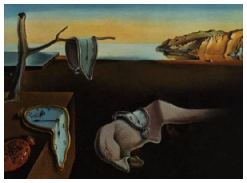


Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *doi: 10.18637/jss.v000.i00*

Making, Updating, and Querying Causal Models with `CausalQueries`

Till Tietz
Humboldt University

Lily Medina
University of California, Berkeley

Georgiy Syunyaev
Vanderbilt University

Macartan Humphreys
WZB

Abstract

The R package `CausalQueries` can be used to make, update, and query causal models defined on binary nodes. Users provide a causal statement of the form `X -> M <- Y; M <-> Y` which is interpreted as a structural causal model over a collection of binary nodes. Then `CausalQueries` allows users to (1) identify the set of principal strata—causal types—required to characterize all possible causal relations between nodes that are consistent with the causal statement (2) determine a set of parameters needed to characterize distributions over these causal types (3) update beliefs over distributions of causal types, using a `stan` model plus data, and (4) pose a wide range of causal queries of the model, using either the prior distribution, the posterior distribution, or a user-specified candidate vector of parameters.

Keywords: causal model, Bayesian updating, DAG, Stan.

1. Introduction: Causal models

`CausalQueries` is an R package that lets users make, update, and query causal models. Users provide a structural causal model in the form of a statement that reports a set of binary variables and the relations of causal ancestry between them. There are three primary functions. The first, `make_model()`, takes a causal statement and generates a parameter vector that fully describes a probability distribution over all possible types of causal relations between variables ("causal types"). Given a prior distribution over parameters—equivalently, over causal models consistent with the structural model— and data on some or all nodes, the second primary function, `update_model()`, deploys a Stan (Carpenter, Gelman, Hoffman, Lee,

Goodrich, Betancourt, Brubaker, Guo, Li, and Riddell 2017) model to generate a posterior distribution over causal models. The third primary function `query_model()` can then be used to ask a wide range of causal queries, using either the prior distribution, the posterior distribution, or a user-specified candidate vector of parameters.

In the next section we provide a motivating example that uses the three primary functions together. We then describe how the package relates to existing available software. Section 4 gives an overview of the statistical model behind the package. Section 6, Section 7, and Section 8 then describe, in turn, the functionality for making, updating, and querying causal models. We provide further computation details in the final section.

2. Motivating example

Before providing details on package functionality, we give an example of an application of the three primary functions, showing how to use `CausalQueries` to replicate the analysis in (Chickering and Pearl 1996; see also Humphreys and Jacobs 2023).

Chickering and Pearl (1996) seek to draw inferences on causal effects in the presence of imperfect compliance. We have access to an instrument Z (a randomly assigned prescription for cholesterol medication), which is a cause of X (treatment uptake) but otherwise unrelated to Y (cholesterol). We imagine we are interested in three specific queries. The first is the average causal effect of X on Y. The second is the average effect for units for which $X = 0$ and $Y = 0$; this "probability of causation" query asks whether untreated individuals with bad outcomes did poorly *because* they were untreated. The last is the average effect for "compliers": units for which X responds positively to Z. Thus two of these queries are conditional queries, with one conditional on a counterfactual quantity.

The data on Z, X, and Y is given in Chickering and Pearl (1996) and is also included in the `CausalQueries` package. The data looks as follows:

```
R> data("lipids_data")
R>
R> lipids_data
```

```
#>    event strategy count
#> 1 Z0X0Y0      ZXY   158
#> 2 Z1X0Y0      ZXY    52
#> 3 Z0X1Y0      ZXY     0
#> 4 Z1X1Y0      ZXY    23
#> 5 Z0X0Y1      ZXY    14
#> 6 Z1X0Y1      ZXY    12
#> 7 Z0X1Y1      ZXY     0
#> 8 Z1X1Y1      ZXY    78
```

This data is reported in "compact form," meaning it records the number of units ("count") that display each possible pattern of outcomes on Z, X, and Y ("event").

The strategy to analyse this data involves three steps:

- Step 1: generate a model with `make_model`, yielding an object of class `causal_model`

- Step 2: update the model with `update_model`, again yielding an object of class `causal_model`
- Step 3: pose queries of the model with `query_model`, yielding an object of class `model_query`

Users generate the stipulated causal model using `CausalQueries` as follows:

```
R> lipids_model <-  make_model("Z -> X -> Y; X <-> Y")
```

The result is an object of class `causal_model`, with associated `print`, `summary`, and `plot` methods. By default, uniform priors are placed on model parameters.

Users can then *update* beliefs over model parameters by supplying data as follows:

```
R> lipids_model <- update_model(lipids_model, lipids_data)
```

The updated model is also an object of class `causal_model`, though now also containing a posterior distribution over model parameters.

Finally, users can *query* the model. For instance, the three previously mentioned queries, which vary in the type of conditioning imposed, can be formulated as follows:

```
R> lipids_queries <-
+  query_model(
+    lipids_model,
+    queries = list(
+      ATE  = "Y[X=1] - Y[X=0]",
+      PoC  = "Y[X=1] - Y[X=0] :|: X==0 & Y==0",
+      LATE = "Y[X=1] - Y[X=0] :|: X[Z=1] > X[Z=0]"),
+    using = "posteriors"
+  )
```

The output is an object of class `model_query`, with associated `print`, `summary`, and `plot` methods.

The `model_query` object is a data frame containing estimates, and, when available, prior or posterior standard deviations, and credibility intervals.

Table 1 presents the results from the analysis of the lipids data.[1] Rows 1 and 2 in the table replicate the results from Chickering and Pearl (1996), while row 3 provides inferences for the local (complier) average treatment effects ("LATE").

Table 1: Replication of Chickering and Pearl (1996).

label	query	given	mean	sd	cred.low	cred.high
ATE	Y[X=1] - Y[X=0]	-	0.55	0.10	0.37	0.73
PoC	Y[X=1] - Y[X=0]	X==0 & Y==0	0.64	0.15	0.37	0.89
LATE	Y[X=1] - Y[X=0]	X[Z=1] > X[Z=0]	0.70	0.05	0.59	0.80

[1]Note that the "using" column is omitted to simplify output, as all estimands are derived from the posterior distribution.

For visual presentation of the results, output can also be plotted:

```
R> lipids_queries |> plot()
```

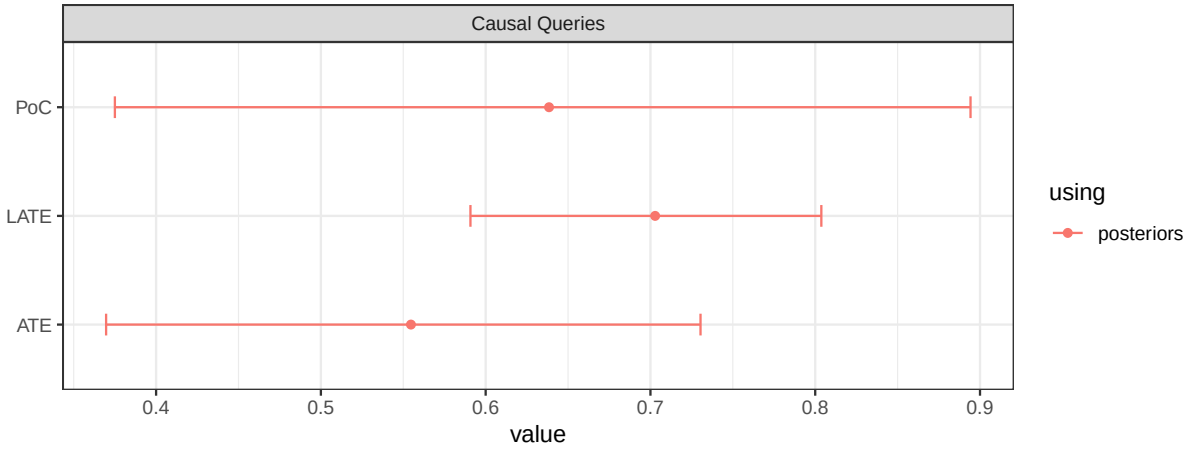


Figure 1: Illustration of queries plotted

These core functions can be combined in a single pipeline as follows:

```
R> make_model("Z -> X -> Y; X <-> Y") |>
+  update_model(lipids_data) |>
+  query_model(
+    queries = list(
+      ATE  = "Y[X=1] - Y[X=0]",
+      PoC  = "Y[X=1] - Y[X=0] :|: X==0 & Y==0",
+      LATE = "Y[X=1] - Y[X=0] :|: X[Z=1] > X[Z=0]"),
+    using = "posteriors") |>
+  plot()
```

As we describe below, the same basic procedure of making, updating, and querying models, can be used (up to computational constraints) for arbitrary causal models, for different types of data structures, and for all causal queries that can be posed of the causal model.

3. Connections to existing packages

The field of causal inference encompasses a wide range of software tools used across various disciplines, including social sciences, natural sciences, computer science, and applied mathematics. This section focuses on the role and capabilities of `CausalQueries` within the specific area of evaluating causal queries on models represented as directed acyclic graphs (DAGs) or structural equation models (SEMs). Table 2 provides a summary of relevant software, highlighting their connections, strengths, and limitations in comparison to `CausalQueries`.

Table 2: Related software.

Software	Source	Language	Availability	Scope
`causalnex`	Beaumont, Horsburgh, Pilgerstorfer, Droth, Oentaryo, Ler, Nguyen, Ferreira, Patel, and Leong (2021)	Python	• pip	• causal structure learning • querying marginal distributions • discrete data
`pclag`	Kalisch, Mächler, Colombo, Maathuis, and Bühlmann (2012)	R	• CRAN • GitHub	• causal structure learning • ATEs under linear conditional expectations, no hidden selection
`DoWhy`	Sharma and Kiciman (2020)	Python	• pip	• identification • average and conditional causal effects • robustness checks
`autobounds`	Duarte, Finkelstein, Knox, Mummolo, and Shpitser (2023)	Python	• Docker • GitHub	• bounding causal effects • partial identification • DAG canonicalization • binary data
`causaloptim`	Sachs, Jonzon, Sjölander, and Gabriel (2023)	R	• CRAN • GitHub	• bounding causal effects • non-identified queries • binary data

`causalnex` is comprehensive software that offers functions for discovering and querying causal models. Like `CausalQueries`, it uses Bayesian methods and supports "`do` calculus" (Pearl 2009). It focuses on conditional probability distribution tables instead of principal strata (causal types). This limits the types of queries and expert information that can be incorporated. For example, knowing conditional probability distributions is not enough to make claims about (or provide priors with respect to) effect monotonicity, complier effects, or the "probability of causation" (Dawid, Musio, and Murtas 2017). However, it allows for efficient handling of simple queries with larger models.

Similar to `causalnex`, `pclag` emphasizes learning about causal structures and uses the resulting DAGs to recover average treatment effects (ATEs) across all learned Markov-equivalent classes from observed data that satisfy linearity of conditional expectations. This approach is also more restrictive than `CausalQueries` in terms of the queries it allows.

`DoWhy` is a feature-rich framework focusing on causal identification, effect estimation, and assumption validation. With a user-specified DAG, it uses do-calculus to find expressions

that identify desired causal effects through Back-door, Front-door, IV, and mediation criteria, and then uses standard estimators to estimate the desired effect. After estimation, `DoWhy` deploys a comprehensive refutation engine with a large set of robustness tests. While this approach efficiently handles varied data types on large causal models, not parameterizing the DAG itself limits the types of queries that can be posed.

The packages most similar to `CausalQueries` for model definition are `autobounds` and `causaloptim`. They deal with discrete causal models, and their definitions of principal strata (causal types) and causal relations on the DAG are very similar to those of `CausalQueries`. Differences arise in how disturbance terms and confounding are defined: implicitly by the causal statement in `CausalQueries` versus explicitly via separate disturbance nodes in `autobounds` and `causaloptim`. While `CausalQueries` assumes a canonical form for input DAGs, `autobounds` and `causaloptim` facilitate canonicalization. The main difference between the methods is in their approach to evaluating queries.

