
NIH Short Course: December 17-20, 2000
Nonlinear Time Series Analysis in Biology

Daniel Kaplan, Macalester College

Leon Glass, McGill University

Thomas Schreiber, MPIPKS

Computer Lab Exercises

Several lab exercises are presented here. Our intention is to cover the material for two labs in each day of the course. However, it is unlikely that you will be able to do all of the lab exercises each day. You may wish to focus on those topics of particular interest to you.

The software that implements the labs is written in the MATLAB language and needs to be run within the MATLAB system. Although some programs are written specifically for the labs, most of the programs are quite general and can be used in research. You are welcome¹ to use the nonlinear time series analysis software after the end of the course. The most reliable way to acquire it is from D. Kaplan's web site. (Since MATLAB system is a commercial product, you will have to make your own arrangements to acquire it.)

A more "industrial-strength" set of software, written in C, C++ and FORTRAN, is available at the TISEAN site. A project is underway to integrate this MATLAB software with the TISEAN software.²

¹The software will be distributed under the provisions of the GNU *copyleft*. This means that you can use it for free, but that there are certain restrictions in how you can redistribute or modify it.

²If you are interested in supporting or contributing to this project, please contact D. Kaplan.

1 Resetting curves for the Poincaré oscillator (LG)

One of the simplest models of a limit cycle oscillation is the Poincaré oscillator. The equations for this model are

$$\begin{aligned}\frac{dr}{dt} &= kr(1-r), \\ \frac{d\phi}{dt} &= 2\pi,\end{aligned}\tag{1}$$

where k is a positive parameter. Starting at any value of r , except $r = 0$, there is an evolution until $r = 1$. The parameter k controls the relaxation rate. In this laboratory we consider the relaxation in the limit $k \rightarrow \infty$.

There are two programs in this laboratory.

- **resetting(b)** This program computes the resetting curve (new phase versus old phase) for a stimulus strength b . The output is a matrix with 2 columns and 102 lines. There are two points just less than and just greater than $\phi = 0.5$. These points are needed especially for the case where $b > 1$.
- **poincare(phizer,b,tau,niter)** This program does an iteration of the periodically stimulated Poincaré oscillator, where **phizer** is the initial phase, **b** is the stimulation strength, **tau** is the period of the stimulation, and **niter** is the number of iterations. It is valid for $0 < \tau < 1$. The output consists of two arrays. **phi** is a listing of the successive phases during the periodic stimulation. **beats** is a listing of the number of beats that occur between successive stimuli.

How to run the programs

1. To compute the resetting curve for $b = 1.10$, you type
 \gg `[phi,phiprime]=resetting(1.10);`
2. To plot out the resetting curve just computed type
 \gg `plot(phi,phiprime,'*')`
3. To simulate periodic stimulation of the Poincaré oscillator type
 \gg `[phi,beats]=poincare(.3,1.13,0.35,100);`

This will generate two time series of 100 iterates from an initial condition of $\phi = 0.3$, with $b = 1.13$, and $\tau = 0.35$. The array **phi** is the successive phases during the stimulation. The array **beats** is the number of beats between stimuli.

- To display the output as a return map, type
`>> plot(phi(2:99),phi(3:100),'*')`

This plots out the successive phases of each stimulus as a function of the phase of the preceding stimulus. The points lie on a one-dimensional curve. The dynamics in this case are chaotic. In fact, what is observed here is very similar to what is actually observed during periodic stimulation of heart cell aggregates described in the first lecture of the course.

- To display the number of beats between stimuli, type
`>> plot(beats,'*')`
- The rotation number gives the ratio between number of beats and the number of stimuli during a stimulation. This is the average number of beats per stimulus. To compute the rotation number type
`>> sum(beats)/length(beats)`

Exercises

Try the following exercises.

- Use the program `resetting` to compute the resetting curves for several values of b in the range from 0 to 2. In particular determine the value of b at which the topology of the resetting curve changes.
- Determine whether or not the successive iterates of `phi` are periodic assuming different value of (b, τ) (use program `testperiod`). (i) What do you find for different values of b and τ ? What is the ratio of the number of stimuli to the number of action potentials? (ii) Find values for which there are different asymptotic behaviors depending on the initial condition? (iii) Find values that give quasiperiodic dynamics (for nonzero b) (iv) Can you find a period-doubling route to chaos?
- (Hard): Determine the dynamics over the (b, τ) parameter plane and draw a diagram with the results. You should get the diagram given in the notes in Fig. ??.
- (Research level). Consider the two dimensional Poincaré oscillator, Equation (1), with finite relaxation time (k is finite). Investigate the dynamics of this equation. What has been found out so far is in Glass, L., J. Sun. Periodic forcing of a limit cycle oscillator: Fixed points, Arnold tongues, and the global organization of bifurcations. Physical Review E 50, 5077-5084 (1994). Any good results on the following questions merit a research publication. (i) What are the dynamics where the period 1 orbit becomes unstable? You need to get some analytic results giving particular consideration to the

presence of sub- and super-critical Hopf bifurcations. (ii) How many stable orbits can exist simultaneously? Describe the different stable periodic orbits for some subset of parameter space. (iii) For what range of parameter values is there chaotic dynamics? (iv) Give an analytic proof of chaos in this example and explore the routes to chaos.

Do you agree that it is important to understand this example as well as the ionic mechanisms of heart cell aggregates to understand the effects of periodic stimulation of the aggregates?

2 Iterating Finite-Difference Equations (LG)

This gives instructions for running the programs to study the quadratic map

$$x_{i+1} = \mu(1 - x_i)x_i \quad (2)$$

using MATLAB.

There are 4 programs.

- `quadratic(xzero,mu,niter)` This program iterates the quadratic map. There are three input arguments: `xzero` is the initial condition; `mu` is the bifurcation parameter in Eq. 2; `niter` is the number of iterations. The output is a vector `y` of length `niter` containing the iterated values.
- `testperiod(y,epsilon,maxper)` This program determines if there is a periodic orbit in the sequence given by the vector `y` whose period is less than or equal to `maxper`. The convergence criterion is that two iterates of `y` are closer than `epsilon`. The output is the period `per`. If no convergence is found the output is `-1`.
- `bifurcation(mubegin,muend)` This plots the bifurcation diagram for 100 steps of μ between `mubegin` and `muend`.
- `cobweb(xzero,mu,nstep)` This program iterates the quadratic map. There are three input arguments: `xzero` is the initial condition; `mu` is the parameter; `nstep` is the number of iterations for which you will compute the cobweb.

2.1 How to run the programs

The following steps give an illustration example of how to run these programs.

1. To generate 100 iterations of the quadratic map with an initial condition of $x_1 = 0.5$, $\mu = 3.973$ type
`>> y=quadratic(0.5,3.973,100);`
2. To plot the time series from this iteration type
`>> plot(y,'+');`
3. To determine if there is a period of length less than or equal to 20 with a convergence of 0.00001 of the time series `y` type
`>> per=testperiod(y,.00001,20)`
 ; In this case there is no period and the program returns `per = -1`. If a value $\mu = 3.2$ had been used to generate the time series in the quadratic program, the program `testperiod` returns a value of `per = 2`.

4. To plot a cobweb diagram for the quadratic map with an initial condition of `xzero=0.3` and $\mu = 3.825$ with 12 steps, type
`>> cobweb(0.3,3.825,12);`

2.2 Exercises

The above programs essentially carry out Computer Projects 1-3 from Kaplan and Glass, pages 51-53 for the quadratic map. You are in a position to carry out the remainder of the Projects 4-5 using the quadratic map.

Project 4 asks you to compute the sequence of periodic orbits encountered as μ is increased. Try to find all ranges of μ that give periodic orbits up to period 6. As μ is increased you should be able to find windows that give periodic windows in the sequence 1,2,4,6,5,3,6,5,6,4,6,5,6. You will want to try to increment μ sufficiently finely to find the different periodic orbits. The sequence of periodic orbits is called the **universal** sequence. It is the same for all maps with a quadratic maximum and a single hump.

