A Practical Guide for the Efficient Formulation and Calibration of Large, Energy Rule-Based Models of Cellular Signal Transduction

Fabian Fröhlich [1]

[1]Department of Systems Biology, Harvard Medical School, Boston, MA 02115, USA,

Abstract

Aberrant signal transduction leads to complex diseases such as cancer. To rationally design treatment strategies with small molecule inhibitors, computational models have to be employed. Energy- and rule-based models allow the construction of mechanistic ordinary differential equation models based on structural insights. The detailed, energy-based decription often generates large models, which are difficult to calibrate on experimental data. In this chapter, we provide a detailed, interactive protocol for the programmatic formulation and calibration of such large, energy- and rule-based models of cellular signal transduction based on an example model describing the action of RAF inhibitors on MAPK signaling. An interactive version of this chapter is available as jupyter notebook at github.com/FFroehlich/energy_modeling_chapter.

1 Introduction

Cells use signal transduction to respond to extracellular cues, controlling a variety of different cellular processes such as proliferation, development and environmental adaption [1]. Signal transduction is initiated by the binding of extracellular ligands to transmembrane receptors [2] and then propagated through phosphorylation cascades [3, 4], ultimately controlling gene expression through modulation of transcription factor activity. Mutations and changes in protein expression levels can dysregulate signal transduction leading to a variety of different diseases such as cancer [5]. Small molecule inhibitors can be used to target specific proteins that participate in aberrant signal transduction [6]. However, feedback mechanisms and cross-talk between signaling cascades, which are ubiquitous to cellular signal transduction, lead to complex, non-linear responses to drug inhibition [7], making it difficult to anticipate the phenotypic response or resistance to these inhibitors. Accordingly, computational models that describe signal transduction, and the effect of drug perturbations on it, are required for the rational selection and dosage of targeted inhibitors [8]. Mechanistic computational models of cellular signal transduction translate biological knowledge into executable mathematical formulations [9]. To this end, ordinary differential equations (ODE) models are a popular modeling approach, as they can describe temporal dynamics and feedback mechanisms [10]. Moreover, state variables and parameters correspond to biophysical entities, facilitating a direct interpretation of model simulations and predictions.

Formulation of such models is often a long, labour intensive process, especially for large models. Instead of directly writing model equations, a tedious and error-prone process, it is advisable to use some kind of abstraction when writing models. For example, the Systems Biology Markup Language (SBML) [11] encodes models using concepts such as reactions and events, which can then be translated into differential equation models. Rule-based formats such as Kappa [12] or BioNetGen [13] employ even higher levels of abstractions and describe model equations in terms of rules, which are generalizations of reactions [14]. While such abstractions can limit what models can be expressed, they also permit a more compact model description. A recent extension to rule-based models are energy rule-based models [15, 16], which provide an additional layer of abstraction by enabling the specification of reaction rates in terms of energies. Conceptually, energies permit the theoretically sound formulation of reaction rates based on transition theory and provide a rigorous way of specifying context dependent reaction rates based on structural insights [17, 18]. Practically, energy based models enforce thermodynamic constraints such as the Wegscheider-Lewis cycle conditions [19], which ensure the conservation of energy in oligomerization processes. This is particularly relevant for the specification of context dependent inhibitor affinities. For example, many RAF inhibitors have different affinities for RAF monomers and dimers, which causes for the clinically observed phenomenon of "paradoxical activation" in which signaling in RAF mutant cancer cells in inhibited, but signaling in RAS mutant cancer cells is amplified [20, 21, 22].

Calibration of large models can also be challenging. During model calibration, parameter values are tuned such that model simulations agree with experimental data, which improves the predictive performance of models. Agreement between model simulations and experiments can be quantified using likelihood-based objective functions, in which case model calibration can be implemented through minimization of the respective objective function. For larger models, gradient-based optimization methods perform well [23]. However, for non-linear models, analytical solutions to model equations are rarely available, which means that numerical integration is necessary for evaluation of the objective function and its derivatives. Accordingly, gradient-based methods not only require integration of model equations, but also sensitivity equations, which quantify the dependence of simulations on model parameters. For large models, adjoint sensitivity analysis is an effective way of computing these sensitivities, but also comes with its drawbacks such as not permitting the computation of cheap approximations to the second order derivative [24].

Model formulation and calibration for large models can be a daunting task [25], even when using specialized software tools. Input formats such as SBML or PEtab [26] permit the standardized problem formulation, which allows the automation of the majority of steps. However, manual interventions are still occasionally necessary to achieve optimal performance. These manual interventions often require substantial practical experience as well as theoretical background knowledge, which creates a high barrier to entry and means that successful development and application of large models is rarely possible without guidance from experts. To rectify this situation, we here provide a detailed step-by-step protocol for the calibration of a large scale model introduced in [27] and [28]. In contrast to previous protocols for ODE model calibrations [29], this protocol includes a description of energy-based model formulation, focuses on large-scale models of intracellular signaling and includes an interactive jupyter notebook which allows reproduction of all results. As in other protocols, theoretical background is provided when appropriate. For additional analysis steps including identifiability, uncertainty analysis or model selection, users should consult previously published reviews and protocols . This protocol uses the python toolboxes PySB , PEtab , AMICI , pyPESTO and fides and is accompanied by a jupyter notebook that permits the interactive reproduction of results.

2 Mathematical Problem Formulation

For ODE models, the temporal evolution of abundances of n_x different molecular species x_i is determined by the vector field $\mathbf{f}$ and the initial conditions $\mathbf{x}_0$:

$$\dot{\mathbf{x}} = \mathbf{f}(t, \mathbf{x}, \boldsymbol{\gamma}), \quad \mathbf{x}(t_0) = \mathbf{x}_0(\boldsymbol{\gamma}). \tag{1}$$

Both $\mathbf{f}$ and $\mathbf{x}_0$ may depend on dynamic parameters $\boldsymbol{\gamma}$, which may include kinetic parameters such as catalytic or binding rates or condition parameters such as ligand or drug concentrations.

Calibration is performed by comparing model simulations, i.e., solutions $\mathbf{x}$ that satisfy (1), to experimental data $\bar{\mathbf{y}}$. As direct measurement of $\mathbf{x}$ is usually experimentally not possible, observables

$$\mathbf{y}(t, \boldsymbol{\gamma}, \boldsymbol{\xi}) = \mathbf{g}(\mathbf{x}(t, \boldsymbol{\gamma}), \boldsymbol{\xi}) \tag{2}$$

are introduced, which may depend on abundances $\mathbf{x}$ as well as observable parameters $\boldsymbol{\xi}$ such as scaling parameters or offsets. The agreement between model observables $\mathbf{y}$ and experimental measurements $\bar{\mathbf{y}}$ can be quantified according to the assumed noise model. A common assumption is that the measurement noise for n_y observables y_i is additive and independent, normally distributed:

$$\bar{y}_i = y_i(t, \boldsymbol{\gamma}, \boldsymbol{\xi}) + \epsilon_i \quad \epsilon_i \overset{id}{\sim} \mathcal{N}(0, \sigma_i^2(\boldsymbol{\psi})), \tag{3}$$

where $\boldsymbol{\psi}$ are noise parameters that encode the standard deviations $\boldsymbol{\sigma}$. Using this noise model, the negative log-likelihood

$$J(\boldsymbol{\theta}) = \frac{1}{2} \sum_{i=k}^{n_k} \sum_{(i,j) \in T_k} \log\left(2\pi\sigma_i^2(\boldsymbol{\Psi}_{ijk}(\boldsymbol{\theta}))\right) + \left(\frac{\bar{y}_{ijk} - y_i(t_j, \boldsymbol{\Gamma}_k(\boldsymbol{\theta}), \boldsymbol{\Xi}_{ijk}(\boldsymbol{\theta}))}{\sigma_i(\boldsymbol{\Psi}_{ijk}(\boldsymbol{\theta}))}\right)^2 \tag{4}$$

statistically rigorously quantifies the agreement between experimental data and model simulations, where $\boldsymbol{\theta}$ are the free, unknown parameters, T_k is the set of timepoints t_j and observable y_i index combinations that were measured in the k-th out of n_k experimental conditions, $\boldsymbol{\Psi}_{ijk}$ is a function that maps free parameters $\boldsymbol{\theta}$ to observable, timepoint and condition specific noise parameters $\boldsymbol{\psi}$, $\boldsymbol{\Xi}_{ijk}$ is a function that maps free parameters $\boldsymbol{\theta}$ to observable, timepoint and condition specific observable parameters $\boldsymbol{\xi}$ and $\boldsymbol{\Gamma}_k$ is a function that maps free parameters $\boldsymbol{\theta}$ to condition specific dynamic parameters $\boldsymbol{\gamma}$. As the logarithm is a strictly monotonously increasing function, the minimization of $J(\boldsymbol{\theta})$ is equivalent to the maximization of the likelihood. Therefore, the corresponding minimization problem

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta} \in \Theta} J(\boldsymbol{\theta}), \tag{5}$$

will infer the Maximum Likelihood Estimate $\boldsymbol{\theta}^*$ for free parameters $\boldsymbol{\theta}$, where the search domain $\Theta \subset \mathbb{R}^{n_\theta}$ restricts the $\boldsymbol{\theta}$ to values that are biologically plausible and are less subject to numerical integration failures. For most problems, Θ is chosen as tensor product of scalar search domains $(\mathcal{L}_l, \mathcal{U}_l)$ with lower and upper boundaries $\mathcal{L}_l < \mathcal{U}_l$ and $\mathcal{L}_l, \mathcal{U}_l \in \mathbb{R} \cup \{-\infty, \infty\}$ for every parameter θ_l.

3 Formulating a Thermodynamic Model of RAF inhibition in PySB

In the following, we will specify the model, which in turn defines the right hand side $\mathbf{f}$ and the initial conditions $\mathbf{x}_0$ of the model equations. We will specify the model using the rule-based modeling framework PySB, which instead of direct specification of $\mathbf{f}$, programmatically encodes model structure in a modular, object oriented way, ensuring reusability, composability and extendability of models [30]. To start model specification, we instantiate a PySB `Model` with name `thermo_raf`.

[1]:
```
from pysb import Model
model = Model('thermo_raf');
```

3.1 Protein Species

As first step of model construction, we have to define all molecules that we want to account for. In PySB, this is done by instantiating the `Monomer` class. Each monomer is initialized by a unique identifier, a list of sites as well as, optionally, a list of possible states for each site. Sites typically correspond to binding domains or residues that can be mutated or modified post-translationally, where the site states encode the different all possible mutations or post-translational modifications for each site. All sites, independent of whether possible states were specified or not, can form a bond with other sites, permitting the description of macromolecular complexes.

Here, we will define monomers `BRAF` and `CRAF` which both feature the interaction sites `RBD` (interaction domain with RAS), `raf` (dimerization domain) and `rafi` (inhibitor binding site). For BRAF we also add a mutation sites for the 600th amino acid, which can take values `V` (valine, wildtype) or `E` (glutamic acid, oncogenic mutation).

[2]:
```
from pysb import Monomer
Monomer('BRAF', ['AA600', 'RBD', 'raf', 'rafi'], {'AA600': ['E', 'V']})
Monomer('CRAF', ['RBD', 'raf', 'rafi']);
```

Next, we will specify initial concentrations. For this model, the concentration of all molecular species will be in μM. However, we will use protein counts per cell derived from absolute proteomics to inform initial conditions. Accordingly, parameters that define initial concentrations have to be transformed from molecule per cell to μM, which is achieved by dividing them by cell volume (here assumed to be $1pL = 10^{-12}L$) and the Avogadro constant (6.02210^{23} molecules) and multiplying with 10^6 to account for the unit prefix μ.

