# LA-ICP-MS Data Reduction/Uncertainty Propagation

## 1. Parameters that must be input before data reduction takes place

These should be input as part of the Project Manager (in addition to the file handling protocol and raw data template). Some of these have already been taken care of for George’s and Jeff’s data files. However, some of these parameters might change too frequently to submit to you for customized data reduction models, and inside of the existing data parsing infrastructure you have, the user should be able to input/tweak these parameters:

Acquisition type: {static , single-collector, multi-sequence}

Machine type: MCollector, SCollector (multi-collector, single-collector)

The next set of parameters depends on the choice of acquisition type.

A. If the acquisition type is **‘static**’ (e.g. George’s data),

Begin a dynamic input table (like the one for the correlation coefficients in lab data) that maps which collectors are measuring which isotopes.

* The table should also include (1) a binary choice for each collector: {Faraday, ion counter}, (2) a value for the ‘relative gain’ of each collector (both Faradays and ion counters) and (3) a value for the ‘relative gain uncertainty’. The relative gain uncertainty is usually expressed as a 1σ relative uncertainty (% or ppm). This table should also have a way to say that the relative gains are part of the input files (for instance for George’s data from the Nu Plasma MC) and do not need to be entered by hand. I have never seen the relative gain uncertainty output in a mass spec file, so this likely will have to be input by hand. Usually, most users assign the same value to each of the relative gain uncertainties (e.g. 3 ppm, or relativeGainOneSigmaAbs = relativeGain \* 3 \* 10-6).

**Static Acquisition Model**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Collector | Isotope | Collector Type | relative gain  (default 1.0) | +/- 1σ relative  (%ion c) or (ppm faraday) | dead time (ns)  ion counter only |
| H2 | 208Pb | Faraday | 0.987887 | 3 ppm |  |
| H1 | 206Pb | Faraday | 0.996723 | 3 ppm |  |
| IC1 | 204Pb | Ion counter | 0.95 | 1 % | 14 |
| IC2 | 202Hg | ion counter | 0.94 | 1 % | 17 |

**new data dictionary terms**: *relativeGain, relativeGainOneSigmaAbs*

Note that there will be a value of both of these for each collector, through the n collectors being used. I don’t know how you want to handle naming. For the equations below, I’ll just do the math for a single collector (e.g. H1) and use relativeGain. You may want to append the name of the collector to the end of the data dictionary variable name. Relative gains and their uncertainties have no units.

* + For collectors is identified as ‘ion counter,’ there should be a place in the table to input the dead time and dead time uncertainty for that collector. The unit for the dead time and its uncertainty is nanoseconds (ns).

**new data dictionary terms:** *deadTime, deadTimeOneSigmaAbs*

* + For collectors identified as ‘Faraday,’ there should be a place to input the ‘resistor used’ and the ‘amplifier noise variance.’ The ‘resistor used is chosen from a constrained list with three choices: {‘1010 Ω’, ‘1011 Ω’, ‘1012 Ω’}. The ‘amplifier noise variance’ units are V/Hz (volts per hertz), and can either be entered by hand, or the user can choose ‘default’ which results in the following **predefined** values, depending on the choice of ‘resistor used’:

per collector

|  |  |
| --- | --- |
| resistor used | value, *amplifierNoiseVariance* (V2/Hz) |
| 1010 Ω | 1.6568×10-10 |
| 1011 Ω | 1.6568×10-9 |
| 1012 Ω | 1.6568×10-8 |

**new data dictionary term:** *amplifierNoiseVariance*

Note: the *amplifierNoiseVariance* does not have an uncertainty (it is an uncertainty, already squared).

* Also for static runs, we need a value for ‘cycle duration’. The units are seconds. This value can be input by hand or calculated from the timestamps in a data file. The unit for the cycle duration is seconds.

**new data dictionary term:** *integrationTime*

Note: the name for the input prompt and the data dictionary term don’t match because the two are synonymous in this context, but integrationTime is more appropriate for ‘single collector’ measurements. The value of integrationTime is the same for all isotopes in ‘static’ acquisitions.

B. If the acquisition type is ‘single collector’ (like Jeff’s):

* Single collector instruments for LA-ICP-MS analyses are always ion counters. Thus, we need to input only the dead time and its uncertainty (*deadTime, deadTimeOneSigmaAbs*), both in units of nanoseconds.
* However, we need a new variable for each isotope, called ‘integration time.’

**new data dictionary term:** *integrationTime*

Note: *integrationTime* takes a unique value for each isotope measured in a ‘single collector’ measurement. You may want to append the name of the isotope to the variable name, e.g. *integrationTime206Pb*.(Note: Isotope.integrationTime)

Example table of integration times

|  |  |
| --- | --- |
| isotope | integration time (seconds, milliseconds) |
| 202Hg | 50 |
| 204Pb | 200 |
| 206Pb | 100 |
| 207Pb | 400 |

## 2. Building measured intensity data covariance matrices for each fraction

For each isotope intensity measurement in each fraction, we’re going to build a covariance matrix with as many rows and columns as the number of cycles (data points) in the baseline and laser-ablation portion of the data. Thus, if there are 15 baselines and 15 cycles during ablation, we need a 30×30 matrix. Let’s call this covariance matrix *Si*. To make recalculation faster, it is probably best to store each *Si* locally rather than recalculating on the fly each time the data is reduced, like we do in Redux.

**Here is how to assemble *Si*:**

The algorithm to assemble *Si* depends on whether the collector being used to measure that isotope is a ‘Faraday’ or ‘ion counter.’ This information is in the table described in 1A for a ‘static’ acquisition, or is assumed to be an ‘ion counter’ for a ‘single collector’ acquisition.

If the collector is a ‘Faraday’

Assemble all of the measured voltages into a column vector of length n, where n is the total number of baseline and ablation cycles. For 15 baselines and 15 ablation cycles, this means n = 30 total cycles and a 30-row × 1-column vector. Call this column vector *measuredIntensityFaraday*.

Multiply measuredIntensityFaraday by a factor equal to the resistance assigned to the Faraday collector divided by 1011. For instance, multiply measuredIntensityFaraday by 3 when the Faraday resistance assigned to the isotope’s collector is 3 × 1011 Ω.

Calculate the effective counts measured, *measuredCountsFaraday*, by multiplying the vector *measuredIntensityFaraday* by the *integrationTime* and the value of *countsPerVolt* that is unique to each choice of ‘resistor used’.

|  |  |
| --- | --- |
| resistor used | *countsPerVolt* |
| 1010 Ω | 624150965 |
| 1011 Ω | 62415096.5 |
| 1012 Ω | 6241509.65 |

Next, calculate the vector that will become the diagonal of Si

SiDiag =

(measuredIntensityFaraday\*countsPerVolt + amplifierNoiseVariance\*countsPerVolt^2) / integrationTime

Note that the order of operations above is correct: countsPerVolt is squared before multiplication by amplifierNoiseVariance.

To create *Si*, arrange the elements of *SiDiag* along the diagonal of a n×n matrix that is otherwise full of zeros.

For now, if any element of SiDiag is less than or equal to zero, replace it with the average of the positive values.

If the collector is an ‘Ion Counter’

Data from ion counters can be reported in two ways--as counts per second or as an equivalent voltage on a (1011Ω) resistor. This will have to be an additional parameter in the instrument setup provided by the user.

A. If the data is reported as a voltage (George), convert intensity to cps

Assemble all of the measured voltages into a column vector of length n, where n is the total number of baseline and ablation cycles. For 15 baselines and 15 ablation cycles, this means n = 30 total cycles and a 30-row × 1-column vector. Call this column vector *measuredIntensityIonCounter*. (note parallels to [If the collector is a ‘Faraday’](#iv213bfrir7y)).

Convert the measured intensities in volts to intensities in counts per second using the [1011Ω value from the resistor used/counts per second table above](#pvosmtygqr7d).

*measuredIntensityIonCounter = measuredIntensityIonCounter* \* *countsPerVolt*

B. If the data is reported in counts per second (Jeff)

Assemble the measured count rates into a column vector of length n, where n is the total number of baseline and ablation cycles. Call this column vector *measuredIntensityIonCounter*.

No matter how you determined *measuredIntensityIonCounter* (for both A and B above):

Calculate the vector

*~~measuredVarianceFromIonCounts~~* ~~=~~ *~~measuredIntensityIonCounter~~* ~~/~~ *~~integrationTime~~*~~^2~~

*measuredVarianceFromIonCounts* = *measuredIntensityIonCounter* / *integrationTime*

and assign the values of *measuredVarianceFromIonCounts* to the diagonal elements of *Si*. ~~Note that the correct order of operations above squares~~ *~~integrationTime~~* ~~before executing division.~~ Note that you never had to square that effing term in the first place.

~~For now, if any element of on the diagonal of Si is less than or equal to zero, replace it with the average of the positive values. Perform this average separately over the baseline and on-peak parts of the data.~~ Note 11/11/14: This will no longer ever happen with new data handling for data near detection limit.

Next, add in the contribution from the uncertainty in the dead time. Square each term in the vector *measuredIntensityIonCounter*, and call this new vector *measuredCountsIntensityCounterSquared*. Next, use this term to calculate a new n×n matrix that is added to *Si*.

*Si* = *Si* + *deadTimeOneSigmaAbs* \* *deadTimeOneSigmaAbs*  \* *measuredCountsIntensityCounterSquared\* Transpose(measuredCountsIntensityCounterSquared)*

To quickly break down the equation above, the *Si* that we originally calculated in [A](#xuphvpmxhlkm) or [B](#510yuxkocj4d) above is incremented by matrix (cell-by-cell) a new matrix that is a product of four terms. The first two terms are the same and are scalar--this just squares *deadTimeOneSigmaAbs.* The third and fourth terms are a column vector and a row vector, respectively. Multiplying a n×1 column vector by a 1×n row vector (its transpose) results in a n×n matrix, the same size as *Si*. Adding this new matrix to *Si* will result in its off-diagonal terms changing from zero to a finite value, but the matrix *Si* will still remain positive definite.

If the collector is an Element 2 SEM

1. Follow the steps for the ‘ion counter’ above to create Si, just as you are doing now (8/17/16).
2. Next, for each integration in the analysis that is in analog mode, zero out all elements of Si in for the corresponding row and column.
3. Divide the mean of the four analog columns by the (ACF) for the integration to calculate *measuredIntensityAnalogMode*.
4. Calculate a single value for Si, on the diagonal, Si(i,i) = (ACF \* 0.05 \* *measuredIntensityAnalogMode*)^2.
5. Multiply the mean of the four analog columns by the ACF *measuredIntensityAnalogMode to calculate the adjustedCounts*.   
   *adjustedCounts* = *ACF* \* *measuredIntensityAnalogMode*.   
   Later, we will change the ACF in the equation above to an isotope-specific value. Use *adjustedCounts* for performing baseline corrections, calculating isotope log-ratios, etc for Pb and U.

**Starting here, intensity is in counts per second.**

**3. Isobaric Interference Correction/In-line Common Pb**

**Correcting 204Hg interference on 204Pb with 202Hg**

Isobaric interference correction:

**new data dictionary terms:** *r202Hg\_204Hg*

Subtract the mercury interference from 204Pb for each baseline and on-peak cycle using the following equation (currently treated automatically, without a variable name):

*i204Pb\_Hgc* = *i204Pb* - *i202Hg* / *r202Hg\_204Hg*

In this equation, i204Pb\_Hgc is the intensity of 204Pb, corrected for the Hg interference, i204Pb is the measured intensity of 204Pb for the baseline and on-peak cycles, i202Hg is the raw intensity of 202Hg during the baseline and on-peak cycles (NOT corrected for baseline), and r202Hg\_204Hg is the natural ratio 202Hg/204Hg, provided by the user. This value should default to 4.346, but should be user-editable.

For the uncertainty propagation:

The intensity of 204Pb already has an covariance matrix, *Si*. The uncertainty deriving from the mercury correction can be added to this matrix, and it can keep its name for simplicity. the formula is

*Si\_i204Pb* = *Si\_i204Pb* + *Si\_i202Hg* / *r202Hg\_204Hg*^2

Note that the order of operations above is intended: the denominator should be squared before the division. *Si\_i202Hg* is the complete *Si* matrix for the 202Hg intensity, with n+p rows and n+p columns corresponding to the p on-peak intensity measurements.

**4. Model the baseline and extrapolate beneath on-peak measurements**

Next we need to use the baseline measurements that occurred before the on-peak measurement to predict the values of the baseline underneath the on-peak measurement. We will do this by fitting a parametric model (a curve with an equation we know) to the noisy baseline data. At present, we have three parametric models at our disposal: the mean (y=a), the line (y=a\*t+b) and the exponential (y=a\*exp(-b\*t)+c). plus another 2. You can fit the mean and the line to the data without using an iterative solver--you have the equations already to do this. The exponential, though, will require an iterative solution (Levenberg-Marquardt), as will any other complex fit that we may add later. **Note that no baseline correction is required for the 202Hg measurement, if present.**

We must update the Levenberg-Marquardt algorithm that we have in place to accept the baseline-related portions of the covariance matrices *Si* that we calculated above, which will weight each point by its uncertainty. The update is a matrix-based algorithm, and will use the solve() function in the Jama package. Here is an outline of the new algorithm, for the three parameters (a,b,c) in the exponential fit.

For each fit function, calculate both the MSWD and BIC, the Bayesian Information Criterion (equations below). Instead of displaying the SSE in red for each, display both numbers, separated by a / or a , (whichever looks better). The MSWD is a measure of how well each individual function fits the data, and the BIC is an objective tool that allows you to choose one fit function over the other--the fit function with the smallest BIC is the best choice, controlling for the fact that the MSWD will always decrease when you use more fit parameters (e.g. 2 for a line vs. 1 for a mean).

