

Ways to improve the efficiency of pharmacometric model implemented in R using deSolve – an example with a PK model for ADC



Yang Zhang (1), Dirk Garmann (1)

(1) Bayer AG, Germany

Introduction

Implementation of a pharmacometric (PMX) model in R is a prerequisite for accomplishing different tasks – from a demonstration plot to a full scale simulation; from parameter estimation to evaluating a trial design.

Numerous attempts to reduce the burden of implementing pharmacometric models in R have been made which resulted in new packages on CRAN as well as github. Yet when drilling down to the implementation of pharmacometric models by numerically solving the ordinary differential equations, LSODA still remains the dominating algorithm and is available in one of the most widely used packages for solving ODEs, deSolve.

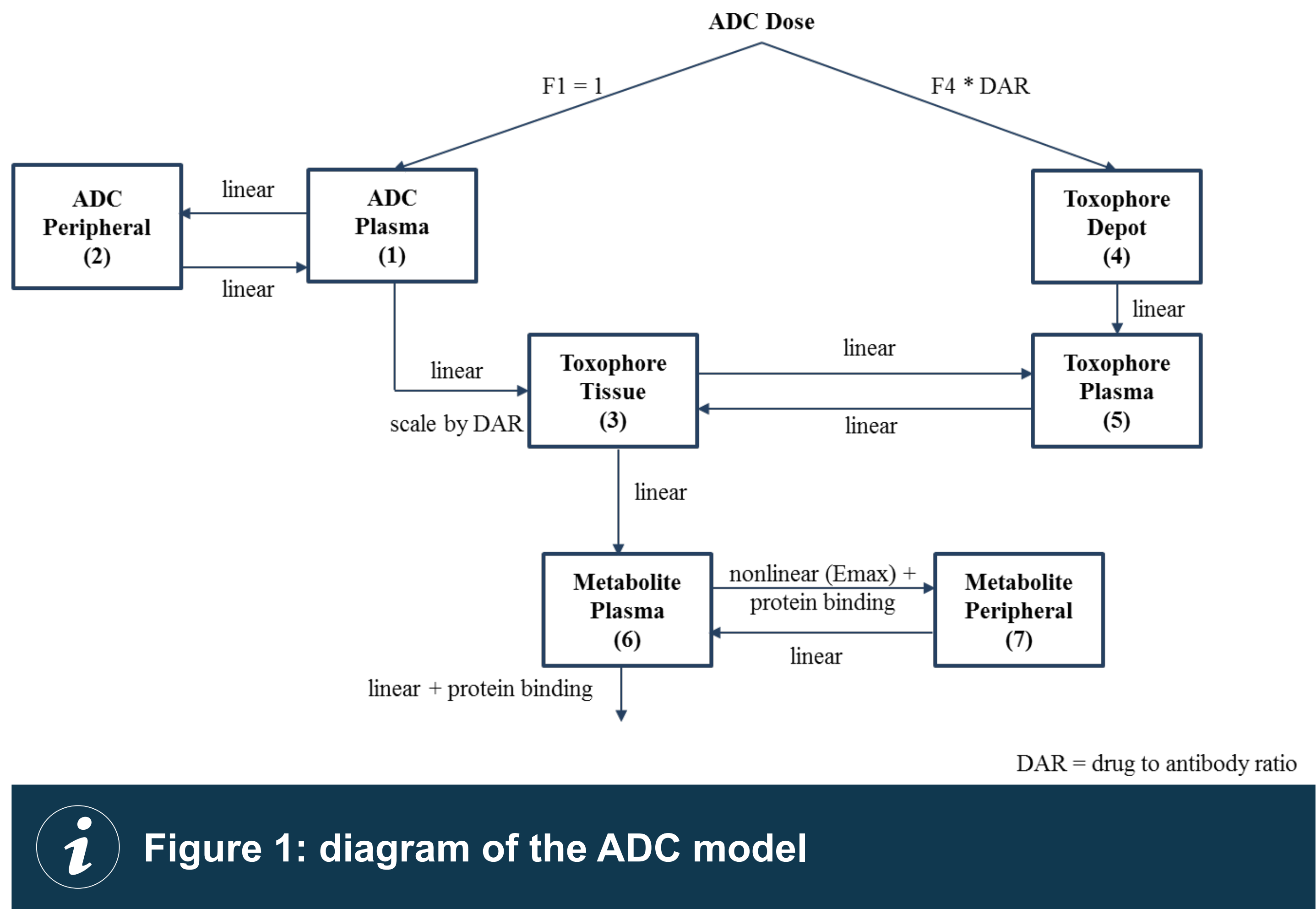
This work is intended to implement a relatively complex compartmental pharmacometric model in R utilizing the default method (LSODA) of deSolve and compare the performance in terms of time of execution among different implementations.

Objectives

To check different implementations of a PMX model in R and compare the time needed to execute

Methods

A PK model for an antibody drug conjugate (ADC) was chosen as an exemplary model due to its complexity as well as a protein binding component. The model consists of 7 compartments with 2 compartments for the ADC, 3 for the toxophore and 2 for the active agent metabolized from the toxophore. The structure of the model can be found in Fig. 1



The model was first implemented purely in R. Then the computation of the derivatives for each compartment (i.e., the right hand sides of the ODEs) was conducted using compiled code written in C++ (via the Rcpp package[1]), C or Fortran (using the interfaces provided by R)[2] with or without explicitly calculation of the Jacobian matrix. In total, eight implementations were compared in terms of total execution time for 100 repetitions.

