REFERENCE

1. Vietnam. Nuclear Research Institute. Regulations for Operation of the Dalat Nuclear Research Reactor. Dalat: Nuclear Research Institute, 1990.
2. Rozon, D.; and Rouben, B. Reactor Fuel Management and Core Analysis. Training Program Reference Text. Bangkok: Chulalongkorn University, 1997.
3. Doval, A.S. Notes on Thermohydraulics. Material of Thermohydraulic Course for the Training MPR First Staff. Argentina: INVAP, 1997.
4. Duderstadt, J.J.; and Hamilton, L.J. Nuclear Reactor Analysis. New York: John Wiley & Sons Inc., 1976.
5. Garland, W. Thermalhydraulic Analysis. Training Program Reference Text. Bangkok: Chulalongkorn University, 1998.
6. El-Walkil, M.M. Nuclear Heat Transport. New York: International Text Book Company, 1971.
7. Koklas, J. Neutronic Analysis of Reactors. Training Program Reference Text. Bangkok: Chulalongkorn University, 1998.
8. Croft, A.; Davidson, R.; and Hargreaves, M. Engineering mathematics. Workingham: Adison-Wesley Publishing Co., 1992.
9. Holman, J.P. Heat Transfer, SI metric edition. Singapore: McGraw-Hill Book Co., 1989.
10. Arpaci, V.S. Conduction Heat Transfer. Massachusetts: Adison-Wesley Publishing Co., 1966.
11. Matos, J.E.; and Freese, K.E. Safety Analyses for HEU and LEU Equilibrium Cores and HEU-LEU Transition Core for the IAEA Generic 10 MW Reactor. Research Reactor Core Conversion Guide Book. Vienne: IAEA-TECDOC 643, April 1992.
12. Kakac, S.; Shah, R.K.; and Aung, W. Handbook of Single-Phase Convective Heat Transfer. New York: John Wiley & Sons Inc., 1987.
13. Bejan, A. Convection Heat Transfer. New York: John Wiley & Sons Inc., 1984.

APPENDIX A

DESCRIPTION OF DALAT NUCLEAR RESEARCH REACTOR

The DNRR is a 500 kW pool type research reactor with natural convection cooling. Light water is used as the moderator and also as the coolant. The fuel assembly consists of three concentric tubes, the two inner ones are circular shaped, and the outer one is hexagonal. The fuel material is U-Al alloy with 36 wt-% enrichment. The amount of U-235 per assembly is about 40 g. The fuel cladding material is pure aluminium. Fuel meat thickness including cladding is 2.5 mm. Presently, there are 104 fuel assemblies, 7 control rods, some beryllium blocks and irradiation channels in the reactor core. To enforce the buoyancy, a chimney of 2 m in height and 0.5 m in diameter is installed above the core. (see figs. A1, A2)

Table A.1. Summary of DNRR technical data

Parameter	Value
Maximum power, kW	500
Fuel assemblies and water channels:	See fig. A3
Number of fuel assemblies	104
Fuel (U-Al alloy, 36 wt-% enrich.) thickness, mm	0.7
Cladding (Al) thickness, mm	0.9
Average water channels thickness, mm	3
Active fuel height, m	0.6
Total fuel assembly height, m	0.8
Chimney:	
Diameter, m	0.5
Height, m	2
Pool water depth (from free surface to core bottom), m	5.8
Radial peaking factor	1.78
Equivalent Channel:	
Wetted perimeter, m	0.38
Flow area, m^2	$5.85\ 10^{-4}$
Hydraulic diameter, m	0.00616
Heated surface, m^2	0.228
Fuel volume, m^3	$3.993\ 10^{-5}$
Fuel (including cladding) cross section, m^2	$0.475\ 10^{-6}$

