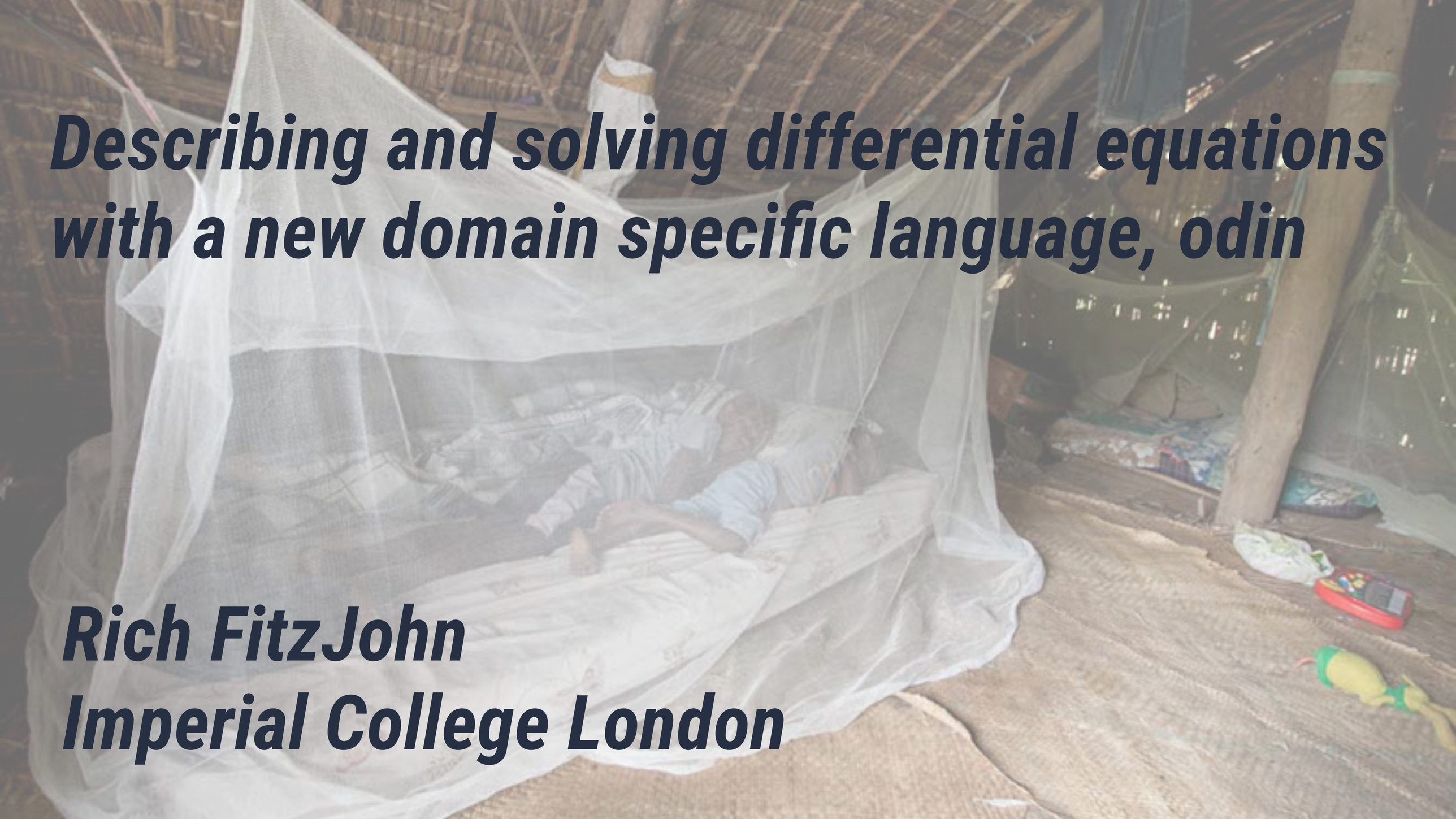
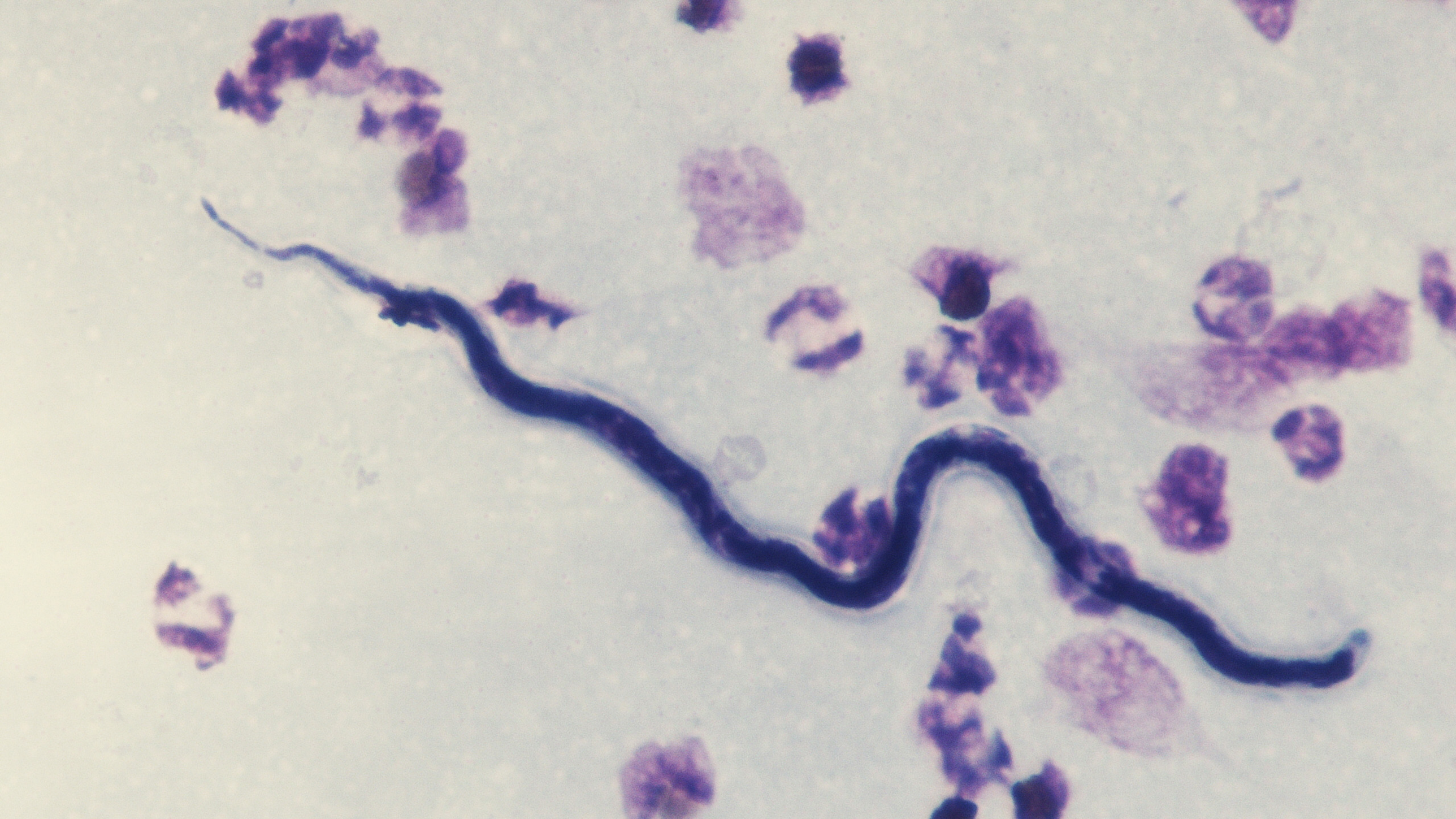
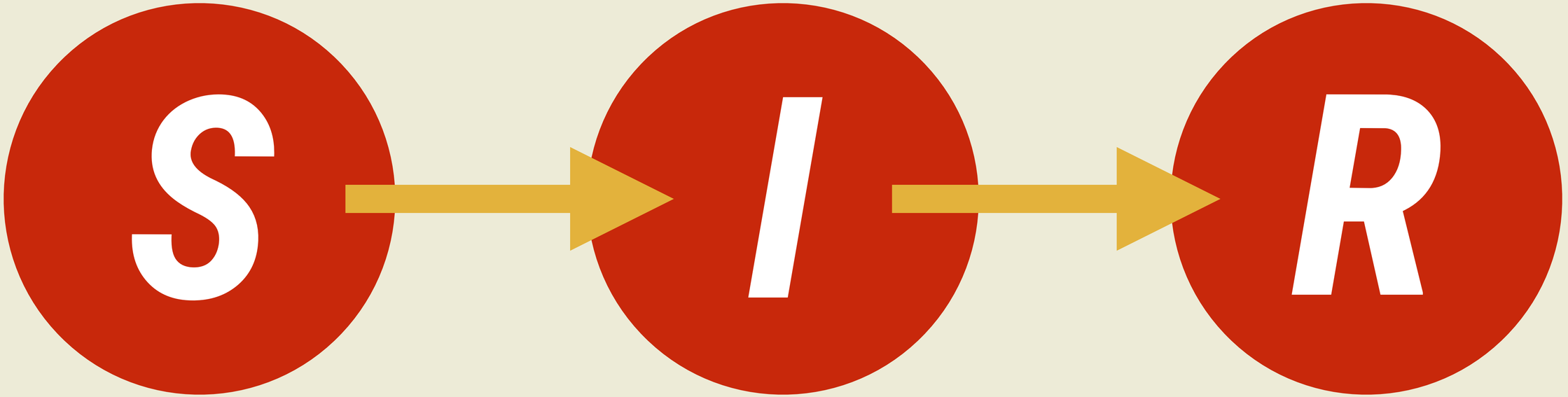


Describing and solving differential equations with a new domain specific language, odin

