J. Feicht Feb 2019

Rev. kernal bt7_LMA_4.8

jfeicht@ligo.caltech.edu

This *Mathematica ver. 11.3* notebook generates the beamtube pressure profile using btwaterleak methodology of particle diffusion. The pressure distribution is the solution to the set of n-coupled differential equations where “n” represents the number of beamtube sections. The surface model is the Dubinin-Radushkevich isotherm.

Options can include the pump arraignments, leaks, temperature profile, etc. Not all of this functionality is included in this version.

Important note # 1. Any data input or changes you make to defined values should use floating point notation, for example use “2.0”, or “2.” instead of symbolic notation “2”. FP notation makes the code run faster because *Mathematica* interprets this as a number, not a symbol or exact value. Symbols can also cause issues with NDSolve, the numerical ODE solver.

Important note # 2. It is always a good idea to quit and restart the kernel after changing variables. *Mathematica* has a very long memory, so to reduce your frustration level and reset it after every run.

Important note # 3. Deleting all code output before saving will *greatly* reduce file size. Always do this!

Clear variables and cache. Remember the warning about resetting the kernal.

In[ ]:=
```
ClearAll ; (*Always quit and restart the kernel,
helps prevents strange behavior and crashes from happening*)
```

```
In[•]:= ClearSystemCache
Out[•]= ClearSystemCache
```

Define tube geometry. Note LIGOs 4 km BT's were baked 150 C in 2 km sections for ~ 1 month

```
In[•]:= Tubelength = 100.0 * 100. (*100 meter long beamtube converted to centimeter*)
Out[•]= 10 000.

In[•]:= Tubediameter = 1.0 * 100 (*nominal 1 meter diameter beamtube converted to centimeter*)
Out[•]= 100.

In[•]:= Tubearea = π Tubediameter^2 / 4.; (*units are centimeter^2*)
```

Define pump speeds.

```
In[•]:= Nopump = 0.0 (*speed cc/sec*)
         (*use this or code in 0.0 when temporary pumps are removed*);

In[•]:= IonpumpS = 500. * 1000. (*500 liter/sec ion pump speed converted to cc/sec*);

In[•]:= MainTurbospeed = 1900. * 1000. (*speed of main turbos are 1900 l/s, converted to cc/sec*);

In[•]:= TempCryopumpS = 1000. * 6.5 × 10^3 ;
        (* These are the temporary cryopumps installed during the inital BT bakeout,
        the value is their water speed in cc/sec.*)

In[•]:= LgIonSpeed = 2500 * 1000.; (*these are the large 2500 l/s ion pumps,
        speed converted to cc/sec*)

In[•]:= Turbospeed = 200. * 1000.
         (*speed of typical DN100 flange turbos are 200 l/s, converted to cc/sec*);
```

Define the number of ports (i.e. the number of pumps). The code evenly distributes the pumps across the length of beamtube you set. LIGO has 17 ports equally spaced over 4 km.

```
In[•]:= Numberofports = 2 (*need to have at least 2 ports,
         one on each end, this value is also the number of pumps. INTEGER*)
Out[•]= 2

In[•]:= Portpitch = Tubelength / (Numberofports - 1)  (*centimeter*)
Out[•]= 10 000.
```

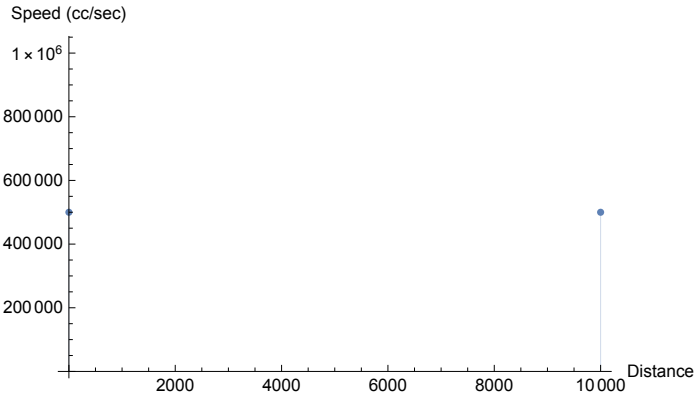
Automatically populate the pump speeds and locations (comment this section out if you want to directly input pump speeds).

```
In[•]:= Do[Port[i] = {(i - 1) * Portpitch,
        IonpumpS}, {i, 1, Numberofports}]; (*set pump vector and speeds*)

In[•]:= Do[Print[Port[i]], {i, 1, Numberofports}];
        (*output is a vector reported as the distance from "one end", then "pump speed" *)
        {0., 500 000.}
        {10 000., 500 000.}
```

Plot pump speeds (cc/sec) along beamtube.

```
In[•]:= portspeedtable = Table[Port[j], {j, 1, Numberofports}];
        (*first need to make a table of the indexed ordered pairs*)

In[•]:= ListPlot[portspeedtable, {Joined → False},
         AxesLabel → {"Distance", "Speed (cc/sec)"}, Filling → Bottom]
```

