Training of boolean logic models of signalling networks using prior knowledge networks and perturbation data with *CellNOptR*

Camille Terfve, Julio Saez-Rodriguez

August 11, 2011

Contents

1 A few words before we start...

This software is written in the R language, so in order to use it you will need to have R installed on your computer. For more information and download of R, please refer to `http://www.r-project.org/`. For more information about how to install R packages, please refer to `http://cran.r-project.org/doc/manuals/R-admin.html#Installing-packages`. This package relies on a bioconductor package called RBGL, which you can install by typing:

```
> source("http://bioconductor.org/biocLite.R")
> biocLite("RBGL")
```

Before starting this tutorial you also need to install the package *CellNOptR*. On mac and linux machines you can install a source package by typing:

```
> install.packages("path_to_CellNOptR/CellNOptR_1.2.1.tar.gz",
+     repos = NULL)
```

or, using the R GUI by clicking on "Packages & Data" then "Package installer", then choosing "local source" from the dropdown menu, clicking "install", choosing *CellNOptR_1.2.1.tar.gz* and finally clicking "open".

On windows you can install a source package by typing:

```
> install.packages("path_to_CellNOptR/CellNOptR_1.2.1.tar.gz",
+     type = "source", repos = NULL)
```

You can also install the binary package from the local zip file by typing

```
> install.packages("path_to_CellNOptR/CellNOptR_1.2.1.zip", repos = NULL)
```

or install the local binary from the menu in the GUI. A series of books about R can be found on the R project website (`http://www.r-project.org/`), and many tutorials are available on the internet. If you are a complete beginner, all you need to know is that by typing "?nameOfFunction" you get the help page about the function that you are interested in, and by typing

```
> a <- 4 + 5
```

you assign the value 4+5=9 to the variable a, and if the expression on the right of the arrow is a function then you assign the result of the function to the variable on the left side of the arrow. If you type the name of your variable only, R shows you the content of this variable.

2 Introduction

The package *CellNOptR* integrates prior knowledge about protein signalling networks and perturbation data to infer functional characteristics of a signalling network. This package is based on methods described in[4], and implemented in the Matlab toolbox *CellNOpt* (available at `http://www.ebi.ac.uk/saezrodriguez/software.html#CellNetOptimizer`). More information about the methods and application of the Matlab pipeline can be found in reference [2]. This is a reduced version of *CellNOpt* that performs optimisation of boolean networks, but continuous (ordinary differential equation-based) and constrained fuzzy logic versions are in development in the Matlab pipeline. This package also includes some data importing and normalising capabilities that are performed using the *DataRail* toolbox [3] in the MatLab pipeline (available at `http://www.ebi.ac.uk/saezrodriguez/software.html#DataRail`).

A typical analysis using *CellNOptR* will be described here in detail on a toy example and on a realistic example. The whole analysis can also be performed in one step using a wrapper function, and this is what is described in section 6.

You can either copy and paste the lines of code in this document (not including the >) or type them yourself. For easiness of use we also provide (in the doc folder of this package) a text file CellNOptR-examples.R which contains the R commands in this tutorial, as well as other examples of code. We recommend that you use that script as a template for your analyses. In the doc folder of this package you will also find a file called CellNOptR0_1flowchart.pdf which graphically describes the pipeline of this package and places all of the functions of this package relative to each other. Squared nodes in this flowchart are steps that should not be omitted in a normal analysis, whereas elliptic nodes are optional in most analyses.

The first step of an analysis with *CellNOptR* is to load the library, and create a directory where you can perform your analysis.

```
> library(CellNOptR)
```

```
> dir.create("CNOR_analysis")
> setwd("CNOR_analysis")
```

3 Loading the data and prior knowledge network.

The example that we use is the toy model example from CellNOpt, which is a data set and associated network that have been created in silico. This data and network can be found in the inst/ToyModel directory of this package. The data is read using the function readMIDAS, which as the name states expects a MIDAS formatted csv file (see the documentation of DataRail and [3] for more information about that file format).

Please note that this data is already normalised for boolean modelling. If it had not been the case we would have had to normalise the data first to scale it between 0 and 1, which can be done using the *normaliseCNOlist* function of CellNOptR (see the help of this function for more information about the normalisation procedure). This normalisation procedure is the one used in [4] as implemented in *DataRail*.

```
> cpfile <- dir(system.file("ToyModel", package = "CellNOptR"),
+     full = TRUE)
> file.copy(from = cpfile, to = getwd(), overwrite = TRUE)
> dataToy <- readMIDAS(MIDASfile = "ToyDataMMB.csv")
> CNOlistToy <- makeCNOlist(dataset = dataToy, subfield = FALSE)
```

For simplicity reasons, the above steps are not computed here and instead we will load the data already formatted as a CNOlist.