Both `autobounds` and `causaloptim` build on seminal approaches in Balke and Pearl (1997) to construct bounds on queries, using constrained polynomial and linear optimization, respectively. In contrast, `CausalQueries` uses Bayesian inference to generate a posterior over the causal model, which is then queried (consistent with Chickering and Pearl 1996; Zhang, Tian, and Bareinboim 2022). A key difference is the target of inference. The polynomial and linear programming approach is suited to handling larger causal models. However, due to their similar model parameterization, `autobounds`, `causaloptim`, and `CausalQueries` face similar constraints as parameter spaces expand rapidly with model size. The Bayesian approach to model updating and querying is more efficient because a model can be updated once and queried multiple times, while expensive optimization runs are needed for each separate query in `autobounds` and `causaloptim`.

In summary, the main strength of `CausalQueries` is its ability to let users define arbitrary DAG and pose any queries on it, using a canonical procedure to form Bayesian posteriors for those queries, regardless of whether they are identified. For instance, if researchers want to learn about the local average treatment effect and their model meets the conditions in Angrist, Imbens, and Rubin (1996), updating will recover valid estimates as data grows, even if researchers are unaware that the local average treatment effect is identified or unfamiliar with the specific estimation method proposed by Angrist *et al.* (1996).

There are two main limitations of the models that `CausalQueries` can handle. First, `CausalQueries` is designed for models with a relatively small number of binary nodes. Since it does not limit the range of possible causal relationships in a model, the parameter space expands quickly with the model's complexity. This complexity growth depends on the causal structure and increases rapidly with the number of parents influencing a child node. A chain model like $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E$ has only 18 parameters ($2^1 + 4 \times 2^2$), while a model in which A, B, C, D are all direct ancestors of E has $65,544$ parameters ($4 \times 2^1 + 2^{(4^2)}$). Switching from binary to nonbinary nodes has similar effects. The restriction to binary nodes is for computational, not conceptual, reasons.[2]

[2]For more on computational constraints and strategies for updating and querying large models, see the `CausalQueriesTools` package available via `devtools::install_github("till-tietz/CausalQueriesTools")`. The main approach here is to divide large causal models into modules, update each module, and then reassemble them to pose queries. Also, see section 9.4.1 of Humphreys and Jacobs (2023) for a method that represents non-binary data as a profile of outcomes on multiple binary nodes.

Second, the package is designed for learning about populations from independently sampled units. Therefore, it does not inherently address issues of clustering, hierarchical structures, or purposive sampling. However, the broader framework can be adapted for these purposes (see section 9.4 of Humphreys and Jacobs 2023). The targets of inference are case-level or population-level quantities; `CausalQueries` is not well-suited for estimating sample quantities.

4. Statistical model

The core conceptual framework used by `CausalQueries` is that described in Pearl's *Causality* (Pearl 2009). It can be summarized as follows (using notation from Humphreys and Jacobs 2023):

Definition 1 *A "**causal model**" is:*

1. *an ordered collection of "endogenous nodes"* $Y = \{Y_1, Y_2, \ldots, Y_n\}$
2. *an ordered collection of "exogenous nodes"* $\Theta = \{\theta^{Y_1}, \theta^{Y_2}, \ldots, \theta^{Y_n}\}$
3. *a collection of functions* $F = \{f_{Y_1}, f_{Y_2}, \ldots, f_{Y_n}\}$ *with* f_{Y_j} *specifying, for each* j*, how outcome* Y_j *depends on* θ_j *and realizations of endogenous nodes prior to* Y_j*.*
4. *a probability distribution,* λ*, over* Θ*.*

By default, `CausalQueries` assumes that endogenous nodes are binary. When defining a causal structure, we specify which endogenous nodes are (possible) direct causes of a node, Y_j, given the other nodes in the model. These nodes are referred to as the parents of Y_j, denoted as PA_j (where uppercase PA_j represents the collection of nodes, and lowercase pa_j represents a specific set of values these nodes might assume). With discrete-valued nodes, it is possible to identify all potential responses of a node to its parents, which we call "nodal types." If a node i can take on k_i possible values, then the set of possible values that the parents of j can assume is $m_j := \prod_{i \in PA_j} k_i$. Consequently, there are $k_j^{m_j}$ different ways that node j might respond to its parents. For binary nodes, this simplifies to $2^{\left(2^{|PA_j|}\right)}$. Thus, an endogenous node with no parents has 2 nodal types; a binary node with one binary parent has four types; and a binary node with two parents has 16 types, and so forth.

The complete set of possible causal reactions of a given unit to all possible values of its parents is represented by its collection of nodal types at each node. We refer to this collection as a unit's "causal type," denoted as θ. These causal types correspond to the principal strata, which are familiar from the study of instrumental variables (Frangakis and Rubin 2002).

The approach used by `CausalQueries` is to align the domain of exogenous nodes θ^{Y_j} with the number of nodal types for Y_j. The function f^j then determines the value of y by simply reporting the value of Y_j implied by the nodal type and the values of the parents of Y_j. Therefore, if $\theta^j_{pa_j}$ is the value for j when parents have values pa_j, then $f_{Y_j}(\theta^j, pa_j) = \theta^j_{pa_j}$. Practically, this means that, given the causal structure, learning about the model reduces to learning about the distribution, λ, over the nodal types.

In scenarios without unobserved confounding, we assume that the probability distributions over the nodal types for different nodes are independent: $\theta^i \perp\!\!\!\perp \theta^j, i \neq j$. In this case, we use a

categorical distribution to specify $\lambda^j_x := \Pr(\theta^j = \theta^j_x)$. From this independence, the probability of a given causal type θ_x is $\prod_{i=1}^n \lambda^i_x$. For example, $\Pr(\theta = (\theta^X_1, \theta^Y_{01})) = \Pr(\theta^X = \theta^X_1)\Pr(\theta^Y = \theta^Y_{01}) = \lambda^X_1\lambda^Y_{01}$. In cases where confounding is present, the logic remains the same, but we need to specify enough parameters to capture the joint distribution over nodal types for different nodes.

For instance, in the Lipids model, the joint distribution of nodal types can be simplified as shown in Equation 1.

$$\Pr(\theta^Z = \theta^Z_1, \theta^X = \theta^X_{10}, \theta^Y = \theta^Y_{11}) = \Pr(\theta^Z = \theta^Z_1)\Pr(\theta^X = \theta^X_{10})\Pr(\theta^Y = \theta^Y_{11}|\theta^X = \theta^X_{10}) \quad (1)$$

And so, for this model, λ would include parameters that represent $\Pr(\theta^Z)$ and $\Pr(\theta^X)$ but also the conditional probability $\Pr(\theta^Y|\theta^X)$:

$$\Pr(\theta^Z = \theta^Z_1, \theta^X = \theta^X_{10}, \theta^Y = \theta^Y_{11}) = \lambda^Z_1\lambda^X_{10}\lambda^{Y|\theta^X_{10}}_{11} \quad (2)$$

Representing beliefs *over causal models* thus requires specifying a probability distribution over λ. This distribution might be degenerate if users wish to specify a particular model. `CausalQueries` also allows users to specify parameters, α, of a Dirichlet distribution over λ^j for each node Y^j (and similarly for conditional distributions in cases of confounding). If all entries of α are 0.5, this corresponds to Jeffreys priors. By default, `CausalQueries` assumes a uniform distribution, meaning all nodal types are equally likely, which corresponds to α being a vector of 1's.[3]

Updating is then done with respect to beliefs over λ. In the Bayesian approach we have:

$$p(\lambda|D) = \frac{p(D|\lambda)p(\lambda)}{\int_{\lambda'} p(D|\lambda')p(\lambda')}$$

where $p(D|\lambda')$ is calculated under the assumption that units are exchangeable and independently drawn. In practice this means that the probability that two units have causal types θ_i and θ_j is simply $\lambda'_i\lambda'_j$. Since a causal type fully determines an outcome vector $d = \{y_1, y_2, \ldots, y_n\}$, the probability of a given outcome ("event"), w_d, is given simply by the probability that the causal type is among those that yield outcome d. Thus, from λ we can calculate a vector of event probabilities, $w(\lambda)$, for each vector of outcomes, and under independence, we have:

$$D \sim \text{Multinomial}(w(\lambda), N)$$

Thus for instance in the case of an $X \to Y$ model, and letting w_{xy} denote the probability of a data type $X = x, Y = y$, the event probabilities are:

[3]While flexible, using the Dirichlet distribution does constrain the types of priors that can be represented; see Irons and Cinelli (2023) for a discussion of these constraints and an approach to incorporating richer priors using multiple Beta distributions.

$$w(\lambda) = \begin{cases} w_{00} & = & \lambda_0^X(\lambda_{00}^Y + \lambda_{01}^Y) \\ w_{01} & = & \lambda_0^X(\lambda_{11}^Y + \lambda_{10}^Y) \\ w_{10} & = & \lambda_1^X(\lambda_{00}^Y + \lambda_{10}^Y) \\ w_{11} & = & \lambda_1^X(\lambda_{11}^Y + \lambda_{01}^Y) \end{cases}$$

For a more complex example, Table 3 illustrates key values for the Lipids model. We see here that we have two types for the exogenous node Z, four for X (representing the strata familiar from instrumental variables analysis: never takers, always takers, defiers, and compliers) and four for Y. For Z and X we have parameters corresponding to probability of these nodal types. For instance `Z.0` is the probability that $Z = 0$. `Z.1` is the complementary probability that $Z = 1$. Things are a little more complicated for distributions on nodal types for Y however: because of confounding between X and Y we have parameters that capture the conditional probability of the nodal types for Y *given* the nodal types for X. We see there are four sets of these parameters. The next to final column shows a sample set of parameter values. Together, the parameters describe a full joint probability distribution over types for Z, X and Y that is faithful to the graph.

Table 3: Nodal types and parameters for Lipids model.

node	nodal_type	param_set	param_names	param_value	priors
Z	0	Z	Z.0	0.84	1
Z	1	Z	Z.1	0.16	1
X	00	X	X.00	0.12	1
X	10	X	X.10	0.08	1
X	01	X	X.01	0.11	1
X	11	X	X.11	0.69	1
Y	00	Y.X.00	Y.00_X.00	0.38	1
Y	10	Y.X.00	Y.10_X.00	0.00	1
Y	01	Y.X.00	Y.01_X.00	0.20	1
Y	11	Y.X.00	Y.11_X.00	0.42	1
Y	00	Y.X.01	Y.00_X.01	0.40	1
Y	10	Y.X.01	Y.10_X.01	0.38	1
Y	01	Y.X.01	Y.01_X.01	0.06	1
Y	11	Y.X.01	Y.11_X.01	0.16	1
Y	00	Y.X.10	Y.00_X.10	0.71	1
Y	10	Y.X.10	Y.10_X.10	0.15	1
Y	01	Y.X.10	Y.01_X.10	0.05	1
Y	11	Y.X.10	Y.11_X.10	0.09	1
Y	00	Y.X.11	Y.00_X.11	0.65	1
Y	10	Y.X.11	Y.10_X.11	0.04	1
Y	01	Y.X.11	Y.01_X.11	0.01	1
Y	11	Y.X.11	Y.11_X.11	0.30	1

These parameters again imply a probability distribution over data types. For instance the

probability of data type $Z = 0, X = 0, Y = 0$ is:

$$w_{000} = \Pr(Z = 0, X = 0, Y = 0) = \lambda_0^Z \lambda_{00}^X (\lambda_{00}^{Y|\lambda_{00}^X} + \lambda_{01}^{Y|\lambda_{00}^X}) + \lambda_0^Z \lambda_{01}^X (\lambda_{00}^{Y|\lambda_{01}^X} + \lambda_{01}^{Y|\lambda_{01}^X})$$

The value of the `CausalQueries` package is that it enables users to specify arbitrary models of this form, determine all the implied nodal and causal types, and update these models using given priors and data. This is achieved by calculating event probabilities based on all possible parameter vectors and subsequently the likelihood of the data given the model. Additionally, the package allows users to pose arbitrary queries on a model to evaluate the values of estimands of interest, which are functions of the values or counterfactual values of nodes, *conditional* on the values or counterfactual values of nodes.