### Levenberg-Marquardt with Matrices

Setup: you have measured n intensities (here, for the baseline only), and you are trying to fit m parameters (for an exponential, m = 3: a, b, and c). For George and exponential fit: n = 15 and m = 3. Pick out the first n rows and columns of Si that correspond to the baseline measurement, and call them *Sib*. Thus, *Sib* is an n×n covariance matrix.

Use the solutions to the weighted least squares fits as initial values for the parameters of the Levenberg-Marquardt solver. For instance, use the slope and y-intercept of the least-squares linear fit as initial values for the line + overdispersion solver. To estimate the initial value for the overdispersion itself, use the covariance matrix of the values to be fit (here, *Sib*), and the MSWD from the least squares fit (without overdispersion)

avgs2 = mean(diag(Sib)) % this is the mean of the values on the diagonal of Sib

odinit = avgs2\*(MSWD - 1) % odinit is the initial value to use for the overdispersion

If each intensity measurement is time-stamped, then use a numerical representation of the time for the value of t. If there is no time-stamp available, then assign each a positively incrementing negative integer such that the last measurement has a time-value of -1. Thus, for 15 measurements, the first measurement would get a time of -15, the second a time of -14, and so on until the final, fifteenth measurement received a time of -1.

The line numbers with asterisks are specific to the exponential fit, similar code with different values will be used for other models (e.g. power law, or exponential with no constant).

**Levenberg-Marquard Algorithm for exponential fit:**

|  |  |  |
| --- | --- | --- |
|  | *command* | *comment* |
| 1 | lambda = 1000 | scalar damping factor |
| 2 | maxiter = 100  chiTolerance = 0.001 | maximum number of iterations  convergence criterion for Χ2 |
| 3 | t = <time for each measurement> | see above. vector with n rows. |
| 4 | y = <measured baseline intensities> | vector with n rows, 1 column |
| 5\* | a = 0.1, b = 0.1, c = 0 | initialize fit parameters |
| 6 | yhat = <empty vector with n rows> | initialize yhat as a vector with n rows (1 column) |
|  | Start a loop here, which will iterate until a convergence test is passed or the maximum number of iterations (maxiter) has been exceeded |  |
| 7\* | for each of the n measurements  yhat(i) = a \* exp(-b \* t(i)) + c  end this for loop | evaluate the value of the exponential function with the proposed values of a, b, and c at each of the n times |
| 8 | Matrix r = y.minus(yhat) | r = y - yhat (subtract two vectors)  Matrix r is a vector with n rows |
| 9 | X2 = transpose(r).times(Sib.solve(r)) | transpose(r)\* (Sib.solve(r))  X2 is a scalar |
| 10 | J = <empty matrix with n rows and m columns> | initialize J, the Jacobian matrix |
| 11\* | For each row *i* in the matrix J  J(i,1) = exp(-b\*t(i))  J(i,2) = -a\*t(i)\*exp(-b\*t(i))  J(i,3) = 1  End this for loop | each element in the three equations given is a scalar, so the equations describe scalar multiplication and exponentiation. Don’t forget negative sign in front of J(i,2). |
| 12 | Matrix JTWJ = transpose(J).times(Sib.solve(J)) | JTWJ = tranpose(J) \* (Sib.solve(J))  JTWJ is an m×m matrix. |
| 13 | ldJTWJ = lambda\*diag(JTWJ) | extract the diagonal elements of JTWJ, multiply each by lambda, and make a new matrix the same size as JTWJ, with these values on the diagonal and zeros otherwise. |
| 14 | Matrix h = (JTWJ.plus(ldJTWJ) ).solve**(** (transpose(J).times( Sib~~s~~.solve(r) ) **)** | We can break this up into several smaller pieces if necessary. I wanted to limit the number of matrices that are initialized, calculated, then used only once.  The pieces would look like this: A = JTWJ.plus(ldJTWJ)  B = Sib~~s~~.solve(r)  C = transpose(J).times(B)  h = A.solve(C)  h is a m×1 vector. |
| 15\* | anew = a + h(1)  bnew = b + h(2)  cnew = c + h(3) | update the values of a, b, and c by incrementing by the proposed offsets calculated in h |
| 16 | calculate yhat as in line 7, but use the new values of the parameters calculated in 15 (here, anew, bnew, and cnew) | yhat is a vector with n rows and 1 column. You can write over the previous value of yhat. |
| 17 | Calculate r ~~from~~ with the expression in line 8, using the new value of yhat from line 16 | r is a vector with n rows and 1 column. You can write over the previous value of r. |
| 18 | Calculate X2new with the expression in line 9, using the new value of r from line 17, and the original *Sib* (the matrix Sib never changes) | X2new is a scalar. Do not write over the previous value of X2. |
| 19 | If X2new > X2  multiply lambda by 10,  increment iteration count by 1, and  go back to line 7. | In your ExponentialFitFunction, this is the line ‘if (chi2aNew > chi2a)’ |
|  | Else |  |
| 21\* | a = anew  b = bnew  c = cnew | In your ExponentialFitFunction, this is a = a + delta\_aM.get( 0, 0 );  ...etc. |
| 22 | If (1-X2new/X2) < chiTolerance | In your ExponentialFitFunction, you define the variable deltaChiIsInTolerance |
| 23 | Calculate a new value for all elements of the matrix J using the expressions from step 11 | update J using the values of a, b, and c (which until 21 were anew, bnew, and cnew) |
| 24  24\*  24\*  24\*  24\* | Matrix Vp = (transpose(J).times((Sib).solve(J))).inverse()  aOneSigmaAbs = sqrt(Vp(0,0))  bOneSigmaAbs = sqrt(Vp(1,1))  cOneSigmaAbs = sqrt(Vp(2,2))  BIC = X2+log(det(Sib))+ m\*ln(n)  exit, function has been fit | Calculate summary statistics, save off values of fit parameters (e.g. a, b, and c, and X2) and exit fit function  Vp = (JT\*Sib\*J)-1  Vp is an m×m matrix that contains the variances for the fit parameters on its diagonal.  n is number of points being fit. ln is the natural log. In general,  BIC = X2 + log(det(Sib)) + m \* ln(n) |
| 25 | Else (if (1-X2new/X2) ≥ chiTolerance) | Else |
| 26 | lambda = lambda / 10  If iterations < maxIter  iterations ++  X2 = X2new  goto line 7  else  exit function, go for linear fit. | divide lambda by 10  If the maximum number of iterations has not been exceeded, start again with the new parameter values at line 7 |

To convert chi2 to MSWD divide by (n - count of parameters). For exponential, the count of parameters is 3.

**For George**: add some uncertainty to approximate the uncertainty contribution from the baseline subtraction, instead of the line-fitting and uncertainty propagation above.

**For mean and line fits to baseline data:**

***Mean:***

Create *Sib*, the covariance matrix for the baseline intensity measurements, as described in the beginning of the Levenberg-Marquardt instructions. For n baseline intensity measurements,

*Sib* is an n×n covariance matrix. The formula for the (weighted) mean also requires a column vector with n rows (and one column) filled with ones, hereafter called *onesVector*. Recall that the measuredIntensity is in cps for both Faraday and ion counter data (and the measuredIntensityVariance has units cps^2).

It is most efficient to calculate the uncertainty (here, the variance) in the weighted mean intensity first. The formula is

meanIntensityVariance = 1 / **(**transpose(onesVector).times( Sib.solve(onesVector) )**)**

where the bold parentheses are for clarity. The one sigma absolute uncertainty in the mean is the square root of its variance. The denominator of the fraction is a scalar, and so is *meanIntensityVariance*.

The weighted mean baseline intensity can now be calculated as

meanIntensity =

meanIntensityVariance.times**(**transpose(onesVector).times( Sib.solve(measuredIntensity) )**)**

where measuredIntensity is either *measuredIntensityFaraday* or *measredIntensityIonCounter*, depending on the collector used. Note that *meanIntensity* is also a scalar.

To estimate the goodness of fit, you can no longer calculate the residuals and increment the sse in the same loop. First create a column vector (n rows, 1 column) of residuals, where each residual is the *measuredIntensity* minus the *meanIntensity*. The goodness of fit is now measured by *X2*, a weighted version of the sum of the squared errors.

*X2* = residuals.transpose().times( Sib.solve(residuals) )

For George’s data, *skip the calculation the of X2*. This is the only part of the mean calculation that will ‘blow up’ for George’s data.

To compare goodness of fit between multiple models (e.g. fit functions: mean vs. line vs. exponential), calculate the Bayesian Information Criterion, or BIC. This will replace SSE as the number in red displayed for each fit function. ~~The formula is~~

~~BIC =~~ *~~X2~~* ~~+~~ *~~m~~* ~~\* log(~~*~~n~~*~~)~~

~~where~~ *~~X2~~* ~~is the chi-square value calculated above, and for each fit function,~~ *~~m~~* ~~is the number of model parameters (1 for a mean, 2 for a line, 3 for an exponential), and~~ *~~n~~* ~~is the number points being fit (here, the number of baseline integrations,~~ *~~n~~*~~).~~ So for the mean,

For the mean, without overdispersion

BIC = (r'\*(Sib\r) + log(det(Sib)) + m\*log(n)

BIC = residuals.transpose().times( (Sib.solve(residuals) ) + log(det(Sib)) + m\*log(n)

where ln() and log() are both the natural logarithm and m = 1 for the mean.

To calculate the log(det(Sib)) term, instead of calculating the determinant, calculate the sum of the eigenvalues. Sib has only positive real eigenvalues, so use the JAMA function getRealEigenvalues(Sib). Take the natural logarithm of each element in the resulting vector, then sum the logged values.

***Line:***

Find these two lines in your line fitting code

Matrix C = aM.transpose().times( aM );

Matrix D = aM.transpose().times( bM );

Change the lines to

Matrix C = aM.transpose().times( Sib.solve(aM) )

Matrix D = aM.transpose().times( Sib.solve(bM) )

Calculate the vector fitM as you did before.

The covariance matrix for the y-intercept and slope, analogous to Vp from Levenberg-Marquardt, is

Matrix Vp = C.inverse();

and the one sigma uncertainties in the y-intercept and slope are then

yintOneSigmaAbs = sqrt(Vp(0,0))

slopeOneSigmaAbs = sqrt(Vp(1,1))

You will need to replace the code for the goodness of fit. You already calculate the residuals as a double-precision array. Convert these into a column vector called *residuals*, then calculate

X2 = residuals.transpose().times( Sib.solve(residuals) )

The X2 measure of goodness of fit is an uncertainty-weighed measure of the sum of the squared errors (SSE).

Calculate and display the Bayesian Information Criterion (instead of SSE),

For the line fit, without overdispersion,

BIC = (r'\*(Sib\r) + log(det(Sib)) + m\*log(n)

BIC = transpose(residuals).times( (Sib.solve(residuals) ) + log(det(Sib)) + m\*log(n)

where n is the number of points being fit (could be different for baselines and on-peak), and ln() and log() are both the natural logarithm, and m = 2 for a line.

**Subtract baseline from on-peak measurements**

There are p on-peak measurements (15 for George’s data, 300 for Jeff’s data), which should be arranged in a column vector called *measuredIntensityOnPeak*. Using the chosen model (mean, linear, exponential, etc), extrapolate the intensity of the baseline underneath the on-peak measurements. Subtract the extrapolated baseline intensity from the on-peak intensity to calculate the baseline-corrected on-peak intensity. If there are p on-peak measurements, then this results in a vector with p rows and one column, called *peakIntensityBLcorr*.

**5. Propagate uncertainties in baseline-corrected on-peak measurement (TRACK P VS N)**

Propagating the uncertainty in the baseline correction into the corrected on-peak measurements will result in a p×p covariance matrix, *Sopbc*, with rows and columns corresponding to each corrected on-peak measurement. The uncertainty propagation relies on building three component matrices, J11, J21, and J22. The calculation for J11, J21, and J22 will be different for the mean, line, and exponential fits through the baseline data, but then will be combined into the same uncertainty propagation equations.

The sizes of J11, J21, and J22, depend on the number of parameters used in the baseline fit model, m (m=1 for mean, m=2 for line, m=3 for exponential, etc), the number of baseline measurements taken, n (n=15 for George and n=300 for Jeff), and the number of on-peak intensities measured, p (p=15 for George’s data, p=300 for Jeff’s data). J11 has m rows and n columns, J21 has p rows and m columns, and J22 has p rows and p columns.

A. Component matrices if the baseline was fit with a **mean**:

J11 is a row vector with all elements the same value: 1/n (where n is the number of baseline integrations)

J21 is a column vector with p rows, where each element is -1

J22 is an identity matrix with p rows and p columns (1s on the diagonal, 0s off the diagonal)

B. Component matrices if the baseline was fit with a **line**:

J11 has m=2 rows and p columns. Using the variables from the line fit. Using the variables from the line fit algorithm, calculate

matrix D = transpose(Sib.solve(aM))

J11 = C.times(D)

where C was calculated previously ([see above](#cyvwzgrblb0k)) as

Matrix C = aM.transpose().times( Sib.solve(aM) ) <- this is not new code

J21 has p rows and m=2 columns. The first column of J21 consists of the p on-peak intensity measurements (not baseline-corrected), *measuredIntensityOnPeak*. In the second column, all elements are ones.

J22 is a diagonal matrix with p rows and p columns, with the best-fit line slope *a* for each diagonal element and zeros off the diagonal.

C. Component matrices if the baseline was fit with a **L-M optimization**:

J11 is simply the transpose of the matrix J used in the L-M algorithm to fit the baseline.

J21 is calculated just like the matrix J in the L-M algorithm (step 11 above), but at the p times (ti) corresponding to the p on-peak measurements. So use the a and b parameters for the best fit to the baseline data, and use the t(i) corresponding to the on-peak times.

J22 is again a diagonal matrix, with zeros in all the off-diagonal elements. But unlike the two versions of J22 above for the mean and the line, each of the p elements on the diagonal are different. They can be calculated with the equation

J22(i,i) = -a\*b\*exp(-b\*t(i)), where the a and b are the L-M best-fit parameters. Use the values of a and b from the background fit, and the t(i) for the y-values.

