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

Till Tietz ⓘ
WZB

Lily Medina
UC Berkeley

Macartan Humphreys ⓘ
WZB

Georgiy Syunyaev ⓘ
Vanderbilt University

Abstract

A guide to the R package `CausalQueries` for making, updating, and querying causal models

Keywords: causal models, stan, bayes.

1. Introduction: Causal models

1.1. What CausalQueries does

`CausalQueries` is an R package that lets users make, update, and query causal models. The base specification asks users to provide a statement that reports a set of binary variables and the relations of causal ancestry between them: which variables are direct causes of other variables, given the other variables in the model. Once provided to `make_model()`, `CausalQueries` generates a set of parameters that fully describe all possible types of causal relations between variables ("causal types"), given the causal structure. Given a prior over distributions over causal types and data over some or all nodes, `update_model()` deploys a `stan` model in order to generate a posterior distribution over parameters. The function `query_model()` can then be used to ask any causal query of the model, using either the prior distribution, the posterior distribution, or a user specified candidate vector of parameters.

We illustrate these three core functions by showing how to use `CuasalQueries` to replicate

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

query	given	mean	sd	cred.low.2.5%	cred.high.97.5%
Y[X=1] - Y[X=0]	-	0.56	0.10	0.37	0.73
Y[X=1] - Y[X=0]	X==0 & Y==0	0.64	0.15	0.38	0.90
Y[X=1] - Y[X=0]	X[Z=1] > X[Z=0]	0.70	0.05	0.59	0.80

Rows 1 and 2 replicate results in Chickering and Pearl (1996); row 3 returns inferences for complier average effects.

the analysis in Chickering and Pearl (1996) (see also Humphreys and Jacobs (2023)). Chickering and Pearl (1996) seek to draw inference 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 $Y = 0$. The last is the average treatment 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.

Our data on Z, X, and Y is complete for all units and looks, in compact form, 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
```

With `CausalQueries`, you can create the model, input data to update it, and then query the model for results.

```
R> make_model("Z -> X -> Y; X <-> Y") |>
+   update_model(lipids_data, refresh = 0) |>
+   query_model(query = "Y[X=1] - Y[X=0]",
+               given = c("All",  "X==0 & Y==0", "X[Z=1] > X[Z=0]"),
+               using = "posteriors")
```

The output is a data frame with estimates, posterior standard deviations, and credibility intervals. For example the data frame produced by the code above is shown in Table 1.

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.

1.2. Connections to existing packages

- Embed the *software* into the respective relevant literature.
- compare with bareinboim package
- knox bounds package

2. Statistical model

MH: simplify ; clarify how we can get away with simpler structure; Embed the *methods*

The core conceptual framework is described in Pearl's *Causality* (Pearl 2009) but can be summarized as follows (using the notation used in Humphreys and Jacobs (2023)):

Definition 1 *A "causal model" includes:*

1. *an ordered collection of "endogenous nodes" $Y = \{Y_1, Y_2, \ldots, Y_n\}$*

2. *an ordered collection of "exogenous nodes" $\Theta = \{\theta^{Y_1}, \theta^{Y_1}, \ldots, \theta^{Y_n}\}$*

3. *a collection of functions $F = \{f_{Y_1}, f_{Y_2}, \ldots, f_{Y_n}\}$ specifying, for each node j, how outcome y_j depends on θ_j and realizations of endogenous nodes prior to j.*

4. *a probability distribution over Θ, λ*

In the usual case we take the endogenous nodes to be binary.[1] When we specify a causal structure we specify which endogenous nodes are (possibly) direct causes of a node, Y_j, given other nodes in the model. These nodes are called the parents of Y_j, PA_j (we use upper case PA_j to indicate the collection of nodes and lower case pa_j to indicate a particular set of values that these nodes might take on). In this case, with discrete valued nodes, it is possible to identify all possible ways that a node might relate to its parents. We call these "nodal types," and they correspond to principal strata familiar, for instance, in the study of instrumental variables (Frangakis and Rubin 2002). If node Y_i can take on k_i possible values then the set of possible values that can be taken on by parents of j is $m := \prod_{i \in PA_j} k_i$, then there are k_j^m nodal types — distinct ways that j might respond to its parents—with each types corresponding to a distinct way that j takes for a given constellation of values of its parents. In the case of binary nodes this becomes $2^{\left(2^{|PA_j|}\right)}$. Thus for an endogenous node with no parents there are 2 nodal types, for a binary node with one binary parent there are four types, for a binary node with 2 parents there are 16, and so on.

The set of all possible causal reactions of a given unit to all possible values of parents is then given by its collection of nodal types at each node. We call this collection a unit's "causal type", θ.

---

[1] `CausalQueries` can be used also to analyse non binary data though with a cost of greatly increased complexity. See section 9.4.1 of Humphreys and Jacobs (2023) for an approach that codes non binary data as a profile of outcomes on multiple binary nodes.

The approach used by `CausalQueries` is to let the domain of θ^{Y_j} be coextensive with the number of nodal types for Y_j. 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. Thus if $\theta^j_{pa_j}$ is the value for j when parents have values pa_j, then we have simply that $f_{y_j}(\theta^j, pa_j) = \theta^j_{pa_j}$. The practical implication is that, given the causal structure, learning about the model reduces to learning about the distribution λ over the nodal types.

In cases in which there is no unobserved confounding, we take the probability distributions over the nodal types for different nodes to be independent: $\theta^i \perp\!\!\!\perp \theta^j, i \neq j$. In this case we use a categorical distribution to specify the $\lambda^{j'} := \Pr(\theta^j = \theta^{j'})$. From independence then we have that the probability of a given causal type θ' is simply $\prod_{i=1}^n \lambda^{i'}$.

In cases in which there is confounding, the essential logic is the same except that we need to specify enough parameters to capture the joint distribution over nodal types for different nodes.

For concreteness: table illustrating all these values for a $X \rightarrow Y$ model

Representing beliefs over causal models thus requires specifying a probability distribution over λ. This might be a degenerate distribution if users want to specify a particular model. `CausalQueries` allows users to specify parameters, α of a Dirichlet distribution over λ. If all entries of α are 0.5 this corresponds to Jeffrey's priors. The default behavior is for `CausalQueries` to assume a uniform distribution – that is, that all nodal types are equally likely – which corresponds to α being a vector of 1s.

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

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

$p(D|\lambda')$ is calculated under the assumption that units are exchangeable and independently drawn, and of course, under the model. 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:

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

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

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

The value of the `CausalQueries` package is to allow users to specify models of this form, figure out all the implied causal types, and then update given priors and data by calculating event probabilities implied by all possible parameter vectors and in turn the likelihood of the data given the model. In addition, the package allows for arbitrary querying of a model to assess the values of estimands of interest that a re funciton of the values or counterfactual values of nodes conditional on values or counterfactual values of nodes.

TILL - Embed the *software* into the respective relevant literature. - For the latter both competing and complementary software should be discussed (within the same software environment and beyond), bringing out relative (dis)advantages. All software mentioned should be properly `@cited`'d. (See also Using BibTeX for more details on BibTeX.)

The particular strength of `CausalQueries` is to allow users to specify arbitrary DAGs, arbitrary queries over nodes in those DAGs, and use the same canonical procedure to learn about those queries whether or not the queries are identified. Thus is principle if researchers are interest in learning about a quantitiy like the local average treatment effect and their model in fact satisfies the conditions in Angrist, Imbens, and Rubin (1996), then updating will recover valid estimates even if researchers are unaware that the local average treatment effect is identified and are ignorant of the estimation procedure proposed by Angrist *et al.* (1996).

There are two broad limitation on the sets of models handled natively by `CausalQueries`. First `CausalQueries` is designed for models with a relatively small number over binary nodes. Because there is no compromise made on the space of possible causal relations implied by a given model, the parameter space grows very rapidly with the complexity of the causal model. The complexity also depends on the causal structure and grows rapidly with the number of parents affecting a given child. A chain model of the form $A \to B \to C \to D \to E$ has just 40 parameters. A model in which A, B, C, D are all direct of E has 65544 parameters. Moving from binary to non binary nodes has similar effects. The restriction to binary nodes is for computational and not conceptual reasons. In fact there are ways to employ `CausalQueries` to answer queries from models with nonbinary nodes but in general the computational costs make analysis of these model prohibitive.

Second the package is geared for learning about populations from samples of units that are independent of each other and are independently randomly sampled from populations. This the basic set up does not address problems of sampling, clustering, hierarchical structures, or purposive sampling, for example. In section X we provide pointers to how all of these can be addressed.

3. Making models

A model is defined in one step using a `dagitty` syntax in which the structure of the model is provided as a statement.

For instance:

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

The statement (in quotes) provides the names of nodes. An arrow ("`->`" or "`<-`") connecting

nodes indicates that one node is a potential cause of another (that is, whether a given node is a "parent" or "child" of another.

Formally a statement like this is interpreted as:

1. Functional equations:

- $Y = f(M, X, \theta^Y)$
- $M = f(X, \theta^M)$
- $X = \theta^X$

2. Distributions on Θ:

- $\Pr(\theta^i = \theta^i_k) = \lambda^i_k$

3. Independence assumptions:

- $\theta_i \perp\!\!\!\perp \theta_j, i \neq j$

where function f maps from the set of possible values of the parents of i to values of node i given θ^i as described above.

In addition, as we did in the Chickering and Pearl (1996) example, it is also possible to use two headed arrows (`<->`) to indicate "unobserved confounding", that is, the presence of an unobserved variable that might influence 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 **?**.

3.1. Graphing

Plotting the model can be useful to check that you have defined the structure of the model correctly. `CausalQueries` provides simple graphing tools that draw on functionality from the `dagitty`, `ggplot2`, and `ggdag` packages.

Once defined, a model can be graphed by calling the `plot()` method defined for the objects with class `causal_model` produced by `make_model()` function.

