Kinetic Evaluations using KinGui 2

Zhenglei Gao

March 12, 2012

1 Introduction

The package KinGui2 is for kinetic evaluations according to the requirements given by the FOCUS Report on Estimating Persistence and Degradation Kinetics (EC Document Reference Sanco/10058/2005, version 2.0, June 2006). It includes a function for conveniently defining the kinetic models with multiple compartments using simple text. It provides all kinetic models recommended in this report and by default does the standard statistical evaluations requested.

Three different 'optimization' algorithms are implemented in this package. In particular the estimation of parameter confidence intervals is much improved as compared to the previous KinGui by providing the methods Iteratively Reweighted Least Squares (IRLS) and Markov Chain Monte Carlo (MCMC).

Note the package can work both in combination with a GUI in KinGUII and as a stand alone R package. It is developed based on the framework set up by Johannes Reinken's R package **mkin**.

2 Theoretical Background

2.1 Methods for Optimization

The package provide the following 3 methods for fitting the parameters of the nonlinear regression defined by the differential equation system which describes the degradation of the substances of interest.

- NLS
- IRLS [1]
- MCMC [2]

2.2 Constrained Optimization

When using the default parametrization, the optimization should satisfy the constriant that the sum of the formation fractions $ff_i, i = 1, \ldots, n$ is 1. Not

Therefore, as a workaround, we transform the formation fractions using the following equations.

$$\begin{aligned} ff_1 &= f_1 \\ ff_2 &= f_2(1 - ff_1) \\ &\cdots \\ ff_n &= f_n(1 - ff_{n-1}) \end{aligned}$$

Inversely, it is

$$\begin{aligned} f_1 &= ff_1 \\ f_2 &= ff_2/(1 - f_1) \\ &\cdots \\ f_n &= ff_n/(1 - f_{n-1}) \end{aligned}$$

A simple mathematical induction will prove that using the transformed $f_i, i = 1, \ldots, n$ satisfies the constraint.

Another possible reparameterization that satisfies these constraints would be

$$\begin{aligned} ff_1 &= \frac{\exp f_1}{\sum \exp f_i} \\ &\cdots \\ ff_n &= \frac{\exp f_1}{\sum \exp f_i} \end{aligned}$$

with $f_1 = 1$ We use the first reparametrization in this version. The second will be implemented in a later version.

2.3 Chi-square error estimation

The smallest relative error(measurement error percentage) resulting in passing the chi-squared test as defined in the FOCUS kinetics report from 2006 is calculated by the function **chi2err**. A few notes:

- With replicate measurements, the calculated mean and observed mean values are used. However, the true replicate values are used for the kinetic fit.
- According to the FK report (page 162-163) time zero is not included in the Chi2 error calculation for metabolites.

3 Examples

3.1 Default kinetic model set up

The two models set up by the following two functions are exactly the same. For R-users, they can use the first way of defining the model. For KinGUII users,

the GUI will generate the long script that will explicitly include everything, e.g., residue data, parameter starting values, in the function inputs.

3.1.1 Simple

```
> complex <- mkinmod.full(
+   parent = list(type = "SFO", to = c("A1", "B1", "C1"),
sink = FALSE),
+   A1 = list(type = "SFO", to = "A2"),
+   B1 = list(type = "SFO"),
+   C1 = list(type = "SFO"),
+   A2 = list(type = "SFO"),
+   inpartri='default',
+   outpartri='default',
+   data=schaefer07_complex_case,
+   weight=NULL)
```

3.1.2 Generated by the GUI

```
> complex <- mkinmod.full(
+ parent = list(time = c(0, 1, 3, 7,14,30,62,100),
+  residue = c(93.2,89.4,79.7,61.1,48.2,15.9,6.5,6),
+  weight = c(   1, 1, 1, 1, 1, 1, 1, 1),
+  to = c("A1","B1","C1"),
+  FF = list(ini   = c(0.1,0.1,0.1),
+            fixed = c(0,0,0),
+            lower = c(0.0,0.0,0.0),
+            upper = c(1.0,1.0,1.0)),
+   sink  = TRUE,
+   type = "SFO",
+   k = list(ini   = 0.1,
+            fixed = 0,
+            lower = 0.0,
+            upper = Inf),
+   M0 = list(ini   = 0.1,
+             fixed = 0,
+             lower = 0.0,
+             upper = Inf)),
+  A1 = list( time = c(0,1,3,7, 14, 30, 62,100),
+  residue = c(NA, NA,0.55,6.87,17.08,21.68,15.77,13.63),
+   weight = c(1,1,1,1,1,1,1,1),
+       to = c("A2"),
+       FF = list(ini   = c(0.1), fixed = c(0),lower =
c(0.0), upper = c(1.0)),
+       sink  = TRUE,
+       type = "SFO",
+       k = list(ini=0.1,fixed=0,lower=0.0,upper=Inf),
+      M0 = list(ini= 0,fixed=1,lower=0.0,upper=Inf)),
+  A2 = list(time = c(   0, 1, 3, 7,14,30,62,100),
+      residue = c(NA,0.55,1.41,0.55,1.29,1.95,3.54,3.86),
+      weight = c(   1, 1, 1, 1, 1, 1, 1, 1),
+      sink  = TRUE,
+      type = "SFO",
+      k = list(ini=0.1,fixed=0,lower=0.0,upper=Inf),
+      M0 = list(ini=0, fixed=1,lower=0.0,upper=Inf)),
+  B1 = list(time = c(    0,1,3,7, 14, 30, 62,100),
+     residue = c(NA,NA,NA,0.55,2.31,15.76,6.36,3.74),
+     weight = c(1,1,1,1,1,1,1,1),
+     sink  = TRUE,
+     type = "SFO",
+     k = list(ini= 0.1,fixed=0,lower=0.0,upper=Inf),
+     M0 = list(ini=0,fixed=1, lower=0.0, upper=Inf)),
+  C1 = list(time = c(0,1,3,7, 14, 30, 62,100),
+      residue = c(NA,0.55,3.2,5.46,12.55,10.45,4.74,4.33),
+      weight = c(1,1,1,1,1,1,1,1),
+      sink  = TRUE,
+      type = "SFO",
+      k = list(ini=0.1,fixed=0, lower=0.0, upper=Inf),
+      M0 = list(ini= 0,fixed=1,lower=0.0, upper=Inf)))
```

3.2 Kinetic model set up for water-sediment studies

```
> a <- mkinmod.full(
+         parent = list(type = "SFO", to = c("A1", "B1",
"C1"),
+                       sink = FALSE),
+         A1 = list(type = "SFO", to = "A2"),
+         B1 = list(type = "SFO"),
+         C1 = list(type = "SFO"),
+         A2 = list(type = "SFO"),
+                   inpartri='water-sediment',
+                   outpartri='water-sediment',
+                   data=schaefer07_complex_case,
+                   weight=NULL)
```

3.3 Fitting the model

3.3.1 NLS

```
> Fit    <- mkinfit.full(
+              complex,
+                 plot      = TRUE,
+                 quiet     = TRUE,
+                 ctr
+                           = kingui.control(
+                                method = 'solnp',
+                             submethod = 'Port',
+                               maxIter = 100,
+                             tolerance = 1E-06,
+                             odesolver = 'lsoda'),
+              irls.control = list(
+                               maxIter = 10,
+                             tolerance = 0.001))
```

```
> kinplot(Fit)
```

