Lab 6: Time Series Analysis

**Purpose**: The purpose of this lab is to establish a foundation for time series analysis of remotely sensed data, usually in the form of a temporally ordered stack of images. You will be introduced to concepts of smoothing, interpolation, linear modeling and phenology. At the completion of the lab, you will be able to perform analysis of multi-temporal data for determining trend and seasonality on a per-pixel basis.

**Prerequisites**: Lab 5

## Multi-temporal data in Earth Engine

Time series in Earth Engine are represented as image collections. This can make time series analysis complicated because

* There is a different time series in each pixel
* The size (length) of the time series vary across pixels
* Missing data may occur in any pixel at any time (e.g. due to cloud masking)

As a result of these complicating factors, analyzing time series in Earth Engine is unlike traditional methods. Specifically, use joins to define temporal relationships between collection items. As you will soon discover, it's possible to perform many traditional time series methods by mapping functions over joined collections.

First, some very basic notation. A scalar pixel at time *t* is given by *pt* and a pixel vector by **p***t*. An estimate is a variable with a hat on: e.g. the estimated pixel value at time is *p̂t*. A time series is just a collection of *N* pixels, sorted chronologically: {**p***t*; *t* = *t0*...*tN*}, where *t* might be in any units, *t0* is the smallest and *tN* is the largest such *t* in the series.

## Data preparation and preprocessing

The first step in analysis of time series data is to import data of interest and plot it at an interesting location.

1. **Load a time series of Landsat data**
   1. Search for Landsat 8 surface reflectance and import the "USGS Landsat 8 Surface Reflectance Tier 1" collection. Name it l8sr.
   2. Make a single point geometry with the geometry drawing tools and position the point in a location of interest. (How about an annual grassland or a deciduous forest?) Name the import roi.
2. **Filtering, masking and preparing bands of interest**
   1. preprocess the Landsat imagery by filtering it to the location of interest, masking clouds, and adding the variables in the model:

// This field contains UNIX time in milliseconds.

var timeField = 'system:time\_start';

// Function to cloud mask from the pixel\_qa band of Landsat 8 SR data.

// (From the Code Editor Examples > Cloud Masking)

function maskL8sr(image) {

// Bits 3 and 5 are cloud shadow and cloud, respectively.

var cloudShadowBitMask = 1 << 3;

var cloudsBitMask = 1 << 5;

// Get the pixel QA band.

var qa = image.select('pixel\_qa');

// Both flags should be set to zero, indicating clear conditions.

var mask = qa.bitwiseAnd(cloudShadowBitMask).eq(0)

.and(qa.bitwiseAnd(cloudsBitMask).eq(0));

// Return the masked image, scaled to reflectance, without the QA bands.

return image.updateMask(mask).divide(10000)

.select('B[0-9]\*')

.copyProperties(image, ['system:time\_start']);

}

// Use this function to add variables for NDVI, time and a constant

// to Landsat 8 imagery.

var addVariables = function(image) {

// Compute time in fractional years since the epoch.

var date = ee.Date(image.get(timeField));

var years = date.difference(ee.Date('1970-01-01'), 'year');

// Return the image with the added bands.

return image

// Add an NDVI band.

.addBands(image.normalizedDifference(['B5', 'B4']).rename('NDVI'))

// Add a time band.

.addBands(ee.Image(years).rename('t'))

.float()

// Add a constant band.

.addBands(ee.Image.constant(1));

};

// Remove clouds, add variables and filter to the area of interest.

var filteredLandsat = l8sr

.filterBounds(roi)

.map(maskL8sr)

.map(addVariables);

1. **Plot the time series at the location of interest**
   1. To visualize the data, make a chart at the location of interest. Add a linear trend line for reference (you'll compute that line soon):

// Plot a time series of NDVI at a single location.

var l8Chart = ui.Chart.image.series(filteredLandsat.select('NDVI'), roi)

.setChartType('ScatterChart')

.setOptions({

title: 'Landsat 8 NDVI time series at ROI',

trendlines: {0: {

color: 'CC0000'

