In [1]:

```
# To support both python 2 and python 3
from __future__ import division, print_function, unicode_literals
%matplotlib inline
import os
import numpy as np
import matplotlib.pyplot as plt
```

Linear Algebra for regression

Let $y(x) = w_0 + w_1\phi_1(x) + w_2\phi_2(x)$ be a model for mapping given input data x to given target data y:

$$\begin{aligned} \phi_i : X &\rightarrow Y, \\ X \ni x &\mapsto y \in Y \\ y(x) &= \phi_i(x),\ i = 1, 2 \end{aligned}$$

where ϕ_1 and ϕ_2 are some chosen functions. For instance, in sound analysis, ϕ_1, ϕ_2 could be sines and cosines, and the representation of an observed signal $y(x)$ would be a Fourier series, where ϕ_i are frequency components.

In the simplest examples, we took $\phi_1(x) = x$ (and $w_2 = 0$) for linear fits, $\phi_i(x) = x^i$ for $i = 1, 2$ for fitting a parabola to observations $y(x)$. When we invoked 'polyfit' (with deg=2) from numpy using the training pairs (x_k, y_k) for $k = 1, \ldots, N$, we obtained w_2, w_1 and w_0 as the estimates that minimised $\sum_k r_k^2$ where $r_k = \hat{y(x_k)} - y(x_k)$. Note the hat on $\hat{y}$-- it signals an estimated value, not a "true" data point.

We'll describe how the estimation of weights is done in the general case, by minimising the *residuals* $r_i = \hat{y(x_i)} - y_i$. For each data point (x_i, y_i),

$$y_i = w_0 \times 1 + w_1 \times \phi_1(x_i) + \cdots + w_p \times \phi_p(x_i) = \begin{pmatrix} w_0 & w_1 & \cdots & w_p \end{pmatrix} \begin{pmatrix} 1 \\ \phi_1(x_i) \\ \vdots \\ \phi_p(x_i) \end{pmatrix}.$$

- This leads to the set of simulataneous linear equations $\mathbf{Aw} = \mathbf{y}$:

$$\overbrace{\begin{pmatrix} 1 & \phi_1(x_1) & \cdots & \phi_p(x_1) \\ 1 & \phi_1(x_2) & \cdots & \phi_p(x_2) \\ \vdots & \vdots & \vdots & \vdots \\ 1 & \phi_1(x_N) & \cdots & \phi_p(x_N) \end{pmatrix}}^{\mathbf{A}} \overbrace{\begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ \vdots \\ w_p \end{pmatrix}}^{\mathbf{w}} = \overbrace{\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix}}^{\mathbf{y}},$$

that we need to solve for $\mathbf{w} = (w_0, w_1, \ldots, w_p)^T$. The matrix $\mathbf{A}$ is called the **design matrix**.

- Let us revisit the **linear regression** method to see how to think about the mismatch between model and data.

- Recall that the least squares solution the weight vector $\mathbf{w}$ to $\mathbf{y} = \mathbf{A}\mathbf{w}$ was found by requiring the residual $\mathbf{r} = \mathbf{y} - \mathbf{A}\mathbf{w}$ to be **orthogonal** to the data matrix $\mathbf{A}$:

$$\mathbf{A}^T\mathbf{r} = 0.$$

This implies

$$\begin{aligned} \widehat{\mathbf{w}} &= (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{y} = \mathbf{A}^+\mathbf{y}, \\ \text{where } \mathbf{A}^+ &\triangleq (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T \end{aligned}$$

called the **Moore-Penrose pseudoinverse** , and the predicted output vector $\widehat{\mathbf{y}}$ for a chosen set of inputs (that are absorbed in $\mathbf{A}$) is $\widehat{\mathbf{y}} = \mathbf{A}\widehat{\mathbf{w}}$,

$$\begin{aligned} \widehat{\mathbf{y}} &= \mathbf{A}(\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{y} = \mathbf{P}_\mathbf{A}\mathbf{y} \\ \text{where } \mathbf{P}_\mathbf{A} &\triangleq \mathbf{A}(\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T. \end{aligned}$$