In the following code, we first introduce two PySB `Expression`s that specify the Avogadro constant and cell volume. As with most model components, PySB automatically creates workspace variables with the respective identifier as variable names, simplifying the programmatic reference to components. Next, we introduce two dynamic parameters (γ, see (1)) as PySB `Parameter`s `BRAF_0` and `CRAF_0` that define initial abundances as molecules per cell and convert these abundances to concentrations in μM using the expressions `initBRAF` and `initCRAF`. These expressions are then used to define the initial abundances for the two monomer species `CRAF` and `BRAF` using the PySB `Initial` class, which specifies $\mathbf{x}_0$ for the respective molecular species x_i. In PySB, a molecular species is defined by a pattern, which is created by invoking the respective monomer with the state of each site as keyword argument. For initial conditions, the respective patterns have to be

explicit, i.e., the states of all sites have to specified. For the initial conditions, we specify that all sites are unbound (denoted by `None`), with the exception of the the `AA600` site. For the `AA600` site we have to pick one of the previously specified states, which we do by specifying the oncogenic variant denoted by `E`. For all molecular species without an explicit instantiation of `Initial`, the respective entry in $\mathbf{x}_0$ is set to 0.

[3]:
```
from pysb import Parameter, Expression, Initial
import sympy as sp

# define Avogadro constant and volume as hardcoded expressions
Expression('N_Avogadro', sp.Float(6.02214076e+23))
Expression('volume', sp.Float(1e-12))

# define initial abundance parameters
Parameter('BRAF_0')
Parameter('CRAF_0')

# convert initial abundances to concentrations
Expression('initBRAF', 1000000.0*BRAF_0/(N_Avogadro*volume))
Expression('initCRAF', 1000000.0*CRAF_0/(N_Avogadro*volume))

# define initial molecular species
Initial(BRAF(AA600='E', RBD=None, raf=None, rafi=None), initBRAF)
Initial(CRAF(RBD=None, raf=None, rafi=None), initCRAF);
```

3.2 Protein Interactions

For rule-based models, all dynamic interactions are specified as rules. PySB `Rule`s are generalizations of biochemical reactions and their action is defined by a reactant pattern and a product pattern. When applied to a molecular species, the action of the rule, i.e., the respective biochemical reaction, is implemented by applying the difference between the reactant and the product pattern to a match of the reactant pattern in a molecular species. In contrast to initials, the reactant pattern and product pattern do not have to be explicit and a single rule can define multiple different biochemical reactions, depending in how often and in how many molecular species the reactant pattern occurs.

Non-thermodynamic rules additionally require forward reaction rates k_f and, if applicable, reverse reaction rates k_r. In contrast, thermodynamic rules are specified in terms of activation energy E_a and the phenomenological constant ϕ, the convex combination parameter which encodes how much changes in free energy affect forwards and reverse rates. For thermodynamic rules, the reaction rate also depends on the free energy difference between reactants and products of the reaction. The forward k_f and reverse k_r reaction rates are computed using the Arrhenius theory of reaction rates

$$k_f = \exp(-(E_a + \phi\Delta G)), \quad k_r = \exp(-(E_a + (\phi - 1)\Delta G))$$

with

$$\Delta G = -\sum_{r \in R} \Delta\Delta G_r + \sum_{p \in P} \Delta\Delta G_p$$

where E_a is the activation energy of the reaction, i.e., the free energy difference between the reactant state and the transition state, ΔG is the free energy difference between the reactant and the product states and $\Delta\Delta G_r$ and $\Delta\Delta G_p$ are the free energy modifiers that apply to the reactants r in R and the products p in P respectively. All energies are assumed to be normalized by the inverse of the product between temperature and Boltzmann constant $\frac{1}{RT}$. The $\Delta\Delta G_r$ and $\Delta\Delta G_p$ energies can be specified by using the PySB `EnergyPattern` class. When calculating reaction rates, energy BioNetGen (eBNG) not only accounts for `EnergyPatterns` that apply to the reactant and product patterns specified in the rule, but also those EnergyPatterns that specifically apply to reactant and product species. Thus, reactions generated by thermodynamic rules do not always have the same reaction rate. In contrast, all reactions generated by a single non-thermodynamic rule will have the same reaction rate, unless a local function is used for rate specification.

In the following we will specify the dimerization of RAF molecules. For this purpose, we introduce three parameters: the activation energy for the binding reaction E_a (Ea suffix), the Gibb's free energy of the formed bond ΔG (dG suffix), and the thermodynamic balance parameter ϕ (phi suffix).

[4]:
```
Parameter('bind_RAF_RAF_Ea')
Parameter('bind_RAF_RAF_dG')
Parameter('bind_RAF_RAF_phi');
```

To enforce that the actication energy $E_a = -\log(k_f)$ encodes the forward reaction rate and $\Delta G = -\log(K) = -\log\left(\frac{k_r}{k_f}\right)$ encodes the affinity constant K, we will pass a specially crafted custom activation energy $E_{a0} = -\phi\Delta G - E_a$, for which we create a custom expression. To then construct homo- and heterodimerization rules for all RAF paralogs, we use `itertools` to loop over all combinations of BRAF and CRAF, implementing equal affinities for all homo- and heterodimers. For every rule we pass the custom energy Ea_0 and then specify an `EnergyPattern` for the single product pattern (`pp`) with $\Delta\Delta G_p = \Delta G$.

[5]:
```
from pysb import Rule, EnergyPattern
import itertools as itt

Expression('Ea0_bind_RAF_RAF',
           -bind_RAF_RAF_phi*bind_RAF_RAF_dG - bind_RAF_RAF_Ea);

for RAF1, RAF2 in itt.combinations_with_replacement([BRAF, CRAF], 2):
    pp = RAF1(raf=1) % RAF2(raf=1)
    Rule(f'{RAF1.name}_and_{RAF2.name}_bind_and_dissociate',
         RAF1(raf=None) + RAF2(raf=None) | pp,
         bind_RAF_RAF_phi, Ea0_bind_RAF_RAF, energy=True)
    EnergyPattern(f'ep_bind_{RAF1.name}_{RAF2.name}', pp, bind_RAF_RAF_dG);
```

3.3 RAF inhibitor

Next, we introduce a RAF inhibitor RAFi. We assume that the inhibitor is added to the cell medium at some point and quickly diffuses in and out of cells. As extracellular space is much bigger than the volume of a cell, we can assume an infinite reservoir of molecules in the extracellular compartment and assume that the intracellular inhibitors concentration will be unaffected by intracellular reactions. Accordingly, we specify the respective initial as fixed, which means that the respective molecular species can participate in reactions, but its concentration will remain constant.

[6]:
```
Monomer('RAFi', ['raf'])

Initial(RAFi(raf=None), Parameter('RAFi_0', 0.0), fixed=True);
```

Next, we define binding reactions for RAFi with both BRAF and CRAF, again implementing the same affinities and activation energies for both rules.

[7]:
```
Parameter('bind_RAFi_RAF_Ea')
Parameter('bind_RAFi_RAF_dG')
Parameter('bind_RAFi_RAF_phi')
Expression('Ea0_bind_RAFi_RAF',
           -bind_RAFi_RAF_phi*bind_RAFi_RAF_dG - bind_RAFi_RAF_Ea)

for RAF in [BRAF, CRAF]:
    Rule(f'RAFi_and_{RAF.name}_bind_and_dissociate',
         RAFi(raf=None) + RAF(rafi=None) | RAFi(raf=1) % RAF(rafi=1),
         bind_RAFi_RAF_phi, Ea0_bind_RAFi_RAF, energy=True)
    EnergyPattern(f'ep_bind_{RAF.name}_RAFi',
                  RAFi(raf=1) % RAF(rafi=1), bind_RAFi_RAF_dG);
```

3.4 Paradoxical Activation

RAF inhibitors inhibit signaling for BRAF mutant cancer cells but promote signaling for RAS mutant cancer cells. At the strucural level, this can be rationalized by assuming that RAF inhibitors have higher affinity towards drug-unbound RAF dimers and lower affinity towards drug bound RAF dimers. The symmetry conveyed by energy conservation of molecular binding reactions implies that RAF inhibitors, promote dimerization at low to medium concentrations and incompletely inhibit signaling even at high concentrations, as they have lower affinity to the second protomer in a RAF dimer. As MAPK signaling for RAS mutant cells is mediated by RAF dimers, respective signaling is amplified, leading to increased growth. In the thermodynamic models, this can be implemented by specifying additional `EnergyPatterns` that controls the Gibbs free energy of $RAF_2 - RAFi$ trimers. Note that we do not specify how these trimers are formed, so the change in energy will equally apply to the rates of all reactions that either consume or produce these trimers. In this example, an decrease in energy would equally increase RAF dimerization when exactly one protomer is inhibitor bound and inhibitor binding to the first protomer in a dimer, thereby implementing the previously described symmetry.

[8]:
```
Parameter('ep_RAF_RAF_RAFi_ddG')
for RAF1, RAF2 in itt.product([BRAF, CRAF], repeat=2):
    EnergyPattern(
        f'ep_{RAF1.name}_{RAF2.name}_mod_RAFi_single',
        RAF1(raf=1, rafi=None) % RAF2(raf=1, rafi=2) % RAFi(raf=2),
        ep_RAF_RAF_RAFi_ddG);
```

To implement the lower affinity to the second protomer, we add an additional `EnergyPattern` that changes the Gibbs free energy of $RAF_2 - RAFi_2$ tetramers.

[9]:
```
Parameter('ep_RAF_RAF_RAFi_RAFi_ddG')
for RAF1, RAF2 in itt.combinations_with_replacement([BRAF, CRAF], r=2):
    EnergyPattern(
        f'ep_{RAF1.name}_{RAF2.name}_mod_RAFi_double',
        RAFi(raf=2) % RAF1(raf=1, rafi=2) % RAF2(raf=1, rafi=3) % RAFi(raf=3),
        ep_RAF_RAF_RAFi_RAFi_ddG);
```

Here we note that these `EnergyPatterns` only specify that changes in affinity are permissible in the model. The nature of these changes has to be encoded in their values, where negative values lead to stabilization and positive values to destabilization of respective complexes.

We will now load the remainder of the model from a file. Briefly, the full model describes MEK and ERK phosphorylation downstream as well as EGF stimulatable EGFR signaling upstream of RAF signaling. Moreover, it incorporates negative feedback from ERK on both MAPK and EGFR signaling. A comprehensive description of this part of the model is available in [27] and [28]. However, parts of the model were substantially simplified account for the reduced set of experimental data considered in this protocol. Specifically, description of EGFR signaling is removed and replaced by a phenomenological description, MEK inhibitors were removed from the model and two-step phosphorylation and dephosphorylation reactions were replaced by linear one-step reactions.

[10]:
```
from pysb import Observable, ANY
def extend_model():
   Content of this function is only included in the interactive notebook.

extend_model();
```

4 Importing data in PEtab Format

With the model at hand, the next step for model calibration is specification of the objective function (4). We will specify the objective function using PEtab [26], which simplifies the definition of multiple experimental conditions. PEtab specification of a calibration problem consists of the tables describing model observables, experimental measurements, experimental conditions and model parameters. Additionally, tables describing the visualization of simulations and data may also be included.

4.1 Observables

We will start the PEtab definition by specifying the model observables $\mathbf{y}$ (see (2)). Observables define the the model quantities that were measured experimentally, here pMEK and pERK. First, we add respective observables to the model using PySB `Observable`s:

[11]:
```
Observable('pMEK_obs', MEK(phospho='p'))
Observable('pERK_obs', ERK(phospho='p'));
```

These observables quantify all MEK (MAP2K1 + MAP2K2) and ERK (MAPK1 + MAPK3) molecules that are phosphorylated on the site `phospho`, which accounts for phosphorylation on S218/S222 or S222/S226 on MEK and T185/Y187 or T202/Y204 on ERK. These PySB `Observable`s define linear sums of model species $\sum_i w_i x_i$, where w_i is an integer that quantifies how often the specified pattern matches the species x_i.