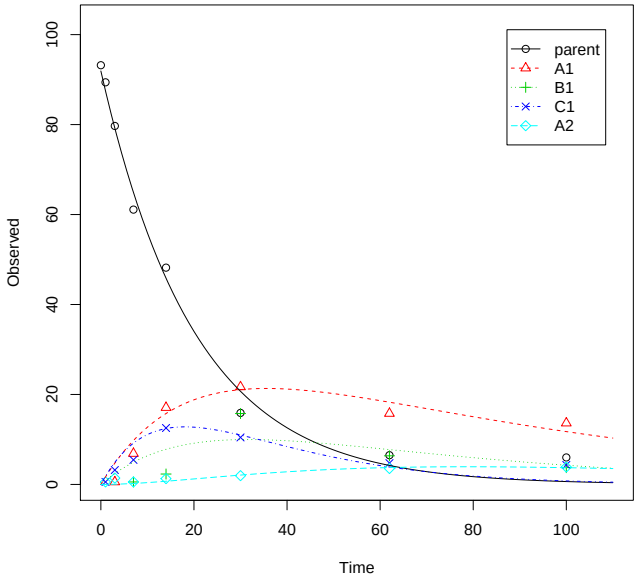


Figure 1: Plot of fitting results using NLS

3.3.2 IRLS

```
> Fit    <- IRLSkinfit.full(
+              complex,
+                 plot      = TRUE,
+                 quiet     = TRUE,
+                 ctr       = kingui.control
+                              (method = 'solnp',
+                              submethod = 'Port',
+                                maxIter = 100,
+                              tolerance = 1E-06,
+                              odesolver = 'lsoda'),
+              irls.control = list(
+                                maxIter = 10,
+                              tolerance = 0.001))
```

3.3.3 MCMC

```
> Fit    <- mcmckinfit.full(
+              complex,
+                 plot      = TRUE,
+                 quiet     = TRUE,
+                 ctr       = kingui.control
+                              (method = 'solnp',
+                              submethod = 'Port',
+                                maxIter = 100,
+                              tolerance = 1E-06,
+                              odesolver = 'lsoda'),
+              irls.control = list(
+                              maxIter = 10,
+                              tolerance = 0.001),
+                              niter=50000,
+                              burninlength = 10000,
+                              outputlength=5000
+                              )
```

3.4 Summary and Report

```
> list2ascii(summary(Fit, cov = TRUE,version=version),
+            'example.kgo')
> kingraph(Fit,'example.kgg')
```

4 Trouble Shooting

4.1 NAs occur in output standard error estimates

In nonlinear models, however, noninvertible Hessians are related to the shape of the posterior density, but how to connect the problem to the question being analyzed can often be extremely difficult. Reasons include:

- Hessian not positive definite, all NAs. This is singularity, which occurs when the objective is flat near the mode, i.e. the objective function is a plateau or a flat line. It indicates the absense of information, and the estimates have infinite variance($1/H \to \infty$ as $H \to 0$)
- Some NA entries $\Rightarrow$ When the Hessian invertable, but contains negative or 0 diagonal values. The optimization might get stuck on a saddle point, or not be able to fine any direction for improvement.
- If the model seems to converge reasonably, but your output shows that some coefficients have (true) zero standard errors, this will almost always be because that coefficient doesn't actually affect the function value. Most commonly, this is just a simple error of including a coefficient doesn't appear in the formula(s) being estimated. More subtly, it can be a parameter which multiplies a series or subcalculation which takes a zero value.

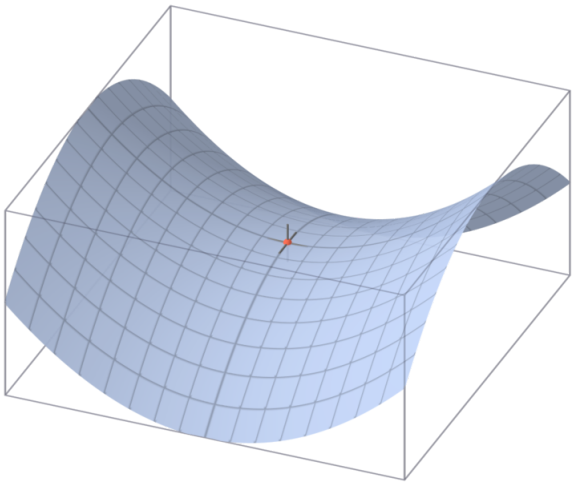


Figure 2: An example of a saddle point

Here is a simple example of going around the "NA problems". This is an output from a dataset provided by Gerald.

```
Optimised parameters:
                  Estimate Std. Error   Lower CI Upper CI   Pr(>t)
```

```
M0_Ipro          8.340e+01  2.088e+00  7.931e+01   87.489  < 2e-16 ***
k_Ipro           2.902e-02  1.983e-03  2.513e-02    0.033  < 2e-16 ***
k_PMPA           1.558e-02  2.918e-03  9.857e-03    0.021 6.39e-07 ***
k_NPMPA          3.227e-10         NA         NA       NA       NA
k_Amino          6.641e-07         NA         NA       NA       NA
f_Ipro_to_PMPA   6.812e-01  6.395e-02  5.558e-01    0.807 3.41e-16 ***
f_Ipro_to_NPMPA  9.381e-01  2.053e-01  5.358e-01    1.341 1.12e-05 ***
f_Ipro_to_Amino  9.996e-01  3.386e+00 -5.638e+00    7.637    0.384
f_NPMPA_to_PMPA  6.967e-01         NA         NA       NA       NA
```

The two 'k' parameters with NA standard error are very close to 0 and the their variability will not affect the objective function as much. A quick thought would be fixing one of them to 0 and rerun the model. Redefine the pathways sometimes would help to solve the numerical problem.

4.2 Choice of optimization algorithms

The program mainly use **solnp**(a general non-linear optimization method using augmented lagrange multiplier) algorithm for optimization purpose. When the program generates insensible results or there are NAs in the report, the user should consider either changing the optimization method or tuning the control parameters for the selected algorithm.

- One probelm for **Rsolnp** is that it may get stuck in a saddlepoint/ridge for very low values of delta for data of a certain scaling, which causes the inversion of the Hessian not available. Depending on the version of the package, the solver either catches these errors (a status code of 2 indicate problems pertaining to the inversion of the Hessian and a code of 0 is locally optimal, and 1 means that the maximum number of iterations was reached without achieving the required tolerance) and reports them or exit prematurely.

  It will also return the parameters up to that point if possible, but it is suggested not to use those parameters to restart the solver, particularly if you are dealing with non-convex/non-smooth problems. Instead, pertrube your original parameters slightly and use a different delta/tol/rho combination.

- LM

A Package Structure

We show a few of the function calling graphs in this section. Calling graphs for some other functions can be found in the *doc* folder.