**Performing the uncertainty propagation**

The Jacobian matrix required for the uncertainty propagation, JOnPeak, is created using J11, J21, and J22. First calculate

JOnPeak1 = J21.times(J11) %Note: this takes time.

JOnPeak1 and J22 now have the same number of rows (p), but potentially different number of columns, n for JOnPeak1 and p for J22. Append the two matrices, so that the p rows are shared and there are now n+p columns for the combined matrix:

JOnPeak = [JOnPeak1 J22]

To calculate the covariance matrix for the baseline-corrected uncertainties, Sopbc, use the uncertainty propagation formula,

Sopbc = JOnPeak.times(Si).times(transpose(JOnPeak))

Sopbc is on peak baseline corrected.

**6. Calculate raw ratios (not yet corrected for fractionation) and their uncertainties (covariance matrices)**

Call the on-peak baseline-corrected intensity for an arbitrary isotope X *iX\_opbc*. For instance, the on-peak baseline-corrected 206Pb intensity would be i206Pb\_opbc. The covariance matrix for each of these intensities is called *Sopbc\_X*, so that for 206Pb, the covariance matrix for each on-peak baseline-corrected intensity is *Sopbc\_i206Pb*.

A. Before evaluating logarithms, check the data to make sure that the isotope is abundant enough to be detected and that there are no negative intensities that would choke the log

First, test that the arithmetic mean of i204Pb\_opbc and i207Pb\_opbc, the two isotopes likely to be the smallest, are significantly different than zero. For each vector of intensities,

1. Calculate the arithmetic mean and standard deviation
2. Reject (from this quick and dirty analysis) all points greater than two standard deviations from the mean
3. Calculate the mean and standard error of the remaining points
4. **If** the mean is not greater than two times the standard error of the mean, then it is ‘below the detection limit’ and will not be considered further for calculations that involve ratios.
5. **Else**, for 204Pb or 207Pb, **if** the isotope is above the detection limit but there are negative intensities in i204Pb\_opbc or i207Pb\_opbc, **but** these negative intensities comprise <10% of the total number of intensities (e.g. 1 out of 15), **then** automatically reject this single intensity/ratio from consideration and proceed as usual. Recall that we have disconnected the 206/204-207/204-208/204 rejection from the 206/238-206/207-208/232 rejection, so that only ratios to 204 are being rejected here. In plots, represent this data point as a single hash mark on the x-axis at the appropriate time point, since it cannot be plotted in log-space and would scale the y-axis badly in (non-logged) ratio space.
6. **Else**, for the 204Pb only, if the 204Pb is above the detection limit and >10% of the i204Pb\_opbc intensities are negative, then directly calculate r206\_204m, r207\_204m, and r208\_204m, without the usual steps of calculating log-ratios and finding a fit to the data. **Use all positive and negative values in i20XPb\_opbc.**
   1. r206\_204m = mean(i206Pb\_opbc)/mean(i204Pb\_opbc)
   2. r207\_204m = mean(i207Pb\_opbc)/mean(i204Pb\_opbc)
   3. r208\_204m = mean(i208Pb\_opbc)/mean(i204Pb\_opbc)

These r20X\_204m values take the place of the intercepts that you’d otherwise calculate. They can next be used in B1 ([here](#xhsyk9wkc83p)) as unkmeas to be fractionation corrected and then used for Pbc-correction. To plot them in ‘Raw Ratios’ or ‘Fit each Unknown’, you’ll have to plot both positive and negative ratios. For the latter, default and force the user to choose the ‘Mean’ fit, plotting and displaying the value calculated above.

Note: the struck-out text below is no longer applicable. ~~Next, test for any non-positive (zero or negative) values in each on-peak baseline-corrected intensity. These will be much more rare than zeros in the baselines. If you find one, replace it with the smallest positive number possible. This will ensure that there are no problems when evaluating the natural logarithms to follow.~~

B. Calculate the natural logarithm of the on-peak baseline-corrected intensities, and convert the covariance matrix to the covariance matrix for the log-intensities.

Evaluate the natural logarithm of each on-peak baseline-corrected intensity. Call these new log-intensities *liX\_opbc*, so that the on-peak baseline-corrected 206Pb log-intensity is *li206Pb\_opbc*.

Calculate the covariance matrix for the log-ratios, *Sopbclr*. Since this new matrix is the same size as *Sopbc*, you can write over it to save space and time. First, create a column vector with p rows called *Jlogr*, where each entry is the reciprocal of the corresponding on-peak baseline-corrected intensity. For each isotope X,

*Jlogr* = 1 / *iX\_opbc*, where the division operation / is element-wise division, not a matrix inverse. For iX\_opbc, use the value corrected for the hidden 10^11 division (use the value that’s been multiplied by 3 for George’s 10^11 ohm amplifiers, not the original value reported by the instrument).

You can now calculate *Sopbclr* as

*Jmat* = *Jlogr*.times(*Jlogr*.transpose())

*Sopbclr* = *Jmat*.arrayTimes(*Sopbc*)

where *Jmat* is a temporary matrix that can be immediately deleted.

C. Now that the intensities have been logged, the ratios become differences.

There are two ways to calculate the data needed for the intercept fits and uncertainties. The best way involves very large matrices, which would still fit into the current Levenberg-Marquardt or line/mean fits, but would likely take a significant amount of time to calculate. We may add this during future development, but for now we will simplify to the three ratios that everyone calculates: 206/207, 206/238, 208/232

*lr206Pb\_207Pb* = *li206Pb\_opbc*- *li207Pb\_opbc*

*lr206Pb\_238U* = *li206Pb\_opbc*- *li238U\_opbc*

*lr208Pb\_232Th* = *li208Pb\_opbc*- *li232Th\_opbc*

Calculate the covariance matrices for each of the log-ratios, *Slr\_X\_Y*

-Note: mean of intensity data (case 6)!

1. For each log-ratio
   1. **ONLY FOR THE SAME ION COUNTER IN BOTH X AND Y** Matrix multiply the column vector for on-peak baseline-corrected intensity of isotope X by transposed column vector for the on-peak baseline-corrected intensity of isotope Y to produce a p×p matrix *Sxyod* (od for off diagonal). Multiply this whole matrix by *deadTimeOneSigmaAbs^2* (careful here for error from JAMA). For instance, for the log-ratio *lr206Pb\_207Pb,*

*Sxyod* = *i206Pb\_opbc*.times(transpose(*i207Pb\_opbc*)) [matrix multiplication]

*Sxyod* = (*deadTimeOneSigmaAbs^2*)*.*times(*Sxyod)* [scalar multiplication]

* 1. *Slr\_X\_Y* = *Sopbclr\_X* + *Sopbclr\_Y* - 2\**Sxyod*

(note that the final term, *Sxyod*, exists only for ion counter in both X and Y)

1. *9.Nov.2014* After making all Slr\_X\_Y matrices, make a large matrix called **SlrAll** that has 6\*p rows and 6\*p columns, where p is the number of on-peak integrations. This will be a block matrix, built out of p-by-p matrices that we have already calculated. For the six p-by-p diagonal blocks, use the Slr\_X\_Y matrices that were just calculated for, in turn, the 206Pb/207Pb, 206Pb/238U, 208Pb/232Th, 206Pb/204Pb, 207Pb/204Pb, and 208Pb/204Pb ratios. Off-diagonal p-by-p blocks are filled if and only if those two ratios share a common isotope, by the Sopbclr matrix for the shared isotope. Therefore, place the Sopbclr\_206Pb matrix in the off-diagonal positions between the 206Pb/207Pb, 206Pb/238U, and 206Pb/204Pb blocks; and place the Sopbclr\_204Pb matrix in the off-diagonal positions between the 206Pb/204Pb, 207Pb/204Pb, and 208Pb/204Pb blocks.

**7a. Calculate intercepts and uncertainties (intercept method)**

You now have three vectors of log-ratios, *lr206Pb\_207Pb*, *lr206Pb\_238U*, and *lr208Pb\_232Th*, and three *Slr\_X\_Y* covariance matrices. These can be fit using the mean, line, and exponential (via Levenberg-Marquardt algorithm) parametric equations, just like the baseline in [Part 4](#oswce4dp7vyl). If the estimated uncertainties in each data point do not account for their observed scatter about the chosen fit function, then an overdispersion term is required--a constant variance added to each data point to bridge that gap.

Next, calculate the derivative of each y-intercept with respect to the measured log-ratios, Jyintlr. There will be a separate Jyintlr for each intercept, six for data reduced with the Pbc-correction scheme B, and three otherwise. The form of Jyintlr depends on whether a mean, line, or exponential was fit to the data. **In all cases below, if overdispersion is turned on, Slr\_X\_Y *includes* the overdispersion term calculated for that fit.**

For a ratio of two means (condition 6 (page 16), >10% negative), use

Jyintlr\_X\_Y = -mean(numerator\_intensities)/mean(denominator\_intensities)^2 \* onesv

where onesv is a row vector of ones, of length p. In other words, Jyintlr is a row vector filled with the same value. The order of operations implied above is correct - square before dividing. - Noah needs to do some work here.

For a weighted mean,

Jyintlr\_X\_Y = meanIntensityVariance.times( Slr\_X\_Y.solve(onesVector) )outputWriter.println();

*<Note - if the Jyintlr math above produces a column vector, transpose to produce a row vector. I don’t know how JAMA treats the solve()>*

For a line,

Jyintlr\_X\_Y = the first row of C.solve( transpose(aM) ).times( inverse(Slr\_X\_Y) )

written as a matrix equation, this would be (C)-1\*aMT\*(Slr)-1 where -1 is a matrix inverse, T is a matrix transpose, and \* is a matrix multiplication. The result of the matrix multiplication has two rows, and Jyintlr is only the first row.

For an exponential, first create a new three-element row vector called Jyintp = [1 0 1].

Next, calculate

Jyintlr\_X\_Y = Jyintp.times( transpose(Jabc).times( Slr\_X\_Y.solve(Jabc) ).solve( transpose(Jabc).times( inverse(Slr\_X\_Y) ) )

In matrix form, the equation is Jyintlr = Jyintp \* (JabcT\*Slr-1\*Jabc)-1 \* JabcT\*Slr-1

After calculating all fit functions for the standards and unknowns, create a matrix for each analysis called **JyintlrAll** (J-y-intercept-log-ratio-All), which will be used for calculating the uncertainty correlations between the different ratios. JyintlrAll has three rows if there is no 204Pb-based Pbc correction (if no scheme B at present), and six rows there is a 204Pb-based Pbc-corrrection (if scheme B at present). JyintlrAll has 3\*p or 6\*p columns, where p is the number of on-peak integrations, with the three and six corresponding to the same conditions above. Thus, JyintlrAll has a block form, where each block consists of a row vector with p columns. Populate the block diagonal of JyintlrAll with the Jyintlr row vectors calculated above: In the first row, first block of columns, use Jyintlr\_lr206\_207, in the second row, second block of columns, use lr206\_238, etc. The off-diagonal blocks are zeros.

Finally, calculate the matrix **Sfci** for each fraction, using the equation

Sfci = JyintlrAll\*SlrAll\*JyintlrAllT where SlrAll was calculated [here](#9imq6o6nwlp4).

**If both X and Y were measured on the same ion counter**,

then in addition to the intercepts and intercept uncertainties, you will also need to keep track of the derivative of the intercept with respect to the dead time.

To do this, create a new column vector with s rows, one for each standard (and 1 column) called *dLrInt\_dDt*, with one element for each standard isotope (log)-ratio X/Y. For each of the standards, calculate the corresponding element of *dLrInt\_dDt*

1. Calculate Jyp. The formula for Jyp varies depending on whether a mean, line, or exponential fit was used (these are all that are available for fitting individual standards--no spline.

A. For a weighted mean (with/without overdispersion):

*Jyp = onesv*

where onesv is a column vector with length the number of on-peak measurements

B. For a linear fit (with/without overdispersion):

*Jyp = [onesv t]*

where t is a column vector with all of the times that the integrations are assigned (currently integers starting with 1, I believe)

C. For an exponential fit (y = a\*exp(b\*x) + c)

*Ja = exp(b\*t);*

*Jba = t.\*Ja; %arrayTimes*

*Jb = a\*Jba; %scalar multiplication*

*~~Jbb = t.\*Jb; %array times~~* note 11/8/14 don’t need this, no H calc

*Jc = onesv;*

*Jyp = [Ja Jb Jc];*

where this is the same code as appears in the gradient for ‘expmat’ and ‘expod’ in the L-M fitting algorithm

2. Calculate *Jpy*. No matter what fit function was used (mean, line, or exp), the formula is the same. However, if overdispersion is on, then the overdispersion term should be added to the diagonal of the covariance matrix (*Slr\_X\_Y*) for the calculation.

Jpy = (*Jyp*.transpose().times( *Slr\_X\_Y.*solve(*Jyp*) ).solve

( ( *Slr\_X\_Y*.solve(*Jyp*) ).tranpose() )

Next, assemble *JIntp*, a row vector that is the derivative of the intercept with respect to the fit parameters. This will vary depending on the fit. For the mean, *JIntp* is simply *Jpy*. For the line fits, JIntp is the first row of *Jpy*. For the exponential fit, *JIntp* is the first row of *Jpy* plus the third row of *Jpy*.

Next, assemble *intDiff*, a column vector with the same length as the number of on-peak measurements. Each element of intDiff is simply the difference in deadtime-corrected intensity in the numerator isotope minus the dead-time corrected intensity of the denominator isotope. For the log-ratio log(X/Y), intDiff(i) = X(i) - Y(i) where X and Y are the intensities, and i the appropriate index.

Finally, calculate the element of *dLrInt\_dDt* for this standard.

*~~dLrInt\_dDt~~* ~~=~~ *~~intDiff~~*~~.transpose().times(~~*~~JIntp~~*~~)~~

dLrInt\_dDt = JIntp.times(intDiff)

Addendum: make a different dLrInt\_dDt for the standards and the unknowns, performing the calculation in the same way for each, just placing the calculated scalar value in the correct vector.