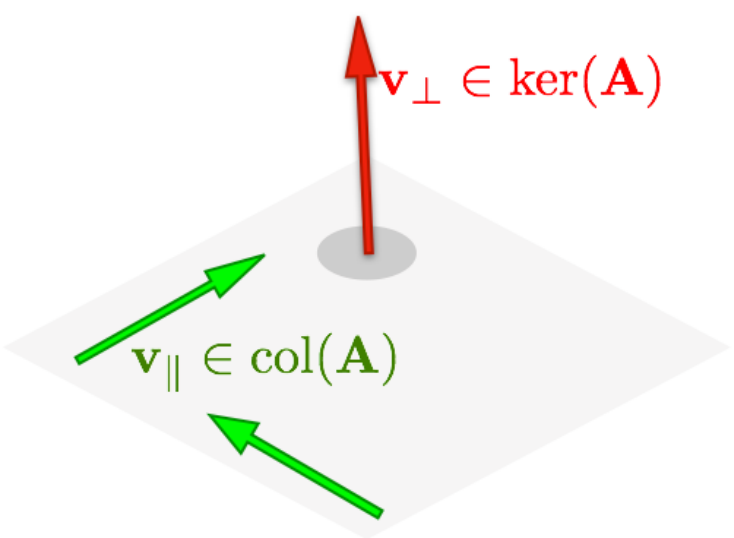
- The matrix $\mathbf{P}_\mathbf{A}$ is a **projection** matrix, which means it satisfies
  - (1) $\mathbf{P}_\mathbf{A}^2 = \mathbf{P}_\mathbf{A}$ (**idempotent**). Projecting twice gives the same result as projecting once.
  - (2) $\mathbf{P}_\mathbf{A}(\mathbf{P}_\mathbf{A} - \mathbf{I}) = 0$, which implies $\mathbf{P}_\mathbf{A}$ has eigenvalues 0, 1.
- Proof of (1):

$$\begin{aligned} \mathbf{P}_\mathbf{A}^2 &= \underbrace{\left(\mathbf{A}(\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\right)}_{\mathbf{P}_\mathbf{A}}\underbrace{\left(\mathbf{A}(\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\right)}_{\mathbf{P}_\mathbf{A}} \\ &= \mathbf{A}(\mathbf{A}^T\mathbf{A})^{-1}\underbrace{\left(\mathbf{A}^T\mathbf{A}\right)}_{\mathbf{I}}(\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T \\ &= \mathbf{P}_\mathbf{A} \end{aligned}$$

- Consequence of (2): Any vector $\mathbf{v}$ in the domain of $\mathbf{P}_\mathbf{A}$ can be written as $\mathbf{v} = \mathbf{v}_\parallel + \mathbf{v}_\perp$. These are the eigenvectors of $\mathbf{P}_\mathbf{A}$ with eigenvalues 1, 0 respectively:

$$\mathbf{P}_\mathbf{A}\mathbf{v}_\parallel = \mathbf{v}_\parallel,\ \mathbf{P}_\mathbf{A}\mathbf{v}_\perp = 0.$$

Thus $\mathbf{v}_\perp$ lies in the **nullspace** or **kernel** of $\mathbf{P}_\mathbf{A}$, $\mathbf{v}_\perp \in \ker(\mathbf{P}_\mathbf{A})$ and $\mathbf{v}_\parallel$ lies in $\text{col}(\mathbf{P}_\mathbf{A})$, the **range** of $\mathbf{P}_\mathbf{A}$.



- We can now see that the predicted output $\hat{y}$ is projection of the training target points defined by the model and the training input data points that make up $\mathbf{A}$.