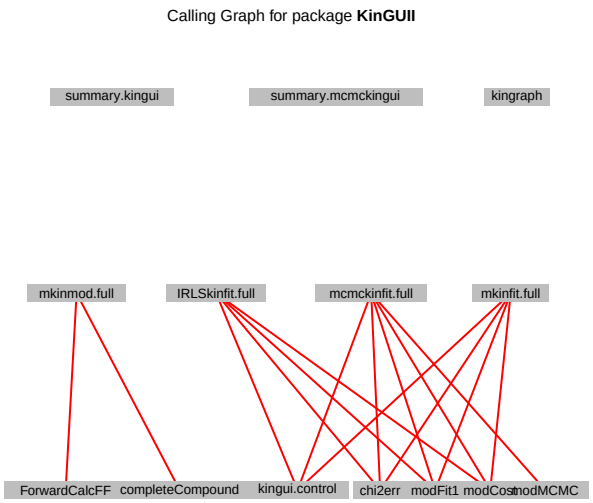
The main functions in this package include **mkinmod.full** for setting up the kinetic model, **mkinfit.full, IRLSkinfit.full, mcmckinfit.full** for fitting

the kinetic model, **summary.kingui, summary.mcmckingui, kingraph** for preparing data for the tables and graphs in the final report.

Every compartment in the kinetic model is a list with components *type, to ,time, residue, sink,M0* and corresponding kinetic parameters for different model type and different study type. For example, we need *k, FF* for SFO model and *k1,k2, g, FF* for HS model . Some of the components can be missing in the input and **CompleteCompound** function will try to set them to default values or reporting an error. The function **mkinmod.full** calls **CompleteCompound** to set up the differiential equations or coefficient matrix to be used in the optimization objective function set up. It will in the same time do the tranformation from the true formation fraction to the "ff"s used for the constrained optimization.

In the next section we will show in detail, using **IRLSkinfit.full** as an example, how we fit the kinetic models.

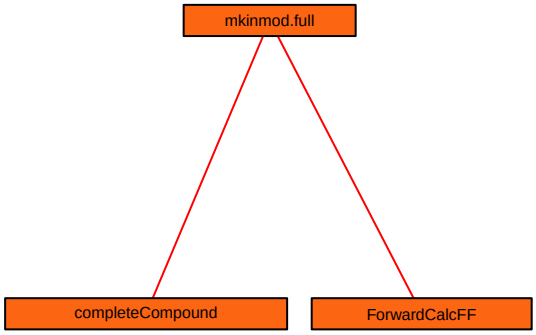
The fitting functions will return an object of class *kingui* or *mcmckingui*,then **summary** can be called for these classes using S3 method.



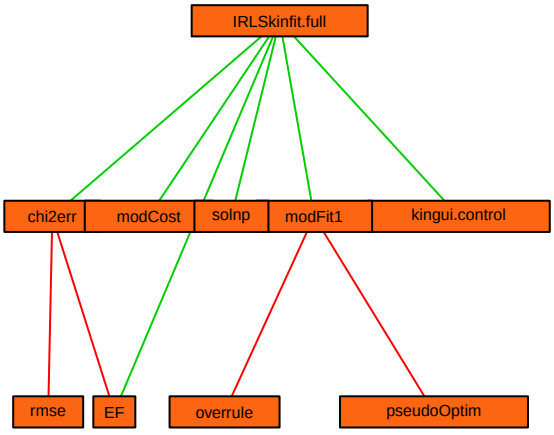
B An example to explain the function structure

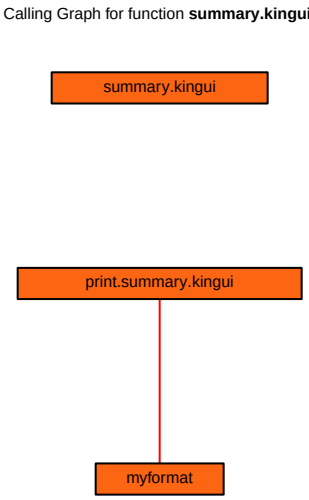
We take one of the main functions **IRLSkinfit.full** as an example and try to explain what it does block by block. The basic structure of this function is shown in Figure 3(A larger file, Xmind file)

Calling Graph for function **mkinmod.full**



Calling Graph for function **IRLSkinfit.full**






```
> IRLSkinfit.full <- function(mkinmodini,
+   eigen = FALSE,
+   plot = FALSE, plottitle='',quiet = FALSE,
+   err = NULL, weight = "none", scaleVar = FALSE,
+   ctr=kingui.control(),irls.control=list(),update=NULL,
+   useHsolnp=FALSE,...)
+ {
+     ##...
+     ## This is the function definition with all the
inputs.
+     ## Please find the help files for the explanation of
all the arguments.
+ }
```

The first step is to set the model input and output parameter settings, whether it is a normal case or a water-sediment study.

```
> ## Get the parametrization for input and output.
> ## Whether it is in the normal or water-sediment study
settings.
> inpartri <- mkinmodini$inpartri
> outpartri <- mkinmodini$outpartri
```

