

R's weirdnesses are fun & useful

Rich FitzJohn

 richfitz

R is a really weird language

There are people who think of it as a statistical package - a free version of stata perhaps.

Like stata, R includes lots of useful things

***Statistics programs
generally include
distributions***

R includes...

***Statistics programs
generally include
statistical tests***

***Statistics programs
generally include
plotting***

***Statistics programs
don't often include
blog generators***

`cran.r-project.org/package=blogdown`

R also has some weird things available in packages

***Statistics programs
don't often include
webservers***

`cran.r-project.org/package=httpuv`

***Statistics programs
don't often include
minecraft clients***

`github.com/ropenscilabs/miner`

Statistics programs don't often include metaprogramming

Metaprogramming is where a program reads, generates, analyses or transforms a program

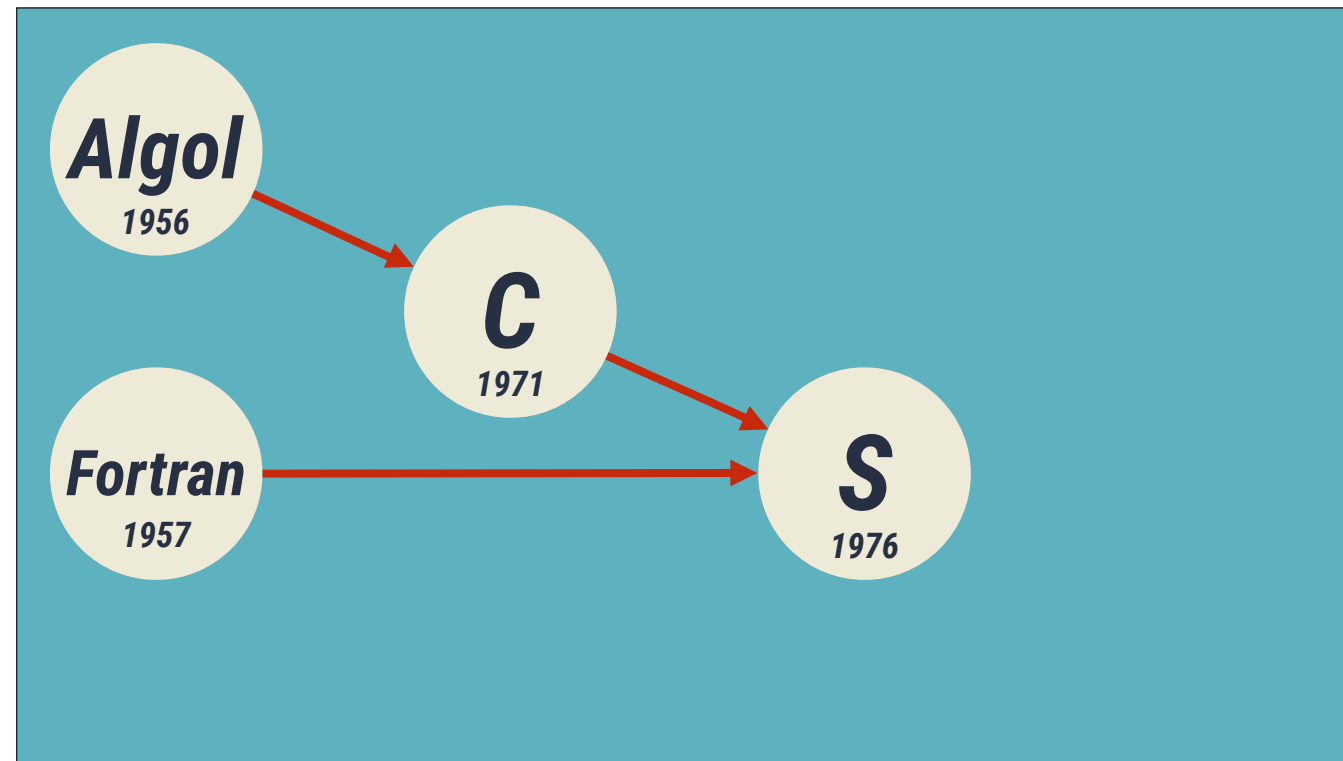
Unlike the other, newer, weirdnesses this one has been here from the beginning



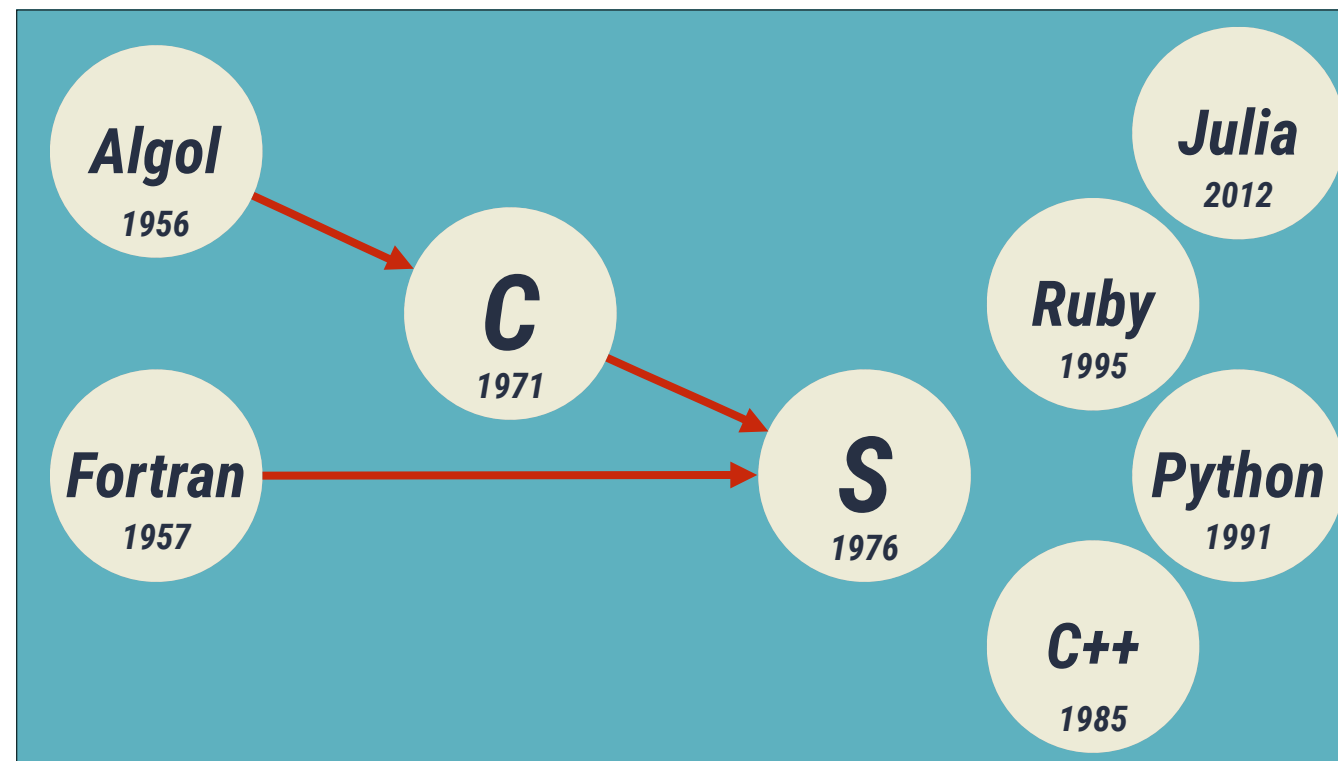
At first metaprogramming seems like a really bizarre thing to do in any language

And it's very unexpected in a statistical package

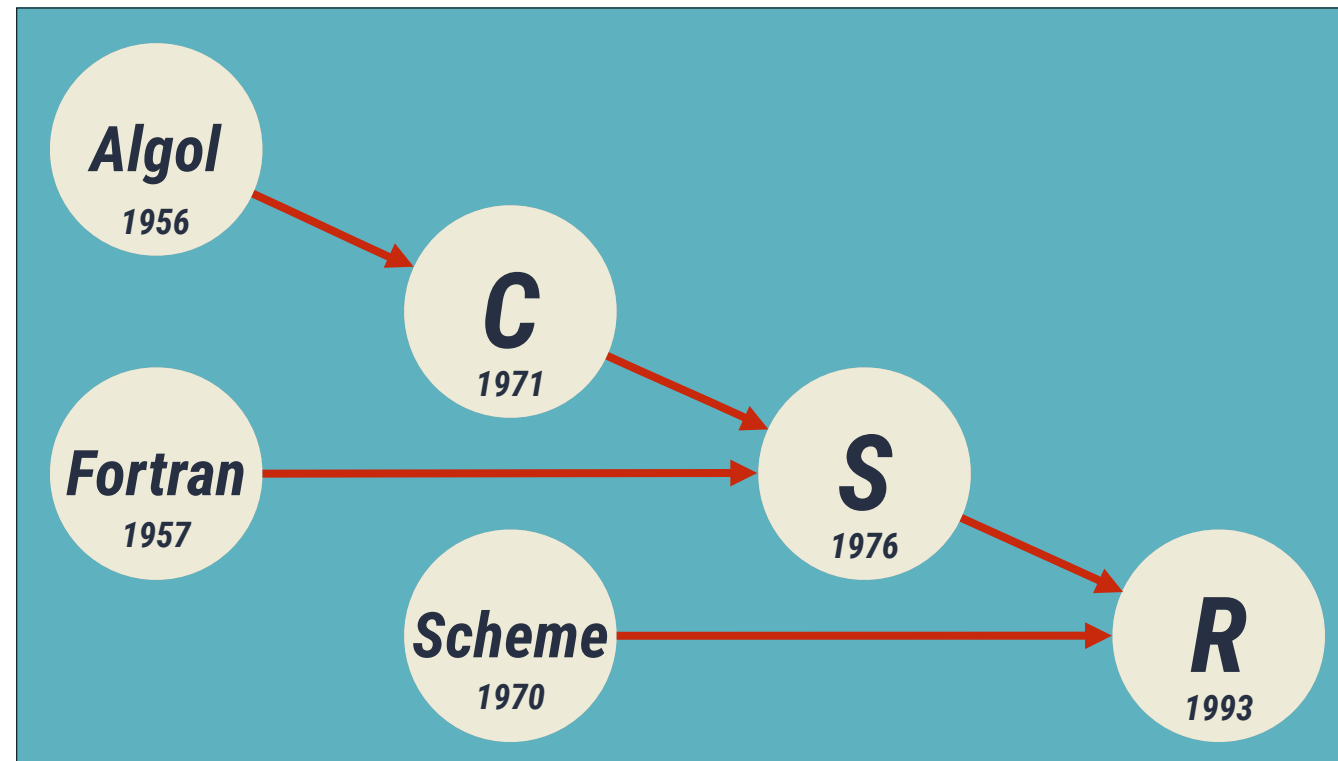
metaprogramming makes much more sense when you know R's history



R has a strange history. It turns up as derivative of **S** - and **S** is ancient, coming out of Bell Labs in 1976.



This is older than **C++**, **Python**, **Ruby** and much older than **Julia**



R was developed as a new implementation of **S** in 1993 by Ross Ihaka and Robert Gentleman. They were heavily influenced by **Scheme** - a language popular in computing science for decades, and which has lots of interesting ideas despite being very small. The one that really turns up is that data and code are the same sort of thing (**homiconicity**)

Generally R looks a lot like C or Fortran (procedural, do this, then that) but sometimes the weird scheme bits shine through

R's weirdnesses are fun & useful

Rich FitzJohn

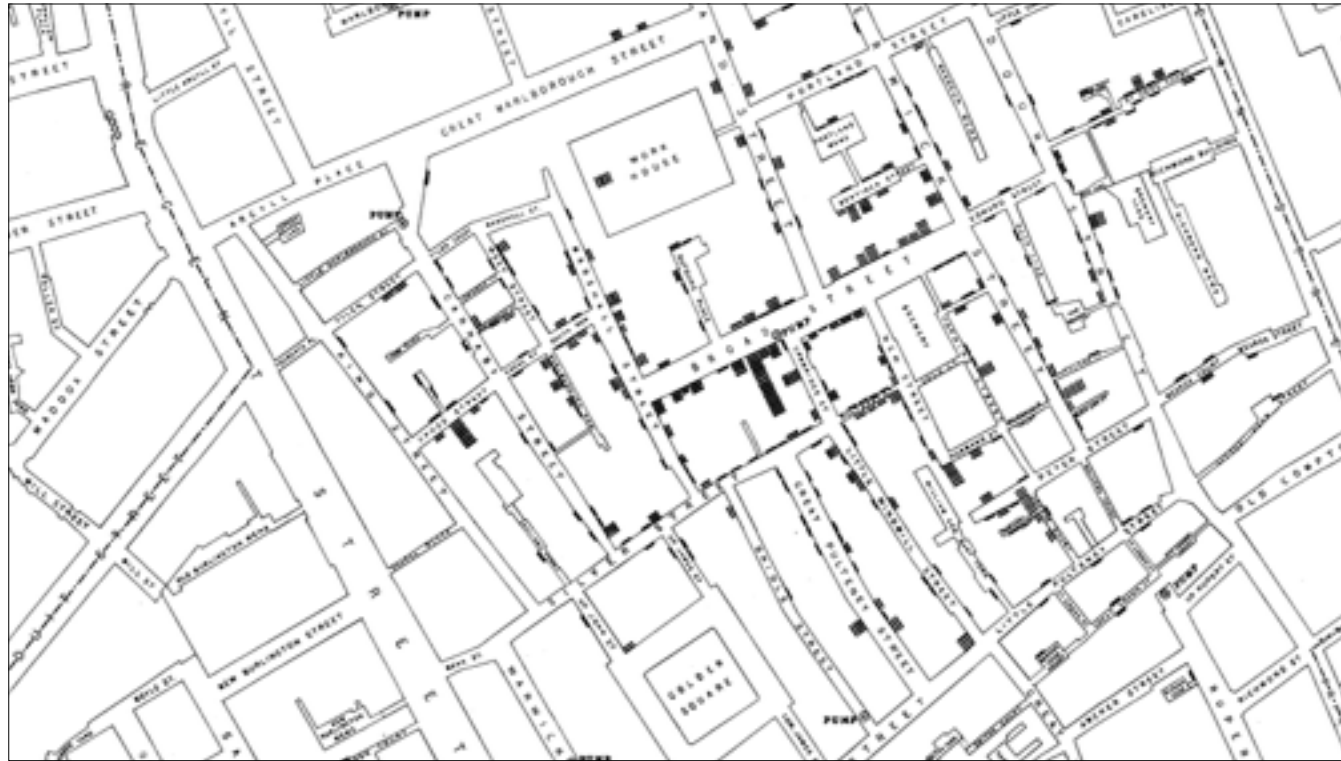


Today I want to talk about how metaprogramming is fun and useful

Using R since 1999, second language (after Python) but my workhorse

I no longer do any data analysis

I build **infrastructure** and this talk discusses some of it



For the last 2.5 years I have worked as a research software engineer in the **Department of Infectious Disease Epidemiology** at Imperial College London

Epidemiological modelling has a long history, but now we use lots of R!