- The **column space** view illustrates the role of the weights in combining features in appropriate proportions:

$$\mathbf{A}\mathbf{w} = \mathbf{y} \Leftrightarrow w_0 \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} + w_1 \begin{pmatrix} \phi_1(x_1) \\ \phi_1(x_2) \\ \vdots \\ \phi_1(x_N) \end{pmatrix} + \cdots + w_p \begin{pmatrix} \phi_p(x_1) \\ \phi_p(x_2) \\ \vdots \\ \phi_p(x_N) \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix}$$

- $N > p + 1$: more data points than features

In this case there are more equations than unknowns the system is called *overdetermined* and there is (in general) no solution.

- $N < p + 1$: fewer data points than features

The system of equations is *underdetermined* and there could be multiple solutions.

- In linear regression methods, the (N-by-(p+1)) matrix $\mathbf{A}$ is **fixed** for a given data set and a set of features ϕ_i. By changing weight vectors $\mathbf{w}$ we obtain all possible N-dimensional vectors that $\mathbf{A}$ is capable of generating. This is called the **column space** of the matrix $\mathbf{A}$. The target vector $\mathbf{y}$ is sought in this column space. The weight vector $\widehat{\mathbf{w}}$ for which the residual vector $\mathbf{r} = \mathbf{y} - \mathbf{A}\widehat{\mathbf{w}}$ is closest to 0 is the outcome of the learning algorithm.
- In neural network models, the feature functions $\phi_i(x_j, \theta_i)$ themselves can depend on additional parameters θ_i that can introduce a non-linear dependencies in the learning algorithm.
- The notation in the general linear case can be forbidding, so let us repeat the content in more concrete settings below.
- We will make the observation that the product of the design matrix $\mathbf{A}$ and the residual vector $\mathbf{r}$ is 0 in a specific example. This will be true for any residual computed from the best-fit weight vector.
- It is this leap into the abstract viewpoint that facilitates a general understanding of the structures (patterns) within data that could be ripe for representing via matrix transformations and decompositions.

Finding the best fit weight vector

Row and column views on matrix-vector multiplication

- The matrix $\mathbf{A}$ can be written as a collection of N row vectors or p column vectors:

$$\mathbf{A} = \begin{pmatrix} \uparrow & & \uparrow \\ \mathbf{a}_1 & \cdots & \mathbf{a}_N \\ \downarrow & & \downarrow \end{pmatrix} = \begin{pmatrix} \longleftarrow & \mathbf{a}_1 & \longrightarrow \\ & \vdots & \\ \longleftarrow & \mathbf{a}_p & \longrightarrow \end{pmatrix}.$$

- Consider the following 2 different perspectives on matrix-vector multiplication $\mathbf{A}\mathbf{w}$:

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1p} \\ a_{21} & a_{22} & \cdots & a_{2p} \\ \vdots & \vdots & \vdots & \vdots \\ a_{N1} & a_{N2} & \cdots & a_{Np} \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_p \end{pmatrix} = \underbrace{\begin{pmatrix} \sum_j a_{1j} w_j \\ \sum_j a_{1j} w_j \\ \vdots \\ \sum_j a_{Nj} w_j \end{pmatrix}}_{\text{row view}} = \underbrace{\sum_j w_j \begin{pmatrix} a_{1j} \\ a_{2j} \\ \vdots \\ a_{Nj} \end{pmatrix}}_{\text{column view}}.$$

- Rewriting to illustrate the column view, we have

$$\mathbf{Aw} = \begin{pmatrix} \uparrow & & \uparrow \\ \mathbf{u}_1 & \cdots & \mathbf{u}_p \\ \downarrow & & \downarrow \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_p \end{pmatrix} = w_1 \begin{pmatrix} \uparrow \\ \mathbf{u}_1 \\ \downarrow \end{pmatrix} + \cdots + w_p \begin{pmatrix} \uparrow \\ \mathbf{u}_p \\ \downarrow \end{pmatrix},$$

- while the row view is rewritten thus:

$$\mathbf{Aw} = \begin{pmatrix} \longleftarrow & \mathbf{v}_1 & \longrightarrow \\ & \vdots & \\ \longleftarrow & \mathbf{v}_N & \longrightarrow \end{pmatrix} (\mathbf{w}) = \begin{pmatrix} \mathbf{v}_1 \cdot \mathbf{w} \\ \vdots \\ \mathbf{v}_N \cdot \mathbf{w} \end{pmatrix}$$

