha) * 100}% CI)`,
    range: [xMin - 0.1 * xRange, xMax + 0.1 * xRange],
    zeroline: true,
    zerolinecolor: 'rgba(0,0,0,0.3)',
    zerolinewidth: 1,
    gridcolor: 'rgba(0,0,0,0.1)',
    showgrid: true,
    tickformat: '.3f'
},
yaxis: {
    title: 'SNPs/Methods',
    tickvals: dframe.map((_, i) => i),
    ticktext: dframe.map(d => d.snp),
    automargin: true,
    categoryorder: 'array',
    tickfont: { size: 10 },
    showgrid: false
},
margin: {
    l: 200,
    r: 80,
    b: 60,
    t: 80,
    pad: 10
},
shapes: [{
    type: 'line',
    x0: 0,
    x1: 0,
    y0: -0.5,
    y1: dframe.length - 0.5,
    line: {
        color: 'rgba(0,0,0,0.5)',
        width: 2,
        dash: 'dash'
    }
}],
annotations: [
    // 标题行
```

```
            {
                xref: 'paper',
                yref: 'paper',
                x: 0.75,
                y: 1.05,
                text: 'Effect Size',
                showarrow: false,
                font: { size: 11, color: 'black' },
                bgcolor: 'rgba(255,255,255,0.9)',
                bordercolor: 'rgba(0,0,0,0.2)',
                borderwidth: 1
            },
            {
                xref: 'paper',
                yref: 'paper',
                x: 0.90,
                y: 1.05,
                text: 'Std Error',
                showarrow: false,
                font: { size: 11, color: 'black' },
                bgcolor: 'rgba(255,255,255,0.9)',
                bordercolor: 'rgba(0,0,0,0.2)',
                borderwidth: 1
            }
        ],
        font: {
            size: 12
        },
        hovermode: 'closest',
        plot_bgcolor: 'rgba(0,0,0,0)',
        paper_bgcolor: 'rgba(0,0,0,0)',
        legend: {
            x: 0.02,
            y: 0.98,
            bgcolor: 'rgba(255,255,255,0.8)'
        }
    };
    // 4. 添加数值标注
    dframe.forEach((d, i) => {
        layout.annotations.push({
            xref: 'paper',
            yref: 'y',
            x: 0.75,
            y: i,
            text: d.effect.toFixed(4),
```

```
                    showarrow: false,
                    font: { size: 9, color: 'black' },
                    bgcolor: 'rgba(255,255,255,0.9)',
                    bordercolor: 'rgba(0,0,0,0.1)',
                    borderwidth: 0.5
                });
                layout.annotations.push({
                    xref: 'paper',
                    yref: 'y',
                    x: 0.90,
                    y: i,
                    text: d.se.toFixed(4),
                    showarrow: false,
                    font: { size: 9, color: 'black' },
                    bgcolor: 'rgba(255,255,255,0.9)',
                    bordercolor: 'rgba(0,0,0,0.1)',
                    borderwidth: 0.5
                });
            });
            // 5. 绘制图表
            if (typeof Plotly !== 'undefined') {
                Plotly.newPlot('forest-plot', [trace], layout).then(() => {
                    console.log('Forest plot generated successfully');
                }).catch(error => {
                    console.error('Error generating plot:', error);
                });
            } else {
                console.error('Plotly.js is not loaded. Please include the Plotly library.');
                const plotDiv = document.getElementById('forest-plot');
                plotDiv.innerHTML = '<p style="color: red; font-size: 16px;">Error: Plotly.js library is required to generate the forest plot.</p><p>Please include: <code>&lt;script src="https://cdn.plot.ly/plotly-latest.min.js"&gt;&lt;/script&gt;</code></p>';
            }
        }
        // ============ 统一的示例生成函数 ============
        /**
         * 根据下拉列表选择生成对应的示例
         * @param {number} exampleNum - 示例编号 (1-4)
         */
        function generateExample(exampleNum) {
            const mrType = document.getElementById('mrTypeSelector').value;
            if (mrType === 'univariable') {
                // 调用单变量MR示例
                switch(exampleNum) {
                    case 1:
```

```
                    generateUnivariableExample1();
                    break;
                case 2:
                    generateUnivariableExample2();
                    break;
                case 3:
                    generateUnivariableExample3();
                    break;
                case 4:
                    generateUnivariableExample4();
                    break;
            }
        } else {
            // 调用多变量MR示例
            switch(exampleNum) {
                case 1:
                    generateMVMRExample1();
                    break;
                case 2:
                    generateMVMRExample2();
                    break;
                case 3:
                    generateMVMRExample3();
                    break;
                case 4:
                    generateMVMRExample4();
                    break;
            }
        }
    }
    // ============ 单变量MR示例 ============
    /**
     * 示例1：SNPs + Methods
     */
    function generateUnivariableExample1() {
        const mrinput1 = new MRInput(
            [1.0, 2.0, 3.0, 4.0],
            [0.1, 0.2, 0.3, 0.4],
            [4.0, 5.0, 6.0, 7.0],
            [0.6, 0.7, 0.8, 0.9],
            ["rs123", "rs456", "rs789", "rs101"]
        );
        const exampleEstMatrix = [
            ['IVW', 'Weighted Median', 'MR-Egger'],
            [0.25, 0.23, 0.24],
```

```
            [0.10, 0.12, 0.11],
            [0.40, 0.34, 0.37],
            [0.08, 0.07, 0.08]
        ];
        console.log('Generating Univariable Example 1...');
        mrForest(mrinput1, 0.001, true, true, exampleEstMatrix, false, false);
    }
    /**
     * 示例2：Methods Only
     */
    function generateUnivariableExample2() {
        const mrinput2 = new MRInput(
            [1.0, 2.0, 3.0, 4.0],
            [0.1, 0.2, 0.3, 0.4],
            [4.0, 5.0, 6.0, 7.0],
            [0.6, 0.7, 0.8, 0.9],
            []
        );
        const exampleEstMatrix = [
            ['IVW', 'Weighted Median', 'MR-Egger'],
            [0.25, 0.23, 0.24],
            [0.10, 0.12, 0.11],
            [0.40, 0.34, 0.37],
            [0.08, 0.07, 0.08]
        ];
        console.log('Generating Univariable Example 2...');
        mrForest(mrinput2, 0.001, false, true, exampleEstMatrix, false, false);
    }
    /**
     * 示例3：SNPs Only
     */
    function generateUnivariableExample3() {
        const mrinput3 = new MRInput(
            [1.0, 2.0, 3.0, 4.0],
            [0.1, 0.2, 0.3, 0.4],
            [4.0, 5.0, 6.0, 7.0],
            [0.6, 0.7, 0.8, 0.9],
            ["snp1", "snp2", "snp3", "snp4"]
        );
        console.log('Generating Univariable Example 3...');
        mrForest(mrinput3, 0.001, true, false, null, false, false);
    }
    /**
     * 示例4：排序
     */
```

```
function generateUnivariableExample4() {
    const mrinput4 = new MRInput(
        [1.0, 2.0, 3.0, 4.0],
        [0.1, 0.2, 0.3, 0.4],
        [4.0, 5.0, 6.0, 7.0],
        [0.6, 0.7, 0.8, 0.9],
        ["rs123", "rs456", "rs789", "rs101"]
    );
    const exampleEstMatrix = [
        ['IVW', 'Weighted Median', 'MR-Egger'],
        [0.25, 0.23, 0.24],
        [0.10, 0.12, 0.11],
        [0.40, 0.34, 0.37],
        [0.08, 0.07, 0.08]
    ];
    console.log('Generating Univariable Example 4 (Sorted)...');
    mrForest(mrinput4, 0.001, true, true, exampleEstMatrix, true, false);
}
// ============ 多变量MR示例 ============
/**
 * MVMR示例1：SNPs + Methods（2个暴露）
 */
function generateMVMRExample1() {
    const mvmrinput1 = new MVMRInput(
        // betaX: 4 SNPs x 2 Exposures
        [
            [0.50, -0.20],
            [0.28, 0.12],
            [-0.55, 0.32],
            [0.22, -0.07]
        ],
        // betaXse: 4 SNPs x 2 Exposures
        [
            [0.11, 0.08],
            [0.13, 0.10],
            [0.24, 0.14],
            [0.09, 0.07]
        ],
        // betaY: 4 SNPs
        [0.25, 0.08, -0.14, 0.30],
        // betaYse: 4 SNPs
        [0.15, 0.12, 0.20, 0.18],
        // snps
        ["rs1001", "rs1002", "rs1003", "rs1004"],
        // exposures
```

```
            ["ExposureA", "ExposureB"]
        );
        // 方法估计值
        const mvmrEstMatrix = [
            ['MVMR-IVW (ExposureA)', 'MVMR-IVW (ExposureB)'],
            [0.3726, -0.1993],
            [-0.7088, -2.3838],
            [1.4541, 1.9853],
            [0.2513, 0.5077]
        ];
        console.log('Generating MVMR Example 1 (SNPs + Methods)...');
        mrForest(mvmrinput1, 0.05, true, true, mvmrEstMatrix, false, true);
}
/**
 * MVMR示例2：Methods Only（3个暴露）
 */
function generateMVMRExample2() {
        const mvmrinput2 = new MVMRInput(
            // betaX: 5 SNPs x 3 Exposures
            [
                [0.45, -0.18, 0.32],
                [0.30, 0.15, -0.22],
                [-0.50, 0.28, 0.41],
                [0.25, -0.10, 0.19],
                [0.38, 0.22, -0.15]
            ],
            // betaXse
            [
                [0.10, 0.07, 0.09],
                [0.12, 0.09, 0.11],
                [0.20, 0.13, 0.15],
                [0.08, 0.06, 0.08],
                [0.11, 0.10, 0.09]
            ],
            // betaY
            [0.28, 0.12, -0.18, 0.35, 0.22],
            // betaYse
            [0.14, 0.11, 0.19, 0.16, 0.13],
            // snps
            ["rs2001", "rs2002", "rs2003", "rs2004", "rs2005"],
            // exposures
            ["Lipids", "BMI", "Blood Pressure"]
        );
        // 方法估计值（仅展示方法）
        const mvmrEstMatrix = [
```

```
            ['MVMR-IVW (Lipids)', 'MVMR-IVW (BMI)', 'MVMR-IVW (BP)', 'MVMR-Egger (Lipids)', 'MVMR-Egger (BMI)', 'MVMR-Egger (BP)'],
            [0.42, 0.31, -0.25, 0.38, 0.29, -0.22],
            [0.18, 0.12, -0.58, 0.15, 0.09, -0.55],
            [0.66, 0.50, 0.08, 0.61, 0.49, 0.11],
            [0.12, 0.10, 0.17, 0.12, 0.10, 0.17]
        ];
        console.log('Generating MVMR Example 2 (Methods Only)...');
        mrForest(mvmrinput2, 0.05, false, true, mvmrEstMatrix, false, true);
    }
    /**
     * MVMR示例3：SNPs Only（2个暴露）
     */
    function generateMVMRExample3() {
        const mvmrinput3 = new MVMRInput(
            // betaX: 4 SNPs x 2 Exposures
            [
                [0.52, -0.25],
                [0.31, 0.18],
                [-0.48, 0.35],
                [0.27, -0.12]
            ],
            // betaXse
            [
                [0.12, 0.09],
                [0.14, 0.11],
                [0.22, 0.15],
                [0.10, 0.08]
            ],
            // betaY
            [0.30, 0.15, -0.20, 0.28],
            // betaYse
            [0.16, 0.13, 0.21, 0.17],
            // snps
            ["rs3001", "rs3002", "rs3003", "rs3004"],
            // exposures
            ["Smoking", "Alcohol"]
        );
        console.log('Generating MVMR Example 3 (SNPs Only)...');
        mrForest(mvmrinput3, 0.05, true, false, null, false, true);
    }
    /**
     * MVMR示例4：Sorted（2个暴露，带方法，排序）
     */
    function generateMVMRExample4() {
```

