rCBF Analysis Using MATLAB

Mark Wolforth Greg Ward Sean Marrett

15 November, 1994

Contents

1 Introduction

This document is the user manual for the rCBF (regional Cerebral Blood Flow) package developed at the Montreal Neurological Institute McConnell Brain Imaging Centre during the summer of 1993. This kinetic analysis package runs under MATLAB, in conjunction with EMMA (Extensible MATLAB Medical Analysis), a package developed at the same time.

The rCBF package described in this document performs a full two-compartment analysis of cerebral blood flow, including performing blood delay and dispersion correction. It is intended as a working example for future developers of MATLAB analysis packages, in addition to being a useful analysis tool. Therefore, this document places more emphasis on the structure and organization of the software than on the actual use of the software.

The authors would like to thank Sean Marrett for being the driving force behind this project. He provided invaluable guidance when all seemed lost.

2 Mathematical Analysis

2.1 Introduction

The two-compartment cerebral blood flow model can be characterized by the following three equations:

$$\frac{dM}{dt} = K_1 C_a(t) - k_2 M(t) \tag{1}$$

$$M(t) = K_1 C_a(t) \otimes e^{-k_2 t} \tag{2}$$

$$A(t) = M(t) + C_a(t) V_0 \tag{3}$$

In these equations, $A(t)$ is the PET data collected over a set of frames, $C_a(t)$ is the delay and dispersion corrected arterial blood sample data, and $M(t)$ is the radioactive tracer activity present in cerebral tissue. We know both $A(t)$ and $C_a(t)$, but cannot know $M(t)$ without knowing V_0.

One additional point to consider is that these equations are written for continuous functions. However, the PET data that is actually available is average activity across each frame. Therefore, in order to solve the equations, we must average the non PET data across frames. By combining equations (2) and (3), and averaging the non PET data over frames, we get:

$$A^*(t) = \frac{\int_{T_1}^{T_2} \left(K_1 \left[C_a(t) \otimes e^{-k_2 t} \right] + C_a(t) V_0 \right) dt}{T_2 - T_1} \tag{4}$$

where $A^*(t)$ is the PET data that is actually collected (by definition, averaged over frames), and $\int_{T_1}^{T_2} dt/(T_2 - T_1)$ performs an averaging over frames (T_1 is the frame start time, T_2 is the frame stop time, and the integral is evaluated for each frame).

We wish to solve equation (4) for K_1, k_2, and V_0. Of course, one approach would be to try to perform an explicit least squares curve fitting. However, this approach would be quite computationally intensive since the fitting would need to be performed for every pixel of a 128×128 pixel image. Fortunately, there is a method of solution that gives good results with reduced computational difficulty.

2.2 Double-Weighted Integration Method

Since the time required to perform a least squares curve fitting would be prohibitive, a simpler approach to solving the problem is required. One technique is to use a weighted integration method. We initially approached the problem by assuming that V_0 was negligibly small, in which case the $C_a V_0$ term is eliminated from equation (4). Taking equation (4) with $C_a V_0$ eliminated, and integrating both sides from time 0 to the end of the last frame, we get:

$$\int_0^T A^*(t) dt = K_1 \int_0^T \frac{\int_{T_1}^{T_2} \left[C_a(u) \otimes e^{-k_2 u} \right] du}{T_2 - T_1} dt \tag{5}$$

We can then take this equation, and divide it by a weighted version of itself:

$$\frac{\int_0^T A^*(t)dt}{\int_0^T A^*(t)tdt} = \frac{K_1 \int_0^T \frac{\int_{T_1}^{T_2}\left[C_a(u)\otimes e^{-k_2 u}\right]du}{T_2-T_1}dt}{K_1 \int_0^T \frac{\int_{T_1}^{T_2}\left[C_a(u)\otimes e^{-k_2 u}\right]du}{T_2-T_1}tdt} \tag{6}$$

K_1 cancels out of this equation, leaving us with an equation that only involves k_2. The left side of equation (6) is easily evaluated by integrating the PET data. The right side of equation (6) is not easily solved for k_2, so a different approach was taken. A look-up table was generated relating values of k_2 to resulting values of the right hand side of equation (6). A linear interpolation was then performed to choose values of k_2 from this look-up table for each point in the left hand side of equation (6).

2.3 Triple-Weighted Integration Method

In order to model the complete two-compartment system, we must be able to solve equation (4). In section 2.2, we solved the equation by multiplying by two different weights and then dividing, and we can take a similar approach with the full two-compartment equation. We can weight equation (4) with *three* different weights and then integrate:

$$\int_0^T w_1 A^*(t)dt = K_1 \int_0^T w_1 \frac{\int_{T_1}^{T_2}\left[C_a(u)\otimes e^{-k_2 u}\right]du}{T_2-T_1}dt + V_0 \int_0^T w_1 \frac{\int_{T_1}^{T_2} C_a(u)du}{T_2-T_1}dt \tag{7}$$

$$\int_0^T w_2 A^*(t)dt = K_1 \int_0^T w_2 \frac{\int_{T_1}^{T_2}\left[C_a(u)\otimes e^{-k_2 u}\right]du}{T_2-T_1}dt + V_0 \int_0^T w_2 \frac{\int_{T_1}^{T_2} C_a(u)du}{T_2-T_1}dt \tag{8}$$

$$\int_0^T w_3 A^*(t)dt = K_1 \int_0^T w_3 \frac{\int_{T_1}^{T_2}\left[C_a(u)\otimes e^{-k_2 u}\right]du}{T_2-T_1}dt + V_0 \int_0^T w_3 \frac{\int_{T_1}^{T_2} C_a(u)du}{T_2-T_1}dt \tag{9}$$

