Eglinton_etal_SI_Dataset

November 21, 2020

0.1 Script for performing all analyses and generating all plots for Eglinton et al. "Basin-scale climate control on terrestrial biospheric carbon turnover" (2020).

Run this script to perform all statistical analyses and generate: Figs. 1-4, Extended Data Figs. 3-6, and Supplementary Information Tables 3-4. This script takes as inputs: catchment outline shapefiles generated in ArcGIS, file containing all biomarker isotope data ('sample_data_weighted.csv'), file containing all catchment property data ('unweighted_statistics.csv'), and file containing all spatially resolved catchment properties weighted by flow-length above sampling location ('weighted_statistics.csv').

```
In [1]: #import packages
        import cmocean as cmo
        import geopandas as gpd
        import matplotlib.pyplot as plt
        import numpy as np
        import pandas as pd
        import statsmodels.api as sm

        from numpy.linalg import eig, inv
```

0.1.1 Define all functions for analysis

The first step is to define all necessary functions to perform the statistical analyses. The linear algebra basis for these functions is largely taken from Draper and Smith (1998) (for MLR) and Legendre and Legendre (1998) (for RDA and related analyses). Note that all functions utilize numpy masked arrays in order to handle missing data.

```
In [2]: #define necessary functions for analysis


        #function for calculating significance p-values
        def pvals(Y, X):
            '''
            Function to calculate correlation p-values between two matrices, X and
            Y. This function can handle missing data and either 1d or 2d matrices.

            Paramters
            ---------
            Y : array-like
        Array of response variables, of shape [`n`,`p`]. Can
        contain missing data points.

    X : array-like
        Array of control variables, of shape [`n`,`m`]. Can
        contain missing data points.

    Returns
    -------
    p : array-like
        Matrix of ordinary least squares p-values calculated using sm.OLS.
    '''

    #make empty matrix to store data
    p = pd.DataFrame(index = X.columns, columns = Y.columns)

    #loop through and calculate p values using sm.OLS
    for cv in X.columns:
        for rv in Y.columns:

            #extract data and get into right format
            x = sm.add_constant(X[cv])
            y = Y[rv]

            #calculate regression
            res = sm.OLS(y, x, missing='drop').fit()

            #extract p-value and store
            p.loc[cv, rv] = res.f_pvalue

    return p

#multiple linear regression function
def MLR(Y, X, standardize = True):
    '''
    Function to perform ordinary least squares (OLS) multiple linear regression
    (MLR) analysis. This function can handle arrays with missing data.

    Parameters
    ----------
    Y : 2d array-like
        Array of response variables, of shape [`n`,`p`]. Can
        contain missing data points.

    X : 2d array-like
        Array of control variables, of shape [`n`,`m`]. Can
        contain missing data points.
```

```
standardize : boolean
    Boolean telling the function to standardize (i.e., whiten) X and Y.
    Defaults to `True`.

Returns
-------
B : 2d array-like
    Parameter matrix, of shape [`m`,`p`].

Yhat : 2d array-like
    Matrix of predicted response variables, of shape [`n`,`p`].

Yr : 2d array-like
    Matrix of residuals, of shape [`n`,`p`].
'''

#if data frames, extract values
if type(X) is pd.DataFrame:
    xdf = True
    xcols = X.columns
    sams = X.index
    X = X.values

else:
    xdf = False

if type(Y) is pd.DataFrame:
    ydf = True
    ycols = Y.columns
    Y = Y.values

else:
    ydf = False

#calcualte length of each control and respones pair and make matrix
nmat = np.dot(1*(~np.isnan(X).T), 1*(~np.isnan(Y)))

#extract shapes
if type(nmat) is np.ndarray:

    if nmat.ndim == 2:
        n,m = np.shape(nmat)
    else:
        n = len(nmat)
        m = 1

else:
    n = m = 1
```

```
#if standardize is false, can calculate using matrix algebra
if standardize is False:

    #make masked arrays to deal with missing data
    X_m = np.ma.array(X,
                      mask = np.isnan(X)
                     )

    Y_m = np.ma.array(Y,
                      mask = np.isnan(Y)
                     )

    #calculate parameter matrix and Yhat matrix
    B = np.ma.dot(np.ma.dot(inv(np.ma.dot(X_m.T, X_m)), X_m.T), Y_m)
    Yhat = np.ma.dot(X_m, B)

    #only keep the data
    if n > 1 or m > 1:
        B = B.data
        Yhat = Yhat.data

    #calcualte residuals
    Yr = Y - Yhat

#else if standardize is true and Y is scalar, calculate results
elif m == 1:

    #whiten entire X matrix
    X = (X - np.nanmean(X,0)) / np.nanstd(X,0,ddof = 1) #n-1 d.o.f.

    #extract non-nan indices and drop nans
    ind = ~((np.isnan(X).any(axis=1) + np.isnan(Y)))
    Xd = X[ind,:]
    Yd = Y[ind]

    #whiten Y array
    Yd = (Yd - Yd.mean()) / Yd.std(ddof = 1) #n-1 d.o.f.

    #calculate parameter matrix and Yhat matrix
    B = np.dot(np.dot(inv(np.dot(Xd.T, Xd)), Xd.T), Yd)
    Yhat = np.dot(X, B)

    #calculate residuals
    Yr = Y - Yhat

#if standardize is true and Y is matrix, loop through and calculate results
else:
        #whiten entire X and Y matrices
        X = (X - np.nanmean(X,0)) / np.nanstd(X,0,ddof = 1) #n-1 d.o.f.
        Y = (Y - np.nanmean(Y,0)) / np.nanstd(Y,0,ddof = 1) #n-1 d.o.f.

        #loop through each column of Y; calculate parameters
        B = np.zeros([n,m])

        for j in range(m):
            #extract j column from Y
            y = Y[:,j]

            #extract non-nan indices and drop nans
            ind = ~((np.isnan(X).any(axis=1) + np.isnan(y)))
            Xd = X[ind,:]
            yd = y[ind]

            #calculate parameter matrix and Yhat matrix
            B[:,j] = np.dot(np.dot(inv(np.dot(Xd.T, Xd)), Xd.T), yd)


        #calculate Yhat and Yr
        Yhat = np.dot(X, B)
        Yr = Y - Yhat

    #if inputs are dataframes, make outputs dataframes as well
    if xdf and ydf:
        B = pd.DataFrame(B,
                         index = xcols,
                         columns = ycols
                        )

        Yhat = pd.DataFrame(Yhat,
                            index = sams,
                            columns = ycols
                           )

        Yr = pd.DataFrame(Yr,
                          index = sams,
                          columns = ycols
                         )

    return B, Yhat, Yr

#function to calculate correlation matrix
def cmat(Y, X, corr = False):
    '''
    Function to calculate correlation matrix between two matrices, X and
    Y. This function can handle missing data and either 1d or 2d matrices.
```

```
Paramters
---------
Y : array-like
    Array of response variables, of shape [`n`,`p`]. Can
    contain missing data points.

X : array-like
    Array of control variables, of shape [`n`,`m`]. Can
    contain missing data points.

corr : boolean
    Tells the function whether or not to calculate covariance
    or correlation matrix; if `False`, covariance matrix is given.
    Defaults to `False`.

Returns
-------
c : array-like
    Covariance or correlation matrix, of shape [`p`,`m`]. If both X
    and Y are 1-dimensional, c is a scalar.
'''

#if data frames, extract values
if type(X) is pd.DataFrame:
    xdf = True
    xcols = X.columns
    X = X.values

else:
    xdf = False

if type(Y) is pd.DataFrame:
    ydf = True
    ycols = Y.columns
    Y = Y.values

else:
    ydf = False

#calcualte length of each control and respones pair and make matrix
nmat = np.dot(1*(~np.isnan(X).T), 1*(~np.isnan(Y)))

#extract shapes
if type(nmat) is np.ndarray:

    if nmat.ndim == 2:
        n,m = np.shape(nmat)
    else:
        n = len(nmat)
        m = 1

else:
    n = m = 1

#if corr is false, can calculate using matrix algebra