```
R> make_model("X -> M -> Y <- X; Z -> Y") |>
+  plot()
```

Alternatively you can provide a number of options to the `plot()` call that will be passed to `CausalQueries:::plot_dag` via the method.

```
R> make_model("X -> M -> Y <- X; Z -> Y") |>
+  plot(x_coord = 1:4,
+       y_coord = c(1.5,2,1,2),
+       textcol = "white",
+       textsize = 3,
+       shape = 18,
+       nodecol = "grey",
+       nodesize = 12)
```

The graphs produced by the two calls above are shown in Figure 1. In both cases the resulting plot will have class `c("gg", "ggplot")` and so will accept any additional modifications available via `ggplot2` package.
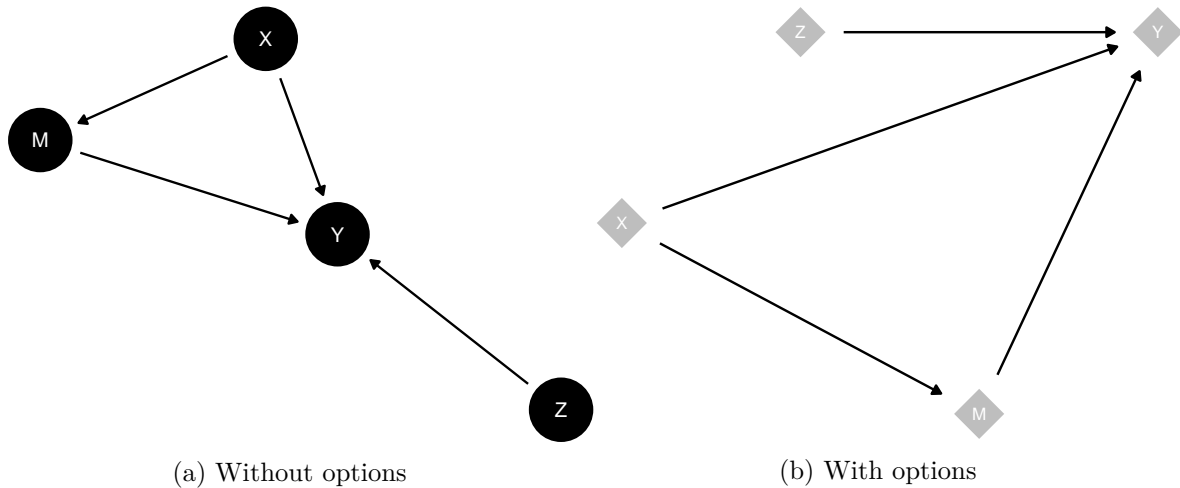

(a) Without options   (b) With options

Figure 1: Examples of model graphs.

3.2. Model characterization

When a model is defined, a set of objects is generated. These are the key quantities that are used for all inference. The table below summarizes the core components of a model, providing a brief explanation for each one.

The first element is a `statement` which defines how the nodes in the model are related, specified by the user using dagitty syntax. The second element, `dag`, is a data frame that outlines the parent-child relationships within the model. `Nodes` is simply a list of the node names in the model. Lastly, `parents_df,` is a table listing the nodes, indicating if they are root nodes, and showing how many parents each node has.

The model includes additional elements, `nodal_types`, `parameters_df,` and `causal_types,` which we explain later in detail.

Element	Description
statement	A character string that describes directed causal relations between variables in a causal model, where arrows denote that one node is a potential cause of another.
dag	A data frame with columns 'parent' and 'children' indicating how nodes relate to each other.
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 of parents they have.
nodal_types	A list with the nodal types in the model. (See Nodal Types Section)

Element	Description
parameters_df	A data frame linking the model's parameters with the nodal types of the model, as well as the family to which they belong. (See Parameters dataframe Section)
causal_types	A data frame listing causal types and the nodal types that produce them. (See Causal Types Section)
P:	Should I include it?
parameter matrix	

Table 2: Core Elements of a Causal Model

After updating a model, two additional components are attached to it:

- A posterior distribution of the parameters in the model, generated by stan. This distribution reflects the updated parameter values.

- A list of objects, which we refer to as `stan_objects`. The `stan_objects` will, at a minimum, include the distribution of nodal types and the data used for updating the model

Optionally, users can choose whether to keep and append additional elements to the `stan_objects`. For instance, they can specify whether to include `w`, which maps parameters to event probabilities, as well as the 'stanfit' object, the output generated by stan

The table below summarizes the objects attached to the model after updating.

Element	Description
posterior_distribution	The posterir distribution of the updated parameters generated by stan.
stan_objects	A list of objects, including, at a minimum, `type_distribution` and `data`.
data	The data used for updating the model, appended to `stan_objects`.
type_distribution	The updated distribution of the nodal types, appended to `stan_objects`.
w	A mapping from parameters to event probabilities, optionally appended to `stan_objects`.
stan_fit	The `stanfit` object generated by stan. This is optionally appended to `stan_objects`.

Parameters dataframe

When a model is created, `CausalQueries` attaches a "parameters dataframe" which keeps track of model parameters, which belong together in a family, and how they relate to causal types. This becomes especially important for more complex models with confounding that might involve more complicated mappings between parameters and nodal types. In the case

with no confounding the nodal types *are* the parameters; in cases with confounding you generally have more parameters than nodal types.

For instance:

```
R> make_model("X->Y")$parameters_df |>
+  knitr::kable()
```

param_names	node	gen	param_set	nodal_type	given	param_value	priors
X.0	X	1	X	0		0.50	1
X.1	X	1	X	1		0.50	1
Y.00	Y	2	Y	00		0.25	1
Y.10	Y	2	Y	10		0.25	1
Y.01	Y	2	Y	01		0.25	1
Y.11	Y	2	Y	11		0.25	1

Each row in the dataframe corresponds to a single parameter.

The columns of the parameters data frame are understood as follows:

- `param_names` gives the name of the parameter, in shorthand. For instance the parameter $\lambda_0^X = \Pr(\theta^X = \theta_0^X)$ has `par_name X.0`. See section @ref(notation) for a summary of notation.
- `param_value` gives the (possibly default) parameter values. These are probabilities.

- `param_set` indicates which parameters group together to form a simplex. The parameters in a set have parameter values that sum to 1. In this example $\lambda_0^X + \lambda_1^X = 1$.
- `node` indicates the node associated with the parameter. For parameter `\lambda^X_0` this is X.
- `nodal_type` indicates the nodal types associated with the parameter.
- `gen` indicates the place in the partial causal ordering (generation) of the node associated with the parameter
- `priors` gives (possibly default) Dirichlet priors arguments for parameters in a set. Values of 1 (.5) for all parameters in a set implies uniform (Jeffrey's) priors over this set.

Below we will see examples where the parameter dataframe helps keep track of parameters that are created when confounding is added to a model.

Nodal types

Two units have the same *nodal type* at node Y, θ^Y, if their outcome at Y responds in the same ways to parents of Y.

A node with k parents has 2^{2^k} nodal types. The reason is that with k parents, there are 2^k possible values of the parents and so 2^{2^k} ways to respond to these possible parental values. As a convention we say that a node with no parents has two nodal types (0 or 1).

When a model is created the full set of "nodal types" is identified. These are stored in the model. The subscripts become very long and hard to parse for more complex models so the model object also includes a guide to interpreting nodal type values. You can see them like this.