**7b. Calculate offsets and uncertainties (downhole/Iolite method)**

This will closely follow the technique we used before, but in log-ratio space. Parts 1, 2, and 3 below correspond to the three radio buttons below the ‘Downhole’ fractionation technique in the left-hand panel of the Manage Raw Data Session.

For each ratio X\_Y of interest (ie 206/207, 206/238, 208/232):

1. Plot the log-ratios of the standards, all in the same graph, all different colors (last time, we used a blue to green color map). This will be a bunch of semi-parallel curves.

Next, calculate the weighted mean of all of the fraction trends. First, calculate the sum of the inverses of all of the Slr\_X\_Y (n of these) covariance matrices for each standard. Call this sumInvSlr\_X\_Y. Second, for each standard, calculate Slr\_X\_Y.solve(lr\_X\_Y), where the lr is the vector of log-ratios, and sum these. Call this sumInvSlrTimeslr. The first of these matrices will be square and have the same number of rows and columns as there are on-peak integrations; the second will be a column vector with the same number of rows as the first.

Finally, calculate

wtdMeanStdIntegrations = sumInvSlr\_X\_Y.solve( sumInvSlrTimeslr )

and

SwtdMeanStdIntegrations = inverse( sumInvSlr\_X\_Y )

The column vector wtdMeanStdIntegrations will have the same number of rows as there are on-peak integrations--this was our **thick black line-**-, and the square matrix SwtdMeanStdIntegrations will have the same number of rows and columns.

You can now use any of our fit functions to fit wtdMeanStdIntegrations, using SwtdMeanStdIntegrations as the covariance matrix, just like you would for an intercept fit. This fit was our **thick red line**. You can also add a spline fit here, if you want - basically, all the options in the session fits. *Note from Jim - the spline takes too long to calculate currently, so turned off.* For each integration time, calculate the difference between the log-ratio of the standard and the fit function. Call this difference Beta, so that

Beta(*i*) = log(stdXYtrue) - fitFunctionValue

for each integration time *i*, where fitFunctionValue is the value of the **thick red line** at integration time *i* and stdXYtrue comes from the zircon standard measurement model.

For the fit, calculate the value of the fit at each of the standards and the matrix Sf, using the same math as we use in the session fitting. There are instructions for calculating Sf for each of the possible fit functions (mean, line, exp, spline) usually in red in section 9B.

Like the session fitting graph, it would be cool to see in this in ratio/logratio/Beta views with a radio button chooser.

THERE SEEMS TO BE AN ISSUE HERE - IF A SINGLE FRACTION HAS AN ACQUISITION TURNED OFF, THEN ITS SLOGXY DOES NOT MATCH THAT OF THE DOWNHOLE SESSION IN SIZE ...SO I AM SETTING VALUES OF SLOGX\_Y TO ZERO FOR THAT ROW AND COLUMN, EXCEPT THEIR INTERSECTION AND I AM SETTING THE Slr\_X\_Y.solve(lr\_X\_Y) FOR THAT POINT TO 0, WHICH MAKES POINTS IN MEAN ... NEED SOLUTION.

Note: Ignore for now. Use full covariance matrices with no entries deleted. I’ll find a fix, but this won’t do much damage.

2. Plot up a graph of the differences between each standard integration and the interpolated value of the fit function from above. This will look like the ‘raw ratios’ plots, but with differences in log-ratios instead of ratios. For each standard, there are only two options: mean with/without overdispersion.

The covariance matrix for the differences is the covariance matrix for the log-ratios of the standard (Slr\_X\_Y) plus the covariance matrix SwtdMeanStdIntegrations that you calculated in step 1.

You can use the vector of differences plus this covariance matrix to calculate the weighted mean with/without overdispersion. This will produce a weighted mean and an uncertainty. Just like for the intercepts for each standard in the session, you can arrange the one-sigma uncertainties in these weighted means, squared, on the diagonal of a square matrix with a row and column for each standard--this is Ss.This again is a weighted mean of differences, this time differences between the mean downhole trend (thick red line) and the individual downhole measurements (all in log-ratios). For the standard measured at time *t* in the session, calculate a value of Delta.

Delta(*t*) = weightedMean[ fitFunctionForDownHoleXY(i) - log( stdXYmeas(i) ) ]

Translation = weightedMean( BigMasterRedLine - a blue/green line at time t)

3. This is the session view, and should look the same as the intercept session. Use the weighted means (*NoteFromJim: this is the weighted mean of the differences for each standard. NoteFromNoah: Correct.*) and Ss just like you would the intercepts and their Ss matrix. Calculate any of the possible fits to the session here -- mean/line/exp/spline, all with/without overdispersion. This function, DeltaFunction, can be used to interpolate the value of delta at any session time *t* for an unknown, Delta(*t*unk).

4. Correct each unknown using the offsets Beta and Delta you calculated along with the mean standard fit from part 3. For each log-ratio integration of the unknown, calculate

unknownIntegrationXY(i) = unknownMeasuredLogRatioXY(i) + Beta(i) + Delta(t)

You have done this before... see if you can find the code. Plot the corrected unknowns in the same kind of view as part 2 -- for George, 15 corrected on-peak integrations for each standard, with a column for each standard and a row for each ratio (206/207, 206/238, 208/232). As in part 2, the only fit options for each standard will be mean with/without overdispersion. You can’t do this fit yet as I haven’t given you the covariance matrix for each of these.

I will expand more on the uncertainty propagation here later, but the goal will be to calculate lru and Su. These will then be fed into Redux and skip to Step 12.

**8. Fit session data for standards**

If the estimated variances calculated above--mostly amplifier noise and Poisson-distributed noise from counting ions--is calculated correctly, then all of the MSWD values calculated for the intercept or downhole fits will be near unity. It is more likely that these uncertainties will be over- or under-estimated, and the variability of the measured data will be greater or less than the theoretically expected value.

Call the variable *s* the number of standards used in the session. For each standard isotope ratio (206/207, 206/238, 208/232),

Create a column vector with *s* rows (and 1 column) called SsDiag containing all of the intercept variances for the applicable isotope ratio.

Place the vector SsDiag on the diagonal of a s×s matrix called Ss. If one or both collectors are for the isotopes in the isotope ratio are Faradays, then use Ss to fit a mean, line, or exponential to the data with our matrix-based regression algorithms.

If both of the isotopes in the isotope ratio were measured on the SAME ion counter, then calculate the covariance between each pair of standards, and add them to the off-diagonal positions of Ss.

After assembling the vector dLrInt\_dDt above, calculate Ssod (a matrix containing the off-diagonal elements of Ss) as

Ssod = *deadTimeOneSigmaAbs*^2 \* *dLrInt\_dDt*.times( *dLrInt\_dDt*.transpose() )

The matrix Ssod with have s rows and s columns. Next, change the diagonal terms of Ssod to zero, then add this new Ssod (which contains only off-diagonal terms) to Ss (which contains only diagonal terms) to calculate a new Ss.

**9. Calculate uncertainties in standards; fractionation correction uncertainties in unknowns**

**9A.** To determine the covariance matrix for each fit function, you need the **hat matrix** (*hatMatrix*), which transforms the vector of measured values into the vector of fitted values.

For the mean, the hat matrix is

Arow = meanIntensityVariance \* Ss.solveTranspose(onesVector.transpose())

*hatMatrix* = onesVector.times( Arow.transpose() )

where meanIntensityVariance is defined [here](#wpkprf7mxi2n) and onesVector is an

For the line, the hat matrix is

E = (Ss.solveTranspose(aM.tranpose())).transpose()

*hatMatrix* = aM.times(C.solve(E))

where the matrix C is defined [here](#cyvwzgrblb0k)

For the exponential, the hat matrix is

E = (Ss.solveTranspose(J.transpose()).tranpose()

*hatMatrix* = J (JT Ss-1 J)-1 JT Ss-1

To calculate the hat matrix A for the spline, use the variable you’ve already calculated, ImAwalpha.

*hatMatrix* = eye(s) - ImAwalpha

where eye(s) is an s×s identity matrix (ones on the diagonal, zeros off the diagonal).

**9B.** **Calculate the uncertainty in the fit to the session of standards**

For any of the plots, choose a vector of times over which to interpolate and plot. This vector should contain ten to a hundred times more points than the number of standards--enough to appear smooth when plotted if the user enlarges the session view of e.g. the 206/238 ratio to full screen. About 2000 points should take care of most screen resolutions. Call this vector of times *tInterp*.

For each point in *tInterp*, we will calculate the offset between the fit function itself and its uncertainty, which we’ll call *splunct*. For the log-ratios, this offset is symmetric in the positive and negative directions. You can use the fit function itself and *splunct* to calculate the positive and negative uncertainty envelope over all points in *tInterp*, *plusEnv* and *minusEnv*. To convert to (un-logged) ratios or to alphas, convert each point in *plusEnv* and *minusEnv* (do not convert *splunct*--this does not work). The values of *splunct* given below are always multiplied by two to display the 2σ uncertainties.

Once the log-ratios for the standards are fit by a fit function, it is straightforward to calculate the fractionation. In general, we will use the following formula for fractionation correction:

Solving, stdtrue = φ stdmeas

Φ = log(stdtrue) - log(stdmeas)

Where Phi (Φ) is the (log/exp-based) fractionation, stdtrue is the true ratio of the standard, as recorded in the (EARTHTIME) standard model, log(stdmeas) is the interpolated value of the session-based standard fit function at the time that the unknown was measured.

A more conventional measure of the fractionation is φ = exp(Φ), where phi represents the multiplicative offset between the true and measured value of the standard:

stdtrue = φ stdmeas

The following sections will also describe how to calculate *Sf*, the covariance matrix for Φ, the fractionation correction.

***For the mean:***

For the session plot, the 1σ uncertainty in the weighted mean of the standards is

*splunct* = sqrt(*meanIntensityVariance*)

where meanIntensityVariance is the left-over nomenclature for the variance of the mean fit function. We should refactor this sometime soon. It is easiest to calculate these values independently, rather than extract the diagonal of *Sf* and take the square root, as below.

*Sf* has *u* rows and *u* columns, where *u* is the number of unknowns (either unknown samples or unknown values that we’re interpolating over, e.g. 2000). To build *Sf*, simply fill the entire matrix with value of *meanIntensityVariance*, where *meanIntensityVariance* is the left-over nomenclature for the variance of the mean fit function.

***For the line:***

At each point, ti, that you want to determine the uncertainty for the line fit, the 1σ uncertainty (as a scalar) is

splunct =sqrt( [1 *ti*] \* *Vp* \* [1

*ti*] )

Inside the square root above, there are three terms. The first is a one-row by two-column matrix containing a 1 and *ti*, the time of interpolation (at present a number between 0 and 8085 for George’s data). The middle term is *Vp*, the fit parameter covariance matrix. The third term is a two-row by one-column matrix containing a 1 and *ti*.

For the first and third terms of the multiplication: you might want to build a big matrix beforehand with the number of rows being the number of times you're interpolating and two columns. The first column would be ones, and the second columns the times at which you’re doing the interpolation (*tInterp*). This is analgous to the aM matrix you created for the linear regression, but with more x-values. You can then use the getMatrix command in JAMA to pull out a row of this big matrix for the first term of the multiplication, [1 ti], and then transpose it to create the third term of the multiplication.

To calculate *Sf* for the unknowns, build a larger version of the equation for *splunct* above. Construct a matrix *Jf* with u rows and two columns. The first column is full of ones, and the second column contains each of the times at which the unknowns were measured (the first unknown goes in the first row, the second unknown in the second row, etc). Then calculate

*Sf* = *Jf*.times(*Vp*).times( *Jf*.transpose() )

Again, you could calculate *splunct* by extracting the diagonal of *Sf* and evaluating the square root, but the equation above is faster for doing the plotting, as it doesn’t evaluate all of the off-diagonal elements.

***For a Levenberg-Marquardt fit to a non-linear function (e.g. exponential):***

For each point, calculate a row of J just like you did for step 11 in the Levenberg-Marquardt algorithm. For instance, for the exponential, J has three elements,

J(1) = exp(b\*t(i)) Note: this value cannot be negative.

J(2) = a\*t(i)\*exp(b\*t(i))

J(3) = 1

Place these elements in a row matrix called *Jrow* and a column matrix called *Jcolumn*, then calculate,

splunct = sqrt( *Jrow* \* *Vp* \* *Jcolumn* )

In the future, there may be more fit functions than the exponential, but the same principle will apply -- use the calculation for a single row of J, then multiply by Vp, then multiply by the same J as a column.

To calculate *Sf*, create a matrix called *Jf* with *u* rows, where *u* is the number of unknowns, and three columns. For each of the *u* unknown measurement times, evaluate J(1), J(2), and J(3) above, and place them in the appropriate row of *Jf*: the first unknown goes in the first row, the second unknown goes in the second row, etc. Then calculate *Sf*,

*Sf* = *Jf*.times(*Vp*).times( *Jf*.transpose() )

***For the spline:***

There are two parallel processes that this code takes care of: calculating the uncertainties in the fractionation-correction of the unknowns, and calculating the uncertainty envelope for the spline to plot in the session view. Do these two tasks separately. In addition to the value of the smoothing spline evaluated at the times tInt (the times for the unknowns or the times for interpolation-plotting), this code also generates two matrices, Jfg and Jfgamma.

private double calculateInterpolatedSplineValue(double tInt, int index, double[] g, double[] gamma){

double splineFitYValue = //

((tInt - t.get( index )) \* g[index + 1]//

+ (t.get( index + 1 ) - tInt) \* g[index]) / h[index]//

- (tInt - t.get( index )) \* (t.get( index + 1 ) - tInt) / 6.0//

\* ((1.0 + (tInt - t.get( index )) / h[index]) \* gamma[index + 1] //

+ (1.0 + (t.get( index + 1 ) - tInt) / h[index]) \* gamma[index]);

Jfg(k,index) = (t.get( index + 1 )- tInt)/h[index];

Jfg(k,index+1) = (tInt - t.get( index ) )/h[index];

Jfgamma(k,index) = (tInt - t.get( index ))\*(tInt-t.get( index + 1 ))\*(h[index]-tint+t.get( index + 1 )) /(6\*h[index]);

Jfgamma(k,index+1) = (tInt - t.get( index ))\*(tint-t.get( index + 1 ))\*(h[index]+ tInt - t.get( index ) ) /(6\*h[index]);

return splineFitYValue;

}

Jgammag = R.solve( Q.transpose() )

When you calculate *Jgammag*, it will have s-2 rows. Add a row of zeros to the top and bottom of Jgammag, so that it has s rows.

