Modelling the nitrous run tank emptying

Introduction

This paper describes mathematical models of the emptying of a tank of saturated nitrous oxide that empties purely due to its self pressure. This is sometimes called a ‘Vapak’ process (powered by vapour pressure alone).

Firstly, read our ‘The physics of nitrous oxide’ paper as it covers the processes we’ll be modelling herein.

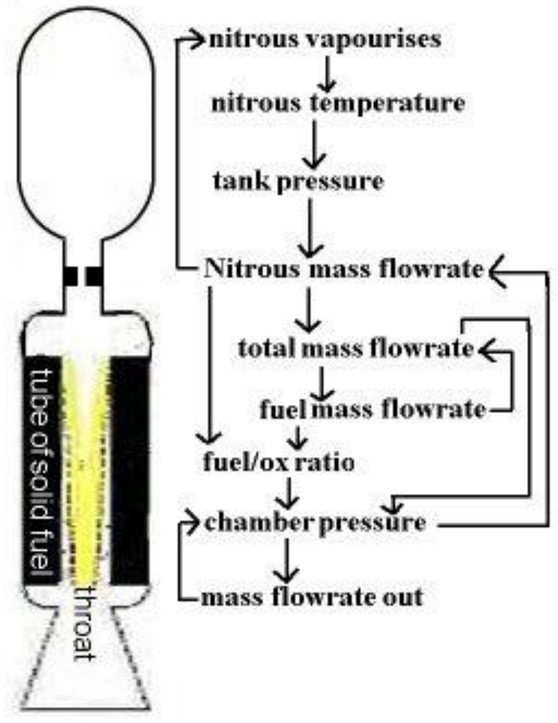
The mathematical model of the liquid phase emptying is based on a model of saturated propane emptying from a tank devised by Dr Bruce Dunn (Ref. 1). Aspirespace gratefully acknowledges the help we received from Dr Dunn in the preparation of our nitrous tank model.

The mathematical model of the vapour phase emptying is all our own work.

The nitrous hybrid as a system

The emptying process is iterative with time, and it’s strongly coupled to the combustion chamber pressure.
So the models of the tank emptying has to be coupled to a simulation of the combustion chamber and nozzle throat of the hybrid rocket motor that it is feeding:

The feedback loops of a nitrous hybrid emptying its liquid are as follows:



Liquid nitrous flows out of the tank causing a drop in the level of liquid nitrous. This causes an increase in the head space of nitrous vapour above the liquid.
The nitrous vapour pressure drops due to this expansion.
Some of the liquid nitrous then vapourises to try to raise the vapour pressure back up.
The energy required to vapourise the liquid comes from the liquid itself, and so its temperature drops.
This lower temperature lowers the tank pressure.

The flowrate of nitrous out of the tank depends on the difference between the tank and combustion chamber pressure.

The fuel mass flowrate eroded from the plastic fuel grain depends on the total flowrate of fuel plus nitrous oxidiser.

The combustion chamber pressure depends on the fuel to oxidiser ratio, the gain of mass in the chamber (fuel flowrate + ox flowrate – flowrate out the nozzle), and the nozzle throat area. But that’s another story: see section 7.5.2 of Ref. 3 for details.

1: The tank emptying liquid nitrous

When the nitrous vapour expands due to the level of the liquid dropping, the pressure drops due to this expansion.
We don't need to know what this pressure drop is to model the tank emptying, instead we estimate how much mass of nitrous liquid, m_v, has been vapourised to try and raise the pressure back up again to as it was.
This is an iterative process; we pick an arbitrary nonzero value for m_v to start with, and the program quickly converges on the actual value, and stays with it as it changes as the tank empties.

We calculate the heat removed (ΔQ) from the liquid nitrous during its vaporisation:
$\Delta Q = m_v H_v$ (equ. 1.1) where H_v is the enthalpy (latent heat) of vaporisation evaluated at the current nitrous temperature.

We then calculate the temperature drop of the remaining liquid nitrous (m_{liquid}) due to losing this heat:
$\Delta T = \dfrac{-\Delta Q}{m_{liquid} C_{liquid}}$ (equ. 1.2) where C_{liquid} is the Specific heat capacity of liquid nitrous at the current temperature.

We then subtract this temperature drop from the current liquid nitrous temperature to get a new lower liquid nitrous temperature.