Next, we construct the PEtab `observables table`. The model quantifies pMEK and pERK in concentrations, but measurements are noise corrupted, measured in fluorescence intensity and also include background fluorescence. As the scaling between concentrations and intensity and the amount of background signal is unknown, we include scaling and offset parameters in the PEtab observable definition `petab.OBSERVABLE_FORMULA`, which specifies the observable function g_i (see (2)). To account for noise corruption, we specify a single noise parameter ψ_i as PEtab noise formula `petab.NOISE_FORMULA`, which corresponds to a Gaussian distribution (default when no `petab.NOISE_DISTRIBUTION` is specified), with the respective parameter as standard deviation σ_i (see (3)). The prefix `noiseParameter1` indicates that the value of the parameter will be provided in the respective column of the `measurement table`, which permits the specification of timepoint and condition specific noise levels via Ψ_{ijk} (see (4)). Here, the specified observable formula g_i does not include any explicit PEtab noise parameters. However, the scale and offset parameters for each observable could be replaced by `observableParameter[0-9]+` placeholders, which would enable the designation of timepoint and condition specific values in the `measurement table`.

[12]:
```
import petab
import pandas as pd
observables_table = pd.DataFrame([
    {
        petab.OBSERVABLE_ID: obs_id,
        petab.OBSERVABLE_FORMULA:
            f'{obs_id}_scale*{obs_id}_obs + {obs_id}_offset',
        petab.NOISE_FORMULA: f'noiseParameter1_{obs_id}'
    } for obs_id in ['pMEK', 'pERK']
]).set_index(petab.OBSERVABLE_ID)
```

4.2 Measurements and Conditions

As experimental measurements, we will load one of the datasets from [27]

[13]:
```
import synapseclient as sc
pd.set_option('display.max_rows', 5)
```

```
pd.set_option('display.max_columns', 6)

syn = sc.Synapse()
syn.login(silent=True);
data_df = pd.read_csv(syn.get('syn22804081').path)
```

The data contains the dose response to multiple RAF and MEK inhibitors as well as respective combinations. However, to minimize computation time requirements, we will only consider data involving the two RAFis vemurafenib and dabrafenib. The model itself only contains generic `RAFi` parameters and species, which we can now map to specific inhibitors using experimental conditions. This permits the simultaneous estimation of inhibitor specific kinetic rates in conjunction with all other model parameters. Such a multi-experiment setup can improve parameter identifiability.

In the following, we will simulatenously generate the `condition table`, encoding the condition specific mapping $\mathbf{\Gamma}_k$ (see (4)) of dynamic paramers γ (see (1)), and `measurement table`, ecoding measurements $\bar{\mathbf{y}}_{ijk}$ and, potentially, respective mappings for observable parameters $\boldsymbol{\xi}$ (see (2)) via $\mathbf{\Xi}_{ijk}$ (see (4)) and noise parameters $\boldsymbol{\psi}$ (see (3)) via $\mathbf{\Psi}_{ijk}$ (see (4)). The `condition table` describes the necessary information about experimentel conditions, i.e., what drug was used and what concentrations were applied. The `measurement table` describes the measurement itself as well as additional information, such as under which experimental conditions the data was measured.

The data we loaded above is in a matrix format, but PEtab requires the data in a long format. Thus, additional processing is necessary. Briefly, the code below extracts all data points and sets what model observable `petab.OBSERVABLE_ID` they belong to, the value of the measurement `petab.MEASUREMENT`, the time of measurement `petab.TIME`, the noise parameter `petab.NOISE_PARAMETERS` and information about the experimental condition.

For the EGF stimulation condition, it is important to consider the experimental setup, as cells are subjected to 24h of pretreatment with drugs. We here assume that this is sufficient time for the system to reach a steady state. To describe such an experimental setup in PEtab, it is necessary to reference the respective experimental condition in the `condition table` by specifying the respective `petab.CONDITION_ID` in the `petab.PREEQUILIBRATION_CONDITION_ID` column. The condition for the actual experiment can then be specified in the `petab.SIMULATION_CONDITION_ID`.

To enable the visualization of data and fitting results, we here also specify the `petab.DATASET_ID`. This does not influence the actual fitting, but serves as grouping identifier during visualization.

[14]:
```
RAFis = ['Vemurafenib', 'Dabrafenib', 'PLX8394', 'LY3009120', 'AZ_628']
MEKis = ['Cobimetinib', 'Trametinib', 'Selumetinib', 'Binimetinib', 'PD0325901']


conditions = dict()


def format_petab(row):
    suffixes = [''] + [f'.{idx}' for idx in range(1,10)]
    datapoints = []
    if row[MEKis].any():
```

```
        return datapoints
    # loop over columns of the data matrix
    for suffix in suffixes:
        # extract data
        datapoint = {
            petab.OBSERVABLE_ID: 'pMEK' if row.pMEK else 'pERK',
            petab.MEASUREMENT: row[f'Mean{suffix}'],
            petab.TIME: row['Time_EGF'],
            petab.NOISE_PARAMETERS: row[f'Std{suffix}'],
        }
        # extract condition information
        if (row[RAFis] != 0).sum() != 1 or \
                ((row['Vemurafenib'] == 0) and (row['Dabrafenib'] == 0)):
            continue

        # find first nonzero
        rafi = RAFis[(row[RAFis] != 0).argmax()]

        # extract drug concentration
        drug_conc = row[f'Concentration (uM){suffix}'] if row[rafi] == -1 \
            else row[rafi]
        # condition id must be sanitzed, must match '^[a-zA-Z]+[\w_]*$'
        drug_str = f'{rafi}_{drug_conc}'.replace('.', '_').replace('-', '_')

        condition_str = drug_str + (
            f'__EGF_{row["EGF"]}' if row['EGF'] > 0 else ''
        )
        condition = {
            petab.CONDITION_ID: condition_str,'RAFi_0': drug_conc,
            'EGF_0': row["EGF"], 'bind_RAFi_RAF_Ea': f'bind_{rafi}_RAF_Ea',
            'bind_RAFi_RAF_dG': f'bind_{rafi}_RAF_dG',
            'ep_RAF_RAF_RAFi_ddG': f'ep_RAF_RAF_{rafi}_ddG',
            'ep_RAF_RAF_RAFi_RAFi_ddG': f'ep_RAF_RAF_{rafi}_{rafi}_ddG'
        }
        # set baseline for datapoint
        datapoint[petab.PREEQUILIBRATION_CONDITION_ID] = drug_str
        # set id for condition and datapoint
        datapoint[petab.SIMULATION_CONDITION_ID] = condition_str
        datapoint[petab.DATASET_ID] =  ('EGF__' if row['EGF'] > 0 \
                                        else 'ctrl__') + rafi
        condition[petab.CONDITION_ID] = condition_str

        datapoints.append(datapoint)
        if condition_str not in conditions:
            conditions[condition_str] = condition

    return datapoints
```

```
measurement_table = pd.DataFrame([
    d
    for ir, row in data_df.iterrows()
    for d in format_petab(row)
])
condition_table = pd.DataFrame(conditions.values()).set_index(petab.
 ↪CONDITION_ID);
```

In the code above, we extracted experimentally measured standard deviations, which can be noisy as they are often computed from a small number of biological or technical replicates. As the magnitude of standard deviations determines the importance of datapoints during calibration, outliers can lead to overemphasis or ignorance of individual datapoints, resulting in poor fits to the overall data. To avoid such issues, we use the same averaged standard deviations for all datapoints of each observable. For averaging we compute the root mean square, as variances are additive, but standard deviations are not.

[15]:
```
import numpy as np
for group, frame in measurement_table.groupby(petab.OBSERVABLE_ID):
    measurement_table.loc[measurement_table[petab.OBSERVABLE_ID]==group,
                          petab.NOISE_PARAMETERS] = \
        np.sqrt(frame[petab.NOISE_PARAMETERS].apply(np.square).mean())
measurement_table;
```

4.3 Parameters

The last table necessary for model training is the `parameter table`. This table describes all free parameters $\boldsymbol{\theta}$ (`petab.PARAMETER_ID`), whether they have to be estimated (`petab.ESTIMATE`) as well as their scales (`petab.PARAMETER_SCALE`), nominal values (`petab.NOMINAL_VALUE`), boundary values $\mathcal{L}_l$ (`petab.LOWER_BOUND`) and $\mathcal{U}_l$ (`petab.UPPER_BOUND`) and information about priors.

Both parameter boundaries and priors can be used to restrict the search space to biologically plausible parameter values, or prevent numerical issues in regions of parameter space where numerical integration of model equations is challenging. Parameter boundaries set hard thresholds that cannot be exceeded, while parameter priors are soft constraints that also regularize the calibration. While priors are necessary for Bayesian analysis such as parameter sampling, they are not generally required for optimization based approaches.

For most models, free parameters $\boldsymbol{\theta}$ will primarily occur in the model specification as dynamic parameters $\boldsymbol{\gamma}$. But, all of previously described tables may introduce new free parameters: noise parameters $\boldsymbol{\psi}$ in noise formulas $\boldsymbol{\sigma}$ or respective condition specific mappings $\boldsymbol{\Psi}_{ijk}$, observable paramers $\boldsymbol{\xi}$ in observable formulas $\boldsymbol{g}$ or respective condition specific mappings $\boldsymbol{\Xi}_{ijk}$ as well as the condition specific dynamic parameter mappings $\boldsymbol{\Gamma}_k$. Moreover, the respective tables, i.e., the respective mappings $\boldsymbol{\Psi}_{ijk}$, $\boldsymbol{\Xi}_{ijk}$ and $\boldsymbol{\Gamma}_k$ may set numerical values for the noise, observable and dynamic parameters, rendering them non-free. As the parameter table must only describe free parameters $\boldsymbol{\theta}$, some processing is usually necessary to identify free and non-free parameters.

In the following, we will construct a list of `condition_pars`, which are either mapped to free,

condition-specific parameters, such as the generic RAFi energy parameters, or set to non-free parameters with fixed numeric values, such as the `RAFi_0` initial concentration. To construct the list of free parameters, we identify all dynamic parameters in the model that are not contained in these `condition_pars`, and add all newly introduced condition specific parameters in the `condition table` as well as the four offset and scaling parameters we introduced in the `observables table`.

[16]:
```
import numpy as np
condition_pars = [
    par.name
    for par in model.parameters
    if par.name in condition_table.columns
]

free_parameters = [
    par.name for par in model.parameters
    if par.name not in condition_pars
] + [
    name
    for par in condition_pars
    for name in np.unique(condition_table[par])
    if isinstance(name, str)
] + [
    'pMEK_offset','pMEK_scale', 'pERK_offset', 'pERK_scale'
]
```

All of the parameters in the model follow a consistent naming scheme, which makes it easier to programmatically specify parameter boundaries, scales and whether parameters are to be estimated. Multiple studies have demonstrated that parameter estimation is more efficient when estimating parameters on a logarithmic scale [23, 31]. Yet, the thermodynamic formulation we employed for model construction relies on an exponential dependency between energies and kinetetic rates, which complements these practical insights with a theoretical foundation why a logarithmic scale is more natural. However, this also suggests that energetic parameters, in contrast to kinetic rates, should be estimated on a linear scale.

The upper and lower boundaries should ideally be set to biologically plausible values. However, most models employ simplified descriptions of the underlying biochemical processes, which means that model parameters may no longer have a one-to-one correspondance to true biochemical constants and plausible parameter ranges may be difficult to derive. Moreover, plausible parameter ranges may not be known for all parameters. Therefore, in many applications parameter boundaries are initially based on educated guesses and then refined based on estimation results such that optima are contained in the specified ranges and integration failures are not too frequent. Therefore, the boundaries presented here are based on the values used in the original publication [27] and were then refined to work with the simplified model.

Similar to parameter boundaries, nominal values can be difficult to derive. However, the numerical values are only important for model calibration when respective parameters are not estimated. In this example, we do not estimate the initial concentrations and ϕ parameters and only set the nominal parameter values of those parameters to non-trivial values. The initial concentrations are set to approximate molecular counts as measured in the original publication and ϕ parameters are

all set to 1, meaning that free energy differences only affect the reverse rate.