}},

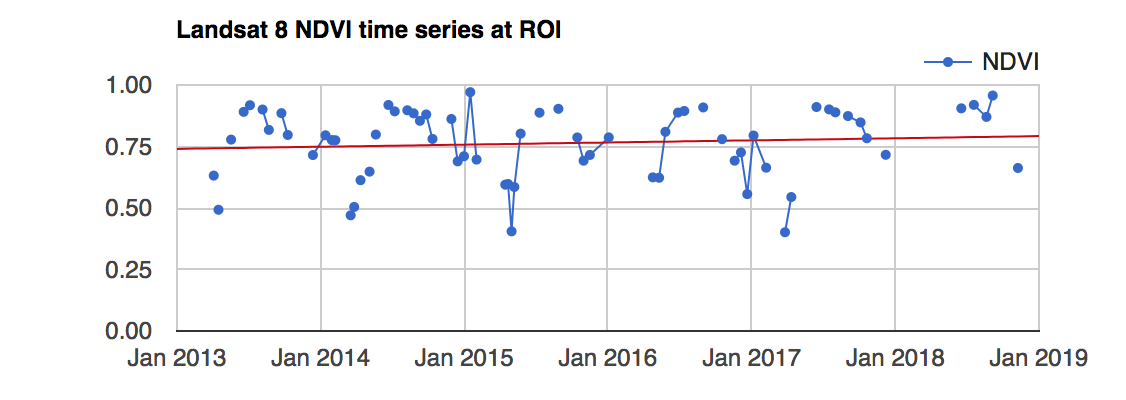
lineWidth: 1,

pointSize: 3,

});

print(l8Chart);

You should see something like this:



## Linear modeling of time

Lots of interesting analyses can be done to time series by harnessing the [linearRegression() reducer](https://developers.google.com/earth-engine/api_docs#eereducerlinearregression). For example,

1. **Estimate linear trend over time**

Consider the following linear model, where *et* is a random error:

*pt* = *β*0 + *β*1*t + et* (1)

This is the model behind the trendline added to the chart you just created. This model is useful for detrending data and reducing [stationarity](https://en.wikipedia.org/wiki/Stationary_process) in the time series ([Shumway and Stoffer 2017](http://www.stat.pitt.edu/stoffer/tsa4/tsaEZ.pdf)). For now, the goal is to discover the values of the *β*'s in each pixel.

* 1. To fit this trend model to the Landsat-based NDVI series using ordinary least squares (see Lab 5), use the linearRegression() reducer:

// List of the independent variable names

var independents = ee.List(['constant', 't']);

// Name of the dependent variable.

var dependent = ee.String('NDVI');

// Compute a linear trend. This will have two bands: 'residuals' and

// a 2x1 band called coefficients (columns are for dependent variables).

var trend = filteredLandsat.select(independents.add(dependent))

.reduce(ee.Reducer.linearRegression(independents.length(), 1));

// Map.addLayer(trend, {}, 'trend array image');

// Flatten the coefficients into a 2-band image

var coefficients = trend.select('coefficients')

.arrayProject([0])

.arrayFlatten([independents]);

The coefficients image is a two band image in which each pixel contains values for *β0* and *β1*.

* 1. Use the model to "detrend" the original NDVI time series:

// Compute a de-trended series.

var detrended = filteredLandsat.map(function(image) {

return image.select(dependent).subtract(

image.select(independents).multiply(coefficients).reduce('sum'))

.rename(dependent)

.copyProperties(image, [timeField]);

});

// Plot the detrended results.

var detrendedChart = ui.Chart.image.series(detrended, roi, null, 30)

.setOptions({

title: 'Detrended Landsat time series at ROI',

lineWidth: 1,

pointSize: 3,

});

print(detrendedChart);

1. **Estimate seasonality with a harmonic model**

Consider the following linear model, where *et* is a random error, *A* is amplitude, *ω* is frequency, and *φ* is phase:

*pt* = *β*0 + *β*1*t + A*cos(2π*ωt - φ*) + *et*

= *β*0 + *β*1*t + β*2cos(2π*ωt*) + *β*3sin(2π*ωt*) + *et* (2)

Note that *β*2 = *A*cos(*φ*) and *β*3 = *A*sin(*φ*), implying *A* = (*β*22 + *β*32)½ and *φ* = atan(*β*3/*β*2). (See [Shumway and Stoffer (2017)](http://www.stat.pitt.edu/stoffer/tsa4/tsaEZ.pdf) equations 4.1 - 4.2). To fit this model to the time series, set *ω*=1 (one cycle per unit time) and use ordinary least squares regression.