Encryption

Differential equations

Docker

Objectives for this talk

1. R has some strange features that make it surprisingly powerful. These should be used with care
2. Three packages that do interesting things
3. Three fields that you may not have encountered with R

Don't try to learn everything - It's going to be very light touch and not very deep

If one section seems uninteresting to you, just wait 10 minutes and the next one will be totally different

Take home one new package, idea, or way of thinking about R

Encryption

Why encrypt things from R?

Encrypt and save csv

```
write.csv(mydata, "secret.csv")
```

We want to encrypt a csv file.

But the encryption tools won't work with this function.

Encrypt and save csv

```
tmp <- tempfile()  
write.csv(mydata, tmp)
```

So we first must write it out in plain text

Encrypt and save csv

```
tmp <- tempfile()
write.csv(mydata, tmp)
bytes <- readBin(tmp, ...)
enc <- sodium::data_encrypt(bytes, key)
```

Then read it back up and encrypt that data

Encrypt and save csv

```
tmp <- tempfile()
write.csv(mydata, tmp)
bytes <- readBin(tmp, ...)
enc <- sodium::data_encrypt(bytes, key)
enc
[1] a7 8e 31 99 3b 7b ac 58 4e 35 37 79
[13] 53 10 4c fe 5e 78 de 4e 4d 25 77 26
```

This involves working with **raw vectors** which you don't generally see unless you go out of your way

Encrypt and save csv

```
tmp <- tempfile()
write.csv(mydata, tmp)
bytes <- readBin(tmp, ...)
enc <- sodium::data_encrypt(bytes, key)
writeBin(enc, "secret.csv")
file.remove(tmp)
```

Decrypt and read csv

```
enc <- readBin("secret.csv", ...)  
bytes <- sodium::data_decrypt(enc, key)  
tmp <- tempfile()  
writeBin(bytes, tmp)  
mydata <- read.csv(tmp)  
file.remove(tmp)
```

A simpler interface

```
cyphr::encrypt(write.csv(mydata, "secret.csv"), key)
```

```
mydata <- cyphr::decrypt(read.csv("secret.csv"), key)
```

A simpler interface

```
cyphr::encrypt(write.csv(mydata, "secret.csv"), key)  
# Write mydata to temp file using write.csv  
# Encrypt temp file contents to "secret.csv" using key  
# Delete temp file
```


A simpler interface

```
cyphr::encrypt(write.csv(mydata, "secret.csv"), key)
# Decide on a temporary file tmp
# Detect filename is second argument "secret.csv"
# Rewrite expression as write.csv(mydata, tmp)
# Evaluate new expression (in same environment as old)
# Read in tmp as bytes
# Encrypt the contents with cyphr::encrypt(bytes, key)
# Save encrypted data as secret.csv
# Delete the temporary file tmp
```

Expressions are data

```
as.list(quote(saveRDS(mydata, "secret.rds")))
[[1]]
saveRDS

[[2]]
mydata

[[3]]
[1] "secret.rds"
```

This works because in R, expressions are simply data!

You can walk through the tree and work with parts of the expression at will

This sort of processing is used in all sorts of places:

- automatic plot axes
- **library**
- data.frames that build out of the names

A simpler interface

```
cyphr::encrypt(write.csv(mydata, "secret.csv"), key)
# Write mydata to temp file using write.csv
# Encrypt temp file to "secret.csv" using key
# Delete temp file

mydata <- cyphr::decrypt(read.csv("secret.csv"), key)
# Decrypt "secret.csv" into temp file using key
# Read mydata from temp file using read.csv
# Delete temp file
```

A simpler interface

```
cyphr::encrypt(saveRDS(mydata, "secret.rds"), key)
# Write mydata to temp file using saveRDS
# Encrypt temp file to "secret.rds" using key
# Delete temp file

mydata <- cyphr::decrypt(readRDS("secret.rds"), key)
# Decrypt "secret.rds" into temp file using key
# Read mydata from temp file using readRDS
# Delete temp file
```

We can change the target function to read and write different types of files and everything just works

Encrypting an analysis

```
mydata <- read.csv("secret.csv")  
  
newdata <- my_analysis_function(mydata)  
  
saveRDS(newdata, "export.rds")
```

The idea is that it can then just be taken to an existing analysis and wrapped around the code that already exists

Rather than having to replace every input/output line with 5 lines of repetitive and error-prone code, or using special encryption/decryption functions we can change an analysis very simply

Encrypting an analysis

```
mydata <- cyphr::decrypt(read.csv("secret.csv"), key)  
newdata <- my_analysis_function(mydata)  
cyphr::encrypt(saveRDS(newdata, "export.rds"), key)
```

This does not work with plotting (yet)

Alternative approaches would be to use encrypted volumes but these are less portable and awkward to share



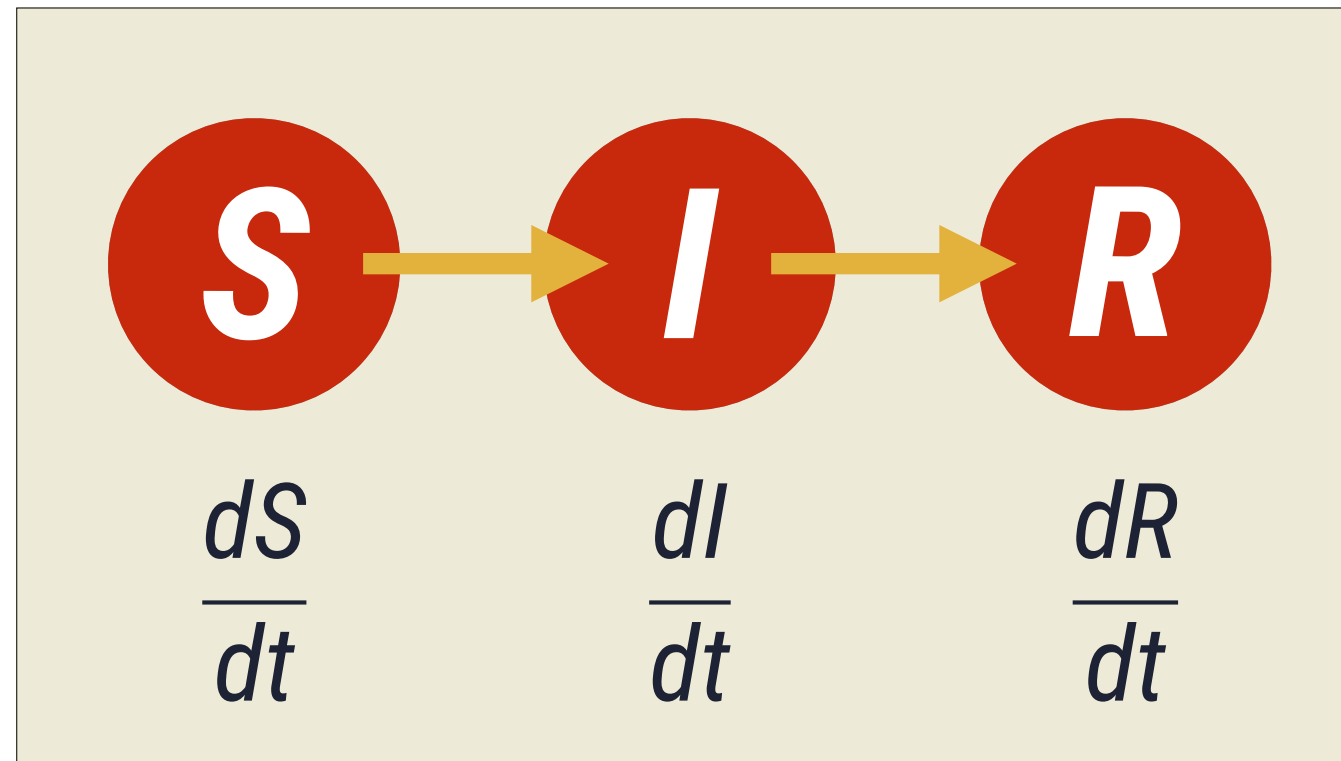
A little goes a long way - talk about how this breaks referential transparency and so needs to be used with care

Talk about where this is used elsewhere in the R ecosystem - **library**, **dplyr**, **subset**, etc

Talk about how programming with these functions can be hard and how a whole new package (**rlang**) exists to try and simplify programming with NSE. Not best used everywhere, but when used lightly can be **very expressive**

Differential equations

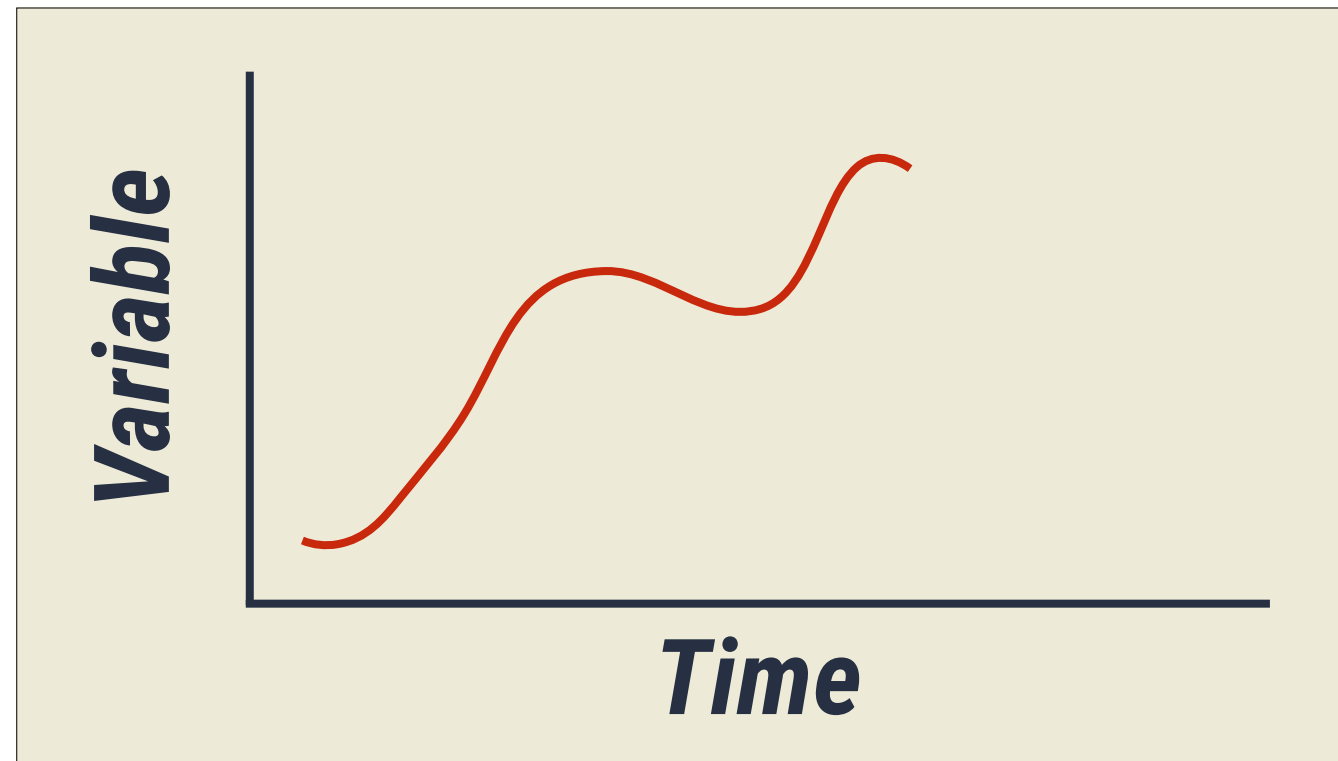
Something completely different!