Out[•]=



Define starting pressure, temperature and gas amu.

```
In[•]:= startpress = 1.0 (*Starting pressure in Torr,
         note this is a molecular diffusion flow code,
         but btwater starts at 20 Torr. Should we stay out of viscous flow regime?*);

In[•]:= amu = 18. (*water*);

In[•]:= Tkelvin = 300. (*Kelvin*);
```

Define physical constants, conversion factors, variables, etc., see btwater3da for additional information.

```
alpha = 0.5 ; (*accomodation coefficient, dimensionless*)
rp = 0.7 ;(*potential height fraction, dimmensionless*)
sigma0 = 150. ; (*number of monolayers at start. Note this variable has changed
 definition from earlier versions of wb, waterbake, btwater, so be careful*)
Tao = 1.0 × 10^-12 ; (*molecule oscillation period in seconds*)
tresmin = 1.0 × 10^-6 ; (*min residence time in seconds*)
tresmax = 1.0 × 10^10 ; (*max residence time in seconds*)
emtmin = Log[tresmin / Tao]; (*log ratio*)
emtmax = Log[tresmax / Tao];
vt296 = 5.263 × 10^4 Sqrt[18.0 / amu]; (*thermal velocity at 296 Kelvin,
in cm/sec. Note multiply by Sqrt[Tkelvin/296] to adjust for temperature. Is WRT nitrogen*)
particlescm3 = 296. / Tkelvin 3.0 × 10^16 Ptorr ; (*density*)
particlescm2 = 1.0 × 10^15 * sigma0 ; (*initial surface loading;
monolayers*particles per monolayer*)

tdr0 = 1.0 × 10^4 (*the DR peak energy in Kelvin*);
```

```
pipeconductance = 3.81 * Tubediameter^3 / Tubelength (Tkelvin / amu)^1/2
  (*handbook formula molec flow conductance for a long round pipe in liters/second,
  used for comparison, setting system time constant*)
```

```
1555.43
```

```
timeconstant = IntegerPart[(Tubearea * Tubelength) * (1. / 1000.) / pipeconductance] (* cm^3 * liter/10^3 cm^3 / liter/sec *)
  (*get an estimate of the system time constant volume/(liter/sec), used this to set how long to run
   DE solver. Typically 1 or 2 time constants is sufficient if using constant ajj*)
```

```
50
```

Set how many sections to break the calculation (beamtube) into, normally use 100 sections, higher values cost solve time.

```
Sections = 100(* this is the INTEGER number of calculation sections that the beamtube
  will be broken into, i.e. the number of coupled differential equations to solve,
 affects pump placement calc, etc. 100 is a reasonable number,
 high numbers increase computation time*) ;
```

Establish which tube sections have pumps, then map the pump location & speed to each tube section.

```
(*You may ask why we need this section? The reason is because the number of sections
 is a user defined variable. Although the position of the pump stays fixed,
the particular section where the placement of pumps,
leaks etc. can change. This piece of code puts the pumps in their correct section*)

Seclength = Tubelength / Sections (*length of each "calculation" section*)
```

```
100.
```

```
Sections (*how many sections were requested*)
```

```
100
```

```
sectionrange[1] = {1, Interval[{0, Seclength}]};(*defines the first section*)
```

```
sectionrange[Sections] =
  {Sections, Interval[{(Tubelength - Seclength + .0001), Tubelength}]};
 (*defines the last section*)
```

```
Do[sectionrange[i] = {i, Interval[{((i - 1) * Seclength + .0001), (i * Seclength)}]},
  {i, 2, Sections - 1}] ;(*Defines the intermediate sections. The 0.0001 is a
 (poor) way to stop MemberIntervalQ from duplicating pumps by not allowing
 succesive intervals to touch. I've seen a switch in the code that fixes
 this but I havent (re)found it yet, so using this lame approach for now*)
```

```
Table[sectionrange[i], {i, 1, Sections}] ;(*take a look at the sections and their range*)
```

```
Do[findport[i, n] = Boole[IntervalMemberQ[sectionrange[i][[2]], Port[n][[1]]]],
 {n, 1, Numberofports}, {i, 1, Sections}]
 (* find the section (i) that has the pump port(n)*)
```

```
Do[tablel[j] = Table[findport[i, j], {i, 1, Sections}], {j, 1, Numberofports}]
(*intentionally misspelled*)
```

Check the pump (port) distribution for overlaps. Should return “true” if no overlaps.

```
portboole = Sum[tablel[i], {i, 1, Numberofports}] ;
(*make a boolean table of which tube section has the port*)
```

```
Total[portboole] == Numberofports
 (*this should return TRUE if the number of ports and the boolean sum are the
  same. If FALSE there is a section overlap and a port is being counted twice*)
```