Project 5 asks you to determine Feigenbaum's number. Feigenbaum's number is defined as follows. Call Δ_n the range of μ values that give a period n orbit. Then Feigenbaum found that in a sequence of period-doubling bifurcations

$$\lim_{n \rightarrow \infty} \frac{\Delta_n}{\Delta_{2n}} = 4.4492 \dots$$

The constant, 4.6692 ... is now called **Feigenbaum's number**.

According to Feigenbaum, he initially discovered this number by carrying out numerical iterations on a hand calculator. As the period of the cycle gets longer, the range of parameter values over which a given period is found gets smaller. Therefore, it is necessary to think carefully about what is involved in the calculation. Try to numerically compute

$$\frac{\Delta_8}{\Delta_{16}}$$

and

$$\frac{\Delta_6}{\Delta_{12}}.$$

You will want to vary μ over a range of values. How should you locate the value of μ where the period changes?

The behaviors found for the quadratic map here are also found in other simple maps, complicated equations, and a variety of experimental systems. It is this **universal** behavior that has attracted the attention of physicists and others.

By making appropriate modifications in the MATLAB programs, you can adapt the programs so that they carry out similar computations for the single-humped sine map

$$x_{i+1} = \mu \sin(\pi x_i), \tag{3}$$

where $0 < \mu < 1$.

3 Lag embedding, Poincaré maps, Poincaré sections (DK)

Objectives

This lab demonstrates two important techniques in nonlinear time series analysis: processing data sampled from a continuous-time system to produce a discrete-time representation via the Poincaré section and Poincaré map; and constructing a multi-dimensional representation of a single signal using lag embedding. Some issues to examine are how the choice of cut in a Poincaré section affects the results and how the choice of embedding lag affects the gross shape of the “attractor.”

Data Sets

Several synthetic and real-world data sets are provided:

Rossler The full x, y, z state of the Rossler system is sampled by the program `makerossler`.

This integrates the differential equations of the Rossler system for whatever duration and whatever initial condition you specify. You can set the time between samples. For example,

```
>> [t,x,y,z] = makerossler(1000, [1 0 0], .1);
```

will integrate the Rossler equations starting at an initial condition $x = 1, y = 0, z = 0$ and sampling every 0.1 second.³

Several plots of the x, y, z data are easy to make: A time series plot of one variable:

```
>> plot(t,x)
```

The trajectory in the true state space

```
>> plot3(x,y,z)
```

You can use the rotation tool in the Figure window (the button with an arrow chasing it's own tail in the Figure window): press the button and then click and drag in the Figure window to change the perspective on the three dimensional plot.

To make an animation of the trajectory, use

```
>> comet3(x,y,z)
```

You can rotate this perspective using the rotation tool while the animation is in progress.

Lorenz Gives the full x, y, z state of the Lorenz system.

```
>> [t,x,y,z] = makelorenz(100,[2 0 20], .01);
```

Normal sinus rhythm The file `nsr.dat` contains an intracellular electrogram from normal sinus rhythm, provided by Nitish Thakor at Johns Hopkins University. You load this

³“Second” should really read “time unit.” There are no physical units specified in the Rossler equations.

time series with the command

```
>> load nsr.dat
```

which creates a variable `nsr`. You can plot it as a time series with

```
>> plot(nsr)
```

or, say, the first 1000 points with

```
>> plot(nsr(1:1000))
```

Since this is a single-dimensional measurement, you can't plot out a "state space" — first, you'll have to construct the space.

Ventricular fibrillation The file `vf.dat` contains an intracellular electrogram during ventricular fibrillation, again provided by Nitish Thakor. Load the data with the command

```
>> load vf.dat
```

which will create a variable `vf`.

Measles Month-by-month measurements of the number of measles cases in various cities are in the files `balt.dat`, `newyork.dat`, `detroit.dat`.

Tools

This lab is mainly about the visualization of time series and trajectories in real or reconstructed state spaces. In later labs we will deal with analysis and quantitative characterization of these trajectories.

The fundamental tools for visualization are, of course, plotting tools. We've already seen some of these: `plot`, `plot3`, `comet3`.

Another convenient plotting tool is

`scatplot(x,tau)` takes a scalar time series and makes a delay plot, a plot of $x(t+\tau)$ vs $x(t)$.

By default, τ is 1. For example:

```
>> scatplot(x)
```

or

```
>> scatplot(x,20)
```

Sections

To create a Poincaré section, the following tools can be used:

`[t,p = psection(traj,crossing level, plane, times)]` is used to make a classical cutting-plane Poincaré section. `traj` is the collection of time series describing the trajectory. The returned variables are: `t`, the times at which the plane was cut; `p`, the position in the trajectory space of each pass (in a negative direction) through the plane. It is a matrix where there is one variable in each column. (It is very boring to make a Poincaré section of a single variable! Try it sometime.) The other arguments, shown in *italics*, are optional. The "crossing level" and "plane" describe where the cutting plane is

located. By default, the plane cuts the first variable at the mean of that variable. For example, taking x , y , and z to be from the Rossler system, try

```
>> [tt,pp] = psection([x y z]);
```

Note that the three signals have been combined into one matrix to be handed off to `psection` and that `psection` returns two variables, the times of the crossings `tt` and the position `pp` in the full space of the crossings. There is one row for each crossing. Since the default cutting plane is the first variable, you'll see that the first column is constant. The other columns contain the information you want.

In this case the space being cut is three-dimensional, and the cut itself is two-dimensional. You can plot the places where the trajectory passes through the cutting plane with

```
>> plot(pp(:,2), pp(:,3), 'o')
```

(If you had used another cutting plane than the default one, you would have had to plot the points differently.) For example, try

```
>> scatter(pp(:,2))
```

The Poincaré map is the relationship between the position of one pass through the plane with the next pass through the plane. Plotting this out fully would require 4 dimensions. For the Rossler data — but not for all data generally — the passes lie practically on a one-dimensional curve. We can exploit this structure to create a Poincaré map as a two dimensional plot. Try

```
>> scatter(pp(:,2))
```

`dmax(data,timescale,thresh)` finds local maxima in a scalar time series.⁴ The `timescale` sets the meaning of “local.” For an oscillatory signal, it should be set to be about somewhat less than the time between peaks — the precise value usually isn't critical. The optional argument `thresh` is used to discard local maxima that are not large enough to be of interest to you. `dmax` returns the times and heights of the maxima detected. By plotting these out along with the original signal, you can adjust `timescale` as appropriate. For example,

```
>> [tt,aa] = dmax(balt,8);
```

The time scale has been set to 8, since this is somewhat less than the once-a-year peaks expected in measles.

```
>> plot(balt); hold on; plot(tt,aa,'o');hold off
```

The plot suggests that some of the small maxima have been missed. You can change the time scale to pick up these peaks if you like.

⁴Although `dmax` may not seem to be a Poincaré section, you might want to think of it as a section in the space constructed from the signal and its first derivative: the maximum of the signal corresponds to the plane where the derivative is zero.

Embedding

Lag embedding is the fundamental technique in nonlinear time series analysis. This takes a scalar signal $x(t)$ and turns it into a vector signal; at each time t the vector is $(x(t), x(t - \tau), \dots, x(t - (p - 1)\tau))$. The parameter p is called the embedding dimension, the parameter τ is the lag.

`lagembed(x,p,tau)` takes a vector `x` and returns a matrix that is the lag-embedding of embedding dimension `p` and lag `tau`, both of which should be integers.⁵ Try

```
>> traj = lagembed(balt,4,2);
```

Note that `lagembed` doesn't create any new information; it just re-organizes existing information. Each column of the matrix `traj` is the original signal shifted a bit in time (and with a bit cut off the ends of the signal).

Finally, a small program that keeps a “sub-space” of a collection of data stored as a matrix:

`keepPC(data,nkeep)` keeps the `nkeep` largest principal components of the cloud of `data`.