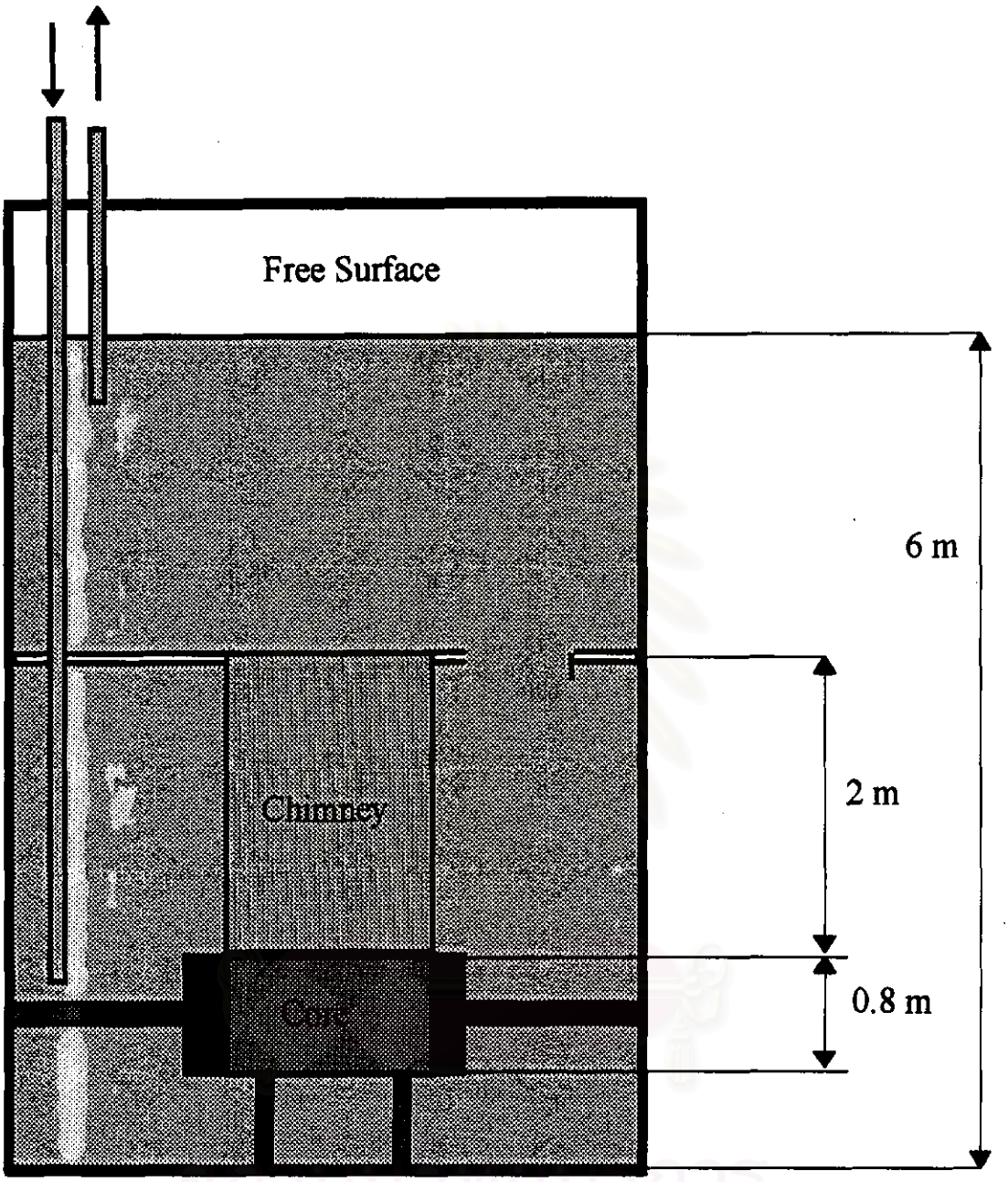

Fig. A1- DNRR reactor tank

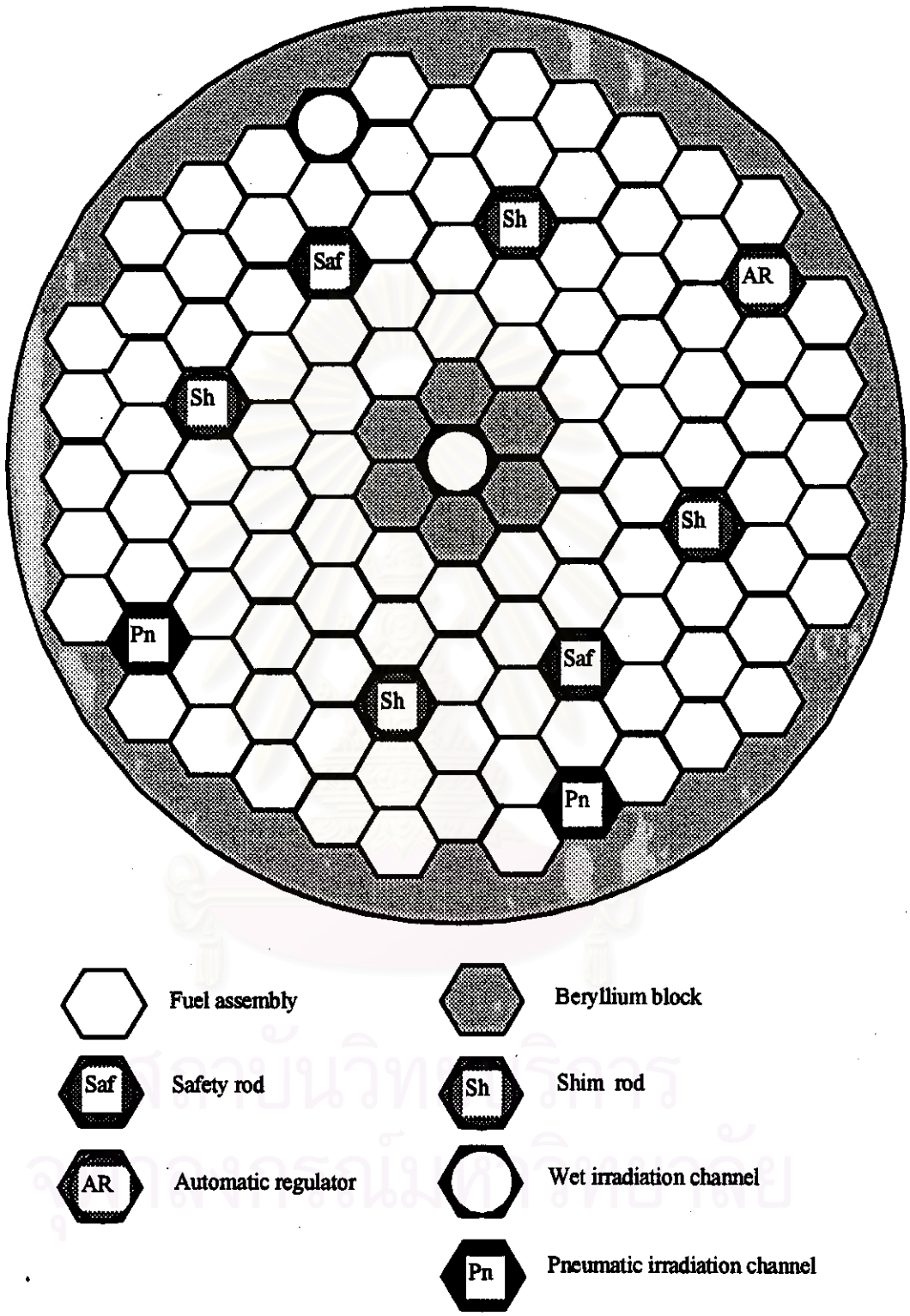

Fig. A2- Core cross section

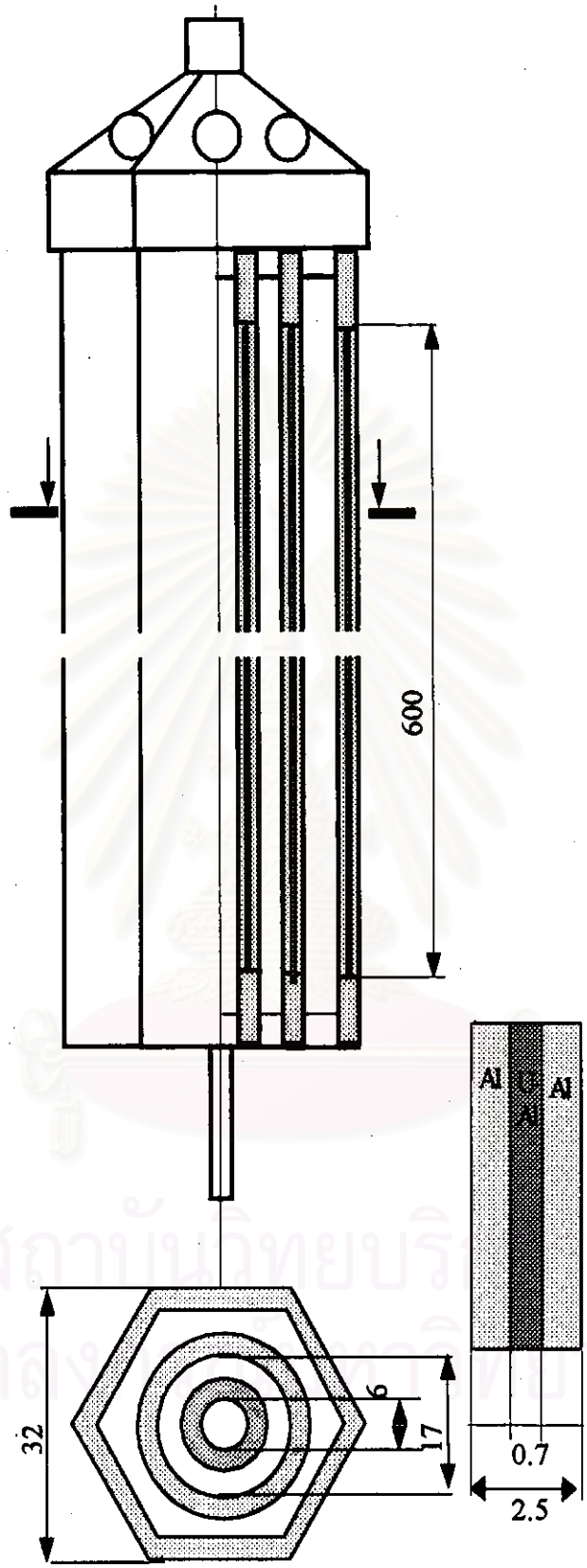

Fig.A3. Fuel Assembly

APPENDIX B

CONSTANTS AND PROPERTIES USED IN DNRR SIMULATION CODE

Table B.1. Neutronic parameters

Parameter	Symbol	Value
Thermal neutron flux at 100% FP (average)	Φ_{FP}	$4x10^{12}$ n/cm^2.s
Prompt neutron life time	ℓ	5.10^{-5} s
Effective delayed neutron fraction (1$)	β	0.0081
Number of delayed neutron groups	N	6

Table B.2. Delayed neutron group constants for U-235 [4]

Group	Decay constant, λ_i, s^{-1}	Fraction, γ_i
1	0.0127	0.038
2	0.0317	0.213
3	0.1155	0.188
4	0.3108	0.407
5	1.3975	0.128
6	3.8723	0.026
Total		1.

Note: $\beta_i = \gamma_i\beta$

Table B.3. Iodine-135 and Xenon-135 constants

Parameter		Symbol	Value
Fission yield:	I-135	γ_I	6.386 %
	Xe-135	γ_{Xe}	0.228 %
Decay constant:	I-135	λ_I	$2.9x10^{-5}$ sec
	Xe-135	λ_{Xe}	$2.1x10^{-5}$ sec
Neutron absorption cross-section of Xe-135		σ_{Xe}	$2.7x10^{6}$ barn
Equilibrium Xe-135 poisoning at full power		ρ_{Xe}^{max}	-1.6 $

Table B.4. Pseudo-fission product group constants

Group	Decay constant, λ_i, s^{-1}	Fission yield, γ_i
1	0.09	0.0144
2	0.01	0.0141
3	0.0004	0.0315
Total		0.0600

Axial neutron flux shape (by Le Vinh Vinh, Sep. 1998)

$$\varphi(z) = 2.0276\times10^{-7}z^4 - 1.7156\times10^{-5}z^3 - 5.3827\times10^{-4}z^2 + 0.0505z + 0.24983$$

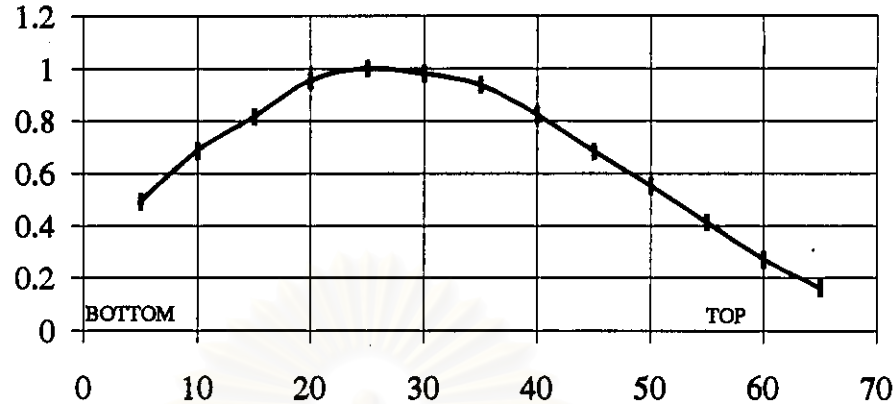

Fig.B.1. Axial neutron flux shape

Temperature Reactivity Coefficients: (by Huynh Ton Nghiem et al., Nov. 1998)