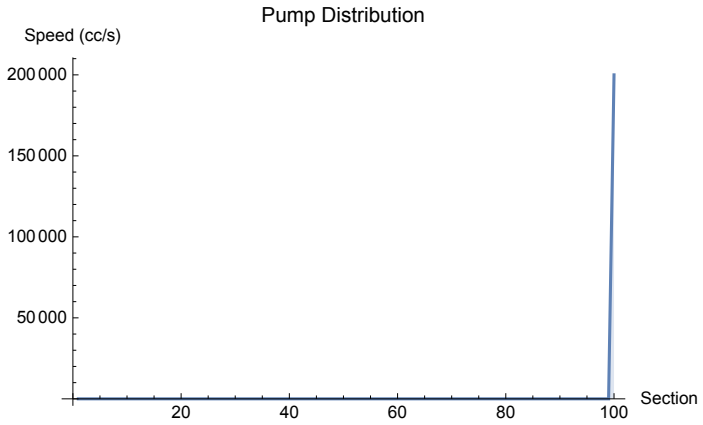
```
True
```

Map the pump speeds to each port. This section includes examples of how to

override or manually introduce values.

```
In[ ]:= Speedy[x_] := Nopump * x  (*construct a mapping function that multiplies by pump speed*)

        Do[g = MapAt[Speedy, portboole, ;;], {n, 1, Sections}];
        (*map the speed over the table, recall that ;; is shorthand for Span*)

In[ ]:= Do[speed[i] = g[[i]], {i, 1, Sections}] (*fyi [[i]] is a compact notation for Part,
        it picks out that element in the list. We need these to be individual
         values so they can go into the ODE by section. Speed should be in cc/sec*)

        (*>>>>>>>>>> the following will overwrite any earlier speed
          definitions so make a choice how you want to do this <<<<<<<<<*)

In[ ]:= speed[101] (*check if working correctly, should return a null*)

Out[ ]= speed[101]

        speed[100] = Turbospeed(*this is how to override the mapped speed for a particular port,
         in this example port 100 would be set to 200 l/s*)

Out[ ]= 200 000.

In[ ]:= (*speed[Sections]=500.*1000.*)(*another example,
        this is how to override the mapped speed of port 100 to 500 l/s*)

In[ ]:= speeds = Table[speed[i], {i, 1
            , Sections}] ;(*allows a quick look to be sure it's working*)

In[ ]:= ListPlot[speeds, Filling → Bottom, Joined → True, PlotLabel → "Pump Distribution",
         AxesLabel → {"Section", "Speed (cc/s)"}] (*speed plot versus section number in cc/sec*)
```

Out[ ]=



Setup beamtube leak(s). This section can be developed further as required. A similar section with programmed temperature can also be developed, is not included in this version of the program.

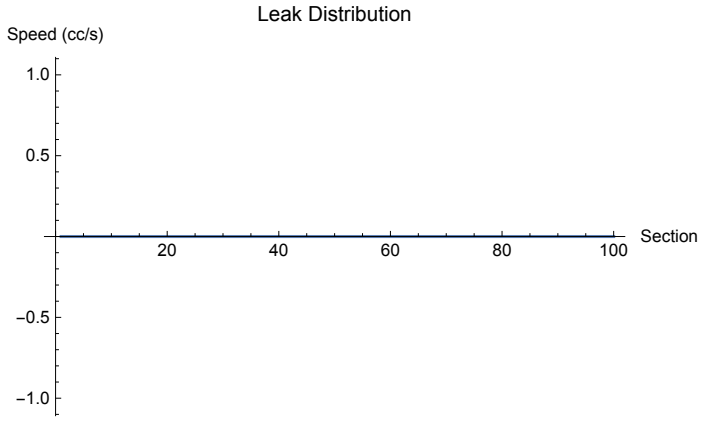
```
Do[qlk[i] = 0.0, {i, 1, Sections}] (*sets the leak rate in each
 section. Note that qlk is a variable in the differential equations,
initializes the solver, so do not remove this loop even if there are no
 leaks as the varible will not be declared and solver will blow up*)
```

In[ ]:=
```
(*qlk[20]= 0.0*1.0 10^-6 (* this shows how to manually introduce a leak. In
 this example a value of 1.0 x 10^-6 cc/sec is intoduced in section 20
 (change leading 0 to 1 to turn leak ON or OFF or surround by (* xxx *)). This
 can be automated in time, rate, etc. if desired*)*)
```

In[ ]:=
```
Leaks = Table[qlk[i], {i, 1, Sections}];
```

In[ ]:=
```
ListPlot[Leaks, Joined → True, PlotLabel → "Leak Distribution",
 AxesLabel → {"Section", "Speed (cc/s)"}]
(*makes a leak rate plot versus section number, units are cc/sec*)
```

Out[ ]=



Set the number of energy bins and set the initial adsorption site occupation probability to 1 (i.e. all sites are occupied).