```
R> make_model("X -> Y")$nodal_types
```

```
$X
[1] "0" "1"
```

```
$Y
[1] "00" "10" "01" "11"
```

```
attr(,"interpret")
attr(,"interpret")$X
  node position display interpretation
1    X       NA      X0          X = 0
2    X       NA      X1          X = 1
```

```
attr(,"interpret")$Y
  node position display interpretation
1    Y        1   Y[*]*       Y | X = 0
2    Y        2   Y*[*]       Y | X = 1
```

Note that we use θ^j to indicate nodal types because for qualitative analysis the nodal types are often the parameters of interest.

Alternatively, the `interpret_type` function can be used to obtain interpretations for the nodal types of each node in the model. For instance, consider the model with two parents $X \rightarrow Y \leftarrow M$. In such a case, the nodal types of Y will have subscripts with four digits, with each digit representing one of the possible combinations of values that Y can take, given the values of its parents X and M. These combinations include the value of Y when:

- X = 0 and M = 0,
- X = 0 and M = 1,
- X = 1 and M = 0,
- X = 1 and M = 1.

As the number of parents increases, keeping track of what each digit represents becomes more difficult. For instance, if Y had three parents, its nodal types would have subscripts of eight digits, each associated with the value that Y would take for each combination of the three parents. The `interpret_type` function provides a clear map to identify what each digit in the subscript represents. See the examples below for models with two and three parents.

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

```
  node position display     interpretation
1    Y        1 Y[*]*** Y | M = 0 & X = 0
2    Y        2 Y*[*]** Y | M = 1 & X = 0
3    Y        3 Y**[*]* Y | M = 0 & X = 1
4    Y        4 Y***[*] Y | M = 1 & X = 1
```

```
R> model <- make_model("X -> Y <- M; W -> Y")
R> interpretations <- interpret_type(model)
```

```
R> interpretations$Y

  node position      display              interpretation
1    Y          1 Y[*]******* Y | M = 0 & W = 0 & X = 0
2    Y          2 Y*[*]****** Y | M = 1 & W = 0 & X = 0
3    Y          3 Y**[*]***** Y | M = 0 & W = 1 & X = 0
4    Y          4 Y***[*]**** Y | M = 1 & W = 1 & X = 0
5    Y          5 Y****[*]*** Y | M = 0 & W = 0 & X = 1
6    Y          6 Y*****[*]** Y | M = 1 & W = 0 & X = 1
7    Y          7 Y******[*]* Y | M = 0 & W = 1 & X = 1
8    Y          8 Y*******[*] Y | M = 1 & W = 1 & X = 1
```

LM: section on lookup types; `interpret_type` EXAMPLE with maybe 2 parents

Causal types

Causal types are collections of nodal types. Two units are of the same *causal type* if they have the same nodal type at every node. For example in a $X \to M \to Y$ model, $\theta = (\theta_0^X, \theta_{01}^M, \theta_{10}^Y)$ is a type that has $X = 0$, M responds positively to X, and Y responds positively to M.

When a model is created, the full set of causal types is identified. These are stored in the model object:

```
R> make_model("X -> Y")$causal_types

        X  Y
X0.Y00  0  00
X1.Y00  1  00
X0.Y10  0  10
X1.Y10  1  10
X0.Y01  0  01
X1.Y01  1  01
X0.Y11  0  11
X1.Y11  1  11
```

A model with n_j nodal types at node j has $\prod_j n_j$ causal types. Thus the set of causal types can be large. In the model $(X \to M \to Y \leftarrow X)$ there are $2 \times 4 \times 16 = 128$ causal types.

Knowledge of a causal type tells you what values a unit would take, on all nodes, absent an intervention. For example for a model $X \to M \to Y$ a type $\theta = (\theta_0^X, \theta_{01}^M, \theta_{10}^Y)$ would imply data $(X = 0, M = 0, Y = 1)$. (The converse of this, of course, is the key to updating: observation of data $(X = 0, M = 0, Y = 1)$ result in more weight placed on θ_0^X, θ_{01}^M, and θ_{10}^Y).)

Till: Show how realise_outcomes connects causal types with data types: Show for something with a do operation and without a do operation; point out causal types on left

In effect; we can 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. The `realise_outcomes` functions achieves this by traversing the DAG, while recording for each node's nodal types, the values implied by realizations on the node's parents. By way of example, consider the first causal type of the $X \to Y$ model above:

1. X is exogenous and has a realized value of 0
2. We substitute for Y the value implied by the 00 nodal type given a 0 value on X, which in turn is 0 (see nodal types)

Recovering implied values on complex nodal types given parent realizations efficiently at scale, exploits the fact that nodal types are the Cartesian Product of possible parent realizations. Finding the index of a Node's realized value given parent realizations in a nodal type is equivalent to finding the row index in the Cartesian Product matrix corresponding to those parent realizations. By definition of the Cartesian product, the number of consecutive 0 or 1 elements in a given column is $2^{columnindex}$, when indexing columns from 0. Given a set of parent realizations R indexed from 0, the corresponding row index in the Cartesian Product Matrix indexed from 0 can thus be computed via: $row = (\sum_{i=0}^{|R|-1}(2^i \times R_i))$.

Calling `realise_outcomes` on the above model thus yields:

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

	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

with rownames indicating nodal types and columns ralized values. Intervening on X with $do(X = 1)$ yields:

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

	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

Parameter matrix

The parameters dataframe keeps track of parameter values and priors for parameters but it does not provide a mapping between parameters and the probability of causal types.

The parameter matrix (P matrix) can be added to the model to provide this mapping. The P matrix has a row for each parameter and a column for each causal type. For instance:

```r
R> make_model("X->Y") |>
+  get_parameter_matrix() |>
+  knitr::kable()
```

	X0.Y00	X1.Y00	X0.Y10	X1.Y10	X0.Y01	X1.Y01	X0.Y11	X1.Y11
X.0	1	0	1	0	1	0	1	0
X.1	0	1	0	1	0	1	0	1
Y.00	1	1	0	0	0	0	0	0
Y.10	0	0	1	1	0	0	0	0
Y.01	0	0	0	0	1	1	0	0
Y.11	0	0	0	0	0	0	1	1

The probability of a causal type is given by the product of the parameters values for parameters whose row in the P matrix contains a 1.

Below we will see examples where the P matrix helps keep track of parameters that are created when confounding is added to a model. CROSS REFERENCE

To speed up different operations it is sometuimes useful to have P added to the model:

- `set_parameter_matrix…`

3.3. Tailoring models

Setting restrictions

TILL: READ THROUGH

When a model is defined, the complete set of possible causal relations are worked out. This set can be very large.

Sometimes for theoretical or practical reasons it is useful to constrain the set of types. In `CausalQueries` this is done at the level of nodal types, with restrictions on causal types following from restrictions on nodal types.

For instance to impose an assumption that Y is not decreasing in X we generate a restricted model as follows:

```r
R> model <- make_model("X->Y") |> set_restrictions("Y[X=1] < Y[X=0]")
```

or:

```r
R> model <- make_model("X->Y") |> set_restrictions(decreasing("X", "Y"))
```

Viewing the resulting parameter matrix we see that both the set of parameters and the set of causal types are now restricted:

```r
R> get_parameter_matrix(model)
```

```
Rows are parameters, grouped in parameter sets

Columns are causal types

Cell entries indicate whether a parameter probability is used
in the calculation of causal type probability

      X0.Y00 X1.Y00 X0.Y01 X1.Y01 X0.Y11 X1.Y11
X.0        1       0       1       0       1       0
X.1        0       1       0       1       0       1
Y.00       1       1       0       0       0       0
Y.01       0       0       1       1       0       0
Y.11       0       0       0       0       1       1


 param_set  (P)
```

Here and in general, setting restrictions typically involves using causal syntax; see Section @ref(syntax) for a guide the syntax used by `CausalQueries`.

Note:

- Restrictions have to operate on nodal types: restrictions on *levels* of endogenous nodes aren't allowed. This, for example, will fail: `make_model("X->Y") |> set_restrictions(statement =  "(Y == 1)")`. The reason is that it requests a correlated restriction on nodal types for `X` and `Y` which involves undeclared confounding.
- Restrictions implicitly assume fixed values for *all* parents of a node. For instance: `make_model("A -> B <- C") |> set_restrictions("(B[C=1]==1)")` is interpreted as shorthand for the restriction `"B[C = 1, A = 0]==1 | B[C = 1, A = 1]==1"`.
- To place restrictions on multiple nodes at the same time, provide these as a vector of restrictions. This is not permitted: `set_restrictions("Y[X=1]==1 & X==1")`, since it requests correlated restrictions. This however is allowed: `set_restrictions(c("Y[X=1]==1", "X==1"))`.

- Use the `keep` argument to indicate whether nodal types should be dropped (default) or retained.
- Restrictions can be set using nodal type labels.