if corr is False:
    #subtract means only
    X = X - np.nanmean(X,0)
    Y = Y - np.nanmean(Y,0)

    #make masked arrays to deal with missing data
    X_m = np.ma.array(X,
                      mask = np.isnan(X)
                     )

    Y_m = np.ma.array(Y,
                      mask = np.isnan(Y)
                     )

    #calculate correlation or covariance
    c = np.ma.dot(X_m.T, Y_m)/(nmat-1) #n-1 d.o.f.

    #if 2d, need to extract just the data
    if X.ndim == 2:
        c = c.data

#else if corr is true and scalar, calculate the correlation value
elif n == 1 and m == 1:
    #extract non-nan indices and drop nans
    ind = ~(np.isnan(X) + np.isnan(Y))
    X = X[ind]
    Y = Y[ind]

    #whiten arrays
    X = (X - X.mean()) / X.std(ddof = 1) #n-1 d.o.f.
    Y = (Y - Y.mean()) / Y.std(ddof = 1) #n-1 d.o.f.

    #calculate correlation and store in matrix
    c = np.dot(X,Y)/(np.sum(ind) - 1)

#if corr is true and matrix, loop through and calculate correlation values
else:
    #loop through each column of X and Y; calculate correlation
    c = np.zeros([n,m])
    for i in range(n):
        for j in range(m):
            #extract i column from X, j column from Y
            x = X[:,i]
            y = Y[:,j]

            #extract non-nan indices and drop nans
            ind = ~(np.isnan(x) + np.isnan(y))
            x = x[ind]
            y = y[ind]

            #whiten arrays
            x = (x - x.mean()) / x.std(ddof = 1) #n-1 d.o.f.
            y = (y - y.mean()) / y.std(ddof = 1) #n-1 d.o.f.

            #calculate correlation and store in matrix
            c[i,j] = np.dot(x,y)/(np.sum(ind) - 1)

#if inputs are dataframes, make outputs dataframes as well
if xdf and ydf:
    c = pd.DataFrame(c,
                     index = xcols,
                     columns = ycols
                    )

return c


#PCA function
def PCA(A, **kwargs):
'''
Function to calculate principal component analysis (PCA) on matrix A.
This function can handle missing data.

Parameters
----------
A : 2d array-like
    Array of variables, of shape [`n`,`p`]. Can contain missing
    data points.

corr : boolean
    Tells the function whether or not to calculate covariance
    or correlation matrix; if `False`, covariance matrix is given.
    Defaults to `False`.

Returns
-------
lam : array-like
```

```
        Vector of eigenvalues (sorted), of length `p`.

    U : 2d array-like
        Vector of eigenvectors (sorted), of shape [`p`,`p`].
    '''

    #calculate covariance matrix
    c = cmat(A, A, **kwargs)
    l, V = eig(c)

    #sort eigenvalues and eigenvectors
    lam = np.abs(np.sort(l)[::-1])
    sort_ind = np.argsort(l)[::-1]
    U = V[:,sort_ind]

    return lam, U

#RDA function
def RDA(Y, X, standardize_Y = True, scale_type = 2):
    '''
    Function to perform redundancy analysis (RDA); that is, perform a
    constrained principal component analysis (PCA) on Y values predicted
    from a multiple linear regression (MLR) on X. This function can
    handle missing data.

    Parameters
    ----------
    Y : array-like
        Array of response variables, of shape [`n`,`p`]. Can
        contain missing data points.

    X : array-like
        Array of control variables, of shape [`n`,`m`]. Can
        contain missing data points.

    standardize_Y : boolean
        Tells the function whether or not to standardize (i.e., whiten) the
        Y matrix. If `False`, the function only subtracts the mean from Y
        (i.e., it does not divide by the standard deviation), resulting in
        covariance rather than correlation. Defaults to `True`.

    scale_type : int
        Integer telling the function what scaling type to use; either 1 or 2.
        Following Legendre and Legendre (1998) notation, if `scale_type` is 2,
        then site scores and loadings are scaled by the square root of the
        eigenvalues. If `scale_type` is 1, then the biplot scores are scaled
        by the square root of the fractional eigenvalues. It is suggested to
        use `scale_type = 2` when `X` contains mostly continuous variables.
        Devaults to `2`.

    Returns
    -------
    lam : array-like
        Array of sorted eigenvalues where the first `p` entries are the
        canonical values (i.e., constrained) and the second `p` entries are
        the non-canonical values (i.e., residuals). Length `2*p`.

    U_c : 2d array-like
        Array of sorted canonical loadings, of shape [`p`,`p`]. If `scale_type`
        is 1, then U_c is simply the eigenvectors. If `scale_type` is 2, then
        loadings are scaled by the square root of the eigenvalues.

    bp_scores : 2d array-like
        Array of biplot (i.e., species) scores, of shape [`m`,`p`]. If `scale_type`
        is 1, then the biplot scores are scaled by the square root of the
        fractional eigenvalues.

    F : 2d array-like
        Array of site scores, of shape [`n`,`p`].

    Z : 2d array-like
        Array of constrained site scores, of shape [`n`,`p`].

    '''
    #calcualte matrix shapes
    n,p = np.shape(Y)
    _,m = np.shape(X)

    #standardize data:
    #whiten control variables
    X = (X - np.mean(X,0))/np.std(X,0,ddof=1)

    #whiten or center response variables
    if standardize_Y is True:
        #whiten response variables
        Y = (Y - np.mean(Y,0))/np.std(Y,0,ddof=1)
    else:
        Y = Y - np.mean(Y,0)

    #perform multiple linear regression
    B, Yhat, Yr = MLR(Y, X, standardize = False)

    #perform PCA on Yhat (canonical)
    lam_c, U_c = PCA(Yhat, corr = False)

    #perform PCA on Yr (non-canonical)
    lam_nc, U_nc = PCA(Yr, corr = False)

    #combine into single eigenvalue vector and eigenvector matrix
    lam = np.append(lam_c, lam_nc)
    U = np.append(U_c, U_nc, axis = 1)

    #calculate site scores (F) and fitted site scores (i.e., sample scores; Z)
    F = np.dot(Y, U_c)
    Z = np.dot(Yhat, U_c)

    #if type 2 scaling, rescale U_c; F; and Z
    if scale_type == 2:
        sqLam_c = np.eye(p, p) * (lam_c**0.5)
        msqLam_c = np.eye(p, p) * (lam_nc**-0.5)

        U_c = np.dot(U_c, sqLam_c)
        F = np.dot(F, msqLam_c)
        Z = np.dot(Z, msqLam_c)

    #calculate explanatory variable scores ("biplot scores")
    corXZ = cmat(X, Z, corr = True)

    if scale_type == 1:
        D = np.eye(p, p) * np.sqrt(lam_c / np.sum(lam))
        bp_scores = np.dot(corXZ.T,D)

    elif scale_type == 2:
        bp_scores = corXZ.T

    #convert everything to dataframes and include index
    U_c = pd.DataFrame(U_c, index = Y.columns)
    bp_scores = pd.DataFrame(bp_scores, index = X.columns)
    F = pd.DataFrame(F, index = Y.index)
    Z = pd.DataFrame(Z, index = Y.index)

    return lam, U_c, bp_scores, F, Z

#RMA regression function
def rma_regression(X,Y):
    '''
    Function for performing reduced major axis regression.

    Parameters
    ----------
    X : array-like
        Array of x values

    Y : array-like
```

```
            Array of y values

        Returns
        -------
        b0 : float
            Intercept value

        b1 : float
            Slope value

        r : float
            Correlation coefficient
        '''

        #drop missing values
        ind = ~(np.isnan(X) | np.isnan(Y))
        X = X[ind]
        Y = Y[ind]

        n = len(X)

        SXY = np.sum((X-X.mean())*(Y-Y.mean()))/(n-1)
        SXX = np.sum((X-X.mean())**2)/(n-1)
        SYY = np.sum((Y-Y.mean())**2)/(n-1)

        #calculate correlation coefficient, r
        r = SXY/((SXX)**0.5 * (SYY)**0.5)

        b1 = np.sign(r)*(SYY/SXX)**0.5
        b0 = Y.mean() - b1*X.mean()

        #calculate parameter uncertainty
        Yhat = b0 + b1*X
        SY2X = np.sum((Y-Yhat)**2)/(n-2)
        SSXX = SXX*(n-1)
        b1_std = np.sqrt(SY2X/SSXX)
        b0_std = np.sqrt(SY2X*(1/n + X.mean()/SSXX))

        return [b0, b1, r, b0_std, b1_std]
```