The following sections review the classes and methods used by `CausalQueries` and the key functionalities for making, updating, and querying causal models.

5. Classes and Methods

`CausalQueries` makes use of two types of object classes, `causal_model` and `model_query`.

An object of class `causal_model` encodes a structural causal model and stores information on parameter values—either provided by the user or set to defaults—as well as prior or posterior distributions over model parameters. An object of class `causal_model` is generated using `make_model()` and can be adjusted using `update_model()` as well as a set of helper functions: `set_confound`, `set_restrictions`, `set_priors`, as describe in sections Section 6 and Section 7 below. Methods `print`, `summary` and `plot` are available for an object of class `causal_model`.

An object of class `model_query` records responses to queries posed of a causal model. Depending on the nature of the query, it can include estimates of effects, prior or posterior standard deviations, and confidence intervals. An object of class `causal_model` is generated using `query_model` as described in Section 8. Methods `print`, `summary` and `plot` are available for an object of class `model_query`.

6. Making models

A model can be defined in a single step in `CausalQueries` by supplying a causal statement—expressed using `dagitty`-style syntax (Textor, van der Zander, Gilthorpe, Liśkiewicz, and Ellison 2016)— to `make_model`. This generates an object of class `causal_model` (see Section 5).

To illustrate, a model where X causes both M and Y, and M also causes Y, can be created as follows:

```
R> model <- make_model("X -> M -> Y <- X")
```

The statement provides the names of nodes as well as arrows ("`->`" or "`<-`") connecting nodes and indicating whether one node is a potential cause of another, i.e., whether a given node is a "parent" or "child" of another. Formally, a statement like this is interpreted as:

1. Functional equations:
   - $Y = f_Y(M, X, \theta^Y)$
   - $M = f_M(X, \theta^M)$
   - $X = f_X(\theta^X)$
2. Distributions on Θ:
   - $\Pr(\theta^i = \theta^i_k) = \lambda^i_k$
3. Independence assumptions:
   - $\theta_i \perp\!\!\!\perp \theta_j, i \neq j$

In addition, as we did in the Chickering and Pearl (1996) example, it is possible to use two-headed arrows ("`<->`") to indicate "unobserved confounding," that is, the presence of an unobserved variable that might influence two or more observed variables. In this case, condition 3 above is relaxed, and the exogenous nodes associated with confounded variables have a joint distribution. We describe how this is done in greater detail in Section 6.3.1.

6.1. Graphing

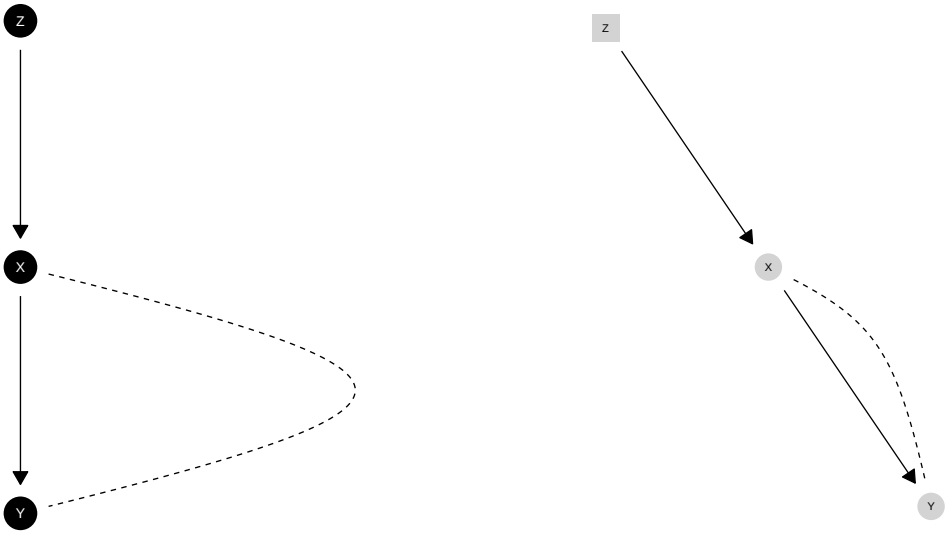
Plotting the model can help users verify that they have correctly defined its structure. `CausalQueries` offers straightforward graphing tools that utilize features from the `ggplot2` and `ggdag` packages. Once a model is defined, it can be graphed by calling the `plot()` method on objects of class `causal_model`. This method is a wrapper for the `plot_model()` function and accepts additional options, which are detailed in `?plot_model`.

Figure 2 shows figures generated by plotting `lipids_model` with and without options. The plots have class `c("gg", "ggplot")` and so will accept any additional layers available for the objects of class `ggplot`.

```
R> lipids_model |> plot()
R>
R> lipids_model |>
+  plot(x_coord = 1:3,
+       y_coord = 3:1,
+       textcol = "black",
+       textsize = 3,
+       shape = c(15, 16, 16),
+       nodecol = "lightgrey",
+       nodesize = 10)
```

6.2. Model inspection

When a model is defined, `CausalQueries` generates a set of internal objects used for all inferential tasks. These include default parameter values, default priors, and matrices that map parameters to causal types and causal types to data types. Although users generally do not need to examine these objects, `CausalQueries` provides two functions, `inspect()` and



(a) Without options

(b) With options

Figure 2: Examples of model graphs. For help on options see `?plot_model`

`grab()`, that allow users to quickly review these elements. The only difference between the two is that `grab()` is quiet and does not produce a printout, whereas `inspect()` does.

Table 4 summarizes features of a causal model that can be examined using `inspect()`.

Table 4: Elements of a model that can be inspected using `inspect()`.

Element	Description
`statement`	A character string describing causal relations using dagitty syntax.
`nodes`	A list containing the nodes in the model.
`parents_df`	A table listing nodes, whether they are root nodes or not, and the number and names of parents they have.
`parameters`	A vector of 'true' parameters.
`parameter_names`	A vector of names of parameters.
`parameter_mapping`	A matrix mapping from parameters into data types.
`parameter_matrix`	A matrix mapping from parameters into causal types.
`parameters_df`	A data frame containing parameter information.
`causal_types`	A data frame listing causal types and the nodal types that produce them.
`nodal_types`	A list with the nodal types of the model.
`data_types`	A list with all data types consistent with the model.
`ambiguities_matrix`	A matrix mapping from causal types into data types.
`prior_hyperparameters`	A vector of alpha values used to parameterize Dirichlet prior distributions; optionally provide node names to reduce output.
`prior_distribution`	A data frame of the parameter prior distribution.
`posterior_distribution`	A data frame of the parameter posterior distribution.

Element	Description
`type_prior`	A matrix of type probabilities using priors.
`type_posterior`	A matrix of type probabilities using posteriors.
`prior_event_probabilities`	A vector of data (event) probabilities given a single realization of parameters.
`posterior_event_probabilities`	A sample of data (event) probabilities from the posterior.
`data`	A data frame with data that was provided to update the model.
`stan_summary`	A `stanfit` summary with processed parameter names.
`stanfit`	An unprocessed `stanfit` object as generated by Stan.
`stan_warnings`	A list of warnings produced by Stan during updating.

6.3. Tailoring models

When a causal statement is provided to `make_model()`, the model is created with a set of default assumptions: specifically, there are no restrictions on nodal types, and flat priors are assumed over all parameters. These features can be modified after the model is created using `set_confounds`, `set_restrictions`, `set_priors`, and `set_parameters`.

Allowing confounding

Unobserved confounding between two (or more) nodes arises when the nodal types for the nodes are not independent. For instance, in the $X \rightarrow Y$ graph, there are 2 nodal types for X and 4 for Y. There are thus 8 joint nodal types (or causal types), as shown in Table 5.

Table 5: Nodal types in $X \rightarrow Y$ model.

	θ_0^X	θ_1^X	$\sum$
θ_{00}^Y	$\Pr(\theta_0^X, \theta_{00}^Y)$	$\Pr(\theta_1^X, \theta_{00}^Y)$	$\Pr(\theta_{00}^Y)$
θ_{10}^Y	$\Pr(\theta_0^X, \theta_{10}^Y)$	$\Pr(\theta_1^X, \theta_{10}^Y)$	$\Pr(\theta_{10}^Y)$
θ_{01}^Y	$\Pr(\theta_0^X, \theta_{01}^Y)$	$\Pr(\theta_1^X, \theta_{01}^Y)$	$\Pr(\theta_{01}^Y)$
θ_{11}^Y	$\Pr(\theta_0^X, \theta_{11}^Y)$	$\Pr(\theta_1^X, \theta_{11}^Y)$	$\Pr(\theta_{11}^Y)$
$\sum$	$\Pr(\theta_0^X)$	$\Pr(\theta_1^X)$	1

Table 5 has eight interior elements so that an unconstrained joint distribution would have seven degrees of freedom. A no-confounding assumption means that $\Pr(\theta^X, \theta^Y) = \Pr(\theta^X)\Pr(\theta^Y)$. In this case, it is sufficient to put a distribution on the marginals, and there would be 3 degrees of freedom for Y and 1 for X, totaling 4 rather than 7. To allow for an unconstrained joint distribution, the parameters data frame for this model would include two parameter families associated with the node Y. Each family represents the conditional distribution of Y's nodal types, given X. For example, the parameter `Y01_X.1` can be

interpreted as $\Pr(\theta^Y = \theta^Y_{01} | \theta^X = 1)$. Refer to Table 3 for an example of a parameter matrix with confounding.

The confounding structure can influence the number of parameters based on the underlying DAG. Table 6 demonstrates the number of independent parameters required for different types of confounding.

Table 6: Number of different independent parameters (degrees of freedom) for different three-node models.

Model	Degrees of freedom
`X -> Y <- W`	17
`X -> Y <- W; X <-> W`	18
`X -> Y <- W; X <-> Y; W <-> Y`	62
`X -> Y <- W; X <-> Y; W <-> Y; X <-> W`	63
`X -> W -> Y <- X`	19
`X -> W -> Y <- X; W <-> Y`	64
`X -> W -> Y <- X; X <-> W; W <-> Y`	67
`X -> W -> Y <- X; X <-> W; W <-> Y; X <-> Y`	127

Setting restrictions

It is often beneficial to constrain the set of types. In `CausalQueries`, this is achieved at the nodal type level, with restrictions on causal types following those on nodal types. For example, in analyses of data with imperfect compliance, such as in our Lipids model example, it is common to impose a monotonicity assumption: that X does not decrease in response to Z. This assumption is necessary to interpret instrumental variable estimates as consistent estimates of the complier average treatment effect. In `CausalQueries`, we can impose this assumption by removing types for which X decreases in Z as follows:

```
R> model_restricted <-
+  lipids_model |>
+  set_restrictions("X[Z=1] < X[Z=0]")
```

If we wanted to retain only this nodal type rather than remove it, we could do so by passing `keep = TRUE` as an argument to the `set_restrictions()` function call. Users can use `inspect(model, "parameter_matrix")` to view the resulting parameter matrix in which both the set of parameters and the set of causal types are restricted.

Restrictions in `CausalQueries` can be set in several other ways described below.

- Using nodal type labels:

  ```
  R> model <-
  +  lipids_model |>
  +  set_restrictions(labels = list(X = "01", Y = c("00", "01", "11")),
  +                   keep = TRUE)
  ```

- Using wildcards in nodal type labels:

```
R> model <- lipids_model |>
+  set_restrictions(labels = list(Y = "?0"))
```

- In models with confounding, restrictions can be added to nodal types conditional on the values of other nodal types using a `given` argument:

```
R> model <- lipids_model |>
+  set_restrictions(labels = list(Y = c('00', '11')), given = 'X.00')
```

Setting restrictions sometimes involves using causal syntax (see Section 8.2 for a guide to the syntax used by `CausalQueries`). The help file in `?set_restrictions` provides further details and examples of restrictions users can set.