[17]:
```
lbs = {
    'Ea': -8, 'dG': -8, 'ddG': -8, 'phi': 0, 'offset': 5e-2,
    'scale': 1e0, 'eq': 1e1, '0': 1e3, 'kcat': 1e-2, 'gexpslope': 1e0,
    'kdeg': 1e-3, 'kM': 1e-3, 'kcatr': 1e-5, 'koff': 1e-5,
}
ubs = {
    'Ea': 8, 'dG': 8, 'ddG': 8, 'phi': 1, 'offset': 2e-1,
    'scale': 1e6, 'eq': 1e5, '0': 1e5, 'kcat': 1e5, 'gexpslope': 1e6,
    'kdeg': 1e2, 'kM': 1e1, 'kcatr': 1e2, 'koff': 1e0,
}

initials = {
    'BRAF_0': 1e3, 'CRAF_0': 1e4, 'MEK_0': 1e5, 'ERK_0': 1e5, 'RAS_0': 5e4,
}


parameter_table = pd.DataFrame([
    {
        petab.PARAMETER_ID: par,
        petab.PARAMETER_SCALE: petab.LIN
        if par.endswith(('_Ea', '_dG', '_ddG', '_phi')) else petab.LOG10,
        petab.LOWER_BOUND: lbs[par.split('_')[-1]],
        petab.UPPER_BOUND: ubs[par.split('_')[-1]],
        petab.NOMINAL_VALUE: initials[par] if par.endswith('_0') else 1.0,
        petab.ESTIMATE: False if par.endswith(('_phi', '_0')) else True,

    } for par in free_parameters
]).set_index(petab.PARAMETER_ID);
```

4.4 SBML export

PEtab expects a model in SBML format (an extension allowing other modeling formats is being developed at the time of writing), but we constructed the model in the PySB/BNGL format. Therefore, we need to export the model in SBML format.

[18]:
```
import libsbml
from pysb.export import export
sbml_reader = libsbml.SBMLReader()
sbml_doc = sbml_reader.readSBMLFromString(export(model, 'sbml'))
sbml_model = sbml_doc.getModel()
```

In the SBML model, PySB `Observable`s are exported as SBML `Parameters` and `AssignmentRule`, with programmatically generated identifiers that do not match the PySB `Observable` names. As we referenced these observable names when specifying the PEtab observables, we have to change

the SBML identifiers to match the PySB names. These changes have to be applied to the SBML `Parameter` as well as the SBML `AssignmentRule` that sets their value.

[19]:
```
# rename pysb exported observables
sbml_model.getParameter('__obs1').setId('pMEK_obs')
sbml_model.getAssignmentRuleByVariable('__obs1').setVariable('pMEK_obs')
sbml_model.getParameter('__obs2').setId('pERK_obs')
sbml_model.getAssignmentRuleByVariable('__obs2').setVariable('pERK_obs');
```

As last step, we create a PEtab `Problem` using all of the previously constructed tables as well as the exported SBML model. As problem construction was a lengthy, error-prone process, we perform a static analysis of the petab problem using the petab library provided linter `petab.lint_problem`. Here we suppress any SBML warnings, as they are only due to the implementation of the SBML export in PySB. These warnings cause the linter to terminate with "Not OK", but as there are no further issues, we can proceed with calibration.

[20]:
```
import petab
import logging
import sys
from petab.models.sbml_model import SbmlModel

petab_problem = petab.Problem(
    model=SbmlModel(sbml_model=sbml_model, sbml_reader=sbml_reader,␣
 ↪sbml_document=sbml_doc),
    measurement_df=measurement_table, condition_df=condition_table,
    observable_df=observables_table, parameter_df=parameter_table
)
petab.sbml.logger.setLevel(logging.ERROR)
petab.lint.logger.setLevel(logging.INFO)
petab.lint.logger.addHandler(logging.StreamHandler(sys.stdout))
petab.lint_problem(petab_problem);
```

```
Checking model...
Checking measurement table...
Checking condition table...
Checking observable table...
Checking parameter table...
Not OK
```

5 Calibrating the model in pyPESTO

Model calibration based on PEtab [26] format is supported by a growing number of tools, including COPASI [32] and data2dynamics [33]. Here we used the python based calibration tool pyPESTO for calibration. pyPESTO uses AMICI [34] for model import and simulation, constructs functions to evaluate the objective function $J(\theta)$ and its gradient $\nabla_\theta J(\theta)$ and provides an interface to a plethora of different optimizers.

To facilitate efficient evaluation of the objective function and its derivatives, we will first compile the model in AMICI. Model compilation should take about a minute on modern desktop machines, but ensures that numerical solutions $\mathbf{x}$ and their sensitivies $\nabla_\gamma \mathbf{x}$, which are required for the evaluation of $\nabla_\theta J(\boldsymbol{\theta})$, are computed using the SUNDIALS solver suite [35]. AMICI compilation is triggered by calling the `compile_model` method of a pyPESTO `PetabImporter` instance. The compiled model will be automatically used by other methods of the same `PetabImporter` instance.

[21]:
```
import pypesto
import pypesto.petab
import amici

importer = pypesto.petab.PetabImporter(petab_problem,
                                       model_name=model.name,
                                       validate_petab=False)

importer.compile_model(verbose=logging.ERROR)
```

Simulations in AMICI are highly customizable and many steps, including everything that was specified in the the petab problem, are automated by pyPESTO. Yet, not all options are set automatically and some user input may be required. For example, the considered example includes preequilibrations (specified by `petab.PREEQUILIBRATION_CONDITION`), which are automatically handled by pyPESTO and AMICI. However, AMICI supports multiple different methods to compute steady states as well as respective sensitivities. By default, a combination of Newton's method and simulation is employed to approximate the steady state [36]. To find a steady state by Newton's method, the Newton-Raphson algorithm is applied to the rootfinding problem

$$\{\mathbf{x} : \mathbf{f}(t, \mathbf{x}, \gamma) = 0\}.$$

In contrast, the simulation based approach numerically integrates (1) until $\mathbf{f}$ is sufficiently small with respect to some norm. For both approaches, AMICI uses a convergence criteria based on a combination of absolute and relative tolerances, similar to error control during numerical integration:

$$\sum_i^{n_x} \frac{f_i(t, \mathbf{x}, \gamma)}{a + r x_i} < 1$$

where a and r are absolute and relative tolerances that can be set via the `Solver` methods `setAbsoluteToleranceSteadyState` and `setRelativeToleranceSteadyState` respectively. In both cases, sensitivites $\nabla_\eta \mathbf{x}_{ss}$ of the steady state $\mathbf{x}_{ss}$ can be computed using the Implicit Function Theorem (IFT) [37], which is the most efficient option when applicable:

$$\nabla_\eta \mathbf{x}_{ss} = \nabla_x \mathbf{f}(t, \mathbf{x}_{ss}, \boldsymbol{\eta})^{-1} \nabla_\eta \mathbf{f}(t, \mathbf{x}_{ss}, \boldsymbol{\eta}).$$

However, sensitivity computation via the IFT requires that the Jacobian $\nabla_x \mathbf{f}(t, \mathbf{x}_{ss}, \gamma)$ is not singular, which is, for example, not the case for models with conservation laws. More details about preequilibration in AMICI is available in the `online documentation`.

As the model we constructed does not account for protein synthesis and degradation, total protein abundances are conserved quantities in the model and the IFT is not applicable. Thus, we here employ a purely simulation based steady state search by setting the allowed newton steps to 0 and compute the steady state sensitivities using forward sensitivity analysis by setting the steady state sensitivity mode accordingly. To apply these options during optimization, we create AMICI `Solver` and `Model` instances from the `PetabImporter` and set respective options.

[22]:
```
import amici
solver = importer.create_solver()
model = importer.create_model()
solver.setNewtonMaxSteps(0)
model.setSteadyStateSensitivityMode(amici.SteadyStateSensitivityMode.
 ↪integrationOnly)
```

AMICI also supports different sensitivity analysis methods for dynamic simulations (specified by `petab.SIMULATION_CONDITION_ID`) [24]. For models with over 100 parameters, it usually makes sense to use adjoint sensitivity analysis [23]. However, for models that also require preequilibration, the computation time of the simulation and gradient evaluation may be dominated by preequilibration and the choice of sensitivity method for dynamic simulations may have negligible impact on computation time. Accordingly, other considerations, such as the availability of approximate second order information should be taken into account when selecting the sensitivity method. In contrast to forward sensitivity analysis, adjoint sensitivity analysis does not provide an approximation to the Hessian of the objective function [24], rendering some optimizers inapplicable and potentially reducing the convergence rate of those that remain applicable. Therefore, we here chose forward sensitivity analysis for dynamic simulations.

[23]:
```
solver.setSensitivityMethod(amici.SensitivityMethod.forward)
```

Another set of crucial parameters to tune are integration step limits and tolerances. Lower tolerances generally improve accuracy of the objective function and its derivatives, which influences local convergence rate and success [38]. However, lower tolerances also increases the computation time [39], so fewer optimization iterations could easily be offsetted by longer compute times per iteration. Moreover, both integration tolerances and step limits influence numerical integrability [39], which can lead to complications during optimization. Low step limits can promote numerical integration failure and, thereby, prevent optimization from exploring difficult to integrate parameter regions. However, if parameters from these regions yield good fits to the data, this can also deteriorate or bias calibration. Accordingly, setting adequate values for step limits and tolerances is integral for model calibration. Nevertheless, there are little to no theoretical or practical guidelines on how to choose adequate values. Ranges of 10^{-8} to 10^{-12} are usually adequate [39], but most of the time values are chosen and updated empirically. Here we chose tolerances of 10^{-12} and a step limit of 10^5 integration steps, which is rather conservative, but ensures adequate local convergence for the considered problem.

[24]:
```
solver.setAbsoluteTolerance(1e-12)
solver.setRelativeTolerance(1e-12)
solver.setMaxSteps(int(1e5))
```

pyPESTO also supports extrapolation methods that iteratively update initial guesses for steady

states by linearly extrapolating previously found steady states based on steady state sensitivities. As this may lead to issues when combined with forward, simulation based sensitivity analysis, we deactivate this option here. To set the option, we create a new `AmiciObjective` instance and pass the model solver and model instance we previously created to ensure that all options that we previously set are applied during evaluation of the objective.

[25]:
```
obj = importer.create_objective(model=model, solver=solver)
obj.guess_steadystate = False
```

Lastly, we instantiate a pyPESTO `Problem` and attach the `AmiciObjective` we created.

[26]:
```
pypesto_problem = importer.create_problem(objective=obj)
```

As gradient based methods are local methods, they have guaranteed convergence to local minima (at least theoretically) and multiple repeated optimization runs initialized at different points in Θ have to be started to explore the objective function landscape [40]. Selecting an appropriate number of optimization runs can be challenging, but should generally be adapted to the downstream analysis. For example, when model selection or uncertainty analysis using profile likelihoods will be performed, it is crucial to ensure that optimization reaches a local optimum and that the local optimum is, or at least yields a fit equivalent to, the global optimum. This can be achieved by running many optimization runs and allocating large computational budgets for each run. For large models, the total computational budget usually is insufficient to achieve these goals and the number of runs and the computational budget has to be limited, with an unknown tradeoff between the two. To deal with this tradeoff, we will here use a two-staged approach where we first run many starts with a low computational budget of 10 minutes. This will just be enough for a handful of iterations and we will then select the most promising runs and continue optimization with a larger computational budget. As optimizer, we employ fides, a high performance trust-region optimizer permitting the direct specification of the computational budgets, which can be interfaced in pyPESTO using the `FidesOptimizer` class.

[27]:
```
from pypesto.optimize import FidesOptimizer
import fides
import logging

optimizer = FidesOptimizer(options={fides.Options.MAXTIME: 600},
                           verbose=logging.ERROR)
```

Now we have everything at hand to run the first calibration stage. Here, we run 100 optimization runs with random starting points (uniformly sampled by default) using the pyPESTO `minimize` function. The calibration is parallelized using pyPESTOs `MultiThreadEngine` with 4 threads.