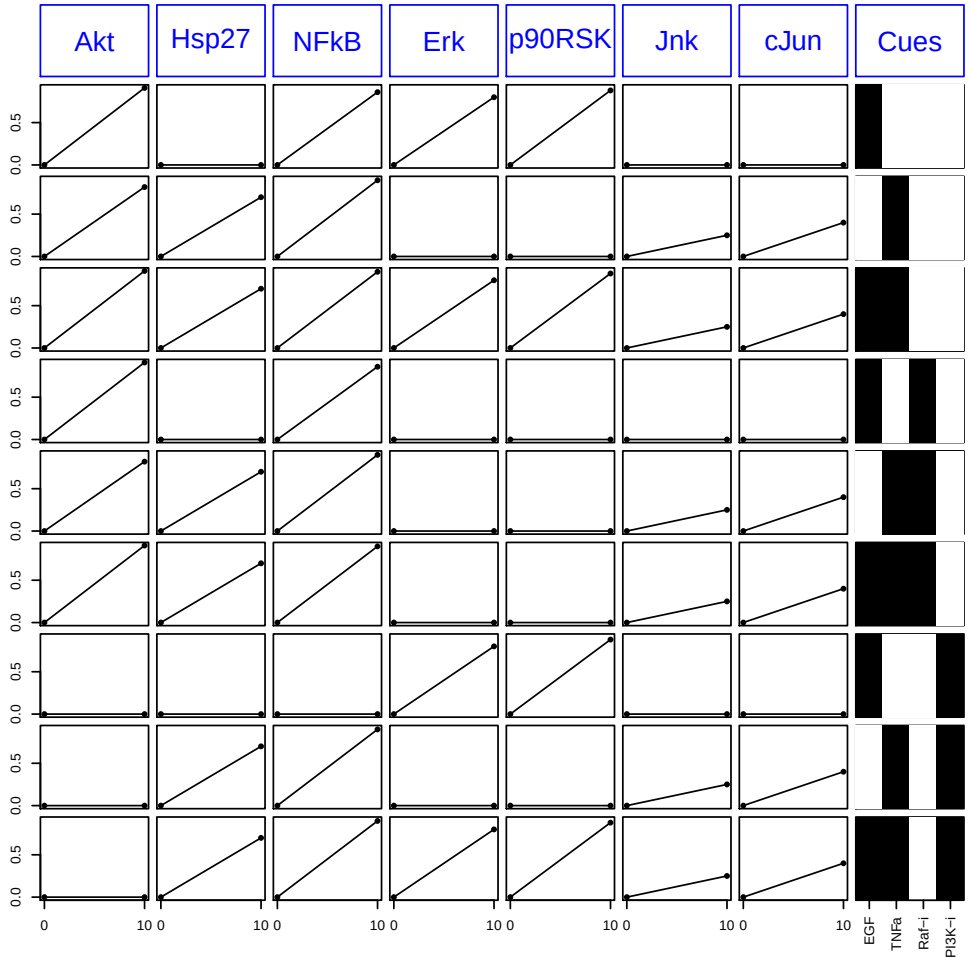
```
> data(CNOlistToy, package = "CellNOptR")
```

A CNOlist is the central data object of this package, it is equivalent to the CNOproject object in *CellNOpt*. It is the object that contains measurements of elements of a prior knowledge network under different combinations of perturbations of other nodes in the network. A CNOlist comprises the following fields: namesSignals, namesCues, namesStimuli, namesInhibitors which are vectors holding the names of the measured, stimulated and inhibited, stimulated, and inhibited species, respectively. The fields valueCues (and its derivatives valueStimuli and valueInhibitors) are boolean matrices that contain for each condition (row) a 1 when the corresponding cue (column) is present, and a zero otherwise. You can have a look at your data and the CNOlist format by typing:

```
> CNOlistToy
```

You can also visualise your data using the function *plotCNOlist* which will produce a plot on your screen with a subplot for each signal and each condition, and an image plot for each condition that contains the information about which cues are present in each condition. This plot can also be produced and stored in your working directory as a single pdf file using the function *plotCNOlistPDF*.

```
> plotCNOlist(CNOlistToy)
```




```
> plotCNOlistPDF(CNOlist = CNOlistToy, fileName = "ToyModelGraph.pdf")
```

We then load the prior knowledge network (PKN), contained in a cytoscape sif format file, using the function *readSif* (this step is not computed here because the model is loaded as a R data object already formatted, similarly to what was done above for the CNOlist). Cytoscape [5] is a software for network visualisation and analysis. You can build a network within cytoscape and simply save it as the default sif file format, which can then be imported in *CellNOptR*. If you choose to do this, then you should make sure that if you have 'and' gates in your network they are present as dummy nodes named 'and' followed by a number from 1 to the number of 'and' nodes that you have.

Alternatively, you can create your network file as a text file formatted as a sif file. Briefly, the expected file format is a tab delimited text file with a line for each directed interaction and the following three elements per line: name of source node, 1 or -1 if the source node is activating or inhibiting the target node, name of target node. The names of the species in the model must match some of the nodes in the model (and this is case sensitive). 'And' hyperedges are expected to be represented in the sif file as dummy nodes named 'and' followed by a number. For example if you have an interaction of the type 'a & b=c', your sif file should contain the following three rows: 'a 1 and1', 'b 1 and1', 'and1 1 c'. Please be aware that when building the scaffold network for optimisation, the software will create all possible 'and' combinations (with maximum 3 inputs) of edges coming into each node, so in the general case it is not necessary to put 2 or 3 input 'and' hyperedges in the prior knowledge network since the software will create them if the corresponding single edges are present.

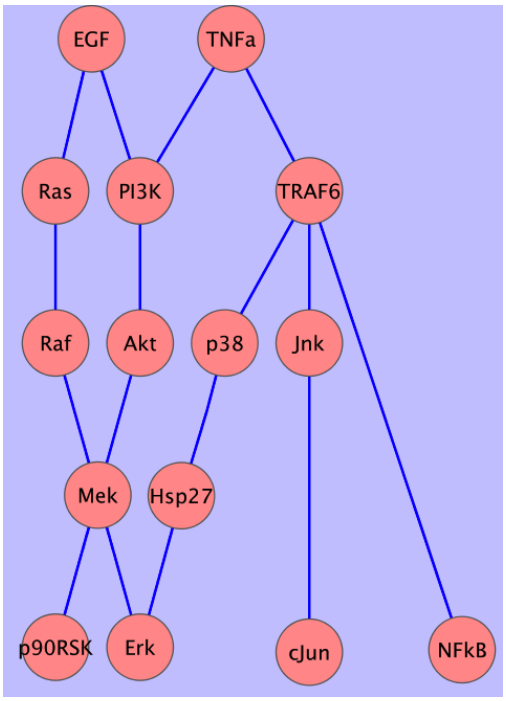


Figure 1: Prior knowledge network (original sif file visualised by cytoscape) for the Toy Model example.

```
> ToyModel <- readSif(sifFile = "ToyPKNMMB.sif")
```

```
> data(ToyModel, package = "CellNOptR")
```

Having loaded both the data set and corresponding model, we run a check to make sure that our data and model were correctly loaded and that our data matches our model (i.e. that species that were inhibited/stimulated/measured in our data set are present in our model). If this is the case, then we can look for the indices of the species that are present in our dataset in our model, using *indexFinder*. This step will have to be repeated every time an operation is done on the model that removes or reorders species.

```
> checkSignals(CNOlistToy, ToyModel)
> indicesToy <- indexFinder(CNOlistToy, ToyModel, verbose = TRUE)
```

```
[1] "The following species are measured: Akt, Hsp27, NFkB, Erk, p90RSK, Jnk, cJun"
[1] "The following species are stimulated: EGF, TNFa"
[1] "The following species are inhibited: Raf, PI3K"
```

The sif model that you have just loaded is visible on figure 1 as displayed by cytoscape.