```
        const mvmrinput4 = new MVMRInput(
            // betaX: 5 SNPs x 2 Exposures
            [
                [0.48, -0.22],
                [0.35, 0.15],
                [-0.52, 0.38],
                [0.29, -0.10],
                [0.41, 0.20]
            ],
            // betaXse
            [
                [0.11, 0.08],
                [0.13, 0.10],
                [0.21, 0.14],
                [0.09, 0.07],
                [0.12, 0.09]
            ],
            // betaY
            [0.32, 0.18, -0.22, 0.26, 0.24],
            // betaYse
            [0.15, 0.12, 0.19, 0.14, 0.13],
            // snps
            ["rs4001", "rs4002", "rs4003", "rs4004", "rs4005"],
            // exposures
            ["Education", "Income"]
        );
        // 方法估计值
        const mvmrEstMatrix = [
            ['MVMR-IVW (Education)', 'MVMR-IVW (Income)', 'MVMR-Median (Education)', 'MVMR-Median (Income)'],
            [0.45, -0.28, 0.41, -0.25],
            [0.22, -0.72, 0.18, -0.68],
            [0.68, 0.16, 0.64, 0.18],
            [0.12, 0.22, 0.12, 0.22]
        ];
        console.log('Generating MVMR Example 4 (Sorted)...');
        mrForest(mvmrinput4, 0.05, true, true, mvmrEstMatrix, true, true);
    }
    // 页面加载完成后自动RUN示例
    document.addEventListener('DOMContentLoaded', function() {
        console.log('DOM loaded, generating default example...');
        setTimeout(() => {
            generateUnivariableExample1();
        }, 100);
    });
```

```
        // 全局错误处理
        window.addEventListener('error', function(e) {
            console.error('Global error:', e.error);
        });
    </script>
</body>
</html>
```

4 Algorithmic Description and User Guide of The Web platform for Meta-MR

4.1 Purpose of the Platform

The platform, Meta-MR (http://www.iaees.org/publications/journals/nb/articles/2027-17(2)/Meta-MR.htm), is a complete web-based computational platform for Mendelian Randomization analysis implemented with HTML + CSS + JavaScript.

Its purpose is to provide an integrated workflow for:

1. Generating harmonized Mendelian Randomization input JSON data.
2. Supporting both:
   - Single-variable MR
   - Multi-variable MR
3. Dynamically loading and executing external MR method JavaScript files.
4. Running multiple MR methods selected by the user.
5. Extracting causal estimates, standard errors, confidence intervals, and p-values.
6. Performing heterogeneity analysis using Cochran’s Q statistic.
7. Performing pooled Meta-MR estimation using **MetaMR()**.
8. Drawing forest plots using mrForest().
9. Exporting results and forest plots into a Word-compatible .doc file.

The overall platform is designed as a stepwise MR pipeline:

text

A0. Generate harmonized JSON data

↓

B1. Select MR type: Single-variable or Multi-variable MR

↓

B2. Select MR methods

↓

B3. Configure method-specific parameters

↓

B4. Dynamically load MR method scripts

↓

B5. Execute selected MR methods

↓

B6. Extract estimates and standard errors

↓

B7. Run Q-test

↓

B8. Run MetaMR pooling

↓

B9. Draw forest plot

↓

B10. Export results to Word

4.2 Overall System Architecture

The system can be divided into the following architectural modules.

4.2.1 Front-End User Interface Layer

The UI is built with HTML and CSS. It contains:

A0 Data Workflow Interface

The A0 section allows users to generate the global MR input JSON data through three workflows:

1. **Direct JSON Input**
2. **File Upload and Harmonization**
3. **AI and Multi-Source GWAS Fetching**

The final generated JSON contains:

javascript

```
{
  "snps": [...],
  "betaX": [...],
  "betaY": [...],
  "betaXse": [...],
  "betaYse": [...],
  "correlation": [[...], [...]],
  "exposure": "...",
  "outcome": "..."
}
```

These values are stored into global variables:

javascript

```
var globalBetaX = null;
var globalBetaY = null;
var globalBetaXse = null;
var globalBetaYse = null;
var globalSnps = null;
var globalCorrelation = null;
var globalExposure = null;
var globalOutcome = null;
```

B MR Analysis Interface

After A0 is completed, the user proceeds to Step B.

The MR interface provides:

1. MR type selection:
   - Single-variable MR
   - Multi-variable MR
2. MR method checkboxes.
3. Parameter panels for each method.

4. Run buttons:

Run html

```
<button id="runSingleMRBtn" onclick="runSingleMR()">
  ▶ Run Single-Variable MR Analysis
</button>

<button id="runMultiMRBtn" onclick="runMultiMR()">
  ▶ Run Multi-Variable MR Analysis
</button>
```

RUN

5. Output textareas:

Run html

```
<textarea id="singleMRResults" class="results-textarea" readonly></textarea>
<textarea id="multiMRResults" class="results-textarea" readonly></textarea>
```

RUN

6. Forest plot containers:

Run html

```
<div id="forest-plot-single"></div>
<div id="forest-plot-multi"></div>
```

RUN

7. Word export buttons.

4.2.2 Application State Layer

The platform uses two types of state.

4.2.2.1 Global MR Data State

These global variables are the bridge between the A0 JSON generator and the MR analysis section:

javascript

```
var globalBetaX = null;
var globalBetaY = null;
var globalBetaXse = null;
var globalBetaYse = null;
var globalSnps = null;
var globalCorrelation = null;
var globalExposure = null;
var globalOutcome = null;
```

They are set by:

javascript

```
function setGlobalDataFromJSON(parsed) {
    globalBetaX = parsed.betaX;
    globalBetaY = parsed.betaY;
    globalBetaXse = parsed.betaXse;
    globalBetaYse = parsed.betaYse;
    globalSnps = parsed.snps;
    globalCorrelation = parsed.correlation || null;
```

```
    globalExposure = parsed.exposure || 'Exposure';
    globalOutcome = parsed.outcome || 'Outcome';
}
```

4.2.2.2 Workflow State Object

The program also maintains a broader state object for workflow progress:

javascript

```
var STATE = {
    workflow: null,
    exposureData: {},
    outcomeData: {},
    harmonizedData: null,
    correlationMatrix: null,
    finalJson: null,
    correlationSource: 'ai',
    aiCorrelationSource: 'ai',
    gwasCorrelationSource: 'ai',
    gwasRawData: {
        exposure: {},
        outcome: {}
    }
};
```

This object stores intermediate file upload data, AI-generated data, harmonized data, correlation matrices, and GWAS fetching results.

4.2.3 Input Data Layer

The input layer supports three workflows.

4.2.3.1 Workflow 1: Direct JSON Input

The user pastes already harmonized JSON into a textarea.

The function:

javascript

```
validateDirectJson()
```

checks that required fields exist and have matching lengths.

Core validation:

javascript

```
function validateDirectJson() {
    var textarea = document.getElementById('jsonDirectInput');
    var parsed = JSON.parse(textarea.value.trim());

    var required = ['snps', 'betaX', 'betaY', 'betaXse', 'betaYse'];

    for (var i = 0; i < required.length; i++) {
        if (!parsed[required[i]] || !Array.isArray(parsed[required[i]])) {
            throw new Error('Missing or invalid field: ' + required[i]);
```

```
        }
    }

    var length = parsed.snps.length;

    if (
        parsed.betaX.length !== length ||
        parsed.betaY.length !== length ||
        parsed.betaXse.length !== length ||
        parsed.betaYse.length !== length
    ) {
        throw new Error('All arrays must have the same length');
    }

    if (parsed.correlation) {
        if (!Array.isArray(parsed.correlation) || parsed.correlation.length !== length) {
            throw new Error('Correlation matrix must match SNP count');
        }
    }

    setGlobalDataFromJSON(parsed);
}
```

4.2.3.2 Workflow 2: File Upload and Harmonization

Users upload exposure and outcome files.

Supported formats include:

text

.csv

.txt

.json

.gz

The parser detects whether the file is compressed, JSON, CSV, or TXT.

Main parsing function:

javascript

```
async function parseFile(file) {
    var buf = await file.arrayBuffer();
    var text;

    if (
        file.name.endsWith('.gz') ||
        (
            new Uint8Array(buf, 0, 2)[0] === 0x1f &&
            new Uint8Array(buf, 0, 2)[1] === 0x8b
        )
```

```
    ) {
        text = pako.ungzip(new Uint8Array(buf), { to: 'string' });
    } else {
        text = new TextDecoder().decode(buf);
    }

    if (file.name.endsWith('.json') || text.trim().startsWith('{')) {
        try {
            return normalizeJsonData(JSON.parse(text));
        } catch (e) {}
    }

    return parseCsvData(text);
}
```

Each row is normalized into a standard format:

javascript

```
{
    rsid: "...",
    ea: "...",
    nea: "...",
    beta: number,
    se: number,
    eaf: number
}
```

4.2.3.3 Workflow 3: AI and Multi-Source GWAS Fetching

The AI workflow lets users enter:

text

Exposure trait(s)

Outcome variable

Analysis type: univariate or multivariate

AI service

API key

The platform can request GWAS-style summary data from external AI APIs.

The prompt structure is:

javascript

```
var prompt =
    'Fetch realistic genetic variant data for a ' + analysisType + ' MR analysis.\n' +
    'Exposure: ' + exposures.join(', ') + '\n' +
    'Outcome: ' + outcome + '\n' +
    'Fetch CSV: SNP,beta,se,effect_allele,other_allele,eaf\n' +
    '15-20 variants. Return:\n' +
    'EXPOSURE_DATA:\n[CSV]\n' +
```

```
    'OUTCOME_DATA:\n[CSV]';
```

The GWAS multi-source module can query selected public sources such as:

text

IEU OpenGWAS
GWAS Catalog
GWAS Atlas
eQTLGen
FinnGen

In the current code, IEU OpenGWAS is the concrete browser-accessible implementation:

javascript

```
async function fetchFromSource(trait, source, pvalThreshold) {
    if (source === 'ieugwas') {
        var mappedTrait = getMappedTraitId(trait, 'ieugwas');
        var url =
            'https://gwas-api.mrcieu.ac.uk/associations/' +
            encodeURIComponent(mappedTrait) +
            '?proxies=0';

        var response = await fetchWithRetry(url);
        var data = await response.json();

        return processSummaryData(data, pvalThreshold, source);
    } else {
        throw new Error('Source not available in browser');
    }
}
```

4.3 Core Data Structures

4.3.1 MRInput Class

The MRInput class represents single-variable MR data.

javascript

```
class MRInput {
    constructor(betaX, betaXse, betaY, betaYse, snps, correlation, alpha) {
        this.betaX = betaX || [];
        this.betaXse = betaXse || [];
        this.betaY = betaY || [];
        this.betaYse = betaYse || [];
        this.snps = snps || [];
        this.correlation = correlation || null;
        this.alpha = alpha || 0.001;
    }

    isValid() {
```

```
        var n = this.betaX.length;

        return n > 0 &&
                this.betaXse.length === n &&
                this.betaY.length === n &&
                this.betaYse.length === n;
    }
}
```

Purpose

It packages the arrays required by single-variable MR methods:

text

betaX	SNP-exposure effect sizes
betaXse	standard errors of betaX
betaY	SNP-outcome effect sizes
betaYse	standard errors of betaY
snps	SNP identifiers
correlation	SNP correlation matrix
alpha	significance level

4.3.2 MVMRInput Class

The MVMRInput class represents multi-variable MR data.

javascript

```
class MVMRInput {
    constructor(betaX, betaXse, betaY, betaYse, snps, exposures, correlation, alpha) {
        this.betaX = betaX || [];
        this.betaXse = betaXse || [];
        this.betaY = betaY || [];
        this.betaYse = betaYse || [];
        this.snps = snps || [];
        this.exposures = exposures || [];
        this.correlation = correlation || null;
        this.alpha = alpha || 0.001;
    }

    isValid() {
        var nSNPs = this.betaX.length;

        if (nSNPs === 0) return false;

        return this.betaXse.length === nSNPs &&
                this.betaY.length === nSNPs &&
                this.betaYse.length === nSNPs;
    }
```