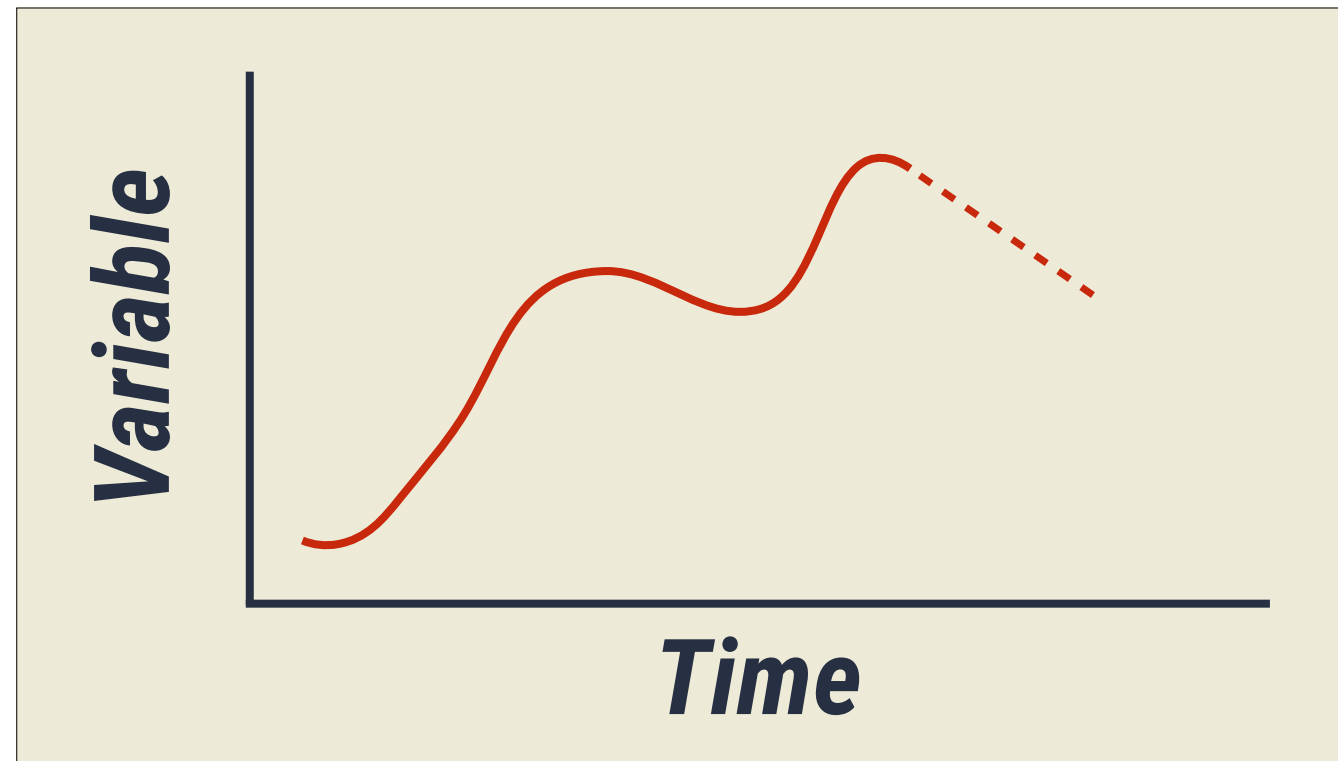
Susceptible, infected, resistant

Described in terms of **rates of change** of the variables with respect to time

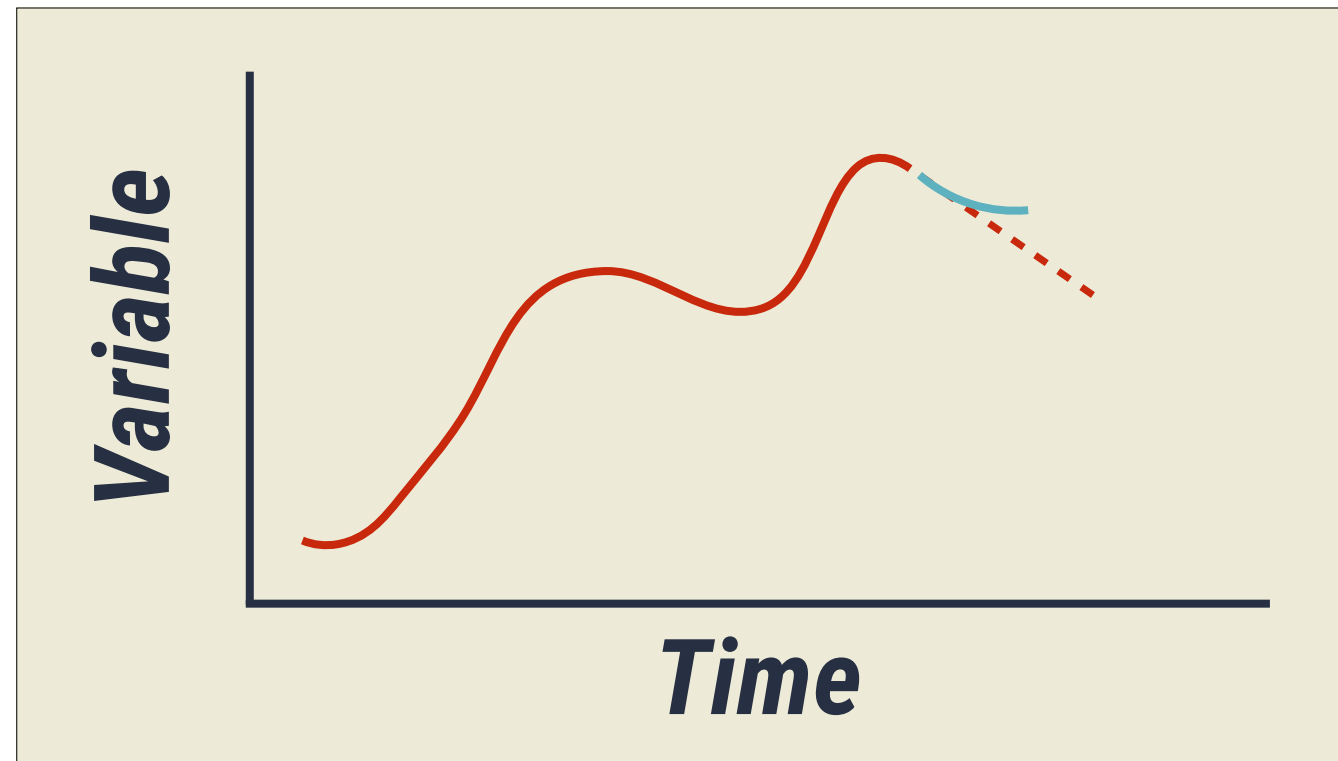
Common type of modelling, and usually intractable analytically



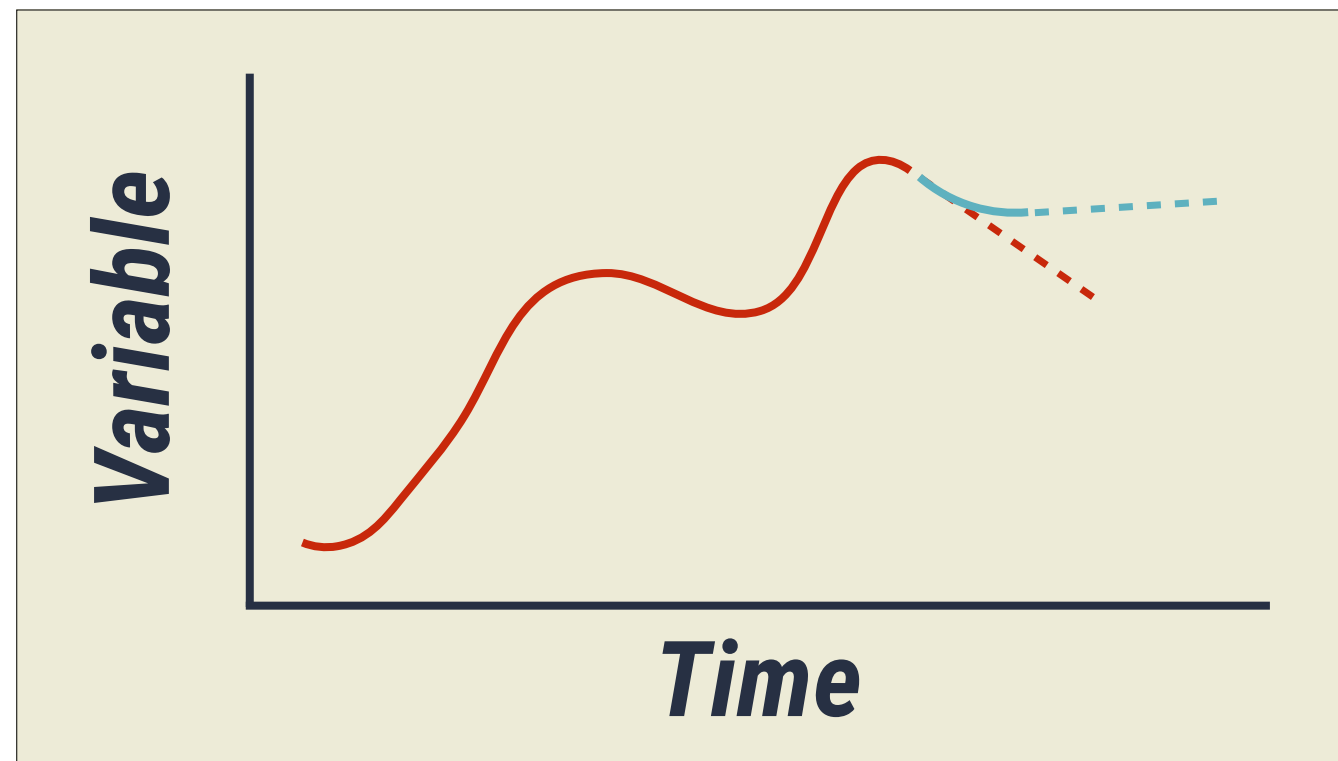
But suppose we did know the position of the variable at some point in time



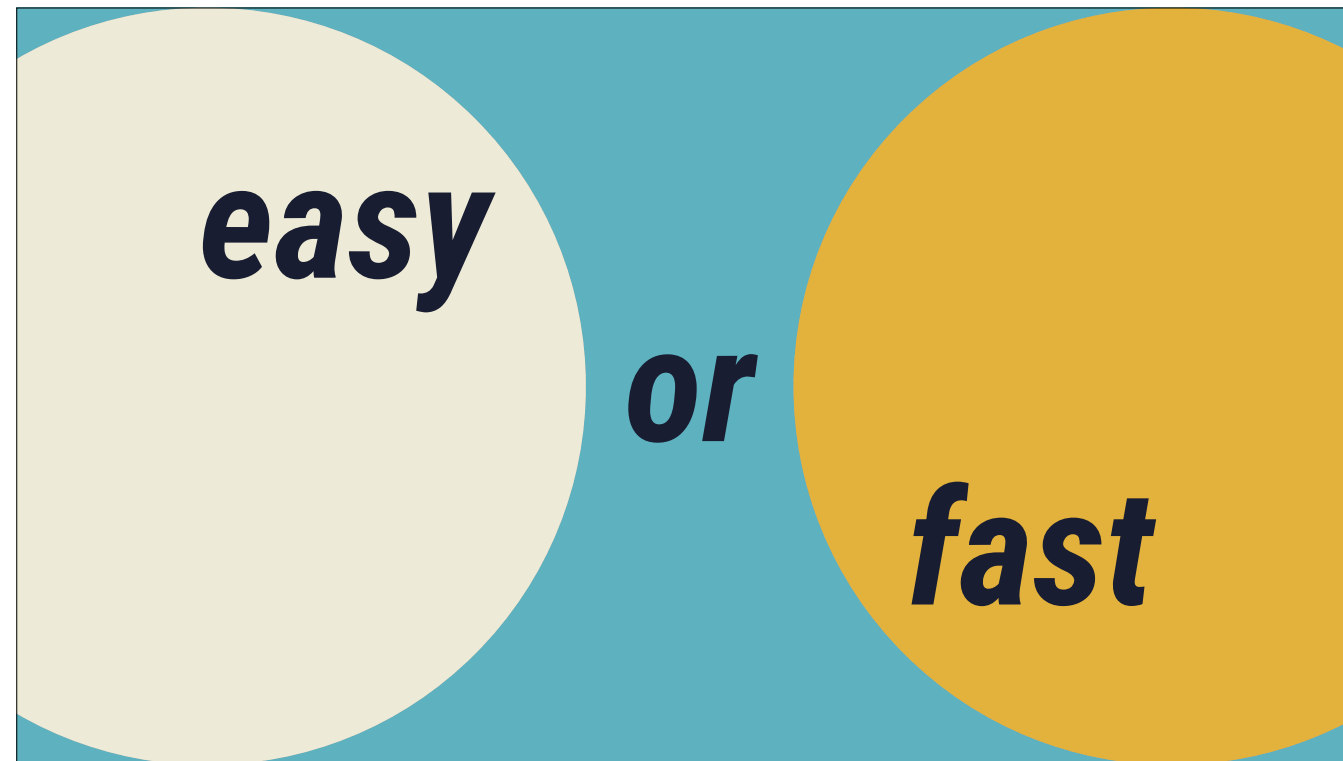
And we can compute its slope with respect to time



We can extrapolate down and correct as we go and work out the rest of the curve



Repeating over and over again



In R there are generally two choices for doing differential equation modelling - easy (expressive) but slow or fast (but harder to write and maintain)

easy

```
lorenz <- function(t, y, parms)
{
  sigma <- parms[1]
  R <- parms[2]
  b <- parms[3]
  y1 <- y[1]
  y2 <- y[2]
  y3 <- y[3]
  list(c(sigma * (y2 - y1),
          R * y1 - y2 - y1 * y3,
          -b * y3 + y1 * y2))
}
```

In the easy form you just write the rhs as an R function - you can take time and whatever parameters you want, the current state of the variables and do anything you want.