The liquid density ρ_{liquid}, the vapour density ρ_{vapour}, and the vapour pressure (tank pressure) are now recalculated based on this lower temperature.

Using this new tank pressure and the current combustion chamber pressure, the mass flowrate of liquid nitrous out of the tank, $\dot{m}_{liquid}$, is now calculated:

Starting with Bernoulii's equation for the flow of nitrous from the injector manifold into the injector orifices:
$P_{manifold} + \frac{1}{2} \rho_{liquid} V_{manifold}^2 = P_{injector} + \frac{1}{2} \rho_{liquid} V_{injector}^2$ (equ. 1.3)

As the nitrous leaves the injector orifices, it breaks into droplets without changing pressure, so $P_{injector} =$ combustion chamber pressure.
The injector pressure drop can be checked to ensure that it is greater than 20% of the combustion chamber pressure for safety as advised in Ref. 3

Substituting for the velocity of the liquid from a rearrangement of the mass continuity equation:
$V = \dfrac{\dot{m}_{liquid}}{\rho_{liquid} A}$ (equ. 1.4) where $\dot{m}_{liquid}$ is the mass flowrate of liquid, and *A* is the cross-sectional area of the manifold or injector orifice.

gives:
$P_{injector} - P_{manifold} = \Delta P = \dfrac{\dot{m}_{liquid}^2}{2\rho_{liquid}} \left(\dfrac{K}{(N A_{injector})^2} - \dfrac{1}{A_{manifold}^2} \right)$ (equ. 1.5) where *N* is the number of orifices.

Note the inclusion of a loss coefficient *K*. This represents the loss of total pressure due to viscous losses/turbulence as the flow flows through the edges of the orifice.
As the static pressure of the nitrous liquid drops as it passes through the orifices, it vapourises. This means that what flows through the injector is a foam of liquid and bubbles, but mostly vapour, and so traditional tables of loss coefficients or discharge coefficients don't work for this mixed fluid. You have to tailor this *K* coefficient until the time taken to empty the tank matches your test results. We've found that a good starting value for *K* is 2.0 for nitrous.

Rearranging, and assuming that $A^2_{manifold}$ is much larger than the injector orifice area gives:

$$\dot{m}_{liquid} = \sqrt{\frac{2\rho_{liquid}\Delta P}{D_{loss}}} \quad \text{(equ. 1.6)} \qquad \text{where } D_{loss} = \frac{K}{\left(N A_{injector}\right)^2} \quad \text{(equ. 1.7)}$$

Having got the mass flowrate of liquid out of the tank we can now integrate to get the mass that has left the tank in this time iteration:

Total system mass m_{total} (liquid + vapour) has decreased by $\dot{m}_{liquid}\Delta t$ (equ. 1.8)

Liquid mass m_{liquid} has decreased by $\dot{m}_{liquid}\Delta t$ (equ. 1.9)

The resulting value for m_{liquid} is the mass of liquid that would be in the tank if the nitrous did not react to the expansion of the nitrous vapour and the ensuing drop in pressure. We'll designate this as m_{liquid_old}

But the nitrous *does* react, both to the increase in nitrous volume and also the drop in temperature.

The densities of the liquid and vapour are functions of temperature only.
The nitrous is constrained to fit into the volume of the tank, so is forced to adhere to a volume formula:

$$V_{vapour} + V_{liquid} = V_{tank} \quad \text{or,} \quad \frac{m_{liquid}}{\rho_{liquid}} + \frac{m_{vapour}}{\rho_{vapour}} = V_{\tan k} \quad \text{(equ. 1.10)}$$

where $m_{total} = m_{liquid} + m_{vapour}$ (equ. 1.11)

Rearranging:

$$m_{liquid} = \frac{\left(V_{\tan k} - \dfrac{m_{total}}{\rho_{vapour}}\right)}{\left(\dfrac{1}{\rho_{liquid}} - \dfrac{1}{\rho_{vapour}}\right)} \quad \text{(equ. 1.12)}$$

we'll designate this value as m_{liquid_new}

The discrepancy between this value and the previous value is the mass of nitrous that has been vapourised:

$$m_v = m_{liquid\_old} - m_{liquid\_new} \quad \text{(equ. 1.13)}$$

With this new value for m_v we can proceed to the next time iteration, and begin the calculation loop again.