In[ ]:=
```
nsites = 1024 (*set 1024 adsorption sites in the distribution, INTEGER*);
```

In[ ]:=
```
Do[ap[n, k] = 1.0, {n, 1, Sections}, {k, 1, nsites}] ;
(*ap is the probablility that a site is occupied. Setting to 1 occupies all
  sites. This is the initial condition. Note that 1≤n≤1Sections, 1≤k≤1024 sites*)
(*this section may be superfluous, gets overwritten when nsnin
 and nsmax are defined and site range is reset*)
```

Set time step size

```
In[ ]:= timestep = 60. (*seconds*) (*LIGO beamtube has low conductance
          and large volume so a very long vol/speed time constant. See DCC docs,
        particularly time constant associated with leak testing. Also see O'Hanlon
         for a  good discussion on leak testing sensitivity vs pump speed,
        time constant, pump arraingments, etc.*) (*timestep is 60 seconds in btwater,
        reevaluate for high conductance systems*);
```

Adsorption surface sites declaration, set range.

```
In[ ]:= nsmin = IntegerPart[ nsites * Tkelvin * emtmin / 3.0 * tdr0 ]
```

```
Out[ ]= 141
```

```
In[ ]:= If[nsmin < 1, nsmin = 1];
```

```
In[ ]:= nsmin (*check the value of nsmin*)
```

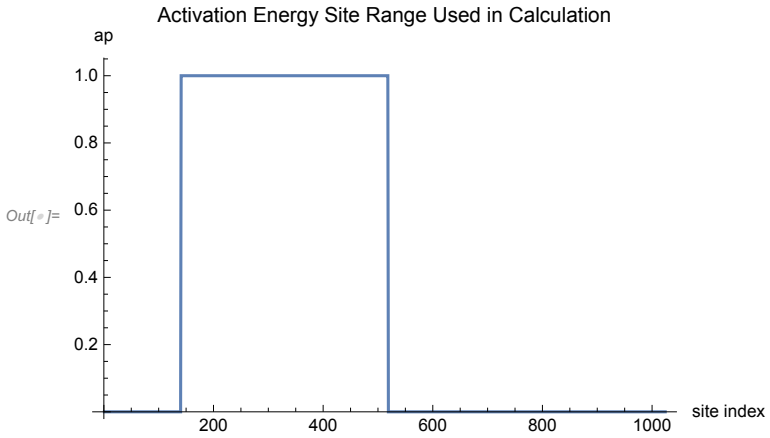
```
Out[ ]= 141
```

```
In[ ]:= nsmax = IntegerPart[ nsites * Tkelvin * emtmax / 3.0 * tdr0 ]
        If[nsmax > nsites, nsmax = nsites];
```

```
Out[ ]= 518
```

```
In[ ]:= Do[ap[n, k] = 0.0, {n, 1, Sections}, {k, nsmin, 1}]; (*deletes negative sites*)
        Do[ap[n, k] = 0.0, {n, 1, Sections}, {k, nsmax, nsites}]; (*deletes sites > nsmax*)
        Do[ap[n, k] = 0.0, {n, 1, Sections}, {k, 1, nsmin}]; (*1 < deletes sites < nsmin*)
        (*this may be an error, need to discuss with Rai*)
        Do[ap[n, k] = 1.0, {n, 1, Sections}, {k, nsmin, nsmax}];
        (*sets all "useful" sites in range nsmin to nsmax =1,
        is temperature dependent. BTW, I don't think the range distribution
          trap was done correctly in btwater >>>> this section needs more review *)
```

```
In[•]:= ListPlot[Table[ap[10, k], {k, 1, nsites}], Joined → True,
         PlotLabel → "Activation Energy Site Range Used in Calculation",
         AxesLabel → {"site index", "ap"}]
```



Out[•]=

Set activation energies (Kelvin), normalize and plot the distribution weight function. Note long tail on distribution. Check normalization by integrating, s/b =1.

```
In[•]:= deltat = 3.0 tdr0/nsites ;(*D-R energy site granularity ~30 K  *)
```

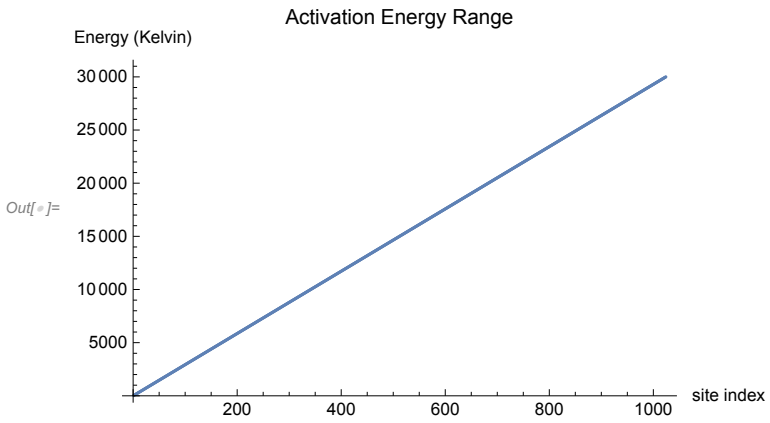
```
In[•]:= tdr0^2 (*just here for checking value, remove later*)
```

```
Out[•]= 1. × 10^8
```

```
In[•]:= sum = 0.0; (*initialize*);
```

```
In[•]:= Do[at[k] = k * deltat, {k, 1, nsites}];
        (*sets the activation temperature at[k] of site k, units are Kelvin*)
```

```
In[•]:= ListPlot[Table[{k, at[k]}, {k, 1, nsites}],
         PlotLabel → "Activation Energy Range", AxesLabel → {"site index", "Energy (Kelvin)"}]
```