You can use any R function no matter how weird

All you need to do is return the derivatives with respect to time in the same order as the variables

easy

```
lorenz <- function(t, y, parms)
{
  sigma <- parms[1]
  R <- parms[2]
  b <- parms[3]
  y1 <- y[1]
  y2 <- y[2]
  y3 <- y[3]
  list(c(sigma * (y2 - y1),
        R * y1 - y2 - y1 * y3,
        -b * y3 + y1 * y2))
}
```

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

Then just pass this in to **deSolve** where there are a lot of great solvers to use


```

void initmod(void (* odeparms)(int *, double *)) {
    int N = 3;
    odeparms(&N, parms);
}

void lorenz(int *n, double *t, double *y, double *dydt, double *yout, int *ip)
{
    double sigma = parms[0];
    double R = parms[1];
    double b = parms[2];
    double y1 = y[0];
    double y2 = y[1];
    double y3 = y[2];
    dydt[0] = sigma * (y2 - y1);
    dydt[1] = R * y1 - y2 - y1 * y3;
    dydt[2] = -b * y3 + y1 * y2;
}

```

fast

- The fast version is a bit more work
- Rather than write in R we write in C
- We can't use names anywhere and need to remember where everything is in our vector
- Uses C's semantics of not returning a vector but writing in place

```

void initmod(void (* odeparms)(int *, double *)) {
    int N = 3;
    odeparms(&N, parms);
}

void lorenz(int *n, double *t, double *y, double *dydt, double *yout, int *ip)
{
    double sigma = parms[0];
    double R = parms[1];
    double b = parms[2];
    double y1 = y[0];
    double y2 = y[1];
    double y3 = y[2];
    dydt[0] = sigma * (y2 - y1);
    dydt[1] = R * y1 - y2 - y1 * y3;
    dydt[2] = -b * y3 + y1 * y2;
}

deSolve::ode(t, y, "lorenz", initfunc = "initmod", dllname = "lorenz")

```

fast

- We need to compile this, then load it into R
- Once in R we can run it basically as before

```

void initmod(void (* odeparms)(int *, double *)) {
    int N = 3;
    odeparms(&N, parms);
}

void lorenz(int *n, double *t, double *y, double *dydt, double *yout, int *ip)
{
    double sigma = parms[0];
    double R = parms[1];
    double b = parms[2];
    double y1 = y[0];
    double y2 = y[1];
    double y3 = y[2];
    dydt[0] = sigma * (y2 - y1);
    dydt[1] = R * y1 - y2 - y1 * y3;
    dydt[2] = -b * y3 + y1 * y2;
}

lorenz <- function(t, y, parms)
{
    sigma <- parms[1]
    R <- parms[2]
    b <- parms[3]
    y1 <- y[1]
    y2 <- y[2]
    y3 <- y[3]
    list(c(sigma * (y2 - y1),
           R * y1 - y2 - y1 * y3,
           -b * y3 + y1 * y2))
}

```

These approaches really aren't that different

```

void initmod(void (* odeparms)(int *, double *)) {
    int N = 3;
    odeparms(&N, parms);
}

void lorenz(int *n, double *t, double *y, double *dydt, double *yout, int *ip)
{
    double sigma = parms[0];
    double R = parms[1];
    double b = parms[2];
    double y1 = y[0];
    double y2 = y[1];
    double y3 = y[2];
    dydt[0] = sigma * (y2 - y1);
    dydt[1] = R * y1 - y2 - y1 * y3;
    dydt[2] = -b * y3 + y1 * y2;
}

lorenz <- function(t, y, parms)
{
    sigma <- parms[1]
    R <- parms[2]
    b <- parms[3]
    y1 <- y[1]
    y2 <- y[2]
    y3 <- y[3]
    list(c(sigma * (y2 - y1),
           R * y1 - y2 - y1 * y3,
           -b * y3 + y1 * y2))
}

```

Unpack our parameters

```

void initmod(void (* odeparms)(int *, double *)) {
    int N = 3;
    odeparms(&N, parms);
}

void lorenz(int *n, double *t, double *y, double *dydt, double *yout, int *ip)
{
    double sigma = parms[0];
    double R = parms[1];
    double b = parms[2];
    double y1 = y[0];
    double y2 = y[1];
    double y3 = y[2];
    dydt[0] = sigma * (y2 - y1);
    dydt[1] = R * y1 - y2 - y1 * y3;
    dydt[2] = -b * y3 + y1 * y2;
}

lorenz <- function(t, y, parms)
{
    sigma <- parms[1]
    R <- parms[2]
    b <- parms[3]
    y1 <- y[1]
    y2 <- y[2]
    y3 <- y[3]
    list(c(sigma * (y2 - y1),
           R * y1 - y2 - y1 * y3,
           -b * y3 + y1 * y2))
}

```

Unpack our variables

```

void initmod(void (* odeparms)(int *, double *)) {
    int N = 3;
    odeparms(&N, parms);
}

void lorenz(int *n, double *t, double *y, double *dydt, double *yout, int *ip)
{
    double sigma = parms[0];
    double R = parms[1];
    double b = parms[2];
    double y1 = y[0];
    double y2 = y[1];
    double y3 = y[2];
    dydt[0] = sigma * (y2 - y1);
    dydt[1] = R * y1 - y2 - y1 * y3;
    dydt[2] = -b * y3 + y1 * y2;
}

lorenz <- function(t, y, parms)
{
    sigma <- parms[1]
    R <- parms[2]
    b <- parms[3]
    y1 <- y[1]
    y2 <- y[2]
    y3 <- y[3]
    list(c(sigma * (y2 - y1),
           R * y1 - y2 - y1 * y3,
           -b * y3 + y1 * y2))
}

```

Compute our derivatives

```

void initmod(void (* odeparms)(int *, double *)) {
    int N = 3;
    odeparms(&N, parms);
}

void lorenz(int *n, double *t, double *y, double *dydt, double *yout, int *ip)
{
    double sigma = parms[0];
    double R = parms[1];
    double b = parms[2];
    double y1 = y[0];
    double y2 = y[1];
    double y3 = y[2];
    dydt[0] = sigma * (y2 - y1);
    dydt[1] = R * y1 - y2 - y1 * y3;
    dydt[2] = -b * y3 + y1 * y2;
}

lorenz <- function(t, y, parms)
{
    sigma <- parms[1]
    R <- parms[2]
    b <- parms[3]
    y1 <- y[1]
    y2 <- y[2]
    y3 <- y[3]
    list(c(sigma * (y2 - y1),
           R * y1 - y2 - y1 * y3,
           -b * y3 + y1 * y2))
}

```

What is this? It's a spell that we write

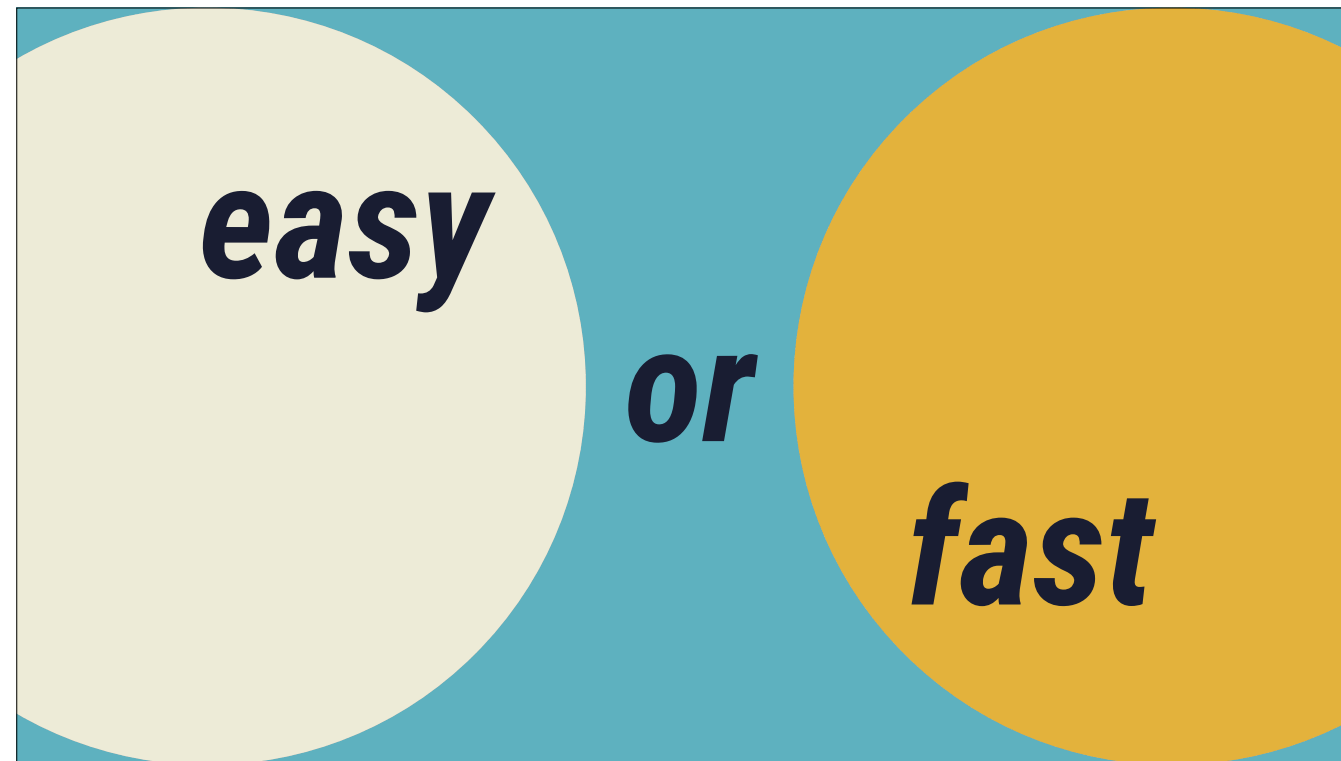
(it's actually a function that takes a function pointer as an argument, then calls that function on the address of stack allocated integer, along with the vector of parameters that deSolve will pass in)

```
void initmod(void (* odeparms)(int *, double *)) {
    int N = 3;
    odeparms(&N, parms);
}

void lorenz(int *n, double *t, double *y, double *dydt, double *yout, int *ip)
{
    double sigma = parms[0];
    double R = parms[1];
    double b = parms[2];
    double y1 = y[0];
    double y2 = y[1];
    double y3 = y[2];
    dydt[0] = sigma * (y2 - y1);
    dydt[1] = R * y1 - y2 - y1 * y3;
    dydt[2] = -b * y3 + y1 * y2;
}
```



(it's actually a function that takes a function pointer as an argument, then calls that function on the address of stack allocated integer, along with the vector of parameters that deSolve will pass in)