By multiplying equation (7) by $V_0 \int_0^T w_3(\int_{T_1}^{T_2} C_a(u)du)/(T_2-T_1)dt$ and equation (9) by $V_0 \int_0^T w_1(\int_{T_1}^{T_2} C_a(u)du)/(T_2-T_1)dt$, and then subtracting the two, we may eliminate the V_0 term. A similar operation can be performed on equation (8) and equation (9). This leaves two equations that do not contain V_0. They may then be divided to produce:

$$\frac{\begin{array}{r}\int_0^T w_3(t)\frac{(\int_{T_1}^{T_2} C_a(u)du)}{(T_2-T_1)}dt \cdot \int_0^T w_1(t)A^*(t)dt - \\ \int_0^T w_1(t)\frac{(\int_{T_1}^{T_2} C_a(u)du)}{(T_2-T_1)}dt \cdot \int_0^T w_3(t)A^*(t)dt\end{array}}{\begin{array}{r}\int_0^T w_3(t)\frac{(\int_{T_1}^{T_2} C_a(u)du)}{(T_2-T_1)}dt \cdot \int_0^T w_2(t)A^*(t)dt - \\ \int_0^T w_2(t)\frac{(\int_{T_1}^{T_2} C_a(u)du)}{(T_2-T_1)}dt \cdot \int_0^T w_3(t)A^*(t)dt\end{array}} = \frac{K_1\left\{\begin{array}{r}\int_0^T w_3(t)\frac{(\int_{T_1}^{T_2} C_a(u)du)}{(T_2-T_1)}dt \cdot \int_0^T w_1(t)\frac{(\int_{T_1}^{T_2} C_a(u)\otimes e^{-k_2 u}du)}{(T_2-T_1)}dt - \\ \int_0^T w_1(t)\frac{(\int_{T_1}^{T_2} C_a(u)du)}{(T_2-T_1)}dt \cdot \int_0^T w_3(t)\frac{(\int_{T_1}^{T_2} C_a(u)\otimes e^{-k_2 u}du)}{(T_2-T_1)}dt\end{array}\right\}}{K_1\left\{\begin{array}{r}\int_0^T w_3(t)\frac{(\int_{T_1}^{T_2} C_a(u)du)}{(T_2-T_1)}dt \cdot \int_0^T w_2(t)\frac{(\int_{T_1}^{T_2} C_a(u)\otimes e^{-k_2 u}du)}{(T_2-T_1)}dt - \\ \int_0^T w_2(t)\frac{(\int_{T_1}^{T_2} C_a(u)du)}{(T_2-T_1)}dt \cdot \int_0^T w_3(t)\frac{(\int_{T_1}^{T_2} C_a(u)\otimes e^{-k_2 u}du)}{(T_2-T_1)}dt\end{array}\right\}} \tag{10}$$

The K_1 term cancels out of both the numerator and denominator of equation (10), leaving an equation that only involves k_2. As with the equation in section 2.2, this is very difficult to solve for k_2. Therefore, a look-up table was again used. Once the table matching values of k_2 with values of the right hand side of equation (10) has been created, we may evaluate k_2 through simple lookup. With the k_2 data computed, finding K_1 is simply a matter of evaluating either the numerator or denominator of equation (10) without cancelling K_1. With both K_1 and k_2 known, we may find V_0 by evaluating equation (4).

2.4 Delay and Dispersion Correction

Since the blood samples used to estimate $C_a(t)$ are taken from the subject's arm, the activity data gathered from them must be corrected for delay and dispersion within the body. In order to perform the delay correction, we used the technique of Iida, by performing a least squares fitting of the equation:

$$A^*(t) = \frac{\int_{T_1}^{T_2} \left(\alpha \left[C_a(t) \otimes e^{-\beta t}\right] + \gamma C_a(t)\right) dt}{T_2 - T_1} \tag{11}$$

where $A^*(t)$ in this case is the average activity over grey matter, $\int_{T_1}^{T_2} dt$ represents integration over frames, α, β, and γ are the fitting parameters, and $C_a(t)$ is the blood data. The delay itself enters this equation by generating $C_a(t)$ from:

$$C_a(t) = \bar{g}(t + \delta) \tag{12}$$

The function $\bar{g}$ is the result of performing dispersion correction on the blood sample data. This is represented by the equation:

$$\bar{g}(t) = g(t) + \tau \frac{dg}{dt} \tag{13}$$

which is an implicit deconvolution of the equation:

$$g(t) = \bar{g}(t) \otimes \left[\frac{1}{\tau} e^{\frac{-t}{\tau}}\right] \tag{14}$$

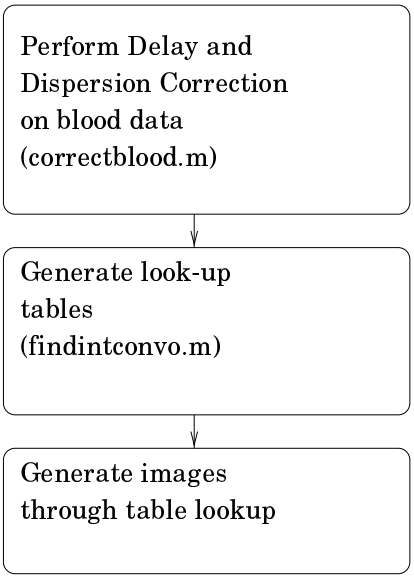

Figure 1: General program flow for rCBF analysis

3 MATLAB Implementation

3.1 Introduction

The previous section described the mathematical model used to represent cerebral blood flow. Both the simplified one-compartment model (neglecting V_0), and the full two-compartment model were implemented numerically in MATLAB.

3.2 Program Structure

The diagram in figure 1 shows the general flow of performing rCBF analysis using MATLAB. First, the blood data is prepared for analysis by performing delay and dispersion correction. Next the lookup tables used in the analysis are calculated. Finally, the actual images are generated through table lookup using the lookup tables computed in the previous step.