Rich FitzJohn
Imperial College London




CLINIQUE DES AMIS
DE COTONOU
LG



$$\frac{dS}{dt} \qquad \frac{dI}{dt} \qquad \frac{dR}{dt}$$

BETTER SOFTWARE BETTER RESEARCH

www.software.ac.uk

Alex Hill

RESIDE@IC

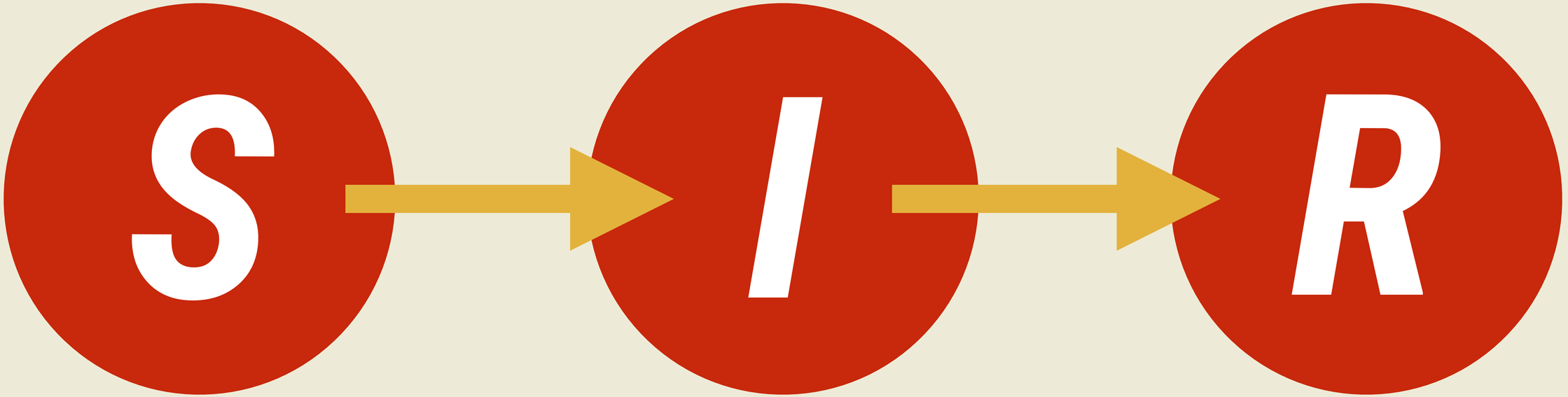
Emma Russell

James Thompson

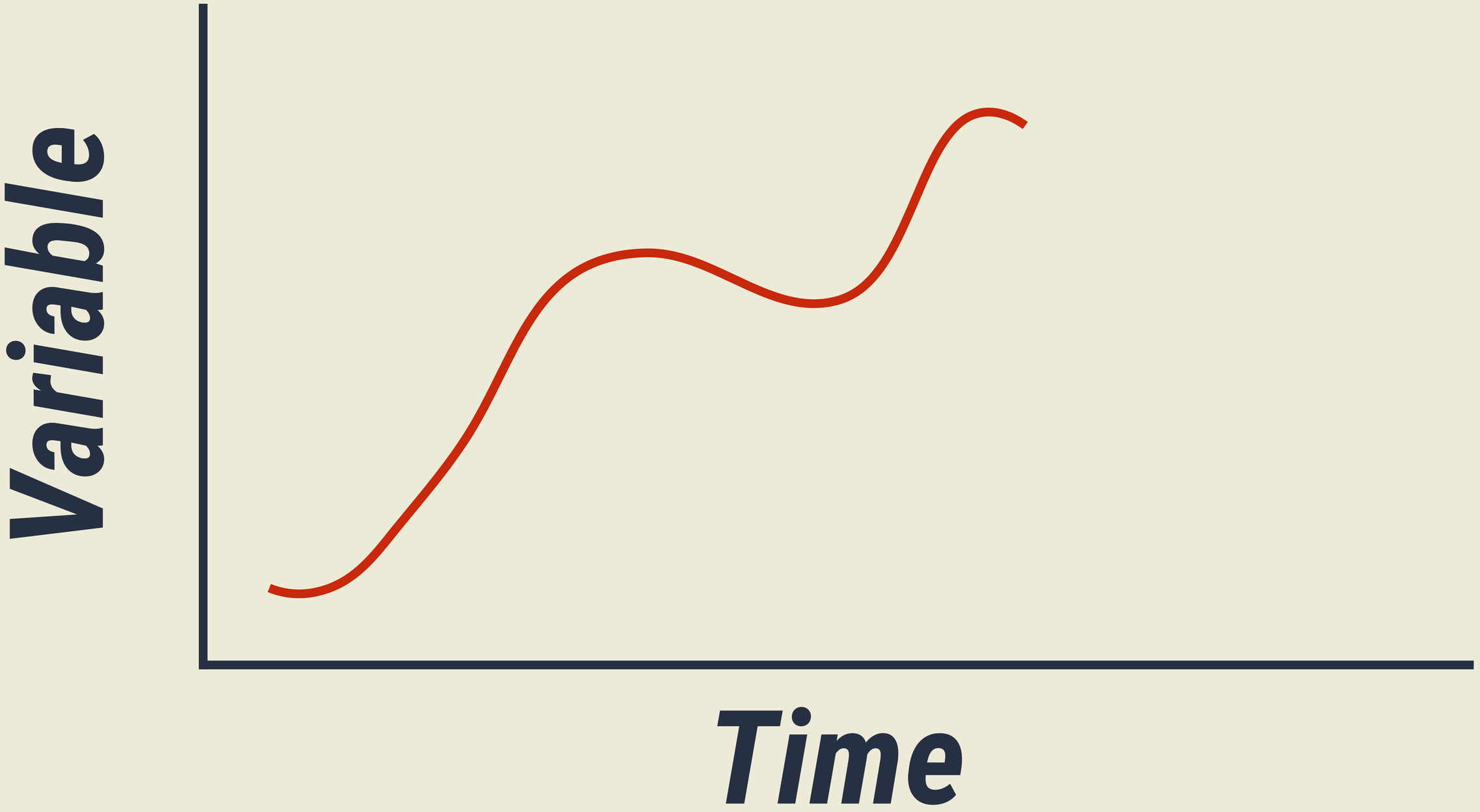
Rob Ashton

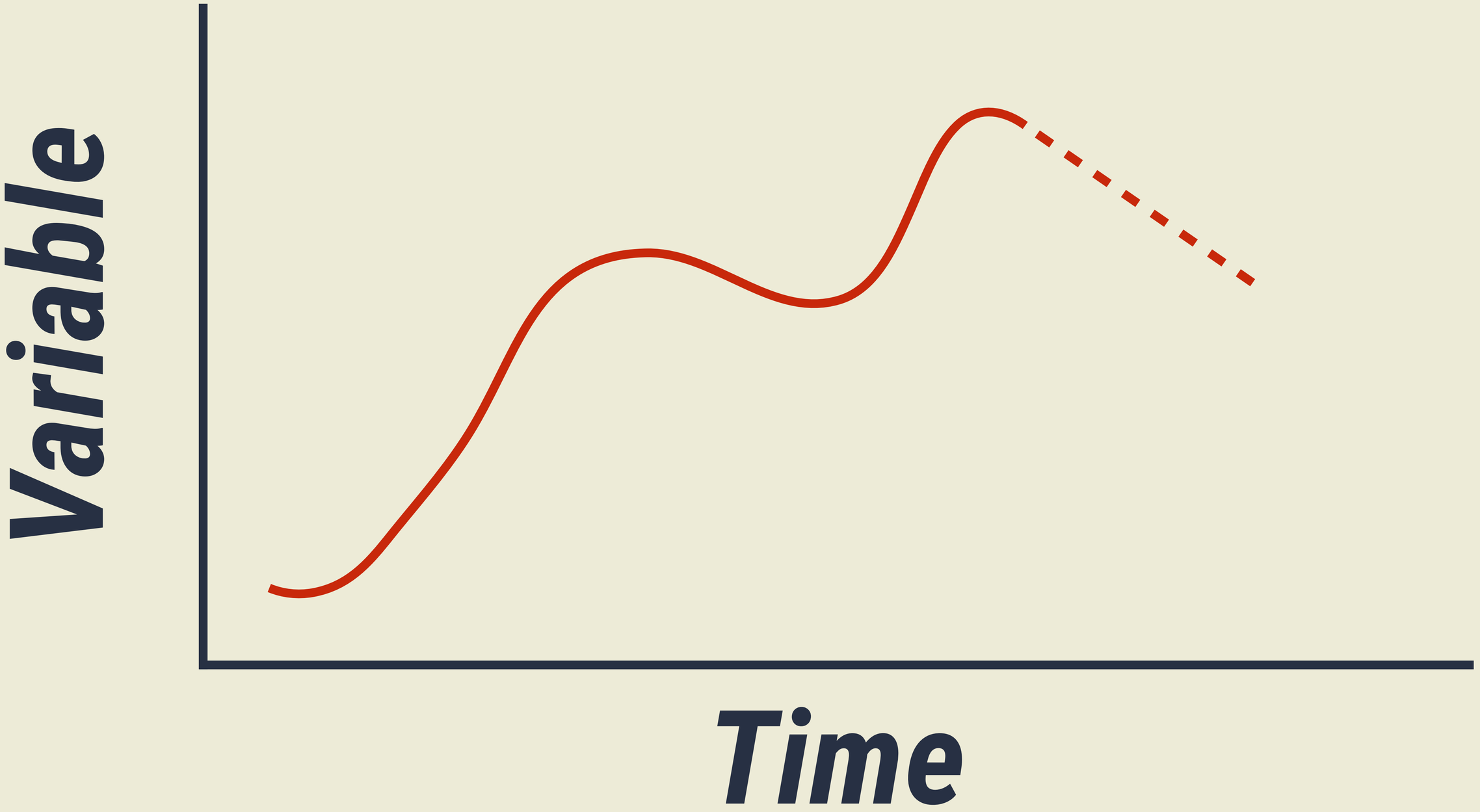
Wes Hinsley

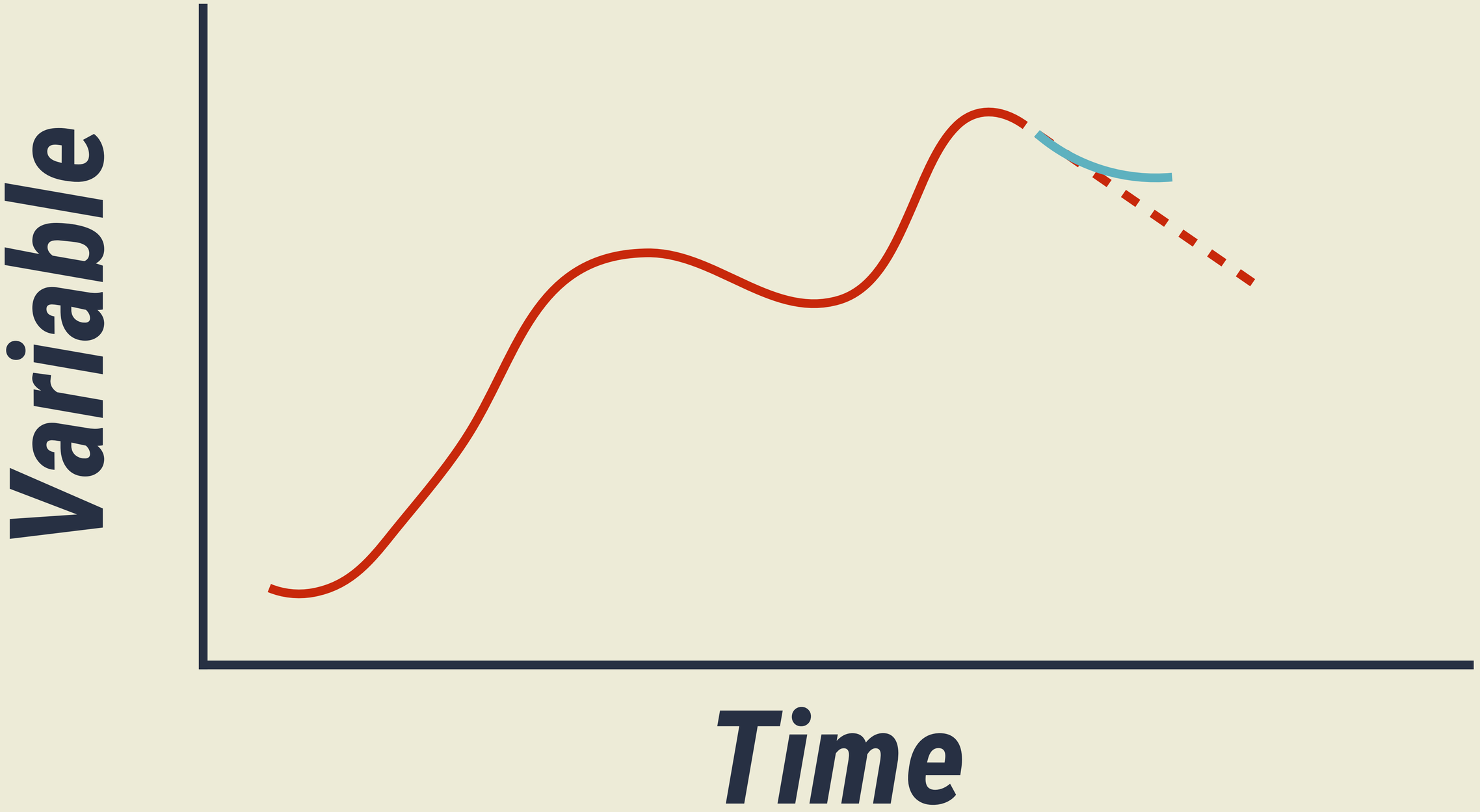
reside-ic.github.io

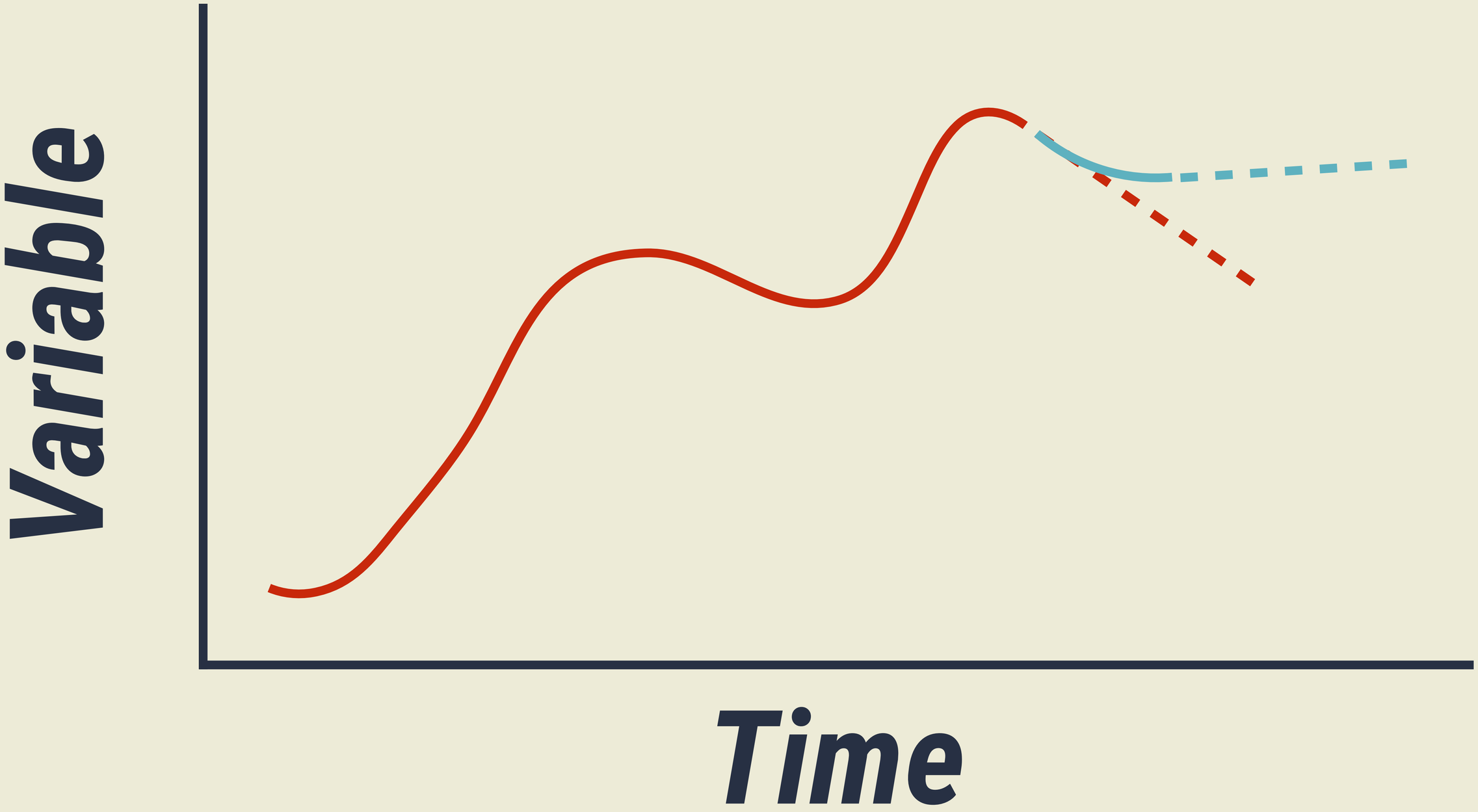


$$\frac{dS}{dt} \qquad \frac{dI}{dt} \qquad \frac{dR}{dt}$$


Variable
Time


Variable
Time


Variable
Time


Variable
Time

deSolve

```r
sir <- function(t, y, p) {
  S <- y[[1L]]
  I <- y[[2L]]
  R <- y[[3L]]
  beta <- p$beta
  gamma <- p$gamma

  list(c(-beta * S * I / N,
         beta * S * I / N - gamma * I,
         gamma * I))
}

deSolve::ode(t, y, sir)
```

```
sir <- function(t, y, p) {
  S <- y[[1L]]
  I <- y[[2L]]
  R <- y[[3L]]
  beta <- p$beta
  gamma <- p$gamma

  list(c(-beta * S * I / N,
         beta * S * I / N - gamma * I,
         gamma * I))
}
```

```
sir <- function(t, y, p) {
  S <- y[1:10]
  I <- y[11:20]
  R <- y[21:30]

  ...
}
```

```
sir <- function(t, y, p) {
  with(as.list(y, p), {
    list(c(-beta * S * I / N,
           beta * S * I / N - gamma * I,
           gamma * I))
  }
}
```

```
static double parms[2];

void initmod(void (* odeparms)(int *, double *)) {
  int N = 2;
  odeparms(&N, parms);
}

void sir(int *n, double *t, double *y, double *dydt, double *yout, int *ip) {
  double beta = parms[0];
  double gamma = parms[1];
  double S = y[0];
  double I = y[1];
  double R = y[2];

  double N = S + I + R;
  dydt[0] = -beta * S * I / N;
  dydt[1] <- beta * S * I / N - gamma * I;
  dydt[2] <- gamma * I;
}
```

```
static double parms[2];

void initmod(void (* odeparms)(int *, double *)) {
  int N = 2;
  odeparms(&N, parms);
}

void sir(int *n, double *t, double *y, double *dydt, double *yout, int *ip) {