0.2 Define all functions for plotting

The second step is to define all functions that will be used to make all figures for the manuscript.

```
In [3]: #define function to plot river basins on map, color coded by variable value
        def plot_basins_colored(shps, pts, cvar,
                                ax = None,
                                cmap = 'jet',
                                proj = 'robin',
                                vmin = 0,
                                vmax = 1,
                                msize = 100):
            '''
            Function to plot river basins (and sample collection points) on a global map
            and color code by a given variable value.

            Parameters
            ----------
            shps : GeoDataFrame
                Geodataframe containing the basin outline polygon shapefiles and the
                color-code variable.

            pts : GeoDataFrame
                Geodataframe containing the sample collection point shapefiles (pour points)
                and the color-code variable.

            cvar : str
                Variable to use for color coding

            ax : mpl.AxisSubplot or None
                Axis to plot on, or `None`. If `None`, creates an axis. Defaults to `None`.

            cmap : str
                Colormap to use for filling in basin colors. Defaults to "jet".

            proj : str
                Global map projection to use. Defaults to "robin".

            vmin : int or float
                Minimum value for color scale, defaults to 0.

            vmax : int or float
                Maximum value for color scale, defaults to 1.

            msize : int or float
                Size of pour point markers, defaults to 100.

            Returns
            -------
            ax : mpl.AxisSubplot
                Updated matplotlib axis containing data.
            '''

            #make axis if necessary
            if ax is None:
                fig, ax = plt.subplots(1,1)
            #import global continent data
            world = gpd.read_file(gpd.datasets.get_path('naturalearth_lowres'))

            #plot continent outlines
            world.to_crs({'proj': proj}).plot(linewidth = 0.25,
                                              edgecolor='lightgray',
                                              color='lightgrey',
                                              ax = ax,
                                             )

            #plot river data
            shps.to_crs({'proj': proj}).dropna(subset = [cvar]).plot(column = cvar,
                                                                     vmin = vmin,
                                                                     vmax = vmax,
                                                                     linewidth = 0.25,
                                                                    edgecolor = 'black',
                                                                    ax = ax,
                                                                    cmap = cmap,
                                                                    legend = True
                                                                   )

            #plot again as circles to better show small basins
            pts.to_crs({'proj': proj}).dropna(subset = [cvar]).plot(marker = 'o',
                                                                    markersize = msize,
                                                                    column = cvar,
                                                                    vmin = vmin,
                                                                    vmax = vmax,
                                                                    linewidth = 0.25,
                                                                    edgecolor = 'black',
                                                                    ax = ax,
                                                                    cmap = cmap,
                                                                   )

            return ax

        #MLR plotting function
        def meas_pred_plot(X,Y,Yhat,ax = None, **kwargs):
            '''
            Function to plot measured vs. predicted response variables resulting
            from multiple linear regression (MLR) analysis. Predicted responses
            are what will be used for redundancy analysis (RDA).

            Parameters
            ----------
            Y : array-like
                Array of measured values, of length `n_sam`.
```

```
Yhat : array-like
    Array of predicted values, of length `n_sam`.

ax : mpl.AxesSubplot or None
    Axis to store plot to; if `None`, function creates an axis. Defaults
    to `None`.

Returns
-------
ax : mpl.AxesSubplot
    Axis containing plot.
'''

#extract shape
n,k = np.shape(X)

#make axis if necessary
if ax is None:
    fig, ax = plt.subplots(1,1)

#extract max and min values
xmin = np.min([Y.min(), Yhat.min()])
xmax = np.max([Y.max(), Yhat.max()])
dx = xmax - xmin

#make data scatterplot
ax.scatter(Y, Yhat, **kwargs)

#make 1:1 line
lims = np.array([xmin, xmax])
ax.plot(2*lims,2*lims,
       color = 'k',
       linewidth = 2
       )
#calculate RMSE and plot
rmse = (np.mean((Y-Yhat)**2))**0.5

ax.plot(2*lims, 2*lims + rmse,
       color = 'k',
       linewidth = 0.5
       )

ax.plot(2*lims, 2*lims - rmse,
       color = 'k',
       linewidth = 0.5
       )

#calculate R2
```

```
    r = cmat(Y.values, Yhat.values, corr = True)
    R2 = r**2
    R2adj = 1 - ((1 - R2)*(n-1))/(n - k - 1)

    #add RMSE and R2 text
    text = 'RMSE = %.2f; R2_adj = %.2f' %(rmse, R2adj)
    ax.text(0.05, 0.9, text, transform = ax.transAxes)

    #set limits and labels
    ax.set_xlim([xmin - 0.1*dx, xmax + 0.1*dx])
    ax.set_ylim([xmin - 0.1*dx, xmax + 0.1*dx])
    ax.set_xlabel(r'measured value (Y)')
    ax.set_ylabel(r'predicted value ($\hat{Y}$)')

    return ax

#RDA plotting function
def RDA_plot(bp_scores, U, S, lam, ax = None, **kwargs):
    '''
    Function to plot redundancy analysis (RDA) biplot, showing site scores,
    species scores, and loadings. This function can take either fitted
    (i.e., Z) or non-fitted (i.e., F) site scores.

    Parameters
    ----------
    bp_scores :
        Array of biplot (i.e., species) scores, of shape [`m`,`p`]. If `scale_type`
        is 1, then the biplot scores are scaled by the square root of the
        fractional eigenvalues.

    U :
        Array of sorted canonical loadings, of shape [`p`,`p`]. If `scale_type`
        is 1, then U_c is simply the eigenvectors. If `scale_type` is 2, then
        loadings are scaled by the square root of the eigenvalues.

    S :
        Array of either site scores (F) or fitted site scores (Z), of shape
        [`n`,`p`].

    lam :
        Array of sorted eigenvalues where the first `p` entries are the
        canonical values (i.e., constrained) and the second `p` entries are
        the non-canonical values (i.e., residuals). Length `2*p`.

    ax : mpl.AxesSubplot or None
        Axis to store plot to; if `None`, function creates an axis. Defaults
        to `None`.
    Returns
    -------
    ax : mpl.AxesSubplot
        Axis containing plot.
    '''

    #extract lengths
    n,p = np.shape(S)
    m,_ = np.shape(bp_scores)

    #if things are dataframes, extract data and names separately
    if type(bp_scores) is pd.DataFrame:
        cvs = bp_scores.index.values
        bp_scores = bp_scores.values

    if type(U) is pd.DataFrame:
        rvs = U.index.values
        U = U.values

    if type(S) is pd.DataFrame:
        sams = S.index.values
        S = S.values

    #make axis if necessary
    if ax is None:
        fig, ax = plt.subplots(1,1)

    #extract max and min values of site scores
    smin = np.min(S[:,:2])
    smax = np.max(S[:,:2])
    ds = smax - smin

    #extract max and min values of loadings / bp_scores
    xmin = np.min([np.min(bp_scores[:,:2]), np.min(U[:,:2])])
    xmax = np.max([np.max(bp_scores[:,:2]), np.max(U[:,:2])])
    dx = xmax - xmin

    #calculate scalar to multiple loadings / bp_scores by for clarity
    sc = np.abs(np.round(ds/dx))

    #plot site scores
    ax.scatter(S[:,0], S[:,1], **kwargs)

    #add bp_scores as black arrows
    for i in range(0,m):
        ax.arrow(0, 0, sc*bp_scores[i,0], sc*bp_scores[i,1],
                 head_width=0.1,
                 head_length=0.2,
                 fc='k',
                 ec='k'
                )

        #add text if the name exists
        try:
            ax.text(1.1*sc*bp_scores[i,0], 1.1*sc*bp_scores[i,1],
                    cvs[i],
                    verticalalignment = 'center',
                    horizontalalignment = 'center',
                   )
        except NameError:
            pass

    #add loadings as red arrows
    for i in range(0,p):
        ax.arrow(0, 0, sc*U[i,0], sc*U[i,1],
                 head_width=0.1,
                 head_length=0.2,
                 fc=[0.7,0.1,0.0],
                 ec=[0.7,0.1,0.0]
                )

        #add text if the name exists
        try:
            ax.text(1.1*sc*U[i,0], 1.1*sc*U[i,1],
                    rvs[i],
                    color = [0.7,0.1,0.0],
                    verticalalignment = 'center',
                    horizontalalignment = 'center',
                   )
        except NameError:
            pass

    #set axis limits
    sclim = np.ceil(np.max(np.abs([sc*xmin, sc*xmax, smin, smax])))

    ax.set_xlim([-sclim, sclim])
    ax.set_ylim([-sclim, sclim])

    #add label text
    pct1 = 100*lam[0]/np.sum(lam)
    xlab = 'PC1 (%.0f pct. variance explained)' %(pct1)

    pct2 = 100*lam[1]/np.sum(lam)
    ylab = 'PC2 (%.0f pct. variance explained)' %(pct2)

    ax.set_xlabel(xlab)
```

```
    ax.set_ylabel(ylab)


    return ax

#correlation table plotting function
def corr_boxes_plot(C, p,
                    ax = None,
                    size_scale = 500,
                    cmap = 'jet',
                    alpha = 0.05):
    '''
    Function for plotting correlations as colored boxes where the size and color
    correspond to correlation strength.

    Paramters
    ---------
    c : array-like
        Covariance or correlation matrix, of shape [`p`,`m`]. If both X
        and Y are 1-dimensional, c is a scalar.

    p : array-like
        Matrix of ordinary least squares p-values calculated using sm.OLS.

    ax : mpl.AxesSubplot or None
        Axis to store plot to; if `None`, function creates an axis. Defaults
        to `None`.

    size_scale : int or float
        Maximum size of boxes to be made. Note, size scales with square of the
        correlation (i.e., R2 value).

    cmap : str
        Colormap to use for filling in basin colors. Defaults to "jet".

    alpha : float
        Significant p-value below which squares will be surrounded by a thick black
        box.

    Returns
    -------
    ax : mpl.AxesSubplot
        Axis containing plot
    '''

    #make axis if it doesn't exist
    if ax is None:
        fig, ax = plt.subplots(1,1, figsize = (5,10))
```

```
#extract shapes and labels
n,m = np.shape(C)

x_labels = C.columns.values
y_labels = C.index.values[::-1]

#make X and Y grids
X,Y = np.meshgrid(range(m),range(n))

#flatten all arrays
X = X.flatten()
Y = Y.flatten()[::-1]
cvals = C.values.flatten()
ps = p.values.flatten()


#plot colored squares
sc = ax.scatter(X, Y,
                s = size_scale * cvals**2,
                marker = 's',
                c = cvals,
                cmap = cmap,
                vmin = -1,
                vmax = 1
               )

#plot black boxes around statistically significant relationships
ind = ps <= alpha

boxes = ax.scatter(X[ind], Y[ind],
                   s = size_scale * cvals[ind]**2,
                   marker = 's',
                   facecolor = 'None',
                   edgecolor = 'k',
                   linewidth = 2
                  )

#add axis tick labels
ax.set_xticks(range(m))
ax.set_xticklabels(x_labels.tolist(), rotation = 90)

ax.set_yticks(range(n))
ax.set_yticklabels(y_labels.tolist())

#add colorbar
plt.colorbar(sc, label = 'Correlation coefficient (r)')
```

```
        return ax
```

0.3 Perform statistical analyses

0.3.1 Step 1: Import data

Import the following files

1) Catchment outline shapefiles generated in ArcGIS,
2) File containing all biomarker isotope data, reported as averages for each sample type within each basin (sample_data_average.csv),
3) File containing all catchment property data (unweighted_statistics.csv),
4) file containing all spatially resolved catchment properties weighted by flow-length above sampling location (weighted_statistics.csv).

Generate log transformed variables and extract control and response variables to be used for statistical analyses.

```
In [4]: #import all files

        #sample (isotope) data
        sam_df = pd.read_csv('input_data/sample_data_average.csv',
                             index_col = 0,
                            )

        # sam_df = pd.read_csv('input_data/sample_data_all.csv',
        #                      index_col = 0,
        #                     )

        #used for saving all vs. averaged figures separately
        fig_folder = 'average_data'
        # fig_folder = 'all_data'

        #catchment properties
        stats_df = pd.read_csv('input_data/unweighted_statistics.csv',
                               index_col = 0,
                              )

        #spatially weighted catchment properties
        wstats_df = pd.read_csv('input_data/weighted_statistics.csv',
                                index_col = 0,
                                header = [0, 1] #hierarchical columns
                               )

        #make list of e-folding weighting distances (used in ArcGIS)
        ws = [50000,
              10000,
              5000,
              1000,
              750,
              500,
              250,
        #           100 #omit 100km since this is less than the resolution of most raster datasets
             ]

        #make log-transformed variables to test if any relationships require data transformation
        #sample variables
        sam_df['log_POC_14C'] = np.log10(sam_df['POC_14C'])
        sam_df['log_FA_14C'] = np.log10(sam_df['FA_14C'])
        sam_df['log_lignin_14C'] = np.log10(sam_df['lignin_14C'])

        #unweighted catchment statistics
        stats_df['log_tss_yield'] = np.log10(stats_df['tss_yield'])
        stats_df['log_POC_yield'] = np.log10(stats_df['POC_yield'])
        stats_df['log_area'] = np.log10(stats_df['area'])
        stats_df['abs_lat'] = np.abs(stats_df['lat'])

        #weighted catchment statistics
        for w in ws:
            col = 'w_' + str(w) + '_km'
            wstats_df.loc[:, (col, 'log_P_mean')] = np.log10(wstats_df.loc[:, (col, 'P_mean')])
```

```
/usr/local/lib/python3.7/site-packages/ipykernel_launcher.py:40: RuntimeWarning: divide by zero
/usr/local/lib/python3.7/site-packages/ipykernel_launcher.py:41: RuntimeWarning: divide by zero
/usr/local/lib/python3.7/site-packages/ipykernel_launcher.py:42: RuntimeWarning: divide by zero
```

