Inference problem: pseudo-code

To help us figure out exactly what we need from each tool / where different responsibilities lie / where we can expect some issues

```
#
# Get entities from front end
#
model = cellmlmanip.load_model('example.cellml')
protocol = fc.Protocol('example.txt')
data_set = front_end.get_data_set()
spec = front_end.get_parsed_fitting_spec()
# Leaving this open for now, not sure what it'd look like to parse it

front_end.ensure_compatibility(model, protocol, data_set, spec)
# This was probably already done before we got here. If not, there's some room
# for optimisation (later).

#
# Setting up a fitting problem
#

# Find _all_ parameters that _may_ be used in fitting later. The result maps
# cellmlmanip variables to desired units.
parameter_units = spec.find_all_parameters(model)

# Compile WebLab Model, associate it with the Protocol.
# At this stage, we _must_ tell the protocol what units we want the parameters
# in. But we don't know if all parameters are used to calculate the model
# outputs yet (see below).
protocol.set_model(model, parameter_units)

# Get parameters to alter during fitting.
# The result is an ordered mapping from names to initial values
parameters = spec.find_parameters(protocol.model)
# Depending on whether we combine this with the next step or not, this can
# either be a `spec` method or a method implemented here (using
# `spec.annotations` or something of the sort).

# Get hardcoded prior or boundaries
prior = spec.get_hardcoded_boundaries(protocol.model, parameters)
# This needs to:
#  1. Search the protocol-modified model for any parameters that declare the
#     desired annotations.
#  2. Check that those parameters can be combined into a multivariate prior, if
#     desired. I.e. any "a" parameter needs a "b" parameter if we're using
#     "exp2" rates. This will require some inspection of the model equations.
# Considerations:
#  - If we do this _after_ applying the protocol, all variables that are not
#    required to calculate the protocol outputs will have been removed. This is
#    useful if we have an AP model with 10 currents in exp2 form, but only one
#    that we want to fit to.
```

```
#  - This could be an issue if we have a rate "k=a" (explicit lack of V
#    dependence) but want a "b" so that an "a-b" pair can be formed. Users
#    can't hack this in in this case, since the b would have been removed by
#    this stage.
#  - But the first part is probably very desirable? So maybe the pairing needs
#    to be lenient and create a slightly different prior if a param is missing?
# PINTS implementation
#  Once the param pairs are known, and a single conductance has been identified
#  (in case of the initial/default exp2-rate ion current spec), a prior can be
#  created. This shouldn't be hard, e.g., these would exist (hardcoded):
#
#  class RateBoundary(pints.Boundary):
#    def n_parameters(self):
#      return 2
#    ...
#  class ConductanceBoundary(pints.Boundary):
#   def n_parameters(self):
#     return 1
#   ...
#
#  so that the spec code would just do
#   priors = [RateBoundary() for pair in rate_parameter_pairs]
#   priors.append(ConductanceBoundary(min_conductance))
#   return CompositeBoundary(priors)
#
#  (I've noticed we don't actually have a CompositeBoundary in PINTS, although
#   we have the equivalent code for LogPrior, which is easily ported,
#   https://github.com/pints-team/pints/issues/1209 .)
#
# Finally, if we anticipate inference rather than just fitting, we can actually
# create a pints.LogPrior here, and convert it to a pints.Boundaries using
# `boundaries = pints.LogPDFBoundaries(log_prior)` if needed.

# Get output being fit to
output_name = spec.find_output(protocol)
# Again: This can be either part of spec, or done locally using info from spec.
# Note that this is a protocol output, not a model variable.
# Another question: I've assumed here that the units of the output are set by
# the protocol. So then we have input-parameters=set-in-spec, but
# output=set-in-protocol. So slight inconsistency? Is that OK?

# Create stats model for PINTS
class StatsModel(pints.ForwardModel):
    # Might want an __init__ here so that we don't reference global properties
    # later.
    def n_parameters():
        return len(parameters)
    def n_outputs(self):
        return 1    # Can extend to multi-output later
    def simulate(self, parameter_values, times):

        # Pass parameters to Protocol
        for parameter, value in zip(parameters, parameter_values):
```

```
            protocol.set_model_variable(parameter, value)
            # Method doesn't exist yet. See
            # https://github.com/ModellingWebLab/weblab-fc/issues/213