Setting Priors

Priors on model parameters can be added to the parameters data frame and interpreted as alpha parameters of a Dirichlet distribution. The Dirichlet distribution is a probability distribution over an $n-1$ dimensional unit simplex. It is a generalization of the Beta distribution and is parameterized by an n-dimensional positive vector α. For example, a Dirichlet distribution with $\alpha = (1, 1, 1, 1, 1)$ provides a probability distribution over all non-negative 5-dimensional vectors that sum to 1, such as $(0.1, 0.1, 0.1, 0.1, 0.6)$ or $(0.1, 0.2, 0.3, 0.3, 0.1)$. This specific value for α implies that all such vectors are equally likely. Different values for α can be used to adjust the expectation and certainty for each dimension. For instance, the vector $\alpha = (100, 1, 1, 1, 100)$ would place more weight on distributions that are close to $(0.5, 0, 0, 0, 0.5)$.

In `CausalQueries`, priors are generally specified over the distribution of nodal types.[4] For example, in a model represented by $X \rightarrow Y$, there is one Dirichlet distribution over the two types for θ^X and another Dirichlet distribution over the four types for θ^Y. Importantly, it is implicitly assumed that priors are independent across families. Thus, in a model represented by $X \rightarrow Y$, we specify beliefs over λ^X and λ^Y separately. `CausalQueries` does not allow users to specify correlated beliefs over these parameters.[5]

Prior hyperparameters are set to unity by default, corresponding to uniform priors. Users can retrieve the model's priors as follows:

```
R> lipids_model |>
+  inspect("prior_hyperparameters", nodes = "X")
```

```
#>
#> prior_hyperparameters
#> Alpha parameter values used for Dirichlet prior distributions:
#>
#> X.00 X.10 X.01 X.11
#>    1    1    1    1
```

Alternatively users can set Jeffreys priors using `set_priors()` as follows:

[4] If there is confounding in the model, priors are specified over the conditional distribution of nodal types.

[5] Users can specify beliefs about λ^Y given θ^X if a model involves possible confounding. However, this refers to beliefs over a joint distribution, not jointly distributed beliefs.

```
R> model <- lipids_model |>
+  set_priors(distribution = "jeffreys")
```

Users can also provide custom priors. The simplest way to specify custom priors is to add them as a vector of numbers using `set_priors()`. For instance:

```
R> lipids_model |>
+  set_priors(param_names = c("X.10", "X.01"), alphas = 3:4) |>
+  inspect("prior_hyperparameters", nodes = "X")
```

```
#>
#> prior_hyperparameters
#> Alpha parameter values used for Dirichlet prior distributions:
#>
#> X.00 X.10 X.01 X.11
#>    1    3    4    1
```

The priors here should be interpreted as indicating $\alpha_X = (1, 3, 4, 1)$, which implies a distribution over $(\lambda^X_{00}, \lambda^X_{10}, \lambda^X_{01}, \lambda^X_{11})$ with expectation $\left(\frac{1}{9}, \frac{3}{9}, \frac{4}{9}, \frac{1}{9}\right)$.

Providing priors as a vector of numbers for larger models can be hard. For that reason, `set_priors()` allows for more targeted modifications of the parameter vector. For instance:

```
R> lipids_model |>
+  set_priors(statement = "X[Z=1] > X[Z=0]", alphas = 3) |>
+  inspect("prior_hyperparameters", nodes = "X")
```

```
#>
#> prior_hyperparameters
#> Alpha parameter values used for Dirichlet prior distributions:
#>
#> X.00 X.10 X.01 X.11
#>    1    1    3    1
```

While setting highly targeted priors is convenient and flexible, it should be done with caution. Assigning priors to specific parameters in complex models, particularly those involving confounding, can significantly impact inferences. Additionally, note that flat priors over nodal types do not necessarily equate to flat priors over queries. Flat priors over parameters within a parameter family assign equal weight to each nodal type, which can lead to strong assumptions about causal quantities of interest. For example, in an $X \rightarrow Y$ model where negative effects are excluded, the average causal effect implied by flat priors is 1/3. This can be demonstrated by querying the model as follows:

```
R> query <-
+  make_model("X -> Y") |>
+  set_restrictions(decreasing("X", "Y")) |>
+  query_model("Y[X=1] - Y[X=0]", using = "priors")
```

More subtly, the *structure* of a model, coupled with flat priors, has substantive importance for priors on causal quantities. For instance, with flat priors, prior on the probability that X has a positive effect on Y in the model $X \rightarrow Y$ is centered on 1/4. But prior on the probability that X positively affects Y in the model $X \rightarrow M \rightarrow Y$ is centered on 1/8. Caution regarding

priors is essential when models are not identified, as is the case for many models considered here. For some quantities, the marginal posterior distribution reflects the marginal prior distribution (Poirier 1998).

Setting Parameters

By default, models include a vector of parameter values within the `parameters_df` data frame. These values are useful for generating data or for scenarios like process tracing, where inferences about causal types (θ) are made from case-level data, assuming the model is known. The process of setting parameters is similar to setting priors. The key difference is that while the α value assigned to nodal types can be any positive number—reflecting our confidence in the parameter value—the parameter values themselves must be within the unit interval, $[0, 1]$. If parameter values provided are outside this interval, they are normalized to fit within it.

The causal model below has two parameter sets, one for X and one for Y, with two nodal types for X and four for Y. The key feature of the parameters is that they must sum to 1 within each parameter set.

```
R> make_model("X -> Y") |>
+  inspect("parameters")

#>
#> parameters
#> Model parameters with associated probabilities:
#>
#>  X.0  X.1 Y.00 Y.10 Y.01 Y.11
#> 0.50 0.50 0.25 0.25 0.25 0.25
```

The example below illustrates a change in the value of the parameter that corresponds to a positive effect of X on Y. Here, the nodal type `Y.Y01` is set to be 0.7, while the other nodal types of this parameter set were re-normalized so that the parameters in the set still sum up to one.

```
R> make_model("X -> Y") |>
+  set_parameters(statement = "Y[X=1] > Y[X=0]", parameters = .7) |>
+  inspect("parameters")

#>
#> parameters
#> Model parameters with associated probabilities:
#>
#>  X.0  X.1 Y.00 Y.10 Y.01 Y.11
#>  0.5  0.5  0.1  0.1  0.7  0.1
```

6.4. Drawing and manipulating data

Once a model has been defined, it is possible to simulate data from the model using the `make_data()` function. For instance, this can be useful for assessing a model's expected performance given data drawn from some speculated set of parameter values.

Drawing data basics

Generating data requires a specification of parameter values. The parameter values in the parameters data frame are used by default. Otherwise users can provide parameters on the fly.

```
R> lipids_model |>
+  make_data(n = 4)
```

```
#>   Z X Y
#> 1 0 0 0
#> 2 0 0 1
#> 3 1 1 0
#> 4 1 1 1
```

The resulting data is ordered by data type, as shown in the example above. Users can also specify parameters directly or draw parameters from a prior or posterior distribution by specifying `param_type` argument in the `make_data()` call.

Drawing incomplete data

`CausalQueries` can be used when researchers have gathered different amounts of data for different nodes. For example, a researcher might gather data on X and Y for all units, but only have data on M for some units. The `make_data()` function enables users to simulate such data by specifying a data strategy that outlines the probabilities of observing data for different nodes, potentially based on previously observed nodes.

```
R> sample_data <-
+  lipids_model |>
+  make_data(n = 8,
+            nodes = list(c("Z", "Y"), "X"),
+            probs = list(1, .5),
+            subsets = list(TRUE, "Z==1 & Y==0"))
```

Reshaping data

Data produced by `make_data()` typically comes in a "long" format, where each row represents a single observation. However, for model updating, the data should be in a "compact" format that summarizes the number of units for each data type, organized by data "strategy," which indicates the nodes for which data was collected. The `CausalQueries` package provides function `collapse_data()` that allow users to convert data to compact format.

```
R> sample_data |>
+  collapse_data(lipids_model)
```

```
#>     event strategy count
#> 1  Z0X0Y0      ZXY     0
#> 2  Z1X0Y0      ZXY     1
#> 3  Z0X1Y0      ZXY     0
#> 4  Z1X1Y0      ZXY     1
```

```
#> 5   Z0X0Y1      ZXY     0
#> 6   Z1X0Y1      ZXY     0
#> 7   Z0X1Y1      ZXY     0
#> 8   Z1X1Y1      ZXY     0
#> 9     Z0Y0       ZY     1
#> 10    Z1Y0       ZY     2
#> 11    Z0Y1       ZY     1
#> 12    Z1Y1       ZY     2
```

In the same way, it is possible to move from compact to long format using `expand_data()`.[6]

7. Updating models

The approach used by the `CausalQueries` package to update parameter values given observed data relies on the Stan programming language (Carpenter *et al.* 2017). Below we explain the data required by the generic Stan program implemented in the package, the structure of that program, and then show how to use the package to produce posterior draws of parameters.

7.1. Data for Stan

We use a generic Stan program that works for all binary causal models. The main advantage of the generic program is that it allows us to pass the details of the causal model as data inputs to Stan instead of generating individual Stan programs for each causal model. Appendix B provides the complete Stan model code.

The data required by the Stan program includes vectors of observed data and priors on parameters, as well as a set of matrices needed for the mapping between events, data types, causal types, and parameters. In addition, data passed to `stan` includes counts of all relevant quantities as well as start and end positions of parameters pertaining to specific nodes and distinct data strategies. The internal function `prep_stan_data()` takes the model and data as arguments and produces a list with all objects that are required by the generic Stan program. Package users do not need to call the `prep_stan_data()` function directly.

7.2. How the Stan program works

The Stan model involves the following elements: (1) a specification of priors over sets of parameters, (2) a mapping from parameters to event probabilities, and (3) a likelihood function. Below, we describe each of those elements in more detail.

Probability distributions over parameter sets

The causal structure provided by a DAG simplifies the problem of generating a probability distribution over all parameters by focusing on distributions over sets of parameters. In the absence of unobserved confounding, these sets correspond to the nodal types for each node, resulting in a probability distribution over these nodal types. For example, in the $X \rightarrow Y$

[6]Note that `NA`'s are interpreted as data not being sought.

model, there are two parameter sets. The X nodal types are represented by a 2-dimensional Dirichlet distribution, $(\lambda_0^X, \lambda_1^X) \sim \text{Dirichlet}(\alpha_0^X, \alpha_1^X)$, and the Y nodal types are represented by a 4-dimensional Dirichlet distribution, $(\lambda_{00}^Y, \lambda_{10}^Y, \lambda_{01}^Y, \lambda_{11}^Y) \sim \text{Dirichlet}(\alpha_{00}^Y, \alpha_{10}^Y, \alpha_{01}^Y, \alpha_{11}^Y)$.

In cases involving confounding, these parameter sets are defined for a given node, conditional on the values of other nodes.

Event probabilities

We calculate the probability of data types for any parameter vector λ. This is done using a matrix that maps from parameters into data types.

In cases without confounding, there is a column for each data type; the matrix indicates which nodes in each set "contribute" to the data type, and the probability of the data type is found by summing within sets and taking the product over sets. To illustrate, we can examine the parameter mapping matrix for a simple model using the `inspect()` function as follows:

```
R> make_model("X -> Y") |>
+  inspect("parameter_mapping")

#>
#> parameter_mapping (Parameter mapping matrix)
#>
#>   Maps from parameters to data types, with
#>   possibly multiple columns for each data type
#>   in cases with confounding.
#>
#>       X0Y0 X1Y0 X0Y1 X1Y1
#> X.0      1    0    1    0
#> X.1      0    1    0    1
#> Y.00     1    1    0    0
#> Y.10     0    1    1    0
#> Y.01     1    0    0    1
#> Y.11     0    0    1    1
```

The probability of each data type can be determined using the parameter mapping matrix by combining a parameter vector with the corresponding column of the matrix. For instance, in the model above, the probability of the data type `X0Y0`, denoted as w_{00}, is calculated as $\lambda_0^X \times (\lambda_{00}^Y + \lambda_{01}^Y)$. This represents the product of the probability of `X.0` and the sum of the probabilities for `Y.00` and `Y.01`.