For inlet coolant temperature T_1 (within operational temperature range):

$$\bar{\alpha}_{T_1} = -0.01172 \pm 0.00242 \text{ \$/°C.}$$

For power (from 10 to 90%):

$$\bar{\alpha}_p = 2.5742\times10^{-5}p - 5.9736\times10^{-3} - 4.1113\times10^{-2}p^{-1}, \text{ \$/\%FP.}$$

Table B.5. Experimentally determined water and fuel coefficients

p %	T_1 °C	T_2 °C	T_w °C	T_f °C	$\bar{\alpha}_w$ \$/°C	$\bar{\alpha}_f$ \$/°C
10	21.39	28.34	24.87	28.64	0.0036	-0.015
20	21.52	32.96	27.24	33.66	0.0012	-0.013
30	21.71	34.42	28.07	36.18	0.0030	-0.015
40	21.90	35.61	28.76	39.98	0.0024	-0.014
50	22.13	37.73	29.93	43.41	0.0019	-0.014
60	22.36	40.34	31.35	46.13	0.0019	-0.014
70	22.70	42.10	32.40	48.97	0.0015	-0.013
80	23.01	43.78	33.40	51.72	0.0010	-0.013
90	23.43	45.55	34.49	55.41	-2E-04	-0.011

From the discussion of temperature reactivity coefficients in Chapter 4, section 4.3, the coefficients $\bar{\alpha}_f$ and $\bar{\alpha}_w$ can be derived from the experimentally determined values $\bar{\alpha}_{T_1}$ and $\bar{\alpha}_p$. Specifically, we find that:

$$\bar{\alpha}_f = 3.3970\times10^{-6}T_f^2 - 1.8497\times10^{-4}T_f - 1.2057\times10^{-2}, \text{ \$/°C,}$$

and $$\bar{\alpha}_w = -1.7535\times10^{-5}T_w^2 + 7.5595\times10^{-4}T_w - 4.9236\times10^{-3}, \text{ \$/°C.}$$

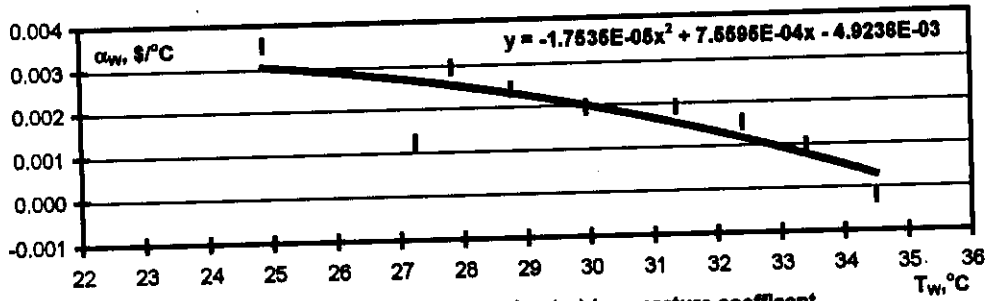

Fig.B.2. Water (coolant/moderator) temperature coefficent

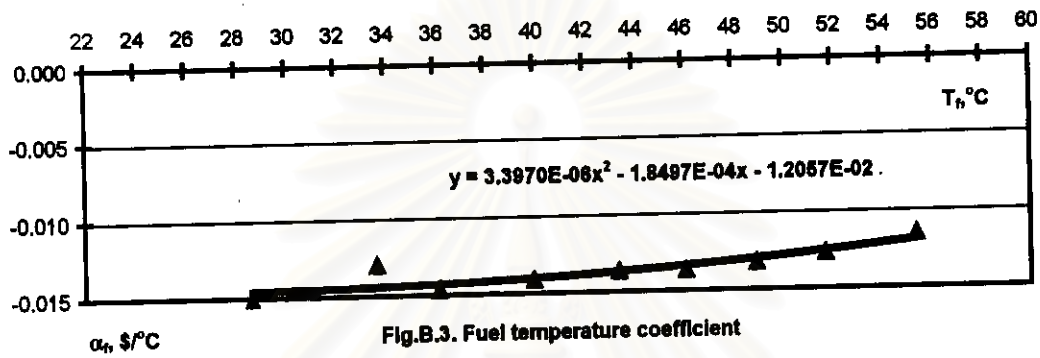

Fig.B.3. Fuel temperature coefficient

Table B.6. Thermal properties of fuel and cladding materials [2]

Material	Thermal Conductivity, W/mK	Specific Heat, J/cm^3K
UAl_x-Al Fuel	158	1.985 + 0.0010T
6061 Aluminium Cladding	180	2.069 + 0.0012T

Table B.7. Experimentally determined flow and heat transfer coefficient

p, %	10	20	30	40	50	60	70	80	90
T_1, °C	21.3	21.5	21.7	21.9	22.1	22.4	22.7	23	23.4
T_2, °C	28	33	34.4	35.6	37.7	40.3	42.1	43.8	45.6
T_f, °C	28.5	33.7	36.2	40	43.4	46.1	49	51.7	55.4
W_{core}, kg/s	0.56	1.12	1.67	2.22	2.76	3.29	3.80	4.30	4.77
h, W/m^2K	554	656	756	752	782	856	890	920	907

APPENDIX C

DRSIM CODE DESCRIPTION

DRSIM, the coupled thermal-hydraulics and point kinetics code to simulate the DNRR operation, is written in the C language for PCs in the DOS environment. The values of most geometry parameters and nuclear constants are assigned directly in the code. Only some control parameters are given as input.

1. Input

Control parameters are entered in an input file, optionally DRSIM.INP, as follows:

Line 1:	**t_start t_trans t_op [comment]**
t_start	start time of the transient [sec.]
t_trans	transient time [sec.]
t_op	operation time [sec.] before the transient if entered positive. It is used to calculate decay heat and xenon poisoning. If t_op is given a negative value (t_op<0), all fission products are assumed to be at the equilibrium build-up.
Line 2:	**rho tol_P [comment]**
rho	initial reactivity of the reactor if entered zero or negative. If entered zero or negative, it indicates the reactivity change mode at the initially critical (rho=0) or subcritical (rho<0) reactor. If rho is entered positive, the power change mode is considered.
tol_P	pressure tolerance [MPa] used to calculate time step
Line 3-6:	**x.t1 x.y1 x.t2 x.y2 x.mode [comment]**
x	power, reactivity, pump flow or inlet coolant temperature.
t1	time [sec] before which value of x-parameter is assigned to y1.
y1	initial value of x-parameter.
t2	either time constant for exponential change of x-parameter or time interval for linear change (ramp). t2=0 for step change.
y2	final value of x-parameter.
mode	0 for linear change, 1 for exponential change of x-parameter.
Line 7:	**p_scram t_delay t_scram worth [comment]**
p_scram	scram setting for power, for example, p_scram=1.1 (110%)
t_delay	time [sec] for the protection system to response to the scram signal.
t_scram	time interval [sec] within which all control rods are fully dropped into the core.
worth	total reactivity worth of all control rods.
Line 8:	**n_t [comment]**
n_t	number of time intervals each of which has different time step control. At least n_t =1.

Line 9 to 8+n_t: **t[] dt_max[] dt_min[] dt_print[] [comment]**

t[0..n_t-1] time [sec] before which time step size is ranged between dt_max and dt_min.

dt_max[] maximum time step [sec]

dt_min[] minimum time step [sec]

dt_print[] print-out time step [sec]

[commnet] any string without a space to comment the line of entry.

Example: Drsim.inp

```
0     600     -1                    Line1>>t_start>>t_trans>>t_op
1     1.e-4                         Line2>>rho>>tol_P
0     0.5     30     1.     1       Line3>>pi.t1>>y1>>t2>>y2>>mode
0     0.      0      0      0       Line4>>ri.t1>>y1>>t2>>y2>>mode
0     50.     60     0      0       Line5>>Gi.t1>>y1>>t2>>y2>>mode
0     22.     0      0      0       Line6>>Ti.t1>>y1>>t2>>y2>>mode
1.1   0.025   1      -10             Line7>>p_scram>>t_delay>>t_scram>>worth
1                                   Line8>>n_t
1     1.e-5   1.e-3  1               Line9>>t[0]>>dt_min[0]>>dt_max[0]>>dt_print[0]
```

Explanation:

Line 1: The transient begins at t_start=0 and will last for t_trans=600s As t_op=-1<0, all fission products are considered at the equilibrium for the constant power pi.y1=0.5 (50%).

Line 2: As rho=1>0, the power change mode will be considered. Pressure tolerance tol_P = 0.0001MPa.

Line 3: Up to time pi.t1=0, power initially is pi.y1=0.5 (50%). The power will be changed by exponential (pi.mode=1) with the time constant (period) pi.t2=30s until it reaches pi.y2=1 (100%).

Line 4: For reactivity change. But this line is optional in the power change mode.