NB
Bear in mind that this model is an approximation only, which uses simple integration routines. Occasionally, the model goes awry just as the last of the liquid nitrous is emptying. Add the following check to catch this, and use it to trigger engine burnout.

if ($m_{liquid\_new} > m_{liquid\_old}$) then trigger burnout

2: The vapour-only phase

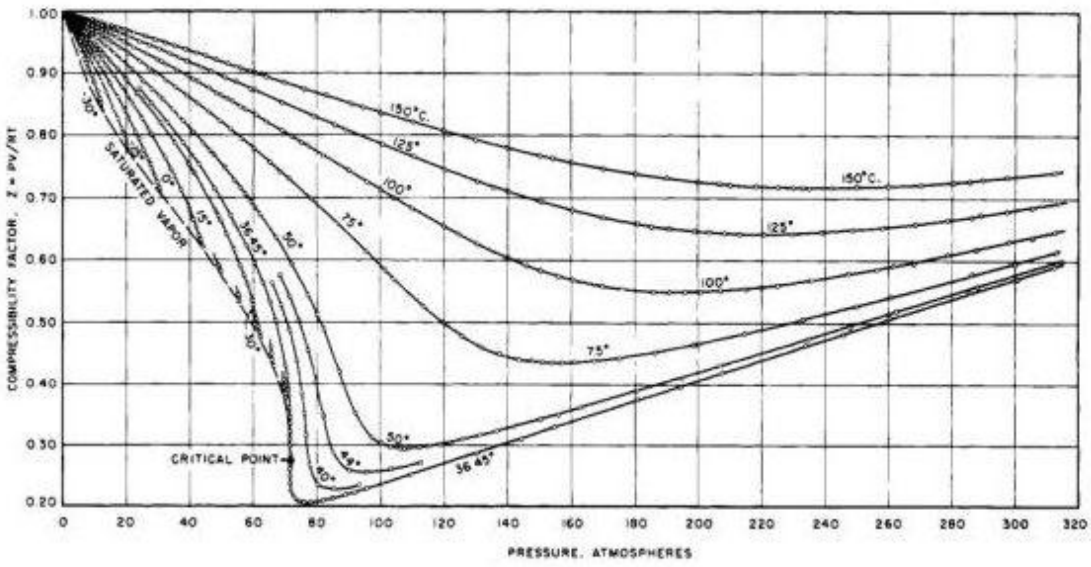
After all the liquid nitrous has run out of the run-tank, there will still be some vapour remaining. Even if you started with a tank completely full of liquid, some vapour will be created as the tank empties.
This vapour is dense enough to erode the hybrid fuel grain and so produce thrust, though it burns fuel-rich (too little oxidiser), and this 'vapour-only' phase doesn't last long.

From our hybrid firing data, we've learned a few surprising things about this 'vapour-only' phase of combustion:

1) It transpires that the pressure loss that occurs as the vapour flows through the injector orifices is identical to when the liquid was flowing through it (the head loss coefficient *K* is the same). This proves that the liquid vaporises completely to vapour inside the orifices, assuming that you use numerous small orifices as we do.
2) The vapour emptying out of the run-tank can be modelled as a standard 'isentropic' process. That means that very little energy is wasted (increase of entropy) during the emptying, and no heat is transferred from the tank walls to the vapour.
3) Therefore the vapour pressure and temperature drop rapidly as the tank empties and the vapour expands.
4) The vapour is not an 'ideal gas'. Intermolecular forces (the forces between the vapour molecules) are noticeably at work, so nitrous vapour expands differently to that of an ideal gas (you have to add a compressibility factor to the standard ideal gas equation).

With the above in mind, a simple mathematical model will simulate the tank emptying of nitrous vapour, which is now described:

Firstly, the 'real gas' aspect of the nitrous vapour has to be modelled: nitrous's compressibility factor (Z) has to be calculated. Its graph is:



Our sims show that the originally saturated nitrous vapour remains on the dotted saturated vapour line shown in this graph as it empties from the tank. We've approximated this vapour curve as a simple straight line running from the Critical point to the point (0,1) at zero pressure. Thus, the compressibility factor is just a linear function of tank vapour pressure.