Of course, the actual program implementation is slightly more complex than this. Figure 2 shows the full structure of the `rcbf2` MATLAB function, which performs a full two-compartment rCBF analysis. All of the main functions that are called by `rcbf2` are shown. The `resampleblood` function returns the blood data to `rcbf2` in an evenly sampled time domain (sampled every 1/2 second). This blood data is then passed to `correctblood`, which performs dispersion correction, and then delay correction by calling `fit_b_curve`. Finally, useful lookup tables are calculated by `findintconvo`.

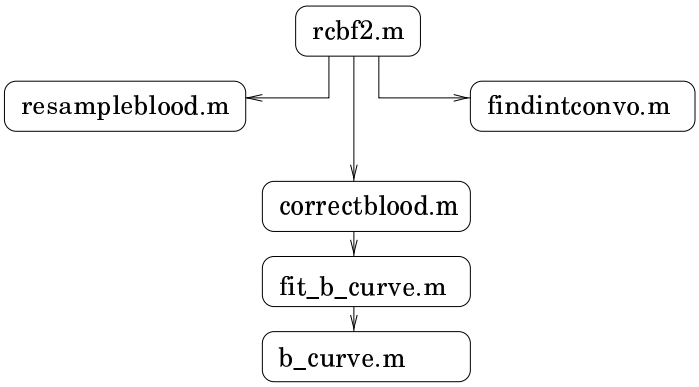

Figure 2: Structure of rcbf2

3.3 Annotated Program Listings

This section contains listings of all of the MATLAB programs comprising the rCBF package. Each program is broken into functional sections, which are presented immediately after the description of that section. Note that the code presented here is not *exactly* what you will see if you type, for example, `type rcbf2.m` in MATLAB. In the interests of concentrating on the numerical analysis, we have generally neglected code dealing with processing input arguments, reporting progress, and making some operations optional. Nevertheless, if you are writing MATLAB programs to perform similar analyses, it can be instructional to read this code and its accompanying comments.

3.3.1 RCBF1

1. The function declaration line. The output arguments `K1` and `k2` are both whole images: that is, for 128×128 PET data, they will be column vectors with 16,384 elements. `filename` is the name of the MINC file to process, `slice` specifies which slice from the file to process, and `progress` indicates whether or not the program should print progress information as it goes. `filename` and `slice` are both required; if `progress` is not given, it defaults to 1 ("true").

   ```
   function [K1,k2] = rcbf1 (filename, slice, progress)
   ```

2. Get the images and image information. This includes all of the frames for the slice being analyzed, the frame start times, the frame lengths, the mid-frame times, and the blood data. The frame time information is returned by simple enquiries with the data handle `img` returned by `openimage`. The blood data is returned by `resampleblood`, which gives the blood data in an evenly sampled time domain (every half second). The blood data is then cross-calibration corrected by multiplying by the factor `XCAL`. The cross-calibration converts from $\mathrm{Bq/g_{blood}}$ to $\mathrm{nCi/mL_{blood}}$, taking into account the calibration between the well counter and the PET scanner. In order to maintain consistent units

throughout the analysis, this data must then be converted from $\mathrm{nCi/ml_{blood}}$ *back* to $\mathrm{Bq/g_{blood}}$. The actual PET images are then retrieved, and the units are again converted to $\mathrm{Bq/g_{tissue}}$. Any negative values present in the images are set to zero.

```
img = openimage(filename);
FrameTimes = getimageinfo (img, 'FrameTimes');
FrameLengths = getimageinfo (img, 'FrameLengths');
MidFTimes = FrameTimes + (FrameLengths / 2);

[g_even, ts_even] = resampleblood (img, 'even');

% Apply the cross-calibration factor, and convert back to Bq/g_blood
XCAL = 0.11;
g_even = g_even*XCAL*37/1.05;

Ca_even = g_even;                % no delay/dispersion correction

PET = getimages (img, slice, 1:length(FrameTimes));
PET = PET * 37 / 1.05;           % convert to decay / (g_tissue * sec)
PET = PET .* (PET > 0);          % set all negative values to zero
ImLen = size (PET, 1);           % num of rows = length of image
```

3. Calculate some useful integrals that are used several times in the rest of the program. `PET_int1`, and `PET_int2` are weighted integrals of the PET data across frames (integrated with respect to time). In particular, `PET_int1` is the numerator of the left-hand-side of equation 6, and `PET_int2` is the denominator. To get clean images out of the analysis, we wish to set all points outside of the head to zero. Therefore, we create a simple mask, and apply it to the weighted integrals. Note the use of `rescale` to perform an "in-place" multiplication on `PET_int1`. This has the same effect as the more conventional MATLAB `PET_int1 = PET_int1 .* mask`, but without making a copy of `PET_int1`.

```
PET_int1 = trapz (MidFTimes, PET')';
PET_int2 = trapz (MidFTimes, PET' .* (MidFTimes * ones(1,ImLen)))';

mask = PET_int1 > mean (PET_int1);
rescale (PET_int1, mask);
rescale (PET_int2, mask);
```

4. Calculate the left hand side of equation (6).

```
rL = PET_int1 ./ PET_int2;
```

5. Assign the k_2 values to be used in the lookup table, and then generate some more useful weighted integrals. `findintconvo` computes the function

$$C_a(t) \otimes e^{-k_2 t} \tag{15}$$

at all points in the evenly-resampled time domain `ts_even`. Then, it integrates across each individual frame (which run from T_1 to T_2; T_1 and T_2 in the following formula are implicitly functions of the frame). Then, the weighted integrals across *all* frames are computed:

$$\int_0^T \frac{\int_{T_1}^{T_2} \left[C_a(u) \otimes e^{-k_2 u}\right] du}{T_2 - T_1} w_i dt \tag{16}$$

Here, w_i is the weighting function; for the double-weighted analysis it is either 1 or t as in the right hand side of equation (6).