        # Ignore times, these are fixed in protocol and known to match the data

        # Run protocol
        protocol.run(verbose=False, write_out=False)
        # I'm disabling verbose output here because I don't see us using a
        # centralised tool like WL for day to day fitting. Debugging would be
        # hard? So I'm assuming for the moment that we've tested offline and
        # don't need elaborate output here.

        # Get output and return
        return protocol.output_env.look_up(output_name).array

# Define optimisation/inference problem
times, values = front_end.get_data_set_as_numpy_arrays(spec, data_set)
# The above step would involve converting data to the units specified by the
# protocol linked to the spec.
stats_model = StatsModel()
problem = pints.SingleOutputProblem(stats_model, times, values)
# Notes:
# 1. As above, times are ignored here, assumed the same as those generated by
#    protocol. We could do a single protocol run with the default parameters to
#    check this, if we liked.
# 2. This would become pints.MultiOutputProblem when fitting to multi-varied
#    data (model then has n_outputs > 1), but is otherwise the same!

# Define error measure
error = spec.create_error_measure(problem)
# Where spec does e.g. `return pints.MeanSquaredError(problem)`.
# For inference, this becomes e.g. `return pints.GaussianLogLikelihood` (which
# should maybe be the standard, as PDFs can also be passed to a PINTS optimiser
# (they will automatically be maximised, instead of minimised like an error).

#
# Run optimisation
#

# Starting point
x0 = np.array(parameters.values())
# This can be something nicer, e.g. `x0 = boundaries.sample(n=1)[0]`, which is
# supported for pints boundaries (but we'd need to implement that for the rate
# boundary above).
# However, I'd advocate for repeated fits here anyway, which requires some
# extra infra I'll discuss below.

method = pints.CMAES    # Room to customise / get from spec
opt = pints.OptimisationController(error, x0, boundaries, method=method)

log_file = 'optimisation_output.txt'    # Room to customise / get from spec
opt.set_log_to_screen(False)
```

```
opt.set_log_to_file(log_file, csv=False)

opt.set_max_unchanged_iterations(200, threshold=1e-11) # Room to customise/spec
opt.set_max_iterations(None)                           # Room to customise/spec

n_cores = 4  # Room to customise / get from spec
opt.set_parallel(n_cores)

xbest, fbest = opt.run()

# Instead of the above, it could be good to do repeated fits by default, and
# bake this into the WL infrastructure.
# The four-ways-of-fitting code had lots of nice methods for this, e.g. to
# store parameters from a single run, load/save them etc., and I did a tidied
# up version for the fitting tutorial (link).

# so we can probably mine all that to set up something nicer here.
# Not directly relevant for fitting spec design though: I think we can easily
# tack this on later without changing the core functionality.
#
# We'll also want a try-catch block, something to catch numpy warnings, etc.
# (Fitting tutorial has some info on that too)

# We now have xbest: an array of parameter values in the units specified by
# the fitting spec

# Map parameters to values
obtained_parameter_values = dict(zip(parameters.keys(), xbest))

# Return parameters to user
front_end.return_to_user(xbest)

# As discussed, we may also want to generate an updated model here! Which could
# look something like
output_model = non_destructive_cellml_tool.load_model(model_file)
for parameter, value in obtained_parameter_values.items():

    current_unit = parameter_units(parameter)
    desired_unit = cellmlmanip.parse_unit(
        parameter.get_annotation('original_unit'))
    # (I'm inventing some cellmlmanip here, but this is definitely possible)
    # (Also current_unit will be a protocol unit, rather than a cellmlmanip one,
    # so would need some extra bits here. And parameter_units might have variables
    # from a different model object, but all very fixable!)

    # Get value in correct unit
    if current_unit != desired_unit:
        value = cellmlmanip.convert_quantity(value, current_unit, desired_unit)

    # Modify original XML serialisation in minimally invasive manner
    variable = output_model.get_variable(
        parameter.get_annotation('original_component'),
        parameter.get_annotation('original_name'))
```

```
    variable.set_value_or_initial_value(value)

output_model.write_to_disk_with_minimal_changes(front_end.some_file_name())
# Note that this is not something cellmlmanip can do (by design!), but again
# not super relevant for designing the fitting spec I think.
```