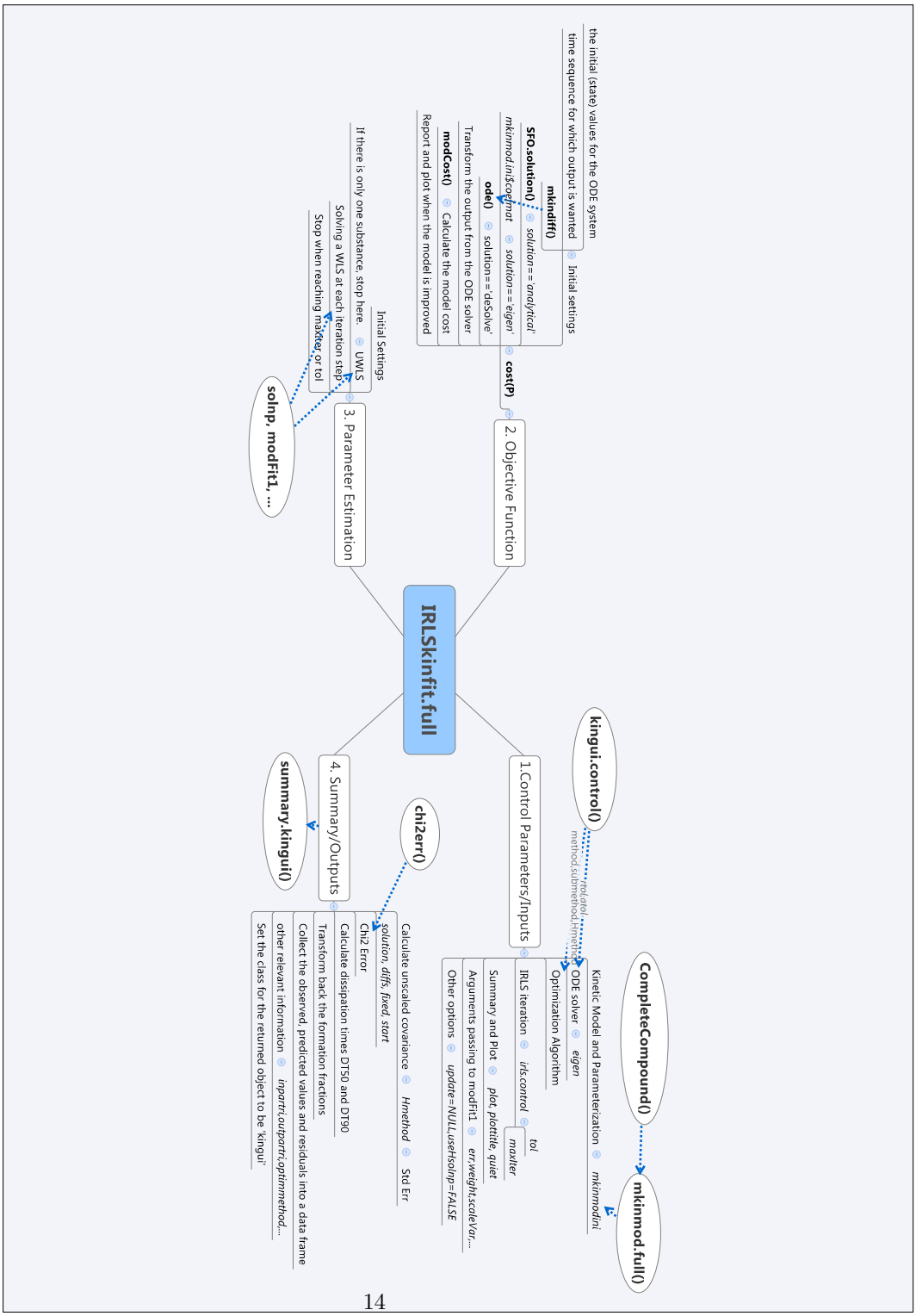


Figure 3: IRLSkinfit.full XMind

B.1 Control Parameters

Next we set the control parameters for the ODE solver and the optimization procedure.

```
> ## ## Control parameters ####
> method <- ctr$method
> odesolver <- ctr$odesolver
> atol <- ctr$atol
> rtol <- ctr$rtol
> control <- ctr$control
> marqctr <- ctr$marqctr
> goMarq <- ctr$goMarq
> submethod <- ctr$submethod
> Hmethod1 <- ctr$Hmethod1
> Hmethod2 <- ctr$Hmethod2
```

Get the initial values, the parameters to be optimized and the fixed ones.

```
> parms.ini <- mkinmodini$parms.ini
> state.ini <- mkinmodini$state.ini
> lower <- mkinmodini$lower
> upper <- mkinmodini$upper
> fixed_parms <- mkinmodini$fixed_parms
> fixed_initials <- mkinmodini$fixed_initials
> mod_vars <- names(mkinmodini$diffs)
> observed<-mkin_wide_to_long(mkinmodini$residue,
+                                  time='time')
> observed$err <-c(as.matrix(mkinmodini$weightmat))
> ## Subset dataframe with mapped (modelled) variables
>     observed<-subset(observed,name %in%
names(mkinmodini$map))
> ## Get names of observed variables
>     ## NOTE HERE: the order may not be the same as the
input mkinmod.full differential equations list.
>     obs_vars = unique(as.character(observed$name))
> ## Name the parameters if they are not named yet
>     ## usually they are already names
>     if(is.null(names(parms.ini))) names(parms.ini) <-
mkinmodini$parms
> if(is.null(names(state.ini))) names(state.ini) <-
mod_vars
> ## Seperate parameters to be optimised and fixed
>     parms.fixed <- parms.ini[fixed_parms]
> optim_parms <- setdiff(names(parms.ini), fixed_parms)
> parms.optim <- parms.ini[optim_parms]
> state.ini.fixed <- state.ini[fixed_initials]
> optim_initials <- setdiff(names(state.ini),
fixed_initials)
> state.ini.optim <- state.ini[optim_initials]
> state.ini.optim.boxnames <- names(state.ini.optim)
> state.ini.fixed.boxnames <- names(state.ini.fixed)
> if(length(state.ini.optim) > 0) {
+         names(state.ini.optim) <-
paste('M0',names(state.ini.optim),sep="_")
+     }
> if(length(state.ini.fixed) > 0) {
+         names(state.ini.fixed) <-
paste('M0',names(state.ini.fixed),sep="_")
+     }
```

If updating from previous fit or new starting point manually set by the user, then just switch the starting values for the parameters to be optimized. The user can change the control parameters for the ODE solver and the optimzation

algorithm as well.

```
> ## # If updating from previous fit or new starting
point.####
>     if(!is.null(update)){
+      ## update can be a fit object from a previous fit or
defined by the user manually.
+      parms.optim <- update$par[optim_parms]
+      state.ini.optim <-update$par[names(state.ini.optim)]
+      if(!is.null(update$ctr)){
+          ctr <- update$ctr
+          ## Control parameters ####
+          method <- ctr$method
+          odesolver <- ctr$odesolver
+          atol <- ctr$atol
+          rtol <- ctr$rtol
+          control <- ctr$control
+      }
+  }
```

Decide whether to use analytical solutions or the default numerical ODE solver for the ODE system.

```
> if (length(mkinmodini$map) == 1) {
+         solution = "analytical"
+     } else {
+         if (is.matrix(mkinmodini$coefmat) & eigen)
solution = "eigen"
+         else solution = "deSolve"
+     }
```

```
> ## Create a function calculating the differentials
specified by the model
> ## if necessary.
> ## This function definition is the same for all kinds
parametrization !!
> if(solution == "deSolve") {
+     mkindiff <- function(t, state, parms) {
+         time <- t
+         diffs <- vector()
+         for (box in mod_vars)
+         {
+             diffname <- paste("d", box, sep="_")
+             diffs[diffname]<-with(
+                as.list(c(time,state,parms)),
+                eval(parse(text=mkinmodini$diffs[[box]])))
+         }
+         return(list(c(diffs)))
+     }
+ }
```

B.2 Set up the objective function for optimization

Inside the **IRLSkinfit.full** function, we define a **cost** function.

```
> ## Define the model cost function, should be the same for
all kinds of optimization ##
>     cost <- function(P){
+         ## omitted here, we explain it in section A.1
+     }
```

Here we describe this internal function block by block.

First we need to get the initial values for the ODE system, set the time sequence for which output is wanted. The differential equations are defined by **mkindiff** function.

```
> ##names(P) <- pnames
> assign("calls", calls+1, inherits=TRUE)
> if(length(state.ini.optim) > 0) {
+     odeini <- c(P[1:length(state.ini.optim)],
state.ini.fixed)
+     names(odeini) <- c(state.ini.optim.boxnames,
+                        state.ini.fixed.boxnames)
+ } else {
+     odeini <- state.ini.fixed
+     names(odeini) <- c( state.ini.fixed.boxnames)
+ }
> ## has to change the odeini order since it is different
from the mod_vars order.
> odeini <- odeini[mod_vars]
> odeparms <- c(P[(length(state.ini.optim) + 1):length(P)],
parms.fixed)
> outtimes = unique(observed$time)
> evalparse <- function(string)
+ {
+     eval(parse(text=string), as.list(c(odeparms,
odeini)))
+ }
```

