Modelling the tank emptying: code

Here is an extract from our hybrid simulator, dealing with the nitrous tank emptying.
Please read our 'modelling the tank emptying' paper for explanation of the code.
The code is written in C++, but is very nearly standard C
This code has to be embedded into a hybrid simulation in order to calculate combustion chamber pressure. (You could use test data: a pressure trace of chamber pressure.)

Software

Firstly, a listing for subroutines that calculate nitrous oxide physical properties

```
/*****************************************************************************
 ** File       : nitrous oxide.cpp                                           **
 **                                                                          **
 ** Description : Nitrous oxide physical properties                          **
 **             From Engineering Sciences Data Unit 91022                    **
 **                                                                          **
 ** Created     : 6/6/2000 Rick Newlands, Aspirespace                        **
 *****************************************************************************/

/* Functions tested and checked 6/6/2000                                     */
/* Note that functions are only valid to 36.0 deg C                          */
/* except nox_CpL which is stated to be valid only up to 30.0 deg C */
/* and nox_KL which currently roofs at 10.0 deg C                            */

/* This C++ version 4/3/03 Rick Newlands       */
/* initial Revision Rick Newlands, Aspirespace */

#include "stdafx.h"  /* (standard C++ header) */
#include <math.h>
#include "nitrous_oxide.h" /* header file */
```

```
const float pCrit    = 72.51f; /* critical pressure, Bar Abs */
const float rhoCrit = 452.0f;  /* critical density, kg/m3 */
const float tCrit   = 309.57f; /* critical temperature, Kelvin (36.42 Centigrade) */

static int dd;
static double bob, rab, shona, Tr;


/* signum of a number, used below */
short int SGN(double bob)
{
  short int signum;

  if (bob >= 0.0)
    signum = 1;
  else
    signum = -1;

  return (signum);
}


/* Nitrous oxide vapour pressure, Bar */
double nox_vp(double T_Kelvin)
{
  const float p[4] = {1.0f, 1.5f, 2.5f, 5.0f};
  const float b[4] = {-6.71893f, 1.35966f, -1.3779f, -4.051f};

  Tr = T_Kelvin / tCrit;
  rab = 1.0 - Tr;
  shona = 0.0;

  for (dd = 0; dd < 4; dd++)
    shona += b[dd] * pow(rab,p[dd]);
```

```
  bob = pCrit * exp((shona / Tr));

  return(bob);
}


/* Nitrous oxide saturated liquid density, kg/m3 */
double nox_Lrho(double T_Kelvin)
{
  const float b[4] = {1.72328f, -0.8395f, 0.5106f, -0.10412f};

  Tr = T_Kelvin / tCrit;
  rab = 1.0 - Tr;
  shona = 0.0;

  for (dd = 0; dd < 4; dd++)
    shona += b[dd] * pow(rab,((dd+1) / 3.0));

  bob = rhoCrit * exp(shona);

  return(bob);
}


/* Nitrous oxide saturated vapour density, kg/m3 */
double nox_Vrho(double T_Kelvin)
{
  const float b[5] = {-1.009f, -6.28792f, 7.50332f, -7.90463f, 0.629427f};

  Tr = T_Kelvin / tCrit;
  rab = (1.0 / Tr) - 1.0;
  shona = 0.0;