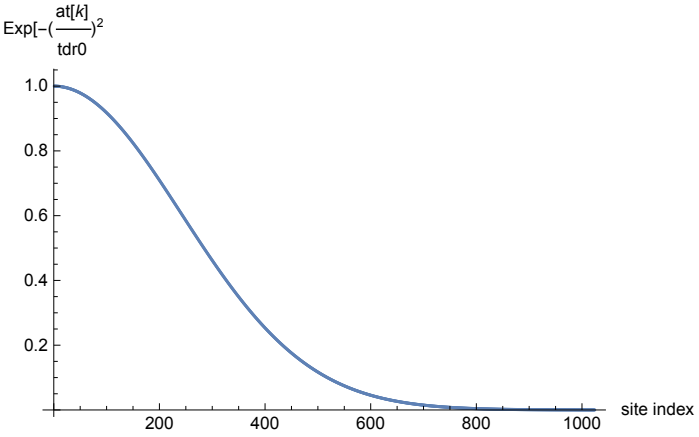


Out[•]=

In[ ]:=
```
Do[w[k] = 2.0 * at[k] * deltat/tdr0^2 Exp[-(at[k]/tdr0)^2], {k, 1, nsites}]
(*see Dubinin-Radushkevich isotherm*)
```

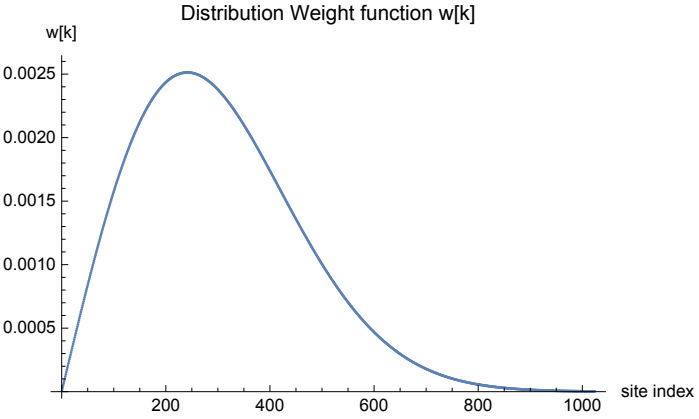
In[ ]:=
```
ListPlot[Table[{k, Exp[-(at[k]/tdr0)^2]}, {k, 1, nsites}],
 AxesLabel → {"site index", "Exp[-(at[k]/tdr0)^2"}]
(*have a quick look at the Exp[-(at[k]/tdr0)^2 term*)
```

Out[ ]=



In[ ]:=
```
ListPlot[Table[{k, w[k]}, {k, 1, nsites}], PlotLabel → "Distribution Weight function w[k]",
 AxesLabel → {"site index", "w[k]"}] (*plot w[k]*)
```

Out[ ]=



In[ ]:=
```
Do[sum = sum + w[k], {k, 1, nsites}]
```

In[ ]:=
```
Do[w[k] = w[k]/sum, {k, 1, nsites}] ;
(*normalize the weight function. w[k] is dimensionless*)
```

In[ ]:=
```
energies = Interpolation[Table[{w[k]}, {k, 1, nsites}]];
(*construct a interpolating function of w[k]*)
```

In[ ]:=
```
Integrate[energies[k], {k, 1, nsites}]
(*integrate the normalized w[k} to check that the area = 1*)
```

Out[ ]=
```
0.999992
```

Set the initial adsorption site occupation probability to 1 (i.e. all sites are occupied).

```
In[ ]:= Do[ap[n, k] = 1.0, {n, 1, Sections}, {k, 1, nsites}] ;
        (*ap is the probablility that a site is occupied. Setting to 1 occupies all
          sites. This is the initial condition. Note that 1≤n≤1Sections, 1≤k≤1024 sites*)
```

Define the p[t] differential equation coefficients.

```
In[ ]:= v = vt296 * Sqrt[Tkelvin / 296.] ;(*molecular speed adjustment for temperature,
        wrt N2, units are cm/sec*)

In[ ]:= aa = (2. * v * .5 Tubediameter) / (3. Seclength^2) ;(* units are cm/sec * cm/cm^2 = 1/sec *)
        bb = 4. / Tubediameter ;(* units are 1/cm *)
        dd = 1.0 / (π (.5 Tubediameter)^2 Seclength) ; (* units are 1/cm^3 *)

In[ ]:= aa (*review the values*)
        bb
        dd

Out[ ]= 176.615

Out[ ]= 0.04

Out[ ]= 1.27324 × 10^-6
```

Set initial pressure in each section. Startpress defined previously.

```
In[ ]:= Do[ystart[n] = startpress, {n, 1, Sections}]
        (*sets initial pressure in all sections, units are Torr*)
```