Obviously real world cases are going to be more complicated than this but it would be nice to be able to remove the tradeoff here



```
lorenz <- odin::odin({  
  ## Derivatives  
  deriv(y1) <- sigma * (y2 - y1)  
  deriv(y2) <- R * y1 - y2 - y1 * y3  
  deriv(y3) <- -b * y3 + y1 * y2  
  
  ## Initial conditions  
  initial(y1) <- 10.0  
  initial(y2) <- 1.0  
  initial(y3) <- 1.0  
  
  ## parameters  
  sigma <- user()  
  R      <- user()  
  b      <- user()  
})
```

odin

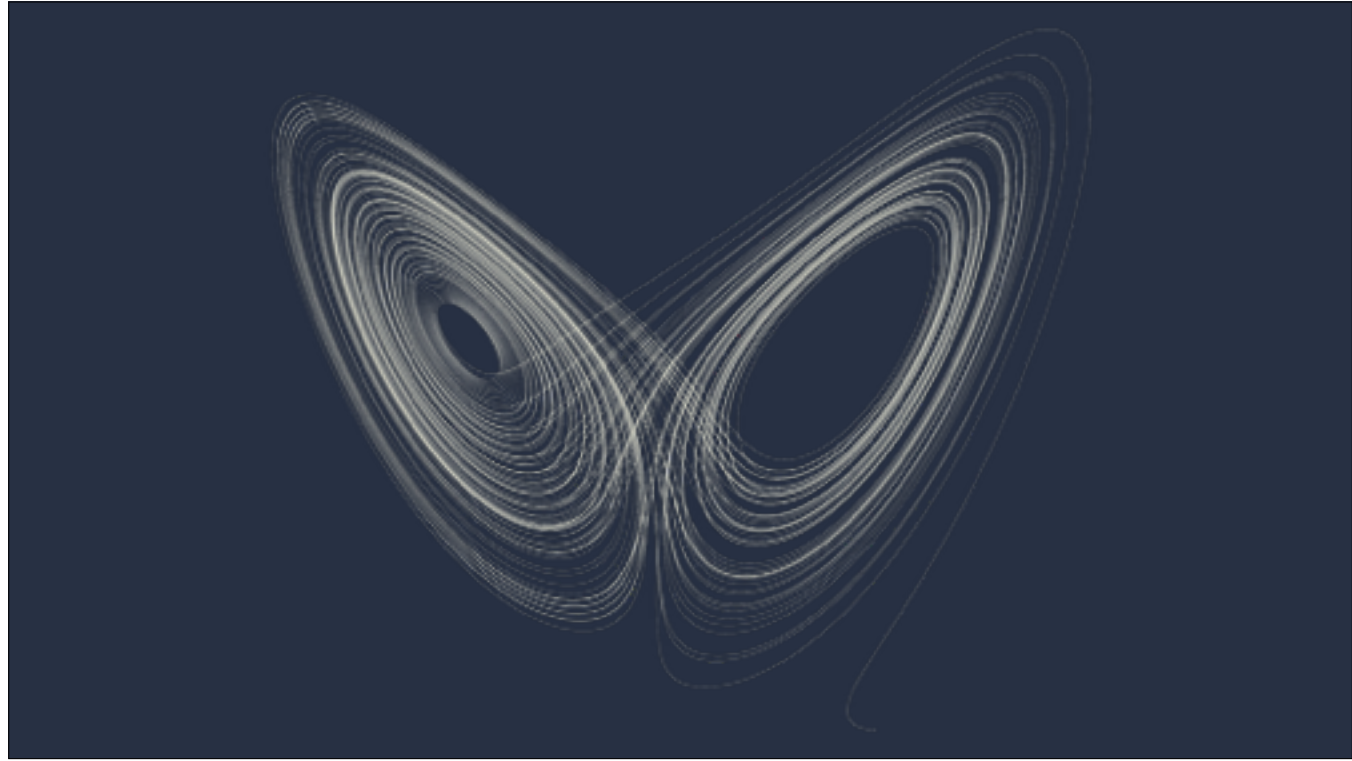
Odin is a "domain specific language"

Same idea as the encryption package but more extreme - now we take a set of R code and never run it - we just use it define a problem. We can pull it apart and do things to it - in this case generate C code, compile that, load it into R and wrap that up as an R object

```
lorenz <- odin::odin({  
  ...  
  sigma <- user()  
  R     <- user()  
  b     <- user()  
})  
  
model <- lorenz(sigma = 10.0,  
                R = 28.0,  
                b = 8 / 3)  
t <- seq(0, 50, length.out = 10000)  
y <- model$run(t)
```

odin

name the args in the call



Rewriting expressions

```
deriv(y1) <- sigma * (y2 - y1)
```

```
list(`  
  deriv(y1),  
  sigma * (y2 - y1))
```

```
dydt[0] = sigma * (y2 - y1);
```

Rewriting expressions

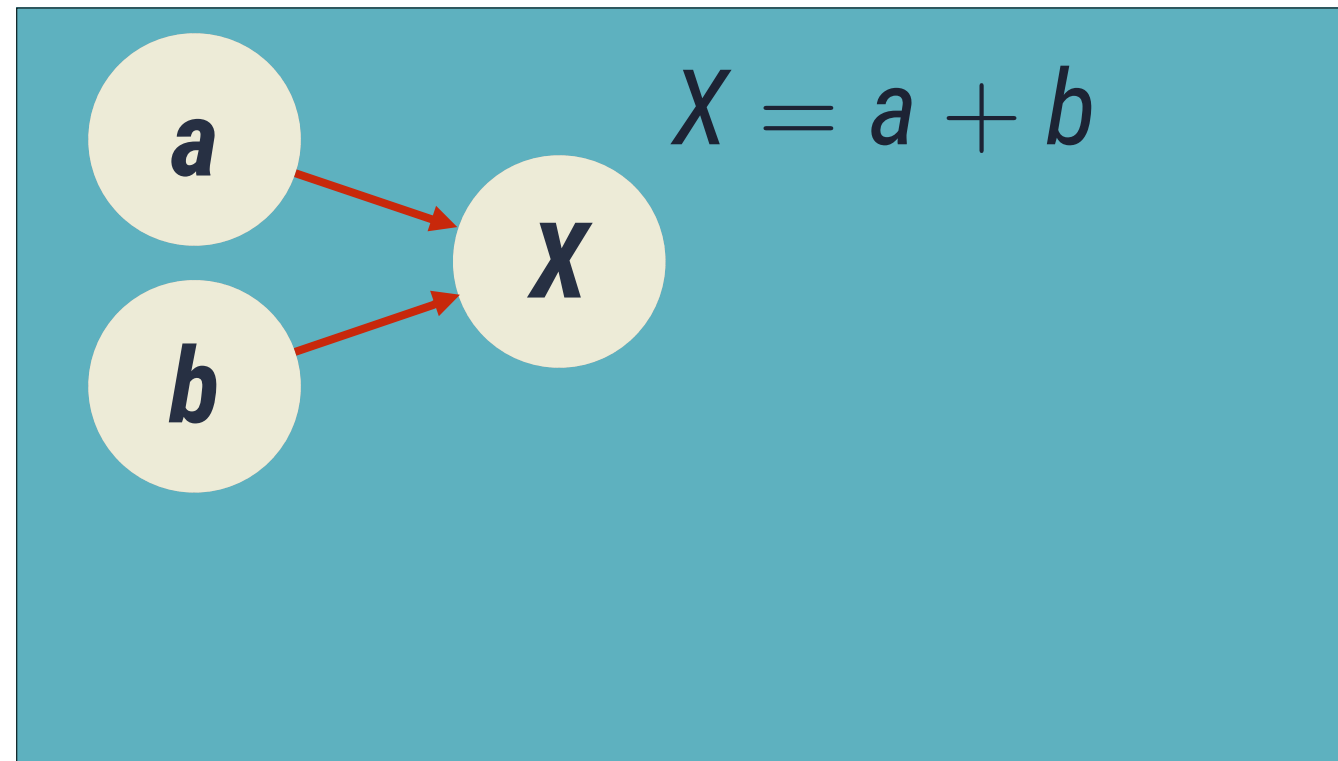
```
deriv(y1[]) <- sigma * (y2[i] - y1[i])  
  
list(`  
  deriv(y1[]),  
  sigma * (y2[i] - y1[i]))  
  
for (size_t i = 0; i < len_y1; ++i) {  
  dydt[i] = sigma * (y2[i] - y1[i]);  
}
```

this approach scales up through application of simple rules to support things like automatically working over arrays

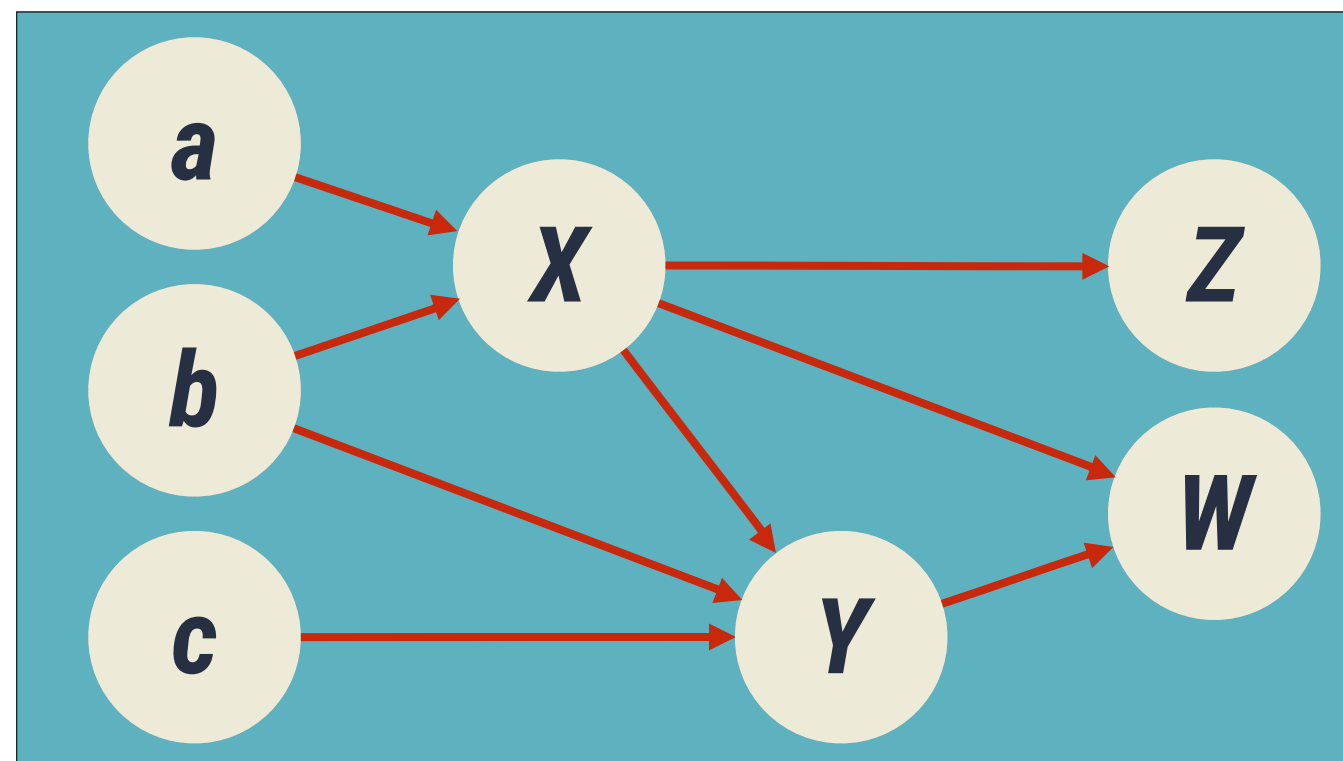
```
lorenz <- odin::odin({  
  ## Derivatives  
  deriv(y1) <- sigma * (y2 - y1)  
  deriv(y2) <- R * y1 - y2 - y1 * y3  
  deriv(y3) <- -b * y3 + y1 * y2  
  
  ## Initial conditions  
  initial(y1) <- 10.0  
  initial(y2) <- 1.0  
  initial(y3) <- 1.0  
  
  ## parameters  
  sigma <- user()  
  R      <- user()  
  b      <- user()  
})
```

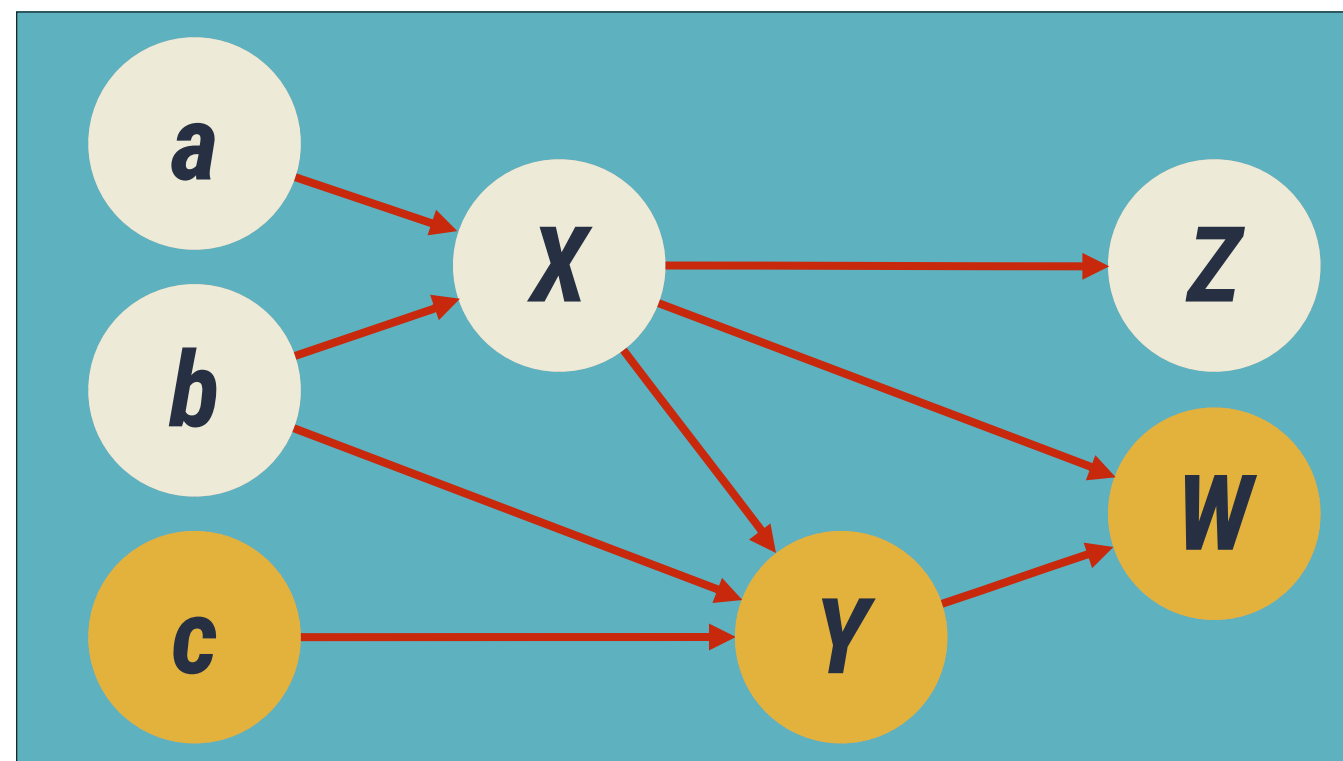
Odin is a "domain specific language"