Questions

1. Embed the time series


```
>> simple = [1:10]';
```

 (making sure it's a column vector) for various embedding lags and dimensions. By looking at the matrix produced by `lagembed` you should be able to see exactly what is going on in lag embedding.
2. Plot the Lorenz data in the original x, y, z coordinates to see the shape of the attractor in the actual state space. Now use lag embedding on just a single coordinate to produce a reconstructed model of the attractor. Since we're visualizing the attractor, we're pretty much restricted to using a dimension of $p = 2$, so you can use `scatplot` to form and plot the embedding in one command. You might also want to use `lagembed` with a dimension of $p = 3$ along with `plot3`. Note that `plot3` takes three vectors as arguments, so you will have to do something like:


```
>> foo = embed(x,3,10);
>> plot3(foo(:,1), foo(:,2), foo(:,3));
```

Find an embedding lag that gives the most faithful-looking reconstruction of the trajectory; compare the reconstructed trajectory to the original `[x y z]`. Try lags that are very short, lags that are a fraction of the typical period of oscillation of the signal, and lags that are longer than the period of oscillation. Try embeddings of all three state variables individually.

⁵In principle, τ does not need to be an integer, but here we are dealing with sampled data.

The program `acf(data)` computes the autocorrelation function of a scalar signal. One rule of thumb for selecting an embedding lag is to pick the lag that corresponds to the first zero-crossing of the autocorrelation function or, for signals without regular oscillations, the lag at which the autocorrelation function falls to about $\frac{1}{e} \approx 0.37$. (You can use `acf(data,maxlag)` to restrict the plot to lags less than `maxlag`.)

3. Construct a Poincaré section of the Rossler data and generate a Poincaré map from this. How do the results depend on the way in which the Poincaré section is taken, that is, on the position and orientation of the cutting plane?

Create a Poincaré map of the Lorenz data. It is not as easy as it might seem to create a map that shows a simple structure. It's hard to find a good Poincaré cutting plane. Try both the classical Poincaré cut (`psection`) and local maxima of one signal.

4. Create lag embeddings of the real-world signals, picking the embedding lag to produce a “chaotic-looking” trajectory. Experiment with ways to take Poincaré sections of these reconstructions. For the cardiac data, a typical Poincaré section is measuring the reconstructed model state once per “beat.” For the measles data, the section is measuring things once per outbreak.
5. The use of principal components allows one to use a rather high embedding dimension and project this down into a lower-dimensional space while retaining much of the (linear) information in the signal. Try using a very short embedding lag — much shorter than you found in (1) — and a high embedding dimension. The product of the embedding dimension and embedding lag is called the “embedding window” and reflects the amount of the signal that is incorporated in each point in the embedding. Use `keepPC` to keep the two or three largest principal components of the embedding. Plot these out and compare them qualitatively to the results you got using the best embedding lag you found in (1).

4 Function Estimation and Information (DK)

Objectives

This lab will introduce some concepts from function estimation. The emphasis will be on how to construct dynamical models from data and how to use those models to generate synthesized data, make predictions, and measure the determinism of data.

There are, of course, many different types of models. This lab will focus on one simple type: the locally linear model. This is not the best type of model for all purposes, but it is generally very good for most purposes. It has the great advantage of having only one parameter that needs to be set: the number of neighbors to use in fitting the locally linear model. It is easy to find reasonable values of this parameter. Of course, you may also need to embed a time series in order to generate the model, so the embedding parameters are needed as well.

Whatever the technical architecture of a model, the abstract structure is the same. Given an input, the model produces an output that corresponds to the input. There are two basic steps:

Training In the training stage, often called *fitting*, we provide some inputs and corresponding outputs. The model “learns” from this training data how to translate from an input to an output.

Evaluation Using the trained model, we can now provide an input. The model, having already been trained, translates this input into an output.

This lab focusses on how to set up the training data, and various ways of evaluating the model.

The Training Data

We will be modeling dynamical systems. In a dynamical system, the present state determines the future state. Our data, when suitably processed and embedded, provide a measurement of the present state. We will call these measurements the “pre-image.” The system’s dynamics translate the pre-image into the future, which we’ll call the “post-image.”

In order to train the model, we need to provide the pre-images and the corresponding post-images. As a simple example, consider some data from the quadratic map:

```
>> load quad.dat
```

Plotting the first few points versus time shows the data look irregular

```
>> plot(quad(1:100))
```

but plotting $x(t+1)$ vs $x(t)$

```
>> scatterplot(quad)
```

shows how the past value, $x(t)$ determines the future value, $x(t+1)$. In this case, the state is a scalar and an appropriate state space is one dimensional.

Although the `quad` data is 1000 points long, let’s take just the first 500 points as the training data. First, we need to take the pre-image data:

```
>> pre = quad([1:500]);
```

This⁶ gives the state of the system at each time 1 to 500; there is one row for each time point.

Next, we need to take the post-images that correspond to each of the 500 pre-images. At each time, t , the post-image is $x(t + 1)$. So, our post-images are⁷

```
>> post = quad(1+[1:500]);
```

But, you object. The post-images are the same data as the pre-images. This is true. But each row in the post-image is different than the corresponding row in the pre-image. It is the nature of dynamics that the post-image at time t is the pre-image at time $t + 1$.

Let's do an example more generally. Suppose we take the x, y, z signals from the Lorenz system. These were generated with the command

```
>> [t,x,y,z] = makelorenz(100, [1 0 10], .1);
```

which you might need to re-run if you have deleted or changed the variables. The state of the system as it evolves over time can be represented by the matrix `state = [x y z]`. Each row in this matrix gives the state at a single time. Suppose we wish to take as our training data the first 999 points of this state:

```
>> pre = state([1:999], :);
```

The corresponding post-image is the same set of states, but offset by one time step⁸

```
>> post = state(1+[1:999], :);
```

Similarly, we might have a state reconstructed via an embedding of a measured signal. The embedding is a matrix and we construct the pre- and post-images for the training data in the same way as above.

Do keep in mind that if you have N rows in your state matrix, you will not be able to use all N of them as pre-images. The reason is that you need to have a post-image for each pre-image, and so you could use at most $N - 1$ of the rows as pre-images. Things can be even more restrictive, since we will often want to reserve some of our data for evaluation of the model.

Tools

The main program is `linpredict`. This is a very general program. The program takes at least 4 arguments:

pre The training pre-images.

⁶Of course I could have written the command as

```
>> pre = quad(1:500);
```

Writing `[1:500]` instead of `1:500` gives exactly the same result. However, later I'll be adding 1 to each of the points in `1:500`.

⁷Following up on the last footnote, note that `1+[1:500]` in the command below has the same meaning as `2:501`. Writing `1+1:500` would be equivalent to `2:500` which is not what we need.

⁸Actually, the offset doesn't need to be 1. It could be any integer, positive or negative. We'll explore this later.

post The training post-images.

k The number of neighbors to use in constructing the local linear model. This is the only parameter of the model, and it is quite easy to find a reasonable value for it. Here are the general principles:

- **k** must be larger than the dimension of the input space.
- **k** must be smaller than the length of the training set.
- **k** The smaller **k** is, the more local the model and the more jagged the model can be. The larger **k** is, the smoother the model.
- **k** If **k** is similar in size to the number of training data points, the model is essentially globally linear.

newpre another set of pre-images; the ones that you want to evaluate the model at.

The first three arguments set up the training of the model. The fourth argument evaluates the trained model at the given data points. This is potentially confusing, so let me emphasize the point: `linpredict` trains and evaluates the model all in one go.