* 1. The setup for fitting the model is to first add the harmonic variables (the third and fourth terms of equation 2) to the image collection.

// Use these independent variables in the harmonic regression.

var harmonicIndependents = ee.List(['constant', 't', 'cos', 'sin']);

// Add harmonic terms as new image bands.

var harmonicLandsat = filteredLandsat.map(function(image) {

var timeRadians = image.select('t').multiply(2 \* Math.PI);

return image

.addBands(timeRadians.cos().rename('cos'))

.addBands(timeRadians.sin().rename('sin'));

});

* 1. Fit the model as with the linear trend, using the linearRegression() reducer:

var harmonicTrend = harmonicLandsat

.select(harmonicIndependents.add(dependent))

// The output of this reducer is a 4x1 array image.

.reduce(ee.Reducer.linearRegression({

numX: harmonicIndependents.length(),

numY: 1

}));

* 1. Plug the coefficients in to equation 2 in order to get a time series of fitted values:

// Turn the array image into a multi-band image of coefficients.

var harmonicTrendCoefficients = harmonicTrend.select('coefficients')

.arrayProject([0])

.arrayFlatten([harmonicIndependents]);

// Compute fitted values.

var fittedHarmonic = harmonicLandsat.map(function(image) {

return image.addBands(

image.select(harmonicIndependents)

.multiply(harmonicTrendCoefficients)

.reduce('sum')

.rename('fitted'));

});

// Plot the fitted model and the original data at the ROI.

print(ui.Chart.image.series(

fittedHarmonic.select(['fitted','NDVI']), roi, ee.Reducer.mean(), 30)

.setSeriesNames(['NDVI', 'fitted'])

.setOptions({

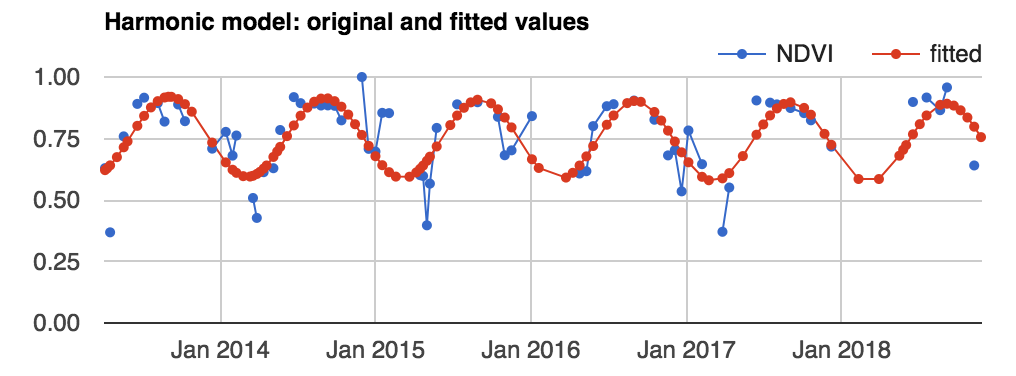
title: 'Harmonic model: original and fitted values',

lineWidth: 1,

pointSize: 3,

}));

You should see something like:



* 1. Although any coefficients can be mapped directly, it is useful and interesting to map the phase and amplitude of the estimated harmonic model. First, compute phase and amplitude from the coefficients, then map:

// Compute phase and amplitude.

var phase = harmonicTrendCoefficients.select('sin')

.atan2(harmonicTrendCoefficients.select('cos'))

// Scale to [0, 1] from radians.

.unitScale(-Math.PI, Math.PI);

var amplitude = harmonicTrendCoefficients.select('sin')

.hypot(harmonicTrendCoefficients.select('cos'))

// Add a scale factor for visualization.