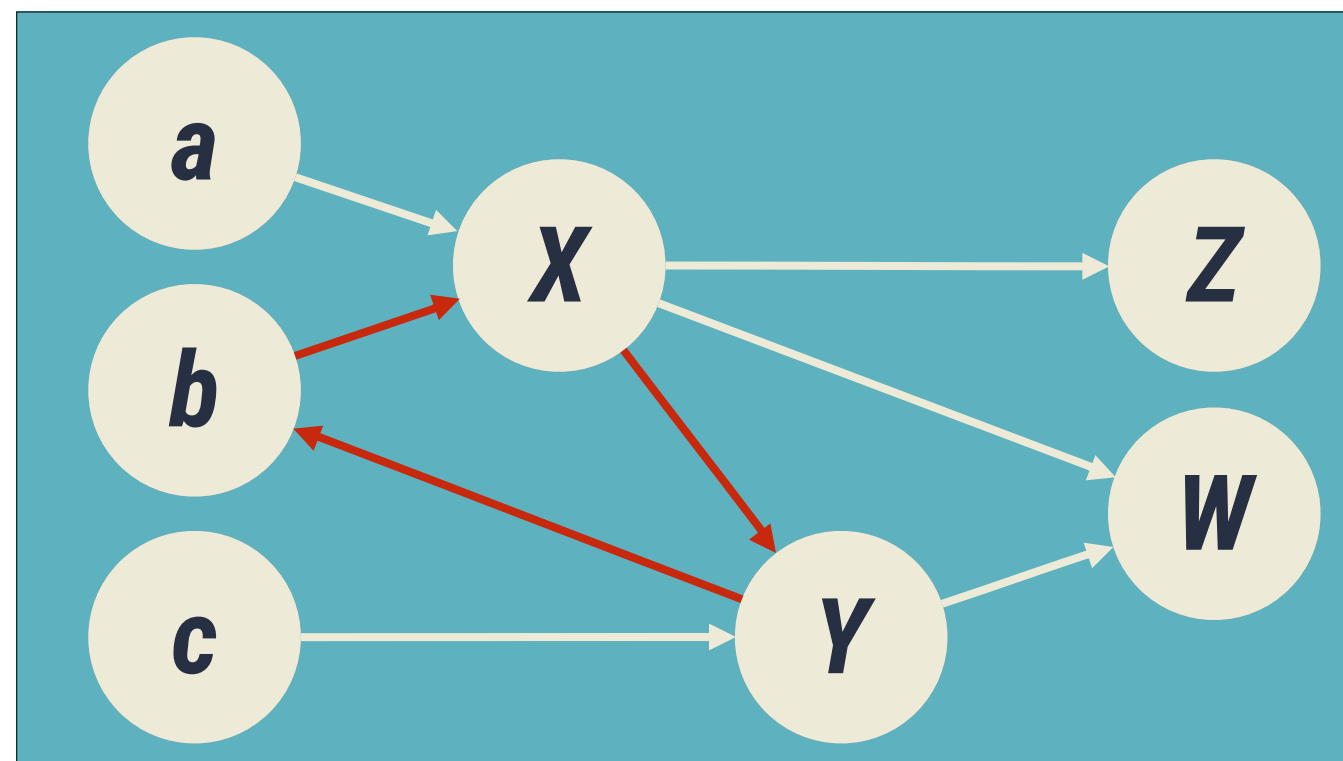
Same idea as the encryption package but more extreme - now we take a set of R code and never run it - we just use it define a problem. We can pull it apart and do things to it - in this case generate C code, compile that, load it into R and wrap that up as an R object

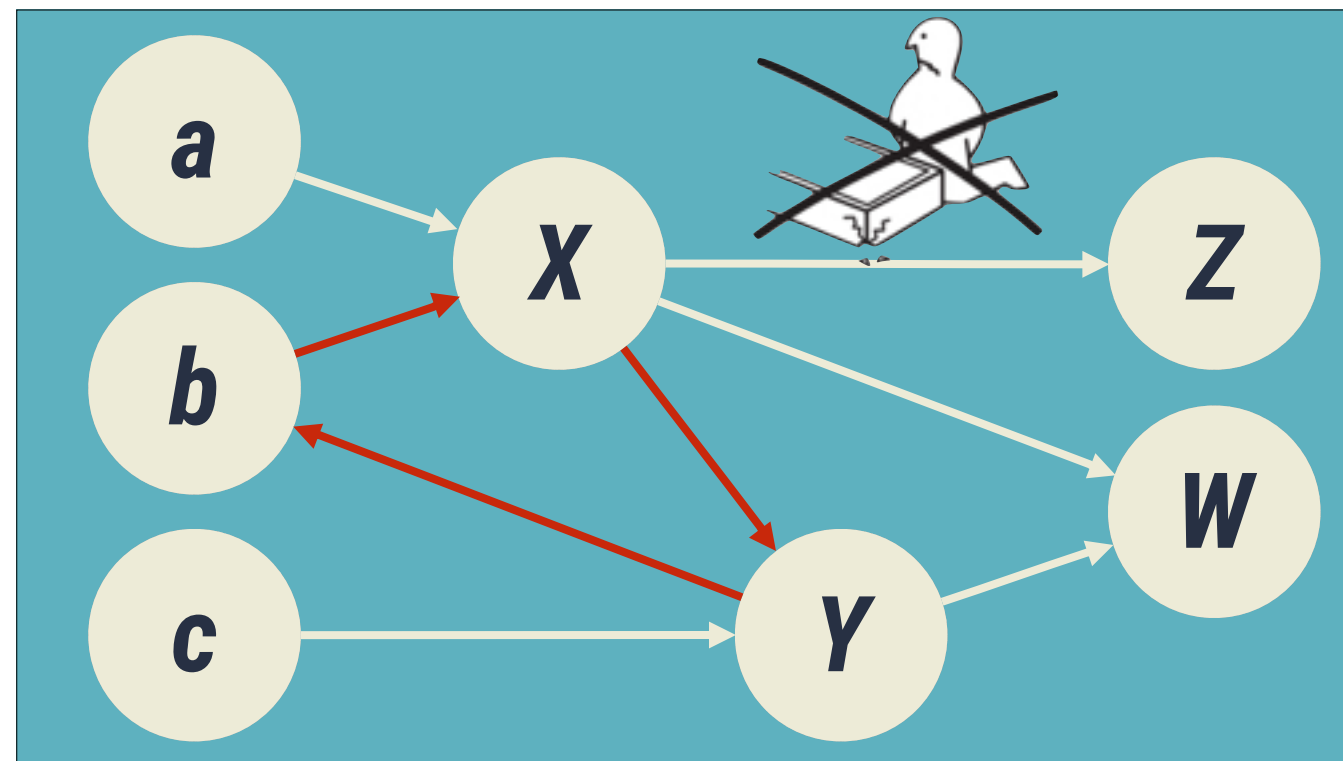


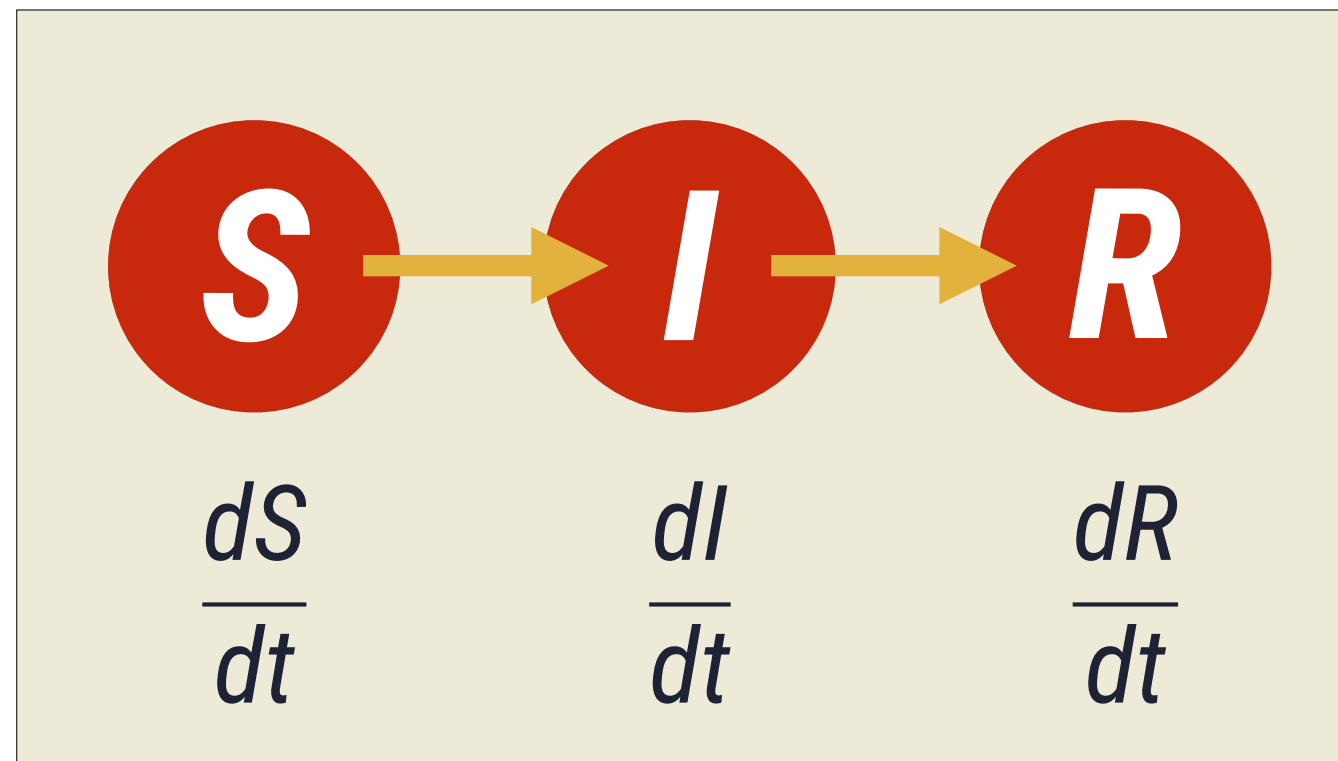
transition missing here - return to the SIR model then display this