Line 5: Pump flow initially is Gi.y1=50m^3/h. Gi.mode=0 means the pump flow will be changed linearly from Gi.y1 to Gi.y2 (0) within Gi.t2 (60s) starting at Gi.t1=0.

Line 6: Inlet coolant temperature initially is Tin.y1=22°C. The step change of T_{in} (Tin.t2=0) to Tin.y2=0. It does not mean T_{in} now becomes zero but it will be the same as water temperature of the upper pool.

Line 7: Scam power setting is p_scram=1.1 (110%). Whenever scram signal is created, it will take t_delay=0.025s for protection system to respond to the signal before all control rods with worth=-10$ begins to drop into the core within t_scram=1s.

Line 8: There will be n_t=1 time interval for the entire transient.

Line 9: Initially, the minimum time step is dt_min[0]=10^{-5}s and the maximum one is dt_max[0]=10^{-3}s. Time step for print-out is dt_print[0]=1s. As there only 1 time interval, the initial time t[0] can be given any value.

Optionally, the code can restore neutronic and thermal-hydraulic parameters from a file if they have been saved at any time during a transient. That allows the code to continue the transient from the saved time without restart from the beginning.

2. Output

Any parameter at any time can be printed-out to the screen and to an ASCII file, optionally DRSIM.OUT. The most important parameters are power, period, core flow, core inlet and outlet coolant temperatures, maximum fuel surface temperature, margins to ONB and DNB. The code can save all current parameters to a file for later use.

The Excel program is convenient for processing the results and to create charts of the transient parameters.

3. Listing

The code is composed of 6 modules:

1) DRSIM	controls the running process: control parameters, time step, print-out, etc.
2) KINETICS	solves the point kinetics equations to obtain the neutron power given net reactivity.
3) XEPOISON	solves the iodine-135 and xenon-135 build-up equations to obtain the xenon reactivity feedback given the neutron power.
4) DEC_HEAT	solves the pseudo-fission product build-up equations to obtain decay heat fraction given the neutron power.
5) THEHYD	forms node-link circuit and solves thermal-hydraulic equations to obtain various thermal-hydraulic parameters of all nodes and links as well as the temperature feedback.
6) WATER	contains the curve fits of water thermodynamic properties.