  for (dd = 0; dd < 5; dd++)
    shona += b[dd] * pow(rab,((dd+1) / 3.0));
```

```
  bob = rhoCrit * exp(shona);

  return(bob);
}


/* Nitrous Enthalpy (Latent heat) of vaporisation, J/kg */
double nox_enthV(double T_Kelvin)
{
  const float bL[5] = {-200.0f, 116.043f, -917.225f, 794.779f, -589.587f};
  const float bV[5] = {-200.0f, 440.055f, -459.701f, 434.081f, -485.338f};

  double shonaL, shonaV;


  Tr = T_Kelvin / tCrit;
  rab = 1.0 - Tr;
  shonaL = bL[0];
  shonaV = bV[0];

  for (dd = 1; dd < 5; dd++)
  {
    shonaL += bL[dd] * pow(rab,(dd / 3.0)); /* saturated liquid enthalpy */
    shonaV += bV[dd] * pow(rab,(dd / 3.0)); /* saturated vapour enthalpy */
  }

  bob = (shonaV - shonaL) * 1000.0; /* net during change from liquid to vapour */

  return(bob);
}


/* Nitrous saturated liquid isobaric heat capacity, J/kg K */
double nox_CpL(double T_Kelvin)
```

```
{
  const float b[5] = {2.49973f, 0.023454f, -3.80136f, 13.0945f, -14.518f};

  Tr = T_Kelvin / tCrit;
  rab = 1.0 - Tr;
  shona =  1.0 + b[1] / rab;

  for (dd = 1; dd < 4; dd++)
    shona += b[(dd+1)] * pow(rab,dd);

  bob = b[0] * shona * 1000.0; /* convert from KJ to J */

  return(bob);
}


/* liquid nitrous thermal conductivity, W/m K */
double nox_KL(double T_Kelvin)
{
  const float b[4] = {72.35f, 1.5f, -3.5f, 4.5f};

  /* max. 10 deg C */
  if (T_Kelvin > 283.15)
    Tr = 283.15 / tCrit;
  else
    Tr = T_Kelvin / tCrit;

  rab = 1.0 - Tr;
  shona = 1.0 + b[3] * rab;

  for (dd = 1; dd < 3; dd++)
    shona += b[dd] * pow(rab,(dd / 3.0));

  bob = b[0] * shona / 1000; /* convert from mW to W */
```

```
  return(bob);
}


/* nox_temperature based on pressure (bar) */
double nox_on_press(double P_Bar_abs)
{
  const float p[4] = {1.0f, 1.5f, 2.5f, 5.0f};
  const float b[4] = {-6.71893f, 1.35966f, -1.3779f, -4.051f};

  double pp_guess, step, tempK;

  step = -1.0;
  tempK = (tCrit - 0.1) - step;

  do /* iterative loop */
  {
    do
    {
      tempK += step;
      Tr = tempK / tCrit;
      rab = 1.0 - Tr;
      shona = 0.0;

      for (dd = 0; dd < 4; dd++)
        shona += b[dd] * pow(rab,p[dd]);

      pp_guess = pCrit * exp((shona / Tr));
    }
    while ( ((pp_guess - P_Bar_abs) * SGN(step)) < 0.0);

    step = step / (-2.0); /* reduce step size */
  }
  while (fabs((pp_guess - P_Bar_abs)) > 0.01);
```

```
  bob = tempK;

  return(bob); /* return temperature */
}
```

```
*************************************************************************
/* header file nitrous oxide.h */

/* Nitrous oxide vapour pressure, Bar */
double nox_vp(double T_Kelvin);

/* Nitrous oxide saturated liquid density */
double nox_Lrho(double T_Kelvin);

/* Nitrous oxide saturated vapour density */
double nox_Vrho(double T_Kelvin);

/* Nitrous oxide Enthalpy (Latent heat) of vapourisation */
double nox_enthV(double T_Kelvin);

/* Nitrous oxide saturated liquid isobaric heat capacity */
double nox_CpL(double T_Kelvin);

/* Nitrous oxide liquid thermal conductivity, W/m K */
double nox_KL(double T_Kelvin);

/* mean temperature K based on pressure */
double nox_on_press(double P_Bar_abs);
```