Inference Problem Specification - Draft 2

Trying again but simpler and more in line with existing front-end work.

Data set

- Is linked to a single protocol
- Consists entirely of “column” time-series data
- All columns define units
- All columns in data set are mapped to a protocol output (this happens in the data set)
    - Later, allow mapping columns to protocol inputs
- The fitting spec decides what protocol outputs (and hence mapped dataset columns) to fit to, and how

Protocol

- Its outputs must have (and already do have) units.
- Only the ones referenced by the fitting spec (and hence matched to data columns) will be used when fitting.

Future work:

- Add units to protocol inputs, allow the fitting spec to set e.g. time sampling points, temperature, or fixed [K]i and [K]o.
- Don’t calculate (or calculate dependencies for) unmatched protocol outputs.
- **Maybe:** Allow some data columns to be used as vector inputs, but this requires semantics to define discontinuities and/or interpolation methods in data sets (e.g. voltage steps).

Model

- The parameters come from the model, *not* the protocol.
- Parameters and rates are annotated. Suggested names: “exp2_V_positive_rate”, “exp2_V_negative_rate”, “exp2_a_parameter”, “exp2_b_parameter”. Conductances and permeabilities will need to be in a category e.g. “maximum_conductance_or_permability”
- Any model compatible with the protocol specified by the data set selected for fitting, is deemed compatible with the fitting spec.
- Once fitting begins, the model is manipulated by the protocol class. *After that*, the search for applicable parameters begins.
- When output is shown, applicable parameters are indicated using their original model name. We might have to think of a nice way to arrange this, so that they’re not called “potassium_current_h_gate$p_1_converted” or something like that. We’ll also have to carefully include the units, as these might have been changed from the original model formulation.

Fitting spec

- Is (initially) fixed to a single protocol (and hence any linked data set), but can be used with various models (anything compatible with the protocol and having the parameters being fit).
- User chooses a hard-coded rate equation type from a drop-down. Currently there’s exactly one type, called “exp2_rates”.
- When fitting starts, the protocol chosen is applied to the selected model (so the model equations are manipulated). Once this is finished, applicable parameters are searched for.

In the exp2 case, we get a list of v-pos, v-neg, alpha, beta, and conductance/permeability parameters. Hard-coded magic inside the exp2 class uses cellmlmanip to create pairings of (alpha, beta, pos/neg) parameters, from these it creates a set of 2-dimensional priors (which can be sampled from). Similarly, a single conductance_or_permeability parameter is found, and given a univariate prior (which can be sampled from). Upper and lower bounds are hardcoded.
    - Start with just a hardcoded prior.
    - Later we can allow the user to write Python code against a defined API to calculate other priors.
- Now we can fit, and get some sort of output
    - Initially single protocol output compared to the single mapped dataset column via a hardcoded metric.
- Parameter values (point estimates, not distributions) are passed back to the user, somehow (see model section). To be in exported CellML it will have to be one parameter set, but we should allow the fitting spec to output other named files like an MCMC chain too (like when protocols define their outputs).
  My major concern here is we now have parameters for a manipulated model. If I were using this, I would want to see the exact equations used in fitting, so we'd have to do something with cellmlmanip to output the final (post-protocol modifications) model.
- The *implementation* of fitting specs can provide any files it likes back to the front-end in a COMBINE Archive. Some metadata files [link to docs] allow the front-end to interpret what data makes sense to plot and how.
- Finally, the user will want to update the original, unprocessed CellML model. So it'd make a lot of sense if the final parameters we had were in terms of the unmodified model. Not sure what the best way to go about this would be.

Future work:

- The lower and upper bound for the conductance could be set based on a value in the data.

What is needed on the back-end?

- Update cellmlmanip to track the original name & units for converted variables - Michael
- Update protocol analysis to report on (inputs and) outputs - Jonathan
- Start with a single hardcoded fitting run with fixed model, protocol & dataset - Michael
    - See discussion on multivariate prior for the "Oi WL" approach
- Work backwards from hardcoded version to develop a fitting spec spec
- Write a fitting spec parser (part of weblab-fc or building on it) - Michael
    - With an analysis task to check syntax and inform the front-end what parameters, protocol output(s), etc. the fitting spec works with
- Write a fitting spec runner (using weblab-fc and pints) - Michael

What is needed on the front-end?

1. Adding fitting specifications: done if the spec is just a (collection of) file(s).
2. Mapping dataset columns to protocols - Helen (and Jonathan)
3. Annotating models with the extra parameter tags - Jonathan
    a. ~~Adding these tags to the ontology~~
    b. This will be needed to make any progress on generalising the fitting implementation, so should be done early. Michael/Chon will provide some sample models to test with.
4. Running a fitting experiment and producing results - Helen
    a. Underlying bits - Helen + Jonathan

    b. More on the UI - Helen
5. Comparing results with each other & data: #131, #135 - Helen
    a. Includes the graph overlays, and matrix-style displays of available results
    b. The main matrix needed is model vs protocol+data, from a single fitting spec, see User Interface section below

Inference Problem Specification - First draft

Fitting-spec project issues:

- https://github.com/ModellingWebLab/project_issues/issues/60
- https://github.com/ModellingWebLab/project_issues/issues/74

Fitting UI project issues:

- https://github.com/ModellingWebLab/project_issues/issues/65

In this document I describe what we need from a model, FC protocol, and data set to do inference. This leads to proposals for fitting syntax and UI.