Here's an example using the quadratic data.

```
>> pre = quad([1:500]);
```

```
>> post = quad(1+[1:500]);
```

Since the state space is one dimensional — there's just one column in `pre` — we need $k \geq 2$. We'll take

```
>> k=10;
```

The three variables `pre`, `post`, and `k` provide all the information needed to train the model. But we also need to tell `linpredict` where to evaluate the model. We'll going to specify another set of pre-images, and `linpredict` will return to us the model's corresponding post-images. Since the pre-image space is one-dimensional, we can in this case make a graph of post-image vs pre-image. For the data themselves this is

```
>> plot(pre,post, '.')
```

which sketches out the shape of the quadratic function. Let's make a similar graph for the model, by asking `linpredict` to evaluate the model at a series of points from -0.5 to 1.5 :

```
>> newpre = [(-.5):.1:1.5];
```

(N.B.: `newpre` should have one column for each dimension in the state space. Since in our example the state space has dimension 1, `newpre` has one column. `linpredict` would complain if you made `newpre` a row vector.) Now, we simultaneously train and evaluate the model

```
>> modelvals = linpredict(pre,post,k,newpre);
```

The variable `modelvals` now contains the post-images that correspond to each of the points in `newpre`. We can make a plot comparing the data with the model as follows:

```
>> plot(pre,post, '.', newpre, modelvals, 'x');
```

Note that the locally linear model follows the data pretty closely, but it also extends the

model beyond the domain of the training data. This extrapolation is linear, because for the extrapolation, no matter how far out we go, the “locale” in “locally linear” is the same points at the extreme of the training data.

In general, the pre-images and post-images will not be one-dimensional, and we will not be able to make plots like the above. Instead, we will need to be able to use other techniques to judge whether the fitted function is reasonable and to use the fitted function to study the dynamics of the experimental system we are interested in.

Evaluating the Fitted Function

What are we to do with this fitted function? The function comprises our model of the dynamics of the system; although it isn't in the form of an algebraic expression, it plays the same role in the dynamics:

$$x_{t+1} = f(x_t)$$

the fitted function is the $f(\cdot)$ that we estimate from the dynamics. We can therefore use it to synthesize data, just as we could if we had an algebraic form for $f(\cdot)$.

Here's an example based on data from the Lorenz system:

```
>> [t,x,y,z] = makelorenz(100, [1 0 10], .1);
```

```
>> state = [x y z];
```

```
>> pre = state([1:500],:);
```

```
>> post = state(1+[1:500],:);
```

```
>> k = 50;
```

Now we have all the information for a model.

Let's evaluate the model at $(0,0,0)$; we know that the real Lorenz system has a fixed point at the origin, and so the next state should be the same, $(0,0,0)$.

```
>> linpredict(pre,post,k,[0 0 0]) => ans: 0.0608 0.2234 4.9179
```

So, the model isn't exactly right. But perhaps it captures the essential dynamics of Lorenz even if the details aren't exact. One way to find out is to continue the dynamics starting at the new point we just calculated:

```
>> linpredict(pre,post,k,[0.0608 0.2234 4.9179]) => ans: 0.3559 0.8088 7.4691
```

We could continue on in this way indefinitely, generating a trajectory of the model dynamics. This is, admittedly, quite tedious. To save typing, the function `freerun` does the job for us:

```
>> traj = freerun(pre,post,k,[0 0 0],1000);
```

The last argument says to run the model for 1000 steps. We can look at the trajectory in the usual way:

```
>> plot3(traj(:,1), traj(:,2), traj(:,3))
```

You might imagine using free running of the model in the following way: construct the model and free run it from the first point in the data. If the model is right, then the model trajectory should be close to the data's trajectory. However, one serious shortcoming of this

approach is that it uses the same data for constructing the model and for evaluating it. This is called “in-sample” testing and tends to overestimate the quality of the model.

One way to evaluate the similarity of the model dynamics to the real dynamics is to divide your data into two sets, a training set and a test set. This is “out-of-sample” testing. Use the training set to create your training pre- and post-images. Then free run the model from the first point in your test data. Ideally, the model should follow the same trajectory as the test data. By comparing the two trajectories, you can get some idea of how good the model is.

Note that “comparing the two trajectories” and “get some idea” are rather vague. One problem for free-running is that if the real system is chaotic, even if the model is almost exactly right the chaotic sensitive dependence on initial conditions will tend to pull the model data away from the real data. For chaotic data, comparing the two trajectories means something like comparing the shapes and locations of the two attractors, and not expecting the model predictions to stay close to reality for long prediction times.

Even for chaotic data, the predictions may be good for short forecasts. This suggests another strategy, make many short forecasts from the points in your testing set. That is, make a short forecast starting from the first point in the testing set and compare it to the real value. Then do this again starting from the second point in the testing set, and then the third, and so on.

Leave-one-out cross-validation is a way to use all of our data for training, without running into the problems of in-sample testing. The idea is to make many models. For each model, we leave out one point. We use that point to test the model. As a result, we can use all of our data for training even while reserving all of the data for testing!

The function `L00prediction` runs `linpredict` in a way that implements leave-one-out (LOO) cross validation. To illustrate, let’s pick up on the Lorenz example:

```
>> [preds,actual] = L00prediction(pre,post,50);
```

The returned variable `preds` contains the predictions from each of the models. The variable `actual` contains the corresponding actual value. Insofar as they are similar, the model is good.

Let’s plot the first column of `preds` versus the first column of `actual`: if they are similar, the graph should lie on the diagonal line.

```
>> plot(preds(:,1), actual(:,1), '.');
```

The residuals are the difference between the predictions and the actual values

```
>> resids = preds(:,1) - actual(:,1);
```

The smaller the residuals, the better the fit of the model. (We take the first column in the above to translate back from the the embedding to a scalar signal.) A standard way to measure the quality of the model is to take the variance of the residuals: `var(resids)`. Using this, you can calculate an r^2 statistic for the model by taking

```
>> rsq = 1 - var(resids)/var(actual(:,1))
```

You can also do the standard sorts of regression residual diagnostic plots

```
>> plot(resids)
```

```
>> plot(resids, actual(:,1), 'x')
```

Any structure in these plots suggests that the model is not capturing all of the structure in the data, and that you should try a smaller k . (However, a smaller k may possibly lead to larger residuals, since fewer data points are being averaged together to produce each prediction.)

Although the measurement of the size of residuals in terms of their variance is almost universal in the context of linear regression, we are performing a nonlinear regression and may not have a good reason to think that the residuals are normally distributed. Nonlinear models sometimes produce very good estimates for most points and terrible estimates for some points (such as the ones at the edge of the cloud of points). Some estimates of the size of the residuals that is robust to outliers are the median square value of the residual, and the mean of the logarithm of the absolute value of the residual:

```
>> median( resid.^2)
```

```
>> mean( log( abs( resid ) ) )
```

Although I have been using the word “prediction,” for purposes such as evaluating the determinism of a model it is not clear that one wants to do predictions. For example, it might be preferable, acknowledging the role of sensitive dependence on initial conditions in chaotic systems, to make “post-dictions”: predict the past. We might even prefer to do something even funnier, “dura-diction”: use the past and the future to predict the present. There is nothing mathematically odd about this.

To implement pre- or dura-diction, we can use `lagembed` to create a matrix that is the series of columns of the signal delayed by different amounts. Pull off one of these columns to be the “post-image” and use the rest of them as the pre-image (making sure to delete the post-image column).

```
>> state = lagembed(x,4,3);
```

```
>> dura = state(:,3);
```

```
>> pre = state(:, [1 2 4]);
```

```
>> [preds,actual] = L00prediction(pre,dura,50);
```

(Note that `linpredict`, and by extension `L00prediction`, can use a “post-image” that has a different number of columns than the pre-image.)

Questions

- Free-run the Lorenz data with different values of k . Do the results depend strongly on k ? When k is very large (a large fraction of the number of data points in the pre-image set), the global dynamics are approximately linear because the “locale” is practically the whole set. How does this global linearity appear in the trajectories generated by the model?
- Create a free-running model of the Baltimore measles data. Try an embedding dimen-

sion of $p = 3$, a lag of $\tau = 4$ and $k = 30$. Run the model for 100 steps starting at `[1 1 1]`. Do the results resemble the original data? (Notice the period of the model as opposed to the original data.) What happens if you run the model further forward in time?