In cases with confounding, the approach is similar, except that the parameter mapping matrix can contain multiple columns for each data type to capture non-independence between nodes.

In the case of incomplete data, we first identify the set of data strategies, where a collection of a data strategy might be of the form "gather data on X and M, but not Y, for n_1 cases and gather data on X and Y, but not M, for n_2 cases." Within a data strategy, the probability of an observed event is given by summing the probabilities of the types that could give rise to a particular pattern of incomplete data.

Data probability

Once we have the event probabilities in hand for each data strategy, we are ready to calculate the probability of the data. For a given data strategy, this is given by a multinomial distribution with these event probabilities. When there is incomplete data, and so there are multiple data strategies, the probability of the data is given by the product of the multinomial probabilities for data generated by each strategy.

7.3. Implementation

The `update_model()` function is used to update a model by appending a posterior distribution over the model parameters. This function utilizes `rstan::sampling()` to draw from the posterior distribution, and users can pass any additional arguments that `rstan::sampling()` accepts. Since model updating can be slow for complex models, Appendix A demonstrates how users can employ parallelization to enhance computation speed. Appendix C offers an overview of model updating benchmarks, assessing the impact of model complexity and data size on updating times.

Users have the option to provide a `data` argument when calling `update_model()`. This argument should be a data frame that includes some or all of the nodes in the model. It is optional; if no data is provided, the Stan model will still run, and the resulting posterior distribution added to the model will be interpreted as draws from the prior distribution.

7.4. Incomplete and censored data

`CausalQueries` assumes that missing data is missing at random, conditional on observed data. For instance, in an $X \rightarrow M \rightarrow Y$ model, a researcher might have chosen to collect data on M in a random set of cases in which $X = 1$ and $Y = 1$. If there are positive relations at each stage, one may be more likely to observe M in cases in which $M = 1$. However, the observation of M is still random and conditional on the observed X and Y data. The Stan model in `CausalQueries` takes account of this kind of sampling by assessing the probability of observing a particular pattern of data within each data strategy.[7]

Additionally, you can specify when data has been censored, allowing the Stan model to account for it. For example, consider a scenario where we only observe X when $X = 1$ and not when $X = 0$. This type of sampling is non-random and depends on observable variables. You can address this by informing Stan that the probability of observing a particular data type is 0, regardless of parameter values. This is achieved using the `censored_types` argument in `update_model()`.

To illustrate, in the example below, we observe perfectly correlated data for X and Y. If we are aware that data in which $X \neq Y$ has been censored, then when we update, we do not move towards a belief that X causes Y.

```
R> data <- data.frame(X = rep(0:1, 5), Y = rep(0:1, 5))
R>
R> list(
```

[7]For further discussion, see Section 9.2.3.2 in Humphreys and Jacobs (2023).

```
+  uncensored =
+    update_model(make_model("X -> Y"),
+                 data),
+  censored =
+    update_model(make_model("X -> Y"),
+                 data,
+                 censored_types = c("X1Y0",  "X0Y1"))
+  ) |>
+  query_model("Y[X=1] - Y[X=0]", using = "posteriors")
```

```
#>
#> Causal queries generated by query_model (all at population level)
#>
#> |label           |model      |using      |  mean|    sd| cred.low| cred.high|
#> |:---------------|:----------|:----------|-----:|-----:|--------:|---------:|
#> |Y[X=1] - Y[X=0] |uncensored |posteriors | 0.590| 0.194|    0.168|     0.897|
#> |Y[X=1] - Y[X=0] |censored   |posteriors | 0.018| 0.315|   -0.614|     0.629|
```

7.5. Output

The main output of the `update_model()` function is a model that includes a posterior distribution of the model parameters, stored as a data frame within the model list. You can access this posterior distribution directly using the `grab()` function (or use the `inspect()` function for a more detailed output) as shown below:

```
R> model <-
+  make_model("X -> Y")  |>
+  update_model()
R>
R> posterior <- inspect(model, "posterior_distribution")
```

```
#>
#> posterior_distribution
#> Summary statistics of model parameters posterior distributions:
#>
#>   Distributions matrix dimensions are
#>   4000 rows (draws) by 6 cols (parameters)
#>
#>      mean   sd
#> X.0  0.51 0.28
#> X.1  0.49 0.28
#> Y.00 0.25 0.19
#> Y.10 0.25 0.19
#> Y.01 0.25 0.19
#> Y.11 0.25 0.20
```

Additionally, a distribution of causal types is stored by default. Optionally, the `stanfit` object and a distribution over event probabilities can also be saved as shown below:

```
R> lipids_model <-
+  lipids_model |>
+  update_model(keep_fit = TRUE,
+               keep_event_probabilities = TRUE)
```

The summary of the Stan model can be accessed using `inspect()` function and is saved in the updated model object by default. This provides two measures to help assess convergence.

```
R> make_model("X -> Y")  |>
+  update_model(keep_type_distribution = FALSE) |>
+  inspect("stan_summary")
```

```
#>
#> stan_summary
#> Stan model summary:
#>
#> Inference for Stan model: simplexes.
#> 4 chains, each with iter=2000; warmup=1000; thin=1;
#> post-warmup draws per chain=1000, total post-warmup draws=4000.
#>
#>             mean se_mean   sd   2.5%   25%   50%   75% 97.5% n_eff Rhat
#> X.0         0.50    0.01 0.29   0.03  0.25  0.49  0.74  0.97  3036    1
#> X.1         0.50    0.01 0.29   0.03  0.26  0.51  0.75  0.97  3036    1
#> Y.00        0.25    0.00 0.19   0.01  0.09  0.21  0.37  0.70  2031    1
#> Y.10        0.25    0.00 0.19   0.01  0.09  0.21  0.37  0.71  4633    1
#> Y.01        0.25    0.00 0.20   0.01  0.09  0.20  0.37  0.72  4162    1
#> Y.11        0.25    0.00 0.20   0.01  0.09  0.20  0.37  0.71  4701    1
#> lp__       -7.53    0.04 1.65 -11.75 -8.37 -7.15 -6.32 -5.44  1368    1
#>
#> Samples were drawn using NUTS(diag_e) at Mon Feb 10 17:53:42 2025.
#> For each parameter, n_eff is a crude measure of effective sample size,
#> and Rhat is the potential scale reduction factor on split chains (at
#> convergence, Rhat=1).
```

This summary provides information on the distribution of parameters and convergence diagnostics, summarized in the `Rhat` column. The last row shows the unnormalized log density on Stan's unconstrained space, which is intended to diagnose sampling efficiency and evaluate approximations.[8] This summary can also include summaries for the transformed parameters if users retain these.

7.6. Convergence problems and diagnostics

There is no guarantee that updating will produce reliable posterior draws. Indeed for some models, convergence failure are predictable. Fortunately, `stan` provides warnings that alert users to possible problems. These warnings are retained by `CausalQueries` and repeated in model summaries or when queries are posed.

[8]See Stan documentation for more details.

In the example below the missing data on M means that there is no information to assess whether the strong relation between X and Y is due to positive effects.

```
R> model <-
+  make_model("X -> M -> Y") |>
+  update_model(data = data.frame(X = rep(0:1, 10000), Y = rep(0:1, 10000)),
+               iter = 5000,
+               refresh = 0)
```

The print and summary methods returns warnings alerting users to the problem thus:

```
R> model
```

```
#>
#> Causal statement:
#> M -> Y; X -> M
#>
#> Number of nodal types by node:
#> X M Y
#> 2 4 4
#>
#> Number of causal types: 32
#>
#> Model has been updated and contains a posterior distribution with
#> 4 chains, each with iter=5000; warmup=2500; thin=1;
#> Use inspect(model, 'stan_summary') to inspect stan summary
#>
#> Warnings passed from rstan during updating:
#> The largest R-hat is 1.73, indicating chains have not mixed
#> Bulk Effective Samples Size (ESS) is too low
#> Tail Effective Samples Size (ESS) is too low
```

If users wish to run more advanced diagnostics of performance, they can retain and access the "raw" Stan output as follows:

```
R> model <-
+  make_model("X -> Y") |>
+  update_model(refresh = 0, keep_fit = TRUE)
```

Note that the raw output uses labels from the generic Stan model: `lambda` for the vector of parameters, corresponding to the parameters in the parameters data frame (`inspect(model, "parameters_df")`), a vector `types` for the causal types (`inspect(model, "causal_types")`) and `event_probabilities` for the event probabilities (`inspect(model, "event_probabilities")`).

```
R> model |>
+  inspect("stanfit")
```

```
#>
#> stanfit
#> Stan model summary:
```

```
#> Inference for Stan model: simplexes.
#> 4 chains, each with iter=2000; warmup=1000; thin=1;
#> post-warmup draws per chain=1000, total post-warmup draws=4000.
#>
#>             mean se_mean   sd   2.5%   25%   50%   75% 97.5% n_eff Rhat
#> lambdas[1]  0.50    0.01 0.29   0.03  0.25  0.50  0.75  0.97  2558    1
#> lambdas[2]  0.50    0.01 0.29   0.03  0.25  0.50  0.75  0.97  2558    1
#> lambdas[3]  0.25    0.00 0.20   0.01  0.09  0.20  0.37  0.71  2012    1
#> lambdas[4]  0.26    0.00 0.20   0.01  0.10  0.21  0.38  0.72  4663    1
#> lambdas[5]  0.25    0.00 0.20   0.01  0.08  0.20  0.37  0.71  4851    1
#> lambdas[6]  0.25    0.00 0.19   0.01  0.09  0.20  0.36  0.71  4266    1
#> types[1]    0.13    0.00 0.14   0.00  0.03  0.08  0.18  0.49  2317    1
#> types[2]    0.12    0.00 0.13   0.00  0.02  0.07  0.18  0.49  2150    1
#> types[3]    0.13    0.00 0.14   0.00  0.03  0.08  0.18  0.51  3350    1
#> types[4]    0.13    0.00 0.13   0.00  0.03  0.08  0.18  0.50  3460    1
#> types[5]    0.12    0.00 0.13   0.00  0.03  0.08  0.18  0.49  3676    1
#> types[6]    0.12    0.00 0.14   0.00  0.02  0.08  0.18  0.51  3694    1
#> types[7]    0.12    0.00 0.13   0.00  0.03  0.08  0.18  0.47  3424    1
#> types[8]    0.12    0.00 0.13   0.00  0.03  0.08  0.18  0.49  3166    1
#> lp__       -7.55    0.05 1.66 -11.87 -8.39 -7.15 -6.33 -5.45  1357    1
#>
#> Samples were drawn using NUTS(diag_e) at Mon Feb 10 17:53:59 2025.
#> For each parameter, n_eff is a crude measure of effective sample size,
#> and Rhat is the potential scale reduction factor on split chains (at
#> convergence, Rhat=1).
```

Users can then pass the `stanfit` object to other diagnostic packages such as `bayesplot`.

8. Queries

`CausalQueries` provides functionality to pose and answer elaborate causal queries. The key approach is to code causal queries as functions of causal types and return a distribution over the queries implied by the distribution over causal types. The primary approach is to use `query_model()` to generate an object of class `model_query` (see Section 5) which prints to a table or can be plotted directly. The next sections describe how such queries are calculated and the syntax used for posing queries.

8.1. Calculating factual and counterfactual quantities

An essential step in calculating most queries is assessing what outcomes will arise for causal types given different interventions on nodes. In practice, we map from causal types to data types by propagating realized values on nodes forward in the DAG, moving from exogenous or intervened upon nodes to their descendants in generational order. An internal function, `realise_outcomes()`, achieves this by traversing the DAG while recording the values implied by realizations on the node's parents for each node's nodal types.