Determine the outgassing rate in each section. This rate then gets pushed to the differential equation solver to compute the system pressure.
Uses updates in ystart[n] to calculate desorption probability. Follows methodology of probev subroutine in btwaterlk3da .

```
In[•]:= probev[] := Module[{sumb, suma, rt, tg, tt, temit, tads, tau, pequil, ax, j, n},
    (*timestep=ts, which is similar to btwater*)
    Do[sumb = 0;
     suma = 0;
     Do[sumb = sumb + ap[n, k] * w[k] (*w[k] is the normalization factor*);
      rt = at[k] / Tkelvin (*rt is the ratio of at[k],
      the activation temperature of the site, to the surface abs temp, dimensionless*) ;
      tg = rt (1.0 - rp);
      (*rp is the repulsive potential ratio, set to 0.7 currently, dimensionless*)
      tt = (1.0 + tg) Exp[-tg];
      temit = Tao * Exp[rt] (*this is the emission time, seconds*);
      tads = 4.0 * sigma0 / (30.0 * alpha * ystart[n] * v * tt) (*tads is the (re)adsorption time,
      with sigma0 the inital monolayers and alpha is the accom. coeff.,
      v=molec speed, ystart is the pressure in each section, dims are seconds *);
      tau = temit * tads / (temit + tads); (*dimensions are seconds*)
      pequil = temit / (temit + tads); (*dimensions are seconds*)
      If[tads ≤ 0.0 || tads > 1.0 × 10^12 , {tau = temit, pequil = 0.0}];
      ax = Exp[-timestep / tau];
      ap[n, k] = ap[n, k] * ax + pequil * (1.0 - ax);
      If[ap[n, k] > 1.0, ap[n, k] = 1.0];
      (*ap is the probability that a site is occupied*)
      If[ap[n, k] < 0.0, ap[n, k] = 0.0];
      suma = suma + ap[n, k] * w[k], {k, nsmin, nsmax}];
     ajj_n = (sumb - suma) * sigma0 / (30.0 * timestep) (*outgassing rate/section,
     converts monolayer/sec to torr-cc/sec cm^2*), {n, 1, Sections}];
    Return[{ax, suma , sumb, pequil, rt, tg, temit, tads, tau, timestep, tau, ajj_n}]
   ];
```

Determine the pressure distribution in the beamtube by using the most current ajj (outgassing rates) and the last pressure distribution. These values populate the differential equation solver which computes the new system pressure/section, i.e. the new ystart[n]. The iteration rate is timestep. The new ystart[n] are evaluated

by the probev module and the outgassing rates are updated, etc. The calculation continues until the desired run time is reached.

In[ ]:=
```
Do[ajj_n = 0.0, {n, 1, Sections}] ; (*initialize ajj to zero/section,
as ajj will be evaluated by probev module. Keep in mind the
 comment in btwater3da note says this is  inconsistent with later revs*)
```

In[•]:=

```
calcpressdist[time_] := Module[{currenttime = time}, probev[];

  sectionrates = Table[ajj_n, {n, 1, Sections}] ;
  (* these are the new ajj that get fed back into the calculation*)

  diffeqns = Table[{p_(i+1)'[t] ==
      aa (p_i[t] + p_(i+2)[t] - 2.0 p_(i+1)[t]) + bb ajj_(i+1) + dd (qlk[i + 1] - speed[i + 1] * p_(i+1)[t]),
     p_(i+1)[0] == ystart[i + 1]}, {i, 1, Sections - 2}] ;
  (*this generates the ODE's for sections n+1 to n-1 of the n beam tube sections*)

  Do [If[p_k < 0.0 || p_(k+1) < 0.0, p_k'[t] = 0], {k, 3, Sections - 3}];

  firstsection = {p_1'[t] ==
     aa (p_2[t] - p_1[t]) + bb ajj_1 + dd (qlk[1] - speed[1] * p_1[t]), p_1[0] == ystart[1]} ;
  (*this generates the ODE for section 1 of the n beam tube sections*)

  If[p_1 < 0.0 || p_2 < 0.0, p_1'[t] = 0]; (*1st section*)

  lastsection = {p_Sections'[t] == aa (p_(Sections-1)[t] - p_Sections[t]) + bb ajj_Sections +
      dd (qlk[Sections] - speed[Sections] * p_Sections[t]), p_Sections[0] == ystart[Sections]} ;
  (*this generates the ODE for the last section of the n beam tube sections*)

  If[p_(Sections-1) < 0.0 || p_Sections < 0.0, p_Sections'[t] = 0]; (*last section*)

temp = Prepend[diffeqns, firstsection]; (* add the first sections to ODE list*)

  diffeqns = Append[temp, lastsection];
  (* add a the last sections, this completes the ODE list*)

  vars2 = Table[p_j[t], {j, 1, Sections}]; (*define the variables*)

  btw = NDSolve[diffeqns, vars2, {t, 1, timestep}];
  (*solve the equation, code picks optimum solver*)

  secpresvstime = Table[vars2 /. First[btw], {t, 1, timestep}];
 (*these are the solutions, this can be a really big file,
  it contains a value for every time set from 1 to timestep,
  for each section (typically 100 sections)*)
  Do[ystart[j] = Last[secpresvstime][[j]], {j, 1, Sections}];
  (*these are the new pressure values for use in the iteration*)

  data[currenttime] = Table[ystart[j], {j, 1, Sections}]
  (*these data are the pressure in each section for each timestep*)

 ]
```