Jf = Jfg.plus( Jfgamma.times(Jgammag) )

Re-create Jf each time a point is discarded.

Sf = Jf.times(hatMatrix).times(Ss).times(hatMatrix.transpose()).times(Jf.transpose())

*Sf* has *u* rows and *u* columns, where *u* is the number of unknowns (either unknown samples or unknown values that we’re interpolating over, e.g. 2000).

*stdunct* = sqrt( diag( *Sf* ) );

In other words, the value of stdunct for the *i*-th unknown is the square root of the element of *Sf* in (row *i*, column *i*).

To plot the uncertainty envelope for the spline, add and subtract twice the value of the appropriate element of *stdunct* from the appropriate element of the spline evaluation.

**9C. Calculate overdispersion, if required**

Use the Levenberg-Marquard code we have worked on to calculate the mean, line, and exponential fits with overdispersion.

In the fit results/slider panel to the right of the session plots, you are using the bottom of each box beside a ratio for an overdispersion check-box. Beside the overdispersion, display the square root of the overdispersion that you have solved for (od in the code). You can give this parameter the name ξ (Greek letter xi). For instance, display “ ξ = 0.02 “ for a calculated overdispersion of od = 0.0004 = ξ2. Two significant figures will be enough for ξ.

For the spline:

There is no way to uniquely determine the appropriate overdispersion for the spline fit. Instead, the user will have to choose it based on what (s)he feels is the appropriate smoothness for the data. When the spline fit is selected, default to no overdispersion (even if it has been selected already for a previous parametric fit). In the spline fit, beside the overdispersion checkbox, provide three text boxes for user input, labeled “Start”, “Stop”, and “Step Size.” These will designate a set of trial values for ξ.

For each trial value of ξ in ascending order, square it to calculate ξ2 (otherwise known as the overdispersion or *od*), add this value to each term on the diagonal of the measured covariance matrix that is usually input to the spline, and send this new covariance matrix *Sod* to the spline fitting function, leaving all other inputs unchanged.

Sod = S + ξ2 \* eye(s)

Once the spline fitting function is done, calculate

logRoughness = log( g.transpose().times( K.times(g) ) / s )

where g is the vector of spline values and K is calculated at the beginning of the spline fit (along with Q, R, and h, K does not change depending on the y values and their uncertainties). log() is the natural log and s is the number of standards.

If the logRoughness is greater than -41, then save off all spline fit artifacts for future use:

* alphaBest, g, gamma, and redChiSqBest (the final value of X2)

and then proceed to the next larger value of ξ. If the logRoughness is less than the threshold above, then the spline resembles a line, and you can stop increasing the value of ξ.

Once either the “Stop” value has been reached or the threshold reached, create a user-movable slider bar that lets the user select between the discrete valid values of ξ calculated above. Because the spline fit artifacts have been saved, plotting should happen quickly.

A button should call up a plot of logRoughness vs. ξ. That is, put logRoughness on the y-axis and ξ on the x-axis. Let the user drag a marker along the values of this function (instead of the slider bar), so that when you drag the marker to a specific value of ξ, it stays on the function value of logRoughness, and the spline plot is changed to reflect the new value of ξ. Plotting this graph in real time would be an effective ‘progress bar’ substitute.

**10. Fractionation-correct unknowns, propagate uncertainties**

For each of the three ratios used (206/207, 206/238, 208/232), calculate Φ(t), the inter-isotope or inter-element fractionation as a function of t, the session time, using the equation [here](#h00insamra7e), from the fit to the standard log-ratios. The uncertainty in Φ(t) is exactly the same as the uncertainty in the logratio fit function.

To fractionation-correct the unknowns, calculate the corrected log-ratio of the unknown as

Φ = log(stdtrue) - log(stdmeas)

log(unkcorr) = log(unkmeas) + Φ

log(unkcorr) = log(unkmeas) + log(stdtrue) - log(stdmeas(t))

where corr denotes a corrected (log)-ratio. The stdtrue ratio is provided in the mineral standard model. The log(stdmeas(t)) is the interpolated value of the standard log-ratio at the time in the session at which the unknown was measured.

Note: use upperPhi in data dictionary for Φ

**11. Propagate systematic uncertainty into intercept uncertainties of unknowns**

For each of the three ratios we’re using (206/207, 206/238, 208/232), create a new matrix called *Su* with *u* rows and *u* columns, where *u* is the number of unknowns measured. For this step, lump all of the unknowns together, even if they have been divided into aliquots by the user in the initial setup.

Begin by populating the diagonal of *Su* with with the variances from each of the intercept fit functions for each of the unknowns. If the session fit includes overdispersion, add this term to each diagonal entry as well (it is the same for all, and is referred to as *od* or pod(m) in our L-M code -- not ξ).

For the downhole method, *Su* contains the variances from each of the weighted means from the downhole-corrected unknowns.

Add to *Su* the *u*-by-*u* matrix *Sf* that you calculated [here](#p10k4r9fi9mz) (using the session times from the unknowns).

Finally, if both isotopes in the ratio have been measured on the same ion counter, create *Suod*, a matrix containing only off-diagonal elements, with the same formula as that for Ssod. Just use the *dLrInt\_dDt* vector of length *u* that you created for the unknowns.

Suod = *deadTimeOneSigmaAbs*^2 \* *dLrInt\_dDt*.times( *dLrInt\_dDt*.transpose() )

As with the standards, change all the diagonal entries of *Suod* to zero, then add it into Su.

The matrix *Su*, along with the three vectors of fractionation-corrected log-ratios for the 206/207, 206/238, and 208/232 are the output of the ‘Tripoli’ side of this program, and should be pushed to ‘Redux’.

The diagonal of Su contains the variances for the log-ratio of each of the fractions. This number should take the place of the sqrt(sum(intercept + fit + od)) calculation that you had implemented before.

After finishing the calculation for **Su**, which includes the addition of **Sf** above, replace each diagonal entry of **Sfci** with the appropriate term from **Su**, corresponding to the fraction for that **Sfci** and to each of the fractionation-corrected intercepts. Add a row and column at the end of Sfci that, on the diagonal, contains the variance for the log-ratio lr238\_235s, computed as (r238\_235sOneSigmaAbs/r238\_235s)^2 . The resulting matrix, **SfciTotal**, has four rows and four columns for no Pbc correction or Scheme A, and seven rows and columns for Scheme B. <Add in fractionation-correction uncertainty here later. For now, ignore the X/204 ratios> Call this new matrix, with the diagonals replaced, **SfciTotal**. There is one **SfciTotal** for each fraction.

Next, assemble the matrix **JSomeAllRatios**, which will convert the three ratios we use in the session fits to the five ratios (206/238, 207/235, 207/206, 238/206, 208/232) used in Conventional and T-W Concordia plots, respectively, followed by the measured 206/204, 207/204, 208/204, and finally the sample 238/235. These are seven columns and eight rows in JSomeAllRatios.

JSomeAllRatios = [0 1 0 0 0 0 0; -1 1 0 0 0 0 1; -1 0 0 0 0 0 0; 0 -1 0 0 0 0 0; 0 0 1 0 0 0 0; 0 0 0 1 0 0 0 ; 0 0 0 0 1 0 0; 0 0 0 0 0 1 0]; Where the lines above are rows separated by semicolons.

Calculate **SfciTotalAll** = JSomeAllRatios \* SfciTotal \* Transpose(JSomeAllRatios)

Note: the transpose in the equation for SfciTotalAll is now in the correct position. Next, assemble the matrix Jlrr, which is an eight by eight matrix that contains the the following ratios on the diagonal. These are ratios, not log-ratios (they have already been exponentiated). **Jlrr** = diag( r206\_238fc, r207\_235fc, r207\_206fc, r238\_206fc, r208\_232fc, r206\_204fc, r207\_204fc, r208\_204fc ).

Calculate a covariance matrix for all the ratios you need for a fraction, SrAll:

SrAll = Jlrr\*SfciTotalAll\*Transpose(Jlrr)

**SrAll** (ratios All) is a eight-by-eight matrix per fraction that contains all the uncertainties and covariances (used to calculate rhos) needed to make Conventional and Tera-Wasserburg plots *for non-common-Pb corrected ratios*. These are the real (not fake) covariance/correlation terms.

Rho = sigmaxy/sqrt(sigma^2\_x \* sigma^2\_y) where sigmaxy is the off-diagonal term in the covariance matrix, and sigma^2\_x and sigma^2\_y are variances that are on the diagonal. rhoxy = (x,y) / sqrt( (x,x)\*(y,y) ).

If Scheme B of Pbc correction: ~~Crop the last three rows and columns of SrAll and save it off, call it~~ **~~SrAll204~~**~~.~~ We will use all of **SrAll** for Scheme B.

**Su** can be used to calculate the data table output (un-logged) ratio uncertainty using the scheme that we devised:

Determine the upper and lower bounds of the 95% confidence interval of the log-ratio by adding and subtracting the 2σ uncertainty of the mean (all in log-ratio space). Next, convert the mean and the upper and lower bounds into isotope ratios (ie exponentiate), then calculate the differences between the mean and the upper bound and the mean and the lower bound. If the first two nonzero digits of these distances are the same, then they represent the ±2σ uncertainty in the date to our required numeric precision. If the two distances are different within our numerical precision, then they define an asymmetric 95% confidence interval, and should be reported separately as +2σ and -2σ.

**12. Common Pb / Pb Loss correction schemes**

Add a new tab to the project manager called “Common Pb / Pb Loss Correction”, with the same basic setup as the present system that distributes fractions between ‘sample’, ‘primary std’, ‘secondary std’, etc. Instead of choosing between these categories, however, we want to distribute the *samples* (not standards) chosen in the previous tab between different common Pb / Pb loss correction schemes. All samples should default to the ‘none’ category (ie no Pbc correction), with the choice to add a new correction from a constrained list (add a new box) and drag fractions into it from the default ‘none’ category. For now, we will support three correction schemes, in addition to ‘none’, called

A. 206Pb-207Pb-238U, Initial Pbc = r207\_206c, assumed concordant. Yields a date only (t Pbc corrected)

B. 204Pb-206Pb-207Pb-238U, Initial Pbc = r206\_204c and r207\_204c. Works like common Pb correction in TIMS Redux, i.e. you get all three dates (206/238, 207/235, 207/206) and isotopic ratios (206/238, 207/235, and 207/206). You get six dates (three for B1, three for B2 with S-K Solve)

C. (+/- 204Pb) - 206Pb-207Pb-208Pb-238U-232Th, U-Pb and U-Th assumed concordant, Yields One Date Only

D. Anderson (2002), 206Pb-207Pb-208Pb-238U-232Th, tPbLoss

In this way, the user can distribute the sample fractions between the options of providing no Pbc correction and the options above. Unlike the fraction distribution between sample and standard categories, however, these choices can be altered later (e.g. after data reduction). Choices made before analysis can be changed without re-reducing all data (1-11 above), and we will look to provide other, more convenient ways of changing the correction scheme (e.g. from the ‘corrected (log-)ratios’ set of plots or interactively in the concordia plot.

Furthermore, the list of isotopes that accompanies A-C above is also a required list of isotopes for the correction to be made. Therefore, for instance, option C should be disabled for fractions without both 208Pb and 232Th data. This could perhaps be accomplished by marking them in red with a cross-hatch when they are present in a box with this option chosen.

Each scheme A-C above requires an additional choice of ‘Pbc IC Model’, which are parallel to the models we use for ID-TIMS Redux. Choices available at present are ‘Stacey-Kramers’, and any user-defined models. There should be a button here to create a new Pbc model (including uncertainties).

If the Stacey-Kramers model is chosen, provide a place to input the relative uncertainties and correlation coefficients (just like ID-TIMS). Also, provide a 2-way radio-button option to provide the numerical “Estimated S-K Pbc Date” or “Solve”. Additional calculations for this “S-K Solve” option are given below.

Protocol = Scheme (A-D) + Pbc Model (user-generated or ‘S-K Solve’).

Many of the methods will require a Newton-Raphson solver, like the type we use for solving for Age207206. The format general format uses a unique t0, f0, and fp for each system

Newton-Raphson Scalar Solver:

tnew = <evaluate t0 expression, a function of measured parameters>

**Do**

t = tnew

f0 = <evaluate f0 expression, a function of measured parameters and t>

fp = <evaluate fp expression, a function of measured parameters and t>

tnew = t - f0(t)/fp(t)

**While** abs(t-tnew)/tnew > t\_relativeTolerance

We will usually use a t\_relativeTolerance of 1e-7.

**A1. 206Pb-207Pb-238U, assumed concordant, user-generated**

From the output of Section 10 above, calculate

*r238\_206fc* = 1/exp(*lr206\_238r*)

and

*r207\_206fc* = exp(*lr207\_206r*)

Determine *r207\_206c* (= *r207\_204c*/*r206\_204c*) from the Initial Pb model, or allow the user to input it manually for each fraction without having to create and save the model.