  double beta = parms[0];
  double gamma = parms[1];
  double S = y[0];
  double I = y[1];
  double R = y[2];

  double N = S + I + R;
  dydt[0] = -beta * S * I / N;
  dydt[1] <- beta * S * I / N - gamma * I;
  dydt[2] <- gamma * I;
}
```

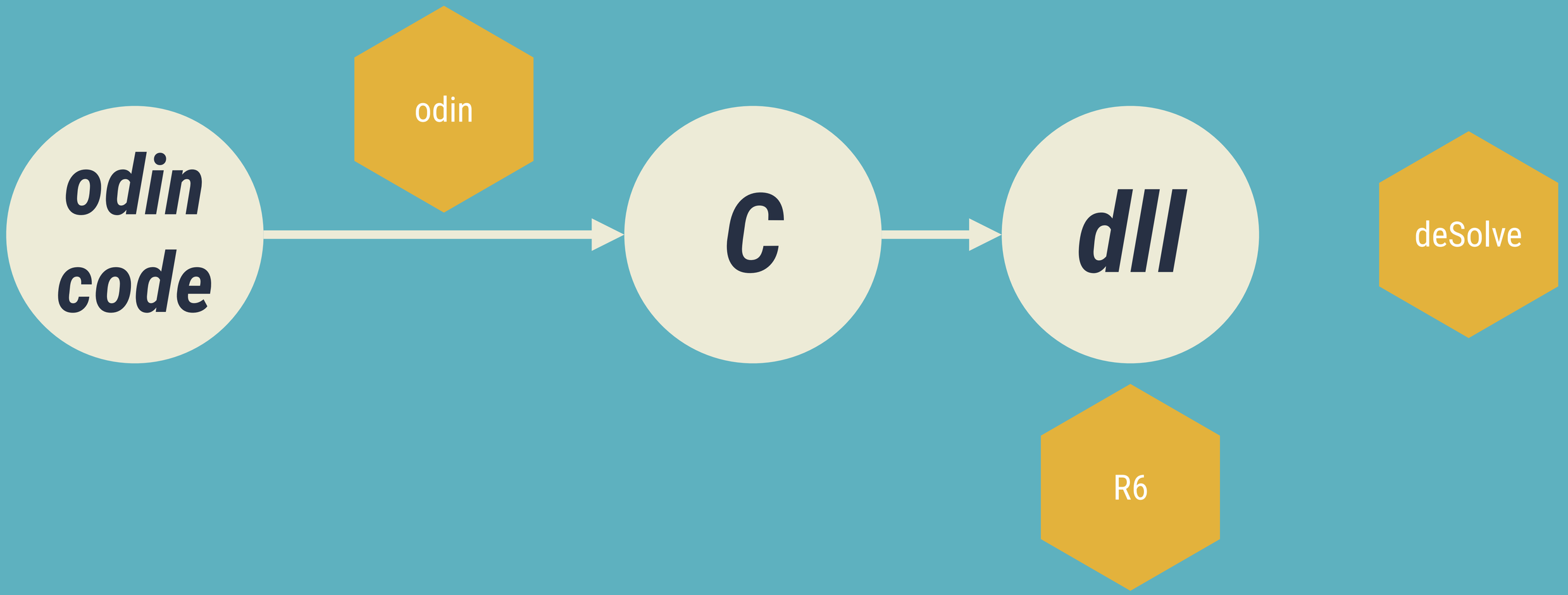
```
static double parms[2];

void initmod(void (* odeparms)(int *, double *)) {
  int N = 2;
  odeparms(&N, parms);
}

void sir(int *n, double *t, double *y, double *dydt, double *yout, int *ip) {
  double beta = parms[0];
  double gamma = parms[1];
  double S = y[0];
  double I = y[1];
  double R = y[2];

  double N = S + I + R;
  dydt[0] = -beta * S * I / N;
  dydt[1] <- beta * S * I / N - gamma * I;
  dydt[2] <- gamma * I;
}
```

expressive

fast


expressive
and
fast

odin


odin
code
odin
C
dll
deSolve
R6

```
sir <- odin::odin({
  deriv(S) <- -beta * S * I / N
  deriv(I) <- beta * S * I / N - gamma * I
  deriv(R) <- gamma * I

  initial(S) <- 1000
  initial(I) <- 1
  initial(R) <- 0

  N <- S + I + R

  beta <- user(0.2)
  gamma <- user(0.1)
})
```

```
sir <- odin::odin({
  ...
  beta  <- user(0.2)
  gamma <- user(0.1)
})

model <- sir(beta = 0.5,
             gamma = 0.9)
```

```
sir <- odin::odin({
  ...
  beta  <- user(0.2)
  gamma <- user(0.1)
})

model <- sir(beta = 0.5,
             gamma = 0.9)

t <- seq(0, 50, length.out = 101)
y <- model$run(t)
```

Code is data

```
deriv(I) <- beta * S * I / N - gamma * I
```

Code is data

```
deriv(I) <- beta * S * I / N - gamma * I

list(<-,
     list(deriv, I),
     list(-,
          list(*, beta, list(*, S, list(/, I, N)))
          list(*, gamma, I)))
```

Code is data

```
deriv(I) <- beta * S * I / N - gamma * I
```

```
(<- (deriv I) (- (* beta (* S (/ I N))) (* gamma I)))
```

Code is data

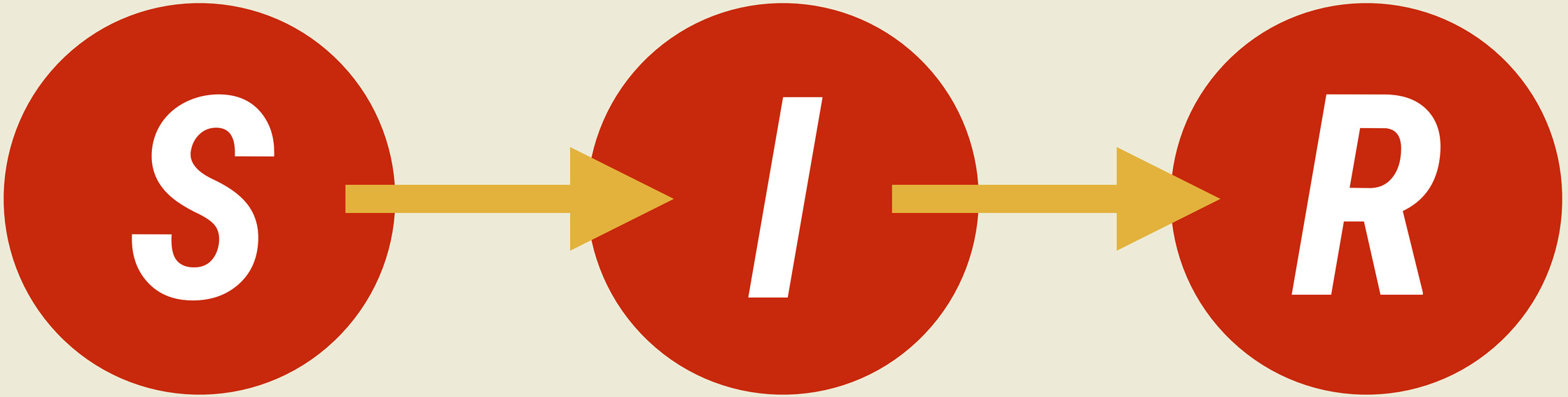
```
deriv(I) <- beta * S * I / N - gamma * I
```

```
(<- (deriv I) (- (* beta (* S (/ I N))) (* gamma I)))
```

```
dydt[1] = beta * S * I / (double) N - gamma * I;
```