To illustrate, consider the first causal type of a $X \rightarrow Y$ model:

1. θ_0^X implies that, absent intervention on X, X has a realized value of 0; θ_{00}^Y implies that, absent intervention on Y, Y has a realized value of 0 regardless of X.
2. We substitute for Y the value implied by the 00 nodal type given a 0 value on X, which in turn is 0.

The `realise_outcomes()` function, when called on this model, outputs the realized values for all causal types, with row names indicating the corresponding causal types.

```
R> make_model("X -> Y") |>
+  realise_outcomes()
```

```
#>      X Y
#> 0.00 0 0
#> 1.00 1 0
#> 0.10 0 1
#> 1.10 1 0
#> 0.01 0 0
#> 1.01 1 1
#> 0.11 0 1
#> 1.11 1 1
```

Intervening on X (see Pearl 2009) with $do(X = 1)$ yields:

```
R> make_model("X -> Y") |>
+  realise_outcomes(dos = list(X = 1))
```

```
#>      X Y
#> 0.00 1 0
#> 1.00 1 0
#> 0.10 1 0
#> 1.10 1 0
#> 0.01 1 1
#> 1.01 1 1
#> 0.11 1 1
#> 1.11 1 1
```

Similarly, `realise_outcomes()` can return the realized values on all nodes for each causal type given arbitrary interventions.

8.2. Causal syntax

`CausalQueries` provides syntax for the formulation of various causal queries including queries on all rungs of the "causal ladder" (Pearl 2009): prediction, such as the proportion of units where Y equals 1; intervention, such as the probability that $Y = 1$ when X is *set* to 1; counterfactuals, such as the probability that Y would be 1 were $X = 1$ given we know Y is 0 when X was observed to be 0. Queries can be posed at the population level or case level and can be unconditional (e.g., what is the effect of X on Y for all units) or conditional (for

example, the effect of X on Y for units for which Z affects X). This syntax enables users to write arbitrary causal queries to interrogate their models.

The core of querying involves determining which causal types correspond to specific queries. Users can use logical statements to inquire about observed conditions without interventions for factual queries. For instance, consider the query about the proportion of units where Y equals 1, expressed as `"Y == 1"`. Here, the logical operator `==` signifies that `CausalQueries` should consider units that meet the strict equality condition where Y equals 1.[9] When this query is executed, the `get_query_types()` function identifies all types that result in $Y = 1$ without any interventions.

```
R> make_model("X -> Y")  |>
+  get_query_types("Y==1")

#>
#> Causal types satisfying query's condition(s)
#>
#>  query =  Y==1
#>
#> X0.Y10  X1.Y01
#> X0.Y11  X1.Y11
#>
#>
#>  Number of causal types that meet condition(s) =  4
#>  Total number of causal types in model =  8
```

The key to forming causal queries is being able to ask about the values of variables, given that the values of some other variables are "controlled." This corresponds to the application of the *do* operator in Pearl (2009). In `CausalQueries`, this is done by putting square brackets, `[ ]`, around variables that are intervened upon.

For instance, consider the query `Y[X=0]==1`. This query asks about the types for which Y equals 1 when X is set to 0. Since X is set to zero, X is placed inside the brackets. Given that Y equals 1 is a condition about potentially observed values, it is expressed using the logical operator `==`. The set of causal types that meets this query is quite different:

```
R> make_model("X -> Y")  |>
+  get_query_types("Y[X=1]==1")

#>
#> Causal types satisfying query's condition(s)
#>
#>  query =  Y[X=1]==1
#>
#> X0.Y01  X1.Y01
#> X0.Y11  X1.Y11
#>
#>
```

[9] `CausalQueries` also accepts `=` as a shorthand for `==`, but `==` is preferred as it is the standard logical operator.

```
#>  Number of causal types that meet condition(s) =  4
#>  Total number of causal types in model =  8
```

When a node has multiple parents, it is possible to set the values of none, some, or all of the parents. For instance if $X1$ and $X2$ are parents of Y then `Y==1`, `Y[X1=1]==1`, and `Y[X1=1, X2=1]==1` queries cases for which $Y = 1$ when, respectively, neither parents values are controlled, when $X1$ is set to 1 but $X2$ is not controlled, and when both $X1$ and $X2$ are set to 1. For instance:

```
R> make_model("X1 -> Y <- X2")  |>
+  get_query_types("X1==1 & X2==1 & (Y[X1=1, X2=1] > Y[X1=0, X2=0])")

#>
#> Causal types satisfying query's condition(s)
#>
#>  query =  X1==1&X2==1&(Y[X1=1,X2=1]>Y[X1=0,X2=0])
#>
#> X11.X21.Y0001  X11.X21.Y0101
#> X11.X21.Y0011  X11.X21.Y0111
#>
#>
#>  Number of causal types that meet condition(s) =  4
#>  Total number of causal types in model =  64
```

In this case, the aim is to identify the types for which in fact $X1 = 1$ and $X2 = 1$ *in addition* $Y = 0$ when $X1 = X2 = 0$, and $Y = 1$ when $X1 = X2 = 1$.

Conditional queries

Many queries of interest are "conditional" queries. For example, the effect of X on Y for units for which $W = 1$ or the effect of X on Y for units for which Z positively affects X. Such conditional queries are posed in `CausalQueries` by providing a `given` statement and the `query` statement or by placing the condition following a `:|:` separator in the query expression. For instance `"Y[X=1]==1 :|: X==0"` asks for the probability that $Y = 1$ when X is set to 1 for a case in which in fact $X = 0$. The entire query then becomes: for what units does the `query` condition hold among those units for which the `given` condition holds? The two parts can each be calculated using `get_query_types`. Thus, for instance, in an $X \rightarrow Y$ model, the probability that X causes Y given $X = 1 \,\&\, Y = 1$ is the probability of causal `X1.Y11` type divided by the sum of the probabilities of types `X1.Y11` and `X1.Y01`. In practice, this is done automatically for users when they call `query_model()` or `query_distribution()`.

Complex expressions

Many queries involve complex statements across multiple sets of types, which can be constructed using relational operators. For instance, users can query whether X has a positive effect on Y by checking if Y is greater when X is set to 1 compared to when X is set to 0. This is expressed as `"Y[X=1] > Y[X=0]"`. The query "X has some effect on Y" is given by `"Y[X=1] != Y[X=0]"`.

Linear operators can also be used over a set of simple statements. Thus `"Y[X=1] - Y[X=0]"` returns the average treatment effect. In essence, rather than returning a `TRUE` or `FALSE` for the two parts of the query, the case memberships are forced to numeric values (1 or 0), and the differences are taken, which can be a 1, 0 or −1 depending on the causal type. Averaging provides the share of cases with positive effects, less the share of cases with negative effects.

```
R> make_model("X -> Y") |>
+  get_query_types("Y[X=1] - Y[X=0]")
```

```
#> X0.Y00 X1.Y00 X0.Y10 X1.Y10 X0.Y01 X1.Y01 X0.Y11 X1.Y11
#>      0      0     -1     -1      1      1      0      0
```

Nested queries

`CausalQueries` lets users pose nested "complex counterfactual" queries. Rather than stipulating a value to which some node is to be set, the user can set the value to the value that the node would take given actions taken on ancestor nodes. For instance `"Y[M=M[X=0], X=1]==1"` queries the types for which Y equals 1 when X is set to 1, while keeping M constant at whatever value it would take if X were set to 0.

8.3. Quantifying queries

To provide a *quantitative* answer to a query, it is necessary to assign probabilities to the causal types that correspond to the query.

Queries by hand

Queries can be calculated directly from the prior distribution or the posterior distribution provided by Stan. For instance, the following call plots the posterior distribution for the probability that Y is increasing in X for the $X \rightarrow Y$ model. The resulting plot is shown in Figure 3.

```
R> data  <- data.frame(X = rep(0:1, 50), Y = rep(0:1, 50))
R>
R> model <-
+  make_model("X -> Y") |>
+  update_model(data, iter  = 4000, refresh = 0)
R>
R> model |>
+  grab("posterior_distribution")  |>
+  ggplot(aes(Y.01 - Y.10)) + geom_histogram()
```

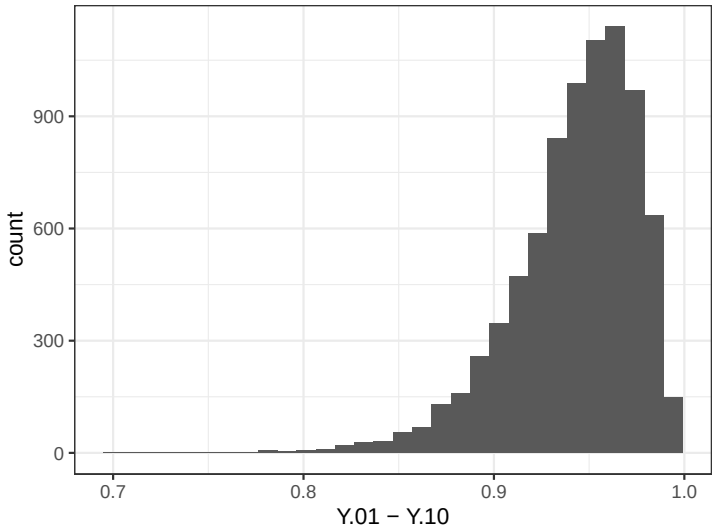


Figure 3: Posterior on "Probability Y is increasing in X".

Query distribution

It is generally helpful to use causal syntax to define the query and calculate the query with respect to the prior or posterior probability distributions. This can be done for a list of queries using `query_distribution()` function as follows:

```
R> queries <-
+  make_model("X -> Y") |>
+  query_distribution(
+    query = list(increasing = "(Y[X=1] > Y[X=0])",
+                 ATE = "(Y[X=1] - Y[X=0])"),
+    using = "priors")
```

The result is a data frame with one column per query and rows for draws from prior or posterior distributions as requested.

The core function `query_model` implements `query_distribution` and reports summaries of distributions.

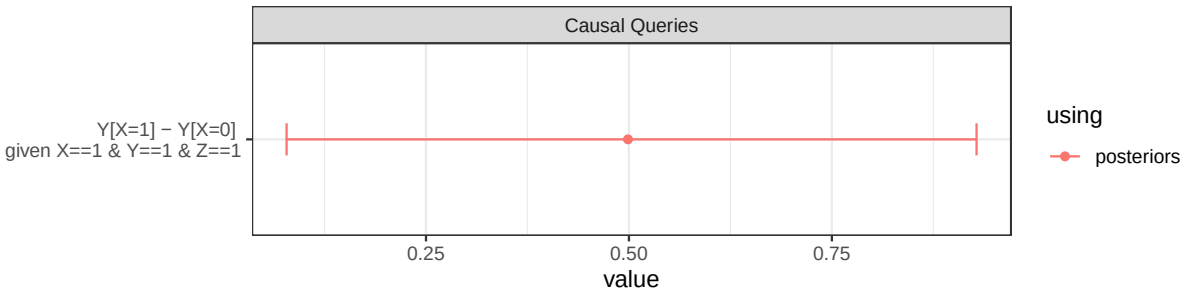
Case queries

The commands `query_distribution()` and `query_model()` can also be used when one is interested in assessing the value of a query about a new case that we might confront.

