First I want to apologise for any misunderstandings this document will create due to the fact that I am neither an expert in statistics nor programming and thus can easily get the terminology wrong. Please, take all the content with a grain of salt and use the program provided as the basis for your understanding.

1 Overview of current status and functionality

The code base provided does include part of the functionality of PFIM but not all. On the other hand it also has additional features that come in handy.

The current status is that the code was written with the idea in mind to convert it into an R-package, but without having done all the necessary documentation, etc., i.e., it is code in development – and if there would not have been the collaboration with Doug it still would have stayed a small project with slow, but steady progress.

1.1 General program functionality

The normal execution (using the main functions) would be as follows: Read in all the functions from the R-subfolder. Then use the specific functions to get to the final results...

model.readPFIM reads in an example using the original PFIM structure

model.defaultXYZ reads in a predefined example based on my new model structure

model.define creates the model structure based on user input or an example provided from model.readPFIM or model.defaultXYZ

model.plot plots the solution (or the sensitivities) of the provided model based on the given arms, doses, samples, etc. - equivalent to the "initial design"

model.solve wrapper for model.solveAE and model.solveDE for plain solution of the model based on algebraic equation or differential equations
This function is in general not called by the user, but by model.plot or model.sens

model.sens wrapper for model.solveAE, model.sensNum, and model.sensJac. The first includes the possibility to derive the sensitivities from a parsed model (as PFIM). model.sensNum derives the partial derivatives numerically using the Jacobian-function. model.sensJac solves a system of ODEs that simultaneously derives the ODE solution and its partial derivative (huge speedup).

model.fish contains for-loop for the different arms and calls model.fish.bloc to actually derive the PFIM for a specific given protocol

model.out.det_rse derives the standard PFIM output for a given population fisher information matrix

In the tests-subfolder there are three files to test specific parts of the program

run.testSens compares the three different ways to obtain the sensitivities (Hessian as in PFIM, Jacobian, or in parallel with ODEs)

run.testFishBloc code to test subparts of the PFIM calculation to see e.g., if `sum(diag(A %*% B)) == sum(A * B)` for symmetric matrices...

run.testVsPFIM compares the performance and outcomes of myPFIMv2 with PFIM based on PFIM examples

1.2 Possibilities to specify a model

Currently the models can be defined either using my own structure (with a bit of a user-friendly interface) or one can read in a PFIM model specification.

One can specify the model equations

- using the parser as in PFIM
- using a function(time, modelParameters, armsParameters, modelStructure)
- providing a differential equation function(time, states, modelParameters, armsParameters)
- providing ODE including partial derivatives for faster computation

1.3 Deriving the sensitivities to parameters

As mentioned before the code includes the possibility to derive the sensitivities from a parsed model (as in PFIM). model.sensNum derives the partial derivatives numerically using the Jacobian-function. model.sensJac solves a system of ODEs that simultaneously derives the ODE solution and its partial derivative (huge speedup but needs a mathematically skilled person and some patience to set up for a specific example).

1.4 Arms, groups, doses, etc.

I wanted flexibility and maybe tried to get too much of it. **Currently no covariates can be specified** (which would also distinguish two arms or subgroups). But doses and design decisions, etc. can be passed through the program to obtain different protocols or design layouts, dosing regimens, etc. for different arms (it is just a list of parameters passed through to the model function).

For doses in ODE models I use the nice functionality of *deSolve* to specify specific times at which the state vector can be updated, for instance by adding a dose to compartment one or four or ...

1.5 PFIM

Currently no IOV is incorporated which means that I just managed the first steps, but did not have sufficient time to investigate how to encode the IOV clean and modular. Additionally, most of the designs investigated by me an co-workers so far were parallel designs.

The original PFIM code did not make use of the fact that almost all matrices involved are symmetric matrices: some easy simplifications for speedup and numerical robustness are available. Also, one should not use `solve(var)` to obtain the inverse of the variance matrix (I think Doug mentioned that as well). I use

```
mEigen <- eigen(var, symmetric=T)
invvar <- tcrossprod(mEigen$vectors %*% diag(1/mEigen$values), mEigen$vectors)
```

This might not be the optimal solution, but it is more robust than using solve especially when the example is close to singular (I cannot remember if Cholesky would be happy about a singular matrix).