```
R> make_model("S -> C -> Y <- R <- X; X -> C -> R") |>
+  set_restrictions(labels = list(C = "1000", R = "0001", Y = "0001"),
+                   keep = TRUE)
```

- Wild cards can be used in nodal type labels:

```
R> make_model("X->Y") |>
+  set_restrictions(labels = list(Y = "Y?0"))
```

- adding "given" for restrictions with models with confounding

Allowing confounding

(Unobserved) confounding between two nodes arises when the nodal types for the nodes are not independently distributed.

In the $X \rightarrow Y$ graph, for instance, there are 2 nodal types for X and 4 for Y. There are thus 8 joint nodal types (or causal types):

		θ^X		
		0	1	Sum
θ^Y	00	$\Pr(\theta_0^X, \theta_{00}^Y)$	$\Pr(\theta_1^X, \theta_{00}^Y)$	$\Pr(\theta_{00}^Y)$
	10	$\Pr(\theta_0^X, \theta_{10}^Y)$	$\Pr(\theta_1^X, \theta_{10}^Y)$	$\Pr(\theta_{10}^Y)$
	01	$\Pr(\theta_0^X, \theta_{01}^Y)$	$\Pr(\theta_1^X, \theta_{01}^Y)$	$\Pr(\theta_{01}^Y)$
	11	$\Pr(\theta_0^X, \theta_{11}^Y)$	$\Pr(\theta_1^X, \theta_{11}^Y)$	$\Pr(\theta_{11}^Y)$
	Sum	$\Pr(\theta_0^X)$	$\Pr(\theta_1^X)$	1

This table has 8 interior elements and so an unconstrained joint distribution would have 7 degrees of freedom. A no confounding assumption means that $\Pr(\theta^X|\theta^Y) = \Pr(\theta^X)$, or $\Pr(\theta^X, \theta^Y) = \Pr(\theta^X)\Pr(\theta^Y)$. In this case we just 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.

```
R> confounded <- make_model("X->Y ; X <-> Y")
R>
R> confounded$parameters_df |> knitr::kable()
```

param_names	node	gen	param_set	nodal_type	given	param_value	priors
X.0	X	1	X	0		0.50	1
X.1	X	1	X	1		0.50	1
Y.00_X.0	Y	2	Y.X.0	00	X.0	0.25	1
Y.10_X.0	Y	2	Y.X.0	10	X.0	0.25	1
Y.01_X.0	Y	2	Y.X.0	01	X.0	0.25	1
Y.11_X.0	Y	2	Y.X.0	11	X.0	0.25	1
Y.00_X.1	Y	2	Y.X.1	00	X.1	0.25	1
Y.10_X.1	Y	2	Y.X.1	10	X.1	0.25	1
Y.01_X.1	Y	2	Y.X.1	01	X.1	0.25	1
Y.11_X.1	Y	2	Y.X.1	11	X.1	0.25	1

We see here that there are now two parameter families for parameters associated with the node Y. Each family captures the conditional distribution of Y's nodal types, given X. For instance the parameter `Y01_X.1` can be interpreted as $\Pr(\theta^Y = \theta_{01}^Y|X = 1)$.

To see exactly how the parameters map to causal types we can view the parameter matrix:

```
R> get_parameter_matrix(confounded) |> knitr::kable()
```

	X0.Y00	X1.Y00	X0.Y10	X1.Y10	X0.Y01	X1.Y01	X0.Y11	X1.Y11
X.0	1	0	1	0	1	0	1	0
X.1	0	1	0	1	0	1	0	1
Y.00_X.0	1	0	0	0	0	0	0	0
Y.10_X.0	0	0	1	0	0	0	0	0
Y.01_X.0	0	0	0	0	1	0	0	0
Y.11_X.0	0	0	0	0	0	0	1	0
Y.00_X.1	0	1	0	0	0	0	0	0
Y.10_X.1	0	0	0	1	0	0	0	0
Y.01_X.1	0	0	0	0	0	1	0	0
Y.11_X.1	0	0	0	0	0	0	0	1

Importantly, the P matrix works as before, despite confounding. We can assess the probability of causal types by multiplying the probabilities of the constituent parameters.

- Unlike nodal restrictions, a confounding relation can involve multiple nodal types simultaneously. For instance `make_model("X -> M -> Y") |> set_confound(list(X = "Y[X=1] > Y[X=0]"))` allows for a parameter that renders X more or less likely depending on whether X has a positive effect on Y whether it runs through a positive or a negative effect on M.
- The parameters needed to capture confounding relations depend on the direction of causal arrows. For example compare:
  - `make_model("A -> W <- B ; A <-> W; B <-> W")$parameters_df |> dim` In this case we can decompose shocks on A, B, W via: $\Pr(\theta^A, \theta^B, \theta^W) = \Pr(\theta^W|\theta^A, \theta^A)\Pr(\theta^A)\Pr(\theta^B)$, and we have 68 parameters.
  - `make_model("A <- W -> B ; A <-> W; B <-> W")$parameters_df |> dim` In this case we have $\Pr(\theta^A, \theta^B, \theta^W) = \Pr(\theta^A|\theta^W)\Pr(\theta^B|\theta^W)\Pr(\theta^W)$ and just has just 18 parameters.

```
R> make_model("A <- X -> B; A <-> X; B <-> X") |>
+  plot()
```
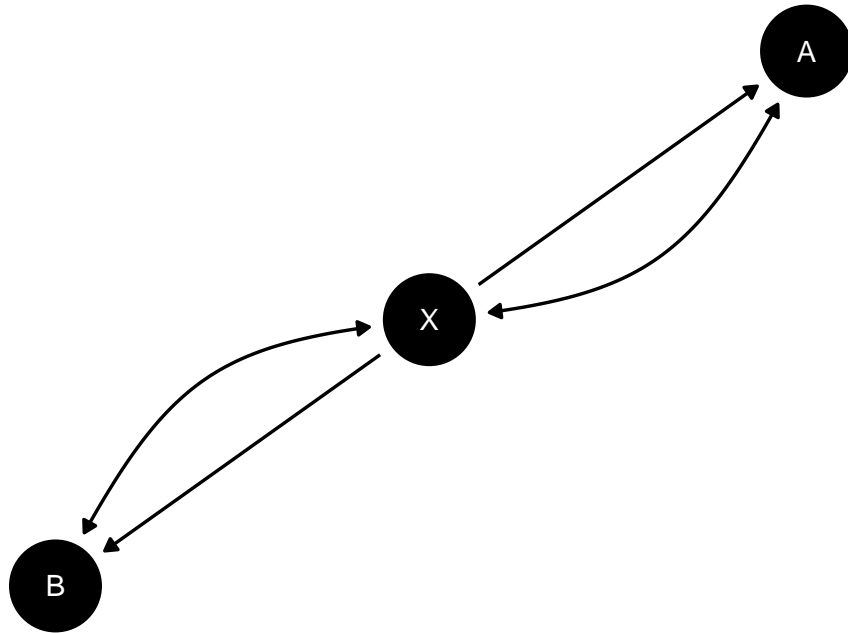
Setting Priors

Priors on model parameters can be added to the parameters dataframe. The priors are interpreted as "alpha" arguments for a Dirichlet distribution. The Dirichlet distribution is a probability distribution over an $n-1$ dimensional unit simplex. It can be thought of as a generalization of the Beta distribution and is parameterized by an n-dimensional vector positive vector α. Thus for example a Dirichlet with $\alpha = (1, 1, 1, 1, 1)$ gives a probability distrubution over all non negative 5 dimensional vectors that sum to 1, e.g. (0.1, 0.1, 0.1, 0.1, 0.6) or (0.1, 0.2, 0.3, 0.3, 0.1). This particular value for α implies that all such vectors are equally likely. Other values for α can be used to control the expectation for each dimension as well as certainty. Thus for instance the vector $\alpha = (100, 1, 1, 1, 100)$ woudl result in more weight on distributions that are close to (.5, 0, 0, 0. .5).

In `CausalQueries` priors are generally specified over the distribution of nodal types (or over the conditional distribution of nodal types, when there is confounding). Thus for instance in an $X \to Y$ model we have one Dirichlet distribution over the two types for θ^X and one Dirichlet distribution over the four types for θ^Y.

To wit:

```
R> make_model("X->Y") |> get_priors()

 X.0  X.1 Y.00 Y.10 Y.01 Y.11
   1    1    1    1    1    1
```

Here the priors have not been specified and so they default to 1, which corresponds to uniform priors.

Alternatively you could set jeffreys priors like this:

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

```
+  set_priors(distribution = "jeffreys") |>
+  get_priors()

 X.0  X.1 Y.00 Y.10 Y.01 Y.11
 0.5  0.5  0.5  0.5  0.5  0.5
```

Custom priors are most simply specified by being added as a vector of numbers using `set_priors`. For instance:

```
R> make_model("X->Y") |>
+  set_priors(1:6) |>
+  get_priors()

 X.0  X.1 Y.00 Y.10 Y.01 Y.11
   1    2    3    4    5    6
```

The priors here should be interpreted as indicating:

- $\alpha_X = (1,2)$, which implies a distribution over $(\lambda_0^X, \lambda_1^X)$ centered on $(1/3, 2/3)$.
- $\alpha_Y = (3,4,5,6)$, which implies a distribution over $(\lambda_{00}^Y, \lambda_{10}^Y, \lambda_{01}^Y \lambda_{11}^Y)$ centered on $(3/18, 4/18, 5/18, 6/18)$.

For larger models it can be hard to provide priors as a vector of numbers and so `set_priors` can allow for more targeted modifications of the parameter vector. For instance:

```
R> make_model("X->Y") |>
+  set_priors(statement = "Y[X=1] > Y[X=0]", alphas = 3) |>
+  get_priors()

 X.0  X.1 Y.00 Y.10 Y.01 Y.11
   1    1    1    1    3    1
```

As setting priors simply requires mapping alpha values to parameters; the process boils down to selecting rows of the `parmeters_df` dataframe, at which to alter values. When specifying a causal statement as above, CausalQueries internally identifies nodal types that are consistent with the statement, which in turn identify parameters to alter priors for. Note that there is nothing particularly unique about setting priors via causal syntax, as it is simply an abstract way of specifying how to subset the `parameters_df` dataframe. To wit; we can achieve the same result as above by specifying:

```
R> make_model("X->Y") |>
+  set_priors(nodal_type = "01", alphas = 3) |>
+  get_priors()

 X.0  X.1 Y.00 Y.10 Y.01 Y.11
   1    1    1    1    3    1
```

or even:

```
R> make_model("X->Y") |>
+  set_priors(param_names = "Y.01", alphas = 3) |>
+  get_priors()

 X.0  X.1 Y.00 Y.10 Y.01 Y.11
```

```
   1    1    1    1    3    1
```

As such `set_priors` allows for the specification of any non-redundant combination of arguments on the `param_names`, `node`, `nodal_type`, `param_set` and `given` columns of `parameters_df` to uniquely identify parameters to set priors for. Alternatively a fully formed subsetting statement may be supplied to `alter_at`. Since all these arguments get mapped to the parameters the identify internally they may be used interchangeably.[2]

```
R> make_model("X-> M -> Y; X <-> Y") |>
+  set_priors(node = "Y", nodal_type = c("01","11"), given = "X.1", alphas = c(3,2)) |>
+  get_priors()

    X.0      X.1     M.00     M.10     M.01     M.11 Y.00_X.0 Y.10_X.0
      1        1        1        1        1        1        1        1
Y.01_X.0 Y.11_X.0 Y.00_X.1 Y.10_X.1 Y.01_X.1 Y.11_X.1
      1        1        1        1        3        2

R> make_model("X-> M -> Y; X <-> Y") |>
+  set_priors(alter_at = "node == 'Y' & nodal_type %in% c('01','11') & given == 'X.1'", al
+  get_priors()

    X.0      X.1     M.00     M.10     M.01     M.11 Y.00_X.0 Y.10_X.0
      1        1        1        1        1        1        1        1
Y.01_X.0 Y.11_X.0 Y.00_X.1 Y.10_X.1 Y.01_X.1 Y.11_X.1
      1        1        1        1        3        2
```

It should be noted that while highly targeted prior setting is convenient and flexible; it should be done with caution. Setting priors on specific parameters in complex models; especially models involving confounding, may strongly affect inferences in intractable ways.

We aditionally note that flat priors over nodal types do not necessarily translate into flat priors over queries. "Flat" priors over parameters in a parameter family put equal weight on each nodal type, but this in turn can translate into strong assumptions on causal quantities of interest.

For instance in an $X \to Y$ model model in which negative effects are ruled out, the average causal effect implied by "flat" priors is $1/3$. This can be seen by querying the model:

```
R> make_model("X -> Y") |>
+  set_restrictions(decreasing("X", "Y")) |>
+  query_model("Y[X=1] - Y[X=0]", n_draws = 10000) |>
+  kable(digits = 2)
```

query	given	using	case_level	mean	sd	cred.low.2.5%	cred.high.97.5%
Y[X=1] - Y[X=0]	-	parameters	FALSE	0.33	NA	0.33	0.33

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

---

[2]See `?set_priors` and `?make_priors` for many more examples.

Again, you can use `query_model` to figure out what flat (or other) priors over parameters imply for priors over causal quantities:

```
R> make_model("X -> Y") |>
+  query_model("Y[X=1] > Y[X=0]", n_draws = 10000) |>
+  knitr::kable()
```

query	given	using	case_level	mean	sd	cred.low.2.5%	cred.high.97.5%
Y[X=1] > Y[X=0]	-	parameters	FALSE	0.25	NA	0.25	0.25

```
R> make_model("X -> M -> Y") |>
+  query_model("Y[X=1] > Y[X=0]", n_draws = 10000) |>
+  knitr::kable()
```

query	given	using	case_level	mean	sd	cred.low.2.5%	cred.high.97.5%
Y[X=1] > Y[X=0]	-	parameters	FALSE	0.125	NA	0.125	0.125

Caution regarding priors is particularly important when models are not identified, as is the case for many of the models considered here. In such cases, for some quantities, the marginal posterior distribution can be the same as the marginal prior distribution (Poirier 1998).

The key point here is to make sure you do not fall into a trap of thinking that "uninformative" priors make no commitments regarding the values of causal quantities of interest. They do, and the implications of flat priors for causal quantities can depend on the structure of the model. Moreover for some inferences from causal models the priors can matter a lot even if you have a lot of data. In such cases it can be helpful to know what priors on parameters imply for priors on causal quantities of interest (by using `query_model`) and to assess how much conclusions depend on priors (by comparing results across models that vary in their priors).

Setting Parameters

By default, models have a vector of parameter values included in the `parameters_df` dataframe. These are useful for generating data, or for situations, such as process tracing, when one wants to make inferences about causal types (θ), given case level data, under the assumption that the model is known.

The logic for setting parameters is similar to that for setting priors: effectively we need to place values on the probability of nodal types. The key difference is that whereas the *alpha* value placed on a nodal types can be any positive number—capturing our certainty over the parameter value—the parameter values must lie in the unit interval, $[0, 1]$. In general if parameter values are passed that do not lie in the unit interval, these are normalized so that they do.

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

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

```
 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 Y in the case it is increasing in X. Here nodal type Y.Y01 is set to be .5, while the other nodal types of this parameter set were renormalized so that the parameters in the set still sum to one.

```
R> make_model("X->Y") |>
+  set_parameters(statement = "Y[X=1] > Y[X=0]", parameters = .5) |>
+  get_parameters()

      X.0       X.1      Y.00      Y.10      Y.01      Y.11
0.5000000 0.5000000 0.1666667 0.1666667 0.5000000 0.1666667
```

3.4. Drawing and manipulating data

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

Drawing data basics

By default, the parameters used are taken from `model$parameters_df`.

```
R> model <- make_model("X -> M -> Y")
R>
R> sample_data_1 <-
+  model |> make_data(n = 4)
R>
R> sample_data_1 |> kable()
```

X	M	Y
0	1	0
0	1	0
1	0	0
1	1	0

However you can also specify parameters directly or use parameter draws from a prior or posterior distribution. For instance:

```
R> make_data(model, n = 3, param_type = "prior_draw")
```

Note that the data is returned ordered by data type as in the example above.

Drawing incomplete data

`CausalQueries` can be used in settings in which researchers have gathered different amounts of data for different nodes. For instance gathering X and Y data for all units but M data only for some.