0.3.2 Step 2: Determine optimum e-folding distance for statistical analyses

For a given set of control and response variables, loop through each e-folding weighting distance, w, determine the multiple liner regression (MLR) r^2 value. Take the value that maximizes MLR r^2 to be the "optimum" value and use this for all subsequent statistical analyses.

```
In [5]: #set control variables
        #include non-weighted catchment properties
        stats_cvs = [
        #               'long', #does not statistically correlate with any response variable
        #               'lat', #use absolute values
        #               'area', #use logged version
                      'runoff',
        #               'tss_yield', #use logged version
        #               'POC_yield', #use logged version
                      'perm_cont',
                      'perm_discont',
                      'log_tss_yield',
                      'log_POC_yield',
        #               'log_area' #does not statistically correlate with any response variable
        #               'abs_lat', #remove due to near perfect covariance with MAT
                    ]

        #include weighted catchment properties
        wstats_cvs = [
                      'elev_mean',
        #               'slope_mean', #does not statistically correlate with any response variab
        #               'f_floodplain', #does not statistically correlate with any response vari
                      'f_anthro',
                      'T_mean',
                      'T_CV',
        #               'P_mean', #use logged version
                      'P_CV',
                      'soilcarbonC_mean',
                      'NPPB_mean',
                      'tauC_total_mean',
                      'tauB_soil_mean',
                      'log_P_mean'
                     ]

        #list response variables
        rvs = [
               'POC_d13C',
        #        'POC_Fm', #use age instead of Fm since tau in years
               'POC_14C',
               'FA_d13C',
        #        'FA_Fm', #use age instead of Fm since tau in years
               'FA_14C',
        #        'FA_uggOC',
        #        'lignin_Fm', #use age instead of Fm since tau in years
               'lignin_14C',
        #        'lignin_uggOC',
        #        'log_POC_14C',
        #        'log_FA_14C',
        #        'log_lignin_14C'
              ]

        #define empty series to store resutling r^2 values
        R2_vals = pd.DataFrame(index = ws, columns = rvs)

        #loop through each w value, perform MLR, and store statistics
        for w in ws:

            #make merged dataframe with all data for a given w value
            col = 'w_' + str(w) + '_km'

            #make merged control variables dataframe
            all_stats_df = pd.merge(stats_df[stats_cvs], wstats_df[col][wstats_cvs],
                                    left_index = True,
                                    right_index = True,
                                   )

            #make total dataframe
            df = pd.merge(sam_df, all_stats_df,
                          left_index = True,
                          right_index = True,
                         )

            #standardize (whiten) the data since variables have drastically different dynamic ra
            dfst = (df - df.mean(axis=0))/df.std(axis = 0, ddof = 1) #n-1 d.o.f.

            #extract X (control) and Y (response) matrices
            X = dfst[stats_cvs + wstats_cvs + ['sample_type']]
            Y = dfst[rvs]

            #perform MLR
            B, Yhat, Yr = MLR(Y, X, standardize = False)

            #ensure nan where missing to prevent spurriously high correlation values
            Yhat[np.isnan(Y)] = np.nan
            Yr[np.isnan(Y)] = np.nan

            #calculate adjusted R2 values
            n,k = np.shape(Y)
            r = cmat(Y.values, Yhat.values, corr = True)

            R2 = r**2
            R2adj = 1 - ((1 - R2)*(n-1))/(n - k - 1)

            #store in matrix
            R2_vals.loc[w,:] = np.diag(R2adj)
```

0.4 Step 3: Perform all statistical analyses

Using an optimal e-folding distance of 500km, perform all subsequent statistical analyses and export resulting tables.

```
In [6]: #make data frame of all variables to be used for final MLR analysis

        #set best-fit weighting e-folding distance
        # wbest = 250
        wbest = R2_vals.mean(axis=1).idxmax()
        col = 'w_' + str(wbest) + '_km'

        #make merged control variables dataframe
        all_stats_df = pd.merge(stats_df[stats_cvs], wstats_df[col][wstats_cvs],
                              left_index = True,
                              right_index = True,
                             )

        #make total dataframe
        df = pd.merge(sam_df, all_stats_df,
                      left_index = True,
                      right_index = True,
                     )

        #standardize (whiten) the data since variables have drastically different dynamic ranges
        dfst = (df - df.mean(axis=0))/df.std(axis = 0, ddof = 1) #n-1 d.o.f.

        #extract X (control) and Y (response) matrices
        X = dfst[stats_cvs + wstats_cvs + ['sample_type']]
        Y = dfst[rvs]

        #perform MLR
        B, Yhat, Yr = MLR(Y, X, standardize = False)

        #perform RDA
        lam, U, bp_scores, F, Z = RDA(Y,X,scale_type=2)

        #make table of correlation coefficients and save
        C = cmat(Y, X, corr = True)
        C.to_csv('output_data/' + fig_folder + '/correlation_matrix.csv')

        #make table of correlation p-values (Pearson, OLS) and save
        p = pvals(Y, X)
        p.to_csv('output_data/' + fig_folder + '/p-value_matrix.csv')

        #make table of constrained site scores
        Z = pd.DataFrame(Z, index = Y.index)
        Z.to_csv('output_data/' + fig_folder + '/constrained_site_scores.csv')

        #make table of biplot (species) scores
        B = pd.DataFrame(bp_scores, index = X.columns)
        B.to_csv('output_data/' + fig_folder + '/biplot_scores.csv')

        #make table of loadings
        U = pd.DataFrame(U, index = Y.columns)
        U.to_csv('output_data/' + fig_folder + '/response_loadings.csv')
```

```
/usr/local/lib/python3.7/site-packages/numpy/core/fromnumeric.py:2542: FutureWarning: Method .pt
  return ptp(axis=axis, out=out, **kwargs)
```

0.5 Make Figures

0.5.1 Fig. 1 | Riverine biomarker 14C ages.

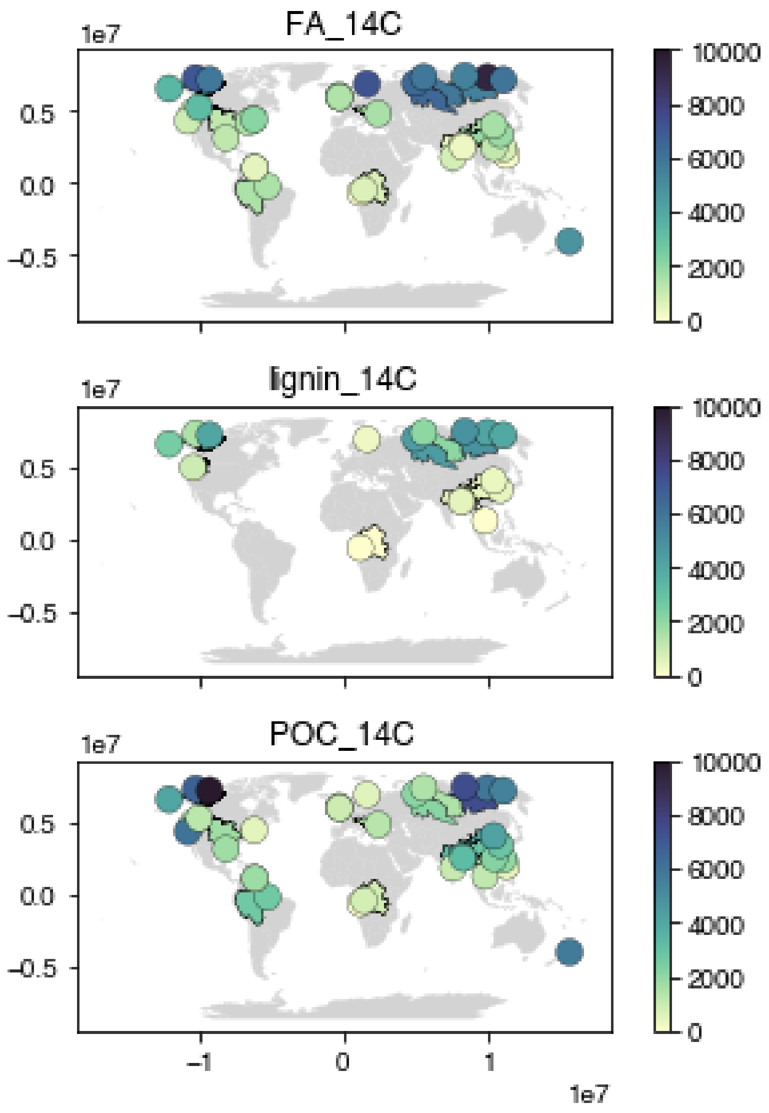
Catchment areas of all rivers analyzed in this study color-coded by a, plant-wax fatty acid and b, lignin phenol 14C ages (Supplementary Information Table 1). Rivers with catchment areas smaller than 30,000 km2 are shown as colored circles for clarity. Legend above panel a applies to both panels. c, Biomarker ages as a function of the absolute latitude at the river mouth, showing both fatty acids (black cirlces) and lignin phenols (white squares).