- The **column space** of a matrix A, denoted col(A) is the subspace spanned by all linear combinations of the columns of A. This is also the range or image of the linear map:

$$\text{col}(A) = \text{im}(A) = \text{range}(A) = AW = \{\mathbf{v} \in V : \mathbf{v} = A\mathbf{w} \text{ for some } \mathbf{w} \in W\}.$$

For any choice of real numbers $(w_1, \ldots, w_p)$, the range V is the set of all vectors of the form

$$\mathbf{v} = w_1 \begin{pmatrix} \uparrow \\ \mathbf{u}_1 \\ \downarrow \end{pmatrix} + \cdots + w_p \begin{pmatrix} \uparrow \\ \mathbf{u}_p \\ \downarrow \end{pmatrix}.$$

Best fit in the range

- The weights w_i quantify the contribution of feature *i* ($1 \leq i \leq p$) that contributes to the match between model output $\mathbf{Aw}$ and target $\mathbf{y}$. The best fit weight vector $\hat{\mathbf{w}}$ is the one for which the residual $\mathbf{r} = \mathbf{y} - \mathbf{A}\hat{\mathbf{w}}$ is smallest.
- The residual cannot have any component in the column space of $\mathbf{A}$ or else a different $\mathbf{w}$ would have projected the data matrix $\mathbf{A}$ along that component. This is the meaning of the equation

$$\mathbf{A}^T \mathbf{r} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

shown above. The residual is said to lie in the **nullspace** or **kernel** of the matrix $\mathbf{A}$, denoted ker($\mathbf{A}$).

- This leads to a formula for the best fit $\hat{\mathbf{w}}$:

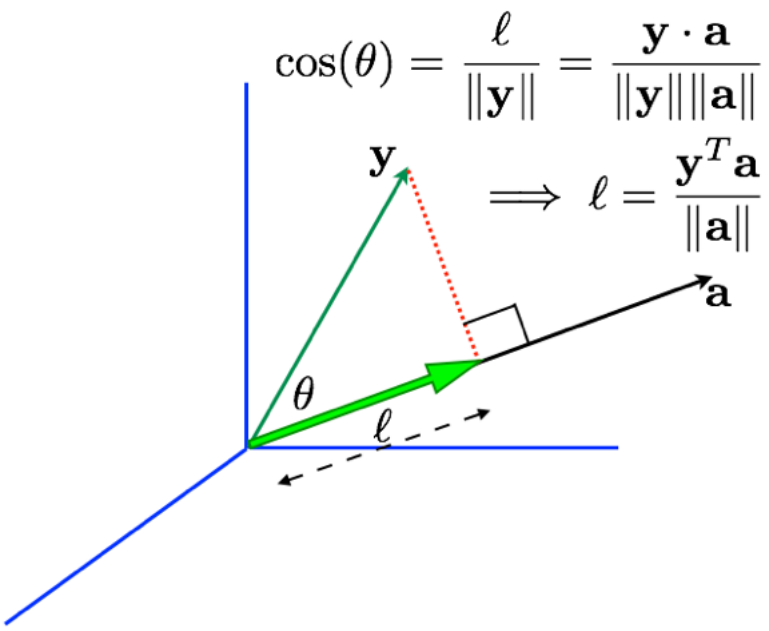
$$\mathbf{A}^T(\mathbf{y} - \mathbf{A}\hat{\mathbf{w}}) = 0 \implies \mathbf{A}^T\mathbf{A}\hat{\mathbf{w}} = \mathbf{A}^T\mathbf{y} \implies \hat{\mathbf{w}} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{y}.$$

- The matrix $\mathbf{A}^+$,

$$\mathbf{A}^+ \triangleq (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T$$

is called the **Moore-Penrose pseudo-inverse** of matrix $\mathbf{A}$.

- To expose the core ideas behind the forbidding notation of projections and column and null spaces, let's focus on a simple **visual** example. The figure shows in green the projection of $\mathbf{y}$ in the direction of $\mathbf{a}$.