The function `make_data` allows you to draw data like this if you specify a data strategy indicating the probabilities of observing data on different nodes, possibly as a function of prior nodes observed.

```
R> set.seed(1)
R> sample_data_2 <-
+  make_data(
+  model,
+  n = 8,
+  nodes = list(c("X", "Y"), "M"),
+  probs = list(1, .5),
+  subsets = list(TRUE, "X==1 & Y==0"))

# A tibble: 2 x 5
  node_names nodes       n_steps probs subsets
  <chr>      <list>      <lgl>   <dbl> <chr>
1 X, Y       <chr [2]>   NA          1 TRUE
2 M          <chr [1]>   NA        0.5 X==1 & Y==0

R> sample_data_2 |> kable()
```

X	M	Y
0	NA	0
0	NA	1
1	NA	0
1	NA	1
1	NA	1
1	1	0
1	NA	0
1	1	0

Reshaping data

Whereas data naturally comes in long form, with a row per observation, as in the examples above, the data passed to `stan` is in a compact form, which records only the number of units of each data type. `CausalQueries` includes functions that lets you move between these two forms in case of need.

```
R> sample_data_1 |> collapse_data(model) |> kable()
```

If data has been gathered using a data strategy the collapsed data groups data types according to which nodes have been observed:

```
R> sample_data_2 |> collapse_data(model) |> kable()
```

In the same way it is possible to move from "compact data" to "long data" using 'expand_data"

4. Updating models

The approach used by the `CausalQueries` package to updating parameter values given observed data uses `stan` and involves the following elements:

- Dirichlet priors over parameters, λ (which, in cases without confounding, correspond to nodal types)
- A mapping from parameters to event probabilities, w
- A likelihood function that assumes events are distributed according to a multinomial distribution given event probabilities.

We provide further details below.

4.1. Data for stan

GS: general edit to this section

We use a generic `stan` model that works for all binary causal models. Rather than writing a new `stan` model for each causal model we send `stan` details of each particular causal model as data inputs.

In particular we provide a set of matrices that `stan` tailor itself to particular models: the parameter matrix (P) tells `stan` how many parameters there are, and how they map into causal types; an ambiguity matrix A tells `stan` how causal types map into data types; and an event matrix E relates data types into patterns of observed data (in cases where there are incomplete observations).

The internal function `prep_stan_data` prepares data for `stan`. You generally don't need to use this manually, but we show here a sample of what it produces as input for `stan`.

Note that NAs are interpreted as data not having been sought. So in this case the interpretation is that there are two data strategies: data on Y and X was sought in three cases; data on Y only was sought in just one case.

4.2. How the stan model works

THIS DISCUSSION IS OUT OF DATE:

The `stan` model works as follows:

MH: ADD DISCUSSION OF parmap;

- We are interested in "sets" of parameters. For example in the $X \rightarrow Y$ model we have two parameter sets (`param_sets`). The first is $\lambda^X \in \{\lambda_0^X, \lambda_1^X\}$ whose elements give the probability that X is 0 or 1. These two probabilities sum to one. The second parameter set is $\lambda^Y \in \{\lambda_{00}^Y, \lambda_{10}^Y, \lambda_{01}^Y \lambda_{11}^Y\}$. These are also probabilities and their values sum to one. Note in all that we have 6 parameters but just $1 + 3 = 4$ degrees of freedom.

- We would like to express priors over these parameters using multiple Dirichlet distributions (two in this case). In practice because we are dealing with multiple simplices of varying length, it is easier to express priors over gamma distributions with a unit scale parameter and shape parameter corresponding to the Dirichlet priors, α. We make use of the fact that $\lambda_0^X \sim Gamma(\alpha_0^X, 1)$ and $\lambda_1^X \sim Gamma(\alpha_1^X, 1)$ then

$\frac{1}{\lambda_0^X + \lambda_1^X}(\lambda_0^X, \lambda_1^X) \sim Dirichlet(\alpha_0^X, \alpha_1^X)$. For a discussion of implementation of this approach in `stan` see https://discourse.mc-stan.org/t/ragged-array-of-simplexes/1382.

- For any candidate parameter vector λ we calculate the probability of *causal* types (`prob_of_types`) by taking, for each type i, the product of the probabilities of all parameters (λ_j) that appear in column i of the parameter matrix P. Thus the probability of a (X_0, Y_{00}) case is just $\lambda_0^X \times \lambda_{00}^Y$. The implementations in `stan` uses `prob_of_types_[i]` $= \prod_j \left(P_{j,i}\lambda_j + (1 - P_{j,i}) \right)$: this multiplies the probability of all parameters involved in the causal type (and substitutes 1s for parameters that are not). (`P` and `not_P` (1-P) are provided as data to `stan`).

- The probability of data types, `w`, is given by summing up the probabilities of all causal types that produce a given data type. For example, the probability of a $X = 0, Y = 0$ case, w_{00} is $\lambda_0^X \times \lambda_{00}^Y + \lambda_0^X \times \lambda_{01}^Y$. The ambiguity matrix A is provided to `stan` to indicate which probabilities need to be summed.

- 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. The probability of an observed event, within a data strategy, is given by summing the probabilities of the types that could give rise to the incomplete data. For example X is observed, but Y is not, then the probability of $X = 0, Y = $ NA is $w_{00} + w_{01}$. The matrix E is passed to `stan` to figure out which event probabilities need to be combined for events with missing data.

- The probability of a dataset is then given by a multinomial distribution with these event probabilities (or, in the case of incomplete data, the product of multinomials, one for each data strategy).

GS: TO CHECK THE PATHS FUNCTIONALITY

4.3. Implementation

To update a CausalQueries model with data use:

```
R> update_model(model, data)
```

where the data argument is a dataset containing some or all of the nodes in the model.

As `update_model()` calls `rstan::sampling` one can pass along all arguments in `...` to `rstan::sampling`. For instance:

- `iter` sets the number of iterations and ultimately the number of draws in the posterior
- `chains` sets the number of chains; doing multiple chains in parallel speeds things up

4.4. Parallelization

If you have multiple cores you can do parallel processing by including this line before running `CausalQueries`:

```
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 2 updating processes with 3 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> 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_2)
R>
R> future.apply::future_lapply(data, function(d) {
+  update_model(
+    model = model,
+    data = d,
+    chains = chains,
+    refresh = 0
+  )
+})
```

4.5. Incomplete and censored data

`CausalQueries` assumes that missing data is missing at random, conditional on observed data. Thus for instance in a $X \to M \to Y$ model one might choose to observe M in a random set of cases in which $X = 1$ and $Y = 1$. In that case if there are positive relations at each stage you may be more likely to observe M in cases in which $M = 1$. However observation of M is still random *conditional* on the observed X and Y data. The `stan` model in `CausalQueries` takes account of this kind of sampling naturally by assessing the probability of observing a particular pattern of data within each data strategy. For a discussion see section 9.2.3.2 of Humphreys and Jacobs (2023).

In addition it is possible to indicate when data has been censored and for the `stan` model to take this into account also. Say for instance that we only get to observe X in cases where $X = 1$ and not when $X = 0$. This kind of sampling is non random conditional on observables. It is taken account however by indicating to `stan` that the probability of observing a particular data type is 0, regardless of parameter values. This is done 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 $X \neg Y$ has been censored then when we update we do not move towards a belief that X causes Y.

```
R> list(uncensored = make_model("X->Y") %>%
```

```
+        update_model(data.frame(X=rep(0:1,5), Y=rep(0:1,5)),
+                     refresh = 0, iter = 3000),
+    censored = make_model("X->Y") %>%
+       update_model(data.frame(X=rep(0:1,5), Y=rep(0:1,5)),
+                    censored_types = c("X1Y0",  "X0Y1"),
+                    refresh = 0, iter = 3000)) %>%
+  query_model(te("X", "Y"), using = "posteriors") |>
+  select(model, query, mean, sd) |>
+  kable(digits = 2)
```

model	query	mean	sd
uncensored	(Y[X=1] - Y[X=0])	0.59	0.20
censored	(Y[X=1] - Y[X=0])	0.01	0.31

4.6. Output

The primary output from `update_model()` is a posterior distribution over model parameters, stored as a dataframe in `model$posterior_distribution`. However another of other objects are also optionally stored:

Models can be queried using the `query_distribution` and `query_model` functions. The difference between these functions is that `query_distribution` examines a single query and returns a full distribution of draws from the distribution of the estimand (prior or posterior); `query_model` takes a collection of queries and returns a dataframe with summary statistics on the queries.