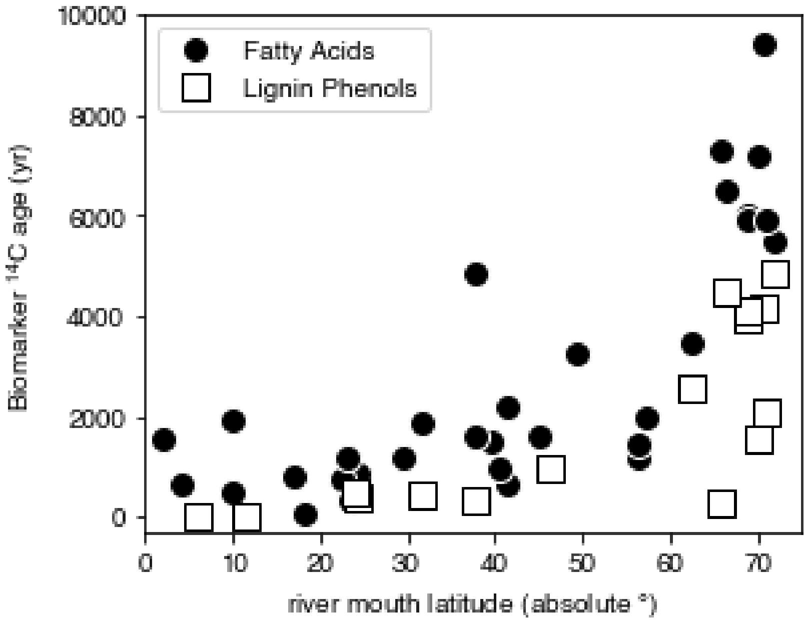
Also included:

0.5.2 Extended Data Fig. 5 | Riverine POC 14C ages.

Catchment areas of all rivers analyzed in this study color-coded by bulk POC 14C ages (Supplementary Information Table 1). Rivers with catchment areas smaller than 30,000 km2 are shown as colored circles for clarity. Legend is the same as in Fig. 1.

```
In [7]: #set projection and colormap
        proj = 'robin'
        cmap = cmo.cm.deep
        vmin = 0
        vmax = 10000
        msize = 100

        #import shapefile dataframe and make points dataframe
        shps = gpd.GeoDataFrame.from_file('river_outlines/All_Catchments_updated.shp')

        #replace shape with linestring to reduce size
        # shps['geometry'] = shps.boundary

        #calculate river means, extract isotope data, and merge with GeoDataFrame
        geodf = pd.merge(sam_df, stats_df,
                         left_index = True,
                         right_index = True,
                        )

        df_av = geodf.groupby(df.index).mean()
        df_av.reset_index()
        shps = shps.merge(df_av, on = 'Descriptio')

        #also calculate sample location points to better show small basins
        pts = gpd.GeoDataFrame(df_av,
                               geometry = gpd.points_from_xy(df_av.long, df_av.lat),
                              )
        pts.crs = {'init': 'epsg:4326', 'no_defs': True}

        #make figure
        fig,ax = plt.subplots(3,1,
                              figsize = (5,6),
                    sharex = True,
                    sharey = True
                    )

#loop through and plot each
cvars = ['FA_14C', 'lignin_14C','POC_14C']

for i, cvar in enumerate(cvars):
    #plot each variable
    ax[i] =  plot_basins_colored(shps, pts, cvar,
                                  ax = ax[i],
                                  cmap = cmap,
                                  proj = proj,
                                  vmin = vmin,
                                  vmax = vmax,
                                  msize = msize
                                 )

    #add title to each axis
    ax[i].set_title(cvar)

plt.tight_layout()
plt.show()

#save figure
fig.savefig('figures/' + fig_folder + '/maps.pdf',
            transparent = True,
            bbox_inches='tight'
           )

#make additional figure for latitude plot
figb, axb = plt.subplots(1,1, figsize = (4.5,3.5))

#plot fatty acid data
axb.scatter(np.abs(df_av['lat']), df_av['FA_14C'],
            s = 100,
            facecolor = 'k',
            edgecolor = 'w',
            label = 'Fatty Acids'
            )

#plot lignin data
axb.scatter(np.abs(df_av['lat']), df_av['lignin_14C'],
            marker = 's',
            s = 100,
            facecolor = 'w',
            edgecolor = 'k',
            label = 'Lignin Phenols'
            )

#add labels and legend
axb.set_xlabel(r'river mouth latitude (absolute $\degree$)')
axb.set_ylabel(r'Biomarker $^{14}$C age (yr)')

axb.legend(loc = 'upper left')

#set limits
axb.set_xlim([0,75])
axb.set_ylim([-300, 10000])

plt.tight_layout()
plt.show()

#save figure
figb.savefig('figures/' + fig_folder + '/latitude_age.pdf',
            transparent = True,
            bbox_inches='tight'
           )
```

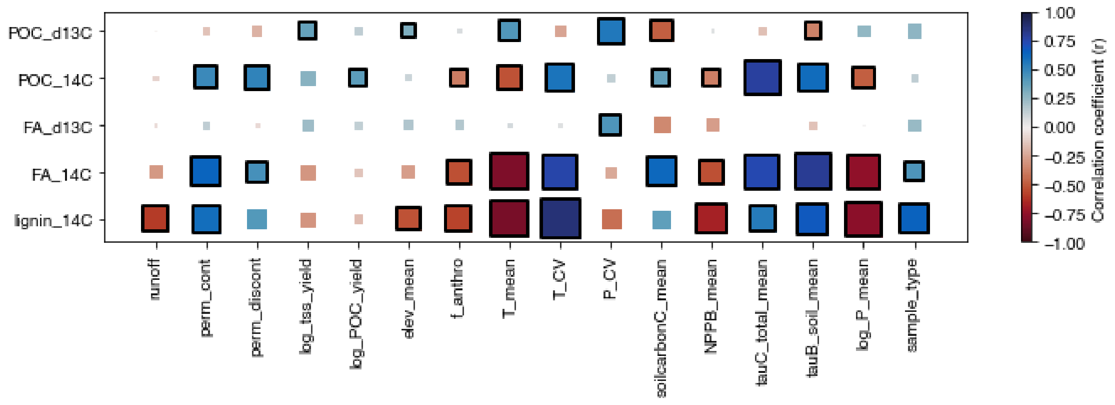

FA_14C
1e7
0.5
0.0
−0.5
10000
8000
6000
4000
2000
0
lignin_14C
1e7
0.5
0.0
−0.5
10000
8000
6000
4000
2000
0
POC_14C
1e7
0.5
0.0
−0.5
10000
8000
6000
4000
2000
0
−1
0
1
1e7

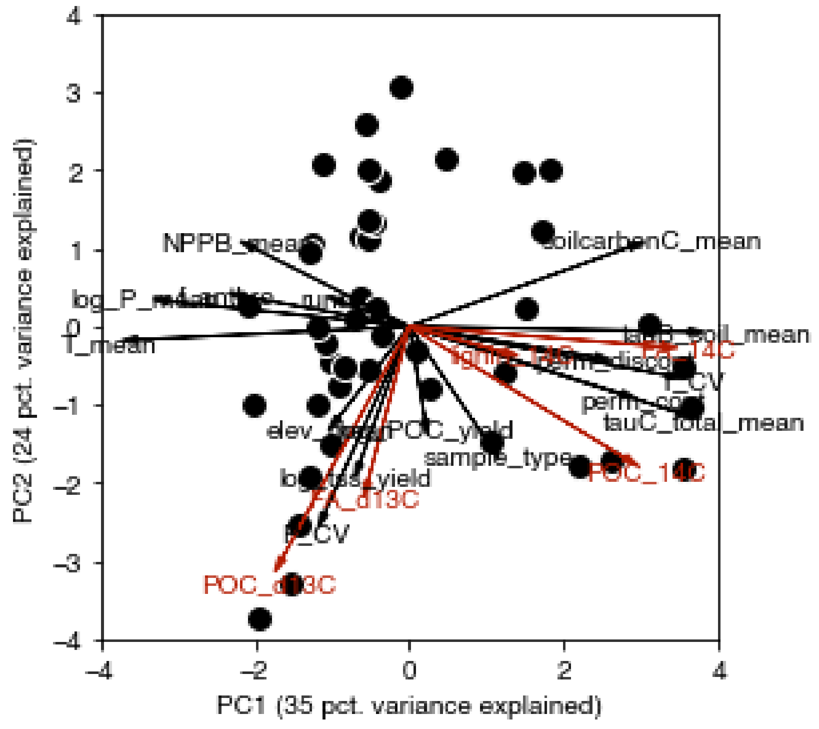


0.5.3 Fig. 2 | Multivariate statistical analysis.

a, Matrix of Pearson correlation coefficients (r values) between environmental control variables (x-axis) and biomarker or POC 14C or δ13C responses (y-axis) (Supplementary Information Table 3). Box sizes and colors correspond to the strength of the correlation (sizes: magnitude only; colors: magnitude and sign). Correlations that are significant at the $p = 0.05$ level are outlined with a thick black border. b, Redundancy analysis triplot showing the first (RDA1) and second (RDA2) canonical axes (Methods). labels show the percent of total sample variance explained by each axis. Environmental control variable scores are plotted as gray arrows, biomarker or POC 14C or δ13C response variable scores are plotted as red arrows, and individual sample scores are plotted as black circles. Environmental and response variable scores are scaled for visual clarity. Numbers and Roman numerals correspond to control and response variables as listed in panel a. Only control variables that are statistically significantly correlated with at least one response variable are shown.

```
In [8]: #make panel a, heat map box plot
        size_scale = 700
        fig,ax = plt.subplots(1,1,figsize = (10,3.5))
        # cmap = cmo.cm.curl_r
        cmap = cmo.cm.balance_r
        alpha = 0.05

        ax = corr_boxes_plot(C.T, p.T,
                             ax = ax,
                             size_scale = size_scale,
                             cmap = cmap,
                             alpha = alpha)

        plt.tight_layout()

        #save figure
        fig.savefig('figures/' + fig_folder + '/correlation_boxes.pdf',
                    transparent = True,
                    bbox_inches='tight'
                   )

        #make panel b, RDA plot
        fig2, ax2 = plt.subplots(1, 1,
                                 figsize = (4,4)
                                )

        RDA_plot(bp_scores, U, Z, lam,
                 ax = ax2,
                 s = 100,
                 facecolor = 'k',
                 edgecolor = 'w'
                )

        plt.tight_layout()

        #save figure
        fig2.savefig('figures/' + fig_folder + '/RDA.pdf',
                    transparent = True,
                    bbox_inches='tight'
                   )
```