[28]:
```
from pypesto.optimize import minimize
result = minimize(
    pypesto_problem, optimizer, n_starts=100,
    engine=pypesto.engine.MultiThreadEngine(4),
    progress_bar=False,
)
```

To assess whether the quality of the calibration, we will now use the PEtab visualization feature.

The visualization instructions are specified in a `visualization table`, where, in the case of line plots, every row corresponds to a single pair of connected lines for data and simulation. Here we make use of the of the `petab.DATASET_ID` values that we specified in the `measurement table`. In the following we will generate seperate plots for each observable and each inhibitor, where every plot shows the dose response in the EGF stimulated and unstimulated condition. The visualization table is then attached to the previously created petab_problem.

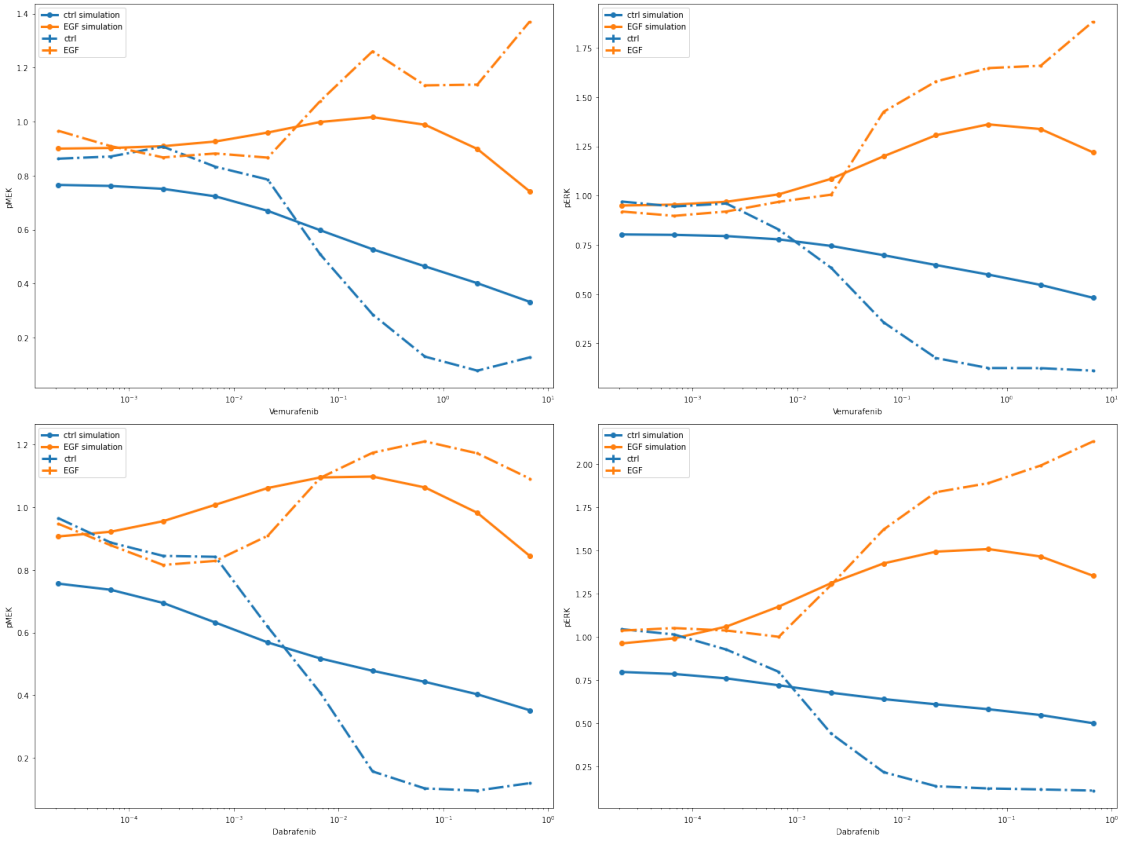
[29]:
```
visualization_table = pd.DataFrame([
    {
        petab.PLOT_ID: f'{rafi}_{obs}',
        petab.PLOT_TYPE_SIMULATION: petab.LINE_PLOT,
        petab.PLOT_TYPE_DATA: petab.MEAN_AND_SD,
        petab.DATASET_ID: condition,
        petab.X_VALUES: 'RAFi_0',
        petab.Y_VALUES: obs,
        petab.X_SCALE: petab.LOG10,
        petab.X_LABEL: rafi,
        petab.Y_LABEL: obs,
        petab.LEGEND_ENTRY: condition.split('__')[0]
    }
    for rafi in RAFis
    for obs in observables_table.index
    for condition in measurement_table[petab.DATASET_ID].unique()
    if condition.split('__')[1] == rafi
])
petab_problem.visualization_df = visualization_table
```

While experimental data for visualization is already available in the `measurement table`, simulations of the calibrated model still have to be computed. To generate model simulations, we extract the parameter values of the best pre-calibration. As optimization results are sorted in increasing order of objective function values, this can be achieved by taking the result at the first index. PEtab compatible simulation results can be extracted using the pyPESTO objective and the AMICI function `amici.petab_objective.rdatas_to_simulation_df`. As the results contains the full parameter vector, including parameter values that were not estimated, but the objective only expects estimated parameters, the parameters need to be subsetted according to the free indices.

[30]:
```
x = pypesto_problem.get_reduced_vector(result.optimize_result.list[0]['x'],
                                       pypesto_problem.x_free_indices)
simulation = pypesto_problem.objective(x, return_dict=True)
simulation_df = amici.petab_objective.rdatas_to_simulation_df(
    simulation['rdatas'],
    model=pypesto_problem.objective.amici_model,
    measurement_df=measurement_table,
)
```

We can now visualize both data and simulation using `petab.visualize.plot_problem`.

[31]:
```
import petab.visualize
import matplotlib.pyplot as plt

plt.rc('font', size=25)
plt.rc('lines', linewidth=3)

petab.visualize.plot_problem(
    petab_problem=petab_problem,
    simulations_df=simulation_df,
);
```



The generated figure suggests that model simulations are qualitatively similar to training data, but there are still large quantitative differences. This suggests that the pre-calibrated parameter values are an adequate first guess, but require further refinement.

Before we proceed with the second calibration round, we can check the accuracy of the objective function gradient at the pre-calibrated parameter values. For this purpose, we use the `check_grad_multi_eps` method, which approximates the objective function gradient using finite differences with different step sizes.

[32]:
```
pd.set_option('display.max_rows', 5)
pd.set_option('display.max_columns', 8)
pypesto_problem.objective.check_grad_multi_eps(x, multi_eps=[1e-1, 1e-3, 1e-5],␣
↪verbosity=0)
```

[33]:

	grad	fd_f	fd_b	fd_c	\
bind_RAF_RAF_Ea	-0.000676	0.066034	-0.082687	-0.008326	
bind_RAF_RAF_dG	151.299971	161.357110	141.392663	151.374887	
...	...	...	...	...	
pERK_offset	-14.721473	-7.347211	-22.133442	-14.740327	
pERK_scale	-1593.045905	-1569.867010	-1616.507993	-1593.187501	

	fd_err	abs_err	rel_err	eps
bind_RAF_RAF_Ea	0.148720	0.007650	0.083453	0.100
bind_RAF_RAF_dG	19.964446	0.074916	0.000495	0.100
...	...	...	...	...
pERK_offset	14.786231	0.018853	0.001279	0.001
pERK_scale	46.640983	0.141596	0.000089	0.001

[28 rows x 8 columns]

The method computes forward (`fd_c`), backward (`fd_b`) and central (`fd_c`) finite differences and compares them to the sensitivity based gradient (`grad`), using the step size `eps` that yields the lowest finite difference approximation error among the provided step sizes. The results show a low error in the finite difference approximation (`fd_err`), indicating accurate evaluation of the objective function. Moreover, the small absolute (`abs_err`) and relative difference (`rel_err`) between sensitivity based gradient and finite difference approximation indicates accurate evaluation of the gradient. The large magnitude of the gradient suggest that we did not yet reach a local optimum and that further optimization will likely improve agreement between data and simulation. In cases where either `fd_err`, `abs_err` or `rel_err` take large values, it is recommended to decrease integration tolerances until adequate numbers are achieved, where entries in `grad` can serve as reference.

We will now perfom this refinement by instantiating a new optimizer with a larger computational budget of two hours.

[33]:
```
optimizer_fine = FidesOptimizer(options={fides.Options.MAXTIME: 7200},
                                verbose=logging.ERROR)
```

To initialize the second calibration round at the pre-calibrated parameter values, we extract the respective values from the results object and set them as guesses in the pyPESTO problem.

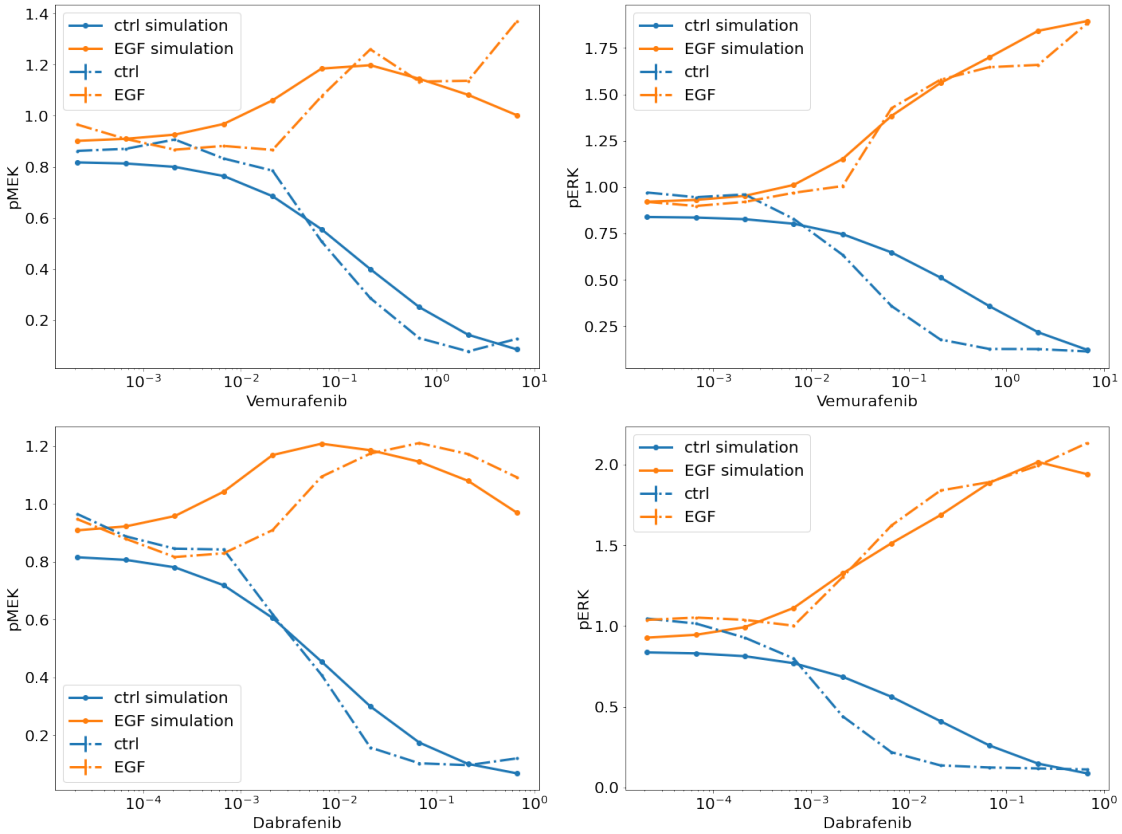
[34]:
```
n_refine = 10
pypesto_problem.x_guesses_full = np.vstack(result.optimize_result.x[:n_refine])
```

For startpoint generation, pyPESTO will first start optimization runs at the provided parameter guesses and only sample new startpoint if more optimization runs than guesses are requested. As for the first calibration run, we run the calibration, using the pyPESTO `minimize` function.