- Use leave-one-out cross validation to assess the modelability of the Lorenz or Rossler data. See if you can use the residuals to help you in choosing an optimal k .
- Assess the modelability of one of the real-world data sets. Note that for data that are very rapidly sampled, prediction over the short term may be very easy just because the data don't change very much over the long term. You can make long-term predictions by changing the relationship between the `pre` and `post` vectors. For example:

```
>> pre = state([1:500],:);
```

```
>> post = state(100+[1:500],:);
```

will build a model that makes predictions 100 time steps ahead. (Such a model isn't suitable for free running, but it may be quite suitable for assessing determinism.)

- One idea for detecting nonlinear dynamics in a model is to try modeling for various k , from large to small. If the predictions are better at small k , that is evidence for nonlinearity. If the predictions are better for k very large — similar to the total number of data points — that is evidence against nonlinearity.

5 Dimensions (DK)

This lab is about the correlation integral and correlation dimension. Calculating the correlation integral is straightforward once the embedding parameters have been chosen. The correlation dimension is derived from the correlation integral, but there are a number of ways of doing this; the theoretically correct method is impossible in practice. Interpretation of the correlation integral is therefore somewhat difficult.

In this lab we'll look at the correlation dimension for a variety of signals: chaotic, random, and deterministic but with a random component.

Tools

The main program is `c2`, which computes the correlation dimension of a time series. The program `lag` embeds the time series. The mandatory arguments are:

- the time series;
- the embedding dimension;
- the embedding lag;

Some additional arguments are optional:

- the decimation factor. The `c2` program is very slow because it is written in an interpreted computer language. In order to speed things up, the program is arranged to allow you to calculate a sample of the correlation integral. Rather than using every reference point in the time series, a sample of reference points is used. By default the decimation factor is 1, but increasing it will correspondingly decrease the computation time (at the cost of a less precise correlation integral).

USE A LARGE DECIMATION FACTOR at first, decreasing it when you find the computation time is satisfactory. What's large? If your time series is length N , then the computational time goes as $\frac{N^2}{\text{decim}}$. You should set `decim` so that this number is no larger than, say, 100,000. That is, for a time series of length 1000, set `decim` to be at least 10 at first. If you get results that you want to refine, you can lower `decim`. For a time series of length 1000, with `decim` set to 10, you can expect the correlation integral calculation to take some tens of seconds. For a time series of length 10000, `decim` should be set to something like 10000, decreasing it as you decide to invest in a more precise calculation.

- the Theiler correction factor. This factor is intended to reduce the influence of linear correlations on computations of short time series. It's default value is reasonable, so you don't need to worry about it except when you want to see what happens without it.
- the embedding dimensions to report. Again, this is worth worrying about only if computation time is an issue.

The program returns two variables:

r a vector of length scales at which the correlation integral has been calculated.

c the correlation integral. Each column is the integral at one of the embedding subspaces. That is, the first column is for an embedding dimension of 1, the second column is for an embedding dimension of 2, and so on. (If the `reportdims` variable is set, then the columns are for the corresponding dimension in `reportdims`.)

It's really much easier to use than the above suggests. Let's try it:

```
>> load rossler.dat
>> x = rossler(:,1); % the x component
>> [r,c] = c2(x,3,3,10); % decimation 10
>> plot(r,c); legend('1', '2', '3')
```

The `legend` command helps to label the curves: there's one for embedding dimension 1, one for dimension 2, and so on. (Use the mouse to drag the legend box to another place if it's in the way.)

Another program, `c2embed` is exactly the same as `c2` but it takes a matrix instead of a time series. This is useful if you already have an embedded cloud of points, perhaps from principal components of an embedding or some other source.

If we plot the correlation integral on log-log axes, then the slope is the correlation dimension.

```
>> plot(log(r), log(c)); legend('1','2','3');
```

You can see that the slope is not the same everywhere; this is the source of some difficulty with the correlation.

We need to pick part of the curve to calculate the slope. One easy way to do this is to specify the upper and lower bounds on the vertical axis since often the curves appear quite straight over the same vertical range. The program `d2byC` will do this:

```
>> dims = d2byC(log(r), log(c), 6, 8)
ans:      ans:  0.9924
          1.7963
          1.8626
```

To illustrate a somewhat longer calculation, we'll calculate the correlation integral for embedding dimensions up to 10:

```
>> [r,c] = c2(x(1:1000),10,3,10);
>> plot(log(r), log(c)); legend(num2str([1:10]'));
>> xlim([-5 5]) % set the x-axis to a better scale
>> dims = d2byC(log(r), log(c), 5,7)
```

```

ans:      ans =
          0.9569
          1.7803
          1.9395
          1.9788
          1.9347
          2.0032
          2.0156
          2.0489
          2.0570
          2.0559

```

Researchers often plot the computed dimension versus the embedding dimension

```
>> plot(1:10, dims, 'x-')
```

and look for “saturation,” a leveling-off of the curve. Here the curve levels off at a dimension of slightly more than 2.

Questions

- Compute the correlation integral of a very simple set of points embedded in 1 dimension:


```
>> x = [1;2;3;4;5;6;7;8;9;10];
```

 (Make sure that `x` is a column vector, that is, a matrix with only one column.) Now compute the correlation dimension of these points (in their 1-dimensional embedding):


```
>> [r,c] = c2embed(x);
>> plot(r,c)
```

 Explain the result in detail. (Note, here we’ve plotted `r` and `c` on linear axes. In the remainder of the lab, we’ll use log-log axes.)
- Compute the correlation dimension of gaussian white noise:


```
>> noise = drand(1000,1);
>> nn = lagembed(noise, 10, 1);
```

 How does the correlation dimension depend on the embedding dimension?
- Compute the correlation dimension of a periodic sine curve:


```
>> s = sin([1:1000]/8)';
```

 How does this depend on the embedding dimension?
- Take a nonlinear measurement transformation of the sine curve, for instance


```
>> x = s.^3;
```

 or


```
>> x = sin( 3*s );
```

 Do the dimensions differ from those of the original sine curve?

- Add a small amount of noise to the sine curve
 $\gg \mathbf{x} = \mathbf{s} + 0.1 * \text{randn}(\text{size}(\mathbf{s}));$
 How does this change the correlation integral?
- Take a length of the Rossler data generated with `makerossler`. Arrange to sample the signal so that there are roughly 25 points per cycle and about 1000 points altogether. What is the dimension? Now generate another such signal from a different initial condition. Add it to the first signal. What is the dimension of the sum?

6 Linear Dynamics and Power spectra (DK)

Objectives

This lab is about the very special functions, sines and cosines, in the analysis of time series data. We'll see why these functions are so important, and how to analyze linear data in terms of them using the fourier transform, power spectrum, and transfer function.

A dynamical system often involves a state whose future value is a function of the past and present:

$$x_{t+1} = f(x_t, x_{t-1}, \dots, x_{t-(p-1)})$$

A linear dynamical system is one where the function $f(\cdot)$ is linear, that is

$$x_{t+1} = a_1x_t + a_2x_{t-1} + \dots + a_px_{t-(p-1)}$$

Such systems often are good models locally; that's why locally linear modeling, as implemented in `linpredict` works so well.

Globally, however, such models have a problem: either they blow up to infinity or they have a stable fixed point at the origin, depending on the stability of the system. Neither of these alternatives is a good model of complex biological or physiological data.⁹

Where globally linear models come in to their own is when we include an *input*:

$$x_{t+1} = a_1x_t + a_2x_{t-1} + \dots + a_px_{t-(p-1)} + b_1y_t + b_2y_{t-1} + \dots + b_my_{t-(m-1)}$$

We needn't worry about where this input is coming from — people who use such linear models often assume that it is some random disturbance, typically “white noise.” If the system is unstable, the input doesn't help; the system still heads off to infinity. But if the system is stable, the input keeps the system away from the fixed point and keeps it doing interesting, potentially complex-looking things.