.multiply(5);

// Compute the mean NDVI.

var meanNdvi= filteredLandsat.select('NDVI').mean();

// Use the HSV to RGB transform to display phase and amplitude.

var rgb = ee.Image.cat([

phase, // hue

amplitude, // saturation (difference from white)

meanNdvi // value (difference from black)

]).hsvToRgb();

Map.addLayer(rgb, {}, 'phase (hue), amplitude (sat), ndvi (val');

1. **More on harmonic models**

It's worth noting that a time series can be decomposed as the sum of sinusoids at different frequencies (See [Shumway and Stoffer (2017)](http://www.stat.pitt.edu/stoffer/tsa4/tsaEZ.pdf), equation 4.7). The harmonic model presented here can be easily extended in this manner by adding bands that represent higher frequencies (e.g. timeRadians.multiply(2).cos().rename('cos2') and the corresponding sin() band for a harmonic component with two cycles per year) and amending the harmonicIndependents variable accordingly. If you're feeling plucky, check out [this implementation](https://code.earthengine.google.com/2669122497313113fc4bb81bc8352828) of that idea for an arbitrary number of harmonic terms. While this will undoubtedly improve the goodness-of-fit of the model, many of the coefficients may be close zero. To estimate the importance of terms representing seasonality or higher-frequency harmonic behavior (e.g. [double-cropping](https://en.wikipedia.org/wiki/Multiple_cropping)), an F-statistic can be used when the model assumptions are satisfied. See for example, [Shumway and Stoffer (2017)](http://www.stat.pitt.edu/stoffer/tsa4/tsaEZ.pdf), equation 2.11.

## Autocovariance and autocorrelation

The [autocovariance](https://en.wikipedia.org/wiki/Autocovariance) of a time series refers to the dependence (specifically the [covariance](https://en.wikipedia.org/wiki/Covariance)) of values in the time series at time *t* with values at time *h* = *t* − *l*, where *l* is the lag. The autocorrelation is the covariance normalized by the standard deviations of the covariates. Specifically, assume our time series is stationary and define the autocovariance and autocorrelation according to [Shumway and Stoffer (2017)](http://www.stat.pitt.edu/stoffer/tsa4/tsaEZ.pdf) equations 1.27 and 1.26, respectively. Comparing values at time *t* to previous values is useful not only for computing autocovariance, but also for a variety of other time series analyses as well, as you'll see shortly.

To combine image data with previous values, in Earth Engine, the first step is to join the previous values to the current values. To do that, use a [join](https://developers.google.com/earth-engine/joins_intro) to create what we'll call a *lagged collection*:

1. **Create a lagged ImageCollection**

Consider the following function to create a lagged collection:

var lag = function(leftCollection, rightCollection, lagDays) {

var filter = ee.Filter.and(  
 ee.Filter.maxDifference({  
 difference: 1000 \* 60 \* 60 \* 24 \* lagDays,  
 leftField: timeField,   
 rightField: timeField  
 }),   
 ee.Filter.greaterThan({  
 leftField: timeField,   
 rightField: timeField  
 }));

return ee.Join.saveAll({

matchesKey: 'images',

measureKey: 'delta\_t',

ordering: timeField,

ascending: false, // Sort reverse chronologically

}).apply({

primary: leftCollection,

secondary: rightCollection,

condition: filter

});

};

This function joins a collection to itself, using a filter that gets all the images before but within a specified time difference (in days) of each image. That list of previous images within the lag time is stored in a property of the image called images, sorted reverse chronologically. For example, to create a lagged collection from the detrended Landsat imagery:

var lagged17 = lag(detrended, detrended, 17);

Why 17 days? Recall that the temporal cadence of Landsat is 16 days. Specifying 17 days in the join gets one previous image, but no more. To compute autocovariance or other interesting statistics, map functions over the lagged collection as in the following sections.

1. **Compute autocovariance and autocorrelation**
   1. The autocovariance reducer expects a set of one-dimensional arrays as input. So pixel values corresponding to time *t* need to be stacked with pixel values at time *t* − *l* as multiple bands in the same image. Consider the the following function for that purpose:

var merge = function(image) {

// Function to be passed to iterate.

var merger = function(current, previous) {

return ee.Image(previous).addBands(current);

};

return ee.ImageCollection.fromImages(

image.get('images')).iterate(merger, image);

};

...and use that function to merge the bands from the lagged collection:

var merged17 = ee.ImageCollection(lagged17.map(merge));