[35]:
```
result_refined = minimize(
    pypesto_problem, optimizer_fine, n_starts=n_refine,
    engine=pypesto.engine.MultiThreadEngine(4),
    progress_bar=False,
)
```

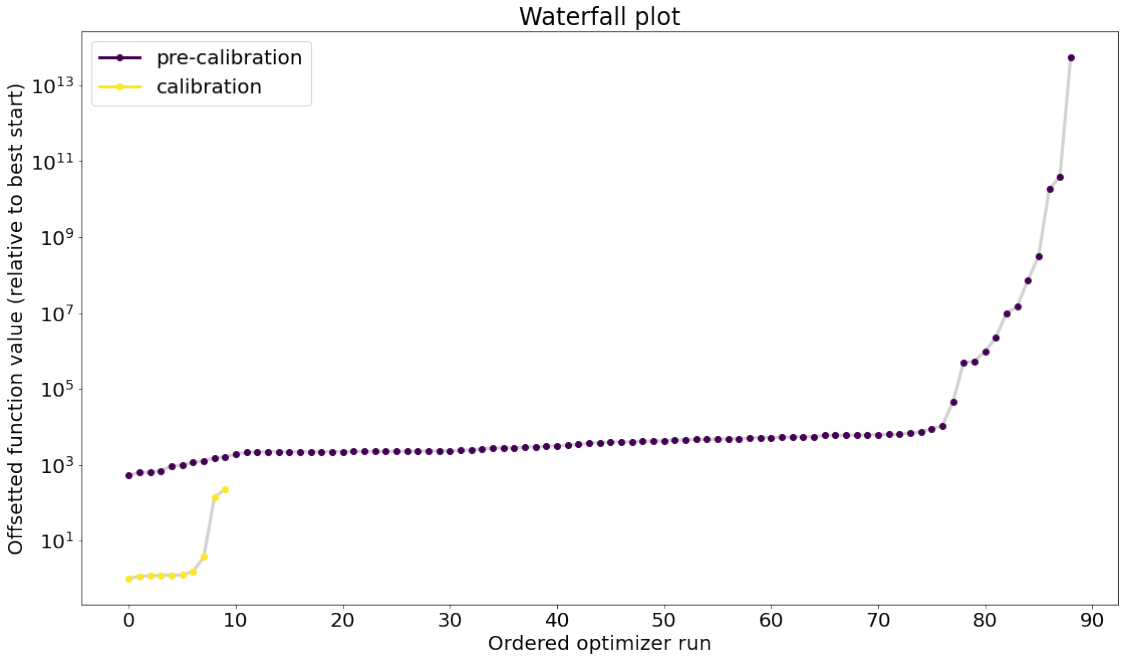
As for the pre-calibration, we now visualize the results using PEtab and the previously constructed visualization table.

[36]:
```
x = pypesto_problem.get_reduced_vector(
    result_refined.optimize_result.list[0]['x'],
    pypesto_problem.x_free_indices
)
simulation = pypesto_problem.objective(x, return_dict=True)
simulation_df = amici.petab_objective.rdatas_to_simulation_df(
    simulation['rdatas'],
    model=importer.create_model(),
    measurement_df=measurement_table,
)

plt.rc('font', size=20)
plt.rc('lines', linewidth=3)

petab.visualize.plot_problem(
    petab_problem=petab_problem,
    simulations_df=simulation_df,
);
```



These generated figure now indicate adequate agreement between experimental data and model simulations. As a last step of the analysis, we now compare the objective function values across both calibration rounds using a waterfall plot.

[37]:
```
from pypesto.visualize import waterfall
waterfall(results=[result,result_refined],
          legends=['pre-calibration', 'calibration']);
```



The generaterd waterfall plot shows sorted objective function values of different optimizer runs for pre-calibration and calibration. The objective function values are normalized to the best objective function value achieved across both runs. We observe that for the majority of fully calibrated runs, we achieve much lower objective function values compared to the pre-calibrated runs, indicating that the higher computational budget facilitated better fits. For a small subset of starts, we observe repeated convergence to similar objective function values that form a plateau in the waterfall plot, suggesting that the respective runs may have converged to the same local minimum. Ideally, the majority of objective function values would lie in a plateau, but this would likely require many more optimizer runs with larger computational budgets and would thus have to be performed on a computer cluster.

6 Discussion

Even with state-of-the-art methods the process of model formulation, problem specification and calibration we described in this protocol remains a labour intensive process, where good understanding of the underlying mathemical concepts is necessary to achieve optimal, or even adequate performance for large kinetic models. We anticipate that the extensive description and code we provide in this protocol will serve as an introduction to model calibration and may be used as template for future research projects.

In terms of future developments, more automation in terms of tolerance selection, sensitivity methods and computational budget allocation will be necessary to improve accessibility of methods and reduce the amount of required user interaction. Some frameworks such as data2dynamics [33] already provide adaptive tolerance updating schemes in cases of numerical integration failure. This could still be complemented by adaptive updating of tolerances based on objective func-

tion evaluation accuracy, but underlying mathematical theory and good benchmark problems are missing. We are convinced that the introduction of large benchmarks in [31, 23], guidelines for their evaluation [41] as well as the standardization of their formulation [26] will prove to be crucial to improve automation of model calibration in the future.

6.1 References

References

[1] Hugo Lavoie, Jessica Gagnon, and Marc Therrien. Erk signalling: a master regulator of cell behaviour, life and fate. *Nature Reviews Molecular Cell Biology*, 21:607–632, 2020.

[2] Axel Ullrich and Joseph Schlessinger. Signal transduction by receptors with tyrosine kinase activity. *Cell*, 61:203–212, 1990.

[3] M M McKay and D K Morrison. Integrating signals from rtks to erk/mapk. *Oncogene*, 26:3113, 2007.

[4] T. Hunter. Signaling–2000 and beyond. *Cell*, 100:113–127, 2000.

[5] Francisco Sanchez-Vega, Marco Mina, Joshua Armenia, Walid K. Chatila, Augustin Luna, Konnor C. La, Sofia Dimitriadoy, David L. Liu, Havish S. Kantheti, Sadegh Saghafinia, Debyani Chakravarty, Foysal Daian, Qingsong Gao, Matthew H. Bailey, Wen-Wei Liang, Steven M. Foltz, Ilya Shmulevich, Li Ding, Zachary Heins, Angelica Ochoa, Benjamin Gross, Jianjiong Gao, Hongxin Zhang, Ritika Kundra, Cyriac Kandoth, Istemi Bahceci, Leonard Dervishi, Ugur Dogrusoz, Wanding Zhou, Hui Shen, Peter W. Laird, Gregory P. Way, Casey S. Greene, Han Liang, Yonghong Xiao, Chen Wang, Antonio Iavarone, Alice H. Berger, Trever G. Bivona, Alexander J. Lazar, Gary D. Hammer, Thomas Giordano, Lawrence N. Kwong, Grant McArthur, Chenfei Huang, Aaron D. Tward, Mitchell J. Frederick, Frank McCormick, Matthew Meyerson, Samantha J. Caesar-Johnson, John A. Demchok, Ina Felau, Melpomeni Kasapi, Martin L. Ferguson, Carolyn M. Hutter, Heidi J. Sofia, Roy Tarnuzzer, Zhining Wang, Liming Yang, Jean C. Zenklusen, Jiashan (Julia) Zhang, Sudha Chudamani, Jia Liu, Laxmi Lolla, Rashi Naresh, Todd Pihl, Qiang Sun, Yunhu Wan, Ye Wu, Juok Cho, Timothy DeFreitas, Scott Frazer, Nils Gehlenborg, Gad Getz, David I. Heiman, Jaegil Kim, Michael S. Lawrence, Pei Lin, Sam Meier, Michael S. Noble, Gordon Saksena, Doug Voet, Hailei Zhang, Brady Bernard, Nyasha Chambwe, Varsha Dhankani, Theo Knijnenburg, Roger Kramer, Kalle Leinonen, Yuexin Liu, Michael Miller, Sheila Reynolds, Ilya Shmulevich, Vesteinn Thorsson, Wei Zhang, Rehan Akbani, Bradley M. Broom, Apurva M. Hegde, Zhenlin Ju, Rupa S. Kanchi, Anil Korkut, Jun Li, Han Liang, Shiyun Ling, Wenbin Liu, Yiling Lu, Gordon B. Mills, Kwok-Shing Ng, Arvind Rao, Michael Ryan, Jing Wang, John N. Weinstein, Jiexin Zhang, Adam Abeshouse, Joshua Armenia, Debyani Chakravarty, Walid K. Chatila, Ino de Bruijn, Jianjiong Gao, Benjamin E. Gross, Zachary J. Heins, Ritika Kundra, Konnor La, Marc Ladanyi, Augustin Luna, Moriah G. Nissan, Angelica Ochoa, Sarah M. Phillips, Ed Reznik, Francisco Sanchez-Vega, Chris Sander, Nikolaus Schultz, Robert Sheridan, S. Onur Sumer, Yichao Sun, Barry S. Taylor, Jioajiao Wang, Hongxin Zhang, Pavana Anur, Myron Peto, Paul Spellman, Christopher Benz, Joshua M. Stuart, Christopher K. Wong, Christina Yau, D. Neil Hayes, Joel S. Parker, Matthew D. Wilkerson, Adrian Ally, Miruna Balasundaram, Reanne Bowlby, Denise Brooks, Rebecca Carlsen, Eric Chuah, Noreen Dhalla, Robert Holt, Steven J.M. Jones,