```
    getNumExposures() {
        return this.betaX.length > 0 ? this.betaX[0].length : 0;
    }

    getNumSNPs() {
        return this.betaX.length;
    }
}
```

Purpose

It supports multiple exposures. Therefore:

javascript

betaX[i][j]

means:

text

The effect of SNP i on exposure j

4.4 Step A0 Algorithm: JSON Data Generation

The A0 step is responsible for generating:

javascript

betaX

betaY

betaXse

betaYse

snps

correlation

These are required by all subsequent MR methods.

4.4.1 A0 Workflow Selection Algorithm

The user selects one of three workflows.

javascript

```
function selectWorkflow(workflow) {
    STATE.workflow = workflow;

    document.getElementById('workflowChoice').style.display = 'none';

    document.querySelectorAll('.workflow-section').forEach(function(section) {
        section.classList.remove('active');
    });

    document.getElementById('workflow-' + workflow).classList.add('active');

    if (workflow === 'ai') {
        initializeAIWorkflow();
    } else if (workflow === 'files') {
```

```
        initializeFilesWorkflow();
    }
}
```

4.4.2 A0 Harmonization Algorithm

The harmonization algorithm finds common SNPs between exposure and outcome datasets.
If alleles are reversed, the outcome beta is flipped.

javascript

```
function harmonizeData(exposureData, outcomeData) {
    var commonRsids = Object.keys(exposureData).filter(function(rsid) {
        return outcomeData[rsid];
    });

    var snps = [];
    var betaX = [];
    var betaY = [];
    var betaXse = [];
    var betaYse = [];

    for (var i = 0; i < commonRsids.length; i++) {
        var rsid = commonRsids[i];
        var exp = exposureData[rsid];
        var out = outcomeData[rsid];

        var betaYValue = out.beta;

        if (
            exp.ea &&
            exp.nea &&
            out.ea &&
            out.nea &&
            exp.ea === out.nea &&
            exp.nea === out.ea
        ) {
            betaYValue = -betaYValue;
        }

        snps.push(rsid);
        betaX.push(exp.beta);
        betaY.push(betaYValue);
        betaXse.push(exp.se);
        betaYse.push(out.se);
    }

    return {
```

```
        snps: snps,
        betaX: betaX,
        betaY: betaY,
        betaXse: betaXse,
        betaYse: betaYse
    };
}
```

4.4.3 A0 Correlation Matrix Algorithm

The system supports three possible correlation matrix sources:

1. AI-fetched matrix
2. Public database matrix
3. Identity matrix fallback

If all external fetching fails, the program uses:

javascript

```
function createIdentityMatrix(snps) {
    var n = snps.length;
    var matrix = [];

    for (var i = 0; i < n; i++) {
        var row = [];

        for (var j = 0; j < n; j++) {
            row.push(i === j ? 1.0 : 0.0);
        }

        matrix.push(row);
    }

    return {
        matrix: matrix,
        source: 'identity',
        method: 'fallback'
    };
}
```

4.4.4 A0 Final JSON Generation

After harmonization and correlation matrix fetching, the final JSON is generated:

javascript

```
var finalJson = {
    snps: harmonized.snps,
    betaX: harmonized.betaX,
    betaY: harmonized.betaY,
    betaXse: harmonized.betaXse,
    betaYse: harmonized.betaYse,
    correlation: correlation.matrix,
```

```
    exposure: 'Exposure',
    outcome: 'Outcome',
    n: harmonized.snps.length,
    metadata: {
        harmonized_snps: harmonized.snps.length,
        correlation_source: correlation.source,
        timestamp: new Date().toISOString()
    }
};
```

Then the global variables are assigned:

javascript

```
setGlobalDataFromJSON(finalJson);
```

4.5 Step B1 Algorithm: MR Type Selection

The user selects:

text

Single-variable MR

or

Multi-variable MR

The corresponding section is displayed by:

javascript

```
function toggleMRType() {
    var isSingle =
        document.querySelector('input[name="mrTypeMain"]:checked').value === 'single';

    document.getElementById('singleMRSection').style.display =
        isSingle ? 'block' : 'none';

    document.getElementById('multiMRSection').style.display =
        isSingle ? 'none' : 'block';
}
```

The default option is:

text

Single-variable MR

4.6 Step B2A: Single-Variable MR Algorithm

4.6.1 Single-Variable MR Method Registry

The system supports 12 single-variable MR methods.

Each method is defined by:

text

method key

display name

external JS file path

function name

javascript

```
var SINGLE_METHOD_CONFIG = {
    ivw: {
        name: 'Inverse-Variance Weighted Method (IVW) MR',
        file: './mr_methods/mr_ivw.js',
        func: 'mr_ivw'
    },
    divw: {
        name: 'Debiased Inverse Variance Weighted Method (dIVW) MR',
        file: './mr_methods/mr_divw.js',
        func: 'mr_divw'
    },
    pivw: {
        name: 'Penalized Inverse-Variance Weighted Method (PIVW) MR',
        file: './mr_methods/mr_pivw.js',
        func: 'mr_pivw'
    },
    median: {
        name: 'The Weighted Median Method (WM) MR',
        file: './mr_methods/mr_median.js',
        func: 'mr_median'
    },
    cml: {
        name: 'Constrained Maximum Likelihood Method (cML) MR',
        file: './mr_methods/mr_cML.js',
        func: 'mr_cML'
    },
    conmix: {
        name: 'Contamination Mixture Model (Conmix) MR',
        file: './mr_methods/mr_conmix.js',
        func: 'mr_conmix'
    },
    egger: {
        name: 'Egger Method (Egger) MR',
        file: './mr_methods/mr_egger.js',
        func: 'mr_egger'
    },
    hetpen: {
        name: 'Heterogeneity Penalization Model (Hetpen) MR',
        file: './mr_methods/mr_hetpen.js',
        func: 'mr_hetpen'
    },
```

```
    lasso: {
        name: 'Lasso Method (Lasso) MR',
        file: './mr_methods/mr_lasso.js',
        func: 'mr_lasso'
    },
    loo: {
        name: 'Leave-One-Out (LOO) Method for IVW MR',
        file: './mr_methods/mr_loo.js',
        func: 'mr_loo'
    },
    maxlik: {
        name: 'Maximum Likelihood Method (MaxLik) MR',
        file: './mr_methods/mr_maxlik.js',
        func: 'mr_maxlik'
    },
    mbe: {
        name: 'Mode-Based Estimation Method (MBE) MR',
        file: './mr_methods/mr_mbe.js',
        func: 'mr_mbe'
    }
};
```

4.6.2 Dynamic Script Loading

MR method files are loaded only when selected by the user.

javascript

```
function loadScript(src) {
    return new Promise(function(resolve, reject) {
        if (loadedScripts[src]) {
            resolve();
            return;
        }

        var script = document.createElement('script');
        script.src = src;

        script.onload = function() {
            loadedScripts[src] = true;
            resolve();
        };

        script.onerror = function() {
            reject(new Error('Failed to load: ' + src));
        };

        document.head.appendChild(script);
```

```
    });
}
```

This prevents loading unnecessary method scripts and allows modular expansion.

4.6.3 Single-Variable Parameter Collection

For each selected method, the program reads the parameter panel values and constructs the method-specific argument object.

Example for IVW:

javascript

```
case 'ivw':
    p.object = obj;
    p.model = document.getElementById('ivw_model').value;
    p.robust = document.getElementById('ivw_robust').value === 'true';
    p.penalized = document.getElementById('ivw_penalized').value === 'true';
    p.weights = document.getElementById('ivw_weights').value;
    p.correl =
        document.getElementById('ivw_corr').value === 'use' && corr
            ? corr
            : 'NA';
    p.alpha = parseFloat(document.getElementById('ivw_alpha').value);
    obj.alpha = p.alpha;
    break;
```

The default values follow the design rule:

text

```
alpha = 0.001
model = RANDOM
correlation = NA
```

4.6.4 Single-Variable Method Invocation

Each selected method is called using the exact function signature expected by its external JavaScript file.

Example:

javascript

```
function callSingleMethod(methodKey, params) {
    var funcName = SINGLE_METHOD_CONFIG[methodKey].func;
    var fn = window[funcName];

    if (typeof fn !== 'function') {
        throw new Error('Function ' + funcName + ' not found.');
    }

    var p = params;

    switch (methodKey) {
        case 'ivw':
```

```
            return fn(
                p.object,
                p.model,
                p.robust,
                p.penalized,
                p.weights,
                p.correl,
                p.alpha
            );

        case 'divw':
            return fn(p.object, p.alpha, p.lambda);

        case 'median':
            return fn(p.object, p.type, p.iterations, p.alpha);

        case 'egger':
            return fn(
                p.object,
                p.model,
                p.robust,
                p.penalized,
                p.weights,
                p.correl,
                p.alpha
            );

        default:
            throw new Error('Unknown method: ' + methodKey);
    }
}
```

4.6.5 Single-Variable MR Execution Flow

The main function is:

javascript

```
async function runSingleMR()
```

Its algorithm is:

text

```
1. Check whether A0 JSON data exists.
2. Read selected single-variable MR methods.
3. For each selected method:
    a. Load the external JS file.
    b. Read method-specific parameters.
    c. Call the corresponding MR function.
```

d. Extract estimate and standard error.
e. Store results for Q-test, MetaMR, and forest plot.
4. Print all method outputs into one textarea.
5. If at least two usable methods exist:
a. Run Qtest().
b. Run MetaMR().
6. Append pooled MetaMR result to forest plot matrix.
7. Draw forest plot.

Simplified implementation:
javascript

```
async function runSingleMR() {
    var selectedMethods = [];

    document
        .querySelectorAll('#singleMethodCheckboxes input[type="checkbox"]:checked')
        .forEach(function(cb) {
            selectedMethods.push(cb.value);
        });

    var qtestData = [];
    var estMatrixNames = [];
    var estMatrixEstimates = [];
    var estMatrixCILower = [];
    var estMatrixCIUpper = [];
    var estMatrixSE = [];

    for (var mi = 0; mi < selectedMethods.length; mi++) {
        var methodKey = selectedMethods[mi];
        var config = SINGLE_METHOD_CONFIG[methodKey];

        await loadScript(config.file);

        var params = getSingleMethodParams(methodKey);
        var result = callSingleMethod(methodKey, params);
        var extracted = extractEstimateAndSE(result, methodKey);

        if (extracted) {
            qtestData.push([mi + 1, extracted.estimate, extracted.se]);

            estMatrixNames.push(config.name);
            estMatrixEstimates.push(extracted.estimate);
            estMatrixSE.push(extracted.se);
        }
```

```
    }

    if (qtestData.length >= 2) {
        var qResult = Qtest(qtestData, 0.1);
        var poolResult = MetaMR(qtestData, 0.05, 0.10);
    }
}
```

4.7 Step B2B: Multi-Variable MR Algorithm

4.7.1 Multi-Variable MR Method Registry

The system supports 7 multi-variable MR methods:

```
javascript
var MULTI_METHOD_CONFIG = {
    mvivw: {
        name: 'Multivariable Inverse-Variance Weighted Method (MVIVW) MR',
        file: './mr_methods/mr_mvivw.js',
        func: 'mr_mvivw'
    },
    mvmedian: {
        name: 'Multivariable Median Method (MVMedian) MR',
        file: './mr_methods/mr_mvmedian.js',
        func: 'mr_mvmedian'
    },
    mvcml: {
        name: 'Multivariable Constrained Maximum Likelihood Method (MVcML) MR',
        file: './mr_methods/mr_mvcML.js',
        func: 'mr_mvcML'
    },
    mvegger: {
        name: 'Multivariable Egger Method (MVEgger) MR',
        file: './mr_methods/mr_mvegger.js',
        func: 'mr_mvegger'
    },
    mvlasso: {
        name: 'Multivariable Lasso Method (MVLasso) MR',
        file: './mr_methods/mr_mvlasso.js',
        func: 'mr_mvlasso'
    },
    mvgmm: {
        name: 'Multivariable Generalized Method of Moments (MVGMM) MR',
        file: './mr_methods/mr_mvgmm.js',
        func: 'mr_mvgmm'
    },
    mvpcgmm: {
        name: 'Multivariable PC-GMM (MVPCGMM) MR',
```

```
        file: './mr_methods/mr_mvpcgmm.js',
        func: 'mr_mvpcgmm'
    }
};
```

4.7.2 Multi-Variable Input Preparation

If the JSON data contains a one-dimensional betaX, it is wrapped into a two-dimensional array.