* 1. Now the bands from time *t* and *h* are all in the same image. Note that the band name of *ph* was the same as *pt*. During the merging process, it gets a '\_1' appended to it. Use a function to convert the merged bands to arrays with bands *pt* and *ph*, then reduce with the covariance reducer:

var covariance = function(mergedCollection, band, lagBand) {

return mergedCollection.select([band, lagBand]).map(function(image) {

return image.toArray();

}).reduce(ee.Reducer.covariance(), 8);

};

var lagBand = dependent.cat('\_1');

var covariance17 = ee.Image(covariance(merged17, dependent, lagBand));

* 1. The output of the covariance reducer is an array image, in which each pixel stores a 2x2 [variance-covariance](https://en.wikipedia.org/wiki/Covariance_matrix) array. The off diagonal elements are covariance, which you can map directly:

Map.addLayer(covariance17.arrayGet([0, 1]), {}, 'covariance (lag=17 days)');

* 1. The diagonal elements of the variance-covariance array are variances. Use this function to compute correlation from the variance-covariance array:

var correlation = function(vcArrayImage) {

var covariance = ee.Image(vcArrayImage).arrayGet([0, 1]);

var sd0 = ee.Image(vcArrayImage).arrayGet([0, 0]).sqrt();

var sd1 = ee.Image(vcArrayImage).arrayGet([1, 1]).sqrt();

return covariance.divide(sd0).divide(sd1).rename('correlation');

};

...and map the correlation:

var correlation17 = correlation(covariance17);

Map.addLayer(correlation17, {min: -1, max: 1},

'correlation (lag = 17 days)');

It's worth noting that you can do this for longer lags as well (try 34 days). Of course, that images list will fill up with all the images that are within *l* of *t*. Those other images are also useful, for example in fitting autoregressive models as described later.

## Cross-covariance and Cross-correlation

Cross-covariance is analogous to auto-covariance, except instead of measuring the correspondence between a variable and itself at a lag, it measure the correspondence between a variable and a covariate at a lag. Specifically, define the cross-covariance and cross-correlation according to [Shumway and Stoffer (2017)](http://www.stat.pitt.edu/stoffer/tsa4/tsaEZ.pdf) equations 1.30 and 1.31, respectively.

You already have all the code needed to compute cross-covariance and cross-correlation! But you do need a time series of another variable. Suppose we postulate that NDVI is related in some way to the precipitation before the NDVI was observed. To estimate the strength of this relationship (in every pixel), load precipitation, join, merge, and reduce as previously:

// Precipitation (covariate)  
var chirps = ee.ImageCollection('UCSB-CHG/CHIRPS/PENTAD');

// Join the t-l (l=1 pentad) precipitation images to the Landsat.

var lag1PrecipNDVI = lag(filteredLandsat, chirps, 5);

// Add the precipitation images as bands.

var merged1PrecipNDVI = ee.ImageCollection(lag1PrecipNDVI.map(merge));

// Compute and display cross-covariance.

var cov1PrecipNDVI = covariance(merged1PrecipNDVI, 'NDVI', 'precipitation');

Map.addLayer(cov1PrecipNDVI.arrayGet([0, 1]), {}, 'NDVI - PRECIP cov (lag = 5)');

// Compute and display cross-correlation.

var corr1PrecipNDVI = correlation(cov1PrecipNDVI);

Map.addLayer(corr1PrecipNDVI, {min: -0.5, max: 0.5}, 'NDVI - PRECIP corr (lag = 5)');

What do you observe from this result? Specifically, how can we do better? One possible drawback of this computation is that it's only based on five days of precipitation, whichever five days came right before the NDVI image. Perhaps precipitation in the month before the observed NDVI is relevant? Test that idea with the following:

// Join the precipitation images from the previous month

var lag30PrecipNDVI = lag(filteredLandsat, chirps, 30);

print(lag30PrecipNDVI);

var sum30PrecipNDVI = ee.ImageCollection(lag30PrecipNDVI.map(function(image) {

var laggedImages = ee.ImageCollection.fromImages(image.get('images'));

return ee.Image(image).addBands(laggedImages.sum().rename('sum'));

}));

// Compute covariance.

var cov30PrecipNDVI = covariance(sum30PrecipNDVI, 'NDVI', 'sum');

Map.addLayer(cov1PrecipNDVI.arrayGet([0, 1]), {}, 'NDVI - sum cov (lag = 30)');

// Correlation.

var corr30PrecipNDVI = correlation(cov30PrecipNDVI);

Map.addLayer(corr30PrecipNDVI, {min: -0.5, max: 0.5}, 'NDVI - sum corr (lag = 30)');

Observe that the only change is to the merge() method. Instead of merging the bands of the NDVI image and the covariate (precipitation), the entire list of precipitation is summed and added as a band (eliminating the need for iterate()).