5. Queries

LM: REVIEW AND FIX SECTION

WRITE SECTION DESCRIBING FACTUAL AND COUNTERFACTULA QUERIES AND THE SYNTAX INCLUDING Role of [] AND OF GIVEN

get_query_types

return sets; discuss how all querties related to these sets

Another way to develop a clearer understanding of how to interpret nodal types subscripts is to consider the conditions they may fulfill. We refer to these conditions as 'causal queries'. Causal queries are useful because they allow us to target and alter specific nodal types, and they come in handy for functions that we will introduce below. For instance, in a model such as $X \to Y \gets M$, you may want to identify the nodal types that correspond to cases where increasing both X and M from 0 to 1 results in an increase in Y. You can do this using the `get_query_types` function as shown below:

start off with querty Y==1 then Y[X=1] ==1 then Y[X=1] >= Y[X=0] then show linear functions possible Y[X=1] - Y[X=0]

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

```
R> query <- 'Y[X=1, M=1] > Y[X=0, M=0]'
R> get_query_types(model, query, map= "nodal_type")
```

```
Nodal types adding weight to query

 query :  Y[X=1,M=1]>Y[X=0,M=0]

 0001    0101
 0011    0111


 Number of nodal types that add weight to query = 4
 Total number of nodal types related to Y = 16
```
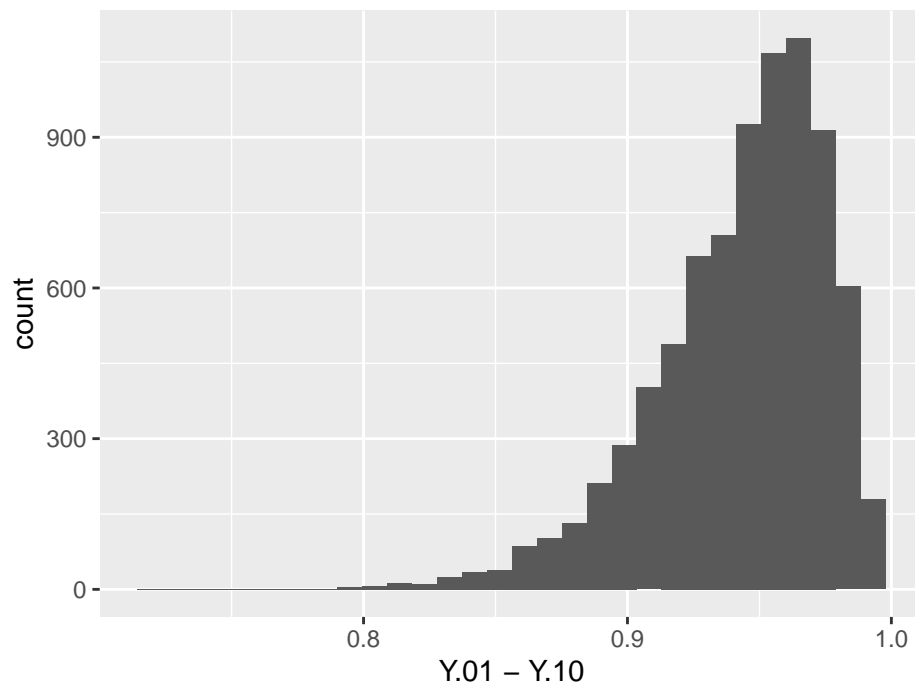
5.1. Query implementation

5.2. Queries by hand

Queries can draw directly from the prior distribution of the posterior distribution provided by stan.

```
R> library(DeclareDesign)
R>
R> data  <- fabricate(N = 10, X = complete_ra(N), Y = X)
R>
R> model <- make_model("X -> Y") |>
+  update_model(data, iter  = 4000)
R>
R> model$posterior_distribution |>
+  ggplot(aes(Y.01 - Y.10)) +
+          geom_histogram()
```

5.3. Query distribution

`query_distribution` works similarly except that distributions over queries are returned:

```
R> make_model("X -> Y") |>
+  query_distribution(
+    query = list(increasing = "(Y[X=1] > Y[X=0])"),
+    using = "priors") |>
+  ggplot(aes(increasing)) +
+  geom_histogram()
```
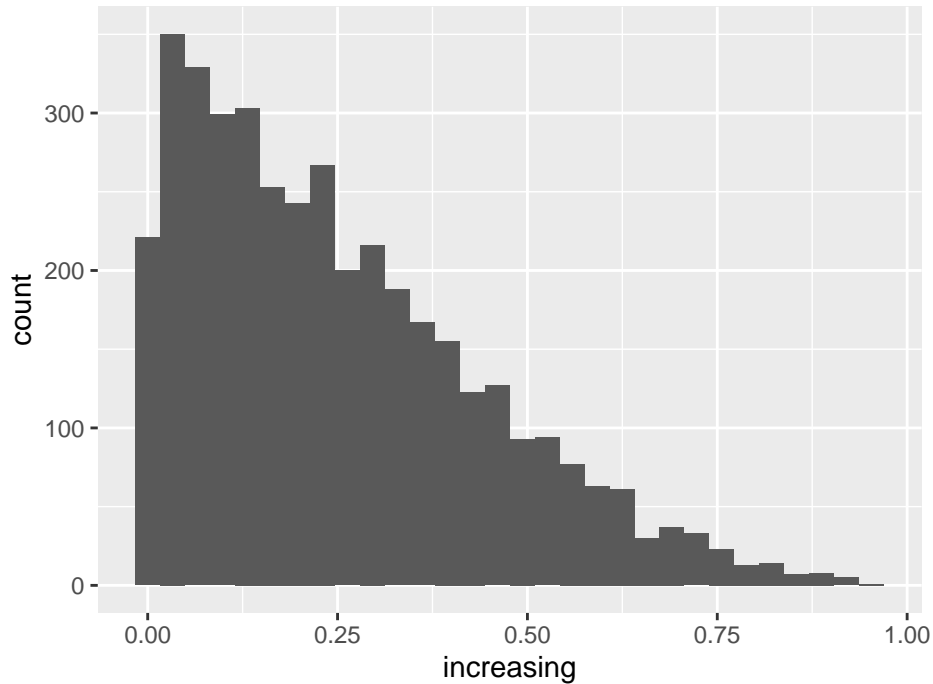
Figure 2: Prior on 'Probability Y is increasing in X'

5.4. Case level queries

Sometimes one is interested in assessing the value of a query for a *particular case*. In a sense this is equivalent to posing a conditional query, querying conditional on valuesin a case. For instance we might consult our posterior lipds 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]",
+            given = c("X==1 & Y==1 & Z==1"),
+            using = "posteriors")

          query                given   using case_level      mean         sd
1 Y[X=1] - Y[X=0] X==1 & Y==1 & Z==1 posteriors      FALSE 0.9541329 0.03575594
  cred.low.2.5% cred.high.97.5%
1     0.8624872       0.9969585
```

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

Subtly though this is, in principle, different to what we would infer for a a 'new case' that we wonder about. When inquiring about a new case, the case level query *updates* on the given information observed in the new case. The resulting inference is different to the inference that would be made from the posterior *given* the features of the case.

To illustrate, consider a model for which we are quite sure that X causes Y but we do not know whether it works through two positive effects or two negative effects.

Thus we do not know if M=0 would suggest an effect or no effect. If asked what we would infer for a case that had $M = 0$ we would not know whether $M = 0$ information is consistent with a positive effect or not. However if provided with a randomly case and learn that it has $M = 0$, then we update about the causal model and infer that there is an effect in this case (but that there wouldn't be were $M = 1$).

```
R> # DO ALL THIS IN ONE TABLE
R>
R> set.seed(1)
R>  model <-
+   make_model("X -> M -> Y") |>
+   update_model(data.frame(X = rep(0:1, 8), Y = rep(0:1, 8)), iter = 10000)
R>
R>  Q <- "Y[X=1] > Y[X=0]"
R>  G <- "X==1 & Y==1 & M==1"
R>  QG <- "(Y[X=1] > Y[X=0]) & (X==1 & Y==1 & M==1)"
R>
R>  # In this case these are very different:
R>  query_distribution(model, Q, given = G, using = "posteriors")[[1]] |> mean()
R>  query_distribution(model, Q, given = G, using = "posteriors",
+   case_level = TRUE)
R>
R>  # These are equivalent:
R>  # 1. Case level query via function
R>  query_distribution(model, Q, given = G,
+    using = "posteriors", case_level = TRUE)
R>
R>  # 2. Case level query by hand using Bayes
R>  distribution <- query_distribution(
+    model, list(QG = QG, G = G), using = "posteriors")
R>
R>  mean(distribution$QG)/mean(distribution$G)
```

5.5. Batch queries

The `query_model` function takes causal queries and conditions (`given`) and specifies the parameters to be used. The result is a dataframe which can be displayed as a table.