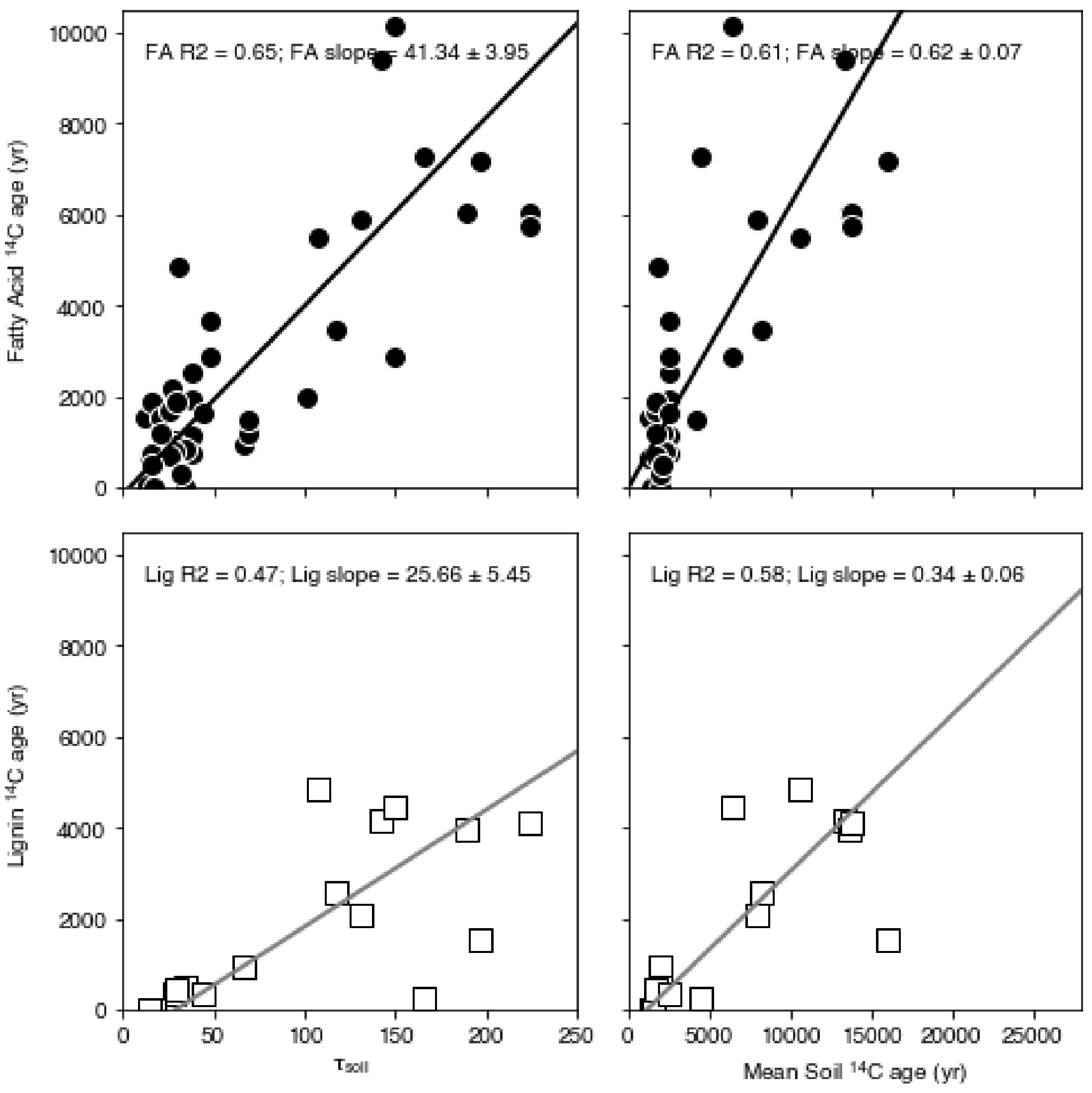


0.5.4 FIg. 3 | Relationships between soil turnover time and biomarker 14C ages.

a, Plant-wax fatty acid and b, lignin phenol 14C ages as a function of catchment soil turnover times, τ_{soil} (Bloom et al. 2016). Solid black lines are reduced major axis regression lines; reported values are the corresponding reduced major axis regression slopes and r^2 values (Methods). Uncertainty ($\pm 1\sigma$) is always smaller than marker points.

```
In [9]: depth = '0-100'
        var = 'Shi_Soil_'+depth+'_MCA'

        #add soil carbon age to dataframe (weighted at optimal e-folding length)
        dfsa = pd.merge(
            df,
            wstats_df['w_500_km'][var],
            left_index = True,
            right_index = True)

        #make figure
        fig, ax = plt.subplots(2, 2,
                               figsize = (7,7),
                               sharex = 'col',
                               sharey = True,
                               )
```

```
#add data
ax[0,0].scatter(dfsa['tauB_soil_mean'],dfsa['FA_14C'],
              s = 100,
              facecolor = 'k',
              edgecolor = 'w'
             )

ax[0,1].scatter(dfsa[var],dfsa['FA_14C'],
              s = 100,
              facecolor = 'k',
              edgecolor = 'w'
             )

ax[1,0].scatter(dfsa['tauB_soil_mean'],dfsa['lignin_14C'],
              marker = 's',
              s = 100,
              facecolor = 'w',
              edgecolor = 'k'
             )

ax[1,1].scatter(dfsa[var],dfsa['lignin_14C'],
              marker = 's',
              s = 100,
              facecolor = 'w',
              edgecolor = 'k'
             )

#add RMA regressions
fat_b0, fat_b1, fat_r, fat_b0_std, fat_b1_std = rma_regression(dfsa['tauB_soil_mean'],df
fas_b0, fas_b1, fas_r, fas_b0_std, fas_b1_std = rma_regression(dfsa[var],dfsa['FA_14C'])

lit_b0, lit_b1, lit_r, lit_b0_std, lit_b1_std = rma_regression(dfsa['tauB_soil_mean'],df
lis_b0, lis_b1, lis_r, lis_b0_std, lis_b1_std = rma_regression(dfsa[var],dfsa['lignin_14

xt = np.linspace(0,300,10)
fat_y = fat_b0 + fat_b1*xt
lit_y = lit_b0 + lit_b1*xt

xs = np.linspace(0,30000,10)
fas_y = fas_b0 + fas_b1*xs
lis_y = lis_b0 + lis_b1*xs

ax[0,0].plot(xt,fat_y,
           color = 'k',
           linewidth = 2
           )

ax[0,1].plot(xs,fas_y,
           color = 'k',
           linewidth = 2
           )

ax[1,0].plot(xt,lit_y,
           color = [0.5,0.5,0.5],
           linewidth = 2
           )

ax[1,1].plot(xs,lis_y,
           color = [0.5,0.5,0.5],
           linewidth = 2
           )

#add R2 text
fat_text = 'FA R2 = %.2f; FA slope = %.2f ś %.2f' %(fat_r**2, fat_b1, fat_b1_std)
fas_text = 'FA R2 = %.2f; FA slope = %.2f ś %.2f' %(fas_r**2, fas_b1, fas_b1_std)

lit_text = 'Lig R2 = %.2f; Lig slope = %.2f ś %.2f' %(lit_r**2, lit_b1, lit_b1_std)
lis_text = 'Lig R2 = %.2f; Lig slope = %.2f ś %.2f' %(lis_r**2, lis_b1, lis_b1_std)

ax[0,0].text(0.05, 0.9, fat_text, transform = ax[0,0].transAxes)
ax[0,1].text(0.05, 0.9, fas_text, transform = ax[0,1].transAxes)

ax[1,0].text(0.05, 0.9, lit_text, transform = ax[1,0].transAxes)
ax[1,1].text(0.05, 0.9, lis_text, transform = ax[1,1].transAxes)

#set limits and labels
ax[0,0].set_xlim([0, 250])
ax[0,1].set_xlim([0, 28000])

ax[0,0].set_ylim([0, 10500])

ax[1,0].set_xlabel(r'$\tau_{soil}$')
ax[1,1].set_xlabel(r'Mean Soil $^{14}$C age (yr)')

ax[0,0].set_ylabel(r'Fatty Acid $^{14}$C age (yr)')
ax[1,0].set_ylabel(r'Lignin $^{14}$C age (yr)')

plt.tight_layout()

#save figure
fig.savefig('figures/' + fig_folder + '/soil_biomarker_age_' + depth + 'cm.pdf',
            transparent = True,
            bbox_inches='tight'
           )
```



0.5.5 Fig. 4 | Environmental controls on soil turnover times and biomarker 14C ages.

Logarithmic plant-wax fatty acid 14C ages (black circles), lignin phenol 14C ages (white squares), and catchment soil turnover times (τ_{soil}; gray triangles) (Bloom et al. 2016) as functions of a, mean annual temperature (MAT) and b, logarithmic mean annual precipitation (MAP) (Supplementary Information Tables 1–2). Solid black, dashed black, and gray lines are fatty acid, lignin phenol, and τ_{soil} reduced major axis re-gression lines, respectively (Methods). Relationship r^2 values are as follows: Panel a: fatty acid 14C ages = 0.59, lignin phenol 14C ages = 0.55, τ_{soil} = 0.84; Panel b: fatty acid 14C ages = 0.55, lignin phenol 14C ages = 0.44, τ_{soil} = 0.74. Uncertainty ($\pm 1\sigma$) is always smaller than marker points.