4 Preprocessing the model

Prior to optimisation, the model has to be pre-processed in 3 steps: removal of non-observable/non-controllable species, compression, and expansion. Each one of these steps is described in more details below.

4.1 Finding and cutting the non observable and non controllable species

Non observable nodes are those that do not have a path to any measured species in the PKN, whereas non controllable nodes are those that do not receive any information from a species that is perturbed in the data.

As we won't be able to conclude anything about these species, we will find them and remove them from the model. Please note that in this particular case there are no nodes to cut, but we still include these steps here because they are necessary in a general case.

```
> ToyNCNOindices <- findNONC(ToyModel, indicesToy, verbose = TRUE)

[1] "The following species are not observable and/or not controllable: "

> ToyNCNOcut <- cutNONC(ToyModel, ToyNCNOindices)
> indicesToyNCNOcut <- indexFinder(CNOlistToy, ToyNCNOcut)
```

4.2 Compressing the model

Compressing the model consists of collapsing paths in which a series of non measured or perturbed nodes input into a measured or perturbed node. This step is performed because such paths do not bring any additional information compared to their compressed version, and unnecessarily complicate the model. Typically this includes linear cascades for examples, but this excludes any node that would be:

1. involved in complex logics (more than one input and also more than one output)
2. involved in self loops

Compression is performed using the function *compressModel*.

```
> ToyNCNOcutComp <- compressModel(ToyNCNOcut, indicesToyNCNOcut)
> indicesToyNCNOcutComp <- indexFinder(CNOlistToy, ToyNCNOcutComp)
```

4.3 Expanding the gates

The last preprocessing step consists in expanding the gates present in the PKN, i.e. creating new logic combinations of gates from the ones present in the prior knowledge network. This is performed in 2 steps: i) any AND node present in the PKN is split into its constituent branches, and ii) every time a nodes gets more than one input, then all 'AND' combinations of the inputs are produced, although only exploring AND gates with 2 and 3 input nodes. This step is performed because although connections between nodes might be known or inferred from functional relationships, the particular logic with which these interactions work or are combined to influence a target node are generally not known.

This step, performed by the function *expandGates*, will create additional fields *SplitANDs* and *newANDs* in the model that inform you about new edges that have been crated from splitting 'AND' hyperedges, and about new hyperedges that have been created from combinations of edges.

```
> ToyNCNOcutCompExp <- expandGates(ToyNCNOcutComp)
```

5 Training of the model

By 'optimising the model', we mean exploring the space of possible combinations of expanded gates in the PKN in order to find the combination that reproduces most closely the data. Comparison between model and data is obtained by simulating the steady state behaviour of the model under all conditions present in the data, and comparing these binary values to the normalised data points. The match between data and model is quantified using an objective function with parameters *sizeFac* and *NAFac*. This function is the sum of a term that computes the fit of the simulated data to the experimental data, a term that penalises increased model size (weighted by the parameter *sizeFac*), and a term that penalises NAs in the output of the simulation (i.e. nodes that are in a non resolved state, typically negative feedbacks; weighted by the parameter *NAFac*). Typically this has the following structure: $\frac{1}{n}\sum_{t,l,k}(M_{t,l,k} - D_{t,l,k})^2 + \alpha\frac{1}{s}\sum_{edges} e_{edges} + \beta n_{NA}$, where n is

the number of data points, M the model output for time t, readout l, condition k, D is the corresponding measurement, α is the size factor, e is the number of inputs for the egde considered (where *edges* are all edges present in the optimised model), s is the number of hyperdeges in the model, β is the NA factor, and n_{NA} is the number of undetermined values returned by the model.

The optimisation itself is done using a genetic algorithm that tries to optimise a string of 0s and 1s denoting the presence or absence of each gate in the model, where the fitness of each individual string is obtained based on the value of the objective function (score). This genetic algorithm uses the following methods: random initialisation of the population (although an initial string is given to the algorithm in the parameter *initBstring*) of size set by *PopSize*, linear ranking based on the scores for fitness assignment (with a default selective pressure of 1.2, set by the parameter *SelPress*), stochastic uniform sampling for selection (with an *elitism* parameter that allows the best x strings to be carried on 'as is' to the next generation), uniform crossover probability, and a mutation probability over the sequence set to 0.5 as default (set by *Pmutation*). The search can be stopped using three conditions: a maximum time in seconds (*MaxTime*), a maximum number of generations (*maxGens*), and a maximum number of stall generations (i.e. generations where the best string is identical, *StallGenMax*).

The genetic algorithm function returns a list object that collects a number of informations such as the best string and corresponding score at each iteration, the average fit at each generation, etc. The function also returns strings that were obtained across the whole optimisation process and that obtained scores that were closed to the best string, where 'close' is defined by a relative tolerance on the score which is set by the parameter *RelTol*. This is an important piece of information because when the data cannot constrain the model tightly then many strings are obtained with a fit that is close to the optimal one, and interpretation of edges present in the optimal model is therefore more subtle.