```
javascript
if (!Array.isArray(globalBetaX[0])) {
    betaX2D = globalBetaX.map(function(v) {
        return [v];
    });

    betaXse2D = globalBetaXse.map(function(v) {
        return [v];
    });
}
```

This makes the data compatible with MVMRInput.

4.7.3 Multi-Variable Parameter Collection

Example for MVIVW:

```
javascript
case 'mvivw':
    p.object = obj;
    p.model = document.getElementById('mvivw_model').value;
    p.robust = document.getElementById('mvivw_robust').value === 'true';
    p.correl =
        document.getElementById('mvivw_corr').value === 'use' && corr
            ? corr
            : 'NA';
    p.alpha = parseFloat(document.getElementById('mvivw_alpha').value);
    obj.alpha = p.alpha;
    break;
```

4.7.4 Multi-Variable Method Invocation

```
javascript
function callMultiMethod(methodKey, params) {
    var funcName = MULTI_METHOD_CONFIG[methodKey].func;
    var fn = window[funcName];

    if (typeof fn !== 'function') {
        throw new Error(
            'Function ' + funcName + ' not found. Check that ' +
            MULTI_METHOD_CONFIG[methodKey].file + ' is loaded.'
        );
    }
```

```
    var p = params;

    switch (methodKey) {
        case 'mvivw':
            return fn(p.object, p.model, p.robust, p.correl, p.alpha);

        case 'mvmedian':
            return fn(p.object, p.type, p.iterations, p.alpha);

        case 'mvcml':
            return fn(
                p.object,
                p.MA,
                p.DP,
                p.random_start,
                p.num_pert,
                p.random_start_pert,
                p.maxit,
                p.alpha
            );

        case 'mvegger':
            return fn(
                p.object,
                p.model,
                p.robust,
                p.orientate,
                p.correl,
                p.alpha
            );

        case 'mvlasso':
            return fn(p.object, p.lambda, p.correl, p.alpha);

        case 'mvgmm':
            return fn(p.object, p.model, p.correl, p.alpha);

        case 'mvpcgmm':
            return fn(p.object, p.model, p.correl, p.r, p.alpha);

        default:
            throw new Error('Unknown method: ' + methodKey);
    }
```

```
}
```

4.7.5 Multi-Variable MR Execution Flow

The function:

```
javascript
async function runMultiMR()
```

performs:

```
text
1. Validate global JSON data.
2. Read selected multi-variable MR methods.
3. Convert betaX and betaXse into 2D arrays if necessary.
4. For each selected method:
    a. Load method JS file.
    b. Read parameters.
    c. Execute method.
    d. Extract one or more exposure-specific estimates.
    e. Store estimates for output and forest plot.
5. Run Qtest if at least two usable estimates exist.
6. Run MetaMR if at least two usable estimates exist.
7. Draw multivariable forest plot.
8. Allow Word export.
```

4.8 Estimate Extraction Algorithm

Different MR method files may return different output formats.

Therefore, the program uses a flexible extractor.

```
javascript
function extractEstimateAndSE(result, methodKey) {
    if (!result) return null;

    var estimate = null;
    var se = null;
    var ciLower = null;
    var ciUpper = null;
    var pvalue = null;

    var estKeys = [
        'Estimate',
        'estimate',
        'causalEffect',
        'theta',
        'beta',
        'Effect',
        'coef',
        'Coefficient'
    ];
```

```
    var seKeys = [
        'StdError',
        'stdError',
        'se',
        'SE',
        'standardError',
        'StdErr',
        'std_error'
    ];

    for (var i = 0; i < estKeys.length; i++) {
        if (result[estKeys[i]] !== undefined && result[estKeys[i]] !== null) {
            estimate = result[estKeys[i]];
            break;
        }
    }

    for (var j = 0; j < seKeys.length; j++) {
        if (result[seKeys[j]] !== undefined && result[seKeys[j]] !== null) {
            se = result[seKeys[j]];
            break;
        }
    }

    if (Array.isArray(estimate)) {
        estimate = estimate[0];
        se = Array.isArray(se) ? se[0] : se;
    }

    if (estimate === null || isNaN(estimate)) return null;

    if (se === null || isNaN(se) || se <= 0) {
        return null;
    }

    return {
        estimate: estimate,
        se: se,
        ciLower: ciLower,
        ciUpper: ciUpper,
        pvalue: pvalue
    };
}
```

Purpose

This allows the platform to integrate heterogeneous MR method files without requiring every method to return identical object keys.

4.9 Q-Test Algorithm

The Qtest() function calculates Cochran's Q statistic across selected MR method estimates.

Input format:

javascript

```
[
    [methodId, estimate, standardError],
    [methodId, estimate, standardError],
    ...
]
```

Core algorithm:

javascript

```
function Qtest(x, p) {
    if (p === undefined) p = 0.1;

    var causalEffects = x.map(function(a) {
        return a[1];
    });

    var inverseVariances = x.map(function(a) {
        return 1 / Math.pow(a[2], 2);
    });

    var k = x.length;
    var weightedSumOfCausalEffects = 0;
    var sumOfInverseVariances = 0;

    for (var i = 0; i < k; i++) {
        weightedSumOfCausalEffects +=
            causalEffects[i] * inverseVariances[i];

        sumOfInverseVariances += inverseVariances[i];
    }

    var theta =
        weightedSumOfCausalEffects / sumOfInverseVariances;

    var Q = 0;

    for (var j = 0; j < k; j++) {
```

```
        Q += inverseVariances[j] *
            Math.pow(causalEffects[j] - theta, 2);
    }

    var degreesOfFreedom = k - 1;
    var pValue = 1 - jStat.chisquare.cdf(Q, degreesOfFreedom);

    return {
        Q: Q,
        pValue: pValue
    };
}
```

Interpretation

text

If p-value < 0.1:

    significant heterogeneity exists.

Else:

    no significant heterogeneity is detected.

4.10 MetaMR Pooling Algorithm

The MetaMR() function pools estimates from multiple MR algorithms.

Input format:

javascript

```
[
    [1, estimate1, se1],
    [2, estimate2, se2],
    [3, estimate3, se3]
]
```

For multi-variable MR, the estimate and SE may be vectors:

javascript

```
[
    [1, [theta11, theta12], [se11, se12]],
    [2, [theta21, theta22], [se21, se22]]
]
```

4.10.1 Homogeneous Pooled Estimate

For each estimate:

text

$weight_i = 1 / SE_i^2$

The pooled causal effect is:

text

$\theta_A = \Sigma(weight_i \times \theta_i) / \Sigma(weight_i)$

The homogeneous pooled standard error is:
text
$SE_{homo} = 1 / sqrt(\Sigma\ weight_i)$

4.10.2 Heterogeneity Test Inside MetaMR

The Q statistic is:
text
$Q = \Sigma\ weight_i \times (\theta_i - \theta_A)^2$

Degrees of freedom:
text
$df = N - 1$

P-value:
text
$p = 1 - \chi^2 cdf(Q, df)$

4.10.3 Heterogeneous Model

If heterogeneity is significant:
text
$p <$ heterogeneityThreshold

then the algorithm estimates:
text
$\tau_A^2 = max(0, (Q - df) / \Sigma\ weight_i)$

and inflates the pooled standard error:
text
$SE_{heter} = sqrt(\tau_A^2 + SE_{homo}^2)$

4.10.4 Core MetaMR Computation

```
javascript
function _computeSingleElement(
    effects,
    ses,
    z_alpha_half,
    alpha,
    heterogeneityThreshold,
    N
) {
    var weights = ses.map(function(se) {
        return 1 / (se * se);
    });
```

```
var weightedSum = 0;
var sumWeights = 0;

for (var i = 0; i < effects.length; i++) {
    weightedSum += effects[i] * weights[i];
    sumWeights += weights[i];
}

var pooledEffect = weightedSum / sumWeights;
var pooledSE_homo = 1 / Math.sqrt(sumWeights);

var pooledCI_homo = [
    pooledEffect - z_alpha_half * pooledSE_homo,
    pooledEffect + z_alpha_half * pooledSE_homo
];

var Q = 0;

for (var j = 0; j < weights.length; j++) {
    var d = effects[j] - pooledEffect;
    Q += weights[j] * d * d;
}

var df = N - 1;
var pValue = 1 - jStat.chisquare.cdf(Q, df);

var isHeterogeneous = pValue < heterogeneityThreshold;

var result = {
    pooledEffect: pooledEffect,
    pooledSE_homo: pooledSE_homo,
    pooledCI_homo: pooledCI_homo,
    isHeterogeneous: isHeterogeneous,
    pooledSE_final: pooledSE_homo,
    pooledCI_final: pooledCI_homo
};

if (isHeterogeneous) {
    var tauSquared_A = Math.max(0, (Q - df) / sumWeights);
    var pooledSE_heter =
        Math.sqrt(tauSquared_A + pooledSE_homo * pooledSE_homo);

    result.tauSquared_A = tauSquared_A;
    result.pooledSE_heter = pooledSE_heter;
```

```
        result.pooledCI_heter = [
            pooledEffect - z_alpha_half * pooledSE_heter,
            pooledEffect + z_alpha_half * pooledSE_heter
        ];

        result.pooledSE_final = pooledSE_heter;
        result.pooledCI_final = result.pooledCI_heter;
    }

    return result;
}
```

4.11 Forest Plot Algorithm

The forest plot is generated by:

```
javascript
mrForest()
```

It supports both:

text
Single-variable MR
Multi-variable MR

It also supports four plot types:

Type	Meaning
Type 1	SNPs + Methods
Type 2	Methods Only
Type 3	SNPs Only
Type 4	Sorted SNPs + Methods

4.11.1 Forest Plot Inputs

```
javascript
mrForest(
    object,
    alpha,
    snp_estimates,
    snp_methods,
    estmatrix,
    ordered,
    isMVMR,
    targetDiv
)
```

Parameters

text
object MRInput or MVMRInput object

alpha significance level
snp_estimates whether SNP Wald estimates are plotted
snp_methods whether MR method estimates are plotted
estmatrix method-level estimates and intervals
ordered whether to sort estimates
isMVMR whether the plot is multivariable
targetDiv plot container ID

4.11.2 SNP-Level Wald Ratio Estimates

For single-variable MR, SNP-level Wald ratio is:

text

$estimate_i = betaY_i / betaX_i$

The confidence interval is approximately:

text

$estimate_i \pm z_{(1-alpha/2)} \times betaYse_i / |betaX_i|$

Code:

```
javascript
var estimate = betaY[i] / betaX[i];

var ciLower =
    estimate -
    (CI_range * betaYse[i]) / Math.abs(betaX[i]);

var ciUpper =
    estimate +
    (CI_range * betaYse[i]) / Math.abs(betaX[i]);
```

4.11.3 Method-Level Estimate Matrix

The method-level estimate matrix is structured as:

```
javascript
[
    methodNames,
    estimates,
    ciLowerValues,
    ciUpperValues,
    standardErrors
]
```

Example:

```
javascript
lastSingleEstMatrix = [
    estMatrixNames,
    estMatrixEstimates,
```

```
    estMatrixCILower,
    estMatrixCIUpper,
    estMatrixSE
];
```

4.11.4 Plotly Rendering

The final plot is rendered using Plotly:

javascript

```
Plotly.newPlot(targetDiv, [trace], layout);
```

Core trace:

javascript

```
var trace = {
    x: dframe.map(function(d) {
        return d.estimate;
    }),
    y: dframe.map(function(_, i) {
        return i;
    }),
    error_x: {
        type: 'data',
        array: dframe.map(function(d) {
            return d.CI_upper - d.estimate;
        }),
        arrayminus: dframe.map(function(d) {
            return d.estimate - d.CI_lower;
        }),
        color: 'rgba(0,0,255,0.5)',
        thickness: 2,
        width: 4
    },
    mode: 'markers',
    type: 'scatter',
    marker: {
        size: 6,
        color: 'rgb(31,19,255)'
    },
    showlegend: false
};
```

4.12 Word Export Algorithm

The export module collects:

text

textarea result content

forest plot SVG

timestamp

title

and creates a Word-compatible HTML document.

javascript

```
function exportToWord(textContent, title, filename) {
    var html =
        '<html xmlns:o="urn:schemas-microsoft-com:office:office" ' +
        'xmlns:w="urn:schemas-microsoft-com:office:word" ' +
        'xmlns="http://www.w3.org/TR/REC-html40">';

    html += '<head><meta charset="utf-8"><title>' + title + '</title>';
    html += '<style>';
    html += 'body{font-family:"Courier New",monospace;font-size:10pt;white-space:pre-wrap;}';
    html += 'h1{font-family:Arial;font-size:16pt;color:#333;}';
    html += '</style></head>';

    html += '<body>';
    html += '<h1>' + title + '</h1>';
    html += '<p>Generated: ' + new Date().toISOString() + '</p>';
    html += '<hr>';
    html += '<pre>' +
        textContent.replace(/</g, '&lt;').replace(/>/g, '&gt;') +
        '</pre>';

    html += '</body></html>';

    var blob = new Blob(['\ufeff', html], {
        type: 'application/msword'
    });

    var url = URL.createObjectURL(blob);
    var a = document.createElement('a');

    a.href = url;
    a.download = filename;

    document.body.appendChild(a);
    a.click();
    document.body.removeChild(a);

    URL.revokeObjectURL(url);
}
```

4.13 Complete Algorithmic Flow

4.13.1 Full Single-Variable MR Flow

text

```
Start
 ↓
User chooses A0 workflow
 ↓
Generate or validate JSON
 ↓
Store betaX, betaY, betaXse, betaYse, snps, correlation globally
 ↓
User clicks “Proceed to MR Analysis”
 ↓
User selects Single-variable MR
 ↓
User selects one or more of 12 MR methods
 ↓
User configures method parameters
 ↓
For each selected method:
    Load ./mr_methods/[method].js
    Build MRInput object
    Read method-specific parameters
    Call method function
    Extract estimate and SE
    Append raw and formatted output to textarea
 ↓
If at least 2 valid estimates:
    Run Qtest()
    Run MetaMR()
 ↓
Build estimate matrix
 ↓
Draw forest plot with mrForest()
 ↓
User optionally changes forest plot type
 ↓
User exports output to Word
End
```

4.13.2 Full Multi-Variable MR Flow

text

```
Start
 ↓
User completes A0 JSON generation
```

↓
User selects Multi-variable MR
↓
Program checks betaX dimensionality
↓
If betaX is 1D:
    Convert betaX to 2D
    Convert betaXse to 2D
↓
User selects one or more of 7 MVMR methods
↓
User configures method parameters
↓
For each selected method:
    Load ./mr_methods/[method].js
    Build MVMRInput object
    Read method-specific parameters
    Call method function
    Extract exposure-specific estimate(s)
    Store output
↓
If at least 2 valid method estimates:
    Run Qtest()
    Run MetaMR()
↓
Draw multivariable forest plot
↓
Export Word file
End

4.14 Important Design Features

4.14.1 Modular MR Method Integration

The platform does not hard-code the mathematical internals of the 12 single-variable and 7 multi-variable MR methods. Instead, it dynamically loads external files:

text
./mr_methods/mr_ivw.js
./mr_methods/mr_divw.js
./mr_methods/mr_pivw.js
...
./mr_methods/mr_mvpcgmm.js

This makes the platform extensible.

4.14.2 Separation Between Data Preparation and MR Analysis

Step A0 is responsible only for generating harmonized JSON.
Step B is responsible for MR computation.

This separation allows the same MR engine to be used regardless of whether the data came from:
text
Direct JSON
Uploaded files
AI-generated data
GWAS API data

4.14.3 Robust Output Extraction

Because different MR methods may return different property names, the extraction layer searches many possible keys:
javascript

```
var estKeys = [
    'Estimate',
    'estimate',
    'causalEffect',
    'theta',
    'beta',
    'Effect',
    'coef',
    'Coefficient'
];
```

This increases compatibility with independently written MR scripts.

4.14.4 Built-In Meta-Analysis

The platform does not merely run MR methods independently. It performs a second-level meta-analysis over method-level causal estimates using:
javascript