Then, since we wish to relate k_2 to the values of these integrals, `findintconvo` computes equation 16 at a wide range of values of k_2 (supplied by the argument `k2_lookup`) for each weighting function w_i. (For the double-weighted method, $w_1 = 1$ and $w_2 = t$.) Note that the supplied value of `k2_lookup` implies an assumption that no voxel in the image will have a k_2 value outside the range $0 \ldots 3\,\mathrm{min}^{-1}$.) See section 3.3.4 for information on the internal details of `findintconvo`.

```
k2_lookup = (0:0.02:3) / 60;
[conv_int1, conv_int2] = findintconvo (Ca_even, ts_even, k2_lookup,...
                            MidFTimes, FrameLengths, 1, MidFTimes);
```

6. Calculate the right side of equation (6).

```
rR = conv_int1 ./ conv_int2;
```

7. Generate the k_2 image through table lookup. This is where we make use of the lookup tables generated in `findintconvo` for great time savings: `findintconvo` only has to compute equation 16 (a comparatively slow operation) a few hundred times, but for that effort we get 16,384 values of k_2 very quickly.

```
k2 = lookup(rR, k2_lookup, rL);
```

8. Generate the K_1 image through table lookup and division. `k2_conv_ints` contains the values of equation 16 at the actual values of k_2 for this image, rather than at the artificial set `k2_lookup`. This step is where we solve the numerator of equation 6 for K_1.

```
k2_conv_ints = lookup (k2_lookup, conv_int1, k2);
K1 = PET_int1 ./ k2_conv_ints;
```

9. Clean up the K_1 image by setting any NaN's and infinities to zero, and close the image data set.

```
nuke = find (isnan (K1) | isinf (K1));
K1 (nuke) = zeros (size (nuke));
closeimage (img);
```

10. Finally, K_1 is converted from the internal units $[g_{blood}/g_{tissue}]$ to the standard units for rCBF analysis $[mL_{blood}/(100\, g_{tissue} \cdot min)]$.

```
rescale (K1, 100*60/1.05);
```

3.3.2 RCBF2

`RCBF2` is essentially an extension of `RCBF1` with the addition of a third weighting function (needed to calculate V_0) and correction for the delay and dispersion of blood. Also, RCBF2 allows the user to specify a list of slices (as a vector) rather than requiring a scalar `slice`. This means that we add a loop through all desired slices, so some of the code is rearranged to avoid redoing the same work several times in the loop.

1. The function declaration line. The output arguments `K1`, `k2`, and `V0` will all be whole images—that is, for 128×128 PET data, they will be matrices with 16,384 elements per column, and one column per slice processed. `delta` will contain the blood delay term δ for each specified slice.

   Of the input arguments, only `filename` (the name of the MINC volume to process) and `slices` (the list of slices to process) are required. The other three are boolean variables (i.e., they should be scalars with a value of either 0 or 1) that default to "true" (1). `progress` controls whether RCBF2 prints progress information; `correction` decides whether or not it performs correction of the blood data for delay and dispersion; and `batch` controls whether selection of the mask used for delay/dispersion correction should be interactive or automatic (the default). The latter two should *not* be changed when RCBF2 is being used for real data analysis, and are merely provided to speed things up when debugging.

```
function [K1,k2,V0,delta] = rcbf2_slice ...
         (filename, slices, progress, correction, batch)
```

2. Initialize the matrices that will hold the K_1, k_2, and V_0 images, as well as the vector to hold the value of δ (the blood delay from equation 12) for each slice.

```
total_slices = length(slices);
K1 = zeros(16384,total_slices);
k2 = zeros(16384,total_slices);
V0 = zeros(16384,total_slices);
delta = zeros(1,total_slices);
```

3. Open the volume and get some preliminary information on frame times and lengths.

```
img = openimage(filename);
if (getimageinfo (img, 'time') == 0)
   error ('Study is non-dynamic');
end

FrameTimes = getimageinfo (img, 'FrameTimes');
FrameLengths = getimageinfo (img, 'FrameLengths');
MidFTimes = FrameTimes + (FrameLengths / 2);
```

4. Read the blood data, perform cross-calibration, and convert units. (See explanation in section 3.3.1.)

```
[g_even, ts_even] = resampleblood (img, 'even');
XCAL = 0.11;
rescale(g_even, (XCAL*37/1.05));
```

5. Create the weighting functions, w_1, w_2, and w_3 from equations 7–9. Also here we initialize the list of k_2 values from which the lookup table will be generated.

```
w1 = ones(length(MidFTimes), 1);
w2 = MidFTimes;
w3 = sqrt (MidFTimes);

k2_lookup = (-10:0.05:10) / 60;
```

6. At this point, we start the processing that is specific to each slice: read the PET data, perform delay/dispersion correction, generate the lookup table, and solve equation 10. Thus, we loop through all user-specified slices:

```
for current_slice = 1:total_slices
```

7. Read in the PET data for the current slice, and rescale it to convert from $\mathrm{nCi/mL_{tissue}}$ to $\mathrm{Bq/g_{tissue}}$.

```
PET = getimages (img, slices(current_slice), 1:length(FrameTimes), PET);
rescale (PET, (37/1.05));
```

8. Compute the weighted integrals of the PET data.

```
PET_int1 = ntrapz (MidFTimes, PET, w1);
PET_int2 = ntrapz (MidFTimes, PET, w2);
PET_int3 = ntrapz (MidFTimes, PET, w3);
```

   This uses `ntrapz`, the CMEX version of `trapz`, which has the useful feature of allowing a weighting function to be supplied. Thus, the first line of code above is equivalent to (but faster and less memory-intensive than)

```
weight = ones (16384,1) * w1';
PETweighted = PET .* weight;
PET_int1 = trapz (MidFTimes, PETweighted');
```

9. Generate a simple mask based on `PET_int1` (which is simply the PET data averaged across all frames), and mask using `rescale`. Also, `mask` is cleared for memory efficiency.

```
mask = PET_int1 > mean(PET_int1);
rescale (PET_int1, mask);
rescale (PET_int2, mask);
rescale (PET_int3, mask);
clear mask;
```

   The next three steps perform delay/dispersion correction of the blood data (see section 2.4).