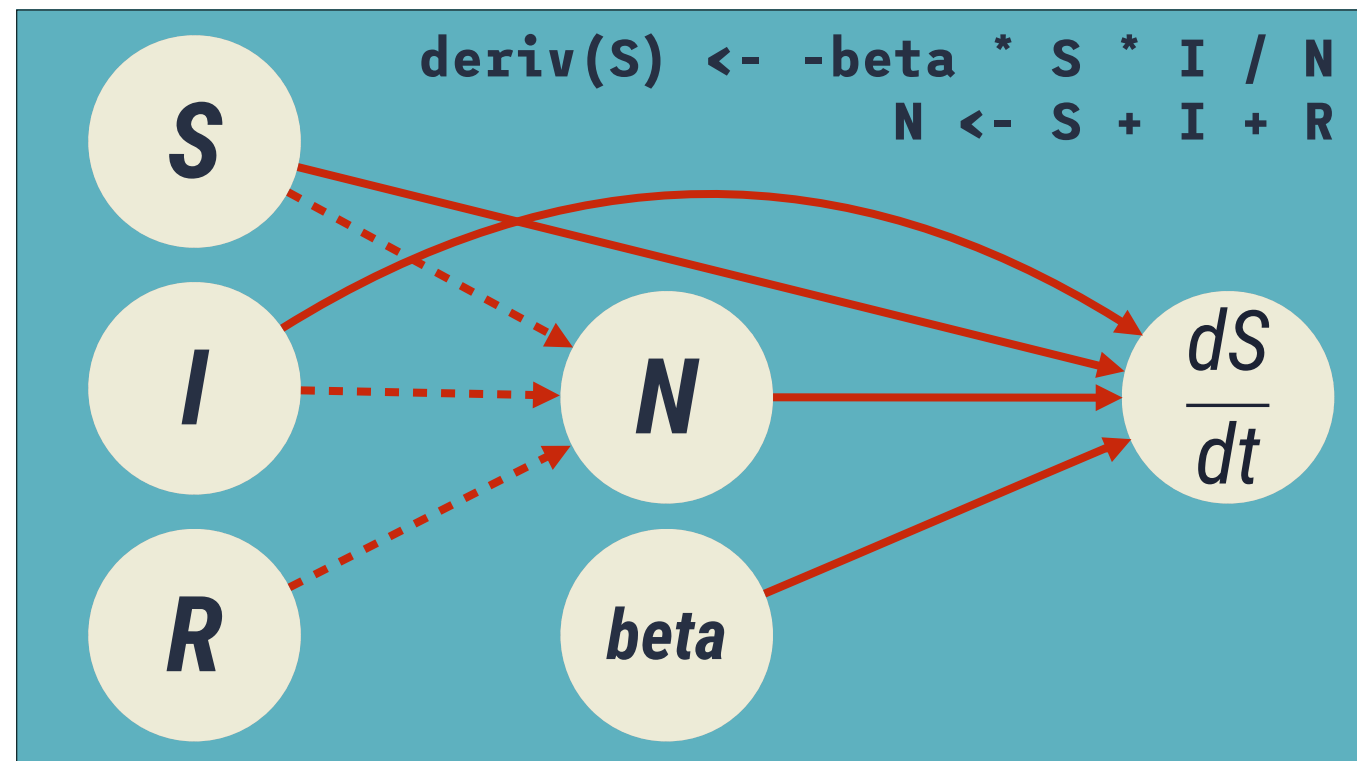


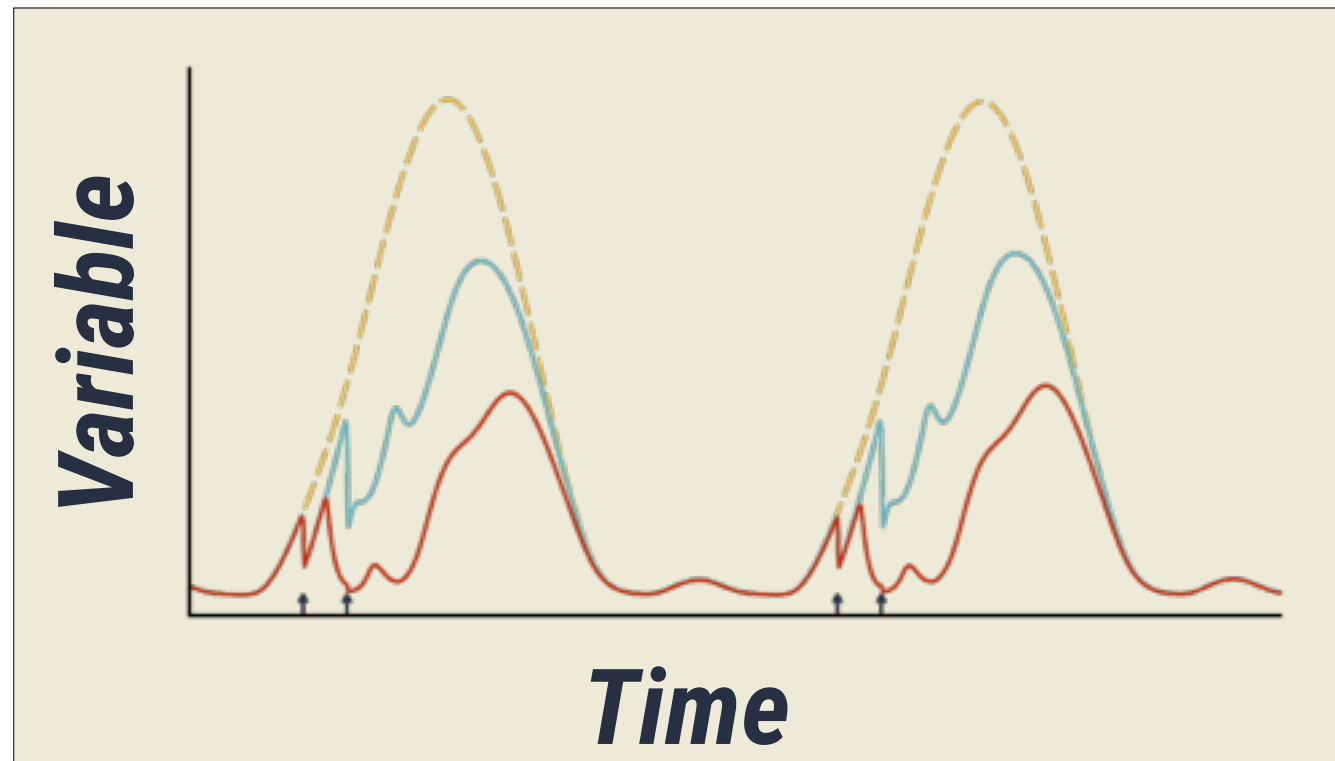


Susceptible, infected, resistant

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

don't forget to fix this slide - the right side needs to go away





Application in an epi context:

- ODE systems with ~10k compartments
- Periodic interventions
- **Delays** where whole parts of the graph are replayed

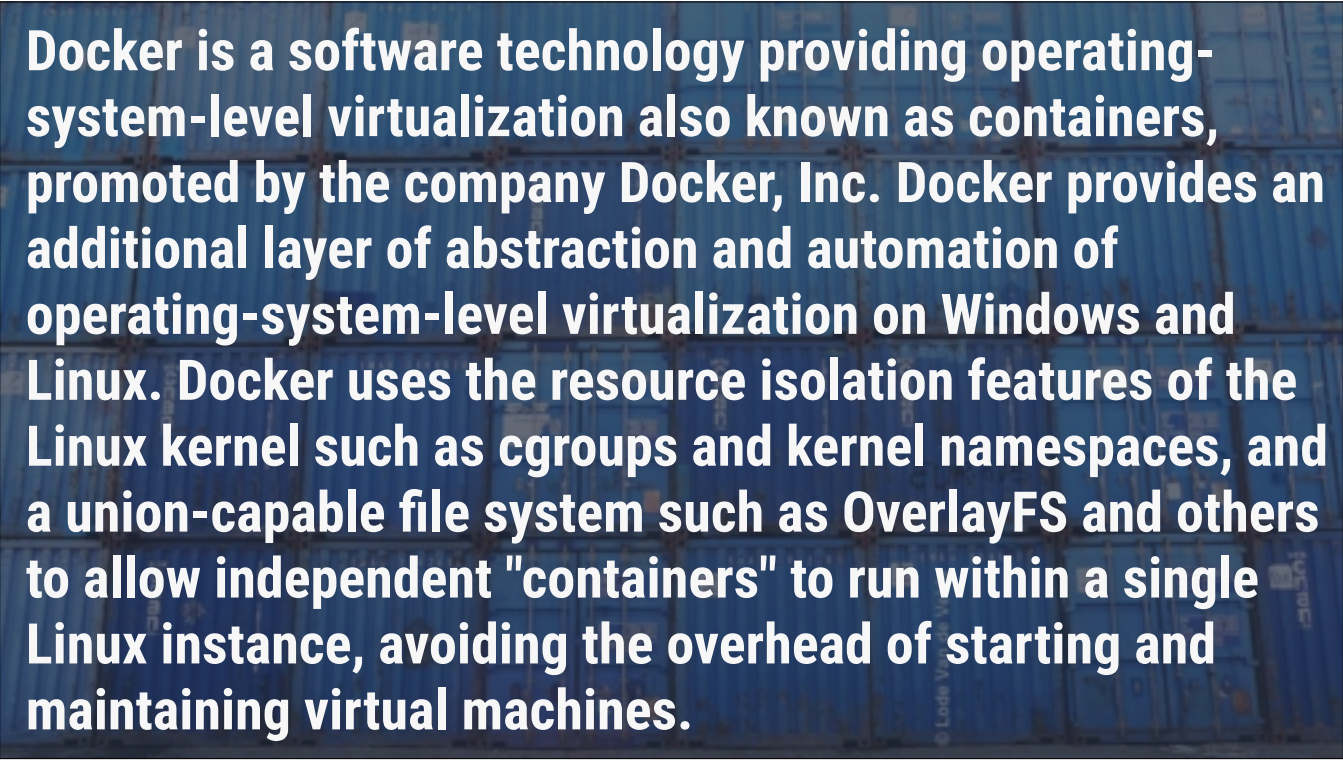
Package is agnostic about how inference with these models would be done - there's lots of different reasons to solve an ODE system and it's out of scope here.

Scope of the package

- Simple delay differential equations (limited by R's DDE solvers)
- Arrays for structured compartment models
- Same interface for discrete time models, which may be stochastic

Docker



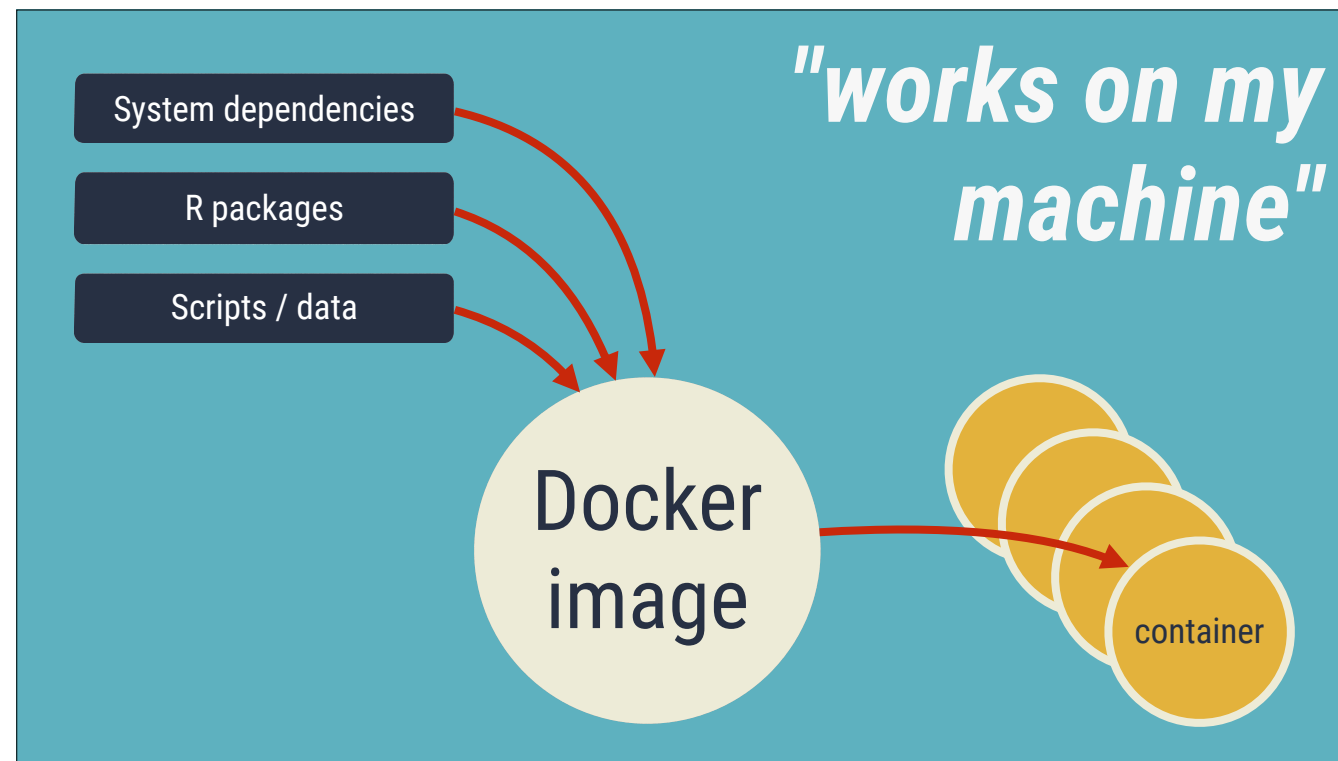


Docker is a software technology providing operating-system-level virtualization also known as containers, promoted by the company Docker, Inc. Docker provides an additional layer of abstraction and automation of operating-system-level virtualization on Windows and Linux. Docker uses the resource isolation features of the Linux kernel such as cgroups and kernel namespaces, and a union-capable file system such as OverlayFS and others to allow independent "containers" to run within a single Linux instance, avoiding the overhead of starting and maintaining virtual machines.