Rewrite rules

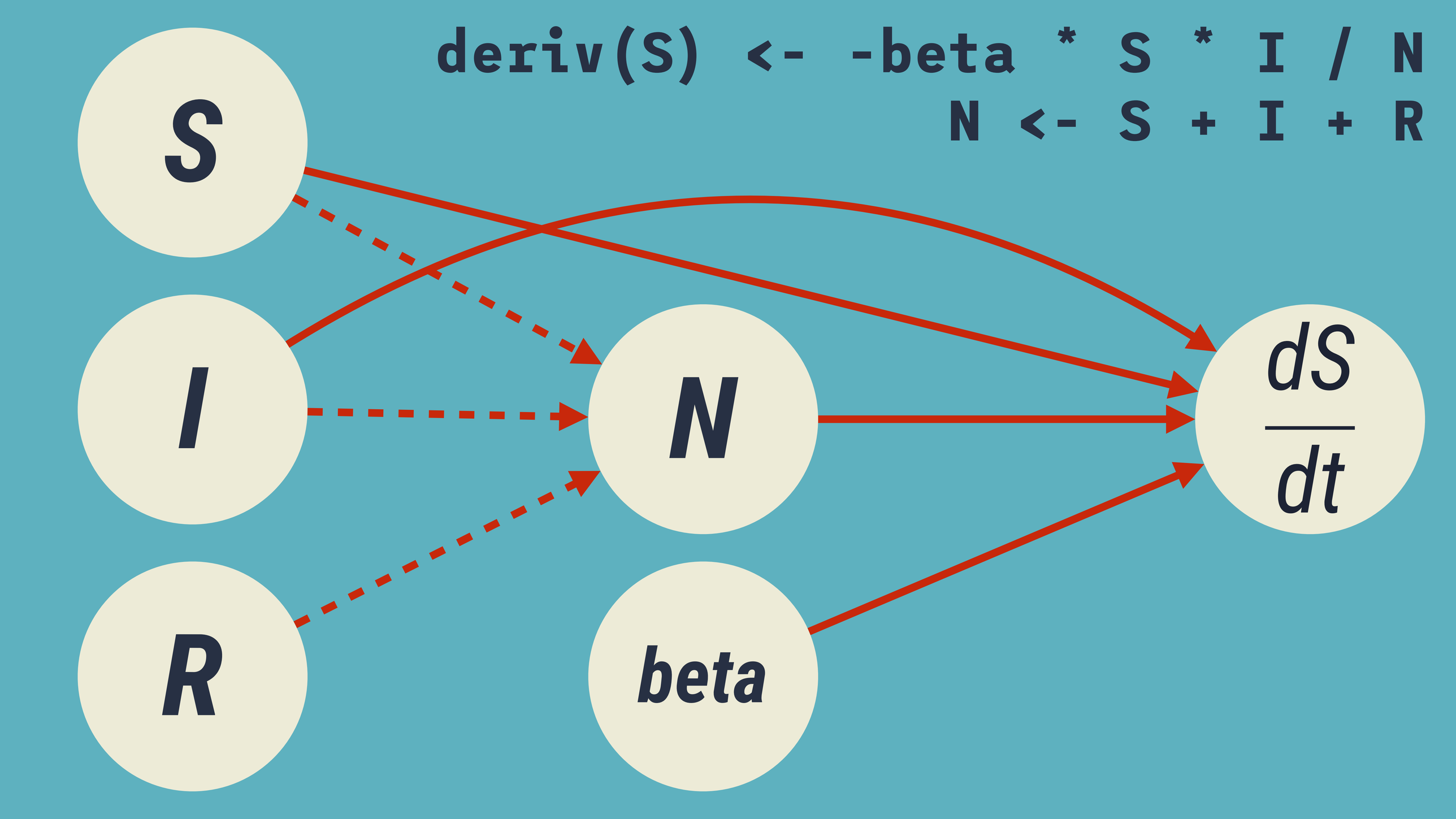
```
x^y => pow(x, y)
log(a, b) => log(a) / log(b)
min(a, b, c, d) => min(a, min(b, min(c, d)))
```

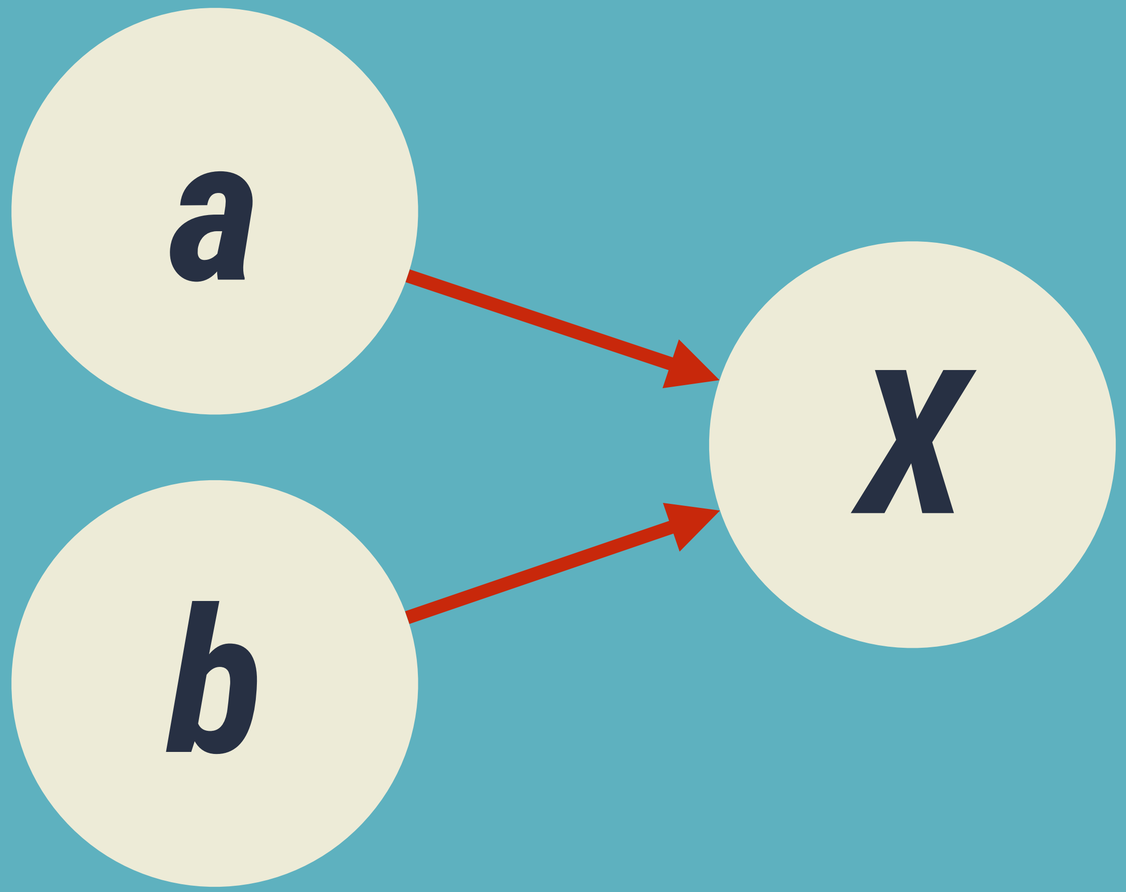


$$\frac{dS}{dt} \qquad \frac{dI}{dt} \qquad \frac{dR}{dt}$$

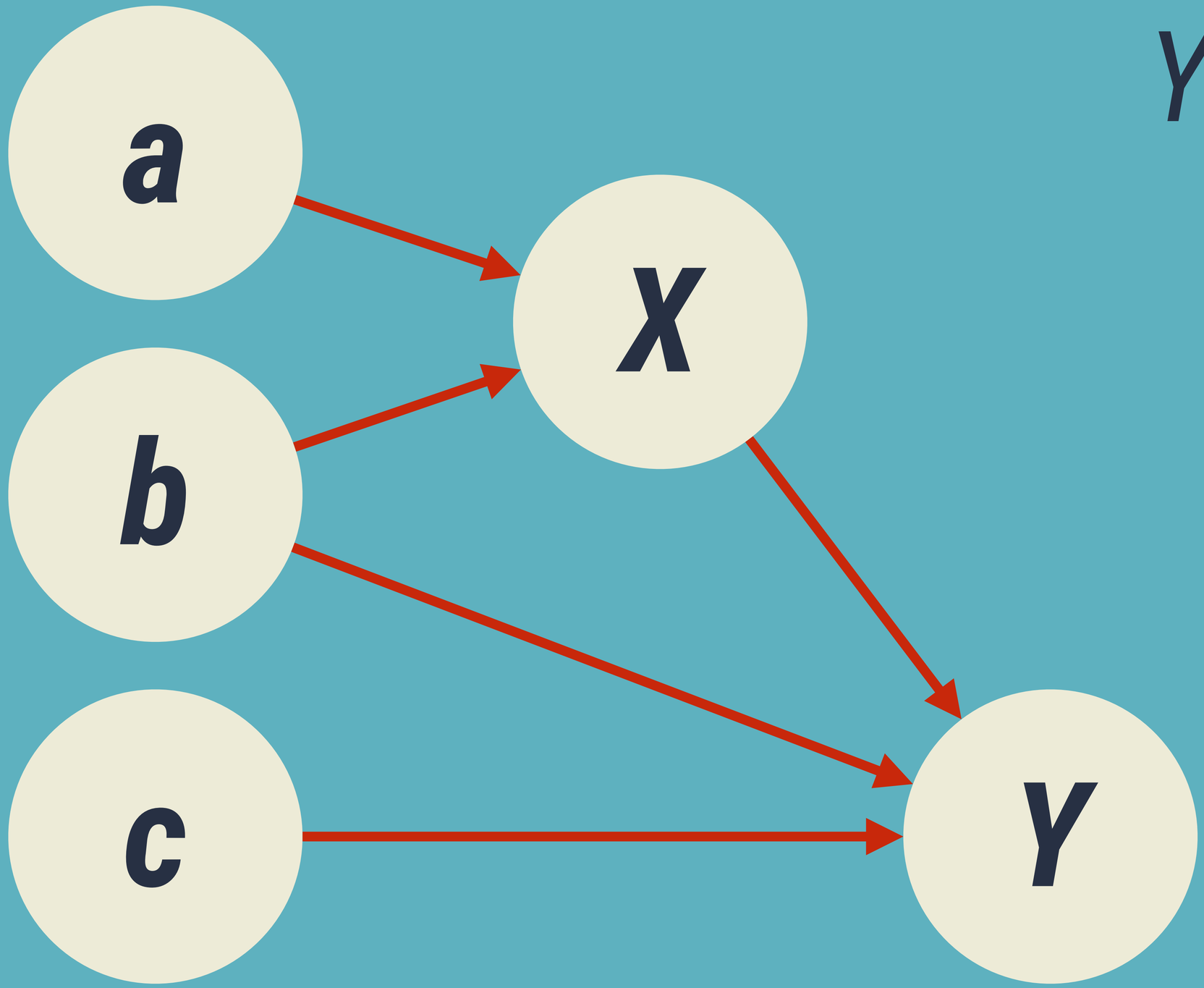
```
sir <- odin::odin({
  deriv(S) <- -beta * S * I / N
  deriv(I) <- beta * S * I / N - gamma * I
  deriv(R) <- gamma * I

  initial(S) <- 1000
  initial(I) <- 1
  initial(R) <- 0

  N <- S + I + R

  beta <- user(0.2)
  gamma <- user(0.1)
})
```

```
sir <- odin::odin({
  deriv(S) <- -beta * S * I / N
  deriv(I) <- beta * S * I / N - gamma * I
  deriv(R) <- gamma * I

  initial(S) <- 1000
  initial(I) <- 1
  initial(R) <- 0

  N <- S + I + R

  beta <- user(0.2)
  gamma <- user(0.1)
})
```