We start off by computing the residual error, which is the minimum error that is unavoidable with a boolean network and comes from the discrete nature of such a model (please remember that although the data is normalised in this pipeline, it is not discretised, and therefore we compare 0/1 values to continuous values between 0 and 1). This value is important because however good is our optimisation, the value of the goodness of fit term cannot go under this residual error. Then we prepare the model for simulation by applying the function *prep4Sim* which adds fields that are needed for the simulation engine. We also create an initial bit string for the optimisation, which in this case is just a string of 1s, but could be a meaningful string if you have prior expectation about the topology of the model.

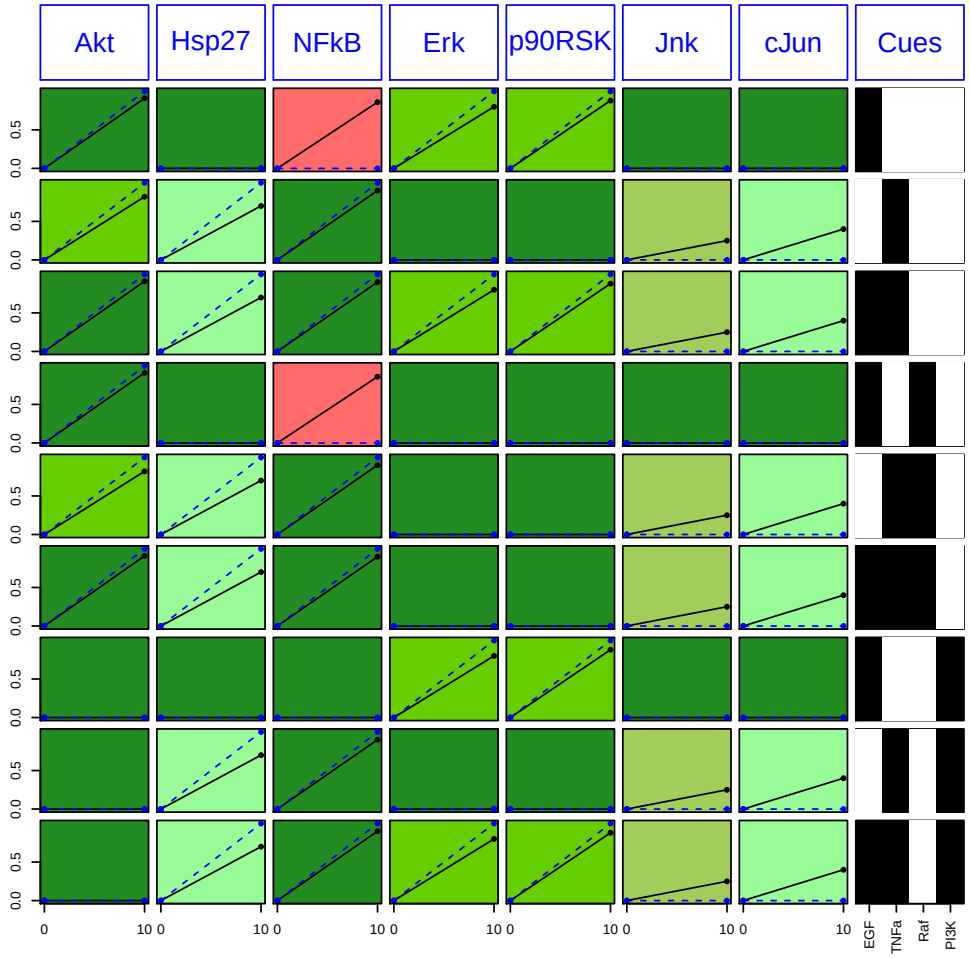
```
> resECNOlistToy <- ResidualError(CNOlistToy)
> ToyFields4Sim <- prep4Sim(ToyNCNOcutCompExp)
> initBstring <- rep(1, length(ToyNCNOcutCompExp$reacID))
```

We can now start the optimisation, in this case with default values for all non essential parameters of the genetic algorithm. If you set the argument verbose to TRUE, this function will print the following information, at each generation: generation number, best score and best string at this generation, stall generation number, average score of this generation and iteration time. You can also find these informations in the object that is returned by this function, as well as the best string the b*bString* field, and strings within the relative tolerance limits in *StringsTol*.

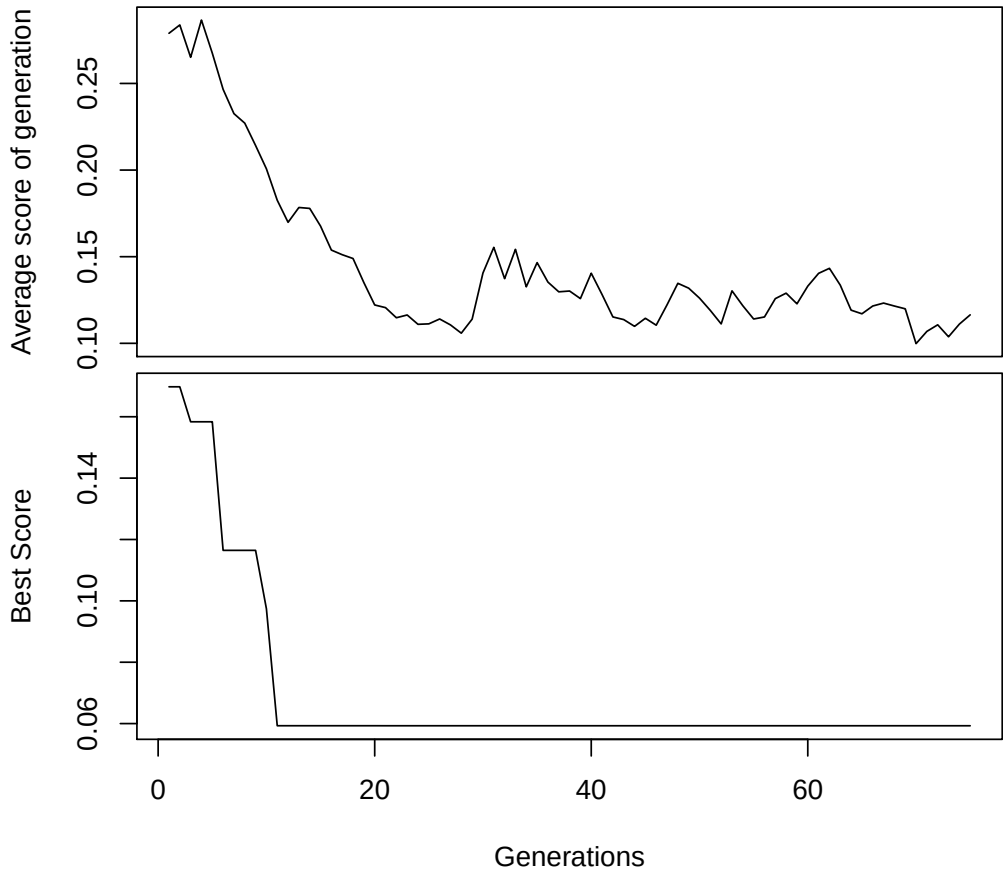
```
> ToyT1opt <- gaBinaryT1(CNOlist = CNOlistToy, Model = ToyNCNOcutCompExp,
+     SimList = ToyFields4Sim, indexList = indicesToyNCNOcutComp,
+     initBstring = initBstring, verbose = FALSE)
```