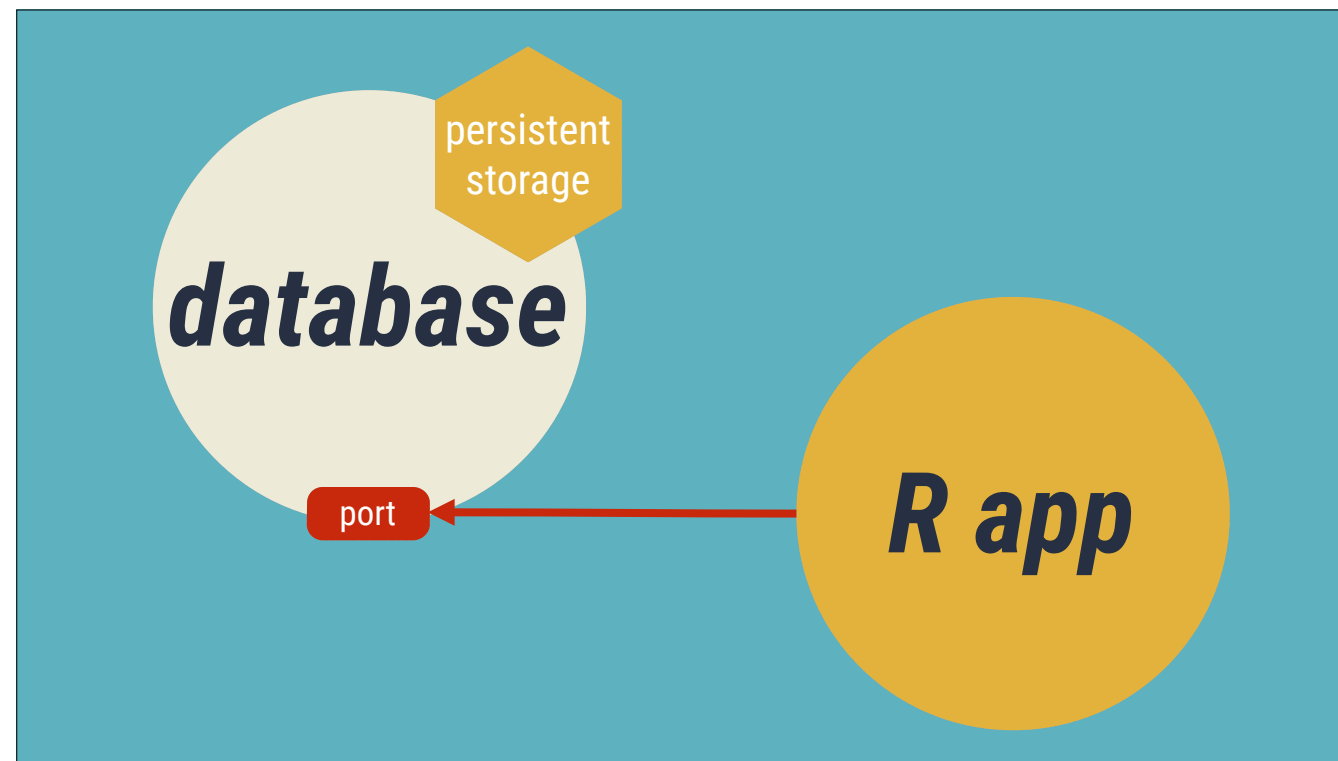
It's really hard to explain what docker is and not be really boring and/or incorrect



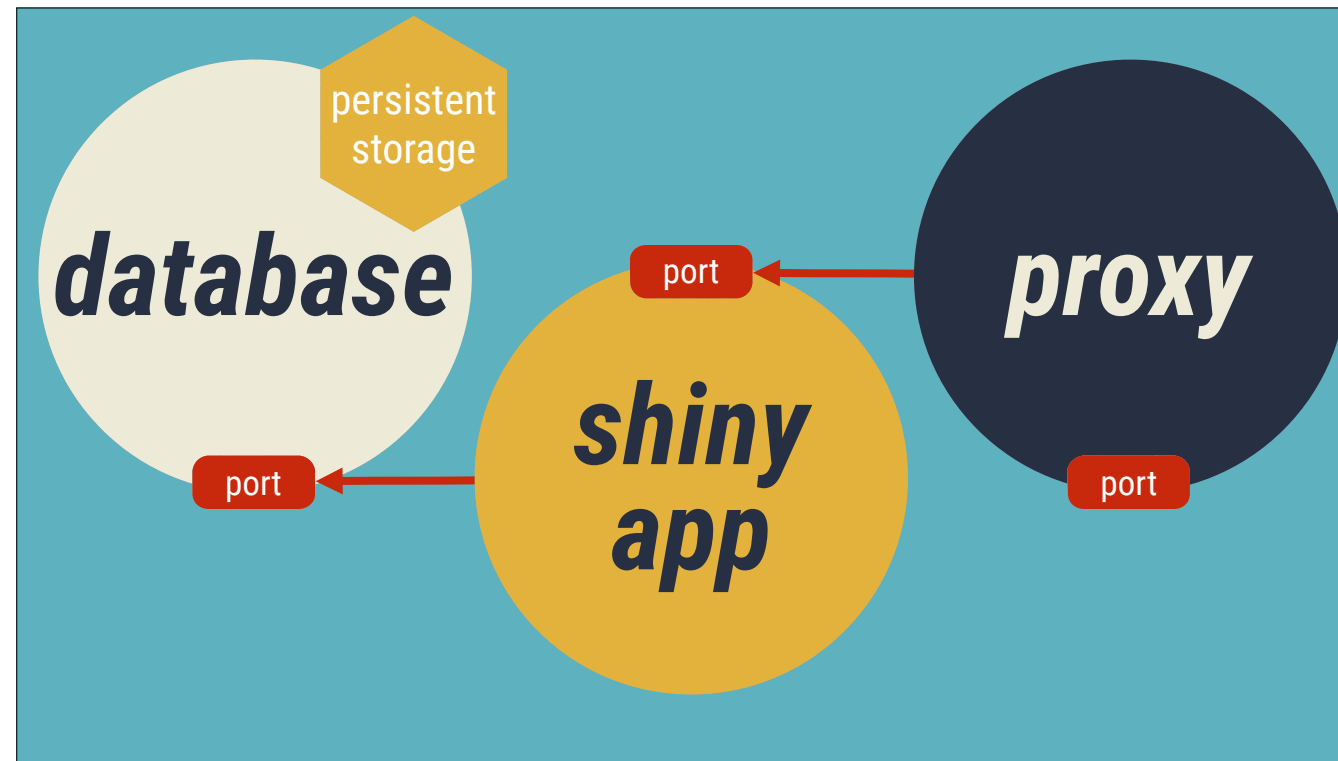
More interesting is to ask **why** one would want to use docker



break into 2?



break into 2 as well





The metaphor is containers because that's like the shipping revolution - pre containers loading and unloading at the docs was labour intensive



but after containers turn up it's much easier to ignore what is inside and just build tools around a standard size

```
/containers/create:
  post:
    summary: "Create a container"
    consumes:
      - "application/json"
    parameters:
      - name: "name"
        in: "query"
        description: "Assign the specified name to the container."
        type: "string"
    responses:
      201:
        description: "Container created successfully"
        schema:
          type: "object"
          description: "OK response to ContainerCreate operation"
          properties:
            Id:
              description: "The ID of the created container"
              type: "string"
```

swagger

OK so docker is awesome, let's use it from R!

```
/containers/create:
post:
  summary: "Create a container"
  consumes:
    - "application/json"
  parameters:
    - name: "name"
      in: "query"
      description: "Assign the specified name to the container."
      type: "string"
  responses:
    201:
      description: "Container created successfully"
      schema:
        type: "object"
        description: "OK response to ContainerCreate operation"
        properties:
          Id:
            description: "The ID of the created container"
            type: "string"
```

where

```
/containers/create:
  post:
    summary: "Create a container"
    consumes:
      - "application/json"
    parameters:
      - name: "name"
        in: "query"
        description: "Assign the specified name to the container."
        type: "string"
    responses:
      201:
        description: "Container created successfully"
        schema:
          type: "object"
          description: "OK response to ContainerCreate operation"
          properties:
            Id:
              description: "The ID of the created container"
              type: "string"
```

parameters

```
/containers/create:
  post:
    summary: "Create a container"
    consumes:
      - "application/json"
    parameters:
      - name: "name"
        in: "query"
        description: "Assign the specified name to the container."
        type: "string"
    responses:
      201:
        description: "Container created successfully"
        schema:
          type: "object"
          description: "OK response to ContainerCreate operation"
          properties:
            Id:
              description: "The ID of the created container"
              type: "string"
```

returning

```
/containers/create:
  post:
    summary: "Create a container"
    consumes:
      - "application/json"
    parameters:
      - name: "name"
        in: "query"
        description: "Assign the specified name to the container."
        type: "string"
    responses:
      201:
        description: "Container created successfully"
        schema:
          type: "object"
          description: "OK response - Container creation"
          properties:
            Id:
              description: "The ID of the created container"
              type: "string"
```

90 methods

10,000 lines

12 versions

```
if tmpfs:
    if version_lt(version, '1.22'):
        raise host_config_version_error('tmpfs', '1.22')
    self["Tmpfs"] = convert_tmpfs_mounts(tmpfs)

if userns_mode:
    if version_lt(version, '1.23'):
        raise host_config_version_error('userns_mode', '1.23')
    self['UsernsMode'] = userns_mode

if pids_limit:
    if version_lt(version, '1.23'):
        raise host_config_version_error('pids_limit', '1.23')
    self["PidsLimit"] = pids_limit

if isolation:
    if version_lt(version, '1.24'):
        raise host_config_version_error('isolation', '1.24')
    self['Isolation'] = isolation

if auto_remove:
    if version_lt(version, '1.25'):
        raise host_config_version_error('auto_remove', '1.25')
    self['AutoRemove'] = auto_remove
```

*if
else
hell*

The python folks have a really wonderful docker client - from a user perspective it's a joy to use but the code behind it hits this version issue head on

But we have this crazy dynamic language, so let's do it a different way.

How to write a function

```
add <- function(a, b) {  
  a + b  
}
```

*How to **build** a function*

```
add <- function(a, b) {  
  a + b  
}  
args <- alist(a =, b =)  
body <- quote(a + b)  
add <- as.function(c(args, body))
```

How to draw an Owl.

"A fun and creative guide for beginners"



Fig 1. Draw two circles



Fig 2. Draw the rest of the damn Owl

stevedore

```
docker <- stevedore::docker_client()
```



(this gif will play on the README at <https://github.com/richfitz/stevedore>)

Stevedore

```
docker <- stevedore::docker_client(  
  api_version = "1.35")
```

Testing packages

1. Install database
2. Configure & set up passwords
3. Use database in package tests
4. Make sure you clean up properly!

OK, so why:

- * you can set up an empty container for testing your package just like travis (but locally)
- * but you can extend that all the way out to cover things like getting database included

```
echo mysql-server mysql-server/root_password password $MYSQL_PASSWORD | \
    debconf-set-selections
echo mysql-server mysql-server/root_password_again password $MYSQL_PASSWORD | \
    debconf-set-selections
apt-get install -y mysql-server

systemctl stop mysql
mv /var/lib/mysql /mnt/data/mysql
ln -s /mnt/data/mysql /var/lib/mysql

echo "alias /var/lib/mysql/ -> /mnt/data/mysql," >> \
    /etc/apparmor.d/tunables/alias
sudo systemctl restart apparmor
systemctl start mysql

mysql -u root -p$MYSQL_PASSWORD -e 'show databases;' | grep teamcity > /dev/null
if [ "$?" = "1" ]; then
    cat > /tmp/database-setup.sql <<EOF
CREATE DATABASE $TEAMCITY_DB_NAME DEFAULT CHARACTER SET utf8 COLLATE utf8_bin;
CREATE USER '$TEAMCITY_DB_USER'@'%' IDENTIFIED BY '$TEAMCITY_DB_PASS';
GRANT ALL ON $TEAMCITY_DB_NAME.* TO '$TEAMCITY_DB_USER'@'%;
EOF
    mysql -u root -p$MYSQL_PASSWORD < /tmp/database-setup.sql
    rm /tmp/database-setup.sql
fi
```

This is the database part of our setup for our continuous integration server VM (mysql not postgres)

Testing packages

```
env <- c("POSTGRES_PASS" = "s3cret!")  
db <- docker$containers$run("postgres", ports = "2222:5432",  
                             rm = TRUE, detach = TRUE,  
                             env = env)
```

Testing packages

```
env <- c("POSTGRES_PASS" = "s3cret!")
db <- docker$containers$run("postgres", ports = "2222:5432",
                             rm = TRUE, detach = TRUE,
                             env = env)
con <- dbConnect(Postgres(), host = "localhost", port = 2222,
                  user = "postgres", password = "s3cret!")
dbWriteTable(con, "table", mydata)
```

Testing packages

```
env <- c("POSTGRES_PASS" = "s3cret!")
db <- docker$containers$run("postgres", ports = "2222:5432",
                             rm = TRUE, detach = TRUE,
                             env = env)
con <- dbConnect(Postgres(), host = "localhost", port = 2222,
                  user = "postgres", password = "s3cret!")
dbWriteTable(con, "table", mydata)
dbGetQuery(con, "SELECT * FROM table LIMIT 20")
```

Testing packages

```
env <- c("POSTGRES_PASS" = "s3cret!")
db <- docker$containers$run("postgres", ports = "2222:5432",
                             rm = TRUE, detach = TRUE,
                             env = env)
con <- dbConnect(Postgres(), host = "localhost", port = 2222,
                  user = "postgres", password = "s3cret!")
dbWriteTable(con, "table", mydata)
dbGetQuery(con, "SELECT * FROM table LIMIT 20")
db$stop()
```

But it's not limited to that - you could control a load balancing AWS cluster using docker swarm for your massively parallel ML pipeline perhaps

Encryption

Differential equations

Docker

3 packages that do completely different things

Encryption

`cyphr` github.com/ropensci/cyphr

Differential equations

`odin` github.com/mrc-ide/odin

Docker

`stevedore` github.com/richfitz/stevedore

All 3 are available only through github at the moment, but will eventually make it to CRAN once they settle down

The common thread is that all exploit R's dynamic language to generate interfaces that narrow the gap between what the user wants to do and what it would take to do it

You don't have to use metaprogramming to do any of these things necessarily - but it's a useful and unusual trick

Just remember it's a bit like marmite!

R's weirdnesses are fun & useful

Rich FitzJohn

 richfitz

Hopefully that I've convinced you that R's weirdnesses are fun and useful and perhaps made you think about a package or an area that you'd not thought about before

1. R has some strange features that make it surprisingly powerful. These should be used with care
2. Three packages that do interesting things, using metaprogramming to build nice interfaces
3. Three fields that you may not have encountered with R