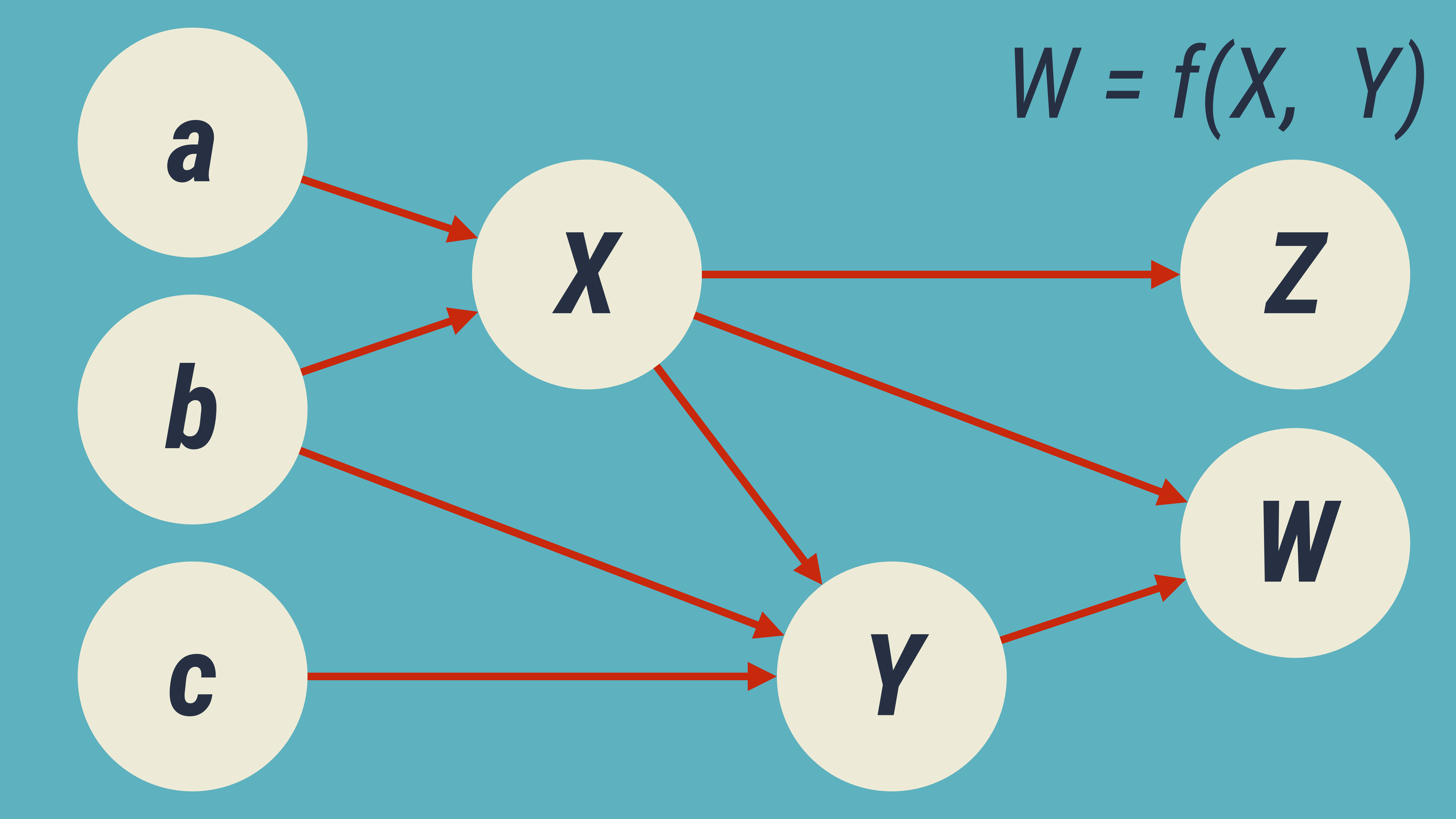

deriv(S) <- -beta * S * I / N
N <- S + I + R
S
I
R
N
beta
dS/dt

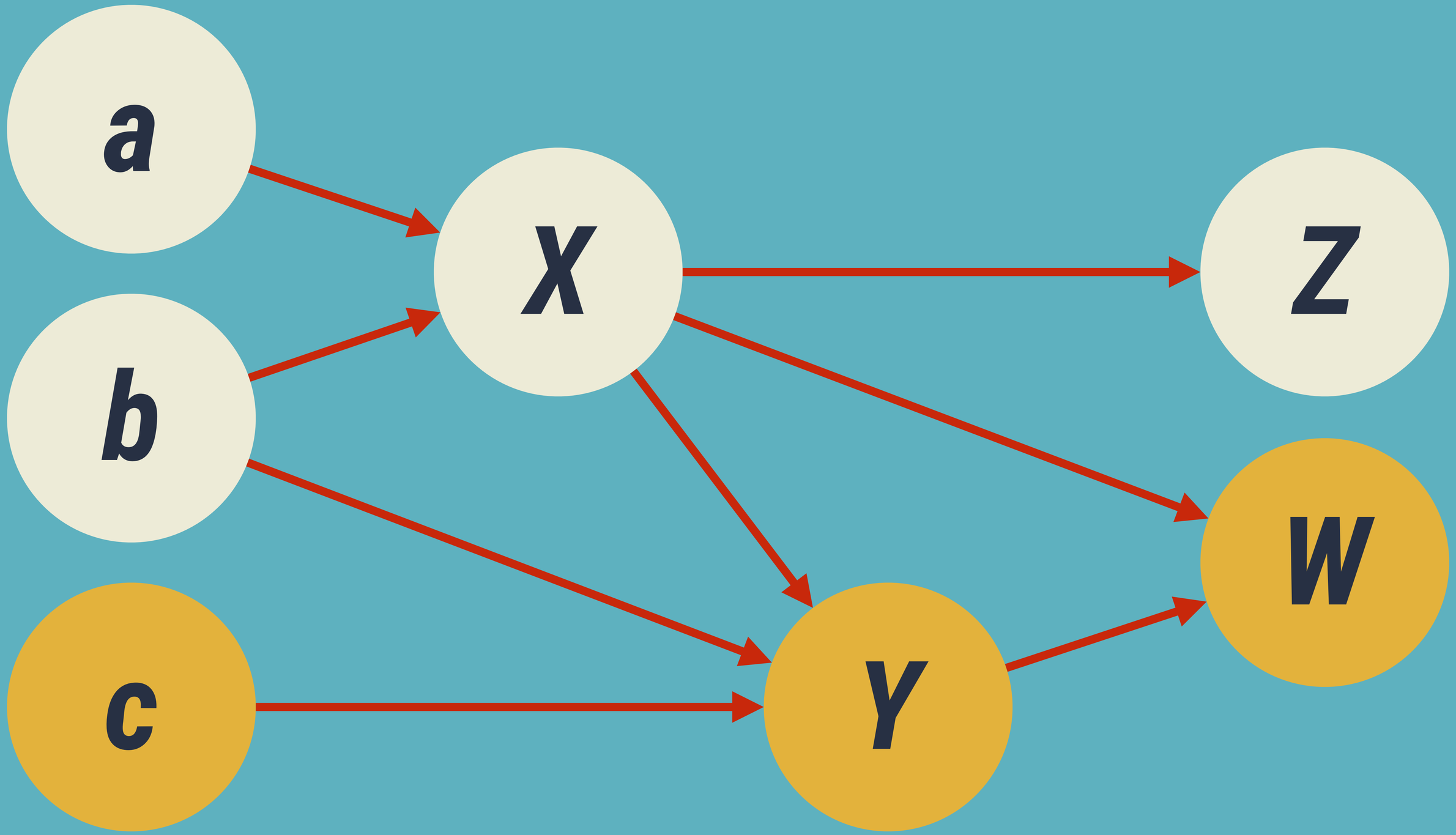


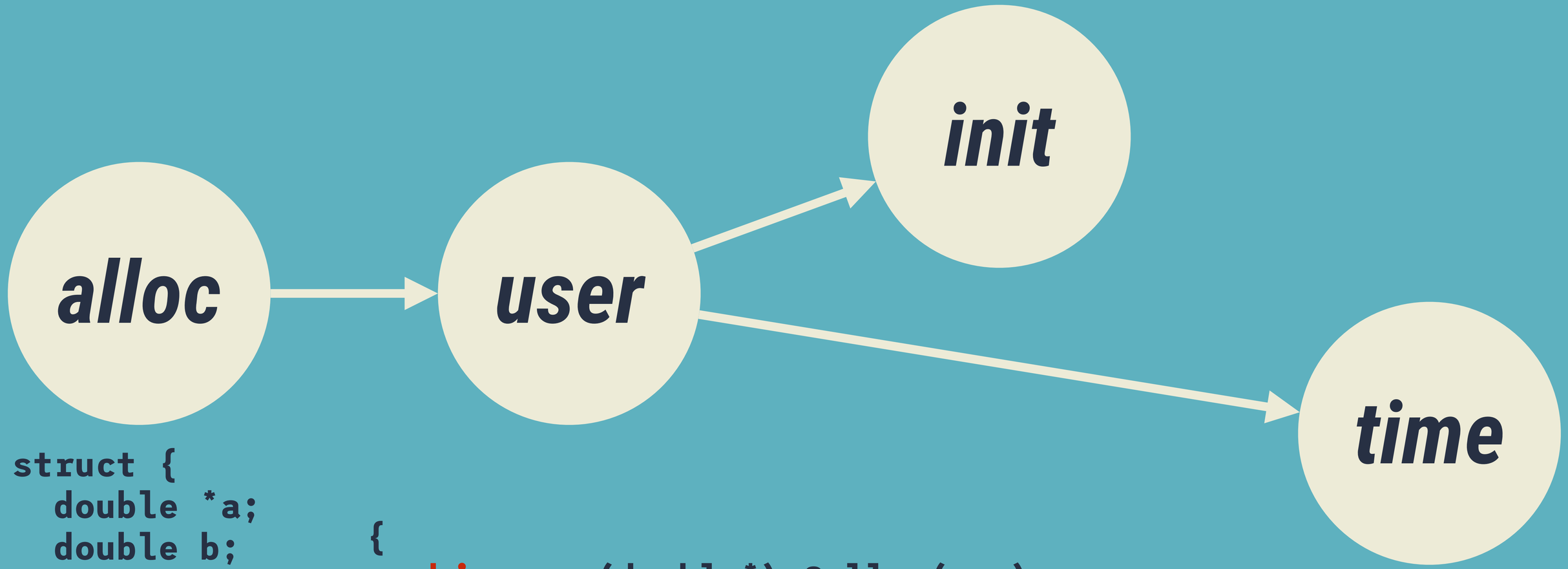
$$X = f(a, b)$$



$$Y = f(b, c, X)$$


W = f(X, Y)
a
b
c
X
Y
Z
W

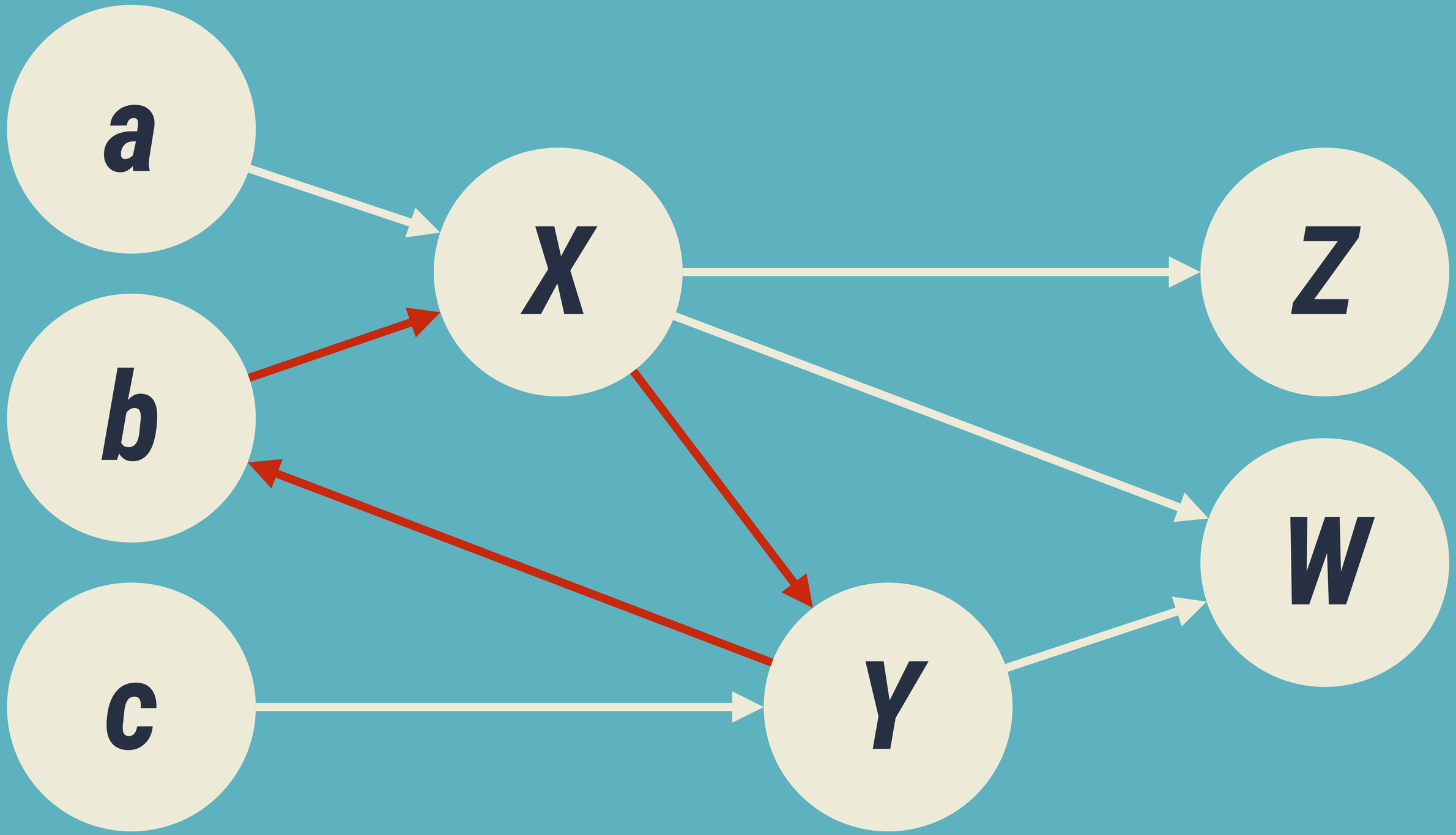

a
b
c
X
Y
Z
W

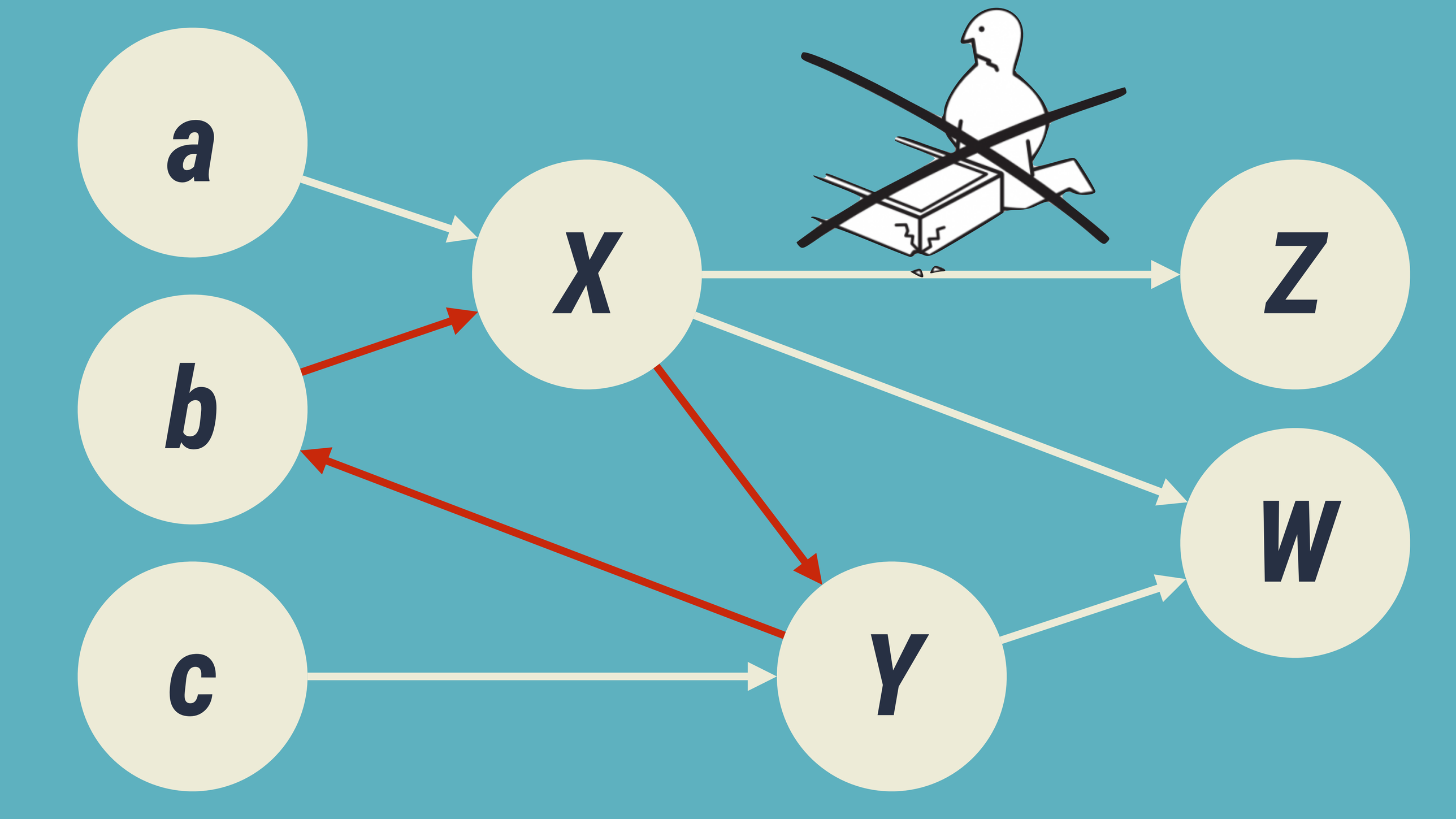



```
struct {
  double *a;
  double b;
  ...
};
```

```
{
  obj->a = (double*) Calloc(...);
  for (...) {
    obj->a[i] = ...
  }
  obj->b = 1;
};
```

```