Solving this system requires the Newton-Raphson scalar solver, with:

*if r207\_206c < 1.11 (note: this is typical range for r207\_206c)*

***t0***= (r207\_206fc-r207\_206c)/r238\_206fc \* (6384.9923\*sqrt(r207\_206c-0.80183883) - 9063.8137) \* 10^6

*else if 1.11 <= r207\_206c < 11.2 (note: unusual, moderately radiogenic range)*

***t0*** = (r207\_206fc-r207\_206c)/r238\_206fc \* (1349.6847\*log(r207\_206c - 0.78795528) -2.5240584) \* 10^6

else (if r207\_206c >= 11.2) *(note: unusual, highly radiogenic range)*

***t0*** = (r207\_206fc-r207\_206c)/r238\_206fc \* (109.56243\*sqrt(r207\_206c-10.725608) - 653.360321) \* 10^6

***f0****(lambda235, lambda238, r238\_235s, r207\_206c, r238\_206fc, r207\_206fc, t)* =

(r238\_206fc - r207\_206c\*r238\_235s + r207\_206fc\*r238\_235s - r238\_206fc\*exp(lambda235\*t) - r207\_206c\*r238\_206fc\*r238\_235s + r207\_206c\*r238\_206fc\*r238\_235s\*exp(lambda238\*t)) / r238\_235s

***fp****(lambda235, lambda238, r238\_235s, r207\_206c, r238\_206fc, t)*  = -(lambda235\*r238\_206fc\*exp(lambda235\*t) - lambda238\*r207\_206c\*r238\_206fc\*r238\_235s\*exp(lambda238\*t))/r238\_235s

**A2. 206Pb-207Pb-238U, assumed concordant, S-K Solve**

This system also requires the Newton-Raphson scalar solver. The functions (f0, fp, t0) to use in the solver depend on the measured data (r238\_206fc and r207\_206fc):

First, calculate the 207Pb/206Pb and 206Pb/238U dates using the measured r207\_206fc and the reciprocal of r238\_206fc. If the 207Pb/206Pb date is less than the 206Pb/238U date, then use the 207Pb/206Pb date and uncertainty. Else, if the 206Pb/238U date is within uncertainty of the 207Pb/206Pb date, then use whichever is more precise. If both of these tests fail, then calculate a common-Pb corrected date:

If r207\_206fc > 1.1651 - 0.63334\*r238\_206fc

***t0***(r238\_206fc, r207\_206fc) = (7000/( 1.0592903\*(r238\_206fc + 0.013921017)) + 4.0914451 + (-7000/(0.88941914\*(r238\_206fc + 0.12713499)) - 5.5997270)\*r207\_206fc ) ~~\*10^6~~

***f0***(lambda235, lambda238, r238\_235s, r238\_206fc, r207\_206fc, t) =

(19220472\*r207\_206fc - 22402053\*r238\_206fc - 121750\*exp(3700\*lambda235) + 121750\*exp(lambda235\*t) + 16786890\*r207\_206fc\*exp(3700\*lambda238) - 121750\*r238\_206fc\*exp(3700\*lambda235) - 16786890\*r207\_206fc\*exp(lambda238\*t) + 121750\*r238\_206fc\*exp(lambda235\*t) + 22402053\*r238\_206fc\*exp(lambda238\*t) - 121750\*r238\_206fc\*exp(lambda235\*t).\*exp(lambda238\*t) + 121750\*r238\_206fc\*exp(3700\*lambda235)\*exp(lambda238\*t) - 22402053)./(6894\*(2435\*exp(3700\*lambda238) - 2435\*exp(lambda238\*t) + 2788)) - (r238\_206fc\*(exp(lambda235\*t) - 1))/r238\_235s

***fp***(lambda235, lambda238, r238\_235s, r238\_206fc, r207\_206fc, t) =

(121750\*lambda235\*exp(lambda235\*t) + 121750\*lambda238\*r238\_206fc\*exp(3700\*lambda235 + lambda238\*t) - 121750\*lambda235\*r238\_206fc\*exp(t\*(lambda235 + lambda238)) - 121750\*lambda238\*r238\_206fc\*exp(t\*(lambda235 + lambda238)) - 16786890\*lambda238\*r207\_206fc\*exp(lambda238\*t) + 121750\*lambda235\*r238\_206fc\*exp(lambda235\*t) + 22402053\*lambda238\*r238\_206fc\*exp(lambda238\*t))/(6894\*(2435\*exp(3700\*lambda238) - 2435\*exp(lambda238\*t) + 2788)) + (2435\*lambda238\*exp(lambda238\*t)\*(19220472\*r207\_206fc + 16786890\*r207\_206fc\*exp(3700\*lambda238) + exp(lambda235\*t)\*(121750\*r238\_206fc + 121750) - 121750\*r238\_206fc\*exp(t\*(lambda235 + lambda238)) - 16786890\*r207\_206fc\*exp(lambda238\*t) + 121750\*r238\_206fc\*exp(3700\*lambda235 + lambda238\*t) - exp(3700\*lambda235)\*(121750\*r238\_206fc + 121750) + 44804106\*r238\_206fc\*exp((lambda238\*t)/2)\*sinh((lambda238\*t)/2) - 22402053))/(6894\*(2435\*exp(3700\*lambda238) - 2435\*exp(lambda238\*t) + 2788)^2) - (lambda235\*r238\_206fc\*exp(lambda235\*t))/r238\_235s

else

**t0** = 4.3\*10^~~9~~ 3

**f0** = @(lambda235, lambda238, r238\_235s, r238\_206fc, r207\_206fc, t) =

(32081229\*r207\_206fc - 35483418\*r238\_206fc - 179750\*exp(4570\*lambda235) + 179750\*exp(lambda235\*t) + 24783930\*r207\_206fc\*exp(4570\*lambda238) - 179750\*r238\_206fc\*exp(4570\*lambda235) - 24783930\*r207\_206fc\*exp(lambda238\*t) + 179750\*r238\_206fc\*exp(lambda235\*t) + 35483418\*r238\_206fc\*exp(lambda238\*t) - 179750\*r238\_206fc\*exp(lambda235\*t).\*exp(lambda238\*t) + 179750\*r238\_206fc\*exp(4570\*lambda235)\*exp(lambda238\*t) - 35483418)/(3447\*(7190\*exp(4570\*lambda238) - 7190\*exp(lambda238\*t) + 9307)) - (r238\_206fc\*(exp(lambda235\*t) - 1))/r238\_235s

**fp** = @(lambda235, lambda238, r238\_235s, r238\_206fc, r207\_206fc, t) =

(179750\*lambda235\*exp(lambda235\*t) + 179750\*lambda238\*r238\_206fc\*exp(4570\*lambda235 + lambda238\*t) - 179750\*lambda235\*r238\_206fc\*exp(t\*(lambda235 + lambda238)) - 179750\*lambda238\*r238\_206fc\*exp(t\*(lambda235 + lambda238)) - 24783930\*lambda238\*r207\_206fc\*exp(lambda238\*t) + 179750\*lambda235\*r238\_206fc\*exp(lambda235\*t) + 35483418\*lambda238\*r238\_206fc\*exp(lambda238\*t))/(3447\*(7190\*exp(4570\*lambda238) - 7190\*exp(lambda238\*t) + 9307)) + (7190\*lambda238\*exp(lambda238\*t)\*(32081229\*r207\_206fc + 24783930\*r207\_206fc\*exp(4570\*lambda238) + exp(lambda235\*t)\*(179750\*r238\_206fc + 179750) - 179750\*r238\_206fc\*exp(t\*(lambda235 + lambda238)) - 24783930\*r207\_206fc\*exp(lambda238\*t) + 179750\*r238\_206fc\*exp(4570\*lambda235 + lambda238\*t) - exp(4570\*lambda235)\*(179750\*r238\_206fc + 179750) + 70966836\*r238\_206fc\*exp((lambda238\*t)/2)\*sinh((lambda238\*t)/2) - 35483418))/(3447\*(7190\*exp(4570\*lambda238) - 7190\*exp(lambda238\*t) + 9307)^2) - (lambda235\*r238\_206fc\*exp(lambda235\*t))/r238\_235s

**B General comments**

If the user selects option B, then add the following ratios to the ‘Fit Each Unknown’ tab of the ‘Intercept’ and ‘Downhole’ methods: 206Pb/204Pb, 207Pb/204Pb, and 208Pb/204Pb.

**B1. 204Pb-206Pb-207Pb-238U, Initial Pbc = r206\_204c and r207\_204c. No assumption of concordance, no SK-Solve.**

For each fraction, input a value for the 206Pb/204Pb, 207Pb/204Pb and optionally a 208Pb/204Pb with the heading “common Pb IC”, as you have in place already. These correspond to r206\_204c, r207\_204c, and r208\_204c in our Redux data dictionary. The third ratio is useful for calculating Th-Pb dates, as you are currently developing capabilities for with Rioux and Cottle.

For uncertainty inputs, as in Section 12.1 A1, there are two levels of uncertainty we need to deal with, the fraction-to-fraction *variability* and the overall *systematic* uncertainty. Each of the two levels gets a separate covariance matrix, which we will assemble from a variable number of input uncertainties and correlation coefficients. Under the heading “fraction-to-fraction variability”, make columns to input the 206Pb/204Pb (1σ abs), 207Pb/204Pb (1σ abs), and if the user input a 208Pb/204Pb ratio, the 208Pb/204Pb (1σ abs). For the correlation coefficients, make a column to input the ρ206Pb/204Pb-207Pb/204Pb, and if the user input a 208Pb/204Pb ratio, two more columns for the ρ206Pb/204Pb-208Pb/204Pb and ρ207Pb/204Pb-208Pb/204Pb. Finally, under the heading “systematic uncertainty”, mimic the exact same columns for the uncertainties (1σ abs and rho values). A radio button switching from absolute to percent uncertainties, like you have in Redux already, would be useful here as well, along with the existing “fill down” buttons.

Jim: new equations for 204-based Pbc correction start here:

To perform the common Pb correction, calculate the following quantities:

First, add the 206/204, 207/204, and 208/204 ratios to the ‘Fractionation-correct Unknowns’ tab.

Let upperPhi\_r206\_207 be the data dictionary term for the variable Φ(t) = log(stdtrue) - log(stdmeas) calculated for the **standard** 206/207 measurements, interpolated at the time of the present unknown.

upperPhi\_r206\_204 = upperPhi\_r206\_207 \* log(*m206*/*m204*)/log(*m206*/*m207*)

upperPhi\_r207\_204 = upperPhi\_r206\_207 \* log(*m207*/*m204*)/log(*m206*/*m207*)

upperPhi\_r208\_204 = upperPhi\_r206\_207 \* log(*m208*/*m204*)/log(*m206*/*m207*)

where *m204*, *m206*, *m207*, and *m208* are the atomic masses of 204Pb, 206Pb, 207Pb, and 208Pb, respectively.

Now fractionation-correct the measured 206/204, 207/204, and 208/204 mean/intercepts for the current unknown using the formula

log(unkcorr) = log(unkmeas) + upperPhi\_rX

where X is the ratio of interest.

You should now have the fractionation-corrected 206/204, 207/204, and 208/204 log-ratios. Use the exponential function to transform these into isotope ratios, and you already also additionally have the fractionation-corrected **unknown** 206/207, 206/238, and 208/232 ratios from Section 10.

Now perform the following calculations for Pbc-corrected ratios (rX\_X\_PbcCorr)

*r207\_206\_PbcCorr* = (r207\_204fc - r207\_204c)/(r206\_204fc - r206\_204c)

*r206\_238\_PbcCorr* = r206\_238fc \* (1 - r206\_204c / r206\_204fc)

*r238\_206\_PbcCorr* = 1/r206\_238\_PbCorr *(note: no date for this ratio, used for T-W plot)*

*r208\_232\_PbcCorr* = r208\_232fc \* (1 - r208\_204c / r208\_204fc)

*r207\_235\_PbcCorr* = r206\_238fc \* r238\_235s / r206\_207fc \*

(1 - r207\_204c / r207\_204fc)

**B2.**  **204Pb-206Pb-207Pb-238U, SK-Solve.** As for A2, there is no need to input the values of the 206Pb/204Pb, 207Pb/204Pb, and 208Pb/204Pb -- they are determined by the Stacey-Kramers algorithm, so these columns can be de-activated. Activate the the S-K uncertainty column. For B2, the user needs to choose a type of date that will be synchronized. Choices are 206Pb/238U, 207Pb/235U, and 208Pb/232Th. Default should be 206Pb/238U.

*Step 1. Calculate an isotope ratio, according to the date type choice above:*

For t < 3.7Ga (if r206\_238fc < 0.7753, if r207\_235fc < 37.2424, if r208\_232 < 0.2003)

*r206\_238\_PbcCorr* = (r206\_238fc\*((c11152 + c974\*exp(c37\*lambda238))/r206\_204fc - 1) - 1)/((c974\*r206\_238fc)/r206\_204fc - 1) - 1

*r207\_235\_PbcCorr* = (r207\_235fc\*((c12998 + (c974\*exp(c37\*lambda235))/c13788)/r207\_204fc - 1) - 1) / ((c974\*r207\_235fc)/(c13788\*r207\_204fc) - 1) - 1

*r208\_232\_PbcCorr* = (r208\_232fc\*((c3123 + c3684\*exp(c37\*lambda232))/r208\_204fc - 1) - 1) /((c3684\*r208\_232fc)/r208\_204fc - 1) - 1

For t >= 3.7Ga (else)

*r206\_238\_PbcCorr* = (r206\_238fc\*((c9307 + c719\*exp(c457\*lambda238))/r206\_204fc - 1) - 1)/((c719\*r206\_238fc)/r206\_204fc - 1) - 1

*r207\_235\_PbcCorr* = (r207\_235fc\*((c10294 + (c719\*exp(c457\*lambda235))/c13788)/r207\_204fc - 1) - 1) / ((c719\*r207\_235fc)/(c13788\*r207\_204fc) - 1) - 1

*r208\_232\_PbcCorr* = (r208\_232fc\*((c29487 + c3321\*exp(c457\*lambda232))/r208\_204fc - 1) - 1)/((c3321\*r208\_232fc)/r208\_204fc - 1) - 1

where c719 = 7.19; c9307 = 9.307; c13788 = 137.88; c10294 = 10.294; c3321 = 33.21; c29487 = 29.487; c974 = 9.74; c11152 = 11.152; c12998 = 12.998; c3684 = 36.84; c3123 = 31.23; c37 = 3.7E9, c457 = 4.57E9

*Step 2: calculate the pertinent isotopic date.*

t206\_238 = 1/lambda238\*log(r206\_238\_PbcCorr + 1)

t207\_235 = 1/lambda235\*log(r207\_235\_PbcCorr + 1)

t208\_232 = 1/lambda232\*log(r208\_232\_PbcCorr + 1)

If you used the t < 3.7 Ga (3.7\*10^9 years) branch above and the calculated date is greater than 3.7 Ga, then recalculate using the t > 3.7 Ga branch, and vice versa.