Listings of 4 modules: KINETICS, XEPOISON, DEC_HEAT and THEHYD are given in the following:

```
/*************************************************************************
                        Point Kinetics Module
*************************************************************************/
#include <math.h>
#include <fstream.h>
//constants:
const int n=6;                  //number of delayed groups
const double beta=0.0081;       //effective delayed fraction
const double   l=5.e-5;         //neutron generation life
const double lambda[]={.0127,.0317,.1155,.3108,1.3975,3.8723};     //decay constants
const double gamma[]={ .038, .213, .188, .407,  .128,  .026};      //group fractions
//variables:
static double c[n];             //delayed neutron precursor at t
static double s;                //source at t
//========== Initialization ==========
double init_ki(double p, double rho)                //Steady-state conditions
{
        rho=rho>0?0:rho*beta;
double Lambda; Lambda=l*(1.-rho);//Lamda=l/k
for(register i=0;i<=n-1;i++)
        c[i]=beta*gamma[i]/lambda[i]/Lambda*p;
        s=rho<0.?-rho/Lambda*p:0.;
return p;
}

double init_ki(double p, double ci[], double si)        //Saved conditions
{
for(register i=0;i<=n-1;i++) c[i]=ci[i];
        s=si;
return p;
}
//========== Save to file ==========
int save_ki(ofstream& fout)
{
if(!fout) return 0;
for(register i=0;i<=n-1;i++) fout<<c[i]<<" "; fout<<s<<endl;
return -1;
}
//========== Solution of Point Kinetic ==========
double kinetics(double p, double rho, double dt)
{
        rho=rho*beta;
double sum1=0;
double sum2=0;
double Lambda;  Lambda = l*(1.-rho);
register i;
for(i=0;i<=n-1;i++){
        sum1+=lambda[i]*c[i]/(1+lambda[i]*dt);
        sum2+=gamma[i]/(1+lambda[i]*dt);
        }
        p=(p+dt*(sum1+s))/(1-dt/Lambda*(rho-beta*sum2));
for(i=0;i<=n-1;i++)
        c[i]=(c[i]+dt*beta*gamma[i]/Lambda*p)/(1+lambda[i]*dt);
return p;
}
```

```
/*************************************************************************
                                Xenon Poisoning Module
*************************************************************************/
#include <math.h>
#include <fstream.h>
//constants
const double gammai=0.06386;    //I yield
const double gammax=0.00228;    //Xe yeild
const double lambdai=0.000029;  //I decay constant
const double lambdax=0.000021;  //Xe decay constant
const double sigmax=2.7e-18;    //Xe xsection
const double ff=4.0e12;         //average full flux
const double rhoxmax=-1.6;      //equilibrium poison at full power
//variables
static double nx, ni;           //Xe and I numbers
static double nxmax;            //Xe equilibrium number at full power
static double rhox;
//========== Initialization ==========
double init_xe(double p, double t)                      //At constant power
{
        nxmax=(gammai+gammax)/(sigmax*ff+lambdax);
if(t==0){       //free of poison
        ni=nx=0;
        }
else if(t<0){   //equilibrium
        ni=gammai/lambdai*p;
        nx=(gammai+gammax)*p/(sigmax*p*ff+lambdax);
        }
else{           //after time t
double lambda=sigmax*p*ff+lambdax;
        ni=gammai/lambdai*p*(1.-exp(-lambdai*t));
        nx=(gammai+gammax)*p/lambda*(1.-exp(-lambda*t))
                -gammai*p/(lambda-lambdai)*(exp(-lambdai*t)-exp(-lambda*t));
        }
return(rhox=nx/nxmax*rhoxmax);
}

double init_xe(double nii, double nxi, double nxmaxi)          //Saved conditions
{
        ni=nii;  nx=nxi; nxmax=nxmaxi;
return(rhox=nx/nxmax*rhoxmax);
}
//========== Save to file ==========
int save_xe(ofstream& fout)
{
if(!fout) return 0;
fout<<ni<<" "<<nx<<" "<<nxmax<<endl;
return -1;
}
//========== Solution of Xe and I buid-up equations ==========
double rho_xe(double p, double dt)
{
if(dt<=0) return rhox;
        ni=(ni+gammai*p*dt)/(1.+lambdai*dt);
        nx=(nx+(gammai*p+lambdai*ni)*dt)/(1.+(sigmax*p*ff+lambdax)*dt);
return(rhox=nx/nxmax*rhoxmax);
}
```

```
/*************************************************************************
                        Decay Heat Module
*************************************************************************/
#include <math.h>
#include <fstream.h>
//constants
const int n_fp=3;                                   //number of fission product groups
const double gamm_fp[]={0.0144,0.0141,0.0315};      //fission product fractions
const double lamb_fp[]={0.09,0.01,0.0004};          //fission product decay constants
//variables
static double w[n_fp];                              //pseudo-fission product concentration
static double p_fp;
//========== Initialization ==========
double init_dh(double p, double t)                          //At constant power
{
double sum=0;
if(t==0)                    //fresh
        for(register i=0;i<=n_fp;i++) w[i]=0;
else if(t<0)                //equilibrium
        for(register i=0;i<=n_fp;i++){
                w[i]=gamm_fp[i]/lamb_fp[i]*p;
                sum+=lamb_fp[i]*w[i];
                }
else                        //after time t
        for(register i=0;i<=n_fp;i++){
                w[i]=gamm_fp[i]/lamb_fp[i]*p*(1.-exp(-lamb_fp[i]*t));
                sum+=lamb_fp[i]*w[i];
                }
return(p_fp=sum);
}

double init_dh(double wi[])                                 //Saved conditions
{
double sum=0;
        for(register i=0;i<=n_fp;i++){
                w[i]=wi[i];
                sum+=lamb_fp[i]*w[i];
                }
return(p_fp=sum);
}
//==========  Save to file ==========
int save_dh(ofstream& fout)
{
if(!fout) return 0;
for(register i=0;i<=n_fp-1;i++) fout<<w[i]<<" ";    fout<<endl;
return -1;
}
//========== Transient Decay Heat Fraction ==========
double decay_heat(double p, double dt)
{
if(dt<=0) return p_fp;
double sum=0;
        for(register i=0;i<=2;i++){
                w[i]=(w[i]+gamm_fp[i]*p*dt)/(1.+lamb_fp[i]*dt);
                sum+=lamb_fp[i]*w[i];
                }
return(p_fp=sum);
}
```

```
/*************************************************************************
                        Thermal-hydraulics Module
*************************************************************************/
#include <iostream.h>
#include <stdlib.h>
#include <iomanip.h>
#include <math.h>
#include <fstream.h>
#include <conio.h>
#include "water.h"
//Inline and define functions
inline double phi(double z)          //Axial flux shape
{
z=z*1.e2;//[cm]
return 2.0276e-7*pow(z,4) - 1.7156e-5*pow(z,3) - 5.3827e-4*pow(z,2) + 0.05050*z + 0.24983;
}
inline double aw(double T)           //Water temperature coefficient
{
double a=-1.7535e-5*T*T+7.5595e-4*T-4.9236e-3;
if(a<-1.2e-2) a= -1.2e-2;
return a;
}
inline double af(double T)           //Fuel temperature coefficient
{
double a=3.3970e-6*T*T-1.8497e-4*T-1.2057e-2;
if(a>-1.e-4) a=-1.e-4;
return a;
}
#define crhof(T) (1.985e6+0.0010e6*(T+273.15))    //U-Al heat capacity
#define crhoc(T) (2.069e6+0.0012e6*(T+273.15))    //Al6061 h.capacity
//function prototypes
double PHI(double z1, double z2, int n=10);
double friction(double Re);
double Nusselt(double Re, double P, double T);
void nodalize();
double init_th(double T);
double init_th(double p, double G, double T);
void th_dynamics(double p, double G, double T, double dt);
double decay_heat(double p, double dt=0);
//structures
struct NODE{
        int id;                 //node index
        int type;               //node type (OPEN or CLOSED)
        double z;               //node level from core mid-plane
        double dz;              //node height
        double elev;            //elevation to link
        double A;               //node area
        double V;               //node volume [m3]
        double dV;              //volume change
        double qf;              //heat generation fraction
        double S;               //heat transfer surface
        double Tc;              //clad temperature [C]
        double T;               //temperature [C]
        double P;               //pressure [MPa]
        double dP;              //pressure change
        double M;               //mass [kg]
        double dM;              //mass change
        double H;               //enthalpy [kJ]
```

```
                double dH;              //enthalpy change
                double Q;               //heat to node [W]
                double h;               //specific enthalpy [kJ/kg]
                double rho;             //density [kg/m3]
                double x;               //quality
                double hN;              //heat transfer coefficient
                double F1,F2,F3,F4,F5;          //F-functions
                double G1P,G2P,G1T,G2T;         //G-functions