```
MetaMR()
```

This produces:
text
pooled causal effect
pooled standard error
confidence interval
heterogeneity result
recommended model

4.14.5 Interactive Visualization

The forest plot can show:
text
SNP-level Wald estimates
MR method estimates
Meta-MR pooled estimate
and can be redrawn interactively by the user.

4.15 Summary

The program implements a full browser-side Mendelian Randomization analysis system. Its algorithmic design consists of:

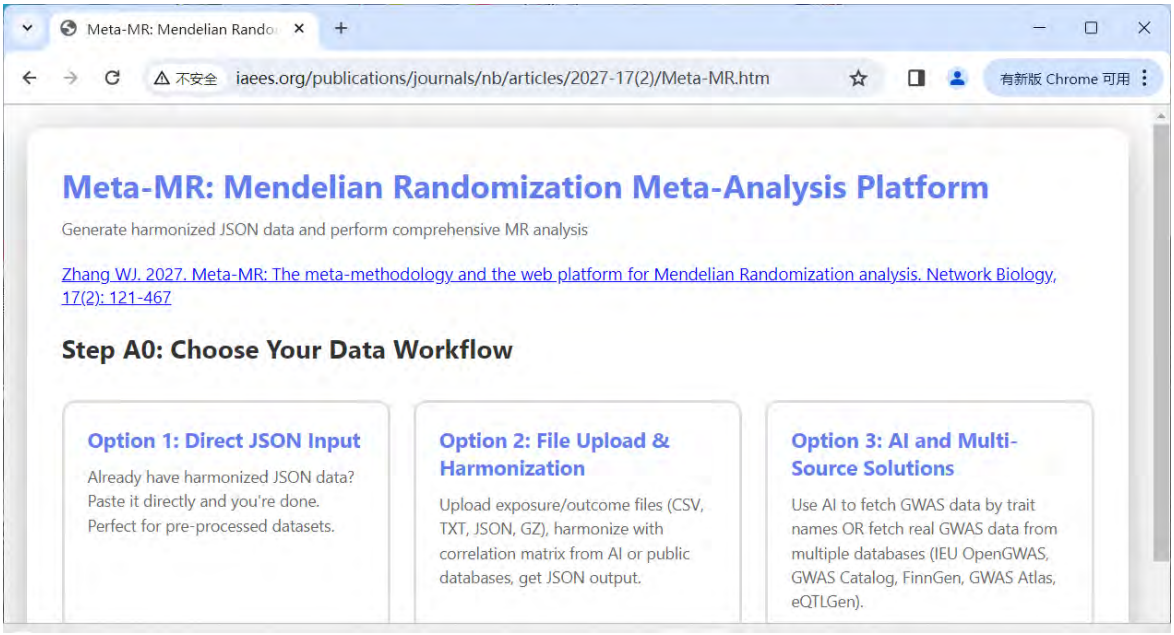
1. **A0 data generation**
   - Direct JSON validation
   - File parsing
   - AI/GWAS fetching
   - Harmonization
   - Correlation matrix generation
2. **MR type selection**
   - Single-variable MR
   - Multi-variable MR
3. **Dynamic MR execution**
   - Loads external MR method scripts from ./mr_methods/
   - Reads method-specific parameters
   - Executes selected methods
4. **Result extraction and display**
   - Extracts estimates, SEs, CIs, p-values
   - Writes all method results into one textarea
5. **Statistical synthesis**
   - Runs Qtest()
   - Runs MetaMR()
6. **Visualization**
   - Draws forest plots using mrForest()
7. **Export**
   - Saves results and plots into Word-compatible .doc files

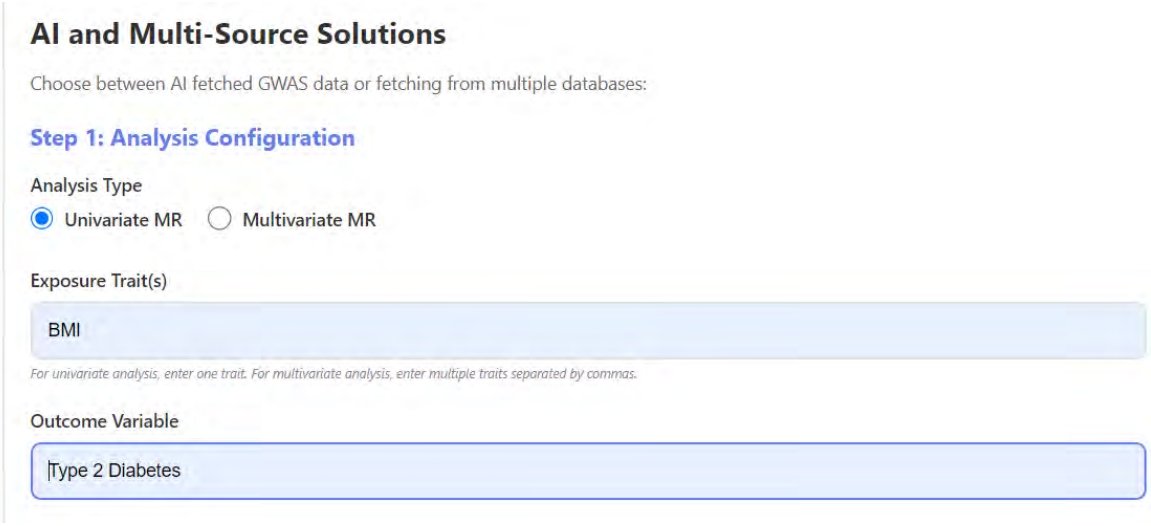
In short, the platform is an integrated data harmonization + MR method execution + Meta-MR pooling + visualization + export system for Mendelian Randomization analysis.

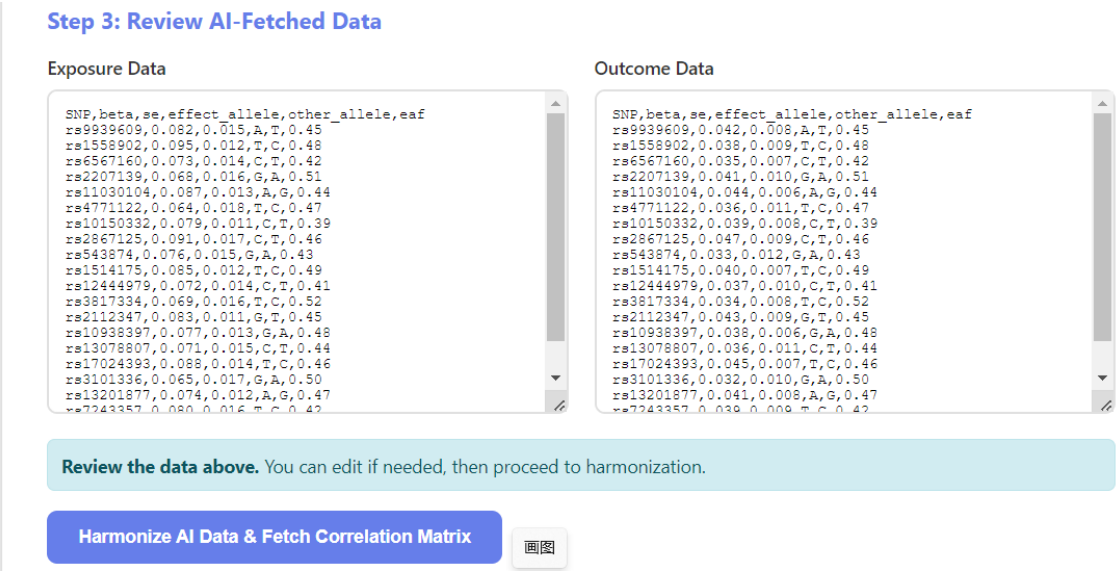
5 Web GUI of Meta-MR

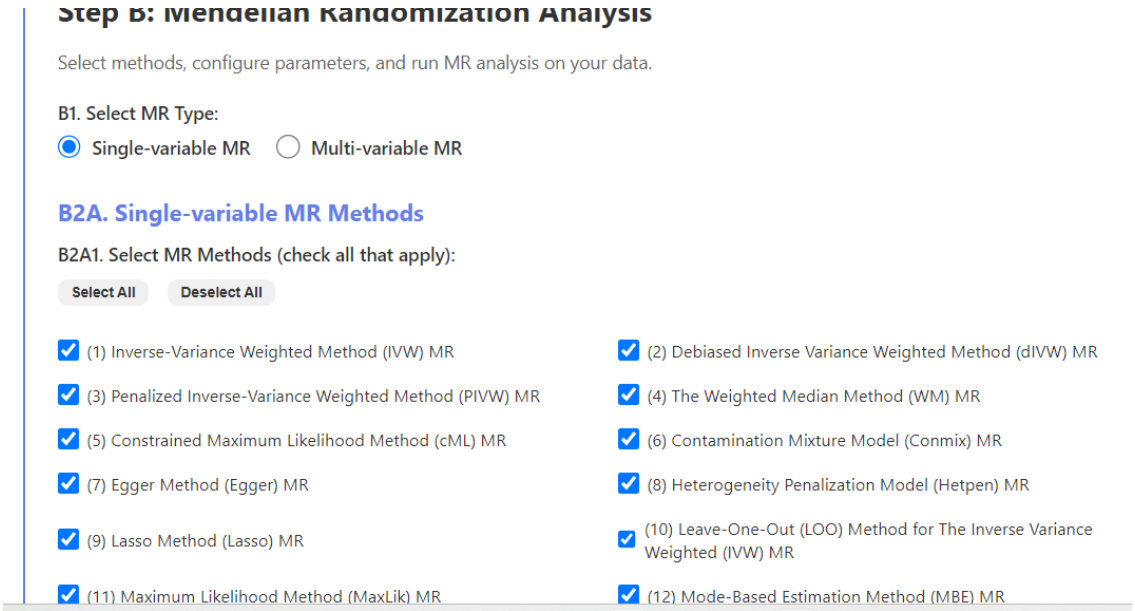
Using Meta-MR, the data formats and files can be found in the following article:
http://www.iaees.org/publications/journals/nb/articles/2026-16(4)/MR-GWAS-JSON.pdf

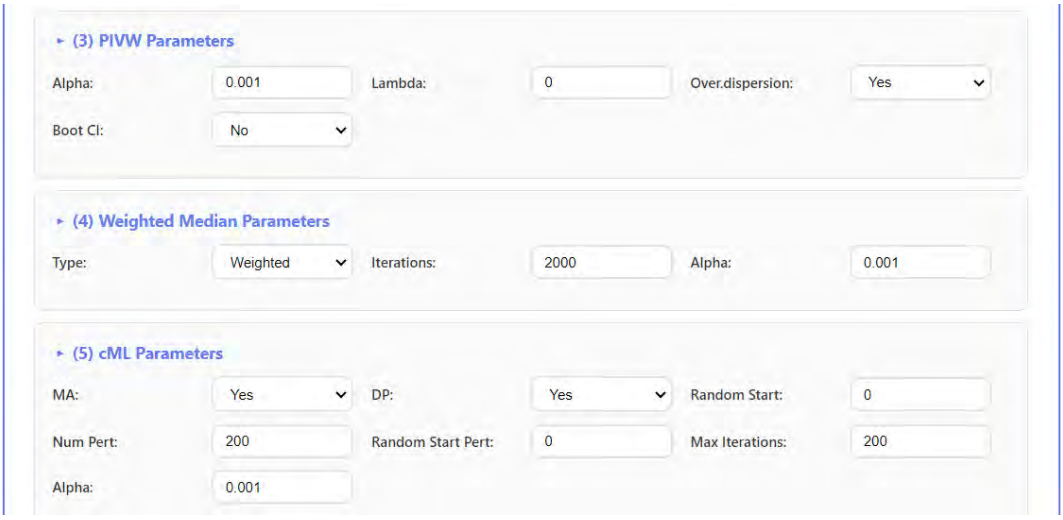
Fig. 1 indicates some of the the web GUI of Meta-MR.

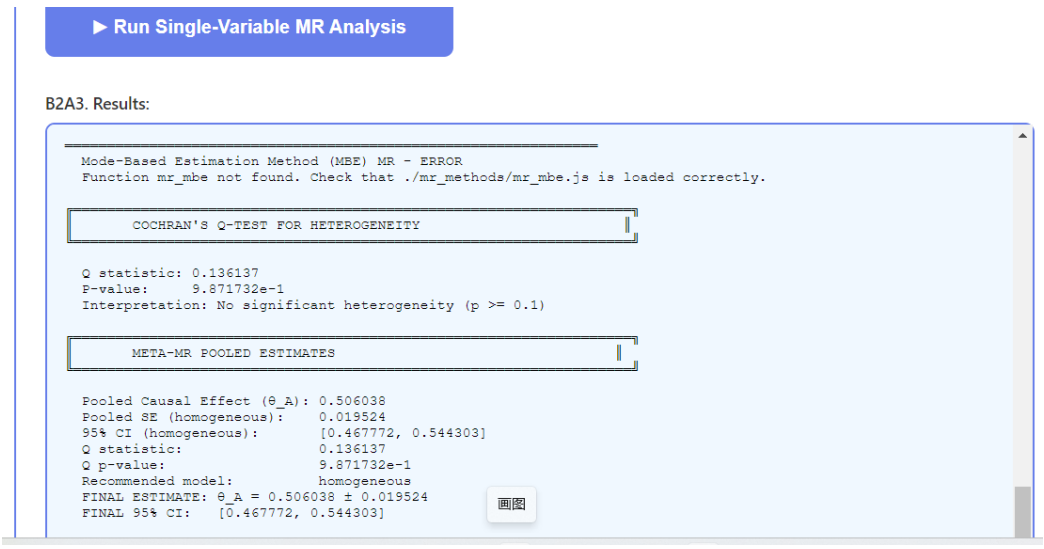

Meta-MR: Mendelian Rando
不安全
iaees.org/publications/journals/nb/articles/2027-17(2)/Meta-MR.htm
有新版 Chrome 可用
Meta-MR: Mendelian Randomization Meta-Analysis Platform
Generate harmonized JSON data and perform comprehensive MR analysis
Zhang WJ. 2027. Meta-MR: The meta-methodology and the web platform for Mendelian Randomization analysis. Network Biology, 17(2): 121-467
Step A0: Choose Your Data Workflow
Option 1: Direct JSON Input
Already have harmonized JSON data? Paste it directly and you're done. Perfect for pre-processed datasets.
Option 2: File Upload & Harmonization
Upload exposure/outcome files (CSV, TXT, JSON, GZ), harmonize with correlation matrix from AI or public databases, get JSON output.
Option 3: AI and Multi-Source Solutions
Use AI to fetch GWAS data by trait names OR fetch real GWAS data from multiple databases (IEU OpenGWAS, GWAS Catalog, FinnGen, GWAS Atlas, eQTLGen).


AI and Multi-Source Solutions
Choose between AI fetched GWAS data or fetching from multiple databases:
Step 1: Analysis Configuration
Analysis Type
Univariate MR
Multivariate MR
Exposure Trait(s)
BMI
For univariate analysis, enter one trait. For multivariate analysis, enter multiple traits separated by commas.
Outcome Variable
Type 2 Diabetes


Step 3: Review AI-Fetched Data
Exposure Data
SNP,beta,se,effect_allele,other_allele,eaf
rs9939609,0.082,0.015,A,T,0.45
rs1558902,0.095,0.012,T,C,0.48
rs6567160,0.073,0.014,C,T,0.42
rs2207139,0.068,0.016,G,A,0.51
rs11030104,0.087,0.013,A,G,0.44
rs4771122,0.064,0.018,T,C,0.47
rs10150332,0.079,0.011,C,T,0.39
rs2867125,0.091,0.017,C,T,0.46
rs543874,0.076,0.015,G,A,0.43
rs1514175,0.085,0.012,T,C,0.49
rs12444979,0.072,0.014,C,T,0.41
rs3817334,0.069,0.016,T,C,0.52
rs2112347,0.083,0.011,G,T,0.45
rs10938397,0.077,0.013,G,A,0.48
rs13078807,0.071,0.015,C,T,0.44
rs17024393,0.088,0.014,T,C,0.46
rs3101336,0.065,0.017,G,A,0.50
rs13201877,0.074,0.012,A,G,0.47
Outcome Data
SNP,beta,se,effect_allele,other_allele,eaf
rs9939609,0.042,0.008,A,T,0.45
rs1558902,0.038,0.009,T,C,0.48
rs6567160,0.035,0.007,C,T,0.42
rs2207139,0.041,0.010,G,A,0.51
rs11030104,0.044,0.006,A,G,0.44
rs4771122,0.036,0.011,T,C,0.47
rs10150332,0.039,0.008,C,T,0.39
rs2867125,0.047,0.009,C,T,0.46
rs543874,0.033,0.012,G,A,0.43
rs1514175,0.040,0.007,T,C,0.49
rs12444979,0.037,0.010,C,T,0.41
rs3817334,0.034,0.008,T,C,0.52
rs2112347,0.043,0.009,G,T,0.45
rs10938397,0.038,0.006,G,A,0.48
rs13078807,0.036,0.011,C,T,0.44
rs17024393,0.045,0.007,T,C,0.46
rs3101336,0.032,0.010,G,A,0.50
rs13201877,0.041,0.008,A,G,0.47
Review the data above. You can edit if needed, then proceed to harmonization.
Harmonize AI Data & Fetch Correlation Matrix
画图


Step B: Mendelian Randomization Analysis
Select methods, configure parameters, and run MR analysis on your data.
B1. Select MR Type:
Single-variable MR
Multi-variable MR
B2A. Single-variable MR Methods
B2A1. Select MR Methods (check all that apply):
Select All
Deselect All
(1) Inverse-Variance Weighted Method (IVW) MR
(2) Debiased Inverse Variance Weighted Method (dIVW) MR
(3) Penalized Inverse-Variance Weighted Method (PIVW) MR
(4) The Weighted Median Method (WM) MR
(5) Constrained Maximum Likelihood Method (cML) MR
(6) Contamination Mixture Model (Conmix) MR
(7) Egger Method (Egger) MR
(8) Heterogeneity Penalization Model (Hetpen) MR
(9) Lasso Method (Lasso) MR
(10) Leave-One-Out (LOO) Method for The Inverse Variance Weighted (IVW) MR
(11) Maximum Likelihood Method (MaxLik) MR
(12) Mode-Based Estimation Method (MBE) MR


(3) PIVW Parameters
Alpha: 0.001
Lambda: 0
Over.dispersion: Yes
Boot CI: No
(4) Weighted Median Parameters
Type: Weighted
Iterations: 2000
Alpha: 0.001
(5) cML Parameters
MA: Yes
DP: Yes
Random Start: 0
Num Pert: 200
Random Start Pert: 0
Max Iterations: 200
Alpha: 0.001


▶ Run Single-Variable MR Analysis
B2A3. Results:
Mode-Based Estimation Method (MBE) MR - ERROR
Function mr_mbe not found. Check that ./mr_methods/mr_mbe.js is loaded correctly.
COCHRAN'S Q-TEST FOR HETEROGENEITY
Q statistic: 0.136137
P-value: 9.871732e-1
Interpretation: No significant heterogeneity (p >= 0.1)
META-MR POOLED ESTIMATES
Pooled Causal Effect (θ_A): 0.506038
Pooled SE (homogeneous): 0.019524
95% CI (homogeneous): [0.467772, 0.544303]
Q statistic: 0.136137
Q p-value: 9.871732e-1
Recommended model: homogeneous
FINAL ESTIMATE: θ_A = 0.506038 ± 0.019524
FINAL 95% CI: [0.467772, 0.544303]
画图

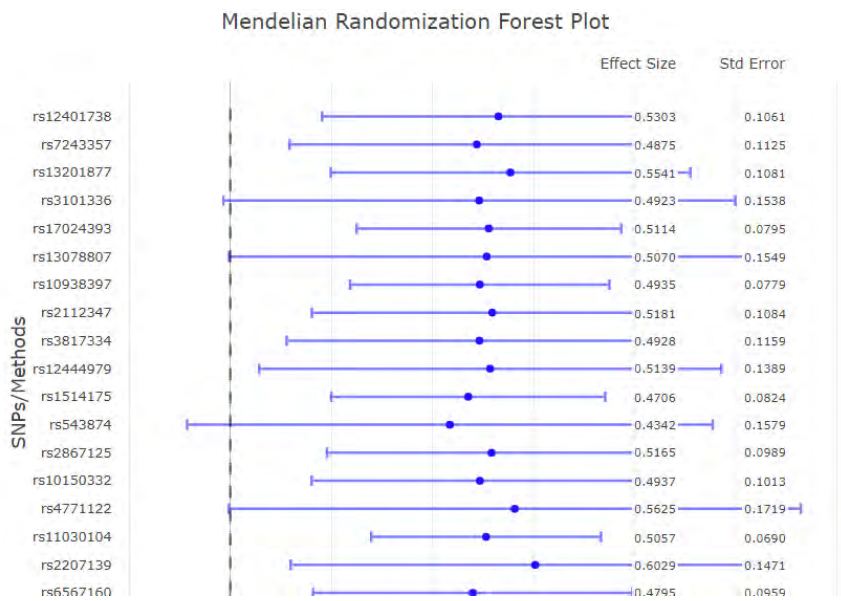


Fig. 1 Web GUI of Meta-MR.

6 Discussion

6.1 Innovations

The present study makes several significant contributions to the field of Mendelian Randomization:

(1) I provide a comprehensive mathematical exposition of twelve single-variable and seven multivariable MR algorithms, presenting their key assumptions, procedural steps, and core formulae in a unified framework. This systematic formalization facilitates understanding of the underlying mechanics, enabling researchers to make informed methodological choices.

(2) I introduce the concept of algorithm-level meta-analysis to MR, extending the established principles of meta-analysis from study-level to method-level synthesis. The pooled estimates from multiple MR algorithms offer enhanced robustness compared to reliance on any single method. As demonstrated by Huang (2023) and Zhang (2024a), algorithm averaging outperforms individual algorithms in terms of bias and efficiency when the biases of individual algorithms are approximately uniformly distributed about zero.

(3) The Meta-MR web platform represents a practical innovation in MR software. Unlike existing R-based tools that require programming expertise and local installation, Meta-MR is entirely browser-based, requiring no software installation, and provides an intuitive point-and-click interface. The platform supports both single-variable and multivariable MR, dynamic loading of external method scripts, and comprehensive result visualization through interactive forest plots.

(4) The integration of AI-assisted GWAS data acquisition represents a forward-looking feature that addresses the challenges of data accessibility. The ability to simulate realistic genetic data using AI models (Zhang, 2026a; Zhang and Qi, 2026) enables researchers to experiment with exposure-outcome pairs and validate analysis pipelines without requiring access to real summary statistics. This is particularly valuable for educational purposes, method development, and preliminary analyses.

(5) The platform incorporates the heterogeneity penalization model and constrained maximum likelihood approaches that enhance robustness to pleiotropy. The systematic implementation of sensitivity analyses, including leave-one-out diagnostics and heterogeneity testing, promotes transparent and rigorous causal inference.

(6) By hosting the platform as an open-access web platform, I ensure broad accessibility to researchers worldwide, democratizing advanced MR analysis capabilities beyond institutions with specialized computational resources.

6.2 Limitations and Future Directions

Despite these contributions, several limitations warrant consideration and suggest directions for future development.