*Step 3: Use the Pbc-corrected date to calculate the S-K Pbc isotopic composition (206/204, 207/204, 208/204)*

Use the S-K model equations we already have in place (the original).

*Step 4: Uncertainty propagation*

Calculate r238\_206fc = 1/r206\_238fc (neither ratio is Pbc-corrected).

You now have all the parameters you have in B1, and can use the same uncertainty propagation equations as B1. For the -fc ratios, use the measured (and fractionation-corrected) ratios you used on the right-hand-side of equations in Step 1, not the Pbc-corrected ratios.

**12.1 Uncertainty propagation for Pbc correction**

**A1.** Full uncertainty propagation requires two different levels of uncertainty to be input: a fraction-to-fraction *variability* and an overall *systematic* uncertainty. Call the first \_OneSigmaAbsVar and the second \_OneSigmaAbsSys. In the long term, we should look to change the model for everything so that it contains a separate uncertainty for both of these categories.

For each Pbc-corrected fraction, calculate the following derivatives:

df0\_dR238\_206fc = -(exp(lambda235\*t) + r76c\*r85s - r76c\*r85s\*exp(lambda238\*t) - 1)/r85s

df0\_dR207\_206fc = 1

df0\_dR207\_206c = r238\_206fc\*exp(lambda238\*t) - r238\_206fc - 1

df0\_dR238\_235s = (r238\_206fc\*(exp(lambda235\*t) - 1))/(r238\_235s^2)

df0\_dLambda238 = r207\_206c \* r238\_206fc\*t\*exp(lambda238\*t)

df0\_dLambda235 = -(r238\_206fc \* t \* exp(lambda235\*t))/r238\_235s

df0\_dt = -(r238\_206fc\*(lambda235\*exp(lambda235\*t) - lambda238\*r207\_206c\*r238\_235s\*exp(lambda238\*t)))/r238\_235s

Calculate the derivatives of t with respect to the variables used in its calculation using the inverse function theorem: dt\_dx = -df0\_dx / df0\_dt

dt\_dR238\_206fc = -df0\_dR238\_206fc / df0\_dt

dt\_dR207\_206fc = -df0\_dR207\_206fc / df0\_dt

dt\_dR207\_206c = -df0\_dR207\_206c / df0\_dt

dt\_dR238\_235s = -df0\_dR238\_235s / df0\_dt

dt\_dLambda238 = -df0\_dLambda238 / df0\_dt

dt\_dLambda235 = -df0\_dLambda235 / df0\_dt

**A2.** For each Pbc-corrected fraction, calculate the following:

*df0\_dR238\_206fc* = - (exp(lambda235\*t) - 1)/r238\_235s - (121750\*exp(3700\*lambda235) - 121750\*exp(lambda235\*t) - 22402053\*exp(lambda238\*t) - 121750\*exp(3700\*lambda235)\*exp(lambda238\*t) + 121750\*exp(lambda235\*t)\*exp(lambda238\*t) + 22402053)/(16786890\*exp(3700\*lambda238) - 16786890\*exp(lambda238\*t) + 19220472)

*df0\_dR207\_206fc* = 1

df0\_dR238\_235s = (r238\_206fc\*(exp(lambda235\*t) - 1))/r238\_235s^2

*df0\_dLambda238* = (((9009500\*exp(3700\*lambda238) + 9009500\*r238\_206fc\*exp(3700\*lambda238))\*(121750\*exp(3700\*lambda235) - 121750\*exp(lambda235\*t) + 22402053))/6894 - (exp(lambda238\*t)\*(121750\*exp(3700\*lambda235) - 121750\*exp(lambda235\*t) + 22402053)\*(2435\*t + 9009500\*r238\_206fc\*exp(3700\*lambda238) - 353\*r238\_206fc\*t - 2435\*r238\_206fc\*t\*exp(3700\*lambda238)))/6894)/(2435\*exp(3700\*lambda238) - 2435\*exp(lambda238\*t) + 2788)^2

*df0\_dLambda235* = - (450475000\*exp(3700\*lambda235) + 450475000\*r238\_206fc\*exp(3700\*lambda235) - 121750\*t\*exp(lambda235\*t) - 450475000\*r238\_206fc\*exp(3700\*lambda235)\*exp(lambda238\*t) - 121750\*r238\_206fc\*t\*exp(lambda235\*t) + 121750\*r238\_206fc\*t\*exp(lambda235\*t)\*exp(lambda238\*t))/(16786890\*exp(3700\*lambda238) - 16786890\*exp(lambda238\*t) + 19220472) - (r238\_206fc\*t\*exp(lambda235\*t))/r238\_235s

df0\_dt = (121750\*lambda235\*exp(lambda235\*t) + 121750\*lambda238\*r238\_206fc\*exp(3700\*lambda235 + lambda238\*t) - 121750\*lambda235\*r238\_206fc\*exp(t\*(lambda235 + lambda238)) - 121750\*lambda238\*r238\_206fc\*exp(t\*(lambda235 + lambda238)) - 16786890\*lambda238\*r207\_206fc\*exp(lambda238\*t) + 121750\*lambda235\*r238\_206fc\*exp(lambda235\*t) + 22402053\*lambda238\*r238\_206fc\*exp(lambda238\*t))/(6894\*(2435\*exp(3700\*lambda238) - 2435\*exp(lambda238\*t) + 2788)) + (2435\*lambda238\*exp(lambda238\*t)\*(19220472\*r207\_206fc + 16786890\*r207\_206fc\*exp(3700\*lambda238) + exp(lambda235\*t)\*(121750\*r238\_206fc + 121750) - 121750\*r238\_206fc\*exp(t\*(lambda235 + lambda238)) - 16786890\*r207\_206fc\*exp(lambda238\*t) + 121750\*r238\_206fc\*exp(3700\*lambda235 + lambda238\*t) - exp(3700\*lambda235)\*(121750\*r238\_206fc + 121750) + 44804106\*r238\_206fc\*exp((lambda238\*t)/2)\*sinh((lambda238\*t)/2) - 22402053))/(6894\*(2435\*exp(3700\*lambda238) - 2435\*exp(lambda238\*t) + 2788)^2) - (lambda235\*r238\_206fc\*exp(lambda235\*t))/r238\_235s

Calculate the derivatives of t with respect to the variables used in its calculation using the inverse function theorem: dt\_dx = -df0\_dx / df0\_dt

dt\_dR238\_206fc = -df0\_dR238\_206fc / df0\_dt

dt\_dR207\_206fc = -df0\_dR207\_206fc / df0\_dt

dt\_dR238\_235s = -df0\_dR238\_235s / df0\_dt

dt\_dLambda238 = -df0\_dLambda238 / df0\_dt

dt\_dLambda235 = -df0\_dLambda235 / df0\_dt

**For A1 and A2:**

To calculate the uncertainty in t, place the oneSigmaVarAbs^2 variances for the variables r238\_206fc, r207\_206fc, r207\_206c, and r238\_235s on the diagonal of an empty 4x4 matrix of zeros (a diagonal covariance matrix--the off-diagonal terms may be filled later on), called *Srpbc*. For r207\_206c, use the oneSigmaVarAbs. Also, assemble the derivatives of t with respect to the same variables, in the same order, in a 4-row by 1-column vector called *Jtrpbc*. Calculate the (scalar) variance in t as

*Su*(i,i) = transpose(Jtrpbc) \* Srpbc \*Jtrpbc.

~~For A2, do this with a 3x3 covariance matrix and a 3-row Jacobian matrix using only r238\_206fc, r207\_206fc, and r238\_235s.~~

For A2, use the above algorithm to calculate t, the Stacey-Kramers date, then use this date to calculate r207\_206c using the usual Stacey-Kramers algorithm. For the uncertainty propagation, use the user-input S-K uncertainty ~~and correlation coefficient,~~ and follow the uncertainty propagation instructions for A1 above.

\*\*\* I am stopping here for now - Jim \*\*\*

For weighted mean calculations, this matrix product is placed in the appropriate diagonal location in the matrix Su of analytical uncertainties for the unknowns. Recall that Su has a row and column for each unknown, so that the (scalar) variance in the Pbc-corrected date for the fourth unknown goes in the fourth row and fourth column of Su.

For adding in the systematic uncertainties for the decay constants, use the same procedure as in 13D below, except using the derivatives of t with respect to lambda238 and lambda235 above. Use these to assemble *JDateLambda*.

We are adding a new level of systematic uncertainty with the systematic uncertainty option for r207\_206c.

For A1, assemble a row vector called *JDatePbcIC* with the value of every unknown’s dt\_dR207\_206c as the elements. Calculate *SuPbcIC* = transpose(*JDatePbcIC*)\* *r207\_206cOneSigmaSysAbs* \**JDatePbcIC*, a square matrix with the same number of rows and columns as the number of unknowns. This matrix should be added to Su (described above) to calculate the analytical+Pbc weighted mean uncertainty, using the usual algorithm for a weighted mean uncertainty with a combined analytical + systematic covariance matrix, which looks like, from 13.B.A, sqrt( 1 / (onesv.transpose().times( SuX.solve(onesv) ) ) ).

Explicit instructions here for calculating weighted means with different levels of systematic uncertainty.

**B1. Uncertainty Propagation**

dR76pbcc\_\_dR207\_204fc = -1/(r206\_204c-r206\_204fc)

dR76pbcc\_\_dR207\_204c = 1/(r206\_204c-r206\_204fc)

dR76pbcc\_\_dR206\_204fc = (r207\_204c-r207\_204fc)/(r206\_204c-r206\_204fc)^2

dR76pbcc\_\_dR206\_204c = -(r207\_204c-r207\_204fc)/(r206\_204c-r206\_204fc)^2

dR68pbcc\_\_dR206\_204fc = (r206\_204c\*r206\_238fc)/r206\_204fc^2

dR68pbcc\_\_dR206\_204c = -r206\_238fc/r206\_204fc

dR68pbcc\_\_dR206\_238fc = 1 - r206\_204c/r206\_204fc

dR86pbcc\_\_dR206\_204fc = -(r206\_204c\*r238\_206fc)/(r206\_204c - r206\_204fc)^2

dR86pbcc\_\_dR206\_204c = (r206\_204fc\*r238\_206fc)/(r206\_204c - r206\_204fc)^2

dR86pbcc\_\_dR238\_206fc = -r206\_204fc/(r206\_204c - r206\_204fc)

dR82pbcc\_\_dR208\_204fc = (r208\_204c\*r208\_232fc)/r208\_204fc^2

dR82pbcc\_\_dR208\_204c = -r208\_232fc/r208\_204fc

dR82pbcc\_\_dR208\_232fc = 1-r208\_204c/r208\_204fc

~~dR75pbcc\_\_dR207\_204fc = (r207\_204c\*r238\_235s\*r206\_238fc)/(r206\_207fc\*r207\_204fc^2)~~

~~dR75pbcc\_\_dR207\_204c = -(r238\_235s\*r206\_238fc)/(r206\_207fc\*r207\_204fc)~~

~~dR75pbcc\_\_dR206\_238fc = -(r238\_235s\*(r207\_204c/r207\_204fc-1))/r206\_207fc~~

~~dR75pbcc\_\_dR206\_207fc = (r238\_235s\*r206\_238fc\*(r207\_204c/r207\_204fc-1))/r206\_207fc^2~~

~~dR75pbcc\_dR238\_235s = -(r206\_238fc\*(r207\_204c/r207\_204fc-1))/r206\_207fc~~

dR75pbcc\_\_dR207\_204fc = (r207\_204c\*r207\_235fc) / r207\_204fc^2

dR75pbcc\_\_dR207\_204c = -r207\_235fc / r207\_204fc

dR75pbcc\_\_dR207\_235fc = 1 - r207\_204c / r207\_204fc

Assemble the matrix **JPbccs** using the derivatives calculated above, with rows corresponding to the variables r68pbcc, r75pbcc, r76pbcc, r86pbcc, r82pbcc, and with columns corresponding to the variables r206\_238fc, r207\_235fc, r207\_206fc, r238\_206fc, r208\_232fc, r206\_204fc, r207\_204fc, r208\_204fc, r206\_204c, r207\_204c, r208\_204c. The resulting matrix should have five rows and eleven columns.

Add three rows and three columns of zeros to the end of the matrix **SrAll**, so that it goes from eight-by-eight to eleven-by-eleven. Replace the last three rows and last three columns of the resulting matrix with the ‘Var’ version of the common Pb IC covariance matrix input in the ‘Common Pb Assignment’ window for that fraction. Calling this new expanded matrix **SrAll204**,

Multiply **Sfc204** = **JPbccs** \* **SrAll204** \* Transpose(**JPbccs**) = the Pbc uncertainties.

The matrix Sfc204 contains the uncertainties and uncertainty correlations for the Pbc-corrected 206Pb/238U, 207Pb/235U, 207Pb/206Pb, 238U/206Pb, and 208Pb/232Th ratios, respectively, to output to the data table and use for conventional or Tera-Wasserburg concordia plotting. Calculate the correlation coefficients to use for the data table and for plotting using [the usual formula](#xbgrzaoxnm2m). Use these ratios as well to calculate all Pbc-corrected dates.