```
R> models <- list(
+ `1` = make_model("X -> Y"),
+ `2` = make_model("X -> Y") |> set_restrictions("Y[X=1] < Y[X=0]")
+ )
R>
R> query_model(
+  models,
```

```
+   query = list(ATE = "Y[X=1] - Y[X=0]", POS = "Y[X=1] > Y[X=0]"),
+   given = c(TRUE,  "Y==1 & X==1"),
+   case_level = c(FALSE, TRUE),
+   using = c("parameters", "priors"),
+   expand_grid = TRUE) |>
+   select(-starts_with("cred")) |>
+
+   kable(digits = 2)
```

model	query	given	using	case_level	mean	sd
1	ATE	-	parameters	FALSE	0.00	NA
2	ATE	-	parameters	FALSE	0.33	NA
1	ATE	-	priors	FALSE	-0.01	0.32
2	ATE	-	priors	FALSE	0.33	0.24
1	ATE	Y==1 & X==1	parameters	FALSE	0.50	NA
2	ATE	Y==1 & X==1	parameters	FALSE	0.50	NA
1	ATE	Y==1 & X==1	priors	FALSE	0.50	0.29
2	ATE	Y==1 & X==1	priors	FALSE	0.50	0.29
1	POS	-	parameters	FALSE	0.25	NA
2	POS	-	parameters	FALSE	0.33	NA
1	POS	-	priors	FALSE	0.25	0.20
2	POS	-	priors	FALSE	0.33	0.24
1	POS	Y==1 & X==1	parameters	FALSE	0.50	NA
2	POS	Y==1 & X==1	parameters	FALSE	0.50	NA
1	POS	Y==1 & X==1	priors	FALSE	0.50	0.29
2	POS	Y==1 & X==1	priors	FALSE	0.50	0.29
1	ATE	-	parameters	TRUE	0.00	NA
2	ATE	-	parameters	TRUE	0.33	NA
1	ATE	-	priors	TRUE	-0.01	NA
2	ATE	-	priors	TRUE	0.33	NA
1	ATE	Y==1 & X==1	parameters	TRUE	0.50	NA
2	ATE	Y==1 & X==1	parameters	TRUE	0.50	NA
1	ATE	Y==1 & X==1	priors	TRUE	0.51	NA
2	ATE	Y==1 & X==1	priors	TRUE	0.50	NA
1	POS	-	parameters	TRUE	0.25	NA
2	POS	-	parameters	TRUE	0.33	NA
1	POS	-	priors	TRUE	0.25	NA
2	POS	-	priors	TRUE	0.33	NA
1	POS	Y==1 & X==1	parameters	TRUE	0.50	NA
2	POS	Y==1 & X==1	parameters	TRUE	0.50	NA
1	POS	Y==1 & X==1	priors	TRUE	0.51	NA
2	POS	Y==1 & X==1	priors	TRUE	0.50	NA

Computational details and software requirements

Version	• 1.0.0
Availability	• Stable Release: https://cran.rstudio.com/web/packages/CausalQueries/index.htm
	• 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
	• Rcpp ($>=$ 0.12.0)
	• 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	• dplyr
	• methods
R-Packages Imports	• dagitty ($>=$ 0.3-1)
	• dirmult ($>=$ 0.1.3-4)
	• 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)

:::

The results in this paper were obtained using R~3.4.1 with the **MASS**~7.3.47 package. R itself and all packages used are available from the Comprehensive R Archive Network (CRAN) at

[https://CRAN.R-project.org/].

Acknowledgments

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.

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.

Frangakis CE, Rubin DB (2002). "Principal stratification in causal inference." *Biometrics*, **58**(1), 21–29.

Humphreys M, Jacobs AM (2023). *Integrated Inferences*. Cambridge University Press.

Pearl J (2009). *Causality*. Cambridge university press.

Poirier DJ (1998). "Revising beliefs in nonidentified models." *Econometric Theory*, **14**(4), 483–509.

More technical details

> Appendices can be included after the bibliography (with a page break). Each section within the appendix should have a proper section title (rather than just *Appendix*).
>
> For more technical style details, please check out JSS's style FAQ at [https://www.jstatsoft.org/pages/view/style#frequently-asked-questions] which includes the following topics:
> - Title vs. sentence case.
> - Graphics formatting.
> - Naming conventions.
> - Turning JSS manuscripts into R package vignettes.
> - Trouble shooting.
> - Many other potentially helpful details…

Using BibTeX

> References need to be provided in a BibTeX file (`.bib`). All references should be made with `@cite` syntax. This commands yield different formats of author-year citations and allow to include additional details (e.g.,pages, chapters, …) in brackets. In case you are not familiar with these commands see the JSS style FAQ for details.
>
> Cleaning up BibTeX files is a somewhat tedious task – especially when acquiring the entries automatically from mixed online sources. However, it is important that informations are complete and presented in a consistent style to avoid confusions. JSS requires the following format.
> - item JSS-specific markup (`\proglang`, `\pkg`, `\code`) should be used in the references.
> - item Titles should be in title case.
> - item Journal titles should not be abbreviated and in title case.
> - item DOIs should be included where available.
> - item Software should be properly cited as well. For R packages `citation("pkgname")` typically provides a good starting point.

6. Appendix: stan code

`prep_stan_data` then returns a list of objects that `stan` expects to receive. These include indicators to figure out where a parameter set starts (`l_starts`, `l_ends`) and ends and where a data strategy starts and ends (`strategy_starts`, `strategy_ends`), as well as the matrices described above.

MOVE TO APPENDIX? ADD DISCUSSION OF PARMAP

Below we show the `stan` code. This starts off with a block saying what input data is to be expected. Then there is a characterization of parameters and the transformed parameters.

Then the likelihoods and priors are provided. `stan` takes it from there and generates a posterior distribution.

```
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;
int<lower=1> n_param_each[n_param_sets];
int<lower=1> n_data;
int<lower=1> n_events;
int<lower=1> n_strategies;
int<lower=0, upper=1> keep_transformed;
vector<lower=0>[n_params] lambdas_prior;
int<lower=1> l_starts[n_param_sets];
int<lower=1> l_ends[n_param_sets];
int<lower=1> node_starts[n_nodes];
int<lower=1> node_ends[n_nodes];
int<lower=1> strategy_starts[n_strategies];
int<lower=1> strategy_ends[n_strategies];
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;
int<lower=0> Y[n_events];
}
parameters {
vector<lower=0>[n_params - n_param_sets] gamma;
}
transformed parameters {
vector<lower=0>[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[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;
  // syntax here to return unity as a vector
  lambdas[l_starts[i]] = lambdas_prior[1]/lambdas_prior[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
parlam  = rep_matrix(lambdas, n_paths) .* parmap; // (usual case): [n_par * n_data] * [n_p
// 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;
 w = w / sum(w);
 w_full = E * w;
}
model {
// Dirichlet distributions (earlier versions used gamma)
for (i in 1:n_param_sets) {
  target += dirichlet_lpdf(lambdas[l_starts[i]:l_ends[i]]  | lambdas_prior[l_starts[i] :l_
  target += -n_param_each[i] * log(sum_gammas[i]);
 }
// Multinomials
for (i in 1:n_strategies) {
  target += multinomial_lpmf(
  Y[strategy_starts[i]:strategy_ends[i]] | w_full[strategy_starts[i]:strategy_ends[i]]);
 }
}
// Option to export distribution of causal types
// Note if clause used here to effectively turn off this block if not required
generated quantities{
vector[n_types] prob_of_types;
if (keep_transformed == 1){
for (i in 1:n_types) {
    prob_of_types[i] = prod(P[, i].*lambdas + 1 - P[,i]);
```

```
}}
 if (keep_transformed == 0){
    prob_of_types = rep_vector(1, n_types);
 }
}
```

Affiliation:

Till Tietz
IPI
Reichpietschufer 50
Berlin Germany
E-mail: ttietz2014@gmail.com
URL: https://github.com/till-tietz

Lily Medina
E-mail: lily.medina@berkeley.edu
URL: https:/lilymedina.github.io/

Macartan Humphreys
E-mail: macartan.humphreys@wzb.eu
URL: https://macartan.github.io/

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