(1) Technical limitations: The current implementation relies on browser-based JavaScript, which imposes constraints on computational performance for analyses involving large numbers of SNPs or intensive bootstrap procedures. While the platform handles typical MR scenarios adequately, future development may consider WebAssembly or server-side computation for more demanding analyses. Additionally, the correlation matrix handling, particularly for large SNP sets, may benefit from optimized algorithms to improve scalability.

(2) Method coverage: Although I have implemented nineteen MR methods, the field continues to evolve rapidly with the development of novel approaches for addressing pleiotropy, weak instruments, and other challenges. Future updates should incorporate emerging methods, particularly those addressing multi-ancestry analyses and integrating functional genomic annotations.

(3) Data integration: The current platform supports harmonization of exposure and outcome data from standard formats but does not yet provide direct integration with major GWAS databases beyond the IEU OpenGWAS API. Extending the multi-source GWAS fetcher to comprehensively query GWAS Catalog, GWAS Atlas, eQTLGen, FinnGen, and the All of Us Research Program (All of Us Research Program Genomics Investigators, 2024) would enhance utility.

(4) Pleiotropy handling: While the implemented methods provide various approaches to pleiotropy, the detection and correction of horizontal pleiotropy remain challenging. Future versions could incorporate more sophisticated approaches, such as colocalization analysis (Henry et al., 2022) and cis-only colocalization (Rasooly et al., 2023), to strengthen causal inference when genetic variants may affect outcomes through multiple pathways.

(5) Ancestry considerations: The reliance on European-ancestry GWAS data limits generalizability to diverse populations. Future developments should support multi-ancestry analyses and provide guidance on ancestry-specific instrument selection, addressing an important equity concern in genetic epidemiology.

(6) Statistical considerations: The current implementation uses alpha thresholds of 0.001 or 0.05 for hypothesis testing and confidence intervals. Following the recommendations of Zhang (2022a-c, 2024a-c), future iterations may offer more flexible significance thresholds and promote the "new statistics" paradigm emphasizing effect sizes, confidence intervals, and replication over dichotomous significance testing.

(7) Validation and benchmarking: Systematic benchmarking of the Meta-MR pooling approach against established simulation frameworks and empirical examples would strengthen the evidence base for its use. While the theoretical foundations are sound, large-scale simulation studies comparing pooled estimates to "true" causal effects under various scenarios would provide valuable guidance for users.

(8) User experience: While the current interface is functional, further refinements to workflow guidance, automated parameter selection, and interactive result interpretation would enhance user experience, particularly for less experienced researchers.

In conclusion, Meta-MR represents a significant step toward making advanced MR methodology more accessible and systematic. By formalizing algorithms and providing an integrated web platform, I aim to promote evidence synthesis, enhance reproducibility, and ultimately facilitate more reliable causal inference in biomedical research.

Acknowledgment

I am thankful to the support of Discovery and Crucial Node Analysis of Important Biological and Social Networks (2015.6-2020.6), from Yangling Institute of Modern Agricultural Standardization, China.

References

Adair JG, Vohra N. 2003. The explosion of knowledge, references, and citations. Psychology's unique response to a crisis. American Psychologist, 58(1): 15-23. https://doi.org/10.1037/0003-066x.58.1.15

All of Us Research Program Genomics Investigators. 2024. Genomic Data in the All of Us Research Program. Nature. doi: 10.1038/s41586-023-06957-x

Antonelli J, Cefalu M. 2020. Averaging causal estimators in high dimensions. Journal of Causal Inference, 8(1): 92-107. https://doi.org/10.1515/jci-2019-0017

Bangert-Drowns RL. 1986. Review of developments in meta-analytic method. Psychological Bulletin, 99(3): 388-399. https://doi.org/10.1037/0033-2909.99.3.388

Bowden J. 2016. Assessing the suitability of summary data for Mendelian randomization analyses using MR-Egger regression: The role of the I2 statistic. International Journal of Epidemiology, 45(6): 1961-1974

Bowden J, Smith GD, Burgess S. 2015. Mendelian randomization with invalid instruments: effect estimation and bias detection through Egger regression. International Journal of Epidemiology, 44: 512-525. doi: 10.1093/ije/dyv080

Bowden J, Smith GD, Haycock PC, Burgess S. 2016. Consistent estimation in Mendelian randomization with some invalid instruments using a weighted median estimator. Genetic Epidemiology, 40(4): 304-314. doi: 10.1002/gepi.21965

Burgess S, Bowden J. 2015. Integrating summarized data from multiple genetic variants in Mendelian randomization: bias and coverage properties of inverse-variance weighted methods. arXiv, 1512.04486

Burgess S, Bowden J, Dudbridge F, Thompson SG. 2016. Robust instrumental variable methods using multiple candidate instruments with application to Mendelian randomization. arXiv, 1606.03729

Burgess S, Butterworth AS, Thompson SG. 2013. Mendelian randomization analysis with multiple genetic variants using summarized data. Genetic Epidemiology, 37: 658-665. doi: 10.1002/gepi.21758

Burgess S, Scott RA, Timpson NJ, et al. 2015. Using published data in Mendelian randomization: a blueprint for efficient identification of causal risk factors. European Journal of Epidemiology, 30(7): 543-552. DOI: 10.1007/s10654-015-0011-z

Burgess S, Yavorska O, Staley J. 2023. MendelianRandomization. https://CRAN.R-project.org/package=MendelianRandomization. Access on 10 April 2024.

Cloudflare. 2025. CGNJS Project's. https://cdnjs.cloudflare.com/. Accessed Jan 25, 2025

del Greco F, Minelli C, Sheehan NA, Thompson JR. 2015. Detecting pleiotropy in Mendelian randomisation studies with summary data and a continuous outcome. Stat Med, 34(21): 2926-2940. doi: 10.1002/sim.6522

Deng MG, Liu F, Liang YH, et al. 2023. Association between frailty and depression: A bidirectional Mendelian randomization study. Science Advances, 9(38): eadi3902. doi: 10.1126/sciadv.adi3902

DePaolo J, Levin MG, Tcheandjieu T, et al. 2023. Relationship between ascending thoracic aortic diameter and blood pressure: A Mendelian randomization study. Arteriosclerosis, Thrombosis, and Vascular Biology, 43: 359-366. doi:10.1161/ATVBAHA.122.318149

European Medicines Agency (EMEA). 2006. Guideline on clinical trials in small populations.

CHMP/EWP/83561/2005. http //www.ema.europa.eu/docs/en_GB/document_library/Scientific_guideline/2009/09/WC500003615.pdf

Extance A. 2025. AI in scientific data generation: Opportunities and ethics. Nature Reviews Methods Primers, 5: 12. doi: 10.1038/s43586-024-00345-7

Fu Y, Xu F, Jiang L, Miao Z, Liang X, Yang J, et al. 2021. Circulating vitamin C concentration and risk of cancers: a Mendelian randomization study. BMC Medicine, 19(1): 171. doi:10.1186/s12916-021-02041-1

Gagne JJ, Thompson L, O'Keefe K, Kesselheim AS. 2014. Innovative research methods for studying treatments for rare diseases: Methodological review. BMJ, 349: 6802. https://doi.org/10.1136/bmj.g6802

Grant AJ, Burgess S. 2020. Pleiotropy robust methods for multivariable Mendelian randomization. arXiv, 2008.11997

GWASLab. 2024. Mendelian Randomization Series No. 1: Basic Concepts Mendelian randomization . https://gwaslab.org/2021/06/24/mr/

Hansen LP. 1982. Large sample properties of generalized method of moments estimators. Econometrica, 1029-1054

Hartwig FP, Smith GD, Bowden J. 2017. Robust inference in summary data Mendelian randomization via the zero modal pleiotropy assumption. International Journal of Epidemiology, 46(6): 1985-1998. doi: 10.1093/ije/dyx102

Hemani G, Tilling K, Davey Smith G, 2017, Orienting the causal relationship between imprecisely measured traits using GWAS summary data. PLOS Genetics, 13(11): e1007081. https://doi.org/10.1371/journal.pgen.1007081

Hemani G, Zheng J, Elsworth B, Wade KH, et al. 2018. The MR-Base Collaboration. The MR-Base platform supports systematic causal inference across the human phenome. eLife, 7: e34408. doi: 10.7554/eLife.34408

Henry A, Gordillo-Maraón M, Finan C. 2022. Therapeutic targets for heart failure identified using proteomics and Mendelian randomization. Circulation, 145(16): 1205-1217. doi: 10.1161/CIRCULATIONAHA.121.056663

https://github.com/alanarivera/

Huang HN. 2023. Combining estimators in interlaboratory studies and meta-analyses. Research Synthesis Methods, 14(3): 526-543. https://doi.org/10.1002/jrsm.1633