Next, capture and store the initial tank values that occurred the instant the last of the liquid ran out of the tank exit: initial vapour temperature *Ti*, initial vapour mass *mi*, initial vapour pressure *Pi*, and initial vapour density *ρi*
Then work out the initial compressibility factor *Zi* using *Pi*.
Next, calculate the mass flowrate of nitrous vapour out of the tank using equations 1.6, 1.7, and 1.9, but working with vapour instead of liquid. The loss coefficient *K* in equation 1.7 remains the same.

Next, the vapour emptying can be modelled as an isentropic process. The inter-relationships between *P, T*, and *ρ* for an isentropic process are the standard isentropic equations:

$$\frac{T_2}{T_1} = \left(\frac{P_2}{P_1}\right)^{\frac{\gamma-1}{\gamma}} = \left(\frac{\rho_2}{\rho_1}\right)^{\gamma-1}$$

(equ. 2.1) where *γ* is the ratio of specific heats which is 1.3 for nitrous vapour. (Averaged over the subcritical temperature range of interest.)

We'll take time 1 as the initial value as the liquid just runs out, and time 2 as a point some time later on in the emptying process.

Starting with the gas equation for a real gas: $P = Z\rho RT$ (equ. 2.2) where R is the specific gas constant for nitrous.

Rearranging equation 2.2, and substituting $\rho = \frac{m}{V}$ (equ. 2.3) where *V* is volume, gives:

$$\frac{T_2}{T_1} = \frac{\frac{P_2}{Z_2 m_2}\left(\frac{V_{tank}}{R}\right)}{\frac{P_1}{Z_1 m_1}\left(\frac{V_{tank}}{R}\right)}$$

(equ. 2.4) where V_{tank} is the run-tank volume.

Rearranging and cancelling gives: $\frac{T_2}{T_1} = \frac{P_2}{P_1}\left(\frac{Z_1 m_1}{Z_2 m_2}\right)$

Using equations 2.1 and substituting temperature for pressure:

$$\frac{T_2}{T_1} = \left(\frac{T_2}{T_1}\right)^{\frac{\gamma}{\gamma-1}}\left(\frac{Z_1 m_1}{Z_2 m_2}\right)$$

Taking the temperatures over to the left-hand side:

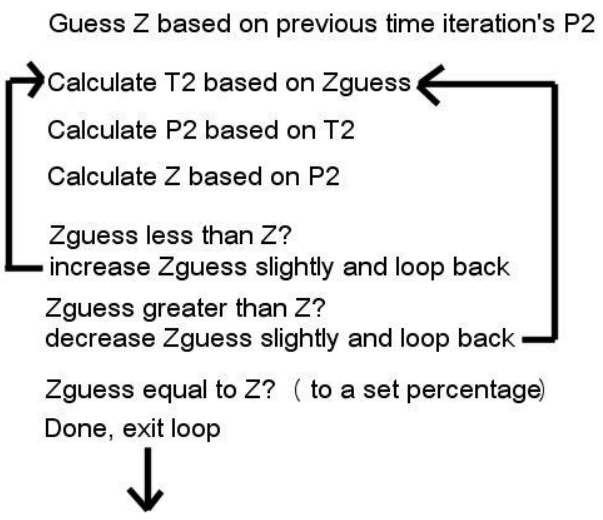
$$\left(\frac{T_2}{T_1}\right)^{\frac{(\gamma-1)-\gamma}{\gamma-1}} = \left(\frac{Z_1 m_1}{Z_2 m_2}\right) \quad \text{or:} \quad \left(\frac{T_2}{T_1}\right)^{\frac{-1}{\gamma-1}} = \left(\frac{Z_1 m_1}{Z_2 m_2}\right)$$

Giving: $$\left(\frac{T_2}{T_1}\right) = \left(\frac{Z_1 m_1}{Z_2 m_2}\right)^{\frac{\gamma-1}{-1}} = \left(\frac{Z_2 m_2}{Z_1 m_1}\right)^{\gamma-1} \qquad \text{(equ. 2.5)}$$

If we use the initial values we stored earlier as time 1, then equation 2.5 gives the temperature at time 2, when the new mass of vapour within the tank is m_2.

Now we can use equations 2.1 to calculate the new vapour pressure and density at time 2.