In a sense, this is equivalent to posing a conditional query, querying conditional on values in a case. For instance, we might consult our posterior for the Lipids model and ask about the effect of X on Y for a case in which $Z = 1$, $X = 1$ and $Y = 1$.

```
R> lipids_model |>
+  query_model(
+    query = "Y[X=1] - Y[X=0] :|: X==1 & Y==1 & Z==1",
```

```
+    using = "posteriors") |>
+  plot()
```

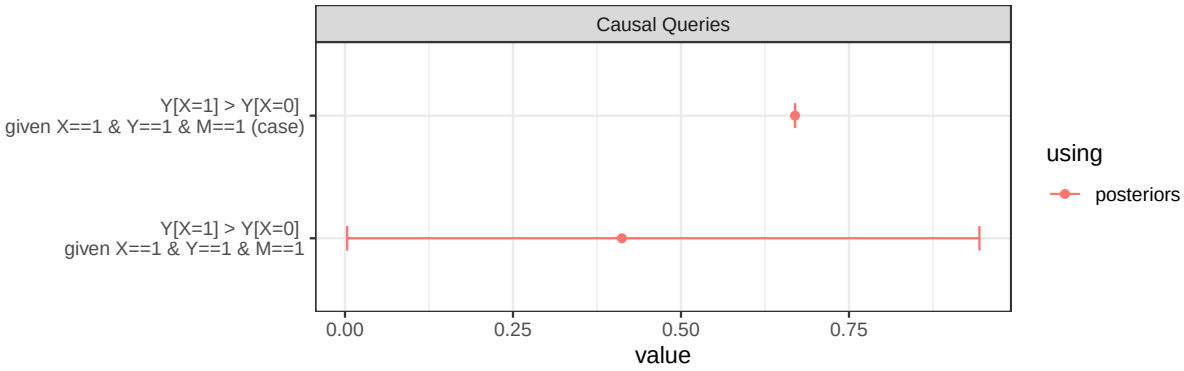


The result is what we should now believe for all cases in which $Z = 1$, $X = 1$, and $Y = 1$. It is the expected average effect among cases with this data type, so this expectation has an uncertainty attached to it, reflecting our uncertainty about the expectation.

This can differ, however, from what we would infer if we were presented with a new case drawn from the population. When examining a new case, we must *update* based on the information provided about that case. This *new* case-level inference is calculated when the `case_level = TRUE` argument is specified. For a query Q and given D, this returns the value $\frac{\int \pi(Q\&D|\lambda_i)p(\lambda_i)d\lambda_i}{\int \pi(D|\lambda_i)p(\lambda_i)d\lambda_i}$, which may differ from the mean of the distribution $\frac{\pi(Q\&D|\lambda)}{\pi(D|\lambda)}$ given by $\int \frac{\pi(Q\&D|\lambda_i)}{\pi(D|\lambda_i)}p(\lambda_i)d\lambda_i$.

To illustrate the difference, consider an $X \rightarrow M \rightarrow Y$ model where we are quite certain that X causes Y, but unsure whether this effect works through two positive or two negative effects. If asked what we would think about effects in cases $M = 0$ (or with $M = 1$), we have little basis to know whether these are cases in which effects are more or less likely. However, if we randomly find a case and we observe that $M = 0$, our understanding of the causal model evolves, leading us to believe there is (or is not) an effect in this specific case. The case-level query gives a single value without posterior standard deviation, representing the belief about this new case.

```
R> make_model("X -> M -> Y") |>
+  update_model(data.frame(X = rep(0:1, 8), Y = rep(0:1, 8)), iter = 4000) |>
+  query_model("Y[X=1] > Y[X=0] :|: X==1 & Y==1 & M==1",
+              using = "posteriors",
+              case_level = c(TRUE, FALSE)) |>
+  plot()
```



Batch queries

The function `query_model()` can also be used to pose multiple queries of multiple models in batch. The function takes a list of models, causal queries, and conditions as inputs. It then calculates population or case level estimands given prior or posterior distributions and reports summaries of these distributions. The result is a data frame of class `model_query` (see Section 5) that can be displayed as a table or used for graphing. The associated `plot` method produces plots with class `c("gg", "ggplot")`.

Since users can adjust multiple lists of features of a query there is an option, `expand_grid = TRUE`, to indicate whether to query with respect to all combinations of supplied arguments.

To illustrate, we return again to the limits model but now consider two versions, one with and one without a monotonicity restriction imposed.

```
R> models <- list(
+  Unrestricted = lipids_model |>
+    update_model(data = lipids_data, refresh = 0),
+
+  Restricted = lipids_model |>
+    set_restrictions("X[Z=1] < X[Z=0]") |>
+    update_model(data = lipids_data, refresh = 0)
+)
```

Table 7 then shows the output from a single call to `query_model()` with the `expand_grid` argument set to `TRUE` to generate all combinations of list elements.

```
R> queries <-
+  query_model(
+    models,
+    query = list(ATE = "Y[X=1] - Y[X=0]",
+                 POS = "Y[X=1] > Y[X=0] :|: Y==1 & X==1"),
+    case_level = c(FALSE, TRUE),
+    using = c("priors", "posteriors"),
+    expand_grid = TRUE)
```

Table 7: Results for two queries on two models.

label	model	query	given	using	case_level	mean	sd
ATE	Unrestricted	Y[X=1] - Y[X=0]	-	priors	FALSE	0.00	0.20
ATE	Restricted	Y[X=1] - Y[X=0]	-	priors	FALSE	0.00	0.23
ATE	Unrestricted	Y[X=1] - Y[X=0]	-	posteriors	FALSE	0.55	0.10
ATE	Restricted	Y[X=1] - Y[X=0]	-	posteriors	FALSE	0.56	0.10
POS	Unrestricted	Y[X=1] > Y[X=0]	Y==1 & X==1	priors	FALSE	0.50	0.22
POS	Restricted	Y[X=1] > Y[X=0]	Y==1 & X==1	priors	FALSE	0.49	0.24
POS	Unrestricted	Y[X=1] > Y[X=0]	Y==1 & X==1	posteriors	FALSE	0.95	0.04
POS	Restricted	Y[X=1] > Y[X=0]	Y==1 & X==1	posteriors	FALSE	0.95	0.04
ATE	Unrestricted	Y[X=1] - Y[X=0]	-	priors	TRUE	0.00	NA
ATE	Restricted	Y[X=1] - Y[X=0]	-	priors	TRUE	0.00	NA

ATE	Unrestricted	Y[X=1] - Y[X=0]	-	posteriors	TRUE	0.55	NA
ATE	Restricted	Y[X=1] - Y[X=0]	-	posteriors	TRUE	0.56	NA
POS	Unrestricted	Y[X=1] > Y[X=0]	Y==1 & X==1	priors	TRUE	0.50	NA
POS	Restricted	Y[X=1] > Y[X=0]	Y==1 & X==1	priors	TRUE	0.49	NA
POS	Unrestricted	Y[X=1] > Y[X=0]	Y==1 & X==1	posteriors	TRUE	0.95	NA
POS	Restricted	Y[X=1] > Y[X=0]	Y==1 & X==1	posteriors	TRUE	0.95	NA

Figure 4 shows the default plot associated with this query.

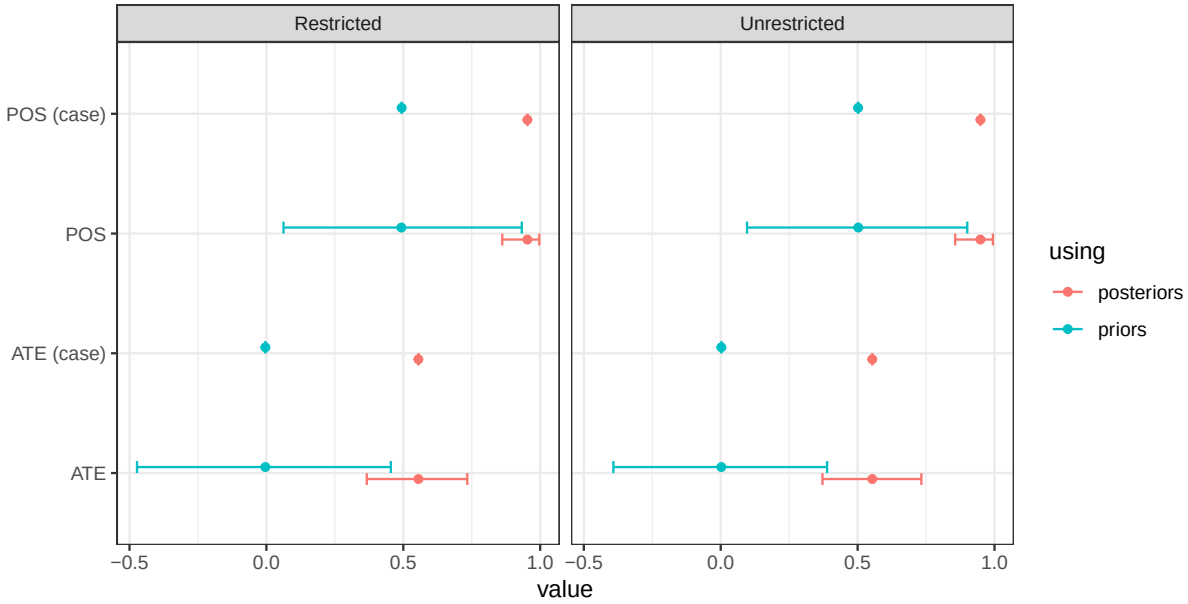


Figure 4: Default plotting for a a set of queries over multiple models.

9. Summary and discussion

`CausalQueries` provides an intuitive user interface to generate, update, and query causal models defined over binary nodes.

A particular strength is the flexibility users enjoy in specifying the structure of causal models and querying them using an integrated framework. Rather than requiring bespoke functions for different types of problems—studying treatment effects in randomized experiments, complier effects in encouragement designs, or mediation quantities in more complex causal structures—a unified procedure is used for defining models and for updating on model parameters. With updated models in hand, queries involving arbitrary *do* operations can be posed using an intuitive syntax.

We identify several areas for future expansion of the package's functionality. One area involves extending the class of models that can be passed to `make_model()` to accommodate non-binary data and hierarchical data structures. Proofs of concept for both extensions are available in Humphreys and Jacobs (2023). Inspired by new work in Irons and Cinelli (2023), we see

potential in allowing the specification of more flexible prior distributions and developing algorithms for faster updating in these settings. Inspired by conditions for identification of mediation quantities in Forastiere, Mattei, and Ding (2018), we see potential for allowing more flexible constraints over the joint distribution of nodal types. For querying, we see scope for facilitating nonlinear complex causal queries, such as risk ratios, which currently require combining multiple simple causal queries. Finally, we see potential for more integrated functionality for model validation, including assessments of the sensitivity of conclusions to priors.

Computational details and software requirements

Version	• 1.3.3
Availability	• Stable Release: https://cran.rstudio.com/web/packages/CausalQueries/index.html • Development: https://github.com/integrated-inferences/CausalQueries
Issues	• https://github.com/integrated-inferences/CausalQueries/issues
Operating Systems	• Linux • MacOS • Windows
Testing Environments OS	• Ubuntu 22.04.2 • Debian 12.2 • MacOS • Windows
Testing Environments R	• R 4.3.1 • R 4.3.0 • R 4.2.3 • r-devel
R Version	• R(>= 3.4.0)
Compiler	• either of the below or similar: • g++ • clang++
Stan requirements	• inline • RcppEigen (>= 0.3.3.3.0) • RcppArmadillo (>= 0.12.6.4.0) • RcppParallel (>= 5.1.4) • BH (>= 1.66.0) • StanHeaders (>= 2.26.0) • rstan (>= 2.26.0)
R-Packages Depends	• methods

R-Packages	• dirmult (>= 0.1.3-4) |
Imports	• dplyr |
	• stats (>= 4.1.1) |
	• rlang (>= 0.2.0) |
	• rstan (>= 2.26.0) |
	• rstantools (>= 2.0.0) |
	• stringr (>= 1.4.0) |
	• ggdag (>= 0.2.4) |
	• latex2exp (>= 0.9.4) |
	• ggplot2 (>= 3.3.5) |
	• lifecycle (>= 1.0.1) |
	• Rcpp (>= 0.12.0) |

The results in this paper were obtained using R~4.3 with the **MASS**~7.3-60 package. R itself and all packages used are available from the Comprehensive R Archive Network (CRAN) at https://CRAN.R-project.org/.

Acknowledgments

We thank Ben Goodrich, who provided generous insights on using `stan` for this project. We thank Alan M Jacobs for key work in developing the framework underlying the package. Our thanks to Cristian-Liviu Nicolescu, who provided wonderful feedback on the use of the package and a draft of this paper. Our thanks to Jasper Cooper for contributions to the generic function to create Stan code, to Clara Bicalho, who helped figure out the syntax for causal statements, to Julio S. Solís Arce who made many vital contributions figuring out how to simplify the specification of priors, and to Merlin Heidemanns who figured out the `rstantools` integration and made myriad code improvements.