                double ONB;             //margin to ONB
                double DNB;             //margin to DNB

                void GetFs();
                void heattrans(double p, double p_fp, int j, double dt=0);
                };

struct LINK{
                int up;         //upstream node number
                int down;       //downstream node number
                double A;       //flow area
                double D;       //hydraulic diameter
                double L;       //link length
                double k;        //resistance coefficient
                double sign;    //flow direction up=-1, down=+1
                double W;       //mass flow
                double dW;      //flow change
                double P;       //pressure
                double T;       //temperature
                double h;       //specific enthalpy
                double rho;     //density
                double x;       //quality
                };

//variables
static NODE *node,*hnode;
static LINK *link,*hlink;
static int ncore, nodes, links;         //total #s
static int nup;                         //upper pool node number
//constants
const int nfb=104;                      //total #fuel bundles
const double PI=3.1416;
const double Ac=342.3e-6;               //clad cross-section area
const double Af=133.1e-6;               //fueled cross-section area
const double g=9.81;                    //gravity [m/s2]
const double FP=5.e5;                   //full power [W]
const double B=0.0875;                  //barometric atm. [MPa]
const double alpha=0.95;                //fraction of fission heat deposited in fuel
const double kr=1.8;                    //radial peaking factor
const double fch=1.8;                   //core channel htc factor
const double Tref=20.;                  //[C]

//========== Range check of water properties ==========
double Clip_P(double P);
double Clip_T(double T);
double Clip_rho(double rho);
double Clip_h(double h);
double Clip_P(double P)
{
if(P<0.000612) P=0.000612;
```

```
if(P>21.5) P=21.5;
return P;
}

double Clip_T(double T)
{
if(T<0.004) T=0.004;
if(T>373.) T=373.;
return T;
}

double Clip_rho(double rho)
{
if(rho<0.0001) rho=0.0001;
if(rho>1000) rho=1000.;
return rho;
}

double Clip_h(double h)
{
if(h<0) h=0.;
if(h>3500) h=3500.;
return h;
}
//========== NODE member functions ==========
void NODE::GetFs()                                    //Get F and G functions of node
{
double h_f=hf(P), h_g=hg(P);
double v_f=vf(P), v_g=vg(P);
double DENOM;

if(h>h_f&&h<h_g){                          //two phase mixture
        F1=h_g*v_f - h_f*v_g;
        F2=v_g - v_f;
        F3=h_f - h_g;
        F4=dhgdP(P)*(v_g-v_f) - dvgdP(P)*(h_g-h_f);
        F5=dhfdP(P)*(v_g-v_f) - dvfdP(P)*(h_g-h_f);
        x=(h-h_f)/(h_g-h_f);
        DENOM=x*F4+(1-x)*F5;
        G1P=(h_g-h_f)/rho/rho/DENOM;
        G2P=(v_g-v_f)/DENOM;
        G1T=G1P*dTdP(P);
        G2T=G2P*dTdP(P);
        }
else{                                      //single phase
        double dhT,dhP,drT,drP;
        if(h<=h_f){                                   //subcooled fluid
                dhT=dhfdTP(P,T); drT=ddfdTP(P,T);
                dhP=dhfdPT(P,T); drP=ddfdPT(P,T);
                }
        else{                                         //superheated steam
                dhT=dhgdTP(P,T); drT=ddgdTP(P,T);
                dhP=dhgdPT(P,T); drP=ddgdPT(P,T);
                }
        F1=rho*dhT+h*drT;
        F2=-drT;
        F3=-rho*rho*dhT;
        F4= h<=h_f? 0:drP*dhT-drT*dhP;
```

```
        F5= h<=h_f? drP*dhT-drT*dhP:0;

        if(h<=h_f){
                G1P=dhT/F5;
                G2P=-drT/F5;
                G1T=-dhP/F5;
                G2T=drP/F5;
                x=0;
                }
        else{
                G1P=dhT/F4;
                G2P=-drT/F4;
                G1T=-dhP/F4;
                G2T=drP/F4;
                x=1;
                }
        }
}

void NODE::heattrans(double p, double p_fp, int j, double dt) //Heat Transfer
{
double Qf=(alpha*p+p_fp)*FP*qf; //heat deposited in fuel
double Qm=(1.-alpha)*p*FP*qf;   //heat generated in moderator

double W,A,D;
if(id>=1000){
        W=hlink[j].W;   A=hlink[j].A;   D=hlink[j].D;
        }
else{
        W=link[j].W;    A=link[j].A;    D=link[j].D;
        }
double Re=fabs(W)*D/A/visx(P,T,x)*1e6;
double Nu=Nusselt(Re,P,T);
        Nu=fch*Nu*pow(visx(P,T,x)/visx(P,Tc,x),0.14);

double hSP=Nu*kx(P,T,x)/D;      //Single phase HTC

double Tsat=Ts(P);              //Local saturated temperature
double DT,TONB,q;
double hNB=0,b=0;

if(Tc>Tsat){                    //Check if nucleate boiling
        q=hSP*(Tc-T)*3.155;     //SP heat flux [Btu/ft2-hr]
        DT=pow(q/15.6/pow(145.04*P,1.156),1./2.3/pow(145.04*P,-0.0234));//Overheating [F]
        TONB=Tsat+5./9.*DT;     //ONB temperatute

        if(Tc>TONB){                            //Nucleate boiling
                DT=9./5.*(Tc-Tsat);             //Overheating [F]
        double qNB=15.6*pow(145.04*P,1.156)
                        *pow(DT,2.3*pow(145.04*P,-0.0234));//NB heat flux [Btu/ft2.hr]
                hNB=qNB/3.155/(Tc-Tsat);        //NB HTC
                b=(Tc-Tsat)/(Tc-T);             //NB fraction
                }
        }

hN=(1.-b)*hSP + b*hNB;          //Total HTC

if(dt<0) dt=0;
```

```
if(dt>0){                              //non-steady
        double C=(crhof(Tc)*Af+crhoc(Tc)*Ac)*dz*nfb;    //Heat capacity [J/K]

        Tc=(Tc+(hN*S*T+Qf)*dt/C)/(1.+hN*S*dt/C);        //Lumped form
        }
else                                   //steady-state
        Tc=T+Qf/S/hN;

Q=hN*S*(Tc-T)+Qm;                      //heat to node

//Margin to ONB [Bergles-Rhosenow]
        q=hSP*(Tc-T)*3.155;     //SP heat flux [Btu/ft2-hr]
        DT=pow(q/15.6/pow(145.04*P,1.156),  1./2.3/pow(145.04*P,-0.0234));//Overheating [F]
        TONB=Tsat+5./9.*DT;     //ONB temperatute
        ONB=(TONB-T)/(Tc-T);  //Margin to ONB

//Margin to DNB [Mirshak]
        DT=Tsat-T;
        double w=W/rho/A;       //Flow velocity
        double CHF=98.15*(1+0.1198*fabs(w)) *(1+0.00914*DT)*(1+1.9*P);//CHF [kW/m2]
        DNB=CHF*1e3/hN/(Tc-T);//Margin to DNB
}
//========== Integral of flux shape function ==========
double PHI(double z1, double z2, int n)
{
if(n<=0) n=10;
double dz=(z2-z1)/n;
double sum=-(phi(z1)+phi(z2))/2;
double z=z1;
        for(int i=0;i<=n;i++,z+=dz) sum+=phi(z);
return sum*dz;
}
//========== Nodalization ==========
void nodalize()
{
const double Hf=0.6;                   //active fuel height [m]
const double Hfen=0.065;               //non-fueled entrance length
const double Hfex=0.065;               //non-fueled exit length
const double Hchim=2.0;                //chimney height
const double Hup=3.0;                  //water column above chimney
const double Hlo=0.3;                  //water layer under core
const double Dchim=0.5;                //chimney diameter [m]
const double Dpool=1.98;               //pool diameter
const double Acore=585.5e-6;           //flow area [m2]
const double Pcore=0.38;               //wetted perimeter [m]
const double k_en_core=7.;             //resistance of core entrance
const double k_ex_core=15.;            // resistance of core exit
const double k_ex_chim=1.5;            //resistance of chimney exit
const double k_pool=2.5;               //resistance of pool path
ncore=10;                              //#core nodes
nodes=ncore+3;                         //# nodes=cores+chimney+upper+lower
links=nodes+2;                         //#links, incl. from and to primary loop
if((node=new NODE[nodes])==NULL) exit(1);
if((link=new LINK[links])==NULL) exit(1);
if((hnode=new NODE[ncore+2])==NULL) exit(1);
if((hlink=new LINK[ncore+1])==NULL) exit(1);

int i,j;
```

```
//Assigning Nodes
double dz,z,A,V,S,kzmax;
        dz=Hf/ncore;
        A=Acore*nfb;
        V=A*dz;
        S=Pcore*dz*nfb;          //heat transfer area
        kzmax=PHI(0,Hf,100);
for(i=0;i<=ncore-1;i++){  //core
        node[i].id=i;
        node[i].type=0;          //closed
        node[i].dz=dz;
        node[i].z=z=(i+0.5)*dz;
        node[i].elev=dz/2;
        node[i].A=A;
        node[i].V=V;
        node[i].S=S;
        node[i].qf=PHI(z-dz/2,z+dz/2)/kzmax;
        }
i=ncore;                 //chimney
        node[i].id=100;
        node[i].type=0;          //closed
        node[i].dz=Hchim;
        node[i].z=Hf+Hfex+Hchim/2;
        node[i].elev=Hchim/2;
        node[i].A=PI*Dchim*Dchim/4;
        node[i].V=node[i].A*Hchim;
        node[i].S=PI*Dchim*Hchim;//chimney wall