What we need from a data set

```
(variable units)+
```

here `variable` is an annotation for a specific variable (an "is" annotation),
`units` is the units the variable is in, and the `+` indicates we need one or more of these.

What we need from a protocol

```
(output variable units)+
(input variable units)*
```

Again, variable is an annotation for a specific variable in the given units.

Examples of outputs are: a vector of currents, or a vector of voltages for AP model fitting.
Examples of inputs are: a vector of voltages for data clamp, a scalar value for internal/external concentrations, and a vector of times to log at.

Allowing a vector of logging times to come from the data ensures that the simulation logs at the sample points, and also takes care of capacitance filtering.

InferenceProblem

```
(output variable units)+
(input variable units)*
(parameter annotation units min max transform=linear)+
(constraint annotation units min max)*
```

Here the outputs are what's fitted to, the inputs are parts of the data set to pass onto the protocol as input.

Parameters are defined either as specific variables (`is` annotations) or variable types (`isVersionOf` annotations). This allows flexibility of models, e.g. all parameters that are "a-type rate coefficient parameters".
In three years of PINTS nobody has asked for anything beyond a rectangular prior - even though we've implemented it, allowed fancy stuff, etc. So I think the "agile" thing to do would be to start with this and wait until somebody demands more.

Constraints are on calculated variables, rather than parameters.
Constraint annotations are to specific variables or to types, e.g. we can add restrictions on all rate coefficients.

Implementation: Sampling starting points

The following routine could be used to find starting points:

1. Transform all parameter boundaries, and store the results as vectors q_min and q_max.
2. Repeatedly sample q ~ U(q_min, q_max), untransform to obtain x, and check x against the constraints.
3. If a point meeting the constraints is found, use it as a starting point. If no point is found in under 100 tries, complain to the user about the chosen constraints.

Detail 1: Advanced parameter finding

The syntax above allows flexibility in model types/structures.

It *might* also let us isolate an ion channel model from an AP model, but this would require some advanced trickery:

1. Figure out which model variables will be clamped / set constant by the protocol.
2. If all concentrations are fixed (to model defaults or experimental conditions), and if voltage is clamped by the protocol, then the ODEs become mostly decoupled.
3. Work back from the outputs to remove all unneeded equations (as is already done in the Protocol class)
4. Only search for parameters within the remaining equations

Problem: Multivariate prior

To implement Kylie's case with the above method, we would ideally define boundaries on all a-type parameters (which is possible using the boundary syntax above and by specifying the type "a parameters" instead of directly choosing the variables), on all b-type parameters (same), but also a minimum and maximum on the "maximum rate over the physiological range", i.e. we have a constraint on k=a+b*60 for positive rates, and on k=a+b*120 for negative rates. There are some complications in setting this up:

1. First we need to know if it's a positive or a negative rate, which is information we could get from an extra annotation (positive rate type and negative rate type)
2. Next, we need to specify some expression to evaluate.
    a. If we say "this is a constraint on an a,b pair" then we need some way for the web lab to match up the as and bs. You can do this from the equations if you know what they look like, but I can't see how you'd do it in a generalisable way.
    b. It looks more promising to me to define a constraint on the rate directly (which can then be evaluated using the full set of parameters). But then you have the problem of needing to substitute V into the equation - and how do we specify that?

So I can see no easy / nice way of implementing this!
We might do an initial version without the whole constraint mechanism, and see if we can't use that?

A while back we discussed using annotations such as:

- isVersionOf "a rate parameter"
- isVersionOf "b rate parameter"

- isVersionOf “forward rate”
- isVersionOf “backward rate”

And then using the equation analysis features of cellmlmanip to pull out e.g. all forward rates appearing in `get_equations_for(iKr)`, and then for each of those find the a & b rate parameter pair in its defining equation(s). Would that work?

Detail 2: Using part of the data as inputs

I’m quite tempted to allow part of the data to be used as inputs to the protocol, e.g. to ensure both use exactly the same logging points in time. But maybe this is an unnecessary complication?

For example, we get into some difficulties trying to use a voltage signal from the data as input to the protocol if it contains discontinuities. The WL/FC way to deal with discontinuities is to specify a time point twice, e.g. [(0, 0), (1, 0), (1, 1), (2, 1)] creates a discontinuous step at t=1, while [(0, 0), (1, 1), (2, 1)] creates a ramp between t=0 and t=1 due to the built-in interpolation. It’s unlikely anybody will have the required format in their data set.