Set calculation loop time or number of cycle iterations

```
In[ ]:= SetSystemOptions["CheckMachineUnderflow" → False]
        (*may be perilous to do this but I'm tired of small numbers warnings. Solution is
          to rewrite this code using arbitrary precision numbers rather than machine
          precision numbers.  NOTE CheckMachineUnderflow only works with Mathematica revison
          11.3 or higher. Set → True to see warnings or → False to suppress warnings*)
Out[ ]= CheckMachineUnderflow → False

In[ ]:= tmax = 100 000
Out[ ]= 100 000

In[ ]:= Do[calcpressdist[time], {time, 1, tmax}] (*set time range,
        call the solve pressure distribution Module*)
```

Take a look at ODE that is being solved

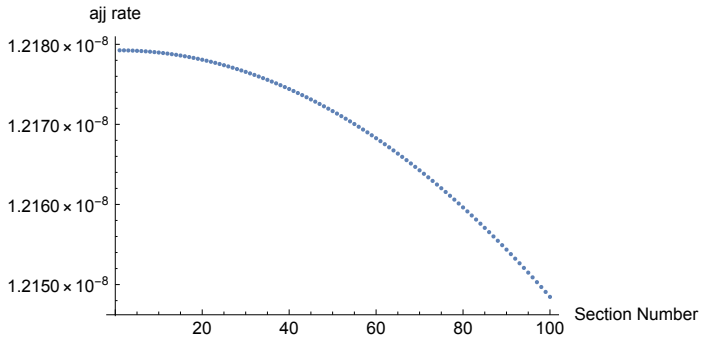
```
In[ ]:= diffeqns; (*remove semicolon to see the differential equations with coefficients*)
```

Plot pressure distribution, outgassing rates, etc.

```
In[ ]:= lastpressure = Table[ystart[j], {j, 1, Sections}]
         (*get the set of pressure values in each section at time=tmax, units are Torr*)
Out[ ]= {2.04811 × 10^-7, 2.04808 × 10^-7, 2.04803 × 10^-7, 2.04794 × 10^-7, 2.04783 × 10^-7, 2.0477 × 10^-7,
         2.04753 × 10^-7, 2.04734 × 10^-7, 2.04712 × 10^-7, 2.04687 × 10^-7, 2.04659 × 10^-7, 2.04629 × 10^-7,
         2.04596 × 10^-7, 2.0456 × 10^-7, 2.04521 × 10^-7, 2.0448 × 10^-7, 2.04436 × 10^-7, 2.04389 × 10^-7,
         2.04339 × 10^-7, 2.04287 × 10^-7, 2.04232 × 10^-7, 2.04174 × 10^-7, 2.04113 × 10^-7, 2.0405 × 10^-7,
         2.03983 × 10^-7, 2.03914 × 10^-7, 2.03843 × 10^-7, 2.03768 × 10^-7, 2.03691 × 10^-7, 2.03611 × 10^-7,
         2.03528 × 10^-7, 2.03443 × 10^-7, 2.03355 × 10^-7, 2.03263 × 10^-7, 2.0317 × 10^-7, 2.03073 × 10^-7,
         2.02974 × 10^-7, 2.02872 × 10^-7, 2.02767 × 10^-7, 2.02659 × 10^-7, 2.02549 × 10^-7,
         2.02436 × 10^-7, 2.0232 × 10^-7, 2.02202 × 10^-7, 2.0208 × 10^-7, 2.01956 × 10^-7, 2.01829 × 10^-7,
         2.017 × 10^-7, 2.01567 × 10^-7, 2.01432 × 10^-7, 2.01294 × 10^-7, 2.01154 × 10^-7, 2.0101 × 10^-7,
         2.00864 × 10^-7, 2.00715 × 10^-7, 2.00563 × 10^-7, 2.00409 × 10^-7, 2.00252 × 10^-7, 2.00092 × 10^-7,
         1.99929 × 10^-7, 1.99764 × 10^-7, 1.99595 × 10^-7, 1.99424 × 10^-7, 1.99251 × 10^-7, 1.99074 × 10^-7,
         1.98895 × 10^-7, 1.98713 × 10^-7, 1.98528 × 10^-7, 1.98341 × 10^-7, 1.9815 × 10^-7, 1.97957 × 10^-7,
         1.97762 × 10^-7, 1.97563 × 10^-7, 1.97362 × 10^-7, 1.97158 × 10^-7, 1.96951 × 10^-7, 1.96741 × 10^-7,
         1.96529 × 10^-7, 1.96314 × 10^-7, 1.96096 × 10^-7, 1.95876 × 10^-7, 1.95652 × 10^-7, 1.95426 × 10^-7,
         1.95197 × 10^-7, 1.94966 × 10^-7, 1.94732 × 10^-7, 1.94494 × 10^-7, 1.94255 × 10^-7, 1.94012 × 10^-7,
         1.93767 × 10^-7, 1.93519 × 10^-7, 1.93268 × 10^-7, 1.93014 × 10^-7, 1.92758 × 10^-7, 1.92499 × 10^-7,
         1.92237 × 10^-7, 1.91972 × 10^-7, 1.91705 × 10^-7, 1.91435 × 10^-7, 1.91162 × 10^-7}
```