**B2. 204Pb-based correction, S-K auto-solve.**

Use B1 uncertainty propagation, according to instructions here.

**13. Calculating dates and weighted means with systematic uncertainties (no Pbc correction)**

**A. Individual dates**

To calculate each date for a fraction, use the existing formulas in Redux and the exponentiated log-ratio output from ‘Tripoli’. For instance, for the 206/238 date, before calculating

age206\_238r = 1/lambda238 \* log(r206\_238r + 1)

calculate

r206\_238r = exp(lr206\_238r)

where lr206\_238r is the log-ratio output pushed across from Tripoli.

Calculate the uncertainty in an individual date in [the same way](#if1ty2xhvx4v) that you presently calculate the uncertainty in the 206/238 ratio from the log-ratio uncertainty, just calculate the dates corresponding to the fraction’s logratio +2σ and -2σ uncertainty.

**B. Weighted Mean Dates**

To calculate weighted mean dates, we will first calculate weighted mean log-ratios, then convert that mean into a date. Likewise the weighted mean uncertainty will be calculated the same way as the date uncertainty, calculating the +2σ and -2σ logratios, then calculating the equivalent dates, and measuring the distance from the mean date to the upper and lower bounds, etc. Do this for parts A through C below.

To calculate a weighted mean in U-Pb\_Redux, we at present have three levels of covariance matrix for the fractions. The first is an analytical covariance matrix, the second is a covariance matrix that contains the fractions' systematic uncertainties from the tracer, and another with contributions from decay constants. We also created a large jacobian matrix that relates the calculated date to the various systematic uncertainties.

For the new log-ratio math, we’re going to change this somewhat.

**A.** The matrix Su is the analytical covariance matrix analog. To calculate the analytical weighted mean, use the same math that we use to calculate a weighted mean with Redux presently:

lruX\_mean = onesv.transpose().times( *SuX*.solve(lruX) ) / (onesv.transpose().times( *SuX*.solve(onesv) ) )

lruX\_mean\_oneSigmaAnalytical = sqrt( 1 / (onesv.transpose().times( SuX.solve(onesv) ) ) )

where lruX is a column vector of the selected ratio of the selected fractions, SuX is the analytical covariance matrix made from the rows/columns of the matrix Su that correspond to these fractions, and onesv is a column vector of ones with the same number of rows as there are selected fractions.

To calculate the MSWD, first calculate a vector of residuals. Using the vector lruX of the (fully corrected output) log-ratios for the unknowns,

*lruX\_resids* = *lruX* - *lruX\_mean*

where lruX is a vector and lruX\_mean is a scalar that gets subtracted from every element of the scalar.

The MSWD can then be calculated as

*lruX\_MSWD* = 1/(*n*-1) \* *lruX\_resids*.transpose().times( *SuX*.solve( *lruX\_resids*) )

where *n* is the number of unknowns included in the calculation.

**B.** This uncertainty needs to be input in the Lab Constants, in a new tab/window, called “Inter-Standard Variability in Normalized Isotope Ratios (1σ)”. There should be three fields to input:

207/206 ratio variability: \_\_\_\_\_\_\_\_\_\_\_\_\_ %

206/238 ratio variability: \_\_\_\_\_\_\_\_\_\_\_\_\_ %

208/232 ratio variability: \_\_\_\_\_\_\_\_\_\_\_\_\_ %

where all three numbers are entered as a relative uncertainty, in percent. Save these as

*r207\_206rStdVariability\_OneSigmaPct*, *r206\_238rStdVariability\_OneSigmaPct*, and *r208\_232rStdVariability\_OneSigmaPct*, respectively.

To propagate these systematic uncertainties, divide each of the three terms above by 100. For each date, for instance the 207/206 date, square the result and add the resulting number to EACH of the elements of Su, on and off the diagonal. Note- for the 207/206, it doesn’t matter that the ratio we calculated was the 206/207 and the input is for 207/206. Call this new covariance matrix *SuInterStdX*, where the X is replaced by the isotope ratio of interest.

Finally, evaluate the weighted mean uncertainty for each log-ratio using [the same equation](#lnkqxdqe2e49) as the analytical weighted mean, using *SuInterStdX* instead of *SuX*.

**C.** The next level of systematic uncertainty for LA-ICP-MS is to propagate in the systematic uncertainty in the standard isotope ratio, from the EARTHTIME mineral standard model. This is simpler to do without using a Jacobian matrix.

~~First, calculate the one-sigma uncertainty in the log-ratio of the standard, in the same way that we calculate the uncertainty in a ratio from a log-ratio: Calculate the mean and ±1σ for the standard ratio in ratio space, then take the logarithm of the mean, upper, and lower bounds. Choose the largest distance between the mean and the upper/lower bound, and use this as an estimate of the one-sigma uncertainty in the log-ratio mean. Square the one sigma uncertainty to~~ calculate the variance in the presently calculated log-ratio (X/Y) of the standard, then add this number to ALL of the elements of *SuInterStdX*, on and off the diagonal.

Call this new covariance matrix *SuInterStdplusStdX*, and use it to calculate the third level of systematic uncertainty using [the same equation](#lnkqxdqe2e49) as the analytical uncertainty, but using *SuInterStdplusStdX instead of SuX.*

**D.** For the the decay constant uncertainties, we will calculate the derivative of the date with respect to any decay constants used. The rules here very similar to the rules used at present for the current systematic weighted mean.

1. Use the decay constant covariance matrix that we’ve been using in TIMS Redux, with the decay constants and decay constant covariance matrix from the EARTHTIME Physical Constants model. At present, these decay constants are: lambda230 lambda231 lambda232 lambda234 lambda235 and lambda238.

2. Evaluate the derivative of each date with respect to the relevant decay constants: For the 207/206 date, there are derivatives (already in Redux) for the 238U and 235U decay constants, for the 206/238 date, there is a derivative for the 238U decay constant, and the derivative for the 208/232 date with respect to the 232Th decay constant is

dAge208\_232r\_\_dLambda232 = -log(r208\_232r+1) / (lambda232^2)

where log() is the natural logarithm. All other derivatives are zero.

For each of the unknowns, generate a ‘weighted mean decay constant sensitivity matrix’, with a row for each unknown and a column for each of the lambdas. For the weighted mean date being evaluated, place the calculated derivative in the appropriate column. So for the 206/238 date, place the derivative of each unknown’s 206/238 date with respect to the 238 decay constant in the row corresponding to the fraction and the column corresponding to the decay constant. Call this matrix *JDateLambda*.

3. Next, use the covariance matrix for the unknowns’ log-ratios to calculate a date covariance matrix for the unknowns, of the same size. For each unknown log-ratio, calculate the derivative of the date with respect to the logratio:

dDate\_\_dLogratio = 1 / ( lambda23(8or2)\*( 1 + exp( -lrX ) ) )

where lrX is either the 206/238 or 208/232 ratio. For the 206/207 ratio:

dDate\_\_dLogRatio = (((-exp(lambda235\*t))\*lambda235 + exp((lambda235 + lambda238)\*t) \* (lambda235 - lambda238) + exp(lambda238\*t)\*lambda238) \* r238235s) / exp(lr206207r) / (-1 + exp(lambda238\*t) )^2

where t is the 207/206 date that has already been calculated, and lr206207r is the corresponding (corrected) log(206/207) ratio for the unknown.

Calculate dDate\_\_dLogratio for each unknown for the ratio of interest, then place the derivatives in a (one) column vector with the same number of rows as there are unknowns. Call this column vector dDate\_\_dLogratio\_vector. Calculate a new matrix called

d2Date\_\_dLogratio2\_matrix =

dDate\_\_dLogratio\_vector.times( dDate\_\_dLogratio\_vector.transpose() )

4a. Calculate the date covariance matrix with the contributions of the uncertainties in A-C above by calculating

*SuDateInterStdplusStdX = d2Date\_\_dLogratio2\_matrix.arrayTimes(SuInterStdplusStdX)*

Note: use .arrayTimes instead of .times from JAMA to do element-wise multiplication.

4b. Calculate the decay constant uncertainty contribution to the date covariance matrix.

*SuDateLambda* =

*JDateLambda*.times(*LambdaCovarianceMatrix*).times( *JDateLambda*.transpose() )

where LambdaCovarianceMatrix is the covariance matrix for the decay constants provided in the EARTHTIME Physical Constants model.

5. Calculate the full weighted mean covariance matrix,

*SuDateInterStdplusStdLambdaX = SuDateInterStdplusStdX + SuDateLambda*

Use this covariance matrix to calculate the weighted mean date uncertainty including all uncertainty contributions: analytical, inter-standard, standard IC, and decay constant. Use [the equation above](#lnkqxdqe2e49) for the total uncertainty, but use *SuDateInterStdplusStdLambdaX* instead of *SuX.* The units of this uncertainty are years, so this value does not need to be converted from log-units.

**D\***: Calculate the one-sigma relative uncertainty in the appropriate decay constant.

**Appendix A. Variables to output to a text file.**

For each isotope (except 202Hg)

1. Raw intensities, baseline and on-peak (as volts or cps)
2. Intensities converted to cps, baseline and on-peak
3. *Si*
4. Fit parameters for baseline (a [b, c], fitParameterCovarianceMatrix, X2)
5. *peakIntensityBLcorr* (baseline-corrected on-peak intensities)
6. J11, J21, J22, JOnPeak, Sopbc
7. ‘below the detection limit’ (boolean) for 204Pb and 207Pb
8. log-intensities (on-peak baseline-corrected)
9. *Jlogr, Jmat, Sopbclr*
10. *lr206Pb\_207Pb, lr206Pb\_238U, lr208Pb\_232Th*
11. *Sxyod, Slr\_X\_Y* for the three log-ratios
12. Fit parameters for three log-ratios (a [b, c], fitParameterCovarianceMatrix, X2)
13. Y-intercept values and uncertainties for three log-ratio fits

**Appendix B1. Converting ratios to log-ratios**

- Take the natural logarithm of the input ratios to convert them to log-ratios.

- Assemble the covariance matrix for the (un-logged ratios), and call it Sr.

- Make a new matrix of zeros, called J, that is the same size as Sr. On each diagonal element of J, set the value to 1/rA\_B where rA\_B is the ratio that corresponds to that row/column in Sr.

- Calculate Slr = J \* Sr \* J , where \* is matrix multiplication. Slr is the covariance matrix for the log-ratios.

Note: there are quicker ways to do this, but we shouldn’t be switching back and forth between ratios and log-ratios very often, and it should only happen with small systems, so programming a faster method wouldn’t really be worth our time.

## Appendix B2. Converting log-ratios to ratios

- Exponentiate the values of the log-ratio to convert them into log-ratios.

- Assemble a covariance matrix for the (un-logged) ratios, and call it Slr.

- Make a new matrix of zeros, called J, that is the same size as Slr. On each diagonal element of J, set the value to rX\_Y, where rX\_Y is the ratio (not logged) corresponding to that row/column

- Calculate Sr = J \* Slr \* J, where \* is matrix multiplication. Sr is the covariance matrix for the ratios.

Note: When we output to the data table, we do not assume that the resulting ratio has a symmetric distribution. Add +2sigma and -2sigma (in log-units) to the log-ratio, then exponentiate the results to get the 2-sigma uncertainty bound for the ratio. If that interval is asymmetric about the mean (the bounds, as ratios, are different distance from the mean ratio), then we report a different value for the +2sigma and -2sigma.

For the mineral standard model, going from 206/207 input to 207/206 output in log-ratios (to use in session fits).

- Convert the ratio covariance matrix to a log-ratio covariance matrix using B1 above.

- For the ratio you want to flip (in this case, the first ratio, which is 206/207 to 207/206), change the sign of all covariance terms on the first row and first column and on the off-diagonal of the covariance matrix. Do nothing to the term on the diagonal (it is always positive).

- You now have the covariance matrix for the log-ratios for, respectively, 207/206, 206/238, 208/232, 238/235. To convert back to a ratio covariance matrix, use Appendix B2 above.

## Appendix C: Other calculations:

For U and Th concentrations:

* Add a line for the U and Th concentration in the Mineral Standards Models, in the Data tab. The label should read “conc. U (ppm)” and there will be no line for this variable in the covariance/correlation matrices.
* The U concentration for the Sri Lanka standard is 564 ppm. For now, call the Th concentration 150 ppm.
* There is no uncertainty reported in George’s spreadsheet, and most people don’t report an uncertainty in this parameter, but you should leave a space open for this. For now, use a 10 ppm (1-sigma, abs) uncertainty for Sri Lanka.
* The U and Th concentration of a standard will always be the concentration listed in its mineral standard model.
* For an unknown, find the two standards that bracket it in time. Call these two times that the standards were measured *t1* and *t2*, and the time that the unknown was measured *tu*. Units don’t matter for the times, as long as they’re all consistent.
* For the unknown and these two standards
  + Call the number of included integrations *n* (= 15 for George, no rejection)
  + Calculate the maximum of 2 and floor(0.1\**n*), and call this *m*. For George this would be *m* = max(2, floor(1.5)) = 2.
* Take the average of the *m* highest intensity integrations in the U and the Th measurements. Call these averages *i1*, *i2*, and *ix* for the first and second standards and the unknown, respectively.
* Calculate the concentration of U or Th in the unknown using the formula *concUnknown* = *concStandard* \* *ix* / ( *i1* + (*t2* - *tu*)/(*t2* - *t1*) \* (*i2* - *i1*) ) where concUnknown is the concentration of the unknown, concStandard is the U or Th standard from the mineral standard model, and concStandard and concUknown have the same units (ppm).
* If there are any leading/trailing unknowns, before the first standard or after the final standard, then use the formula

*concUnknown* = *ix* \* *concStandard* / *i1*

for the unknowns before the first standard (*i1*), and

*concUnknown* = *ix* \* *concStandard* / *in*

for unknowns after the last (*n*th) standard (*in*).