10. First, create a mask that is used to select gray matter only; the mask is a variation on that used to mask the weighted PET integrals above. That is, simply select all voxels whose values are greater than some constant times the mean of `PET_int1`. If non-interactive mode is on (i.e. `batch` = 1, the default behaviour), then this constant is hard-coded to 1.8[1]; otherwise, the function `getmask` is run. This allows the user to select the threshold value while displaying the resulting, masked PET data.

[1] This constant was empirically selected because it works fairly well with $H_2{}^{15}O$ CBF data.

```
if (batch)
  mask = PET_int1 > (1.8*mean(PET_int1));
else
  mask = getmask (PET_int1);
end
```

11. Use the mask to get the mean of all gray-matter voxels. Thus, we reduce the dynamic PET data (16,384 voxels sampled at n points in time) to a single time-activity curve (average of gray-matter activity at n points in time).

```
A = (mean (PET (find(mask),:)))';
clear mask;
```

12. The actual delay/dispersion correction is performed by `correctblood`. See sections 2.4, and 3.3.3 for information on (respectively) the theoretical basis and the implementation of delay/dispersion correction.

```
[ts_even, Ca_even, delta(:,current_slice)] = correctblood ...
    (A, FrameTimes, FrameLengths, g_even, ts_even, progress);
```

13. Generate tables of the three weighted integrals of $Ca(t) \otimes e^{-k_2 t}$. (Actually, we generate tables of this expression integrated across each individual frame, then integrated across all frames; the procedure is identical to that used in RCBF1 except for the addition of a third weighting function and the greatly expanded range of possible values for k_2.)

```
[conv_int1,conv_int2,conv_int3] = findintconvo (Ca_even,ts_even,...
   k2_lookup, MidFTimes, FrameLengths, 1, w2, w3);
```

14. Generate some additional useful integrals. These are used more than once in the subsequent code, and so are calculated in advance to speed up the computation. `Ca_mft` is the blood data averaged over each frame. This is taken as the value of the blood data at the mid-frame time. Note that we must detect when the blood data does not cover all the frames that the PET data does; this is made easy because `nframeint` does the work for us. In particular, if the start time of any frame (some element of `FrameTimes`) is less than the start time of blood data [`ts_even(1)`], then `nframeint` returns `NaN` (not-a-number) in the element of `Ca_mft` corresponding to that frame. Similarly, if the end time of any frame (some element of `FrameTimes+FrameLengths`) is greater than the end time of blood data [`ts_even(length(ts_even))`], then `NaN` is also returned. The frames that fall outside of the blood data are then not used in generating the weighted integrals of the blood data.

```
Ca_mft = nframeint (ts_even, Ca_even, FrameTimes, FrameLengths);
select = ~isnan(Ca_mft);

if (sum(select) ~= length(FrameTimes))
  disp('Warning: blood data does not span frames.');
end

Ca_int1 = ntrapz(MidFTimes(select), Ca_mft(select), w1(select));
Ca_int2 = ntrapz(MidFTimes(select), Ca_mft(select), w2(select));
Ca_int3 = ntrapz(MidFTimes(select), Ca_mft(select), w3(select));
```

15. Generate the lookup table relating k_2 to values of the right hand side of equation (10). We also calculate the left hand side of equation (10), which will be used in the generation of a k_2 image. Here, we see the value of precomputing all the terms of `rL` and `rR`—not only is the code fairly straightforward [as long as you understand the correspondence between the variables `Ca_int`i, `PET_int`i and `conv_int`i, and the terms of equation (10)], but the computation of `rL` and `rR` is very fast.

    Note that since `PET_int`i is an image (i.e. it contains a value for each voxel in the slice), and `Ca_int`i is a scalar, then `rL` will be an image. However, `conv_int`i is a lookup table keyed on `k2_lookup`; that is, there it contains one value for every value of k_2 presumed possible. Thus, `rR` will be also be a lookup table keyed on `k2_lookup`.

    ```
    rL = ((Ca_int3 * PET_int1) - (Ca_int1 * PET_int3)) ./ ...
          ((Ca_int3 * PET_int2) - (Ca_int2 * PET_int3));

    rR = ((Ca_int3 * conv_int1) - (Ca_int1 * conv_int3)) ./ ...
          ((Ca_int3 * conv_int2) - (Ca_int2 * conv_int3));
    ```

16. Since `rL` and `rR` are the left- and right-hand-sides of equation (10), we can use their equality to find k_2 for every voxel. That is, we know `rL` for every voxel, and we know `rR` for a certain set of values of k_2. Thus, we can use `rR` to interpolate values of k_2. First, however, we must invert the current relationship between `k2_lookup` and `rR`; that is, we must make `k2_lookup` a lookup table keyed on the values of `rR`. MATLAB's `sort` function makes this quite easy: we can sort `rR` and get a vector containing the "sort order" in one step. Since we need `k2_lookup` in its original order later on, we make a copy called `k2_sorted` (even though it's now scrambled) that is ordered according to `rR`.