We will now produce plots of our analysis. First, we plot the results of simulating the data with our best model alongside the actual data set in a plot similar to that obtained above with *plotCNOlist*, except that the simulated data is overlaid in dashed blue lines, and the background of the plot reflects the absolute difference between simulated and experimental data (greener=closer to 0; redder=closer to 1; white=NA, either for data or simulation). Second, we will plot the evolution of the average score and best score during the evolution of the population of models, as a function of generations. This is useful to detect problems in the optimisation.

```
> cutAndPlotResultsT1(Model = ToyNCNOcutCompExp, bString = ToyT1opt$bString,
+     SimList = ToyFields4Sim, CNOlist = CNOlistToy, indexList = indicesToyNCNOcutComp,
+     plotPDF = FALSE)
```




```
> plotFit(OptRes = ToyT1opt)
```



If you want a pdf figure to be produced (advised, this will then be linked to your report), type:

```
> cutAndPlotResultsT1(Model = ToyNCNOcutCompExp, bString = ToyT1opt$bString,
+     SimList = ToyFields4Sim, CNOlist = CNOlistToy, indexList = indicesToyNCNOcutComp,
+     plotPDF = TRUE)
> pdf("evolFitToyT1.pdf")
> plotFit(OptRes = ToyT1opt)
> dev.off()
```

6 Writing your results

The next function, *writeScaffold*, allows you to write in a cytoscape sif file the scaffold network that was used for optimisation as well as two corresponding edge attribute files: one that tells you when the edge was called present in the optimised networks (ie 0=absent, 1=present), and one that tells you the weight of each edge as the fraction of models within the relative tolerance distance of the best model's score that actually included the edge. The function also writes the scaffold to a graphviz (`http://www.graphviz.org/Credits.php`) dot file, where the presence/absence is represented by the color of the edge (grey if absent, blue if present) and the weight is represented by the penwidth of the edges.

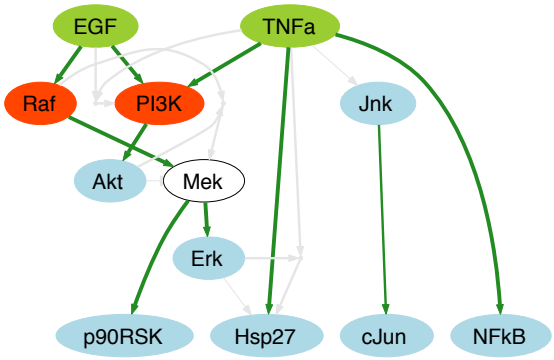


Figure 2: Scaffold network generated by the function writeScaffold for the Toy Model example.

We then also write the prior knowledge network, using the function *writeNetwork*, which again produces a sif file and corresponding attributes, and a dot file. The sif file has a corresponding edge attribute file that contains the present/absent information mapped back to the PKN, and the node attribute file contains information about the status of the node (compressed, non-observable/non-controllable, signal, inhibited, stimulated). The dot file encodes the edge information as above and the node information in the color of the node: signals are in blue, inhibited nodes are in red, stimulated nodes are in green, and compressed/cut nodes are in white with dashed contour. Examples of such files can be found on figures 2 and 3.

You can then write a report that contains relevant information about your analysis and links to the various plots that you created. This will be in the form of an html file called CellNOptReport.html, which will be stored along with all of your plots in a directory that you create and name (the name of that folder is a parameter in the function *writeReport*). Please not that the function *writeReport* will create the folder and move all of the files given by the list *namesFiles* into this directory. The files given by the arguments *dataPlot*, *evolFit1*, *optimResT1* will then be hyperlinked to your html report.

```
> writeScaffold(ModelComprExpanded = ToyNCNOcutCompExp,
+     optimResT1 = ToyT1opt, optimResT2 = NA, ModelOriginal = ToyModel,
+     CNOlist = CNOlistToy)
> writeNetwork(ModelOriginal = ToyModel, ModelComprExpanded = ToyNCNOcutCompExp,
+     optimResT1 = ToyT1opt, optimResT2 = NA, CNOlist = CNOlistToy)
> namesFilesToy <- list(dataPlot = "ToyModelGraph.pdf",
+     evolFit1 = "evolFitToyT1.pdf", evolFit2 = NA, SimResults1 = "ToyNCNOcutCompExpSimResultsT1.pdf",
+     SimResults2 = NA, Scaffold = "ToyNCNOcutCompExpScaffold.sif",
+     ScaffoldDot = "ModelModelComprExpandedScaffold.dot",
+     tscaffold = "ToyNCNOcutCompExpTimesScaffold.EA",
+     wscaffold = "ToyNCNOcutCompExpweightsScaffold.EA",
+     PKN = "ToyModelPKN.sif", PKNdot = "ToyModelPKN.dot",
+     wPKN = "ToyModelTimesPKN.EA", nPKN = "ToyModelnodesPKN.NA")
> writeReport(ModelOriginal = ToyModel, ModelOpt = ToyNCNOcutCompExp,
+     optimResT1 = ToyT1opt, optimResT2 = NA, CNOlist = CNOlistToy,
+     directory = "testToy", namesFiles = namesFilesToy,
+     namesData = list(CNOlist = "Toy", Model = "ToyModel"),
+     resE = resECNOlistToy)
```