To keep things simple, let's consider the simplest, dynamically “interesting” model

$$x_{t+1} = a_1x_t + a_2x_{t-1} + y_t$$

The reason this is interesting is that with the two parameters, a_1 and a_2 , the model is capable of producing oscillations. For example, for $a_1 = 1$, $a_2 = -0.8$, the system is quite like a spring-mass system.

To see this, we can use the program `arma` that takes the various parameters along with an input y_t and produces the corresponding output x_t . (Engineers would typically write the input as x and the output as y , a convention that we violate here because dynamicists typically write the state variable as x .) Let's create an input that is an impulse: a sharp blow.

```
>> y = zeros(100,1); y(1) = 1;
```

⁹Although the stable fixed point might be a good model of dead systems.

```
>> x = arma([1, -.8], 1, y);
>> subplot(2,1,1); plot(y,'o'); subplot(2,1,2); plot(x);
```

This should be pretty much what you expect from a spring: hit it and it vibrates for a while. The result is a sine wave of amplitude that decreases in time.

More interesting is when the input y is white noise:

```
>> y = randn(100,1);
```

```
>> x = arma([1, -.8], 1, y);
>> subplot(2,1,1); plot(y,'o'); subplot(2,1,2); plot(x);
```

Now the signal x looks like something really quite complicated. It may even look chaotic to you although of course it is not since it comes from a linear system. You can still see the ups and downs, but now they are neither regular nor decreasing in amplitude.

Something surprising happens when we try an input that is a sine wave:

```
>> y = sin(0.2*[1:100]');
```

The 0.2 sets the frequency of the sine wave. You can change it to something else if you like.

```
>> x = arma([1, -.8], 1, y);
>> subplot(2,1,1); plot(y,'o'); subplot(2,1,2); plot(x);
```

Notice that the output is a sine wave that has the same frequency as the input. (Well, not exactly: there's a little jiggle in the beginning that is called a "transient." We wouldn't see this if we skipped the first part of the signal.) The amplitude of the output is a little bigger than that of the input, and — if you are very observant — you may notice that the phase of the output is different from the input.

This is one of the amazing things about linear systems: a sine wave input produces a sine wave output (ignoring the transient). The output has the same frequency, although the amplitude and phase may be different. Try it again with a higher frequency

```
>> y = sin(1.2*[1:100]');
```

Here it's easy to see that the phase has been shifted: the output is almost the negative of the input.

Another amazing thing about linear systems: if we give them an input composed of two parts, say

```
>> y = sin(1.2*[1:100]') + sin(0.2*[1:100]');
```

the output will be the sum of the two outputs if we had given the inputs separately. That is, if input y gives output x , and input η gives output ξ , then input $y + \eta$ gives output $x + \xi$. This is the principal of "superposition."

One last amazing fact: any input can be written as the sum of sine waves. This fact was discovered by Fourier in the early 1800s.

Think of the consequences. If any input is a sum of sine waves, and if each of those sine waves produces — in isolation — a sine-wave output, then the output is also a sum of sine waves, each one altered in amplitude and phase by the system. Moreover, we can characterize the system by the way in which it changes the amplitude and phase of each of the component sine-wave inputs. All we need is a way to decompose the input into a sum of sine waves, and the output into a presumably different sum of the same sine waves. Then, we can compare the amplitudes and phases of the corresponding pairs of sine waves and we

will have characterized the system. This characterization is called the transfer function.

Decomposing a Signal into Sine Waves

We're going to see how to decompose a signal into sine waves. Just to make sure that you will not be misled, let me tell you now that the universally used way to do this is with the Fourier Transform, generally implemented as the Fast Fourier Transform (FFT). If you know about the FFT, then feel free to skip this section. Otherwise, you'll take a short tour of sine waves and end up knowing something about the FFT.

The program `ms` generates sine-wave signals in a way that's easy to use:

```
>> s = ms(100,4);
makes a sine wave of length 100 which has 4 cycles.
>> plot(s, 'r');
```

Note that the first point and the last point are not exactly the same, but that if you were to add one more point to the end of the signal it would have the same value (vertical coordinate in the graph) as the first point in the signal.

The number of cycles, 4 in the above example, is only one way of describing how fast the sine wave cycles. A more conventional way is the "frequency," the number of cycles per second, Hertz (Hz). In the above example, if we assumed that the samples are taken 100 times per second, then the frequency would be 4 Hz. But if the samples were taken once per second, the frequency would be 4/100 Hz, or .05 Hz. This is always a bit confusing, so be careful!

Cosine waves are just the same as sine waves, but shifted a bit in phase. We can use a third argument to `ms` to indicate this:

```
>> c = ms(100,4,pi/2);
```

`pi` is a variable in MATLAB that has the numerical value of π . Note that the phase is measured in radians, not degrees.

Try a few different phases, comparing them, to make sure that you understand what the phase argument does in `ms`; it shifts the signal to the left. A phase of 2π (`2*pi` in MATLAB notation) would shift the sine wave by one complete cycle and so you wouldn't notice that it is shifted at all.

Here's another amazing fact of sine waves: if you take a sine wave of some frequency and any phase, it will be the sum of a sine and a cosine of the same frequency. Try the following: make a sine wave of some frequency and phase

```
>> sig = ms(100, 5.4, .9323423);
```

Now make the sine and cosine with the same frequency:

```
>> s = ms(100, 5.4);
>> c = ms(100,5.4, pi/2);
```

The amazing-fact claim is that there is some numbers α and β such that $\mathbf{sig} = \alpha\mathbf{s} + \beta\mathbf{c}$. How to find those numbers α and β ? By standard linear regression, of course. MATLAB makes it very easy to do linear regression: just package `s` and `c` into a matrix and divide this matrix into `sig` using the "under operator" `\`:

```
>> a = [s c]\sig ⇒ ans:  0.5960
                        0.8030
```

This gives the two numbers we need.

To check that the fit is right, we can construct the signal from `s` and `c`:

```
>> fitted = 0.5960*s + 0.8030*c;
>> plot(fitted); or
>> plot(fitted,sig);
```

Rather than do the reconstruction “by hand”, it’s easier and more accurate to do it automatically using matrix multiplication:

```
>> fitted = [s c]*a;
```

(If you are new to matrix multiplication, please note that it is not commutative. That is `[s c]*a` is quite different from `a*[s c]`.)

Remember, our eventual goal is to decompose any signal into sines and cosines. We know now that a signal that consists of a single frequency will, in general, require one sine and one cosine to fit it.

Now let’s move on to more complicated signals.

```
>> x = arma([1, -.8], 1, randn(100,1));
```

I don’t know exactly what your `x` signal will look like, because it’s random. Mine looks roughly like a sine wave whose amplitude is changing and that has about 20 cycles. In order to fit this signal to pure sines and cosines, we’re going to need many of them. `ms` is written to make it easy to generate multiple signals:

```
>> setOfSines = [ms(100, 14:22), ms(100, 14:22, pi/2)];
```

The `setOfSines` consists of several sine and cosine waves with frequencies ranging from 14 to 22 cycles. Let’s fit this to the time series:

```
>> a = setOfSines \x
```

```
ans:      a =
        0.0711
       -0.2247
        0.1859
       -0.0959
```

and so on

Your answers will differ from mine because your random signal differs from mine.

Now, let’s reconstruct the signal using the fitted sines and cosines:

```
>> fitted = setOfSines*a;
>> plot(fitted); hold on; plot(x, 'o'), hold off
```

You probably got a crumby result. The reason is that we forgot a really important sine wave: the cosine of frequency zero. This cosine is just a constant: it’s used to fit the mean of the signal. So, we try again, adding in the special cosine of frequency 0.

```
>> setOfSines = [ms(100, 14:22), ms(100, [0 14:22], pi/2)];
>> a = setOfSines \x;
>> fitted = setOfSines*a;
```

Better, but still not great. If we add more sines and cosines, we will get a better answer.