Katayoon Kasaian, Darlene Lee, Yussanne Ma, Marco A. Marra, Michael Mayo, Richard A. Moore, Andrew J. Mungall, Karen Mungall, A. Gordon Robertson, Sara Sadeghi, Jacqueline E. Schein, Payal Sipahimalani, Angela Tam, Nina Thiessen, Kane Tse, Tina Wong, Ashton C. Berger, Rameen Beroukhim, Andrew D. Cherniack, Carrie Cibulskis, Stacey B. Gabriel, Galen F. Gao, Gavin Ha, Matthew Meyerson, Steven E. Schumacher, Juliann Shih, Melanie H. Kucherlapati, Raju S. Kucherlapati, Stephen Baylin, Leslie Cope, Ludmila Danilova, Moiz S. Bootwalla, Phillip H. Lai, Dennis T. Maglinte, David J. Van Den Berg, Daniel J. Weisenberger, J. Todd Auman, Saianand Balu, Tom Bodenheimer, Cheng Fan, Katherine A. Hoadley, Alan P. Hoyle, Stuart R. Jefferys, Corbin D. Jones, Shaowu Meng, Piotr A. Mieczkowski, Lisle E. Mose, Amy H. Perou, Charles M. Perou, Jeffrey Roach, Yan Shi, Janae V. Simons, Tara Skelly, Matthew G. Soloway, Donghui Tan, Umadevi Veluvolu, Huihui Fan, Toshinori Hinoue, Peter W. Laird, Hui Shen, Wanding Zhou, Michelle Bellair, Kyle Chang, Kyle Covington, Chad J. Creighton, Huyen Dinh, HarshaVardhan Doddapaneni, Lawrence A. Donehower, Jennifer Drummond, Richard A. Gibbs, Robert Glenn, Walker Hale, Yi Han, Jianhong Hu, Viktoriya Korchina, Sandra Lee, Lora Lewis, Wei Li, Xiuping Liu, Margaret Morgan, Donna Morton, Donna Muzny, Jireh Santibanez, Margi Sheth, Eve Shinbrot, Linghua Wang, Min Wang, David A. Wheeler, Liu Xi, Fengmei Zhao, Julian Hess, Elizabeth L. Appelbaum, Matthew Bailey, Matthew G. Cordes, Li Ding, Catrina C. Fronick, Lucinda A. Fulton, Robert S. Fulton, Cyriac Kandoth, Elaine R. Mardis, Michael D. McLellan, Christopher A. Miller, Heather K. Schmidt, Richard K. Wilson, Daniel Crain, Erin Curley, Johanna Gardner, Kevin Lau, David Mallery, Scott Morris, Joseph Paulauskis, Robert Penny, Candace Shelton, Troy Shelton, Mark Sherman, Eric Thompson, Peggy Yena, Jay Bowen, Julie M. Gastier-Foster, Mark Gerken, Kristen M. Leraas, Tara M. Lichtenberg, Nilsa C. Ramirez, Lisa Wise, Erik Zmuda, Niall Corcoran, Tony Costello, Christopher Hovens, Andre L. Carvalho, Ana C. de Carvalho, JosÃľ H. Fregnani, Adhemar Longatto-Filho, Rui M. Reis, Cristovam Scapulatempo-Neto, Henrique C.S. Silveira, Daniel O. Vidal, Andrew Burnette, Jennifer Eschbacher, Beth Hermes, Ardene Noss, Rosy Singh, Matthew L. Anderson, Patricia D. Castro, Michael Ittmann, David Huntsman, Bernard Kohl, Xuan Le, Richard Thorp, Chris Andry, Elizabeth R. Duffy, Vladimir Lyadov, Oxana Paklina, Galiya Setdikova, Alexey Shabunin, Mikhail Tavobilov, Christopher McPherson, Ronald Warnick, Ross Berkowitz, Daniel Cramer, Colleen Feltmate, Neil Horowitz, Adam Kibel, Michael Muto, Chandrajit P. Raut, Andrei Malykh, Jill S. Barnholtz-Sloan, Wendi Barrett, Karen Devine, Jordonna Fulop, Quinn T. Ostrom, Kristen Shimmel, Yingli Wolinsky, Andrew E. Sloan, Agostino De Rose, Felice Giuliante, Marc Goodman, Beth Y. Karlan, Curt H. Hagedorn, John Eckman, Jodi Harr, Jerome Myers, Kelinda Tucker, Leigh Anne Zach, Brenda Deyarmin, Hai Hu, Leonid Kvecher, Caroline Larson, Richard J. Mural, Stella Somiari, Ales Vicha, Tomas Zelinka, Joseph Bennett, Mary Iacocca, Brenda Rabeno, Patricia Swanson, Mathieu Latour, Louis Lacombe, Bernard TÃłtu, Alain Bergeron, Mary McGraw, Susan M. Staugaitis, John Chabot, Hanina Hibshoosh, Antonia Sepulveda, Tao Su, Timothy Wang, Olga Potapova, Olga Voronina, Laurence Desjardins, Odette Mariani, Sergio Roman-Roman, Xavier Sastre, Marc-Henri Stern, Feixiong Cheng, Sabina Signoretti, Andrew Berchuck, Darell Bigner, Eric Lipp, Jeffrey Marks, Shannon McCall, Roger McLendon, Angeles Secord, Alexis Sharp, Madhusmita Behera, Daniel J. Brat, Amy Chen, Keith Delman, Seth Force, Fadlo Khuri, Kelly Magliocca, Shishir Maithel, Jeffrey J. Olson, Taofeek Owonikoko, Alan Pickens, Suresh Ramalingam, Dong M. Shin, Gabriel Sica, Erwin G. Van Meir, Hongzheng Zhang, Wil Eijckenboom, Ad Gillis, Esther Korpershoek, Leendert Looijenga, Wolter Oosterhuis, Hans Stoop, Kim E. van Kessel, Ellen C. Zwarthoff, Chiara Calatozzolo, Lucia Cuppini, Stefania Cuzzubbo, Francesco DiMeco, Gaetano Finocchiaro, Luca Mattei, Alessandro Perin, Bianca Pollo, Chu Chen, John Houck, Pawadee Lo-

havanichbutr, Arndt Hartmann, Christine Stoehr, Robert Stoehr, Helge Taubert, Sven Wach, Bernd Wullich, Witold Kycler, Dawid Murawa, Maciej Wiznerowicz, Ki Chung, W. Jeffrey Edenfield, Julie Martin, Eric Baudin, Glenn Bubley, Raphael Bueno, Assunta De Rienzo, William G. Richards, Steven Kalkanis, Tom Mikkelsen, Houtan Noushmehr, Lisa Scarpace, Nicolas Girard, Marta Aymerich, Elias Campo, Eva GinÃl', Armando LÃşpez Guillermo, Nguyen Van Bang, Phan Thi Hanh, Bui Duc Phu, Yufang Tang, Howard Colman, Kimberley Evason, Peter R. Dottino, John A. Martignetti, Hani Gabra, Hartmut Juhl, Teniola Akeredolu, Serghei Stepa, Dave Hoon, Keunsoo Ahn, Koo Jeong Kang, Felix Beuschlein, Anne Breggia, Michael Birrer, Debra Bell, Mitesh Borad, Alan H. Bryce, Erik Castle, Vishal Chandan, John Cheville, John A. Copland, Michael Farnell, Thomas Flotte, Nasra Giama, Thai Ho, Michael Kendrick, Jean-Pierre Kocher, Karla Kopp, Catherine Moser, David Nagorney, Daniel OâĂŹBrien, Brian Patrick OâĂŹNeill, Tushar Patel, Gloria Petersen, Florencia Que, Michael Rivera, Lewis Roberts, Robert Smallridge, Thomas Smyrk, Melissa Stanton, R. Houston Thompson, Michael Torbenson, Ju Dong Yang, Lizhi Zhang, Fadi Brimo, Jaffer A. Ajani, Ana Maria Angulo Gonzalez, Carmen Behrens, Jolanta Bondaruk, Russell Broaddus, Bogdan Czerniak, Bita Esmaeli, Junya Fujimoto, Jeffrey Gershenwald, Charles Guo, Alexander J. Lazar, Christopher Logothetis, Funda Meric-Bernstam, Cesar Moran, Lois Ramondetta, David Rice, Anil Sood, Pheroze Tamboli, Timothy Thompson, Patricia Troncoso, Anne Tsao, Ignacio Wistuba, Candace Carter, Lauren Haydu, Peter Hersey, Valerie Jakrot, Hojabr Kakavand, Richard Kefford, Kenneth Lee, Georgina Long, Graham Mann, Michael Quinn, Robyn Saw, Richard Scolyer, Kerwin Shannon, Andrew Spillane, Jonathan Stretch, Maria Synott, John Thompson, James Wilmott, Hikmat Al-Ahmadie, Timothy A. Chan, Ronald Ghossein, Anuradha Gopalan, Douglas A. Levine, Victor Reuter, Samuel Singer, Bhuvanesh Singh, Nguyen Viet Tien, Thomas Broudy, Cyrus Mirsaidi, Praveen Nair, Paul Drwiega, Judy Miller, Jennifer Smith, Howard Zaren, Joong-Won Park, Nguyen Phi Hung, Electron Kebebew, W. Marston Linehan, Adam R. Metwalli, Karel Pacak, Peter A. Pinto, Mark Schiffman, Laura S. Schmidt, Cathy D. Vocke, Nicolas Wentzensen, Robert Worrell, Hannah Yang, Marc Moncrieff, Chandra Goparaju, Jonathan Melamed, Harvey Pass, Natalia Botnariuc, Irina Caraman, Mircea Cernat, Inga Chemencedji, Adrian Clipca, Serghei Doruc, Ghenadie Gorincioi, Sergiu Mura, Maria Pirtac, Irina Stancul, Diana Tcaciuc, Monique Albert, Iakovina Alexopoulou, Angel Arnaout, John Bartlett, Jay Engel, Sebastien Gilbert, Jeremy Parfitt, Harman Sekhon, George Thomas, Doris M. Rassl, Robert C. Rintoul, Carlo Bifulco, Raina Tamakawa, Walter Urba, Nicholas Hayward, Henri Timmers, Anna Antenucci, Francesco Facciolo, Gianluca Grazi, Mirella Marino, Roberta Merola, Ronald de Krijger, Anne-Paule Gimenez-Roqueplo, Alain PichÃl', Simone Chevalier, Ginette McKercher, Kivanc Birsoy, Gene Barnett, Cathy Brewer, Carol Farver, Theresa Naska, Nathan A. Pennell, Daniel Raymond, Cathy Schilero, Kathy Smolenski, Felicia Williams, Carl Morrison, Jeffrey A. Borgia, Michael J. Liptay, Mark Pool, Christopher W. Seder, Kerstin Junker, Larsson Omberg, Mikhail Dinkin, George Manikhas, Domenico Alvaro, Maria Consiglia Bragazzi, Vincenzo Cardinale, Guido Carpino, Eugenio Gaudio, David Chesla, Sandra Cottingham, Michael Dubina, Fedor Moiseenko, Renumathy Dhanasekaran, Karl-Friedrich Becker, Klaus-Peter Janssen, Julia Slotta-Huspenina, Mohamed H. Abdel-Rahman, Dina Aziz, Sue Bell, Colleen M. Cebulla, Amy Davis, Rebecca Duell, J. Bradley Elder, Joe Hilty, Bahavna Kumar, James Lang, Norman L. Lehman, Randy Mandt, Phuong Nguyen, Robert Pilarski, Karan Rai, Lynn Schoenfield, Kelly Senecal, Paul Wakely, Paul Hansen, Ronald Lechan, James Powers, Arthur Tischler, William E. Grizzle, Katherine C. Sexton, Alison Kastl, Joel Henderson, Sima Porten, Jens Waldmann, Martin Fassnacht, Sylvia L. Asa, Dirk Schadendorf, Marta Couce, Markus Graefen, Hartwig Huland, Guido Sauter, Thorsten Schlomm, Ronald Simon, Pierre Tennstedt, Oluwole Olabode, Mark

Nelson, Oliver Bathe, Peter R. Carroll, June M. Chan, Philip Disaia, Pat Glenn, Robin K. Kelley, Charles N. Landen, Joanna Phillips, Michael Prados, Jeffry Simko, Karen Smith-McCune, Scott VandenBerg, Kevin Roggin, Ashley Fehrenbach, Ady Kendler, Suzanne Sifri, Ruth Steele, Antonio Jimeno, Francis Carey, Ian Forgie, Massimo Mannelli, Michael Carney, Brenda Hernandez, Benito Campos, Christel Herold-Mende, Christin Jungk, Andreas Unterberg, Andreas von Deimling, Aaron Bossler, Joseph Galbraith, Laura Jacobus, Michael Knudson, Tina Knutson, Deqin Ma, Mohammed Milhem, Rita Sigmund, Andrew K. Godwin, Rashna Madan, Howard G. Rosenthal, Clement Adebamowo, Sally N. Adebamowo, Alex Boussioutas, David Beer, Thomas Giordano, Anne-Marie Mes-Masson, Fred Saad, Therese Bocklage, Lisa Landrum, Robert Mannel, Kathleen Moore, Katherine Moxley, Russel Postier, Joan Walker, Rosemary Zuna, Michael Feldman, Federico Valdivieso, Rajiv Dhir, James Luketich, Edna M. Mora Pinero, Mario Quintero-Aguilo, Jr. Carlotti, Carlos Gilberto, Jose SebastiÃčo Dos Santos, Rafael Kemp, Ajith Sankarankuty, Daniela Tirapelli, James Catto, Kathy Agnew, Elizabeth Swisher, Jenette Creaney, Bruce Robinson, Carl Simon Shelley, Eryn M. Godwin, Sara Kendall, Cassaundra Shipman, Carol Bradford, Thomas Carey, Andrea Haddad, Jeffey Moyer, Lisa Peterson, Mark Prince, Laura Rozek, Gregory Wolf, Rayleen Bowman, Kwun M. Fong, Ian Yang, Robert Korst, W. Kimryn Rathmell, J. Leigh Fantacone-Campbell, Jeffrey A. Hooke, Albert J. Kovatich, Craig D. Shriver, John DiPersio, Bettina Drake, Ramaswamy Govindan, Sharon Heath, Timothy Ley, Brian Van Tine, Peter Westervelt, Mark A. Rubin, Jung Il Lee, NatÃąlia D. Aredes, Armaz Mariamidze, Eliezer M. Van Allen, Andrew D. Cherniack, Giovanni Ciriello, Chris Sander, and Nikolaus Schultz. Oncogenic signaling pathways in the cancer genome atlas. *Cell*, 173:321–337.e10, 2018.