When there is only one substance, the program will automatically choose to use the analytical solution by calling the **type.solution**(e.g, **SFO.solution**) function.

```
> ## Solve the system
>         if (solution == "analytical") {
+             parent.type = names(mkinmodini$map[[1]])[1]
+             parent.name = names(mkinmodini$diffs)[[1]]
+             o <- switch(parent.type,
+                         SFO = SFO.solution(outtimes,
+                         evalparse(parent.name),
+                         evalparse(paste("k", parent.name,
sep="_"))),
+                         FOMC = FOMC.solution(outtimes,
+                         evalparse(parent.name),
+                         evalparse(paste("alpha",
parent.name,  sep="_")), evalparse(paste("beta",
parent.name,  sep="_"))),
+                         DFOP = DFOP.solution(outtimes,
+                         evalparse(parent.name),
+                         evalparse(paste("k1",
parent.name,  sep="_")), evalparse(paste("k2", parent.name,
sep="_")),evalparse(paste("g", parent.name,  sep="_"))),
+                         HS = HS.solution(outtimes,
+                         evalparse(parent.name),
+                         evalparse(paste("k1",
parent.name,  sep="_")),evalparse(paste("k2", parent.name,
sep="_")),evalparse(paste("tb", parent.name,  sep="_"))),
+                         SFORB = SFORB.solution(outtimes,
+                         evalparse(parent.name),
+                         evalparse(paste("k", parent.name,
"bound", sep="_")),
+                         evalparse(paste("k", sub("free",
"bound", parent.name), "free", sep="_")),
+                         evalparse(paste("k", parent.name,
"sink", sep="_")))
+                         )
+             out <- cbind(outtimes, o)
+             dimnames(out) <- list(outtimes, c("time",
sub("_free", "", parent.name)))
+         }
```

If the user manually sets **eigen=TRUE**, then the program use the *coefmat* derived from **mkinmod.full** to solve the ODE system.

```
> if (solution == "eigen") {
+     coefmat.num <- matrix(sapply(as.vector(
+                    mkinmodini$coefmat), evalparse),
+                   nrow = length(mod_vars))
+     e <- eigen(coefmat.num)
+     zz.ev <- e$values
+     if(min(zz.ev)[1]<0){
+         ## in this case cannot use the eigen solution
+         warning("\'coefmat is not positive definite!\n")
+         solution <- 'deSolve' ## switch to deSolve
methods,define the mkindiff function
+         if(solution == "deSolve") {
+             mkindiff <- function(t, state, parms) {
+                 time <- t
+                 diffs <- vector()
+                 for (box in mod_vars)
+                 {
+                     diffname <- paste("d", box, sep="_")
+                     diffs[diffname] <-
with(as.list(c(time,state, parms)),
+             eval(parse(text=mkinmodini$diffs[[box]])))
+                 }
+                 return(list(c(diffs)))
+             }
+         }
+     }else{
+         cc <- solve(e$vectors, odeini)
+         f.out <- function(t) {
+             e$vectors %*% diag(exp(e$values * t),
nrow=length(mod_vars)) %*% cc
+         }
+         o <- matrix(mapply(f.out, outtimes),
+                     nrow = length(mod_vars), ncol =
length(outtimes))
+         dimnames(o) <- list(mod_vars, outtimes)
+         out <- cbind(time = outtimes, t(o))
+     }
+ }
```

Most often we will use the numerical ODE solvers. The default integrator used is 'lsoda'. If the program exit with an error in this step, using **diagnostics()** to see the diagnostic messages in the debug mode. Different error tolerance level can be tried or switching to another solver.

```
> if (solution == "deSolve")
+         {
+             out <- ode(
+                         y = odeini,
+                         times = outtimes,
+                         func = mkindiff,
+                         parms = odeparms,
+                         atol = atol,
+                         rtol = rtol,
+                         method=odesolver
+                         )
+         }
```

Output transformation for models with unobserved compartments like SFORB.

```
> ## Output transformation for models with unobserved
compartments like SFORB
> out_transformed <- data.frame(time = out[,"time"])
> for (var in names(mkinmodini$map)) {
+     if((length(mkinmodini$map[[var]]) == 1) || solution
== "analytical") {
+         out_transformed[var] <- out[, var]
+     } else {
+         out_transformed[var] <- rowSums(out[,
mkinmodini$map[[var]]])
+     }
+ }
> assign("out_predicted", out_transformed, inherits=TRUE)
> if(sum(apply(out_transformed,2,function(x)
sum(is.nan(x))>nrow(out_transformed)-2))>0)
+ {
+     warning('Integration not completed')
+     out_transformed <- apply(out_transformed,2,function(x)
{if(sum(is.nan(x))>nrow(out_transformed)-2) x <-
rep(Inf,nrow(out_transformed)) else x <- x})
+ }
> if(nrow(out_transformed)<length(outtimes))
+ {
+     tmp <- matrix(0,length(outtimes),
+                   ncol(out_transformed)-1)
+     tmpnames <-names(out_transformed)
+     out_transformed <- data.frame(time=outtimes,tmp)
+     names(out_transformed) <- tmpnames
+ }
```

Calculate the model cost using **modCost**.

```
> mC <- modCost(out_transformed, observed, y = "value",
+   err = 'err', weight = weight, scaleVar = scaleVar)
```

Report and/or plot if the model is improved by a certain level defined by *quiet.tol* defined in **kingui.control**.

```
> if (cost.old-mC$model > ctr$quiet.tol) {
+     if(!quiet) cat("Model cost at call ", calls, ": ",
mC$model, "\n")
+     if(plot) {
+         outtimes_plot = seq(min(observed$time),
max(observed$time), length.out=100)
+         if (solution == "analytical") {
+             ## omit here
+         }
+         if(solution == "eigen") {
+             ## omit here
+         }
+         if (solution == "deSolve") {
+             out_plot <- ode(
+                             y = odeini,
+                             times = outtimes_plot,
+                             func = mkindiff,
+                             parms = odeparms)
+         }
+         out_transformed_plot <- data.frame(time =
out_plot[,"time"])
+         for (var in names(mkinmodini$map)) {
+             if((length(mkinmodini$map[[var]]) == 1) ||
solution == "analytical") {
+                 out_transformed_plot[var] <- out_plot[,
var]
+             } else {
+                 out_transformed_plot[var] <-
rowSums(out_plot[, mkinmodini$map[[var]]])
+             }
+         }
+         plot(0, type="n",
+              xlim = range(observed$time), ylim =
range(observed$value, na.rm=TRUE),
+              xlab = "Time", ylab =
"Observed",main=plottitle)
+         col_obs <- pch_obs <- 1:length(obs_vars)
+         names(col_obs) <- names(pch_obs) <- obs_vars
+         for (obs_var in obs_vars) {
+             points(subset(observed, name == obs_var,
c(time, value)),
+                    pch = pch_obs[obs_var], col =
col_obs[obs_var])
+         }
+         matlines(out_transformed_plot$time,
out_transformed_plot[-1])
+         legend("topright", inset=c(0.05, 0.05),
legend=obs_vars,
+                col=col_obs, pch=pch_obs,
lty=1:length(pch_obs))
+     }
+     assign("cost.old", mC$model, inherits=TRUE)
+ }
```

Return an object with class *modCost*.

```
> return(mC)
```

B.3 Optimization to fit the kinetic model

Set the initial values for the optimization. If argument *plot* is TRUE, then open a new plot window.

```
> ## Set up initial values for the optimization ##
> cost.old <- 1e100 ## old cost(SSR)
> calls <- 0  ## number of calls to the cost function
> out_predicted <- NA ## initialize the predicted matrix of
residue at the observing time points.
> if(plot) x11()## open a new plotting window
> method0 <- 'solnp' ## a prefitting step since this is
usually the most effective method
```