The answer will be perfect if we use all of the sines and cosines. But — surprise again! — “all of the sines and cosines” is not that many: we need sines of frequencies 1 to 49 and cosines of frequencies 0 to 50.

```
>> setOfSines = [ms(100, 1:49), ms(100, [0:50], pi/2)];
>> a = setOfSines \x;
>> fitted = setOfSines*a;
```

Now it's a perfect fit (except for the small error due to numerical round-off).

You've just done a Fourier Transform: turning the signal x into the number a is a Fourier Transform. In general, for a signal of length N , where N is even, the sines will have the frequencies 1 to $\frac{N}{2} - 1$ and the cosines will have the frequencies 0 to $N/2$.

You have also done an “Inverse Fourier Transform.” The Inverse Fourier Transform is the operation that takes you from the amplitudes of the sines and cosines back to a signal. When you did

```
>> fitted = setOfSines*a;
```

you were inverting the a to produce $fitted$. (It might seem funny that an inversion is involving multiplication, but then the forward transform involved matrix division using the \backslash operator.)

You might ask, “Why use the specific frequencies 1,2,3, and so on? What about 4.3 or other non-integer frequencies?” The answer is that the integer frequencies (that is, an integer number of cycles over the length of the signal) are special in that they are *orthogonal* to one another; they don't interfere with one another.

Mathematically, a Fourier Transform is generally not written in terms of sine and cosines explicitly, but in terms of complex exponentials. This technicality need not concern us, except to note that the FFT generally is arranged to give the amplitude and phase of each frequency component, rather than the amplitudes of the sines and cosines separately. Of course we understand that the sum of a sine and a cosine of the same frequency gives another wave of that frequency but shifted in phase.

Just for fun, let's look at the FFT of the signal x :

```
>> fft(x)
ans:      ans = 65.8145
          0.2644 + 3.3290i
          0.3524 - 3.0200i
          0.0734 - 4.3711i
          and so on
```

(Your answers will be different, because your signal x is different from mine). One thing to notice is that the first number is the mean of the signal multiplied by it's length.

It's likely that you are not happy about having complex numbers come out of the `fft`. No problem. Let's convert it to amplitude and phase of the sine wave at each frequency.¹⁰

```
>> amp = abs(fft(x));
>> phase = angle(fft(x));
```

¹⁰Why do I say “the” sine wave? Because each complex number pair in the `fft` really corresponds to a sine wave and a cosine wave of the same frequency. But, as we saw, this pair can be represented as a single sine wave shifted in phase.

Now you can plot out the amplitude

```
>> plot(amp)
```

You will likely see a sharp peak near zero and a broad peak near 20 (if you have used the ARMA parameters I gave here). Plotting out the phase

```
>> plot(phase);
```

likely produces something that looks completely random.

Certain things to note about the amplitude and phase of the fft of a signal:

- It is symmetric in amplitude. The left side of the plot matches the right side in a mirror image way.
- It is anti-symmetric in phase. The left side of the plot matches the right side as a mirror image, but turned upside down.
- The phase is always between 0 and 2π .
- The leftmost point in the amplitude plot is related to the mean of the signal: it's the signal length times the mean.

The Transfer Function

Remember that for a linear system, if we put a sine wave in to the system, we will get a sine wave out (aside from a transient). One extremely powerful way to study a system is to see how each type of sine wave in the input gets altered in the output. One type of representation of this is called the “transfer function:” it describes how each frequency of sine wave is altered in amplitude and phase. Computing a transfer function is quite simple using the fft:

- measure both the input and the output of the system
- take the fft of the input and the fft of the output
- divide the output fft by the input fft
- plot the result by looking at amplitude and phase separately

In terms of MATLAB commands:

```
>> input = randn(1000,1); % measure an input
```

```
>> output = arma([1, -.8], 1, input); % measure the output
```

Of course, experimentally, you would measure the data from your system and then read it into the computer as a file.

```
>> xfer = fft(output)./fft(input); % the transfer function
```

```
>> plot(abs(xfer)); % the amplitude
```

```
>> plot(angle(xfer)); % the phase
```

You should see a very strong structure in the two plots; they should look like “mathematical” functions with a bit of fuzz. The fuzz is actually due to round-off error in the computer.

Interpreting the transfer function is not completely easy. The general rule is that each peak in the function corresponds to one resonant mode of the system. More detailed things can be computed.

You should also know that the FFT is not the only way to compute a transfer function. You can also use linear modeling to compute what is called a “parametric estimate” of the transfer function. The FFT method is very quick and easy, however.

Another method for analyzing the linear relationship between input and output is the “coherence function.” We’ll be talking about this on Wednesday.

The Power Spectrum

When you can measure both an input and an output, you should do so. One of the great failings of “chaos theory” has been to suppress the idea of an input; the idea has been that the system generates complex behavior without any sort of input. Of course, linear systems cannot behave this way, since a linear system will always either head off to infinity (which is physically impossible) or decay to zero unless there is some input.

In the cases where you don’t measure an input, there is still something you can do. The idea is to calculate the transfer function as if the input were white noise. That is, we assume that the input is white noise (even though it might not be). Since — as we’ll see below — the fft of white noise has a random phase and statistically constant amplitude, we replace the `fft(input)` part of the calculation with the number 1 — that is, a constant. However, this means that the phase part of the transfer function is complete nonsense, since we can make no valid statistical assumptions about the input phase at any frequency. So, we ignore the phase and consider only the amplitude.

The convention is to plot the amplitude squared. This is called the “power spectrum” of the output signal. But, since we didn’t measure an input, instead of saying “output signal” we just say “signal.”

The power spectrum is easy to calculate from the fft:

```
>> powerspec = abs(fft(output)).^2;
```

Things get a bit more complicated than this for statistical reasons: one gets a better estimate of the power spectrum by averaging in certain ways, deleting trends from data, windowing and other somewhat magical-sounding things.¹¹

Much of the technology for estimating power spectra has been encapsulated in a program called `powerspec`.

```
>> [p,f] = powerspec(output, 1, 100);
```

¹¹As scientists, we don’t believe in magic. Indeed there is no magic in the power spectrum, just badly explained things that therefore sound like magic.

The first argument is the signal. The second is the frequency at which the data were sampled; in this case 1 Hz. This information is used only to help you make a nice plot. The third argument is the length of segments to use in averaging together a nice-looking power spectrum. If you make this number very small, you get a very smooth power spectrum. If you make this number very large (similar to the size of the signal) you get the kind of rough power spectrum that the straight `fft` provides. Choosing a correct value is about as difficult as selecting an embedding lag! (It's faster to compute things if this is a nice number: powers of 2 are particularly nice. But for short signals you will not notice the speed differences.)

To plot the power spectrum

```
>> plot(f,p)
```

The two arguments to `plot` cause the x-axis to be labeled properly in terms of frequency.

Other Magical Properties

The inverse fourier transform of the power spectrum is the autocorrelation function. The two contain exactly the same information.

If you knock the linear system with a sharp pulse, an “impulse,” then the resulting output is called the “impulse response.” The inverse fourier transform of the impulse response is the transfer function.

Questions

1. Generate some white noise using the `randn` function, e.g.

```
>> noise = randn(1000,1);
```

Look at the amplitude and phase of the fourier transform of this. The phase should look random, the amplitude is statistically constant but will jump up and down randomly for any given signal. Use power spectrum (with a very short segment length) to see what the average amplitude is; it should be constant versus frequency.
2. Look at the amplitude of the Baltimore measles data. Convince yourself that it isn't white noise.
3. Look at the power spectrum of some Lorenz-system data.
4. Compute the power spectrum of a pure sine wave. You should get a single narrow peak. Now take the square of a sine wave. How has the frequency of the peak changed. Now take the cube. You should be seeing the “harmonics” of the sine wave's frequency. In general, a nonlinear measurement will cause an increase in the number of peaks in the power spectrum. This is extremely hard to interpret since it looks just like an increase in the number of modes of the system.
5. There are several files — `mg15`, `mg30`, `mg100` — for the Mackey-Glass system at different parameter values, both chaotic and non-chaotic. See if you can distinguish between the various signals based on their power spectra.