[6] Charles Sawyers. Targeted cancer therapy. *Nature*, 432:294–297, 2004.

[7] Piro Lito, Christine A. Pratilas, Eric W. Joseph, Madhavi Tadi, Ensar Halilovic, Matthew Zubrowski, Alan Huang, Wai Lin Wong, Margaret K. Callahan, Taha Merghoub, Jedd D. Wolchok, Elisa de Stanchina, Sarat Chandarlapaty, Poulikos I. Poulikakos, James A. Fagin, and Neal Rosen. Relief of profound feedback inhibition of mitogenic signaling by raf inhibitors attenuates their activity in brafv600e melanomas. *Cancer cell*, 22:668–682, 2012.

[8] Jonathan B. Fitzgerald, Birgit Schoeberl, Ulrik B. Nielsen, and Peter K. Sorger. Systems biology and combination therapy in the quest for clinical efficacy. *Nature Chemical Biology*, 2:458–466, 2006.

[9] Benjamin M Gyori, John A Bachman, Kartik Subramanian, Jeremy L Muhlich, Lucian Galescu, and Peter K Sorger. From word models to executable models of signaling networks using automated assembly. *Molecular Systems Biology*, 13, 2017.

[10] B. B. Aldridge, J. M. Burke, D. A. Lauffenburger, and P. K. Sorger. Physicochemical modelling of cell signalling pathways. *Nature Cell Biology*, 8:1195–1203, 2006.

[11] M. Hucka, A. Finney, H. M. Sauro, H. Bolouri, J. C. Doyle, H. Kitano, A. P. Arkin, B. J. Bornstein, D. Bray, A. Cornish-Bowden, A. A. Cuellar, S. Dronov, E. D. Gilles, M. Ginkel, V. Gor, I. I. Goryanin, W. J. Hedley, T. C. Hodgman, J.-H. Hofmeyr, P. J. Hunter, N. S. Juty, J. L. Kasberger, A. Kremling, U. Kummer, N. Le NovÃĺre, L. M. Loew, D. Lucio, P. Mendes, E. Minch, E. D. Mjolsness, Y. Nakayama, M. R. Nelson, P. F. Nielsen, T. Sakurada, J. C. Schaff, B. E. Shapiro, T. S. Shimizu, H. D. Spence, J. Stelling, K. Takahashi, M. Tomita, J. Wagner, and J. Wang. The systems biology markup language (sbml): a medium for representation and exchange of biochemical network models. *Bioinf.*, 19:524–531, 2003.

[12] Pierre Boutillier, Mutaamba Maasha, Xing Li, HÃľctor F. Medina-Abarca, Jean Krivine, JÃľrÃťme Feret, Ioana Cristescu, Angus G. Forbes, and Walter Fontana. The kappa platform for rule-based modeling. *Bioinformatics*, 34:i583–i592, 2018.

[13] Michael L. Blinov, James R. Faeder, Byron Goldstein, and William S. Hlavacek. Bionetgen: software for rule-based modeling of signal transduction based on the interactions of molecular domains. *Bioinformatics*, 20:3289–3291, 2004.

[14] William S. Hlavacek, James R. Faeder, Michael L. Blinov, Richard G. Posner, Michael Hucka, and Walter Fontana. Rules for modeling signal-transduction systems. *Science's STKE*, 2006:re6–re6, 2006.

[15] Julien F. Ollivier, Vahid Shahrezaei, and Peter S. Swain. Scalable rule-based modelling of allosteric proteins and biochemical networks. *PLOS Computational Biology*, 6:e1000975, 2010.

[16] J. A. P. Sekar, J. S. Hogg, and J. R. Faeder. Energy-based modeling in bionetgen. *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 1460–1467, 2016.

[17] BorisÂăN. Kholodenko. Drug resistance resulting from kinase dimerization is rationalized by thermodynamic factors describing allosteric inhibitor effects. *Cell Reports*, 12:1939–1949, 2015.

[18] Oleksii S. Rukhlenko, Fahimeh Khorsand, Aleksandar Krstic, Jan Rozanc, Leonidas G. Alexopoulos, Nora Rauch, Keesha E. Erickson, William S. Hlavacek, Richard G. Posner, Silvia GÃşmez-Coca, Edina Rosta, Cheree Fitzgibbon, David Matallanas, Jens Rauch, Walter Kolch, and Boris N. Kholodenko. Dissecting raf inhibitor resistance by structure-based modeling reveals ways to overcome oncogenic ras signaling. *Cell Systems*, 7:161–179.e14, 2018.

[19] Rud Wegscheider. ÃIJber simultane gleichgewichte und die beziehungen zwischen thermodynamik und reactionskinetik homogener systeme. *Monatshefte fÃijr Chemie und verwandte Teile anderer Wissenschaften*, 32:849–906, 1911.

[20] Poulikos I. Poulikakos, Chao Zhang, Gideon Bollag, Kevan M. Shokat, and Neal Rosen. Raf inhibitors transactivate raf dimers and erk signalling in cells with wild-type braf. *Nature*, 464:427–430, 2010.

[21] Georgia Hatzivassiliou, Kyung Song, Ivana Yen, Barbara J. Brandhuber, Daniel J. Anderson, Ryan Alvarado, Mary J. C. Ludlam, David Stokoe, Susan L. Gloor, Guy Vigers, Tony Morales, Ignacio Aliagas, Bonnie Liu, Steve Sideris, Klaus P. Hoeflich, Bijay S. Jaiswal, Somasekar Seshagiri, Hartmut Koeppen, Marcia Belvin, Lori S. Friedman, and Shiva Malek. Raf inhibitors prime wild-type raf to activate the mapk pathway and enhance growth. *Nature*, 464:431–435, 2010.

[22] Sonja J. Heidorn, Carla Milagre, Steven Whittaker, Arnaud Nourry, Ion Niculescu-Duvas, Nathalie Dhomen, Jahan Hussain, Jorge S. Reis-Filho, Caroline J. Springer, Catrin Pritchard, and Richard Marais. Kinase-dead braf and oncogenic ras cooperate to drive tumor progression through craf. *Cell*, 140:209–221, 2010.

[23] Alejandro F Villaverde, Fabian FrÃűhlich, Daniel Weindl, Jan Hasenauer, and Julio R Banga. Benchmarking optimization methods for parameter estimation in large kinetic models. *Bioinformatics*, 35:830–838, 2019.

[24] F. FrÃűhlich, B. Kaltenbacher, F. J. Theis, and J. Hasenauer. Scalable parameter estimation for genome-scale biochemical reaction networks. *PLoS Computational Biology*, 13:1–18, 2017.

[25] Ann C. Babtie and Michael P. H. Stumpf. How to deal with parameters for whole-cell modelling. *Journal of the Royal Society Interface*, 14, 2017.

[26] Leonard Schmiester, Yannik SchÃd'lte, Frank T. Bergmann, Tacio Camba, Erika Dudkin, Janine Egert, Fabian FrÃűhlich, Lara Fuhrmann, Adrian L. Hauber, Svenja Kemmer, Polina Lakrisenko, Carolin Loos, Simon Merkt, Wolfgang MÃijller, Dilan Pathirana, Elba RaimÃžndez, Lukas Refisch, Marcus Rosenblatt, Paul L. Stapor, Philipp StÃd'dter, Dantong Wang, Franz-Georg Wieland, Julio R. Banga, Jens Timmer, Alejandro F. Villaverde, Sven Sahle, Clemens Kreutz, Jan Hasenauer, and Daniel Weindl. PetabâĂŤinteroperable specification of parameter estimation problems in systems biology. *PLOS Computational Biology*, 17:e1008646, 2021.

[27] Luca Gerosa, Christopher Chidley, Fabian FrÃűhlich, Gabriela Sanchez, Sang Kyun Lim, Jeremy Muhlich, Jia-Yun Chen, Sreeram Vallabhaneni, Gregory J. Baker, Denis Schapiro, Mariya I. Atanasova, Lily A. Chylek, Tujin Shi, Lian Yi, Carrie D. Nicora, Allison Claas, Thomas S. C. Ng, Rainer H. Kohler, Douglas A. Lauffenburger, Ralph Weissleder, Miles A. Miller, Wei-Jun Qian, H. Steven Wiley, and Peter K. Sorger. Receptor-driven erk pulses reconfigure mapk signaling and enable persistence of drug-adapted braf-mutant melanoma cells. *Cell Systems*, 11:478–494.e9, 2020.

[28] Fabian FrÃűhlich, Luca Gerosa, Jeremy Muhlich, and Peter K. Sorger. Mechanistic model of mapk signaling reveals how allostery and rewiring contribute to drug resistance. *bioRxiv*, page 2022.02.17.480899, 2022.

[29] Alejandro F. Villaverde, Dilan Pathirana, Fabian FrÃűhlich, Jan Hasenauer, and Julio R. Banga. A protocol for dynamic model calibration. *Briefings in Bioinformatics*, 23:bbab387, 2022.

[30] Carlos F. Lopez, Jeremy L. Muhlich, John A. Bachman, and Peter K. Sorger. Programming biological models in python using pysb. *Molecular Systems Biology*, 9:646, 2013.

[31] Helge Hass, Carolin Loos, Elba RaimÃžndez-ÃĄlvarez, Jens Timmer, Jan Hasenauer, and Clemens Kreutz. Benchmark problems for dynamic modeling of intracellular processes. *Bioinformatics*, 35:3073–3082, 2019.

[32] S. Hoops, S. Sahle, R. Gauges, C. Lee, J. Pahle, N. Simus, M. Singhal, L. Xu, P. Mendes, and U. Kummer. Copasi – a complex pathway simulator. *Bioinf.*, 22:3067–3074, 2006.

[33] A. Raue, B. Steiert, M. Schelker, C. Kreutz, T. Maiwald, H. Hass, J. Vanlier, C. TÃűnsing, L. Adlung, R. Engesser, W. Mader, T. Heinemann, J. Hasenauer, M. Schilling, T. HÃűfer, E. Klipp, F. J. Theis, U. KlingmÃijller, B. SchÃűberl, and J.Timmer. Data2dynamics: a modeling environment tailored to parameter estimation in dynamical systems. *Bioinformatics*, 31:3558–3560, 2015.

[34] Fabian FrÃűhlich, Daniel Weindl, Yannik SchÃd'lte, Dilan Pathirana, ÅĄukasz Paszkowski, Glenn Terje Lines, Paul Stapor, and Jan Hasenauer. Amici: High-performance sensitivity analysis for large ordinary differential equation models. *Bioinformatics*, 2021.

[35] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward. Sundials: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transaction Mathematical Software*, 31:363–396, 2005.

[36] Glenn Terje Lines, ÅĄukasz Paszkowski, Leonard Schmiester, Daniel Weindl, Paul Stapor, and Jan Hasenauer. Efficient computation of steady states in large-scale ode models of biochemical reaction networks. *IFAC-PapersOnLine*, 52:32–37, 2019.

[37] A. Fiedler, S. Raeth, F. J. Theis, A. Hausser, and J. Hasenauer. Tailored parameter optimization methods for ordinary differential equation models with steady-state constraints. *BMC Syst. Biol.*, 10, 2016.

[38] Christian TÃűnsing, Jens Timmer, and Clemens Kreutz. Optimal paths between parameter estimates in non-linear ode systems using the nudged elastic band method. *Frontiers in Physics*, 7, 2019.

[39] Philipp StÃd'dter, Yannik SchÃd'lte, Leonard Schmiester, Jan Hasenauer, and Paul L. Stapor. Benchmarking of numerical integration methods for ode models of biological systems. *bioRxiv*, page 2020.09.03.268276, 2020.

[40] Fabian FrÃűhlich, Carolin Loos, and Jan Hasenauer. Scalable inference of ordinary differential equation models of biochemical processes. *Gene Regulatory Networks: Methods and Protocols*, pages 385–422, 2019.

[41] Clemens Kreutz. Guidelines for benchmarking of optimization-based approaches for fitting mathematical models. *Genome Biology*, 20:281, 2019.