    ```
    [rR,sort_order] = sort (rR);
    k2_sorted = k2_lookup (sort_order);
    ```

17. Generate the k_2 image, through a simple table lookup. Values of k_2 are chosen by finding the value of k_2 where `rL` and `rR` are equal.

```
k2 = lookup(rR, k2_sorted, rL);
```

18. Generate the K_1 image by evaluating the numerator of equation (10). All of the time consuming calculations have already been performed, and we can evaluate the $\int_0^T w(\int_{T_1}^{T_2} Ca(u) \otimes e^{-k_2 u} du)/(T_2 - T_1) dt$ terms through table lookup.

```
K1_numer = ((Ca_int3*PET_int1) - (Ca_int1 * PET_int3));
K1_denom = (Ca_int3 * lookup(k2_lookup,conv_int1,k2)) - ...
           (Ca_int1 * lookup(k2_lookup,conv_int3,k2));
K1 = K1_numer ./ K1_denom;
```

19. Generate the V_0 image by evaluating equation (7) directly. Once again, we may get values for the complicated part of the equation through simple table lookup.

```
V0 = (PET_int1 - (K1 .* lookup(k2_lookup,conv_int1,k2))) / Ca_int1;
```

20. Clean up the images by removing NaN's and infinities (setting them to zero).

```
nuke = find (isnan (K1));
K1 (nuke) = zeros (size (nuke));
nuke = find (isinf (K1));
K1 (nuke) = zeros (size (nuke));

nuke = find (isnan (V0));
V0 (nuke) = zeros (size (nuke));
nuke = find (isinf (V0));
V0 (nuke) = zeros (size (nuke));
```

21. Convert from the units used internally to the standard units for rCBF analysis.

```
rescale (K1, 100*60/1.05);
rescale (k2, 60);
rescale (V0, 100/1.05);
```

22. Finally, close the image file so that everything gets cleaned up nicely.

```
closeimage (img);
```

3.3.3 CORRECTBLOOD

1. Function declaration and input/output arguments:

```
function [new_ts_even, Ca_even, delta] = correctblood ...
      (A, FrameTimes, FrameLengths, g_even, ts_even, options)
```

   The input arguments are:

`A:`	PET activity versus time, averaged across grey matter and sampled at the mid-frame times
`FrameTimes:`	the start time for each frame (in seconds, relative to the study start time)
`FrameLengths:`	the length of each frame (in seconds)
`g_even:`	blood data, resampled at an evenly-spaced time domain
`ts_even:`	the time domain at which `g_even` is sampled

   The output arguments are:

`new_ts_even:`	Generally the same as `ts_even`, with some elements possibly chopped off from the end due to shifting of the time scale.
`Ca_even:`	`g_even` after correction for blood dispersion and delay.
`delta:`	The computed delay correction δ, in seconds.

2. Do the initial setup: calculate the mid-frame times, and find frames that fall in the first 60 seconds of the study. (Although the blood data is corrected over the entire study, the delay correction is based on the PET data from the first 60 seconds only, since the PET time-activity-curve is relatively flat by $t = 60$.) because

```
MidFTimes = FrameTimes + FrameLengths/2;
first60 = find (FrameTimes < 60);       % all frames in first
numframes = length(FrameTimes);         %  minute only
```

3. Perform delay dispersion; we use `deriv` to compute a smoothed version of `g_even` and its first derivative, and then replace `g_even` with the dispersion-corrected blood data ($\bar{g}$ in equation 13).

```
[smooth_g_even, deriv_g] = ...
     deriv (3, length(ts_even), g_even, (ts_even(2)-ts_even(1)));
smooth_g_even(length(smooth_g_even)) = [];
deriv_g(length(deriv_g)) = [];
ts_even(length(smooth_g_even)) = [];

g_even = smooth_g_even + tau*deriv_g;
```

4. At this point, we are ready to start the delay correction—i.e., we will find the value of δ that gives a function $C_a(t)$ that best fits the PET activity $A^*(t)$ in equation 12. However, there are three other unknowns in equation 12, and it was found that a full four-parameter fit was highly unstable. Therefore, we select a fixed set of values for δ, perform multiple three-parameter fits (for α, β, and γ), and select the value of δ which resulted in the best fit.

   First, we prepare for the repeated fits. The fixed set of δ-values is selected; we initialize `rss` and `params` (used to select the value of δ which resulted in the best fit); and select values of α, β, and γ to start the fitting. These values are selected as being fairly representative of real data [$\alpha = 0.6\mathrm{mL}_{\mathrm{blood}}/(100\,\mathrm{g}_{\mathrm{tissue}} \cdot \mathrm{min})$, $\beta = \alpha/0.8$, and $\gamma = 0.03\,\mathrm{g}_{\mathrm{blood}}/\mathrm{g}_{\mathrm{tissue}}$], but with α converted to our internal units of $\mathrm{g}_{\mathrm{blood}}/(\mathrm{g}_{\mathrm{tissue}} \cdot \mathrm{sec})$ and β to $1/sec$. The `init` vector holds these values of α, β, and γ.

```
deltas = -5:1:10;
init = [.0001 .000125 .03];
rss = zeros (length(deltas), 1);
params = zeros (length(deltas), 3);
```

5. Now enter the loop through the values of δ in `deltas`. The first step is to copy the current element, `deltas(i)`, to `delta`; this is done solely to make the code easier to read.

```
for i = 1:length(deltas)
   delta = deltas (i);
```

6. Now inside the loop, we perform a three-parameter fit for the current value of δ. This is done by first computing `shifted_g_even`, which is the current "guess" at $C_a(t)$ (see equation 12). Note that the shifted activity is computed from `g_even`, which at this point contains $\bar{g}(t)$, the dispersion-corrected blood data (equation 13).

```
   shifted_g_even = lookup ((ts_even-delta), g_even, ts_even);
   g_select = find (~isnan (shifted_g_even));
```