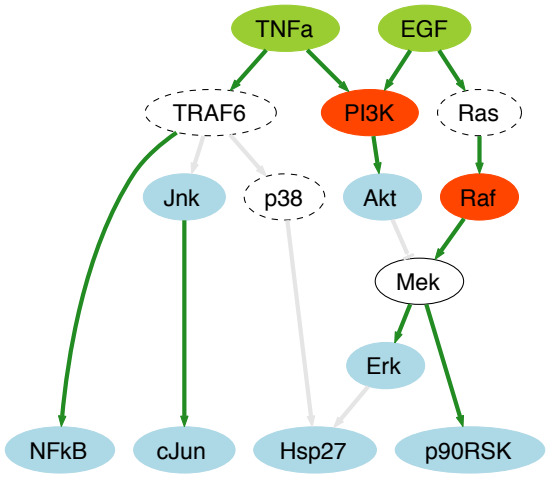


Figure 3: PKN generated by the function writeNetwork for the Toy Model example.

7 The one step version

If you do not want to bother with all of these steps, there is a function that allows you to do the whole analysis in one step. In order to do this, you must first load the model and data, as above (assuming that you have already loaded the library and have copied the relevant files in your working directory).

```
> library(CellNOptR)
> library(RBGL)
> dir.create("CNOR_analysis")
> setwd("CNOR_analysis")
> cpfile <- dir(system.file("ToyModel", package = "CellNOptR"),
+     full = TRUE)
> file.copy(from = cpfile, to = getwd(), overwrite = TRUE)
> dataToy <- readMIDAS(MIDASfile = "ToyDataMMB.csv")
> CNOlistToy <- makeCNOlist(dataset = dataToy, subfield = FALSE)
> ToyModel <- readSif(sifFile = "ToyPKNMMB.sif")
```

Then you have two possibilities, either you keep all optimisation parameters to their default values and just type

```
> CNORwrap(paramsList = NA, Name = "Toy", NamesData = list(CNOlist = "ToyData",
+     Model = "ToyModel"), Data = CNOlistToy, Model = ToyModel)
```

or you want to have some control over the parameters of the optimisation, in which case you create a parameters list with fields *Data* (the CNOlist containing your data), *Model* (your model object), *sizeFac* (default to 1e-04), *NAFac* (default to 1), *PopSize* (default to 50), *Pmutation* (default to 0.5), *MaxTime* (default to 60), *maxGens* (default to 500), *StallGenMax* (default to 100), *SelPress* (default to 1.2), *elitism* (default to 5), *RelTol* (default to 0.1), *verbose* (default to FALSE). You can then call the wrapper function with this set of parameters.

```
> pList <- list(Data = CNOlistToy, Model = ToyModel, sizeFac = 1e-04,
+     NAFac = 1, PopSize = 50, Pmutation = 0.5, MaxTime = 60,
```

```
+     maxGens = 500, StallGenMax = 100, SelPress = 1.2,
+     elitism = 5, RelTol = 0.1, verbose = TRUE)
> CNORwrap(paramsList = pList, Name = "Toy", NamesData = list(CNOlist = "ToyData",
+     Model = "ToyModel"), Data = NA, Model = NA)
```

Both of these versions will generate the graphs produced above on your graphics window, and as pdf's that will be hyperlinked to your html report, and you will be able to find all of these hyperlinked with your html report, all in a directory called by the *Name* parameter, in your working directory.

8 A real example

We will now apply the same procedure to a slightly larger and more realistic data set, which is a part of the network analysed in [4], which comprises 40 species and 58 interactions in the PKN. This network was also used for the signaling challenge in DREAM 4 (see `http://www.the-dream-project.org/`). The associated data was collected in hepatocellular carcinoma cell line HepG2 (see [1]).

First, you load and visualise your data.

```
> library(CellNOptR)
> dir.create("CNOR_analysis")
> setwd("CNOR_analysis")
> cpfile <- dir(system.file("DREAMModel", package = "CellNOptR"),
+     full = TRUE)
> file.copy(from = cpfile, to = getwd(), overwrite = TRUE)
> dreamData <- readMIDAS(MIDASfile = "LiverDataDREAMMIDAS.csv")
> CNOlistDREAM <- makeCNOlist(dataset = dreamData, subfield = FALSE)
> plotCNOlist(CNOlistDREAM)
> plotCNOlistPDF(CNOlist = CNOlistDREAM, fileName = "DREAMdataGraph.pdf")
```

Then, you load the model, check that the data matches the model, and get the indices of measured/inhibited/stimulated species.

```
> DreamModel <- readSif(sifFile = "LiverPKNDREAM.sif")
> checkSignals(CNOlistDREAM, DreamModel)
> indicesDream <- indexFinder(CNOlistDREAM, DreamModel,
+     verbose = TRUE)
```

Alternatively, you can also load the R object that contains the CNOlist and model for this analysis, in which case you would type the following instead of the above.

```
> library(CellNOptR)
> dir.create("CNOR_analysis")
> setwd("CNOR_analysis")
> data(CNOlistDREAM, package = "CellNOptR")
> data(DreamModel, package = "CellNOptR")
> plotCNOlistPDF(CNOlist = CNOlistDREAM, fileName = "DREAMdataGraph.pdf")
> checkSignals(CNOlistDREAM, DreamModel)
> indicesDream <- indexFinder(CNOlistDREAM, DreamModel,
+     verbose = TRUE)
```

You can now start the preprocessing of the model, i.e. find the non-observable/non-controllable species, cut them, recompute the indices, compress the model, recompute the indices, and expand the gates.