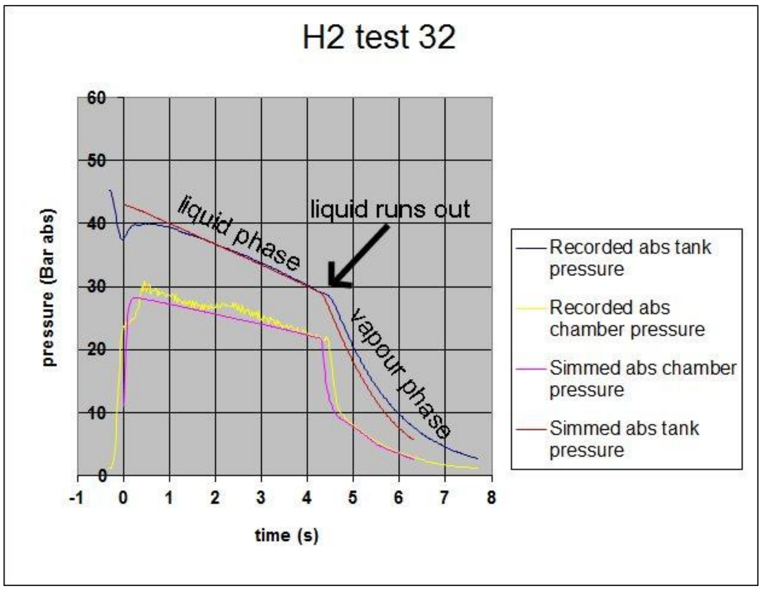
There is one problem though: we need to calculate Z_2 in equation 2.5, but this depends upon P_2, the vapour pressure at time 2, which we calculate *after* calculating equation 2.5
So we need to resort to a recursive loop:


Guess Z based on previous time iteration's P2
Calculate T2 based on Zguess
Calculate P2 based on T2
Calculate Z based on P2
Zguess less than Z?
increase Zguess slightly and loop back
Zguess greater than Z?
decrease Zguess slightly and loop back
Zguess equal to Z? (to a set percentage)
Done, exit loop

Now calculate ρ_2.

Then increment the sim time to a new time 2, and repeat equations 1.6, 1.7, and 1.9, and perform the new loop.

Results
When coupled to a sim of the combustion chamber, the results of the above tank emptying simulation models give a good match with experimental firing data:



Software

Some software extracts of the above models coded in the C++ language are listed below. These comprise a suite of subroutines that give the nitrous properties as functions of temperature etc called nitrous_oxide.cpp and its header file nitrous_oxide.h

Also listed are subroutines coding the tank emptying models described above.

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) */
const float ZCrit   =   0.28f; /* critical compressibility factor */
const float gamma = 1.3;   /* average over subcritical range */

/* ranges of function validity */
const float lower_temp_limit = -90.0 + CENTIGRADE_TO_KELVIN;
const float upper_temp_limit =  36.4 + CENTIGRADE_TO_KELVIN;

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 liquid 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);
}


/* nitrous 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) Rick Newlands, 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 */
/* Empty tank of liquid nitrous */
void Nitrous_tank_liquid(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;

  // Check for model fault at nearly zero liquid oxidiser mass
  // If this occurs, use the fault flag to trigger burnout
  if (hybrid.tank_liquid_mass > old_liquid_nox_mass)
    hybrid.hybrid_fault = 9;