1.6 Optimisation

bf Currently no optimization is included. Often we are more interested in an approximation of the expected standard errors of parameter estimates for a given design than in fiddling around with the time points of the samples.

2 Model list 'structure'

Example given in model.defaultDEjac3 (that is the most complex so far...)

```
model.defaultDEjac3 <- function() {

default <- list()
default$PFIMmodel <- myDEJfunc3 # name of the ODE function (see below)
default$Type <- 'DE'

# dosis per arm - first a vector of times then a vector of doses
default$dosing <- list(cbind(c(0,12,24),c(5*70/0.15,10*70/0.15,10*70/0.15)))

default$parArmsIC <- list(expression(c(0, RateT*V/CLT))) # similar to PFIM

# For the fast computation of sensitivities:
# Partial derivative of ICs wrt parameters - given per arm
default$JacIC=list(expression(c(0,0, 0,RateT/CLT, 0,0,
                                  0,-RateT*V/CLT^2, 0,0, 0,V/CLT)))
# Partial derivative function for observations
# (if not all states are observed)
default$JacObs=TRUE
```

```
# Specification of model parameters, values and variances
default$parModelName <- c("Kd","V","CLD","CLT","CLC","RateT")
default$parModel <- c(350,6,0.2,1,0.4,40)
default$parModelVar <- c(0.05,0.30,0.2,0.25,0.25,0.50)
default$parModelVarType <- 'exp'

# Specification of observations and add/prop error
default$parObsName <- c('TotDrug','TotTarget','Complex')
default$parObsErr <- list(c(0,0.2), c(0,0.1), c(0.1,0.25))

# Times of observation per arm and observation
default$parObsTimes <- list(
    c(10/60/24,1/24,6/24,0.5, 2, 4, 7, 14, 21, 35, 49, 63, 77, 91),
    c(10/60/24, 7, 35, 77, 91),
    c(0.5, 2, 4, 7, 14, 21, 35, 49, 63, 77, 91))

# Currently I need a 'dummy' arm - this should be changed soon
default$parArmsName <- c('dummy')
default$parArms  <- list(c(0))
default$ArmsName <- list('TMDD model')

# For plotting routine
default$TimeName <- 'time (hr)'
default$tRange <- c(0,111)

# Empty...
default$mpOpt <- list()

# To derive PFIM in the end multiply PFIM from protocol with Nr subjects
default$subjects <- c(100)

# Return value: example structure
return(default)
}

# ODE function definition
myDEJfunc3<-function(t, y, p, parArms, mpOpt){
...

# If only the 2 initial vectors are present then numerical sensitivities
# get calculated. If JacState and JacParam are provided then it is assumed
# that the two states (with derivatives dTD and dTT) are observed.
# Otherwise one also needs the partial derivative of observations w.r.t. states.
return(list(c(dTD,dTT), c(TD,TT, C), JacState, JacParam, JacObsParam, JacObsState))
}
```

3 Some possible speedups (without using C-code) and numerical improvements

3.1 Jacobian vs Hessian

The main speedup between PFIM and myPFIMv2 is that for calculating the numerical sensitivities (model.sensNum) I use the function `jacobian()` from the package *numDeriv*. This derives all sensitivities within one function call! In the original PFIM the function `fdHess()$gradient` from the package *nlme* which first derives the Hessian and then the gradient is approximated. Additionally this is done per time-point so for many sampling times this is very time consuming.

3.2 Parallel solving of ODE and sensitivities

As mentioned before: with some mathematics one can get a 100-fold speedup regarding solution of the sensitivities, but it needs some patience as well. See examples model.defaultDEjacY and model.sensJac for details.

3.3 invvar via eigendecomposition

As described above one should avoid deriving inverse matrices at all. I currently still derive the inverse, but as a pseudo-inverse using eigendecomposition. This provides much more robust results (and no crash of the program after 20 minutes of calculation...)

3.4 Use of symmetries when deriving PFIM

Almost all matrices used in the model.fish.bloc function are symmetrical. This should be used to speed up the computation (only minor improvements as matrices are small) and also to increase numerical accuracy (avoid divisions, etc.)

The main change here was that as mentioned before the trace of a matrix multiplication of symmetric matrices can be derived by the sum of their elementwise product, i.e., `sum(diag(A %*% B)) == sum(A * B)`