        node[i].qf=0;
i++;                     //upper pool
        nup=i;
        node[i].id=200;
        node[i].type=1;          //open
        node[i].dz=Hup;
        node[i].z=Hf+Hfex+Hchim+Hup/2;
        node[i].elev=Hup/2;
        node[i].A=A=PI*Dpool*Dpool/4;
        node[i].V=A*Hup;
        node[i].S=PI*Dpool*Hup;//optional
        node[i].qf=0;
i++;                     //lower pool
        node[i].id=300;
        node[i].dz=dz=Hlo+Hfen+Hf+Hfex+Hchim;
        node[i].type=0;          //closed
        node[i].z=-Hfen;
        node[i].elev=0;
        node[i].A=A;
        node[i].V=A*dz-node[ncore].A*(dz-Hlo);
        node[i].S=node[ncore].S;//chimney wall
        node[i].qf=0;

//Assigning Links
j=0;                     //lower-pool to core
        link[j].up=nodes-1;
        link[j].down=j;
        link[j].sign=-1;
        link[j].A=node[0].A;
        link[j].D=4*Acore/Pcore;
        link[j].L=Hfen;
```

```
        link[j].k=k_en_core;

for(j=1;j<=ncore-1;j++){  //in-core
        link[j].up=j-1;
        link[j].down=j;
        link[j].sign=-1;
        link[j].A=link[0].A;
        link[j].D=link[0].D;
        link[j].L=Hf/ncore;
        link[j].k=0;
        }

j=ncore;                        //end-core to chimney
        link[j].up=j-1;
        link[j].down=j;
        link[j].sign=-1;
        link[j].A=link[0].A;
        link[j].D=link[0].D;
        link[j].L=Hf/ncore+Hfex;
        link[j].k=k_ex_core;

j++;                            //chimney
        link[j].up=j-1;
        link[j].down=j;
        link[j].sign=-1;
        link[j].A=node[ncore].A;
        link[j].D=Dchim;
        link[j].L=Hchim;
        link[j].k=k_ex_chim;

j++;                            //surrounding pool
        link[j].up=j-1;
        link[j].down=j;
        link[j].sign=1;
        link[j].A=node[ncore+1].A-node[ncore].A;
        link[j].D=Dpool-Dchim;
        link[j].L=node[nodes-1].dz-Hlo;
        link[j].k=k_pool;

        link[links-2].up=nup;       //to HX
        link[links-2].down=400;

        link[links-1].up=500;       //from pump
        link[links-1].down=nodes-1;
//Hot nodes
for(i=0;i<=ncore+1;i++){
        if(i==ncore+1)  hnode[i]=node[nodes-1];
        else            hnode[i]=node[i];
        hnode[i].qf=kr*hnode[i].qf;
        hnode[i].id=hnode[i].id+1000;
        }
//Hot links
for(j=0;j<=ncore;j++){
        hlink[j]=link[j];
        if(j==0) hlink[j].up=ncore+1;
        }
}
```

```
//========== Friction coefficient for circular tube ==========
double friction(double Re)
{
double f;
if (Re<=0)              f=0;
else if(Re<=1085)       f=64./Re;
else                    f=0.0056+0.5/pow(Re,0.32);
return(f);
}
//========== Nusselt number for circular tube ==========
double Nusselt(double Re, double P, double T)
{
double Nu;
if(Re<=2300) Nu=4.36;   //analytical
else{
        double  Pr=visx(P,T)*cpx(P,T)*1.e-3/kx(P,T);
        double  f=1./pow(1.58*log(Re)-3.28,2);
        Nu=(f/2.)*(Re-500.)*Pr/(1.+12.7*sqrt(f/2.)*(pow(Pr,1.5)-1.));          //Gnielinskiy
        }
return(Nu);
}

//========== Initialization ==========
double init_th(double T)            //Statics
{
double P;
double HH=node[nup].z+node[nup].dz/2;
double rho=dx(B,T);
int i,j;
//Average channel
for(i=0;i<=nodes-1;i++){
        node[i].P=P=B+rho*g*(HH-node[i].z)*1e-6;
        node[i].T=T;
        node[i].Tc=T;
        node[i].rho=dx(P,T);
        node[i].h=hx(P,T);
        node[i].x=Tx(P,node[i].h,1);
        node[i].M=node[i].rho*node[i].V;
        node[i].H=node[i].h*node[i].M;
        node[i].Q=0;

        node[i].GetFs();
        }

for(j=0;j<=links-1;j++)   link[j].W=0;

//Hot channel
for(i=0;i<=ncore+1;i++){
        hnode[i]=node[i];
        if(i==ncore+1)   hnode[i]=node[nodes-1];
        hnode[i].qf=kr*hnode[i].qf;
        hnode[i].id=hnode[i].id+1000;
        }
for(j=0;j<=ncore;j++) hlink[j].W=0;

return (aw(T)+af(T))*(T-Tref);
}
```

```
double init_th(double p, double G, double Tin)        //Steady-state Approximation
{
init_th(Tin);

//Average channel
double epsiW=1e-3;
int max_iter=100;
double p_fp=decay_heat(p);
double DP=node[nodes-1].P*1e6+node[nodes-1].rho*g*node[nodes-1].elev
                -(node[nup].P*1e6+node[nup].rho*g*node[nup].elev);
double Wc=p*5.;
        if(Wc==0) Wc=0.01;      //guessed
int i,j,u,d;
double P_u,P_d, A,D,L;
double W,P,T,Tc,h,rho,x,Re,K,f;
double DPb,DPf,KK;

for(i=0;i<=nodes-1;i++) node[i].Q=node[i].qf*(p+p_fp)*FP;  //heat transfer

int it=0;
do{     it++;    W=Wc;

for(j=0;link[j].down<=nup;j++) link[j].W=W;

h=node[nodes-1].h;
for(i=0;i<=nup;i++){
        node[i].h=h+node[i].Q*1e-3/W;
        h=node[i].h;
        P=node[i].P;
        T=node[i].T=Tx(P,h);
        x=node[i].x=Tx(P,h,1);
        rho=node[i].rho=dx(P,T,x);

        if(i<=ncore-1)    node[i].heattrans(p,p_fp,i+1);
        else              node[i].Tc=T;
        }
DPb=DP; DPf=KK=0;
for(j=0;link[j].down<=nup;j++){
        u=link[j].up; d=link[j].down; L=link[j].L; A=link[j].A; D=link[j].D;
        P_u=node[u].P*1e6+node[u].rho*g*node[u].elev;
        h=W>0?node[u].h:node[d].h;
        rho=node[u].rho;  P=node[u].P; T=node[u].T; x=node[u].x; Tc=node[u].Tc;

        Re=fabs(W)*D/A/visx(P,T,x)*1e6;
        f=friction(Re)*pow(visx(P,T,x)/visx(P,Tc,x),-0.58);  if(j<=ncore) f=1.5*f;
        K=(f*L/D+link[j].k)/2./rho/A/A;

        P_d=P_u-K*fabs(W)*W+rho*g*L*link[j].sign;
        node[d].P=(P_d-node[d].rho*g*node[d].elev)*1e-6;

        DPb+=rho*g*L*link[j].sign;
        DPf+=K*fabs(W)*W;
        KK+=K;
        }
Wc=sqrt(fabs(DPb+DPf)/2/KK)*W/fabs(W);

}while(fabs(W-Wc)>epsiW&&it<=max_iter);
```

```
node[nup].dz=2*(node[nup].P-B)*1e6/g/node[nup].rho;
node[nup].V=node[nup].dz*node[nup].A;
link[links-3].W=W-G;

double Tw,Tf; Tw=Tf=0;
for(i=0;i<=nodes-1;i++){
        node[i].M=node[i].rho*node[i].V;
        node[i].H=node[i].h*node[i].M;
        node[i].GetFs();
        node[i].dP=0;

        if(i<=ncore-1){
                Tw+=node[i].T/ncore;
                Tf+=node[i].Tc/ncore;
                }
        }
//Hot channel
hnode[ncore]=node[ncore];//chiney
hnode[ncore+1]=node[nodes-1];//lower pool

DP=node[nodes-1].P*1e6+node[nodes-1].rho*g*node[nodes-1].elev
        -node[ncore].P*1e6-node[ncore].rho*g*node[ncore].elev;

for(i=0;i<=ncore+1;i++) hnode[i].Q=hnode[i].qf*(p+p_fp)*FP;

it=0;
do{     it++;     W=Wc;

for(j=0;j<=ncore;j++) hlink[j].W=W;

h=hnode[ncore+1].h;
for(i=0;i<=ncore-1;i++){
        hnode[i].h=h+hnode[i].Q*1e-3/W;
        h=hnode[i].h;
        P=hnode[i].P;
        T=hnode[i].T=Tx(P,h);
        x=hnode[i].x=Tx(P,h,1);
        rho=hnode[i].rho=dx(P,T,x);
        hnode[i].heattrans(p,p_fp,i+1);
        }
DPb=DP; DPf=KK=0;
for(j=0;j<=ncore;j++){
        u=hlink[j].up; d=hlink[j].down; L=hlink[j].L; A=hlink[j].A; D=hlink[j].D;

        P_u=hnode[u].P*1e6+hnode[u].rho*g*hnode[u].elev;
        h=W>0?hnode[u].h:hnode[d].h;
        rho=hnode[u].rho; P=hnode[u].P; T=hnode[u].T; x=hnode[u].x; Tc=hnode[u].Tc;

        Re=fabs(W)*D/A/visx(P,T,x)*1e6;
        f=friction(Re)*pow(visx(P,T,x)/visx(P,Tc,x),-0.58);  if(j<=ncore) f=1.5*f;
        K=(f*L/D+hlink[j].k)/2./rho/A/A;

        P_d=P_u-K*fabs(W)*W+rho*g*L*hlink[j].sign;

        if(d<=ncore-1) hnode[d].P=(P_d-hnode[d].rho*g*hnode[d].elev)*1e-6;

        DPb+=rho*g*L*hlink[j].sign;
        DPf+=K*fabs(W)*W;
```

```
        KK+=K;
        }
Wc=sqrt(fabs(DPb+DPf)/2/KK)*W/fabs(W);

}while(fabs(W-Wc)>epsiW&&it<=max_iter);

for(i=0;i<=ncore+1;i++){
        hnode[i].M=hnode[i].rho*hnode[i].V;
        hnode[i].H=hnode[i].h*hnode[i].M;
        hnode[i].GetFs();
        hnode[i].dP=0;
        }
return(aw(Tw)*(Tw-Tref)+af(Tf)*(Tf-Tref));
}

double init_th(ifstream& fin)                    //Saved conditions
{
if(!fin) return 999;

int i,j;
for(i=0;i<=nodes-1;i++) fin>>i>>node[i].P>>node[i].dP>>node[i].T>>node[i].M>>node[i].H
                        >>node[i].rho>>node[i].h>>node[i].Tc;

for(j=0;j<=links-3;j++) fin>>j>>link[j].W;

for(i=0;i<=ncore-1;i++) fin>>i>>hnode[i].P>>hnode[i].dP>>hnode[i].T>>hnode[i].M>>hnode[i].H
                        >>hnode[i].rho>>hnode[i].h>>hnode[i].Tc;

for(j=0;j<=ncore;j++) fin>>j>>hlink[j].W;
fin.close();

double Tw,Tf;    Tw=Tf=0;
for(i=0;i<=nodes-1;i++){
        node[i].GetFs();

        if(i<=ncore-1){
                Tw+=node[i].T/ncore;
                Tf+=node[i].Tc/ncore;
                }
        }

for(i=0;i<=ncore-1;i++)   hnode[i].GetFs();

return(aw(Tw)*(Tw-Tref)+af(Tf)*(Tf-Tref));
}
//========== Thermal-hydraulic Dynamics ==========
void th_dynamics(double p, double G, double Tin, double dt)
{
double p_fp=decay_heat(p,dt);

//Average channel