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

```
/* Linear interpolation routine, with limiters added */
/* incase x isn't within the range range x1 to x2    */
/* Limits updated to allow descending x values
  x  = input value
  x1 = MINIMUM BOUNDS ( minimum x )
  y1 = MINIMUM VALUE  ( output value at MINIMUM BOUNDS )
  x2 = MAXIMUM BOUNDS ( maximum x )
  y2 = MAXIMUM VALUE  ( output value at MAXIMUM BOUNDS )
*/
double LinearInterpolate(double x, double x1, double y1, double x2, double y2)
{
  // This procedure extrapolates the y value for the x position
  // on a line defined by x1,y1; x2,y2

  double c, m, y;  // the constants to find in y=mx+b

  if ( (x1 < x2) && ((x <= x1) || (x >= x2)) ) // ascending x values
  {
    if (x <= x1) return y1; else return y2;
  }
  else if ( (x1 > x2) && ((x >= x1) || (x <= x2)) ) // descending x values
  {
    if (x >= x1) return y1; else return y2;
  }
  else
  {
    m = (y2 - y1) / (x2 - x1);   // calculate the gradient m
    c = y1 - m * x1;          // calculate the y-intercept c
    y = m * x + c;         // the final calculation
    return(y);
  }
}

/* Compressibility factor of vapour (subcritical) on the saturation line */
double compressibility_factor(double P_Bar_abs, double pCrit, double ZCrit)
{
  double Z;

  Z = LinearInterpolate(P_Bar_abs, 0.0, 1.0, pCrit, ZCrit);

  return(Z);
}


/* subroutine to model the tank emptying of vapour only */
/* Isentropic vapour-only blowdown model */
void subcritical_tank_no_liquid(
                                double *vapour_temperature_K,
                                double *vapour_mass,
                                double *vapour_density,
                                double *tank_pressure_bar,
                                double chamber_pressure_bar,
                                double injector_loss_coefficient,
                                double *mdot_tank_outflow, double old_mdot_tank_outflow,
                                double *tank_contents_mass,
                                unsigned char *fault)
{
  static int Aim, OldAim;
  static bool first = true;

  double bob;
  double current_Z, current_Z_guess;
  double delta_outflow_mass;
  double step;

  static double initial_vapour_density, initial_vapour_mass, initial_vapour_pressure_bar;
  static double initial_vapour_temp_K, initial_Z;


  // capture initial conditions
  if (first == true)
  {
    initial_vapour_temp_K       = *vapour_temperature_K;
    initial_vapour_mass         = *vapour_mass;
    initial_vapour_pressure_bar = *tank_pressure_bar;
    initial_vapour_density      = *vapour_density;

    initial_Z = compressibility_factor(initial_vapour_pressure_bar, nox_pCrit, nox_ZCrit);

    old_mdot_tank_outflow = 0.0; // reset

    first = false;
  }


  // calculate injector pressure drop and mass flowrate
  *mdot_tank_outflow = injector_model(*tank_pressure_bar, chamber_pressure_bar,
                                      1.0, *vapour_density,
                                      injector_loss_coefficient,
                                      false,
                                      &*fault);

  /* integrate mass flowrate using Addams second order integration formula */
  /*     Xn=X(n-1) + DT/2 * ((3 * Xdot(n-1) - Xdot(n-2))                    */
  delta_outflow_mass
        = 0.5 * delta_time * (3.0 * *mdot_tank_outflow - old_mdot_tank_outflow);

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


  // drain off vapour
  *vapour_mass -= delta_outflow_mass; // update vapour mass within tank for next iteration


  // initial guess
  current_Z_guess = compressibility_factor(*tank_pressure_bar, nox_pCrit, nox_ZCrit);

  step = 1.0 / 0.9;    // initial step size
  OldAim = 2; Aim = 0; // flags used below to home-in

  // recursive loop to get correct compressibility factor
  do
  {
    // use isentropic relationships
    bob = nox_gamma - 1.0;
```

```
    *vapour_temperature_K = initial_vapour_temp_K
              * pow(((*vapour_mass * current_Z_guess) / (initial_vapour_mass * initial_Z)), bob);

    bob = nox_gamma / (nox_gamma - 1.0);

    *tank_pressure_bar = initial_vapour_pressure_bar
                          * pow((*vapour_temperature_K / initial_vapour_temp_K), bob);

    current_Z = compressibility_factor(*tank_pressure_bar, nox_pCrit, nox_ZCrit);

    OldAim = Aim;

    if (current_Z_guess < current_Z)
    {
      current_Z_guess *= step;
      Aim = 1;
    }
    else
    {
      current_Z_guess /= step;
      Aim = -1;
    }

    /* check for overshoot of target, and if so, reduce step nearer to 1.0 */
    if (Aim == -OldAim)
      step = sqrt(step);

    /* leave loop upon convergence to required accuracy */
  }
  while ( ((current_Z_guess / current_Z) > 1.000001) ||
        ((current_Z_guess / current_Z) < (1.0 / 1.000001)) );

  bob = 1.0 / (nox_gamma - 1.0);

  *vapour_density = initial_vapour_density
                     * pow((*vapour_temperature_K / initial_vapour_temp_K), bob);
}
```

```
/* Subroutine to initialise main 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

Ref. 4: Rocket Propulsion Elements 7th edition
By Sutton and Biblarz