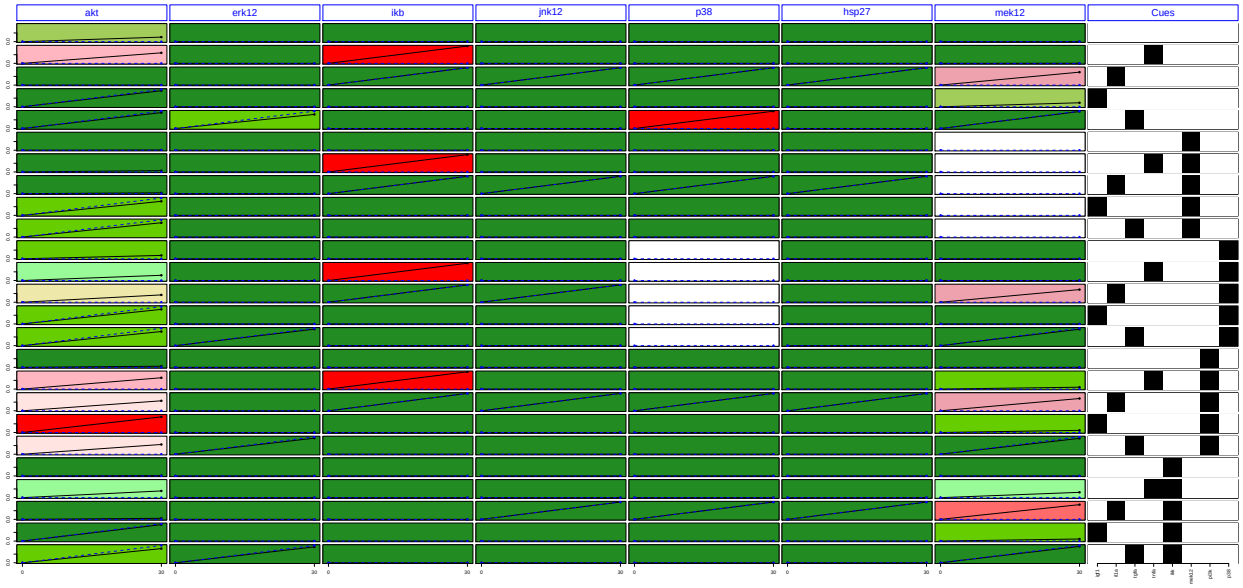


Figure 4: Plot of simulated and experimental data for the DREAM data and model presented above.

```
> DreamNCNOindices <- findNONC(DreamModel, indicesDream,
+     verbose = TRUE)
> DreamNCNOcut <- cutNONC(DreamModel, DreamNCNOindices)
> indicesDreamNCNOcut <- indexFinder(CNOlistDREAM, DreamNCNOcut)
> DreamNCNOcutComp <- compressModel(DreamNCNOcut, indicesDreamNCNOcut)
> indicesDreamNCNOcutComp <- indexFinder(CNOlistDREAM,
+     DreamNCNOcutComp)
> DreamNCNOcutCompExp <- expandGates(DreamNCNOcutComp)
```

Having obtained a pre-processed model, we now compute the residual error, prepare the model for simulation, and run the training function.

```
> resECNOdream <- ResidualError(CNOlistDREAM)
> DreamFields4Sim <- prep4Sim(DreamNCNOcutCompExp)
> initBstring <- rep(1, length(DreamNCNOcutCompExp$reacID))
> DreamT1opt <- gaBinaryT1(CNOlist = CNOlistDREAM, Model = DreamNCNOcutCompExp,
+     SimList = DreamFields4Sim, indexList = indicesDreamNCNOcutComp,
+     initBstring = initBstring, verbose = TRUE, MaxTime = 600)
```

We now plot the results of our training (see figure 4) as well as the information about the evolution of fit during optimisation.

```
> cutAndPlotResultsT1(Model = DreamNCNOcutCompExp, bString = DreamT1opt$bString,
+     SimList = DreamFields4Sim, CNOlist = CNOlistDREAM,
+     indexList = indicesDreamNCNOcutComp, plotPDF = TRUE)
> pdf("evolFitDreamT1.pdf")
> plotFit(OptRes = DreamT1opt)
> dev.off()
> plotFit(OptRes = DreamT1opt)
```