ivonesamplemr. https://github.com/remlapmot/ivonesamplemr

Kim EJ, Hoffmann TJ, Nah G, et al. 2021. Coffee consumption and incident tachyarrhythmias reported behavior, Mendelian randomization, and their interactions. JAMA Internal Medicine, 181(9): 1185-1193. doi: 10.1001/jamainternmed.2022.6962

Kim JY, Song M, Kim MS, et al. 2023. An atlas of associations between 14 micronutrients and 22 cancer outcomes: Mendelian randomization analyses. BMC Medicine, 21(1): 316. doi:10.1186/s12916-023-03018-y

Korn EL, McShane LM, Freidlin B. 2013. Statistical challenges in the evaluation of treatments for small patient populations. Science Translational Medicine, 2178. https://doi.org/10.1126/scitranslmed.3004018

Lavancier F, Rochet P. 2016. A general procedure to combine estimators. Computational Statistics and Data Analysis, 94: 175-192. doi:10.1016/j.csda.2015.08.001

Lin Z, Xue H, Pan W. 2023. Robust multivariable Mendelian randomization based on constrained maximum likelihood. The American Journal of Human Genetics, 110(4): 592-605

Mills MC, Barban N, Tropf FC. 2020. An Introduction to Statistical Genetic Data Analysis. MIT Press, USA

Mitra P, Lian H, Mitra R, Liang H, Xie M. 2019. A general framework for frequentist model averaging.

Science China Mathematics, 62(2): 205-226. doi:10.1007/s11425-018-9403-x

MR dictionary. https://mr-dictionary.mrcieu.ac.uk/

Mrrobust. https://github.com/remlapmot/mrrobust

OneSampleMR, https://remlapmot.github.io/OneSampleMR/

Papadimitriou N, Dimou N, Gill D, Tzoulaki I, Murphy N, Riboli E, et al. 2021. Genetically predicted circulating concentrations of micronutrients and risk of breast cancer: a Mendelian randomization study. International Journal of Cancer, 148(3): 646-653. doi: 10.1002/ijc.33246

Rasooly D, Peloso GM, Pereira AC, et al. 2023. Genome-wide association analysis and Mendelian randomization proteomics identify drug targets for heart failure. Nature Communications, 14: 3826. https://www.nature.com/articles/s41467-023-39253-3

Röver C, Berse T, Hijona M, et al. 2015. Tailoring the Bayesian framework for clinical trials in rare diseases. Pharmaceutical Statistics, 14(6): 489-499. doi: 10.1002/pst.1705

Staley JR, Blackshaw J, Kamat MA, et al. 2016. PhenoScanner: a database of human genotype–phenotype associations. Bioinformatics, 32(20): 3207-3209. https://doi.org/10.1093/bioinformatics/btw373

STROBE-MR. https://www.strobe-mr.org/

Tang JL, Yang ZY. 2015. Systematic review and meta-analysis (Chapter 14). In: Epidemiology Volume 1 (3rd ed) (Li LM, ed). People's Medical Publishing House, Beijing, China. https://www.taobao.com/list/item/753734635956.htm?spm=a21wu.10013406.taglist-content.39.43006f04W2LLpM

The OpenGWAS project. https://gwas.mrcieu.ac.uk/

TwoSampleMR. https://github.com/MRCIEU/TwoSampleMR

Wang B, Guo J, Hu X, et al. 2020. mrrobust: An R package for performing two-sample Mendelian randomization using robust regression methods. Bioinformatics, 36(20): 5034-5035. doi: 10.1093/bioinformatics/btaa688

Wang XT. 2023. Briefly Describe The Common Designs of Mendelian Randomization Analysis. https://mp.weixin.qq.com/s/iY4LXS4Rg_D7Y3tVd5xNTg. Access on 18 March 2024.

XM. 2024. Clinical Research and Medical Statistics. Mendelian Series in R language: Understanding Mendelian randomization in one article. https://mp.weixin.qq.com/s/tsvkrkPom1Gz9n4bn94swg. Access on 5 March 2024.

Xu S, Wang P, Fung WK, Liu Z. 2023. A novel penalized inverse-variance weighted estimator for Mendelian Randomization with applications to COVID-19 outcomes. Biometrics, 79(3): 2184-2195. doi: 10.1111/biom.13732

Yavorska OO, Staley JR. 2023. MendelianRandomization: An R package for performing Mendelian randomization analyses using summarized data. Bioinformatics, 39(1): btac780. doi: 10.1093/bioinformatics/btac780

Yuan S, Mason AM, Carter P, Vithayathil M, Kar S, Burgess S, et al. 2022. Selenium and cancer risk: Wide-angled Mendelian randomization analysis. International Journal of Cancer, 150(7): 1134-1140. doi:10.1002/ijc.33902

Zhang WJ. 2021a. A statistical simulation method for causality inference of Boolean variables. Network Biology, 11(4): 263-273. http://www.iaees.org/publications/journals/nb/articles/2021-11(4)/a-method-for-causality-inference-of-Boolean-variables.pdf

Zhang WJ. 2021b. Causality inference of linearly correlated variables: The statistical simulation and regression method. Computational Ecology and Software, 11(4): 154-161.

http://www.iaees.org/publications/journals/ces/articles/2021-11(4)/causality-inference-of-linearly-correlated-variables.pdf

Zhang WJ. 2021c. Causality inference of nominal variables: A statistical simulation method. Computational Ecology and Software, 11(4): 142-153.
http://www.iaees.org/publications/journals/ces/articles/2021-11(4)/causality-inference-of-nominal-variables-with-statistical-simulation-method.pdf

Zhang WJ. 2022a. Confidence intervals Concepts, fallacies, criticisms, solutions and beyond. Network Biology, 12(3): 97-115.
http://www.iaees.org/publications/journals/nb/articles/2022-12(3)/confidence-intervals-fallacies-criticisms-solutions.pdf

Zhang WJ. 2022b. Dilemma of *t*-tests: Retaining or discarding choice and solutions. Computational Ecology and Software, 12(4): 181-194.
http://www.iaees.org/publications/journals/ces/articles/2022-12(4)/dilemma-of-t-tests.pdf

Zhang WJ. 2022c. *p*-value based statistical significance tests: Concepts, misuses, critiques, solutions and beyond. Computational Ecology and Software, 12(3): 80-122.
http://www.iaees.org/publications/journals/ces/articles/2022-12(3)/p-value-based-statistical-significance-tests.pdf

Zhang WJ. 2024a. MetaAnaly: The platform-independent computational tool for meta-analysis in the paradigm of new statistics. Network Biology, 14(2): 187-214.
http://www.iaees.org/publications/journals/nb/articles/2024-14(2)/MetaAnaly.htm

Zhang W.J. 2024b. SampSizeCal: The platform-independent computational tool for sample sizes in the paradigm of new statistics. Network Biology, 14(2): 100-155.
http://www.iaees.org/publications/journals/nb/articles/2024-14(2)/5-Zhang-Abstract.asp

Zhang WJ. 2024c. Structure comparison and evenness test of biological communities: Several platform-independent computational tools. Computational Ecology and Software, 14(2): 119-136.
http://www.iaees.org/publications/journals/ces/articles/2024-14(2)/CommAnaly.htm

Zhang WJ. 2025a. A web-based data generator for Mendelian Randomization (MR) analysis. Network Pharmacology, 10(3-4): 14-65.
http://www.iaees.org/publications/journals/np/articles/2025-10(3-4)/1-Zhang-Abstract.asp

Zhang WJ. 2025b. Mendelian randomization: Principles and methods. Network Biology, 15(2): 24-47.
http://www.iaees.org/publications/journals/nb/articles/2025-15(2)/1-Zhang-Abstract.asp

Zhang WJ. 2026a. A GWAS data fetcher with AI for Mendelian Randomization analysis. Network Pharmacology, 11(3-4): 62-89.
http://www.iaees.org/publications/journals/np/articles/2026-11(3-4)/1-Zhang-Abstract.asp

Zhang WJ. 2026b. A multi-source GWAS data fetcher for Mendelian Randomization analysis. Network Pharmacology, 11(1-2): 1-61.
http://www.iaees.org/publications/journals/np/articles/2026-11(1-2)/1-Zhang-Abstract.asp

Zhang WJ. 2026c. probFunCal: A webpage calculator for probability distribution functions. Selforganizology, 13(1-2): 1-25
http://www.iaees.org/publications/journals/selforganizology/articles/2026-13(1-2)/1-Zhang-Abstract.asp

Zhang WJ. 2026d. MR-GWAS-JSON: A web tool for multi-source Mendelian Randomization (MR) data fetching, harmonization, and JSON generation. Network Biology, 16(4): 403-486.
http://www.iaees.org/publications/journals/nb/articles/2026-16(4)/1-Zhang-Abstract.asp

Zhang WJ, Liu GH. 2024. Dynamically insert the forest plot into a web page: The full Javascript codes.

Computational Ecology and Software, 14(3): 168-173.
http://www.iaees.org/publications/journals/ces/articles/2024-14(3)/1-Zhang-Abstract.asp

Zhang WJ, Qi YH. 2024. ANOVA-nSTAT: ANOVA methodology and computational tools in the paradigm of new statistics. Computational Ecology and Software, 14(1): 48-67.
http://www.iaees.org/publications/journals/ces/articles/2024-14(1)/4-Zhang-Abstract.asp

Zhang WJ, Qi YH. 2026. Fetching GWAS (Genome-Wide Association Study) data via AI: A web tool to synthesize genotype, phenotype, and summary statistics. Ornamental and Medicinal Plants, 9(1-4): 1-21.
http://www.iaees.org/publications/journals/omp/articles/2026-9(1-4)/1-Zhang-Abstract.asp

Zheng J, Baird D, Borges MC, et al. 2017. Recent developments in Mendelian Randomization studies. Current Epidemiology Reports, 4(4): 330-345. doi: 10.1007/s40471-017-0128-6

Zhu S Xiangjie Kong XJ, Han FL, et al. 2024. Association between social isolation and depression: Evidence from longitudinal and Mendelian randomization analyses. Journal of Affective Disorders, 350: 182-187. doi: 10.1016/j.jad.2024.01.106