- A **numerical** example: Let $\mathbf{a} = (1,1,1)^T$ and $\mathbf{y} = (1,2,2)^T$.
  The angle θ between 2 vectors is related to the dot product between them:

$$\cos(\theta) = \frac{\mathbf{a}^T\mathbf{y}}{\|\mathbf{a}\|\|\mathbf{y}\|}, \text{ where the norms } \|\mathbf{a}\|^2 = \mathbf{a}^T\mathbf{a}, \|\mathbf{y}\|^2 = \mathbf{y}^T\mathbf{y}.$$

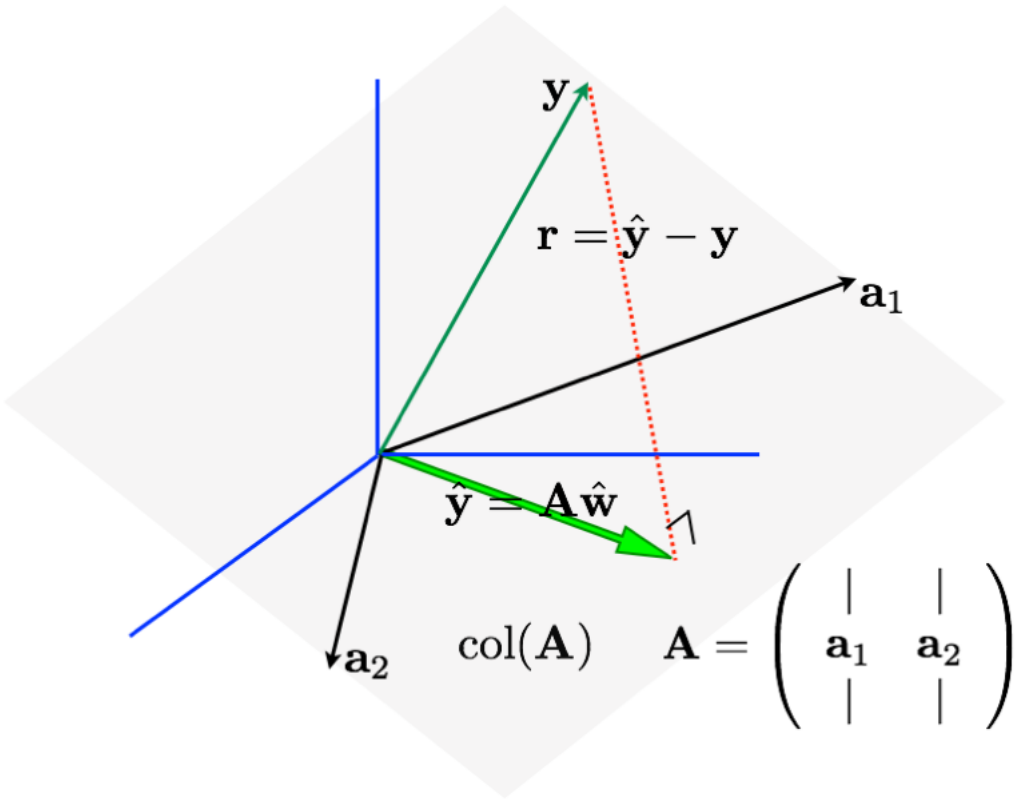
  The projection of $\mathbf{y}$ along the direction $\mathbf{a}$ has length $\text{proj}_{\mathbf{a}}(\mathbf{y})$, and the projected vector is $\text{proj}_{\mathbf{a}}(\mathbf{y})\mathbf{a}$:

$$\text{proj}_{\mathbf{a}}(\mathbf{y})\mathbf{a} = \left(\frac{\mathbf{y}^T\mathbf{a}}{\mathbf{a}^T\mathbf{a}}\right)\mathbf{a} = \frac{5}{3}\begin{pmatrix}1\\1\\1\end{pmatrix}.$$

The *residual* $\mathbf{r} := \mathbf{y} - \text{proj}_{\mathbf{a}}(\mathbf{y})\mathbf{a}$ is

$$\begin{aligned}\mathbf{r} &= (1,2,2)^T - (5/3)(1,1,1)^T \\ &= (1/3)(-2,1,1)^T, \\ &\Rightarrow \mathbf{r} \perp \mathbf{a}.\end{aligned}$$

- For a more general vector space, we show a planar (2-dimensional) column space of $\mathbf{A}$ to illustrate its orthogonality to the residual vector.