Tank emptying subroutines

```
/* Tank emptying code extracts */
/* (c) AspireSpace */

/* Variables are in metric units except where stated otherwise */

#include "nitrous_oxide.h" /* header file for nitrous oxide property calcs subroutines */


/* prototypes */
static double injector_model(double upstream_pressure, double downstream_pressure);


/* square */
double SQR(double bob)
{
  return((bob * bob));
}


#define CENTIGRADE_TO_KELVIN  273.15 // to Kelvin

#define BAR_TO_PASCALS       100000.0
#define PASCALS_TO_BAR (1.0 / BAR_TO_PASCALS)


/* calculate injector pressure drop (Bar) and mass flowrate (kg/sec) */
static double injector_model(double upstream_pressure, double downstream_pressure)
{
  double mass_flowrate;
  double pressure_drop;

  pressure_drop = upstream_pressure - downstream_pressure; /* Bar */
```

```
  /* reality check */
  if (pressure_drop < 0.00001)
    pressure_drop = 0.00001;

  /* is injector pressure drop lower than 20 percent of chamber pressure? */
  if ((pressure_drop / hybrid.chamber_pressure_bar) < 0.2)
      hybrid.hybrid_fault = 3; // too low for safety


  /* Calculate fluid flowrate through the injector, based on the     */
  /* total-pressure loss factor between the tank and combustion chamber */
  /* (injector_loss_coefficient includes K coefficient and orifice cross-sectional areas) */
  mass_flowrate =
      sqrt((2.0 * hybrid.tank_liquid_density * pressure_drop / hybrid.injector_loss_coefficient));

  return(mass_flowrate); /* kg/sec */
}



/* Equilibrium (instantaneous boiling) tank blowdown model */
void Nitrous_tank(void)
{
  double bob;
  double Chamber_press_bar_abs;
  double delta_outflow_mass, deltaQ, deltaTemp;
  double Enth_of_vap;
  double Spec_heat_cap;
  double tc;

  static double lagged_bob = 0.0;


  /* blowdown simulation using nitrous oxide property calcs subroutines */
```

```
/* update last-times values, O = 'old' */
Omdot_tank_outflow = mdot_tank_outflow;


Enth_of_vap = nox_enthV(hybrid.tank_fluid_temperature_K); /* Get enthalpy (latent heat) of vaporisation */
Spec_heat_cap = nox_CpL(hybrid.tank_fluid_temperature_K); /* Get specific heat capacity of the liquid nitrous */

/* Calculate the heat removed from the liquid nitrous during its vaporisation */
deltaQ = vaporised_mass_old * Enth_of_vap;

/* temperature drop of the remaining liquid nitrous due to losing this heat */
deltaTemp = -(deltaQ / (hybrid.tank_liquid_mass * Spec_heat_cap));

hybrid.tank_fluid_temperature_K += deltaTemp; /* update fluid temperature */

/* reality checks */
if (hybrid.tank_fluid_temperature_K < (-90.0 + CENTIGRADE_TO_KELVIN))
{
  hybrid.tank_fluid_temperature_K = (-90.0 + CENTIGRADE_TO_KELVIN); /* lower limit */
  hybrid.hybrid_fault = 1;
}
else if (hybrid.tank_fluid_temperature_K > (36.0 + CENTIGRADE_TO_KELVIN))
{
  hybrid.tank_fluid_temperature_K = (36.0 + CENTIGRADE_TO_KELVIN); /* upper limit */
  hybrid.hybrid_fault = 2;
}

/* get current nitrous properties */
hybrid.tank_liquid_density = nox_Lrho(hybrid.tank_fluid_temperature_K);
hybrid.tank_vapour_density = nox_Vrho(hybrid.tank_fluid_temperature_K);
hybrid.tank_pressure_bar   = nox_vp(hybrid.tank_fluid_temperature_K);  /* vapour pressure, Bar abs */


Chamber_press_bar_abs = hybrid.chamber_pressure_bar; /* Bar Abs */
```

```
/* calculate injector pressure drop and mass flowrate */
mdot_tank_outflow = injector_model(hybrid.tank_pressure_bar, Chamber_press_bar_abs);



/* integrate mass flowrate using Addams second order integration formula */
/* (my preferred integration formulae, feel free to choose your own.)    */
/*     Xn=X(n-1) + DT/2 * ((3 * Xdot(n-1) - Xdot(n-2))                   */
/* O infront of a variable name means value from previous timestep (Old) */
delta_outflow_mass = 0.5 * delta_time * (3.0 * mdot_tank_outflow - Omdot_tank_outflow);

/* drain the tank based on flowrates only */
hybrid.tank_propellant_contents_mass -= delta_outflow_mass; /* update mass within tank for next iteration */

old_liquid_nox_mass -= delta_outflow_mass; /* update liquid mass within tank for next iteration */

/* now the additional effects of phase changes */