The execution environment was a server with CPU base frequency of 2.5GHz running Red Hat Enterprise Linux Server release 6.9 (Santiago). R (3.2.5) with deSolve (version 1.13) and Rcpp (0.12.4) was deployed and the compiler used was GCC version 4.4.7.

Results

For the implementation without explicit computing the Jacobian matrix, the pure R version took more than 1000 seconds to complete 100 executions while using Rcpp could reduce the time to 183 seconds. Using C and Fortran further reduced the time to less than 1.5 seconds. On the other hand, adding a function to compute the Jacobian matrix increased the efficiency of the ODE solver drastically for all except the C version. An implementation with the calculation of both the r.h.s of the ODEs and the

```
# r.h.s. of the ODEs
dA1 <- -k12*A1 + k21*A2 - k13*A1 + input[1]
dA2 <- k12*A1 - k21*A2 + input[2]
dA3 <- k13*dar*A1 - k36*A3 - k35*A3 + k53*A5 + input[3]
dA4 <- -ka*A4 + input[4]
dA5 <- ka*A4 + k35*A3 - k53*A5 + input[5]
dA6 <- k36*A3 - clme*frcc - vmax*v6*frcc/(km + frcc) + k76*A7 + input[6]
dA7 <- vmax*v6*frcc/(km + frcc) - k76*A7 + input[7]
res <- c(dA1, dA2, dA3, dA4, dA5, dA6, dA7)
# ...

// r.h.s. of the ODEs
double dA1 = -k12*A1 + k21*A2 - k13*A1 + r1;
double dA2 = k12*A1 - k21*A2 + r2;
double dA3 = k13*dar*A1 - k36*A3 - k35*A3 + k53*A5 + r3;
double dA4 = -ka*A4 + r4;
double dA5 = ka*A4 + k35*A3 - k53*A5 + r5;
double dA6 = k36*A3 - clme*frcc - vmax*v6*frcc/(km + frcc) + k76*A7 + r6;
double dA7 = vmax*v6*frcc/(km + frcc) - k76*A7 + r7;
NumericVector
  drv = NumericVector::create(
    _["dA1"] = dA1, _["dA2"] = dA2, _["dA3"] = dA3,
    _["dA4"] = dA4, _["dA5"] = dA5, _["dA6"] = dA6,
    _["dA7"] = dA7
  );

//...
/* r.h.s. of the ODEs */
ydot[0] = -k12*y[0] + k21*y[1] - k13*y[0] + r1;
ydot[1] = k12*y[0] - k21*y[1] + r2;
ydot[2] = k13*dar*y[0] - k36*y[2] - k35*y[2] + k53*y[4] + r3;
ydot[3] = -ka*y[3] + r4;
ydot[4] = ka*y[3] + k35*y[2] - k53*y[4] + r5;
ydot[5] = k36*y[2] - clme*frcc - vmax*v6*frcc/(km + frcc) + k76*y[6] + r6;
ydot[6] = vmax*v6*frcc/(km + frcc) - k76*y[6] + r7;
/*...*/

c r.h.s. of the ODEs
ydot(1) = -k12*y(1) + k21*y(2) - k13*y(1) + r1
ydot(2) = k12*y(1) - k21*y(2) + r2
ydot(3) = k13*dar*y(1) - k36*y(3) - k35*y(3) + k53*y(5) + r3
ydot(4) = -ka*y(4) + r4
ydot(5) = ka*y(4) + k35*y(3) - k53*y(5) + r5
ydot(6) = k36*y(3) - clme*frcc - vmax*v6*frcc/(km + frcc) + k76*y(7) + r6
&
ydot(7) = vmax*v6*frcc/(km + frcc) - k76*y(7) + r7

c ...
```

Jacobian in FORTRAN tops the list with more than 3000 times increase in speed compared to the slowest one (see Table 1 for the execution times).

Implementation		Time Reported for 100 executions [s]		
		User	System	Elapse
Without User-defined Jacobian	R	1009.540	8.230	1016.622
	Rcpp	181.523	1.704	183.015
	C	1.364	0.016	1.379
	Fortran	1.385	0.018	1.402
With User-defined Jacobian	R	28.279	0.267	28.511
	Rcpp	6.175	0.064	6.232
	C	9.744	0.078	9.810
	Fortran	0.262	0.009	0.271

Table 1: Execution time as reported by system.time()

Conclusions

When implementing a pharmacometric model in R several approaches are available to improve the performance. As shown in this example the efficiency gain can be quite large. Depending the experience of the user, explicit calculation of Jacobian matrix and / or conducting parts of the computation using compiled codes (C++, C or Fortran) may be considered to accelerate the process.

References

- <https://cran.r-project.org/web/packages/Rcpp/index.html>, accessed Feb 20, 2019
- <https://cran.r-project.org/web/packages/deSolve/vignettes/compiledCode.pdf>, accessed Feb 20, 2019