7. Now perform the fit. The function `delaycorrect` encapsulates the entire fitting procedure into one function; this is equivalent to having one user-supplied function that evaluates the function to minimize, and another function (such as `fmins`) to iteratively evaluate and minimize this function. However, because this method of delay correction was found to be extremely slow, `delaycorrect` was written entirely in C as a special case.

```
final = delaycorrect (init, ...
                      shifted_g_even(g_select), ...
                      ts_even(g_select), ...
                      A, FrameTimes, FrameLengths);
```

8. Save the fit results (α, β, and γ) and compute the residual (sum of the squares of the differences between the data points and points on the fitted curve) for this value of δ. (`fit_b_curve` simply calls `b_curve` to compute the right-hand-side of equation 11, the "blood curve," for the current values of α, β, γ, and δ, at each mid-frame time. Then, it finds the differences between this and the points on the PET activity curve `A`, and returns the sum of the squares of these differences. This is saved to the vector `rss`.

```
params (i,:) = final;
rss(i) = fit_b_curve (final, ...
           shifted_g_even(g_select), ts_even(g_select), ...
           A, FrameTimes, FrameLengths);
```

9. Finally, we re-use the current values of α, β, γ as initial values for the next fit and end the `for` loop.

```
   init = final;
end      % for delta
```

10. At this point, we have a vector of residual sums of squares (a single number estimating the goodness-of-fit for each value of δ). To select the "winning" δ, we simply find the minimum of this vector.

```
[err, where] = min (rss);
delta = deltas (where);
```

11. Now that we have a value for `delta`, perform the actual delay correction (shift the blood data). We must also remove NaN's from the new data. These appear if `lookup` cannot perform a lookup at a point (i.e., there is no corresponding point in the table). They will therefore occur at the end points, where `ts_even` does not span `(ts_even-delta)`.

```
Ca_even = lookup ((ts_even-delta), g_even, ts_even);

nuke = find(isnan(Ca_even));
Ca_even(nuke) = [];
```

12. And finally, make `new_ts_even` a truncated copy of `ts_even` that fits `Ca_even` and reflects the loss of information due to resampling at the very end of the data.

```
new_ts_even = ts_even;
new_ts_even(nuke) = [];
```

3.3.4 FINDINTCONVO

`findintconvo` ("find integrals of convolutions") computes the values of

$$\int_0^T \frac{\int_{T_1}^{T_2} \left[C_a(u) \otimes e^{-k_2 u}\right] du}{T_2 - T_1} w_i \, dt \tag{17}$$

at many different values of k_2, for up to three weighting functions w_i. Here, T_1 and T_2 represent the start and end time any particular frame; in concrete terms, the inner integrand is just the average of $C_a(u) \otimes e^{-k_2 u}$ across each frame, and is computed once for each frame. The outer integral then integrates across all frames to arrive at a single value for each weighting function w_i and each value of k_2.

1. The function declaration line.

```
function [int1, int2, int3] = ...
   findintconvo (Ca_even, ts_even, k2_lookup,...
                 midftimes, flengths, w1, w2, w3, progress)
```

   The input arguments are:

   `Ca_even:` the blood data resampled at an evenly-spaced time domain. Usually, this is as returned by `resampleblood`, and possibly delay- and dispersion-corrected by `correctblood`.

   `ts_even:` the time domain at which `Ca_even` was resampled.

   `k2_lookup:` the list of k_2 values at which to evaluate equation 17. The output will consist of one value per weighting function per value of k_2.

   `midftimes:` the mid-frame times (used to perform the frame-by-frame integration).

   `flengths:` the length of each frame (also used in the frame-by-frame integration).

   `w1,w2,w3:` the three weighting functions w_i. `w1` must be supplied, but if it is an empty matrix, then a weighting function of 1 is assumed. `w2` and `w3` are not required, and if they are not given, then `int2` and `int3` (respectively) will not be defined in the output. Generally, the weighting funtions are simple functions of t such as 1, t, or t^2.

   `progress:` an optional argument indicating whether progress should be reported (in the form of printing a dot when the weighted integrals for each value of k_2 have been computed). This is quite useful, as `findintconvo` is one of the most time-consuming steps of RCBF analysis.

The output arguments `int1`, `int2`, and `int3` are simply vectors containing the value of equation 17 for the three different weighting functions; each element of these vectors corresponds to one value of k_2 as supplied in `k2_lookup`.

2. Do some initial setup. We need to get the sizes of various vectors, and initialize vectors that we will fill element by element later. Initializing vectors to all zero before filling them allows better memory management by MATLAB.

```
NumEvenTimes = length(ts_even);
NumFrames = length(midftimes);
fstart = midftimes - (flengths / 2);

TableSize = length (k2_lookup);
integrand = zeros (NumFrames, 1);

if (nargin >= 6); int1 = zeros (1, TableSize); end;
if (nargin >= 7); int2 = zeros (1, TableSize); end;
if (nargin == 8); int3 = zeros (1, TableSize); end;

% if w1 is empty, assume that it should be all ones

if isempty (w1)
   w1 = ones (size(NumFrames));
end
```

3. Calculate each element of the integrals, one at a time. Unfortunately, there does not seem to be any way to vectorize this operation, and it must therefore be performed within an inefficient `for` loop.

```
for i = 1:TableSize
```

4. First calculate the innermost terms of equation 17. `exp_fun` is $e^{-k_2 t}$, and `convo` is $C_a(t) \otimes e^{-k_2 t}$. We then perform the inner integration, i.e. average across all frames. This is performed by `nframeint`, a special CMEX routine that has been carefully optimised for this very common operation in RCBF analysis. Type `help nframeints` in MATLAB for more information.

```
   exp_fun = exp(-k2_lookup(i) * ts_even);
   convo = nconv(Ca_even, exp_fun, ts_even(2) - ts_even(1));

   integrand = nframeint (ts_even, convo(1:length(ts_even)),...
                          fstart, flengths);
```