/* The following equation is applicable to the nitrous tank, containing saturated nitrous:     */
/* tank_volume = liquid_nox_mass / liquid_nox_density + gaseous_nox_mass / gaseous_nox_density */

/* Rearrage this equation to calculate current liquid_nox_mass */
bob = (1.0 / hybrid.tank_liquid_density) - (1.0 / hybrid.tank_vapour_density);

hybrid.tank_liquid_mass
    = (hybrid.tank_volume - (hybrid.tank_propellant_contents_mass / hybrid.tank_vapour_density)) / bob;

hybrid.tank_vapour_mass = hybrid.tank_propellant_contents_mass - hybrid.tank_liquid_mass;


/* update for next iteration */
bob = old_liquid_nox_mass - hybrid.tank_liquid_mass;
```

```
  /* Add a 1st-order time lag (of 0.15 seconds) to aid numerical  */
  /* stability (this models the finite time required for boiling) */
  tc = delta_time / 0.15;
  lagged_bob = tc * (bob - lagged_bob) + lagged_bob; // 1st-order lag

  vaporised_mass_old = lagged_bob;

  /* update tank contents for next iteration */
  old_liquid_nox_mass = hybrid.tank_liquid_mass;
}


/* Subroutine to initialise program variables */
/* Gets called only once, prior to firing     */
void initialise_hybrid_engine(void)
{
  hybrid.hybrid_fault = 0;
  hybrid.tank_vapour_mass = 0.0;
  mdot_tank_outflow = 0.0;


  /* set either initial nitrous vapour (tank) pressure */
  /* or initial nitrous temperature (deg Kelvin) */
  if (hybrid.press_or_temp == true)
    hybrid.tank_fluid_temperature_K = nox_on_press(hybrid.initial_tank_pressure); // set tank pressure
  else
  { /* set nitrous temperature */
    hybrid.tank_fluid_temperature_K = hybrid.initial_fluid_propellant_temp + CENTIGRADE_TO_KELVIN;
    hybrid.initial_tank_pressure    = nox_vp(hybrid.tank_fluid_temperature_K);
  }


  /* reality check */
  if (hybrid.tank_fluid_temperature_K > (36.0 + CENTIGRADE_TO_KELVIN))
  {
```

```
    hybrid.tank_fluid_temperature_K = 36.0 + CENTIGRADE_TO_KELVIN;
    hybrid.hybrid_fault = 2;
  }

  /* get initial nitrous properties */
  hybrid.tank_liquid_density = nox_Lrho(hybrid.tank_fluid_temperature_K);
  hybrid.tank_vapour_density = nox_Vrho(hybrid.tank_fluid_temperature_K);

  /* base the nitrous vapour volume on the tank percentage ullage (gas head-space) */
  hybrid.tank_vapour_volume = (hybrid.initial_ullage / 100.0) * hybrid.tank_volume;
  hybrid.tank_liquid_volume = hybrid.tank_volume - hybrid.tank_vapour_volume;

  hybrid.tank_liquid_mass = hybrid.tank_liquid_density * hybrid.tank_liquid_volume;
  hybrid.tank_vapour_mass = hybrid.tank_vapour_density * hybrid.tank_vapour_volume;

  hybrid.tank_propellant_contents_mass = hybrid.tank_liquid_mass + hybrid.tank_vapour_mass; /* total mass within tank */

  /* initialise values needed later */
  old_liquid_nox_mass = hybrid.tank_liquid_mass;
  old_vapour_nox_mass = hybrid.tank_vapour_mass;
  hybrid.initial_liquid_propellant_mass = hybrid.tank_liquid_mass;
  hybrid.initial_vapour_propellant_mass = hybrid.tank_vapour_mass;

  /* guessed initial value of amount of nitrous vaporised per iteration */
  /* in the nitrous tank blowdown model (actual value is not important) */
  vaporised_mass_old = 0.001;

  /* individual injector orifice total loss coefficent K2 */
  bob = PI * SQR((hybrid.orifice_diameter / 2.0)); /* orifice cross sectional area */

  hybrid.injector_loss_coefficient
      = (hybrid.orifice_k2_coefficient / (SQR((hybrid.orifice_number * bob)) ) ) * PASCALS_TO_BAR;
}
```

References:

Ref. 1: Dr Bruce P. Dunn
University of British Columbia and Dunn Engineering
Several articles on self pressurised peroxide rockets and experiments on propane tanks, as well as email communications with the author on the subject of numerical modelling of the tank emptying process; many thanks.

Ref.2: Engineering Sciences Data Unit (ESDU) sheet 91022,
Thermophysical properties of nitrous oxide.
Available in hardcopy from some U.K. University libraries, or accessible over the Web to students with an ATHENS password.

Ref.3: Space Propulsion Analysis and Design
by Ronald .W. Humble, Gary .N. Henry and Wiley J. Larson
McGraw Hill Space Technology Series ISBN 0-07-031320-2