{
   deriv[0] = obj->a[0] + c;
};
```


a
b
c
X
Y
Z
W


a
b
c
X
Y
Z
W

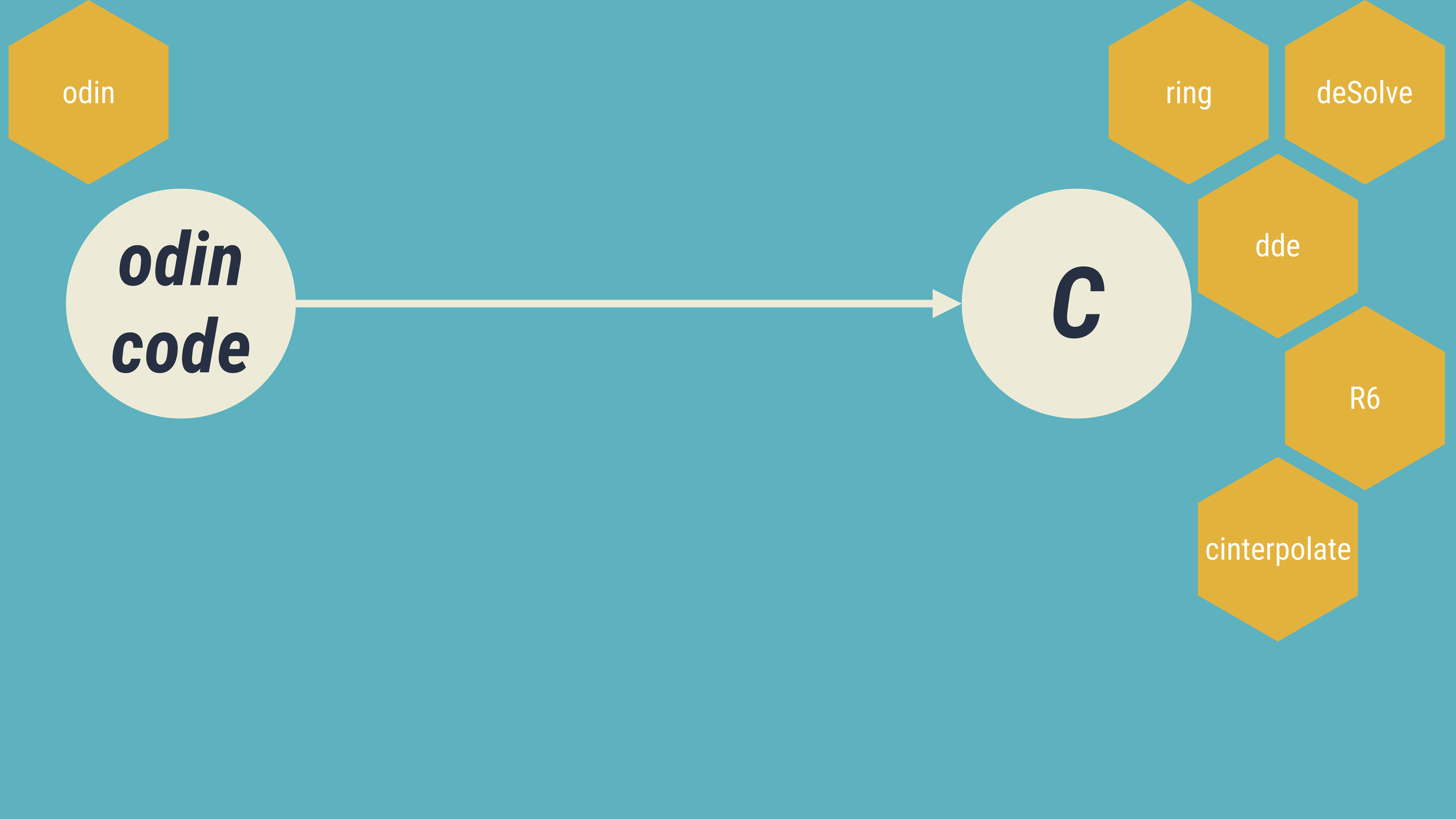
```
sir <- odin::odin({
  deriv(S) <- -new_infections
  deriv(E) <- new_infections - lag_infections
  deriv(I) <- lag_infections - gamma * I
  deriv(R) <- gamma * I

  latency <- 14
  new_infections <- beta * S * I / N
  lag_infections <- delay(new_infections, latency)

  initial(S) <- 1000
  initial(E) <- 0
  initial(I) <- 1
  initial(R) <- 0

  N <- S + I + R

  beta <- 0.2
  gamma <- 0.1
})
```