5. Find any frames that are not completely covered by the `ts_even` time domain. `nframeint` actually does this for us, returning NaN (not-a-number) in any such element of its output vector. Thus, we only use data from frames that are *not* equal to NaN, selected using the built-in `isnan` function.

```
select = ~isnan(integrand);
```

6. Now we calculate the outer integral (the weighted integral across all frames) for each weighting function with the current value of k_2. Again, we use a CMEX routine to enhance performance; `ntrapz` is a replacement for MATLAB's `trapz` function that is much faster, as well as having an optional argument to allow weighting of the integrand. Note the use of `select` to make sure that we only use frames actually spanned by the blood data.

```
int1 (i) = ntrapz(midftimes(select), integrand(select), w1(select));
int2 (i) = ntrapz(midftimes(select), integrand(select), w2(select));
int3 (i) = ntrapz(midftimes(select), integrand(select), w3(select));
```

   (We have omitted the code that makes `w2` and `w3` optional in favour of concentrating on the numerical analysis.)

7. End the `for` loop.

```
end
```

3.3.5 B_CURVE

`bcurve` computes the right-hand-side of equation 11, that is,

$$\frac{\int_{T_1}^{T_2} \left(\alpha \left[C_a(t) \otimes e^{-\beta t}\right] + \gamma C_a(t)\right) dt}{T_2 - T_1} \tag{18}$$

In the early stages of the development of RCBF, `b_curve` was used in the non-linear fitting required for delay correction. Now, the entire fitting procedure for delay correction is encapsulated in the CMEX routine `delaycorrect`. However, `delaycorrect` simply performs one fit (optimising α, β, and γ for a single value of δ); it is called multiple times, and `b_curve` is used to pick which value of δ resulted in the best fit (in the least squares sense). In particular, we wish to satisfy equation 11; therefore, the sum of the squares of the differences between its two sides ($A^*(t)$ and equation 18) for a particular value of δ is computed, and the value of δ that results in the smallest residual is picked as the delay factor. The finding of the residual is actually done in `fit_b_curve`, which due to its simplicity (it has only two line of interesting code, one of which is a call to `b_curve`) is not shown here.

1. Input and output arguments:

```
function integral = b_curve ...
    (args, shifted_g_even, ts_even, A, fstart, flengths)
```

The output argument `integral` is just the values of equation 18 after frame-by-frame integration, i.e. there is one number for each frame.

The input arguments are

`args:`	a three element vector containing α, β, and γ from equation 18. Keep in mind in the following code that `args(1)` is α, `args(1)` is β, and `args(1)` is γ.
`g_even:`	the blood data resampled at an evenly-spaced time domain, and possibly shifted by some delay factor δ. We are not interested here in what that value of δ is, however.
`ts_even:`	the evenly-spaced time domain at which `g_even` is sampled.
`A:`	the PET activity, averaged over gray matter, for the current slice. (This isn't actually used in `b_curve`, but the argument is still here for historical reasons and to make calls to `b_curve` and `fit_b_curve` look the same.)
`fstart:`	the frame start times.
`flengths:`	the frame lengths.

2. Evaluate the first term of the integrand in equation 18.

```
expthing = exp(-args(2)*ts_even);
c = nconv(shifted_g_even,expthing,ts_even(2)-ts_even(1));
c = c (1:length(ts_even));
i1 = args(1)*c;                    % alpha * (convolution)
```

3. Evaluate the second term of the integrand, and the two terms together to get the integrand, `i`.

```
i2 = args(3)*shifted_g_even;              % gamma * g(t - delta)

i = i1+i2;
```

4. Perform the frame-by-frame integration.

```
integral = nframeint (ts_even, i, fstart, flengths);
```

5. Clean up any NaN's that have cropped up by setting them to zero. Also, truncate the function to whatever length `A` is.

```
nuke = find (isnan (integral));
integral (nuke) = zeros (size (nuke));

integral = integral (1:length(A));
```

4 Using the rCBF Package

4.1 Preparing Blood Data

The EMMA function `resampleblood` that is used in rCBF to get the blood data expects the data to either be included in the image MINC file, or to be contained in a netCDF file with a `.bnc` ending.

In order to assist the user by creating the `.bnc` file, there is a utility called `bloodtonc`. This program takes a `.cnt` file, and converts it to a `.bnc` file that will be read by the rCBF package.

To perform the conversion, use FTP to transfer the `.cnt` file from the VAX to the SGI using ASCII mode. Next, type:

```
bloodtonc filename.cnt filename.bnc
```

where `filename` is the name of the file (eg. `arnaud_20547`). In order for `resampleblood` to find the `.bnc` file, it must be in the same directory as the `.mnc` file that contains the images. Therefore, make sure that they are in the same directory before performing the analysis using rCBF.

4.2 Doing the Analysis

Once the blood and image files are prepared, the analysis can be performed. Start MATLAB by typing `matlab` at the shell prompt. Starting the analysis is very straightforward. You may type `help rcbf2` to get information on running the rCBF package. Basically, `rcbf2` is a MATLAB function that returns a K_1, k_2, and V_0 image, as well as the delay found during blood delay correction. It requires the name of the MINC file that contains the images, and the number of the slice to analyze. Therefore, if we wished to analyze slice 12 of `arnaud_20547.mnc`, we would call `rcbf2` as:

```
[K1, k2, V0, delay] = rcbf2('arnaud_20547.mnc', 12);
```

In this case, the semi-colon at the end is *very* important since `rcbf2` will return the entire images. If the semi-colon is omitted, the K_1, k_2, and V_0 values for every pixel will be echoed to the screen, which takes a considerable amount of time.

Once the images have been generated, they may be manipulated using any of the normal EMMA tools (viewed with `viewimage`, saved with `putimages`, etc.).