As long as there is sufficient temporal overlap between the time series, these techniques can be extended to longer lags and longer time series. But watch out. There are some nuances to determining the statistical significance of correlation estimates. See [Shumway and Stoffer (2017)](http://www.stat.pitt.edu/stoffer/tsa4/tsaEZ.pdf) example 1.26 for details.

## Auto-regressive models

The discussion of autocovariance preceded this section in order to introduce the concept of lag. Now that you have a way to get previous values of a variable, it's worth considering auto-regressive models. Suppose that pixel values at time *t* depend in some way on previous pixel values. If you have observed significant, non-zero autocorrelations in a time series, this may not be a crazy assumption. Specifically, you may postulate a linear model such as:

*pt* = *β*0 + *β*1*pt-1* + *β*2*pt-2* + *et* (3)

To fit this model, you need a lagged collection as created previously except with a longer lag (e.g. *l* = 34 days). The next steps are to merge the bands, then reduce with the linear regression reducer.

1. Create a lagged collection, where the images list stores the two previous images:

var lagged34 = ee.ImageCollection(lag(filteredLandsat, filteredLandsat, 34));

1. Merge the bands of the lagged collection such that each image has bands at time *t* and bands at times *t*-1,..., *t* − *l*. Note that it's necessary to filter out any images that don't have two previous temporal neighbors:

var merged34 = lagged34.map(merge).map(function(image) {

return image.set('n', ee.List(image.get('images')).length());

}).filter(ee.Filter.gt('n', 1));

1. Fit the regression model using the linearRegression() reducer:

var arIndependents = ee.List(['constant', 'NDVI\_1', 'NDVI\_2']);

var ar2 = merged34

.select(arIndependents.add(dependent))

.reduce(ee.Reducer.linearRegression(arIndependents.length(), 1));

// Turn the array image into a multi-band image of coefficients.

var arCoefficients = ar2.select('coefficients')

.arrayProject([0])

.arrayFlatten([arIndependents]);

1. Compute the fitted values using an [expression](https://developers.google.com/earth-engine/image_math#expressions). Because this model is a function of previous pixel values, which may be masked, if any of the inputs to equation 3 are masked, the output of the equation will also be masked. That's why you should use an expression here, unlike the previous linear models of time.

// Compute fitted values.

var fittedAR = merged34.map(function(image) {

return image.addBands(

image.expression('beta0 + beta1 \* p1 + beta2 \* p2', {

p1: image.select('NDVI\_1'),

p2: image.select('NDVI\_2'),

beta0: arCoefficients.select('constant'),

beta1: arCoefficients.select('NDVI\_1'),

beta2: arCoefficients.select('NDVI\_2')

}).rename('fitted'));

});

1. Plot the results. Note the missing values that result from masked data.

print(ui.Chart.image.series(

fittedAR.select(['fitted', 'NDVI']), roi, ee.Reducer.mean(), 30)

.setSeriesNames(['NDVI', 'fitted'])

.setOptions({

title: 'AR(2) model: original and fitted values',

lineWidth: 1,

pointSize: 3,

}));

At this stage, note that the missing data has become a real problem. Any data point for which at least one of the previous points is masked or missing is also masked. It may be possible to avoid this problem by substituting the output from equation 3 (the modeled value) for the missing or masked data. Unfortunately, the code to make that happen is not straightforward. If you're feeling brave, check out [this solution](https://code.earthengine.google.com/c5fbadc5136d94e80ba428376b05d148).

## Assignment

[](http://creativecommons.org/licenses/by-sa/4.0/)

This work is licensed under a [Creative Commons Attribution 4.0 International License](http://creativecommons.org/licenses/by/4.0/).

Copyright 2015-2016, Google Earth Engine Team