Also included:

0.5.6 Extended Data Fig. 6 | TSS on soil turnover times and biomarker 14C ages.

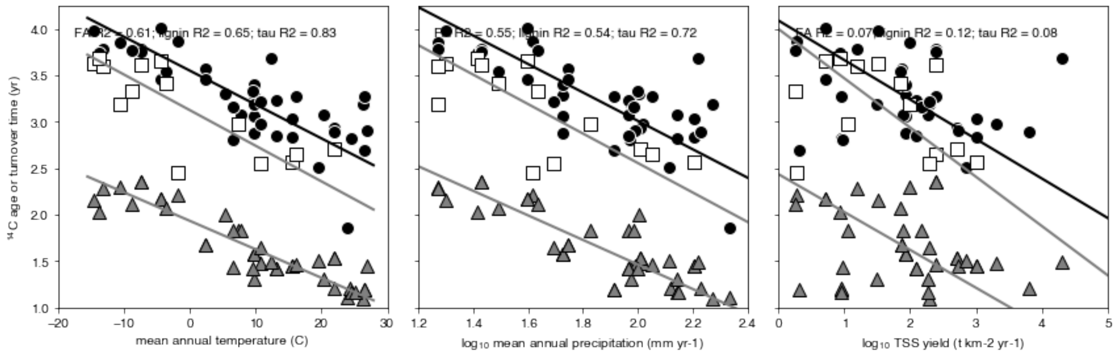
Logarithmic plant-wax fatty acid 14C ages (black circles), lignin phenol 14C ages (white squares), and catchment soil turnover times (τ_{soil}; gray triangles) (Bloom et al. 2016) as functions of total suspended sediment (TSS) yield (Supplementary Information Tables 1–2). Solid black, dashed black, and gray lines are fatty acid, lignin phenol, and τ_{soil} reduced major axis re-gression lines, respectively (Methods). Relationship r^2 values are as follows: fatty acid 14C ages = 0.07, lignin phenol 14C ages = 0.10, τ_{soil} = 0.11. Uncertainty ($\pm 1\sigma$) is always smaller than marker points.

```
In [10]: #make figure
         fig, ax = plt.subplots(1, 3,
                               figsize = (12,4),
                               sharey = True
                               )

         #make a column of MAT in deg. C (rather than Kelvin) for plotting purposes
         df['T_mean_C'] = df['T_mean'] - 273.15

         cvs = ['T_mean_C','log_P_mean','log_tss_yield']

         #loop through and plot data
         for i, cv in enumerate(cvs):

             #extract data and replace -inf values
             logFA = np.log10(df['FA_14C'])
             logFA.replace(-np.inf, np.nan, inplace = True)

             loglig = np.log10(df['lignin_14C'])
             loglig.replace(-np.inf, np.nan, inplace = True)

             logtau = np.log10(df['tauB_soil_mean'])
             logtau.replace(-np.inf, np.nan, inplace = True)


             #log FA 14C age
             ax[i].scatter(df[cv], logFA,
                           s = 100,
                           facecolor = 'k',
                           edgecolor = 'w'
                          )

             #log lignin 14C age
             ax[i].scatter(df[cv], loglig,
                           s = 100,
                           facecolor = 'w',
                           edgecolor = 'k',
                           marker = 's'
                          )
```

```
    #log soil tau
    ax[i].scatter(df[cv], logtau,
                  s = 100,
                  facecolor = [0.5, 0.5, 0.5],
                  edgecolor = 'k',
                  marker = '^'
                 )

    #add RMA regressions
    fa_b0, fa_b1, fa_r, fa_b0_std, fa_b1_std = rma_regression(df[cv], logFA)
    li_b0, li_b1, li_r, li_b0_std, li_b1_std = rma_regression(df[cv], loglig)
    tau_b0, tau_b1, tau_r, tau_b0_std, tau_b1_std = rma_regression(df[cv], logtau)

    print('FA slope = %.3f ś %.3f; lignin slope = %.3f ś %.3f; tau slope = %.3f ś %.3f'
          %(fa_b1, fa_b1_std, li_b1, li_b1_std, tau_b1, tau_b1_std))

    x = np.linspace(df[cv].min() - 1, df[cv].max() + 1,10)
    fa_y = fa_b0 + fa_b1*x
    li_y = li_b0 + li_b1*x
    tau_y = tau_b0 + tau_b1*x

    ax[i].plot(x, fa_y,
            color = 'k',
            linewidth = 2
            )

    ax[i].plot(x, li_y,
            color = [0.5,0.5,0.5],
            linewidth = 2
            )

    ax[i].plot(x, tau_y,
            color = [0.5,0.5,0.5],
            linewidth = 2
            )

    #add R2 text
    text = 'FA R2 = %.2f; lignin R2 = %.2f; tau R2 = %.2f' %(fa_r**2, li_r**2, tau_r**2
    ax[i].text(0.05, 0.9, text, transform = ax[i].transAxes)

#set axis limits
ax[0].set_ylim([1, 4.25])
ax[0].set_xlim([-20,30])
ax[1].set_xlim([1.2,2.4])
ax[2].set_xlim([0,5])

#set axis labels
ax[0].set_ylabel(r'$^{14}C$ age or turnover time (yr)')
```

```
         ax[0].set_xlabel(r'mean annual temperature (C)')
         ax[1].set_xlabel(r'$log_{10}$ mean annual precipitation (mm yr-1)')
         ax[2].set_xlabel(r'$log_{10}$ TSS yield (t km-2 yr-1)')

         plt.tight_layout()

         #save figure
         fig.savefig('figures/' + fig_folder + '/environmental_cross_plots.pdf',
                     transparent = True,
                     bbox_inches='tight'
                    )
```

```
/usr/local/lib/python3.7/site-packages/ipykernel_launcher.py:16: RuntimeWarning: divide by zero
  app.launch_new_instance()
/usr/local/lib/python3.7/site-packages/ipykernel_launcher.py:19: RuntimeWarning: divide by zero


FA slope = -0.037 ś 0.004; lignin slope = -0.038 ś 0.007; tau slope = -0.031 ś 0.002
FA slope = -1.531 ś 0.174; lignin slope = -1.587 ś 0.333; tau slope = -1.322 ś 0.111
FA slope = -0.425 ś 0.084; lignin slope = -0.531 ś 0.175; tau slope = -0.408 ś 0.077
```

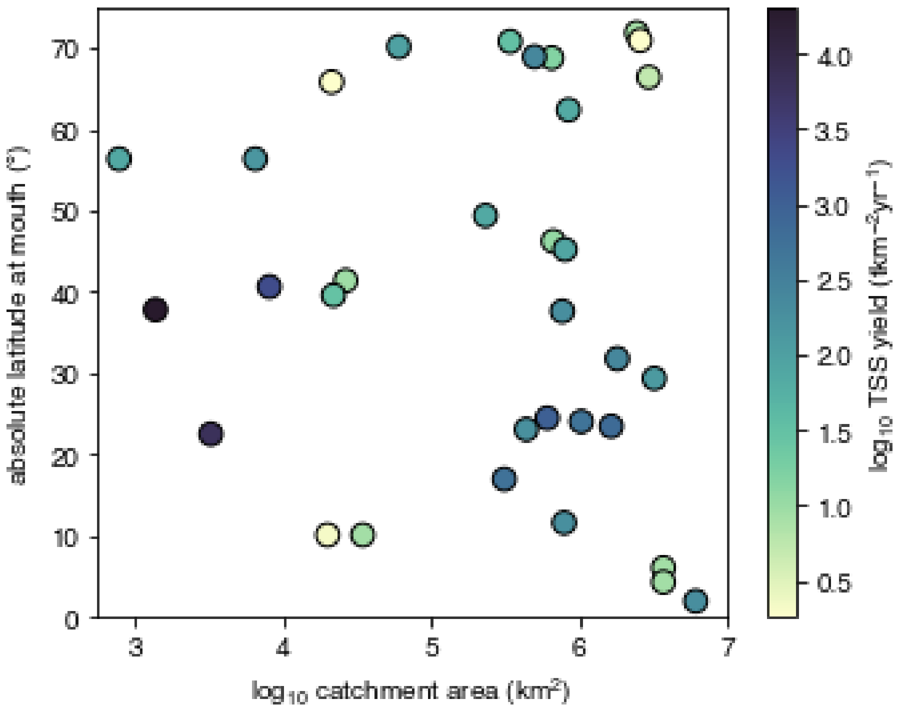


0.5.7 Extended Data Fig. 1 | Catchment latitude as a function of area.

Absolute value of latitude at the river mouth plotted as a function of catchment area for all river basins used in this study. Markers are additionally color coded by total suspended sediment (TSS) yield. There exists no correlation between catchment area and latitude at the river mouth, indicating that our sample set provides adequate global coverage and that covariance between these properties will not bias our results.

```
In [11]: #make plot
         fig, ax = plt.subplots(1,1,
                                figsize = (5,4),
                               )
```

```python
#set constants
cmap = cmo.cm.deep

#plot data
pts = plt.scatter(stats_df['log_area'], stats_df['abs_lat'],
                  marker = 'o',
                  s = 75,
                  c = stats_df['log_tss_yield'],
                  edgecolor = 'k',
                  cmap = cmap,
                 )

#add colorbar
plt.colorbar(pts, label = r'$log_{10}$ TSS yield ($t km^{-2} yr^{-1}$)')

#set limits
ax.set_xlim([2.75,7])
ax.set_ylim([0,75])

#add labels
ax.set_xlabel(r'$log_{10}$ catchment area ($km^2$)')
ax.set_ylabel(r'absolute latitude at mouth ($\degree$)')

plt.tight_layout()

#save figure
fig.savefig('figures/' + fig_folder + '/area_latitude_scatterplot.pdf',
            transparent = True,
            bbox_inches='tight'
           )
```