We first use the solnp algorithm to do an initial optimization, if it fails, we try another method.

```
> pnames=names(c(state.ini.optim, parms.optim))
> fn <- function(P){
+     names(P) <- pnames
+     FF<<-cost(P)
+     return(FF$model)}
> a <- try(fit <- solnp(c(state.ini.optim, parms.optim),
+                       fun=fn,LB=lower,UB=upper,
+                       control=control),silent=TRUE)
> ##optimmethod <- method0
> flag <- 1
> if(class(a) == "try-error")
+ {
+     print('solnp fails, try PORT or other algorithm by
users choice, might take longer time. Do something else!')
+     warning('solnp fails, switch to  PORT or other
algorithm by users choice')
+     ## now using submethod already
+     if(method!='solnp') submethod <- method
+     fit <- modFit1(cost, c(state.ini.optim, parms.optim),
+                    lower = lower, upper = upper,
+                    method=submethod,
+                    control=kingui.control(
+                    method=submethod,
+                    tolerance=ctr$control$tol)$control)
+     flag <- 0 ## change the flag of used methods
+     ##optimmethod <- c(optimmethod,method) ## keep for
later possible use.
+ }
```

If there is only one substance in the model, the optimization will stop and related values including *SSR*, *Var* will be calculated. Note that since we always use 'solnp' in the first step. If getting insensible results and you need to change the optimization algorithm, try to use **mkinfit.full** instead of **IRLSkinfit.full**. This shall be fixed in a later version. If there are more than one compartments, set up the IRLS iteration control parameters and start the IRLS iteration which involving solving a weighted least square problem in each iteration step by **solnp** or **modFit1** or other optimization algorithms.

```
> if(length(irls.control)==0) irls.control <-
list(maxIter=5,tol=1e-05)
> if(is.null(irls.control$tol)) tol <- 1e-05 else tol <-
irls.control$tol
> if(is.null(irls.control$maxIter)) maxIter <- 5 else
maxIter <- irls.control$maxIter
> if(length(mkinmodini$map)==1){
+     ## there is only one parent no need to do the
iteration:
+     maxIter <- 0
+     useHsolnp <- TRUE
+     if(flag==1)## fit from solnp
+     {
+         fit$ssr <- fit$values[length(fit$values)]
+         fit$residuals <-FF$residual$res
+         ## mean square per varaible
+         if (class(FF) == "modCost") {
+             names(fit$residuals)  <- FF$residuals$name
+             fit$var_ms            <- FF$var$SSR/FF$var$N
+             fit$var_ms_unscaled   <-
FF$var$SSR.unscaled/FF$var$N
+             fit$var_ms_unweighted <-
FF$var$SSR.unweighted/FF$var$N
+
+             names(fit$var_ms_unweighted) <-
names(fit$var_ms_unscaled) <-
+                 names(fit$var_ms) <- FF$var$name
+         } else fit$var_ms <- fit$var_ms_unweighted <-
fit$var_ms_unscaled <- NA
+     }
+     err1 <- sqrt(fit$var_ms_unweighted)
+     ERR <- err1[as.character(observed$name)]
+     observed$err <-ERR
+ }
> niter <- 1
> ## insure one IRLS iteration setup the initials
> diffsigma <- 100
> olderr <- rep(1,length(mod_vars))
```

The iteration will continue until either the difference between the calculated sigmas are smaller than a tolerance level or the maximum iteration number has been reached.

```
> while(diffsigma>tol & niter<=maxIter)
+ {
+     ##....
+ }
```

The IRLS algorithm involves solving a weighted least square problem in each iteration step by **solnp** or **modFit1** or other optimization algorithms.

```
> ## # other list need to be attached to fit to give
comparable results as in modFit.
> if(flag==1)## fit from solnp
+ {
+     fit$ssr <- fit$values[length(fit$values)]
+     fit$residuals <-FF$residual$res
+     ## mean square per varaible
+     if (class(FF) == "modCost") {
+         names(fit$residuals)  <- FF$residuals$name
+         fit$var_ms            <-
+             FF$var$SSR/FF$var$N
+         fit$var_ms_unscaled   <-
+             FF$var$SSR.unscaled/FF$var$N
+         fit$var_ms_unweighted <-
+             FF$var$SSR.unweighted/FF$var$N
+         names(fit$var_ms_unweighted) <-
names(fit$var_ms_unscaled) <-
+             names(fit$var_ms) <- FF$var$name
+     } else fit$var_ms <- fit$var_ms_unweighted <-
fit$var_ms_unscaled <- NA
+ }
> err1 <- sqrt(fit$var_ms_unweighted)
> ERR <- err1[as.character(observed$name)]
> observed$err <-ERR
> diffsigma <- sum((err1-olderr)^2)
> cat("IRLS iteration at",niter, "; Diff in error variance
", diffsigma,"\n")
> olderr <- err1
```

```
> if(goMarq==1) {
+
+     print('do a local optmization using marquardt')
+     fit <- modFit1(cost, fit$par, lower = lower, upper =
upper, method='Marq',control=marqctr)
+     flag <- 0
+ }else{
+     flag <- 1
+     ## the next iteration using solnp also
+     a <- try(fit <- solnp(fit$par,fun=fn,LB=lower,
+                           UB=upper,control=control),
+              silent=TRUE)
+     if(class(a) == "try-error")
+     {
+         flag <- 0
+         print('solnp fails during IRLS iteration, try
PORT or other algorithm by users choice.This may takes a
while. Do something else!') ## NOTE: because in kingui we
switch off the warnings, we need to print out the message
instead.
+         warning('solnp fails during IRLS iteration,
switch to  PORT or other algorithm by users choice')
+
+         fit <- modFit1(cost, fit$par, lower = lower,
upper = upper, method=submethod,control=list())
+     }
+ }
```

Add to the number of iteration counter.

```
> niter <- niter+1
```

If the optimization is done via **solnp**, a few values like *ssr*, *residuals* need to be renamed or calculated.

```
> ## # other list need to be attached to fit to give
comparable results as in modFit.
> if(flag==1){
+     ## solnp used
+     optimmethod <- 'solnp'
+     fit$ssr <- fit$values[length(fit$values)]
+     fit$residuals <-FF$residual$res
+     ## mean square per varaible
+     if (class(FF) == "modCost") {
+         names(fit$residuals)  <- FF$residuals$name
+         fit$var_ms            <- FF$var$SSR/FF$var$N
+         fit$var_ms_unscaled   <-
+             FF$var$SSR.unscaled/FF$var$N
+         fit$var_ms_unweighted <-
+             FF$var$SSR.unweighted/FF$var$N
+         names(fit$var_ms_unweighted) <-
names(fit$var_ms_unscaled) <-
+             names(fit$var_ms) <- FF$var$name
+     } else fit$var_ms <- fit$var_ms_unweighted <-
fit$var_ms_unscaled <- NA
+     np <- length(c(state.ini.optim, parms.optim))
+     fit$rank <- np
+     fit$df.residual <- length(fit$residuals) - fit$rank
+ }else{
+     optimmethod <- submethod
+ }
```