6. Play with changing the parameter used in the `arma` program and seeing how the resulting power spectra or transfer functions differ.

7 Surrogate Data (DK)

Objectives

This lab introduces surrogate data: how to generate it; how to use it; and how to interpret it. Since generating surrogate data is so simple, there will be a strong emphasis on the interpretation and on problems and artifacts. But don't be misled; whatever are the problems with the surrogate data technique, not using it is much worse! Surrogate data should be a standard part of your nonlinear time series analysis toolbox.

Tools

Three surrogate-generating programs are provided here:

Phase-randomized surrogates `fftsurr` produces a realization of phase-randomized surrogate data. Each time the program is called, it produces a new realization of surrogate data (controlled by the computer random number generator seed).

```
>> data = load in some data here
>> surr1 = fftsurr(data);
>> surr2 = fftsurr(data);
```

Note that `surr1` and `surr2` are different, but their power spectra (as estimated using the amplitude of the fft) are the same as the original time series:

```
>> psd1 = abs(fft(data));
>> psd2 = abs(fft(surr1));
```

Plot out one power spectrum versus the other (`plot(psd1,psd2,','')`;) (or `plot(log(psd1),log(psd2),','')`) and you will get a straight line: the two spectra are identical.

Amplitude-adjusted surrogates `ampsurr` produces “amplitude-adjusted surrogates.” In phase-randomized surrogates, the power spectrum of the test data and the surrogates are identical. However, the actual values in the signal are not the same. For example, try

```
>> plot( sort(data), sort(surr1), ','')
```

or

```
>> subplot(2,1,1); hist(data); subplot(2,1,2); hist(surr1);
```

with phase-randomized surrogates.

In amplitude-adjusted surrogates,

```
>> asurr1 = ampsurr(data);
```

the amplitudes of the surrogate and data are identical; the surrogate is just a shuffled version of the points in the data. The shuffling has been cleverly done, however, so that the power spectrum of the signal and the surrogate are *almost* the same. Try comparing the power spectra of an amplitude-adjusted surrogate.

Polished surrogates (`polsurr`) produces surrogates that, like amplitude-adjusted surrogates, have the same amplitude distribution as the original data. But the shuffling has

been more carefully done than in amplitude-adjusted data so the power spectrum of the surrogate is even closer to that of the original data.

There is also a program for shuffled surrogates, `surr = shuffledsurr(data);`. This is just like amplitude-adjusted surrogates except the power spectrum is white. The only time you would want to use such surrogates is when all you care about is the amplitude of the data, for example when you want to use shuffled residuals to drive a linear model. *Do not use shuffled surrogates for general purposes.*

To use surrogate data as part of a hypothesis test, you need a test statistic. Some examples of test statistics that are widely used are the correlation dimension and nonlinear predictability. The basic idea is to calculate the test statistic for the original data (called the “test data”) and then calculate the test statistic for a bunch of realizations of surrogate data. You compare the single value for the test data to the bunch of values for the surrogates, perhaps ultimately computing a p-value.

Two important points to keep in mind when considering a test statistic are:

1. It should generally be a single number: a scalar, not a vector. There are occasions when a vector test statistic makes sense, but they are more complicated than the ones we will consider here.
2. Exactly the same parameters and settings should be used for computing the test statistic for the test data as for the surrogate data. For example, it is *illegitimate* to pick some parameters for the test data (for instance, the scaling region for the correlation dimension calculation) and then apply those parameters to the surrogates.

If the above two restrictions have been honored, then it should be possible to implement the test statistic as a computer function that takes exactly one argument — a time series — and returns a single number.

Here are some test statistics that are implemented as programs: you can construct your own with the basic template

```
function result = myteststat(data)
result = complicatedcalculation(data, parameter1,parameter2, and so on);
```

This code would go in a file named `myteststat.m`. The complicated calculation might be nonlinear prediction, or whatever; you can put in whatever program you like to do the actual calculations. It is a matter of personal intellectual integrity that the parameters haven’t been established by looking at the data you are testing.

Here are two test statistics that are somewhat abstract, but have the great virtue of being extremely fast to calculate:

timerev(data,timescale) computes a statistic relating to the time-reversal asymmetry of the data. The parameter `timescale` is an integer describing a time scale of interest; set it to be perhaps one-quarter of the approximate period of oscillations. In order to avoid setting the parameter by looking at the data, you might want to generate a surrogate and base your estimate of the period from the surrogate. Then throw away the surrogate.

crinkle(data) computes a 4th moment of the time series. James Theiler invented this statistic to demonstrate some weaknesses of surrogate data.

To make it easy to do a hypothesis test, we provide a function **surrogatetest** that takes the data and a function implementing a test statistic, and runs many surrogates. **surrogatetest** returns the t-score and a non-parametric p-value (for a left-sided test, so subtract the result from 1 for a right-sided test). It also gives back the test statistic for the test data and the surrogates. An example (using the Baltimore measles data in **balt.dat**):

```
>> [t,r] = surrogatetest(balt, 'crinkle(x)', 'fftsurr') ⇒ ans: t = 801
```

r = 0.9500

By default, 19 surrogates are used. The result is a strong indication of nonlinearity: the t-statistic is huge (a value of 1.73 would be statistically significant at the 95% level for a one-tailed test); the non-parametric p-value is 0.05 for the right-tailed test. If we ran the test again with more surrogates, the non-parametric p-value would be even less — it's 1% for 99 surrogates. Wow! When have you ever seen a result that strong with so little effort!

An important note about programming. Note that in the above, the test statistic was **crinkle(x)**. The variable should always be **x** regardless of the name of the data set. **x** is just a dummy variable. You can also include parameters, but they must be numbers and not variables. For example: **quickde(x,2,3)** but NOT **quickde(x,dim,lag)**.

Questions

- Re-run the analysis of the Baltimore data with amplitude-adjusted surrogates and with polished surrogates. How and why do the results differ?
- Write a program for either nonlinear prediction or for the correlation dimension and apply it to the lorenz data. Remember, your program should take a data set as an input and provide a scalar number as an output. For the dimension calculation, you might want to use the rule-of-five method. For nonlinear prediction, perhaps take the median absolute value of the residual.
- Take a segment of Lorenz or Rossler data and try nonlinear prediction using both short and long prediction horizons. Now try the prediction on some surrogates generated from the data. How do the results differ for different prediction horizons?
- Let's make some data from a linear process with gaussian white noise inputs:


```
>> y = randn(500,1);
>> x = arma([1.95 -.96], 1, y);
```

 Use the ARMA parameters given. You'll see that the result is a smoothly varying oscillation.

This represents a signal for which surrogate data should give a negative result: the data satisfy the null hypothesis of surrogate data.

Test the linear data using surrogate data. Since this is a hypothesis test, you should get a result that causes you to reject the null hypothesis every once in a while; for a

significance level of 5% you should anticipate a false rejection of the null 5% of the time. Repeat the test many times, using new realization of the noisy input y to generate new linear time series x . How often do you falsely reject the null?

- Try the same thing, but using $x.^3$ or $x.^2$ instead of x .

- Let's try a slightly different linear system:

```
>> x = arma([1.5 -.7], 1, y);
```

This system has a fast-decaying impulse response.

Rather than using gaussian white noise as the input, lets use some non-gaussian spikes:

```
>> y = zeros(500,1); y(10) = 7; y(287) = -3; y(403) = -5;
```

```
>> x = arma([1.5 -.7], 1, y);
```

Test this system for “nonlinearity” using the crinkle test statistic. What violates the null hypothesis in this case?

- Generate two ARMA signals with quite different power spectra (such as the two used above) and concatenate them to make a linear, nonstationary system. Test the nonstationary signal using surrogate data.