double P,T,Tc,h,rho,x,dP,dT,dh,drho;
double W,dW,P_u,P_d,dP_u,dP_d, chi_u, chi_d;
double Re,f,K,A,D,L,sign;
int i,j,u,d;
double hin=hx(B,Tin);    //inlet enthalpy

for(i=0;i<=nodes-1;i++){
```

```
        if(i<=ncore-1) node[i].heattrans(p,p_fp,i+1,dt);        //heat transfer
        else{
                node[i].Q=0;
                node[i].Tc=node[i].T;
                }
        if(i<nup){
                node[i].dM=0;
                node[i].dH=node[i].Q*dt*1e-3;
                }
        else if(i==nup){
                node[i].dM=-G*dt;
                node[i].dH=-G*node[i].h*dt;
                }
        else{
                node[i].dM=G*dt;
                node[i].dH=G*hin*dt;
                }
        }

for(j=0;j<=links-3;j++){   //links
        A=link[j].A; D=link[j].D; L=link[j].L; sign=link[j].sign; u=link[j].up; d=link[j].down;

        P_u=node[u].P*1e6+node[u].elev*node[u].rho*g;
        P_d=node[d].P*1e6+node[d].elev*node[d].rho*g;
        dP_u=node[u].dP*1e6;
        dP_d=node[d].dP*1e6;
        W=link[j].W;
        h=link[j].h=W>0? node[u].h:node[d].h;

        if(d<=nup){
                P=link[j].P=node[u].P; rho=link[j].rho=node[u].rho;
                T=link[j].T=node[u].rho; Tc=node[u].Tc; x=node[u].x;
                }
        else{
                P=link[j].P=node[d].P; rho=link[j].rho=node[d].rho;
                T=link[j].T=node[d].T; Tc=node[d].Tc; x=node[d].x;
                }

        if(u==nup||u==nodes-1) chi_u=0.;
        else    chi_u=(node[u].F1+node[u].F2*h) /node[u].M*1e6
                        /(node[u].x*node[u].F4+(1.-node[u].x)*node[u].F5);

        if(d==nup||d==nodes-1) chi_d=0.;
        else    chi_d=(node[d].F1+node[d].F2*h) /node[d].M*1e6
                        /(node[d].x*node[d].F4+(1.-node[d].x)*node[d].F5);

        Re=fabs(W)*D/A/visx(P,T,x)*1e6;
        f=friction(Re)*pow(visx(P,T,x)/visx(P,Tc,x),-0.58);  if(j<=ncore) f=1.5*f;
        K=(f*L/D+link[j].k)/2./rho/A/A;

        dW=A/L*dt*(P_u-P_d-K*fabs(W)*W+sign*rho*g*L + dP_u-dP_d)
                /(1.+K*fabs(W)*A/L*dt+(chi_u+chi_d)*A/L*dt*dt);          //flow equation

        link[j].dW=dW; W+=dW;

        node[u].dM-=W*dt;
        node[d].dM+=W*dt;
```

```
        node[u].dH-=W*h*dt;
        node[d].dH+=W*h*dt;
        }

for(j=0;j<=links-3;j++) link[j].W+=link[j].dW;

for(i=0;i<=nodes-1;i++){  //nodes
        node[i].M+=node[i].dM;
        rho=node[i].rho;

        node[i].H+=node[i].dH;
        h=node[i].h;
        node[i].h=Clip_h(node[i].H/node[i].M);
        dh=node[i].h-h;

        if(node[i].type==1){        //open node
                node[i].dP=dP=node[i].dM*g/2/node[i].A*1e-6;
                drho=(dP-node[i].G2P*dh)/node[i].G1P;
                node[i].rho+=drho;
                }
        else{                       //closed node
                node[i].rho=Clip_rho(node[i].M/node[i].V);
                drho=node[i].rho-rho;
                node[i].dP=dP=node[i].G1P*drho+node[i].G2P*dh;
                }
        dT=node[i].G1T*drho+node[i].G2T*dh;
        node[i].P=Clip_P(node[i].P+dP);
        node[i].T=Clip_T(node[i].T+dT);
        node[i].GetFs();
        }

//Hot Channel
hnode[ncore]=node[ncore];//chimney
hnode[ncore+1]=node[nodes-1];//lower-pool

for(i=0;i<=ncore-1;i++){
        hnode[i].heattrans(p,p_fp,i+1,dt);
        hnode[i].dM=0;
        hnode[i].dH=hnode[i].Q*dt*1e-3;
        }
for(j=0;j<=ncore;j++){    //links
        A=hlink[j].A; D=hlink[j].D; L=hlink[j].L; sign=hlink[j].sign;
        u=hlink[j].up; d=hlink[j].down;
        P_u=hnode[u].P*1e6+hnode[u].elev*hnode[u].rho*g;
        P_d=hnode[d].P*1e6+hnode[d].elev*hnode[d].rho*g;
        dP_u= u==ncore+1? 0:hnode[u].dP*1e6;
        dP_d= d==ncore? 0:hnode[d].dP*1e6;
        W=hlink[j].W;
        h=hlink[j].h=W>0?hnode[u].h:hnode[d].h;
        P=hlink[j].P=hnode[u].P;  T=hlink[j].T=hnode[u].T;
        rho=hlink[j].rho=hnode[u].rho; Tc=hnode[u].Tc; x=hnode[u].x;

        if(u==ncore+1) chi_u=0.;
        else      chi_u=(hnode[u].F1+hnode[u].F2*h) /hnode[u].M*1e6
                  /(hnode[u].x*hnode[u].F4+(1.-hnode[u].x)*hnode[u].F5);

        if(d==ncore) chi_d=0.;
        else      chi_d=(hnode[d].F1+hnode[d].F2*h) /hnode[d].M*1e6
```

```
                /(hnode[d].x*hnode[d].F4+(1.-hnode[d].x)*hnode[d].F5);


        Re=fabs(W)*D/A/visx(P,T,x)*1e6;
        f=friction(Re)*pow(visx(P,T,x)/visx(P,Tc,x),-0.58);
        if(j<=ncore) f=1.5*f;
        K=(f*L/D+link[j].k)/2./rho/A/A;

        dW=A/L*dt*(P_u-P_d-K*fabs(W)*W+sign*rho*g*L + dP_u-dP_d)
                /(1.+K*fabs(W)*A/L*dt + (chi_u+chi_d)*A/L*dt*dt );          //flow equation

        hlink[j].dW=dW; W+=dW;

        hnode[u].dM-=W*dt;
        hnode[d].dM+=W*dt;

        hnode[u].dH-=W*h*dt;
        hnode[d].dH+=W*h*dt;
        }

for(j=0;j<=ncore;j++) hlink[j].W+=hlink[j].dW;

for(i=0;i<=ncore-1;i++){  //nodes
        hnode[i].M+=hnode[i].dM;
        rho=hnode[i].rho;

        hnode[i].H+=hnode[i].dH;
        h=hnode[i].h;
        hnode[i].h=Clip_h(hnode[i].H/hnode[i].M);
        dh=hnode[i].h-h;

        hnode[i].rho=Clip_rho(hnode[i].M/hnode[i].V);
        drho=hnode[i].rho-rho;

        hnode[i].dP=dP=hnode[i].G1P*drho+hnode[i].G2P*dh;
        dT=hnode[i].G1T*drho+hnode[i].G2T*dh;

        hnode[i].P=Clip_P(hnode[i].P+dP);
        hnode[i].T=Clip_T(hnode[i].T+dT);
        hnode[i].GetFs();
        }
}//th_dynamics
//========== Temperature Feedback ==========
double rho_temp(double p, double G, double Tin, double dt)
{
if(Tin<=0) Tin=node[nup].T;

th_dynamics(p,G/3.6,Tin,dt);

double Tw,Tf;    Tw=Tf=0;
extern double dPmax;

for(int i=0;i<=ncore-1;i++){
        if(fabs(node[i].dP)>dPmax) dPmax=fabs(node[i].dP);
        if(fabs(hnode[i].dP)>dPmax) dPmax=fabs(hnode[i].dP);

        Tw+=node[i].T/ncore;
        Tf+=node[i].Tc/ncore;
```

```
        }
return(aw(Tw)*(Tw-Tref)+af(Tf)*(Tf-Tref));
}
//========== Output thermal-hydraulic parameters ==========
int th_out(ofstream& fout)
{
double Tc,ONB,DNB,Q,Tb;
        DNB=ONB=999;Tc=Q=0;

for(int i=0;i<=ncore-1;i++){ //Core
        Q+=node[i].Q;
        if(hnode[i].Tc>Tc){
                Tc=hnode[i].Tc;
                Tb=hnode[i].ONB*(hnode[i].Tc-hnode[i].T)+hnode[i].T;
                }
        if(hnode[i].ONB<ONB)    ONB=hnode[i].ONB;
        if(hnode[i].DNB<DNB) DNB=hnode[i].DNB;
        }
//out put to file
if(!fout) return 0;

fout<<" "<<link[0].W                                    //inlet core flow
        <<" "<<link[ncore-1].W                          //outlet core flow
        <<" "<<node[nodes-1].T                          //inlet to core
        <<" "<<node[ncore-1].T                          //outlet from core
        <<" "<<link[links-3].W                          //pool flow
        <<" "<<link[ncore].W*link[ncore].h-link[0].W*link[0].h      //core flow heat
        <<" "<<Q*1e-3                                   //heat from fuel
        <<" "<<hlink[0].W                               //inlet hot flow
        <<" "<<hlink[ncore].W                           //outlet hot flow
        <<" "<<Tc                                       //max surface temp
        <<" "<<Tb                                       //ONB temp
        <<" "<<ONB                                      //min margin to ONB
        <<" "<<DNB;                                     //min margin to DNB
return -1;
}
//========== Save to file ==========
int save_th(ofstream& fout)
{
if(!fout)  return 0;

int i,j;
for(i=0;i<=nodes-1;i++)
        fout<<i<<" "<<node[i].P<<" "<<node[i].dP<<" "<<node[i].T<<" "<<node[i].M<<" "
                <<node[i].H<<" "<<node[i].rho<<" "<<node[i].h<<" "<<node[i].Tc<<endl;
for(j=0;j<=links-3;j++)
        fout<<j<<" "<<link[j].W<<endl;
for(i=0;i<=ncore-1;i++)
        fout<<i<<" "<<hnode[i].P<<" "<<hnode[i].dP<<" "<<hnode[i].T<<" "<<hnode[i].M<<" "
                <<hnode[i].H<<" "<<hnode[i].rho<<" "<<hnode[i].h<<" "<<hnode[i].Tc<<endl;
for(j=0;j<=ncore;j++)
        fout<<j<<" "<<hlink[j].W<<endl;
return -1;
}
```

BIOGRAPHY

Mr. Nguyễn Thái Sinh was born on January 21, 1961 in Quảng Bình Province, Viet Nam. He graduated as an Engineer in the field of Thermo-Energetics from the Zaporozhye Industrial Institute, Ucraina (former USSR), in 1984. Since then, he has been working for the Dalat Nuclear Research Institute, Vietnam Atomic Energy Commission, as a researcher in reactor physics and engineering and as an engineer-operator of the Dalat Nuclear Research Reactor.

Since June 1997, he has been studying for the Master Degree in Department of Nuclear Technology, Faculty of Engineering, Chulalongkorn University, Bangkok, Thailand.