B.4 Summary Information

Calculating the unscaled covariance to calculate the standard error for the parameter estimates in the **summary** function.

```
> ## Calculating the unscaled covariance
> if(flag!=1) covar <- try(solve(0.5*fit$hessian), silent =
TRUE) else {## solnpused.
+     if(useHsolnp==TRUE) {
+         covar <- try(solve(0.5*fit$hessian), silent =
TRUE)
+         if(sum(fit$hessian==diag(np))==np*np) covar <-
NULL
+     }else covar <- NULL# unscaled covariance
+     fit$solnp.hessian <- fit$hessian
+ }
> if(!is.numeric(covar)){
+     message <- "Cannot estimate covariance directly from
hessian of the optimization"
+     warning(message)
+     print('Now we need to estimate the Hessian matrix
to get the confidence intervals. This may take a while
depending on the problem(Please be patient!)')
+     if(!is.numeric(covar)){
+         fit <- modFit1(cost, fit$par, lower = lower,
upper = upper, method=Hmethod1,control=list())
+         optimmethod <- c(optimmethod,Hmethod1)
+         covar <- fit$covar
+         if(!is.numeric(covar))
+         {
+             message <- "Cannot estimate covariance  from
hessian calculated by gradient or by Hmethod1"
+             warning(message)
+             ## print('go to the third level to calculate
covar')
+             fit <- modFit1(cost, fit$par, lower = lower,
upper = upper, method=Hmethod2,control=list())
+             covar <- fit$covar
+             optimmethod <- c(optimmethod,Hmethod2)
+             if(!is.numeric(fit$covar)){
+                 covar <- fit$covar
+             }else{
+                 covar <- matrix(data = NA, nrow = np,
ncol = np)
+                 warning('covar not estimable')
+             }
+         }else{
+             ##
+         }
+     }
+
+ }else{
+     message <- "ok"
+ }
> rownames(covar) <- colnames(covar) <-pnames
> fit$covar <- covar
```

Here is an extra step if using the trust algorithm to fit the model. Various other information for summary and plotting need to be calculated.

```
> if(method=='trust'){## an extra step if using the trust
algorithm
+     P <- fit$par
+     if (length(state.ini.optim) > 0) {
+         odeini <- c(P[1:length(state.ini.optim)],
state.ini.fixed)
+         names(odeini) <- c(state.ini.optim.boxnames,
state.ini.fixed.boxnames)
+     }
+     else odeini <- state.ini.fixed
+     odeparms <- c(P[(length(state.ini.optim) +
1):length(P)],
+                   parms.fixed)
+                                          #outtimes =
unique(observed$time)
+     out <- ode(y = odeini, times = outtimes, func =
mkindiff,
+                parms = odeparms)
+     out_transformed <- data.frame(time = out[, "time"])
+     for (var in names(mkinmodini$map)) {
+         if (length(mkinmodini$map[[var]]) == 1) {
+             out_transformed[var] <- out[, var]
+         }
+         else {
+             out_transformed[var] <- rowSums(out[,
mkinmodini$map[[var]]])
+         }
+     }
+     assign("out_predicted", out_transformed, inherits =
TRUE)
+ }
> predicted_long <- mkin_wide_to_long(out_predicted, time =
"time")
> fit$predicted <- out_predicted
```

```
> ## We need to return some more data for summary and
plotting
> fit$solution <- solution
> if (solution == "eigen") {
+     fit$coefmat <- mkinmodini$coefmat
+ }
> if (solution == "deSolve") {
+     fit$mkindiff <- mkindiff
+ }
> ## We also need various other information for summary and
plotting
> fit$map <- mkinmodini$map
> fit$diffs <- mkinmodini$diffs
> fit$observed <- mkinmodini$residue
```

Collect initial parameter values in two dataframes *fixed* and *start* to be used in the final report.

```
> if(outpartri=='default'){
+     if(length(state.ini.optim)>0){
+         fit$start0 <- data.frame(initial=state.ini.optim,
+            type=rep("state", length(state.ini.optim)),
+            lower=lower[1:length(state.ini.optim)],
+            upper=upper[1:length(state.ini.optim)])
+     }else{
+         fit$start0 <- data.frame(initial=state.ini.optim,
+          type=rep("state", length(state.ini.optim)),
+          lower=numeric(0),upper=numeric(0))
+     }
+     start0 <- mkinmodini$start[mkinmodini$start$fixed==0,]
+     fit$start0 <- rbind(fit$start0,
+              data.frame(initial=start0$initial,
+                         type=start0$type,
+                         lower=start0$lower,
+                         upper=start0$upper,
+                         row.names =rownames(start0)))
+     fit$fixed0 <- data.frame(value = state.ini.fixed,
+                    type=rep("state",
length(state.ini.fixed)),
+                    by=rep("user", length(state.ini.fixed)))
+     fixed0 <- mkinmodini$start[mkinmodini$start$fixed==1,]
+     if(nrow(fixed0)>0) fit$fixed0 <-
+         rbind(fit$fixed0,
+               data.frame(value=fixed0$initial,
+                          type=fixed0$type,
+                          by=rep('user',nrow(fixed0)),
+                          row.names=rownames(fixed0)))
+ }else{
+     fit$start0 <- NULL
+     fit$fixed0 <- NULL
+ }
> fit$start <- data.frame(initial = c(state.ini.optim,
parms.optim))
> fit$start$type = c(rep("state", length(state.ini.optim)),
rep("deparm", length(parms.optim)))
> fit$start$lower <- lower
> fit$start$upper <- upper
> fit$fixed <-
+     data.frame(value = c(state.ini.fixed, parms.fixed))
> fit$fixed$type = c(rep("state", length(state.ini.fixed)),
rep("deparm", length(parms.fixed)))
> fit$fixed$by <- c(rep("user", length(state.ini.fixed)),
mkinmodini$fixed_flag)
```

Calculate chi2 error levels according to FOCUS (2006) by calling **chi2err()**. 0 values at sample time 0 should not be used. The mean should be used when there are repeated measurements.

```
> observed1 <- observed
> observed1 <- observed[!(observed$time==0 &
observed$value==0),]
> means <- aggregate(value ~ time + name, data =
observed1, mean, na.rm=TRUE)##using the mean of repeated
measurements.
> errdata <- merge(means, predicted_long, by = c("time",
"name"), suffixes = c("_mean", "_pred"))
> errobserved <- merge(observed, predicted_long, by =
c("time", "name"), suffixes = c("_obs", "_pred"))
> errdata <- errdata[order(errdata$time, errdata$name), ]
> errmin.overall <- chi2err(errdata, length(parms.optim) +
length(state.ini.optim),errobserved)
> errmin <- data.frame(err.min = errmin.overall$err.min,
+                      n.optim = errmin.overall$n.optim,
+                      df = errmin.overall$df,
+                      err.sig = errmin.overall$err.sig,
+                      RMSE=errmin.overall$RMSE,
+                      EF=errmin.overall$EF,
+                      R2=errmin.overall$R2)
> rownames(errmin) <- "All data"
```

```
> ## Continued
> for (obs_var in obs_vars)
+ {
+     errdata.var <- subset(errdata, name == obs_var)
+     errobserved.var <- subset(errobserved, name ==
obs_var)
+     if(outpartri=='default'){
+         n.k.optim <- (paste("k", obs_var, sep="_")) %in%
+         (names(parms.optim))+
+          length(grep(paste("f",'.*','to',obs_var,sep="_"),
+          names(parms.optim)))
+     }
+     if(outpartri=='water-sediment'){
+         n.k.optim <- length(grep(paste("k_", obs_var,
'_',sep=""), names(parms.optim)))
+     }
+     n.initials.optim <- as.numeric((paste('M0_',obs_var,
sep="")) %in% (names(state.ini.optim)))
+     n.optim <- n.k.optim + n.initials.optim
+     k1name <- paste("k1", obs_var,  sep="_")
+     k2name <- paste("k2", obs_var,  sep="_")
+     gname <- paste("g", obs_var,  sep="_")
+     tbname <- paste("tb", obs_var,  sep="_")
+     alphaname <- paste("alpha", obs_var,  sep="_")
+     betaname <- paste("beta", obs_var,  sep="_")
+     if (alphaname %in% names(parms.optim)) n.optim <-
n.optim + 1
+     if (betaname %in% names(parms.optim)) n.optim <-
n.optim + 1
+     if (k1name %in% names(parms.optim)) n.optim <-
n.optim + 1
+     if (k2name %in% names(parms.optim)) n.optim <-
n.optim + 1
+     if (gname %in% names(parms.optim)) n.optim <- n.optim
+ 1
+     if (tbname %in% names(parms.optim)) n.optim <-
n.optim + 1
+     errmin.tmp <- chi2err(errdata.var,
n.optim,errobserved.var)
+     errmin[obs_var, c("err.min", "n.optim",
"df",'err.sig','RMSE','EF','R2')] <- errmin.tmp
+ }
> fit$errmin <- errmin
```