References

Angrist JD, Imbens GW, Rubin DB (1996). "Identification of Causal Effects Using Instrumental Variables." *Journal of the American Statistical Association*, **91**(434), 444–455. doi:10.1080/01621459.1996.10476902.

Balke A, Pearl J (1997). "Bounds on Treatment Effects from Studies with Imperfect Compliance." *Journal of the American Statistical Association*, **92**(439), 1171–1176. doi:10.1080/01621459.1997.10474074.

Beaumont P, Horsburgh B, Pilgerstorfer P, Droth A, Oentaryo R, Ler S, Nguyen H, Ferreira GA, Patel Z, Leong W (2021). "**CausalNex**." URL https://github.com/quantumblacklabs/causalnex.

Carpenter B, Gelman A, Hoffman MD, Lee D, Goodrich B, Betancourt M, Brubaker MA, Guo J, Li P, Riddell A (2017). "**Stan**: A Probabilistic Programming Language." *Journal of Statistical Software*, **76**, 1. doi:10.18637/jss.v076.i01.

Chickering DM, Pearl J (1996). "A Clinician's Tool for Analyzing Non-Compliance." In *Proceedings of the National Conference on Artificial Intelligence*, pp. 1269–1276. URL https://cdn.aaai.org/AAAI/1996/AAAI96-188.pdf.

Dawid AP, Musio M, Murtas R (2017). "The probability of causation." *Law, Probability and Risk*, **16**(4), 163–179.

Duarte G, Finkelstein N, Knox D, Mummolo J, Shpitser I (2023). "An Automated Approach to Causal Inference in Discrete Settings." *Journal of the American Statistical Association*, pp. 1–16. doi:10.1080/01621459.2023.2216909.

Forastiere L, Mattei A, Ding P (2018). "Principal ignorability in mediation analysis: through and beyond sequential ignorability." *Biometrika*, **105**(4), 979–986.

Frangakis CE, Rubin DB (2002). "Principal Stratification in Causal Inference." *Biometrics*, **58**(1), 21–29. doi:10.1111/j.0006-341X.2002.00021.x.

Humphreys M, Jacobs AM (2023). *Integrated Inferences: Causal Models for Qualitative and Mixed-Method Research*. Cambridge University Press. doi:10.1017/9781316718636.

Irons NJ, Cinelli C (2023). "Causally Sound Priors for Binary Experiments." *arXiv preprint arXiv:2308.13713*.

Kalisch M, Mächler M, Colombo D, Maathuis MH, Bühlmann P (2012). "Causal Inference Using Graphical Models with the R Package **pcalg**." *Journal of Statistical Software*, **47**, 1–26. doi:10.18637/jss.v047.i11.

Pearl J (2009). *Causality*. Cambridge University Press. ISBN 978-0-521-89560-6.

Poirier DJ (1998). "Revising Beliefs in Nonidentified Models." *Econometric Theory*, **14**(4), 483–509. doi:10.1017/S0266466698144043.

Sachs MC, Jonzon G, Sjölander A, Gabriel EE (2023). "A General Method for Deriving Tight Symbolic Bounds on Causal Effects." *Journal of Computational and Graphical Statistics*, **32**(2), 567–576. doi:10.1080/10618600.2022.2071905.

Sharma A, Kiciman E (2020). "DoWhy: An End-to-End Library for Causal Inference." *arXiv preprint arXiv:2011.04216*.

Textor J, van der Zander B, Gilthorpe MS, Liśkiewicz M, Ellison GT (2016). "Robust Causal Inference Using Directed Acyclic Graphs: the R Package **dagitty**." *International Journal of Epidemiology*, **45**(6), 1887–1894. doi:10.1093/ije/dyw341.

Zhang J, Tian J, Bareinboim E (2022). "Partial Counterfactual Identification from Observational and Experimental Data." In *Proceedings of the 39th International Conference on Machine Learning*, pp. 26548–26558. PMLR. URL https://proceedings.mlr.press/v162/zhang22ab.html.

Appendix A: Parallelization

If users have access to multiple cores, parallel processing can be implemented by including this line before running `CausalQueries`:

```
R> library(parallel)
R>
R> options(mc.cores = parallel::detectCores())
```

Additionally, parallelizing across models or data while running MCMC chains in parallel can be achieved by setting up a nested parallel process. With 8 cores one can run two updating processes with three parallel chains each simultaneously. More generally the number of parallel processes at the upper level of the nested parallel structure are given by $\lfloor \frac{cores}{chains+1} \rfloor$.

```
R> library(future)
R> library(future.apply)
R>
R> chains <- 3
R> cores <- 8
R>
R> future::plan(list(
+      future::tweak(future::multisession,
+                    workers = floor(cores/(chains + 1))),
+      future::tweak(future::multisession,
+                    workers = chains)
+    ))
R>
R> model <- make_model("X -> Y")
R> data <- list(data_1 = data.frame(X=0:1, Y=0:1),
+             data_2 = data.frame(X=0:1, Y=1:0))
R>
R> results <-
+future.apply::future_lapply(
+  data,
+  function(d) {
+    update_model(
+      model = model,
+      data = d,
+      chains = chains,
+      refresh = 0
+    )},
+ future.seed = TRUE)
```

Appendix B: Stan code

Updating is performed using a generic Stan model. The data provided to Stan is generated by the internal function `prep_stan_data()`, which returns a list of objects that Stan expects to receive. The code for the Stan model is shown below. After defining a helper function, the code starts with a block declaring what input data is to be expected. Then, the parameters and the transformed parameters are characterized. Then, the likelihoods and priors are provided. At the end, a block for generated quantities is used to append a posterior distribution of causal types to the model.

```
S4 class stanmodel 'simplexes' coded as follows:
functions{
  row_vector col_sums(matrix X) {
    row_vector[cols(X)] s ;
    s = rep_row_vector(1, rows(X)) * X ;
    return s ;
  }
}
data {
int<lower=1> n_params;
int<lower=1> n_paths;
int<lower=1> n_types;
int<lower=1> n_param_sets;
int<lower=1> n_nodes;
array[n_param_sets] int<lower=1> n_param_each;
int<lower=1> n_data;
int<lower=1> n_events;
int<lower=1> n_strategies;
int<lower=0, upper=1> keep_type_distribution;
vector<lower=0>[n_params] lambdas_prior;
array[n_param_sets] int<lower=1> l_starts;
array[n_param_sets] int<lower=1> l_ends;
array[n_nodes] int<lower=1> node_starts;
array[n_nodes] int<lower=1> node_ends;
array[n_strategies] int<lower=1> strategy_starts;
array[n_strategies] int<lower=1> strategy_ends;
matrix[n_params, n_types] P;
matrix[n_params, n_paths] parmap;
matrix[n_paths, n_data] map;
matrix<lower=0,upper=1>[n_events,n_data] E;
array[n_events] int<lower=0> Y;
}
parameters {
vector<lower=0>[n_params - n_param_sets] gamma;
}
transformed parameters {
vector<lower=0, upper=1>[n_params] lambdas;
```

```
vector<lower=1>[n_param_sets] sum_gammas;
matrix[n_params, n_paths] parlam;
matrix[n_nodes, n_paths] parlam2;
vector<lower=0, upper=1>[n_paths] w_0;
vector<lower=0, upper=1>[n_data] w;
vector<lower=0, upper=1>[n_events] w_full;
// Cases in which a parameter set has only one value need special handling
// they have no gamma components and sum_gamma needs to be made manually
for (i in 1:n_param_sets) {
  if (l_starts[i] >= l_ends[i]) {
    sum_gammas[i] = 1;
    lambdas[l_starts[i]] = 1;
    }
  else if (l_starts[i] < l_ends[i]) {
    sum_gammas[i] =
    1 + sum(gamma[(l_starts[i] - (i-1)):(l_ends[i] - i)]);
    lambdas[l_starts[i]:l_ends[i]] =
    append_row(1, gamma[(l_starts[i] - (i-1)):(l_ends[i] - i)]) /
      sum_gammas[i];
    }
  }
// Mapping from parameters to data types
// (usual case): [n_par * n_data] * [n_par * n_data]
parlam  = rep_matrix(lambdas, n_paths) .* parmap;
// Sum probability over nodes on each path
for (i in 1:n_nodes) {
 parlam2[i,] = col_sums(parlam[(node_starts[i]):(node_ends[i]),]);
 }
// then take product  to get probability of data type on path
for (i in 1:n_paths) {
  w_0[i] = prod(parlam2[,i]);
 }
 // last (if confounding): map to n_data columns instead of n_paths
 w = map'*w_0;
  // Extend/reduce to cover all observed data types
 w_full = E * w;
}
model {
// Dirichlet distributions
for (i in 1:n_param_sets) {
  target += dirichlet_lpdf(lambdas[l_starts[i]:l_ends[i]]  |
    lambdas_prior[l_starts[i] :l_ends[i]]);
  target += -n_param_each[i] * log(sum_gammas[i]);
 }
// Multinomials
// Note with censoring event_probabilities might not sum to 1
for (i in 1:n_strategies) {
```

```
  target += multinomial_lpmf(
  Y[strategy_starts[i]:strategy_ends[i]] |
    w_full[strategy_starts[i]:strategy_ends[i]]/
     sum(w_full[strategy_starts[i]:strategy_ends[i]]));
 }
}
// Option to export distribution of causal types
generated quantities{
vector[n_types] types;
if (keep_type_distribution == 1){
for (i in 1:n_types) {
   types[i] = prod(P[, i].*lambdas + 1 - P[,i]);
}}
 if (keep_type_distribution == 0){
    types = rep_vector(1, n_types);
 }
}
```

Appendix C: Benchmarks

We present a brief summary of model updating benchmarks. Note that these benchmarks are not generally reproducible and depend on the specifications of the hardware system used to produce them. The first benchmark considers the effect of model complexity on updating time. The second benchmark considers the effect of data size on updating time. We run four parallel chains for each model. The results of the benchmarks are presented in Table 9 and Table 10.

Table 9: Benchmark 1.

Model	Number of Model Parameters	`update_model()` Run-Time (seconds)
$X1 \rightarrow Y$	6	7.6
$X1 \rightarrow Y; X2 \rightarrow Y$	20	9.6
$X1 \rightarrow Y; X2 \rightarrow Y; X3 \rightarrow Y$	262	96.4

Table 10: Benchmark 2.

Model	Number of Observations	`update_model()` Run-Time (seconds)
$X1 \rightarrow Y$	10	7.5
$X1 \rightarrow Y$	100	7.8
$X1 \rightarrow Y$	1000	9.8
$X1 \rightarrow Y$	10000	16.2

Increasing the number of parents in a model greatly increases the number of parameters and computational time. The results suggests the computational time is convex in the number of parents. Unless model restrictions are imposed, four parents would yield 65,536 parameters. The rapid growth of the parameter space with increasing model complexity places limits on feasible computability without further recourse to specialized methods for handling large causal models. In contrast, the results suggest that computational time is concave in the size of the data.

Affiliation:

Till Tietz
Humboldt University
Berlin Germany
E-mail: ttietz2014@gmail.com
URL: https://github.com/till-tietz

Lily Medina
University of California, Berkeley
E-mail: lily.medina@berkeley.edu
URL: https://lilymedina.github.io/

Georgiy Syunyaev
Vanderbilt University
E-mail: g.syunyaev@vanderbilt.edu
URL: https://gsyunyaev.com/

Macartan Humphreys
WZB, IPI
Reichpietschufer 50
Berlin Germany
E-mail: macartan.humphreys@wzb.eu
URL: https://macartan.github.io/

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
MMMMMM YYYY, Volume VV, Issue II *Submitted:* yyyy-mm-dd
doi:10.18637/jss.v000.i00 *Accepted:* yyyy-mm-dd