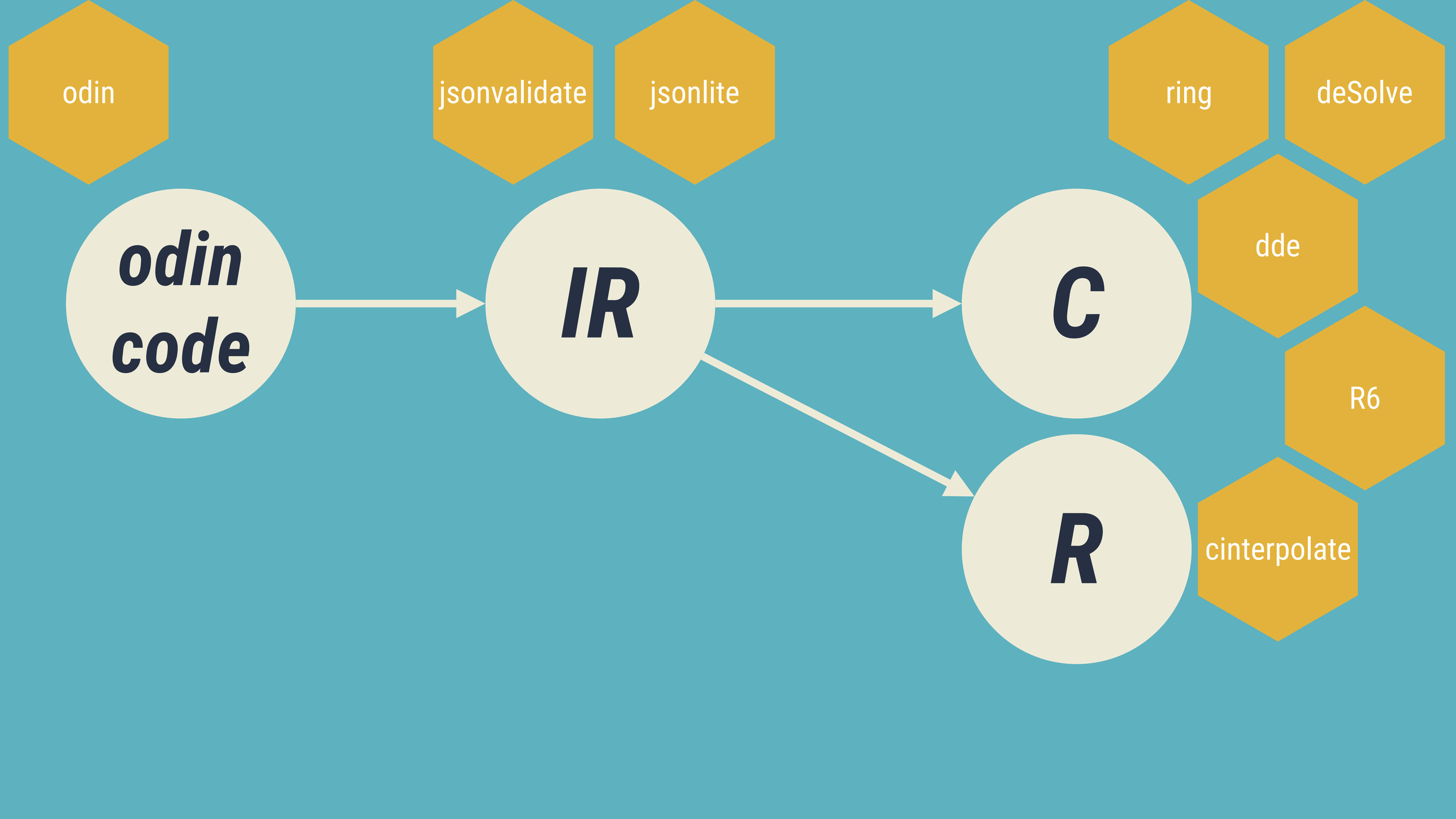
```
void odin_rhs(odin_internal* internal, double t, double * state, double * dstatedt, double * output) {
  double S = state[0];
  double I = state[2];
  double R = state[3];
  dstatedt[3] = internal->gamma * I;
  // delay block for lag_infections
  double lag_infections;
  {
    const double t_true = t;
    const double t = t_true - internal->latency;
    double S;
    double I;
    double R;
    if (t <= internal->initial_t) {
      S = internal->initial_S;
      I = internal->initial_I;
      R = internal->initial_R;
    } else {
      lagvalue(t, internal->odin_use_dde, internal->delay_index_lag_infections,
               internal->dim_delay_lag_infections, internal->delay_state_lag_infections);
      S = internal->delay_state_lag_infections[0];
      I = internal->delay_state_lag_infections[1];
      R = internal->delay_state_lag_infections[2];
    }
    double N = S + I + R;
    double new_infections = internal->beta * S * I / (double) N;
    lag_infections = new_infections;
  }
  double N = S + I + R;
  dstatedt[2] = lag_infections - internal->gamma * I;
  double new_infections = internal->beta * S * I / (double) N;
  dstatedt[1] = new_infections - lag_infections;
  dstatedt[0] = -(new_infections);
}
```

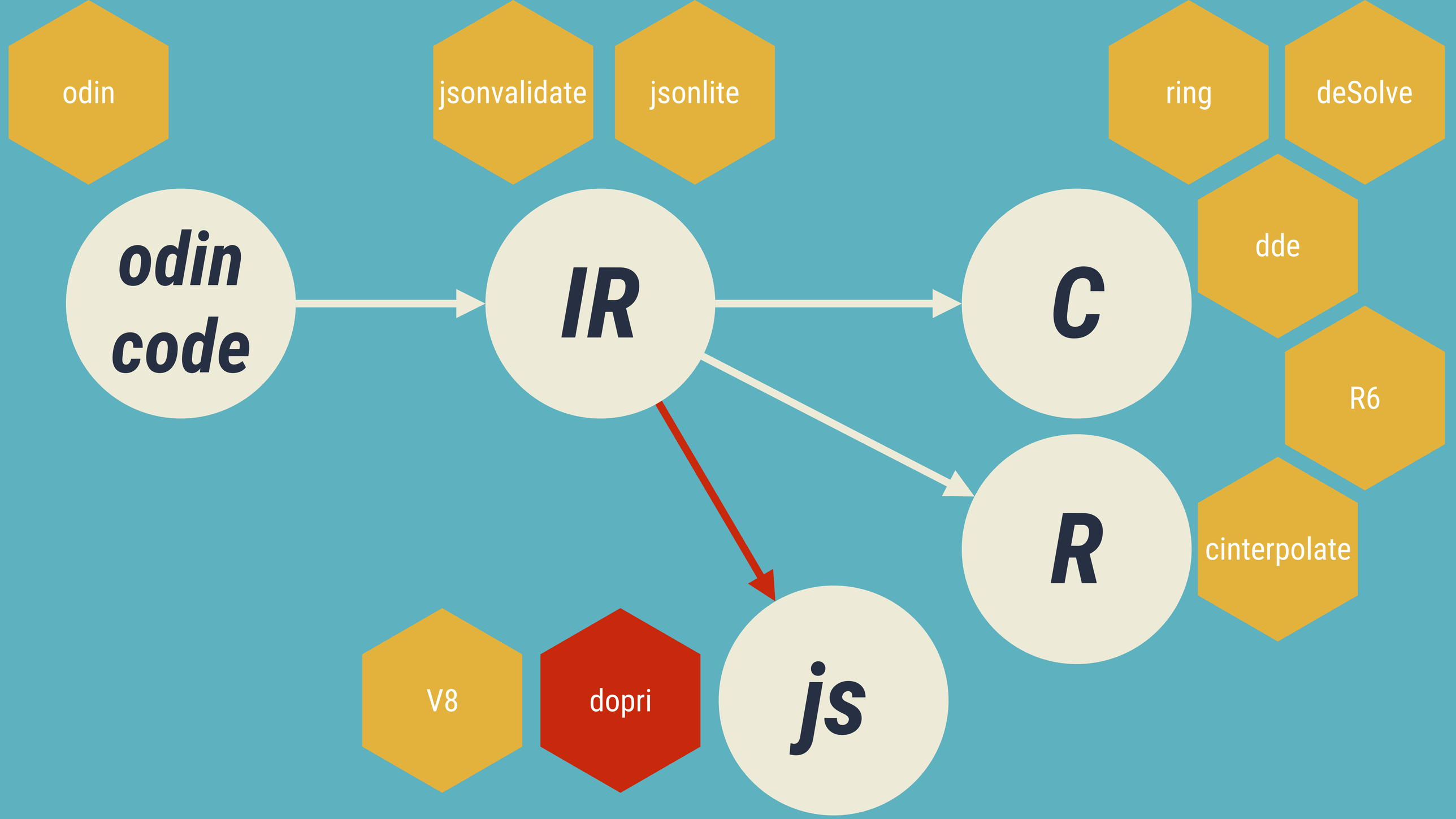
Automatic arrays

```
y[, ] = a[i] + b[j]
```

```
for (int i = 1; i <= internal->dim_y_1; ++i) {
  for (int j = 1; j <= internal->dim_y_2; ++j) {
    internal->y[i - 1 + internal->dim_y_1 * (j - 1)] =
      internal->a[i - 1] + internal->b[j - 1];
  }
}
```


odin
odin code
C
ring
deSolve
dde
R6
cinterpolate


odin
jsonvalidate
jsonlite
ring
deSolve
odin code
IR
C
dde
R6
R
cinterpolate


odin
jsonvalidate
jsonlite
ring
deSolve
odin code
IR
C
dde
R6
R
cinterpolate
V8
dopri
js

odin editor | Write and build | Interact | Help

Model name

model

Upload model file

Browse... No file selected

```
1  deriv(S) <- Births - b * S - beta * S * I / N + delta * R
2  deriv(I) <- beta * S * I / N - (b + sigma) * I
3  deriv(R) <- sigma * I - b * R - delta * R
4
5  initial(S) <- N - I0
6  initial(I) <- I0
7  initial(R) <- 0
8
9  Births <- N / 75
10 b <- 1 / 75
11 N <- 1e7
12 I0 <- user(1)
13 beta <- user(24)
14 sigma <- 12
15 delta <- 1 / 5
16
```

Compile | Reset | Validate | Save

Auto validate

Write odin code

Write code in this editor and press “Compile” and the odin model will be created! A new tab will be opened corresponding to the title of your model. If the tab exists then the previous version will be replaced without warning

Key bits to remember

```
deriv(X) <- ...
```

Specifies that `X` is a *variable* will change over time, but that we can only describe `X` in terms of its rates of change. Every `deriv()` call must be paired with an `initial()` call that describes the initial conditions

```
initial(X) <- ...
```

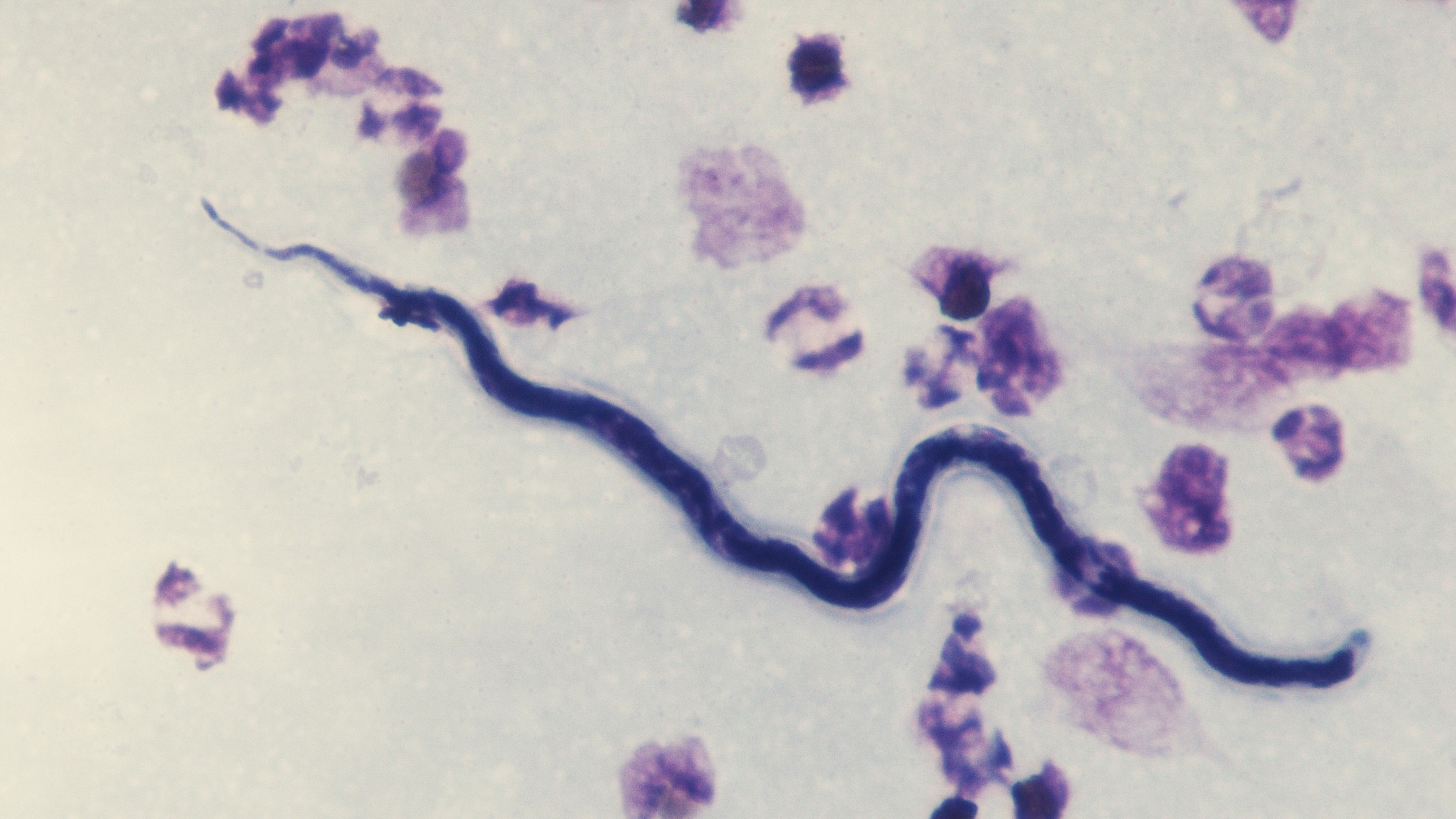
The code here looks like R but is not R. Not everything will work

Validating the code

Select `auto validate` to validate as you type. This may get annoying and/or slow.

odin

https://cran.r-project.org/package=odin
https://richfitz.github.io/odin




CLINIQUE DES AMIS
DE COTONOU
LG

Acknowledgements

MRC Centre for Global Infectious Disease Analysis
Bill and Melinda Gates Foundation
rOpenSci

Centre staff's work included here: Ettie Unwin, Charlie Whittaker, Ellie Sherrard-Smith, Lily Geidelberg

deSolve

Resources

https://cran.r-project.org/package=odin
https://mrc-ide.github.io/odin

richfitz

rgfitzjohn

We're hiring!