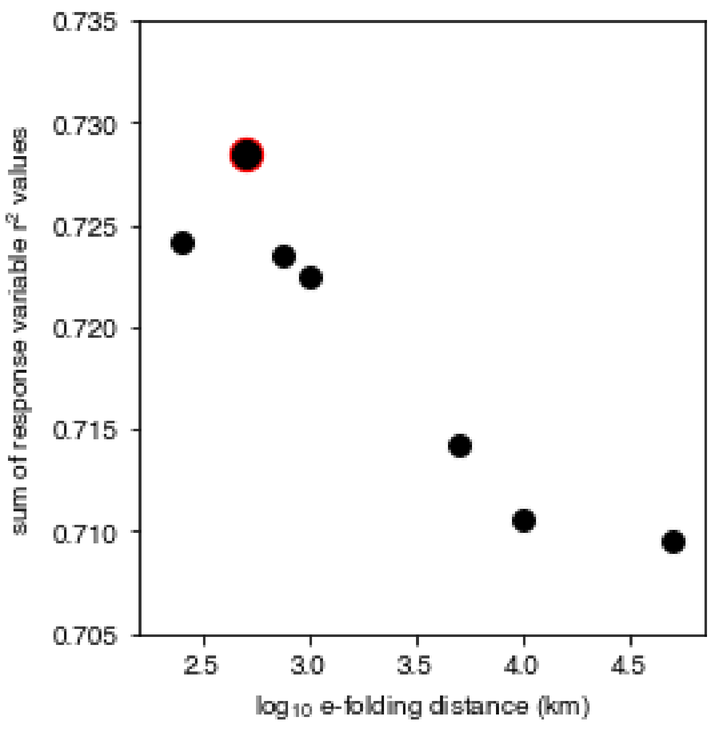


0.5.8 Extended Data Fig. 3 | MLR correlations with e-folding distance.

The sum of multiple linear regression (MLR) r^2 values for all POC and biomarker 14C and δ13C response variables plotted as a function of weighting e-folding distance. Higher e-folding distances result in control variable values that are more uniformly integrated across the river basin and vice versa. Here we choose 500 km as the optimal e-folding distance, as this value maximizes MLR r^2 values.

```
In [12]: #make plot
         fig, ax = plt.subplots(1,1,
                               figsize = (4,4)
                              )


         #add data
         x = np.log10(R2_vals.index)
         y = R2_vals.mean(axis = 1)
         ax.scatter(x,y,
                    s = 100,
                    facecolor = 'k',
                    edgecolor = 'w'
                   )

         #highlight "best" point with big red data point
         ybest = y.max()
```

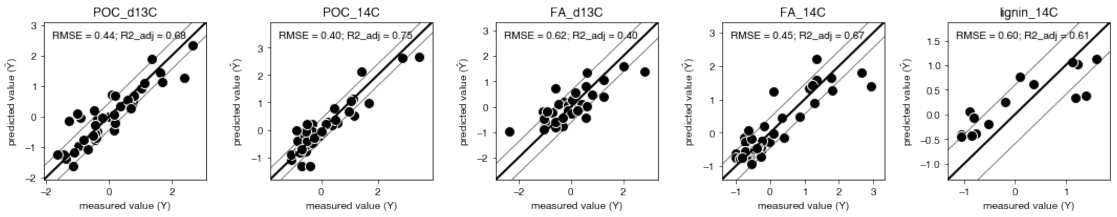
```
xbest = x[y==ybest]

ax.scatter(xbest,ybest,
           s = 150,
           facecolor = 'k',
           edgecolor = 'r',
          )

#set limits
ax.set_xlim([2.2,4.85])
ax.set_ylim([0.705, 0.735])

#set labels
ax.set_xlabel(r'log$_{10}$ e-folding distance (km)')
ax.set_ylabel(r'sum of response variable $r^2$ values')

plt.tight_layout()

#save figure
fig.savefig('figures/' + fig_folder + '/e_folding_r2.pdf',
            transparent = True,
            bbox_inches='tight'
           )
```



0.5.9 Extended Data Fig. 4 | Measured vs. MLR-predicted response variables

Multiple-linear-regression predicted a plant-wax fatty acid 14C age, b lignin phenol 14C age, c bulk POC 14C age, d plant-wax fatty acid 13C composition, and e bulk POC 13C composition as a function of their measured values. For each panel, thick black line is the 1:1 line and thin black lines represent ±1 root mean square error (RMSE) about the 1:1 line; adj. r^2 refers to the adjusted r^2 value that removes the effect of spurrious improvement that result form increasing the number of predictor variables. All measured and predicted values have been scaled such that each distribution has a mean of zero and a standard deviation of unity.

```
In [13]: #make figure
         fig,ax = plt.subplots(1,5,
                                  figsize = (15,3)
                                 )

         #ensure nan where missing to prevent spurriously high correlation values
         Yhat[np.isnan(Y)] = np.nan
         Yr[np.isnan(Y)] = np.nan

         #loop through each response variable and plot
         for i, rv in enumerate(rvs):
             #perform regression
             meas_pred_plot(X,Y[rv],Yhat[rv],
                            ax = ax[i],
                            s = 100,
                            facecolor = 'k',
                            edgecolor = 'w'
                           )

             #add title
             ax[i].set_title(rv)

         plt.tight_layout()

         #save figure
         fig.savefig('figures/' + fig_folder + '/MLR_meas_pred.pdf',
                     transparent = True,
                     bbox_inches='tight'
                    )
```

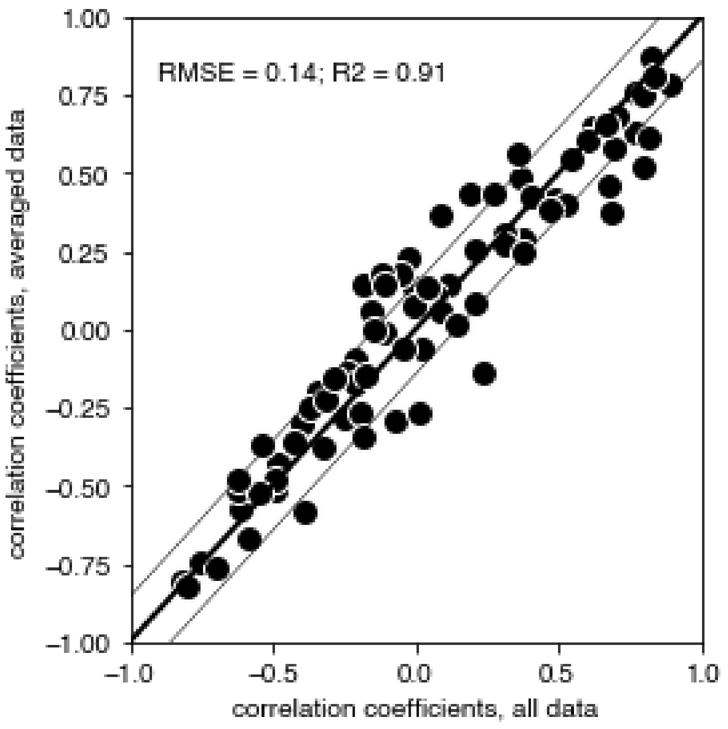


0.5.10 Extended Data Fig. 7 | All vs. averaged correlation coefficients

Cross plot of all correlation coefficients between environmental control variables and isotope response variables when either all or catchment-averaged response variable datasets are used (i.e., Supplementary Information Table 1 or 2). Thick black line is the 1:1 line and thin black lines represent ±1 root mean square error (RMSE) about the 1:1 line. Although some scatter exists, correlation coefficients are largely independent of the chosen response variable dataset used; that is, they plot close to the 1:1 line.

```
In [14]: #import datsets
         all_data = pd.read_csv('output_data/all_data/correlation_matrix.csv', index_col = 0)
         average_data = pd.read_csv('output_data/average_data/correlation_matrix.csv', index_col

         #make figure
         fig, ax = plt.subplots(1,1,
                                figsize = (4,4)
                               )
         #plot data
         ax.scatter(all_data, average_data,
                    s = 100,
                    facecolor = 'k',
                    edgecolor = 'w'
                   )

         #plot 1:1 line
         x = np.array([-1,1])
         y = np.array([-1,1])

         ax.plot(x, y, linewidth = 2, color = 'k')

         #plot RMSE
         rmse = (np.mean((all_data.stack()-average_data.stack())**2))**0.5
         ax.plot(x, y-rmse, linewidth = 0.5, color = 'k')
         ax.plot(x, y+rmse, linewidth = 0.5, color = 'k')

         #set limits and add labels
         ax.set_xlim([-1,1])
         ax.set_ylim([-1,1])

         ax.set_xlabel(r'correlation coefficients, all data')
         ax.set_ylabel(r'correlation coefficients, averaged data')

         #calculate R2 and add text
         b0, b1, r, b0_std, b1_std = rma_regression(all_data.stack(), average_data.stack())

         text = 'RMSE = %.2f; R2 = %.2f' %(rmse, r**2)
         ax.text(0.05, 0.9, text, transform = ax.transAxes)

         plt.tight_layout()
```

```
#save figure
fig.savefig('figures/all_vs_average_r.pdf',
            transparent = True,
            bbox_inches='tight'
           )
```



0.5.11 Calculation fraction of global water/TSS/POC in these watersheds

Use published values from Galy et al. (2015) and Peucker-Ehrenbrink (2009) to calcuate fractional global values.

```
In [15]: #POC flux
         cflux = stats_df['POC_yield']*stats_df['area'] #tC yr-1
         cflux_tot = cflux.sum()/1e6 #total MtC yr-1
         f_cflux = cflux_tot / (157 + 43) #mean total values (bio + petro) from Galy et al. 2015

         #sediment
         sedflux = stats_df['tss_yield']*stats_df['area'] #t yr-1
         sedflux_tot = sedflux.sum()/1e9 #total Gt yr-1
         f_sedflux = sedflux_tot / 18.5 #total value from Peucker-Ehrenbrink (2009)

         #water
         wflux = stats_df['runoff']*stats_df['area']/100000 #km3 yr-1
```

```
    wflux_tot = wflux.sum()
    f_wflux = wflux_tot / 38857 #total value from Peucker-Ehrenbrink (2009)

    #print results
    print('fraction of global water/TSS/POC export: %.2f, %.2f, %.2f' %(f_wflux, f_sedflux,
```

```
fraction of global water/TSS/POC export: 0.42, 0.29, 0.20
```

```
In [ ]:
```