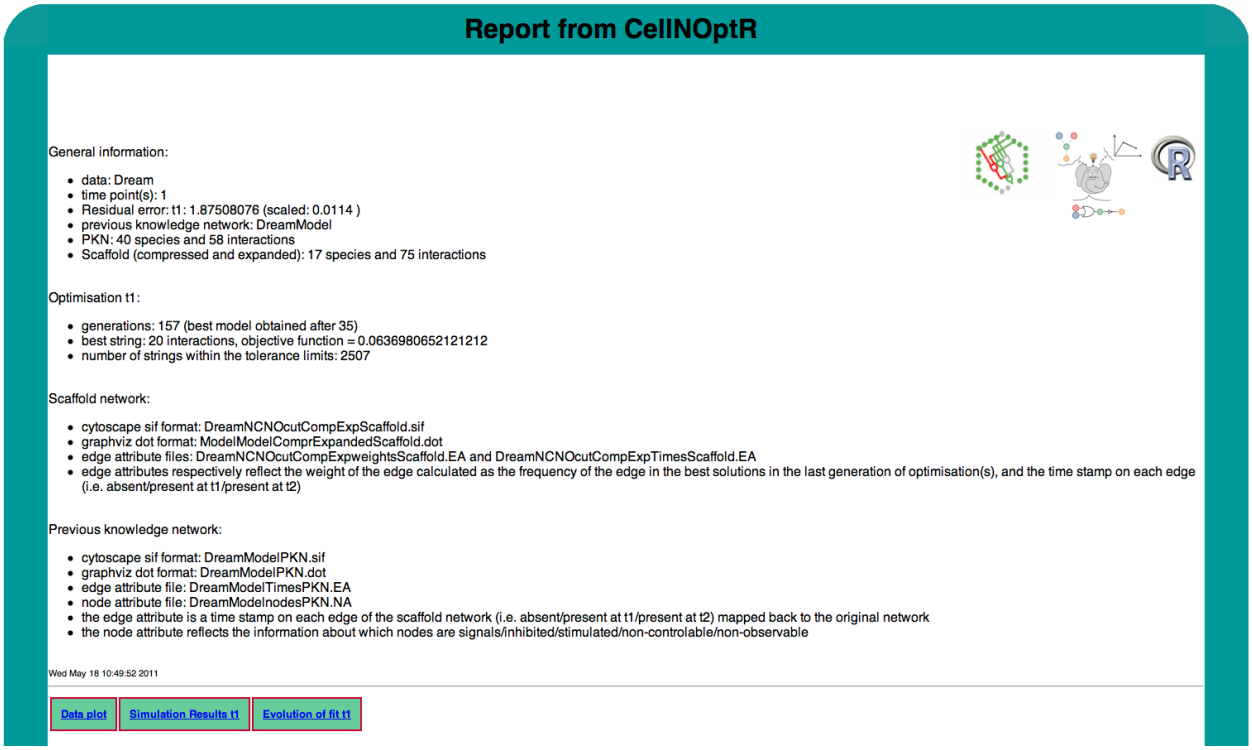


Figure 5: Report generated by CellNOptR for the DREAM data set and model training process.

Our analysis is now complete and we can write our results (scaffold network, prior knowledge network and html report). An example of the html report generated for this analysis is shown on figure 5. The scaffold network produced by *writeScaffold* is shown on figure 6 .

```
> writeScaffold(ModelComprExpanded = DreamNCNOcutCompExp,
+     optimResT1 = DreamT1opt, optimResT2 = NA, ModelOriginal = DreamModel,
+     CNOlist = CNOlistDREAM)
> writeNetwork(ModelOriginal = DreamModel, ModelComprExpanded = DreamNCNOcutCompExp,
+     optimResT1 = DreamT1opt, optimResT2 = NA, CNOlist = CNOlistDREAM)
> namesFilesDream <- list(dataPlot = "DREAMdataGraph.pdf",
+     evolFit1 = "evolFitDreamT1.pdf", evolFit2 = NA, SimResults1 = "DreamNCNOcutCompExpSimResultsT1.pdf
+     SimResults2 = NA, Scaffold = "DreamNCNOcutCompExpScaffold.sif",
+     ScaffoldDot = "ModelModelComprExpandedScaffold.dot",
+     tscaffold = "DreamNCNOcutCompExpTimesScaffold.EA",
+     wscaffold = "DreamNCNOcutCompExpweightsScaffold.EA",
+     PKN = "DreamModelPKN.sif", PKNdot = "DreamModelPKN.dot",
+     wPKN = "DreamModelTimesPKN.EA", nPKN = "DreamModelnodesPKN.NA")
> writeReport(ModelOriginal = DreamModel, ModelOpt = DreamNCNOcutCompExp,
+     optimResT1 = DreamT1opt, optimResT2 = NA, CNOlist = CNOlistDREAM,
+     directory = "testDREAM", namesFiles = namesFilesDream,
+     namesData = list(CNOlist = "Dream", Model = "DreamModel"),
+     resE = resECNOdream)
```

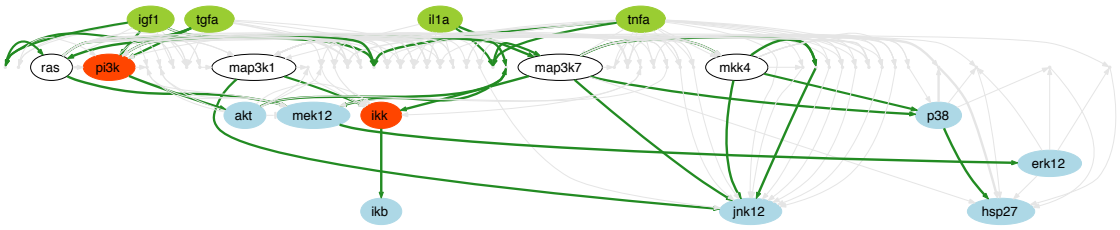


Figure 6: Scaffold network for the DREAM model, as produced by writeScaffold.

References

[1] Alexopoulos, L.G., Saez-Rodriguez, J., Cosgrove, B.D., Lauffenburger, D.A., Sorger, P.K.: Networks inferred from biochemical data reveal profound differences in toll-like receptor and inflammatory signaling between normal and transformed hepatocytes. Molecular & Cellular Proteomics: MCP **9**(9), 1849–1865 (2010).

[2] M.K. Morris, I. Melas, J. Saez-Rodriguez. Construction of cell type-specific logic models of signalling networks using CellNetOptimizer. *Methods in Molecular Biology: Computational Toxicology*, Ed. B. Reisfeld and A. Mayeno, Humana Press.

[3] J. Saez-Rodriguez, A. Goldsipe, J. Muhlich, L.G. Alexopoulos, B. Millard, D.A. Lauffenburger and P.K. Sorger. Flexible informatics for linking experimental data to mathematical models via *DataRail*. *Bioinformatics*, 24:6, 2008.

[4] J. Saez-Rodriguez, L. Alexopoulos, J. Epperlein, R. Samaga, D. Lauffenburger, S. Klamt and P.K. Sorger. Discrete logic modelling as a means to link protein signalling networks with functional analysis of mammalian signal transduction. *Molecular Systems Biology*, 5:331, 2009.

[5] P. Shannon, A. Markiel, O. Ozier, N.S.. Baliga, J.T. Wang, D. Ramage, N. Amin, B. Schiwkowski and T. Ideker. Cytoscape: a software for integrated models of biomolecular interaction networks. *Genome Research*, 13(11):2498-504, 2003.