Develop a simulation model of malaria transmission using R & C++

Work with our team of RSEs and a world-leading team of malaria researchers

reside-ic.github.io

```c
#include <R.h>
#include <Rmath.h>
#include <Rinternals.h>
#include <stdbool.h>
#include <R_ext/Rdynload.h>
typedef struct odin_internal {
  double beta;
  double gamma;
  double initial_I;
  double initial_R;
  double initial_S;
} odin_internal;
odin_internal* odin_get_internal(SEXP internal_p, int closed_error);
static void odin_finalise(SEXP internal_p);
SEXP odin_create(SEXP user);
void odin_initmod_desolve(void(* odeparms) (int *, double *));
SEXP odin_contents(SEXP internal_p);
SEXP odin_set_user(SEXP internal_p, SEXP user);
SEXP odin_metadata(SEXP internal_p);
SEXP odin_initial_conditions(SEXP internal_p, SEXP t_ptr);
void odin_rhs(odin_internal* internal, double t, double * state, double * dstatedt, double * output);
void odin_rhs_dde(size_t neq, double t, double * state, double * dstatedt, void * internal);
void odin_rhs_desolve(int * neq, double * t, double * state, double * dstatedt, double * output, int * np);
SEXP odin_rhs_r(SEXP internal_p, SEXP t, SEXP state);
double user_get_scalar_double(SEXP user, const char *name,
                              double default_value, double min, double max);
int user_get_scalar_int(SEXP user, const char *name,
                        int default_value, double min, double max);
void user_check_values_double(double * value, size_t len,
                              double min, double max, const char *name);
void user_check_values_int(int * value, size_t len,
                           double min, double max, const char *name);
void user_check_values(SEXP value, double min, double max,
                       const char *name);
SEXP user_list_element(SEXP list, const char *name);
odin_internal* odin_get_internal(SEXP internal_p, int closed_error) {
  odin_internal *internal = NULL;
  if (TYPEOF(internal_p) != EXTPTRSXP) {
    Rf_error("Expected an external pointer");
  }
  internal = (odin_internal*) R_ExternalPtrAddr(internal_p);
  if (!internal && closed_error) {
    Rf_error("Pointer has been invalidated");
  }
  return internal;
}
void odin_finalise(SEXP internal_p) {
  odin_internal *internal = odin_get_internal(internal_p, 0);
  if (internal_p) {
    Free(internal);
    R_ClearExternalPtr(internal_p);
  }
}
SEXP odin_create(SEXP user) {
  odin_internal *internal = (odin_internal*) Calloc(1, odin_internal);
  internal->initial_I = 1;
  internal->initial_R = 0;
  internal->initial_S = 1000;
  internal->beta = 0.20000000000000001;
  internal->gamma = 0.10000000000000001;
  SEXP ptr = PROTECT(R_MakeExternalPtr(internal, R_NilValue, R_NilValue));
  R_RegisterCFinalizer(ptr, odin_finalise);
  UNPROTECT(1);
  return ptr;
}
static odin_internal *odin_internal_ds;
void odin_initmod_desolve(void(* odeparms) (int *, double *)) {
  static DL_FUNC get_desolve_gparms = NULL;
  if (get_desolve_gparms == NULL) {
    get_desolve_gparms =
      R_GetCCallable("deSolve", "get_deSolve_gparms");
  }
  odin_internal_ds = odin_get_internal(get_desolve_gparms(), 1);
}
SEXP odin_contents(SEXP internal_p) {
  odin_internal *internal = odin_get_internal(internal_p, 1);
  SEXP contents = PROTECT(allocVector(VECSXP, 5));
  SET_VECTOR_ELT(contents, 0, ScalarReal(internal->beta));
  SET_VECTOR_ELT(contents, 1, ScalarReal(internal->gamma));
  SET_VECTOR_ELT(contents, 2, ScalarReal(internal->initial_I));
  SET_VECTOR_ELT(contents, 3, ScalarReal(internal->initial_R));
  SET_VECTOR_ELT(contents, 4, ScalarReal(internal->initial_S));
  SEXP nms = PROTECT(allocVector(STRSXP, 5));
  SET_STRING_ELT(nms, 0, mkChar("beta"));
  SET_STRING_ELT(nms, 1, mkChar("gamma"));
  SET_STRING_ELT(nms, 2, mkChar("initial_I"));
  SET_STRING_ELT(nms, 3, mkChar("initial_R"));
  SET_STRING_ELT(nms, 4, mkChar("initial_S"));
  setAttrib(contents, R_NamesSymbol, nms);
  UNPROTECT(2);
  return contents;
}
SEXP odin_set_user(SEXP internal_p, SEXP user) {
  odin_internal *internal = odin_get_internal(internal_p, 1);
  internal->beta = user_get_scalar_double(user, "beta", internal->beta, NA_REAL, NA_REAL);
  internal->gamma = user_get_scalar_double(user, "gamma", internal->gamma, NA_REAL, NA_REAL);
  return R_NilValue;
}
SEXP odin_metadata(SEXP internal_p) {
  odin_internal *internal = odin_get_internal(internal_p, 1);
  SEXP ret = PROTECT(allocVector(VECSXP, 4));
  SEXP nms = PROTECT(allocVector(STRSXP, 4));
  SET_STRING_ELT(nms, 0, mkChar("variable_order"));
  SET_STRING_ELT(nms, 1, mkChar("output_order"));
  SET_STRING_ELT(nms, 2, mkChar("n_out"));
  SET_STRING_ELT(nms, 3, mkChar("interpolate_t"));
  setAttrib(ret, R_NamesSymbol, nms);
  SEXP variable_length = PROTECT(allocVector(VECSXP, 3));
  SEXP variable_names = PROTECT(allocVector(STRSXP, 3));
  setAttrib(variable_length, R_NamesSymbol, variable_names);
  SET_VECTOR_ELT(variable_length, 0, R_NilValue);
  SET_VECTOR_ELT(variable_length, 1, R_NilValue);
  SET_VECTOR_ELT(variable_length, 2, R_NilValue);
  SET_STRING_ELT(variable_names, 0, mkChar("S"));
  SET_STRING_ELT(variable_names, 1, mkChar("I"));
  SET_STRING_ELT(variable_names, 2, mkChar("R"));
  SET_VECTOR_ELT(ret, 0, variable_length);
  UNPROTECT(2);
  SET_VECTOR_ELT(ret, 1, R_NilValue);
  SET_VECTOR_ELT(ret, 2, ScalarInteger(0));
  UNPROTECT(2);
  return ret;
}
SEXP odin_initial_conditions(SEXP internal_p, SEXP t_ptr) {
  odin_internal *internal = odin_get_internal(internal_p, 1);
  SEXP r_state = PROTECT(allocVector(REALSXP, 3));
  double * state = REAL(r_state);
  state[0] = internal->initial_S;
  state[1] = internal->initial_I;
  state[2] = internal->initial_R;
  UNPROTECT(1);
  return r_state;
}
void odin_rhs(odin_internal* internal, double t, double * state, double * dstatedt, double * output) {
  double S = state[0];
  double I = state[1];
  double R = state[2];
  dstatedt[2] = internal->gamma * I;
  double N = S + I + R;
  dstatedt[1] = internal->beta * S * I / (double) N - internal->gamma * I;
  dstatedt[0] = -(internal->beta) * S * I / (double) N;
}
void odin_rhs_dde(size_t neq, double t, double * state, double * dstatedt, void * internal) {
  odin_rhs((odin_internal*)internal, t, state, dstatedt, NULL);
}
void odin_rhs_desolve(int * neq, double * t, double * state, double * dstatedt, double * output, int * np) {
  odin_rhs(odin_internal_ds, *t, state, dstatedt, output);
}
SEXP odin_rhs_r(SEXP internal_p, SEXP t, SEXP state) {
  SEXP dstatedt = PROTECT(allocVector(REALSXP, LENGTH(state)));
  odin_internal *internal = odin_get_internal(internal_p, 1);
  double *output = NULL;
  odin_rhs(internal, REAL(t)[0], REAL(state), REAL(dstatedt), output);
  UNPROTECT(1);
  return dstatedt;
}
double user_get_scalar_double(SEXP user, const char *name,
                              double default_value, double min, double max) {
  double ret = default_value;
  SEXP el = user_list_element(user, name);
  if (el != R_NilValue) {
    if (length(el) != 1) {
      Rf_error("Expected a scalar numeric for '%s'", name);
    }
    if (TYPEOF(el) == REALSXP) {
      ret = REAL(el)[0];
    } else if (TYPEOF(el) == INTSXP) {
      ret = INTEGER(el)[0];
    } else {
      Rf_error("Expected a numeric value for %s", name);
    }
  }
  if (ISNA(ret)) {
    Rf_error("Expected a value for '%s'", name);
  }
  user_check_values_double(&ret, 1, min, max, name);
  return ret;
}
int user_get_scalar_int(SEXP user, const char *name,
                        int default_value, double min, double max) {
  int ret = default_value;
  SEXP el = user_list_element(user, name);
  if (el != R_NilValue) {
    if (length(el) != 1) {
      Rf_error("Expected scalar integer for %d", name);
    }
    if (TYPEOF(el) == REALSXP) {
      double tmp = REAL(el)[0];
      if (fabs(tmp - round(tmp)) > 2e-8) {
        Rf_error("Expected '%s' to be integer-like", name);
      }
    }
    ret = INTEGER(coerceVector(el, INTSXP))[0];
  }
  if (ret == NA_INTEGER) {
    Rf_error("Expected a value for '%s'", name);
  }
  user_check_values_int(&ret, 1, min, max, name);
  return ret;
}
void user_check_values_double(double * value, size_t len,
                              double min, double max, const char *name) {
  for (size_t i = 0; i < len; ++i) {
    if (ISNA(value[i])) {
      Rf_error("'%s' must not contain any NA values", name);
    }
  }
  if (min != NA_REAL) {
    for (size_t i = 0; i < len; ++i) {
      if (value[i] < min) {
        Rf_error("Expected '%s' to be at least %g", name, min);
      }
    }
  }
  if (max != NA_REAL) {
    for (size_t i = 0; i < len; ++i) {
      if (value[i] > max) {
        Rf_error("Expected '%s' to be at most %g", name, max);
      }
    }
  }
}
void user_check_values_int(int * value, size_t len,
                           double min, double max, const char *name) {
  for (size_t i = 0; i < len; ++i) {
    if (ISNA(value[i])) {
      Rf_error("'%s' must not contain any NA values", name);
    }
  }
  if (min != NA_REAL) {
    for (size_t i = 0; i < len; ++i) {
      if (value[i] < min) {
        Rf_error("Expected '%s' to be at least %g", name, min);
      }
    }
  }
  if (max != NA_REAL) {
    for (size_t i = 0; i < len; ++i) {
      if (value[i] > max) {
        Rf_error("Expected '%s' to be at most %g", name, max);
      }
    }
  }
}
void user_check_values(SEXP value, double min, double max,
                       const char *name) {
  size_t len = (size_t)length(value);
  if (TYPEOF(value) == INTSXP) {
    user_check_values_int(INTEGER(value), len, min, max, name);
  } else {
    user_check_values_double(REAL(value), len, min, max, name);
  }
}
SEXP user_list_element(SEXP list, const char *name) {
  SEXP ret = R_NilValue, names = getAttrib(list, R_NamesSymbol);
  for (int i = 0; i < length(list); ++i) {
    if(strcmp(CHAR(STRING_ELT(names, i)), name) == 0) {
      ret = VECTOR_ELT(list, i);
      break;
    }
  }
  return ret;
}
```