Calculate dissipation times DT50 and DT90 and formation fractions, we only take the part with **outpartri** being the default one.

```
> parms.all = c(fit$par, parms.fixed)
> fit$distimes <- data.frame(DT50 =
rep(NA, length(obs_vars)), DT90 = rep(NA,
length(obs_vars)),Kinetic=rep(NA,length(obs_vars)),row.names
= obs_vars)
> if(mkinmodini$outpartri=='default'){
+ }
> if(mkinmodini$outpartri=='water-sediment'){
+  }
```

```
> if(mkinmodini$outpartri=='default'){
+     fit$ff <- vector()
+     ff_names = names(mkinmodini$ff)
+     for (ff_name in ff_names){
+         fit$ff[[ff_name]]=eval(parse( text =
+               mkinmodini$ff[ff_name]),
as.list(parms.all))
+     }
+     for (obs_var in obs_vars) {
+         f_tot <- grep(paste(obs_var, "_",sep=''),
names(fit$ff), value=TRUE)
+         f_exp <- grep(paste(obs_var,
"to",obs_var,sep='_'), names(fit$ff), value=TRUE)
+         f_exp1 <- grep(paste(obs_var,
"to",'sink',sep='_'), names(fit$ff), value=TRUE)
+         fit$ff[[paste(obs_var,'to', "sink", sep="_")]] =
+             1 - sum(fit$ff[f_tot])+sum(fit$ff[f_exp])+
+                 sum(fit$ff[f_exp1])
+         type = names(mkinmodini$map[[obs_var]])[1]
+         k1name <- paste("k1", obs_var,  sep="_")
+         k2name <- paste("k2", obs_var,  sep="_")
+         tbname <- paste("tb", obs_var,  sep="_")
+         ## Here we omit  "FOMC", "DFOP","SFORB"
+         if (type == "SFO") {
+             k_name <- paste("k", obs_var,sep="_")
+             k_tot <- parms.all[k_name]
+             DT50 = log(2)/k_tot
+             DT90 = log(10)/k_tot
+         }
+         if (type == "HS") {
+             k1 = parms.all[k1name]
+             k2 = parms.all[k2name]
+             tb = parms.all[tbname]
+             f <- function(t, x) {
+                 fraction = ifelse(t <= tb, exp(-k1 * t),
exp(-k1 * tb) * exp(-k2 * (t - tb)))
+                 (fraction - (1 - x/100))^2
+             }
+             DTmax <- 1000
+             hso1 <- nlminb(0.0001,f, x=50)
+             hso2 <- nlminb(tb,f, x=50)
+             DT50.o <- ifelse(
+                   hso1$objective<=hso2$objective,
hso1$par,hso2$par)
+             DT50 = ifelse(DTmax - DT50.o < 0.1, NA,
DT50.o)
+             ## Here we omit DT90 calculation part
+         }
+         fit$distimes[obs_var, ] =
c(ifelse(is.na(DT50),NA,formatC(DT50,4,format='f')),
ifelse(is.na(DT90),NA,formatC(DT90,4,format='f')),type)
+     }
+ }
```

Collect observed, predicted and residuals into a data frame to write into the final report.

```
> ## Collect observed, predicted and residuals
> observed0 <-  mkin_wide_to_long(mkinmodini$data0,
+                                 time='time')
> observed0$err <- observed$err
> data <- merge(observed, predicted_long, by = c("time",
"name"))
> data0 <- merge(observed0, predicted_long, by = c("time",
"name"))
> names(data) <- c("time", "variable",
"observed","err-std", "predicted")
> names(data0) <- c("time", "variable",
"observed","err-std", "predicted")
> data$residual <- data$observed - data$predicted
> data0$residual <- data$residual
> data$variable <- ordered(data$variable, levels =
obs_vars)
> data0$variable <- data$variable
> tmpid <- is.na(data0$residual) & !is.na(data0$observed)
> data0$'err-std'[tmpid] <- 0
> fit$data <- data[order(data$variable, data$time), ]
> fit$data0 <- data0[order(data0$variable, data0$time), ]
> fit$atol <- atol
> fit$inpartri <- inpartri
> fit$outpartri <- outpartri
> fit$optimmethod <- optimmethod
> class(fit) <- c('kingui',"mkinfit", "modFit")
```

Return the fit object of class "kingui" for summary and plot in the next step.

```
> class(fit) <- c('kingui',"mkinfit", "modFit")
> return(fit)
```

C Change Log

- Version: 1.2012.0131.1200 Fix DT90 calculation problems from Ian Hardy.
- Version: 1.2012.0109.1200 Chi2 error calculation problems from Ian Hardy.
- Version: 1.2011.1122.1607 NA for parent parameters.
- Version: 1.2011.1117.1330 missing R2 problems fixed
- Version: 1.2011.1111.1630 DFOP DT50 problems fixed

- Version: 1.2011.1108.1630 weight 0 problems solved
- Version: 1.2011.1101.1630 weight 0 problems partly solved
- Version: 1.2011.1001.1530
  - under folder doc: adding a vignette.
  - under folder tests: adding a few test cases.
- Version: 1.2011.930.1430
  - summary.mcmckingui: fix MCMC súmmary bugs.
  - plot.mcmckingui: fix traceplot bugs.
- Version: 1.2011.922.1530
  - summary.kingui: fix the confilict between the Viewer and the R output when the sink compartment is turned off.
  - Adding mkinmod.full, *kinfit.full functions to analyze the water-sediment studies. Also adding more options, like entering data as a text/table directly from a single input argument instead of relying on the GUI for data inputs. For R-users without GUI to use the R sourcefunctions for writing their own scripts the scripts will look short and simple, no need to .

References

[1] Z. Gao, J. Green, J. Vanderborght, and W. Schmitt. Improving uncertainty analysis in kinetic evaluations using iteratively reweighted least squares. *Environ Toxicol Chem*, 2011.

[2] L. Görlitz, Z. Gao, and W. Schmitt. Statistical analysis of chemical transformation kinetics using Markov-Chain monte carlo methods. *Environ Sci Technol*, 45(10):4429–37, 2011.