Then again, it would be cool to be able to set e.g. internal/external concentrations via the data set, instead of having to make a specific protocol for each?
(Although doing so wouldn’t be much work: you’d just create 1 protocol with an input and then several protocols that called that 1 protocol internally).

If we remove this we can drop the “input” parts above.

Detail 3: Multiple voltage protocols / fitting to (V, Cai)

Nothing in the above spec stops a data set from having multiple time-series: e.g. (t1, I_pr1) and (t2, I_pr2); or (t1, V) and (t1, Cai).

To create FC protocols that execute multiple voltage protocols, I think we could define 4 FC protocols, and then a 5th that imports and runs all four, and provides access to all their output. Is that right, Jonathan?

To combine the output of multiple variables, we could update the output spec to:
`(output variable units error_measure=root_mean_square weight=1)+`
so that users can define an error measure and a weighting for the resulting error, for each variable. The final error would be E(p) = sum(w_i * E_i(p)).

Detail 4: Advanced post-processing

“Method 2” (fitting to time constants & steady state values) is objectively rubbish and should never be used again.
If people reaaaally want to, they can write post-processing in an FC protocol, and then combine a few FC protocols to get a full data set.
There is still an open ticket to eventually allow Python post-processing in an FC protocol, which could be useful here too.

User interface

Each `InferenceProblem` would have its own matrix page.

Each row would be a protocol+data combination, while each column would be a model.

This would allow us to compare:

- different models, fit to the same data (“which model fits data X best”)
- the same model fit to several data set (“model X fit to cells 1-9”)
- the same model fit to several data/protocol combinations (“model X fit to cell 1 data from method 3 and method 4"

See also https://github.com/modellingweblab/weblab/issues/135.

In addition, I really like the idea of a user having to explicitly say “FC protocol X generates something that I believe is compatible with data set Y”, without disallowing any other protocol to explain the same data etc.

Each square in the matrix would show the latest “version” of a fit.
Because running fits takes a while, I imagine just clicking shouldn’t start/restart a fit, but instead some “(re-)run fit” button would have to be available on the page showing the version. If we really wanted to, we could let users add some details when creating a new version of a fit (e.g. optimiser method).

Generating fitting code.

With the above all in place, we could run a PINTS optimisation, without any code generation.

Steps would include:

1. Determining the parameter and constraint variables in the used model.
2. Creating an instance of an fc.Protocol, with the correct model already set, and code generated and compiled. Tight solver tolerances would need to be set.
3. Creating some kind of Transformation class, that could transfer forwards and backwards.
4. Creating an instance of a pints.Boundaries class that included parameter boundaries and constraints, and had a link to the Transformation. This class would also provide a sample() method for starting points.
5. Creating an instance of a pints.ForwardModel class that took the fc.Protocol and Transformation as input.
6. Setting up a pints.MultiOutputProblem with the required outputs, links to the data etc.
7. Running repeated fits (30?) from random starting points, with CMAES.

Conclusion 1

Based on the above, here are three proposals (I think the first is easiest to implement):

1. Add an optional “fitting” section to fc.Protocol. The Protocol class is already equipped to do the parameter finding etc. that’s required.
2. Like the above, but with an InferenceProblem as a separate entity, defined in much the same way as a regular protocol.
3. Python-based version, see below

Python version

To give users more freedom, we could do define a Python interface called `InferenceProblem`, that is required to provide five lists:

```
(protocol filename)+
(output variable units)+
(input variable units)*
(parameter annotation)+
(constraint annotation)*
```

and a method:

```
fit(self, protocols, output_variables, input_variables,
parameter_variables, constraint_variables)
```

where protocols would be a list of instances of fc.Protocol (or a simplified wrapper), that let users do things like change the parameters (via the variable names in parameter_variables), run simulations and get the outputs (via the variable name in output_variables).

We could drop constraint annotations too, if we just expect users to re-calculate the constraints on intermediate parameters separately. (This is the current strategy in 4-ways and in Kylie's code).

The fit method would do lots of stuff, including setting up transformations, combining the output of multiple protocols, choosing the fitting method etc.

Extra UI work would be needed for the Python version (allowing script to be uploaded), and we'd need some instance of it in the WL, with code to read a PythonFittingSpec and check if it's compatible with models etc.,

Conclusion 2

I think option 1 would be the least work, and fits most naturally with what we already have. It would also be easier for users, I guess, and would stop them from trying to hack things into the python code that could go in the protocol.

A hybrid of option 1 and 3 would also be possible, where we implement option 1 but leave option 3 as a thing to add later, if we find it's required. This would be similar to the current idea of allowing post-processing to happen in pure Python at some point.