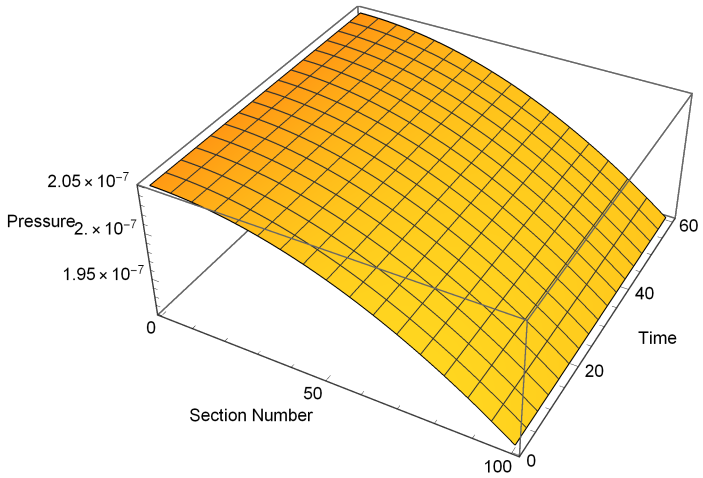
In[ ]:=
```
ListLogPlot[sectionrates, AxesLabel -> {"Section Number", "ajj rate"}]
(*shows outgassing rate/section  at last iteration*)
```

Out[ ]=



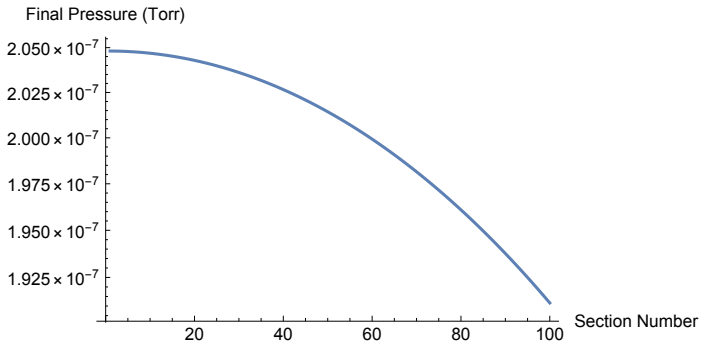
In[ ]:=
```
ListPlot3D[Table[vars2 /. First[btw], {t, 1, timestep, 1}],
 AxesLabel -> {"Section Number", "Time", "Pressure"}] (*make a contour plot*)
```

Out[ ]=



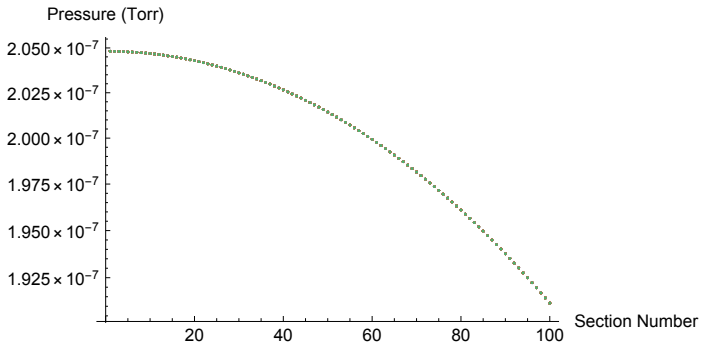
In[ ]:=
```
ListLogPlot[lastpressure, AxesLabel -> {"Section Number", "Final Pressure (Torr)"},
 Joined → True] (*plot lastpressure table*)
```

Out[ ]=



In[•]:=
```
ListLogPlot[Table[vars2 /. First[btw], {t, 1, timestep, 1}],
 AxesLabel -> {"Section Number", "Pressure (Torr)"}]
 (*shows progression over 1 timestep*)
```

Out[•]=



In[•]:=
```
pressuresection45 = Table[data[i][[97]], {i, 1, tmax}];
(*shows how to extract pressure in a particular section
 (section 97 in this example) over the entire time range*)
```

In[•]:=
```
pressuresection2 = Table[data[i][[2]], {i, 1, tmax}];
(*shows how to extract pressure in a particular
 section (section 2 in this example) over the entire time range*)
```

In[•]:=
```
ListLogLogPlot[{pressuresection45, pressuresection2}, Joined → True,
 AxesLabel -> {"Time (sec)", "Pressure (Torr)"}, GridLines → All, Frame → False]
```

Out[•]=