The residual vector is orthogonal to each of the columns of $\mathbf{A}$:

$$\mathbf{r} \cdot \mathbf{a}_i = 0, \ i = 0, 1, \ldots, p.$$

For $i = 0$, $\mathbf{a}_0$ is a vector of ones, so the sum of the residuals, and thus its mean over all data points is 0.

Generating features -- the quadratic example revisited

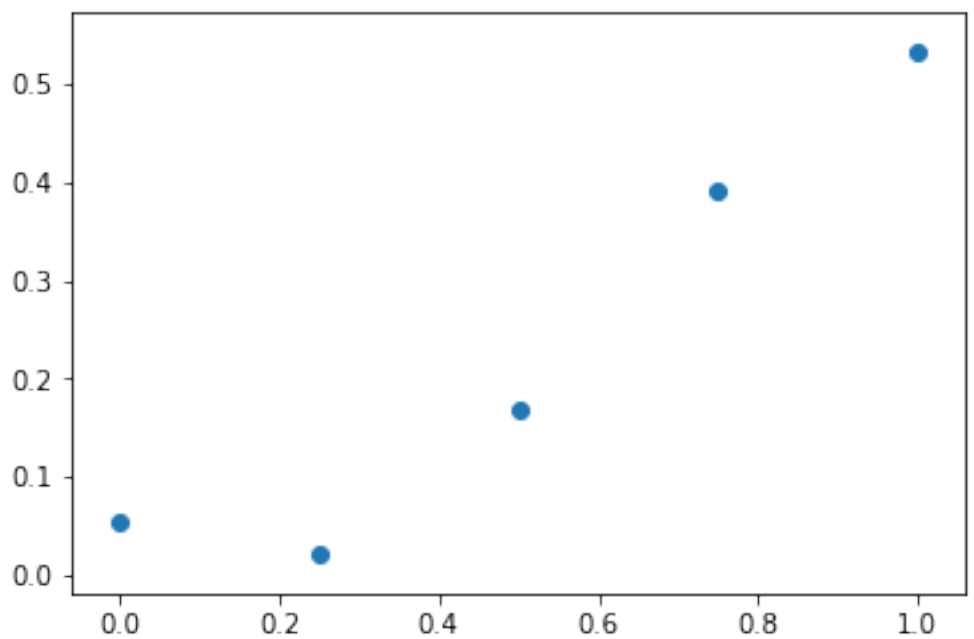
Earlier we used polyfit from numpy to obtain the best fit linear model (straight line) and best fit quadratic model (parabola). We will perform the same parabola learning task by generating the squares of the data points to construct the corresponding design matrix.

In [2]:

```
def f(x):
    return 0.5*(x)*(x**4)/(.05+(x**4))
npts=5
np.random.seed(0)
X=np.linspace(0,1,npts)
y0=f(X)+0.03*np.random.normal(0,1,npts)
plt.scatter(X,y0)
```

Out[2]:

```
<matplotlib.collections.PathCollection at 0x1156552b0>
```



Residuals orthogonal to column space of design matrix

In the next piece of code we first verify that the pseudo-inverse function call returns the same value as that in the definition. We then establish in these simple examples that the least square error is associated with a residual that is in the null space of the image of the design matrix.

First of all note that the design matrix is constructed by choosing $\phi_1(x) = x$ and $\phi_2(x) = x^2$. This means we are seeking to express every target value y_n as a function of the input value x_n as

$$y_n = w_0 + w_1 x_n + w_2 x_n^2,$$

and this expression corresponds to each of the rows of the simultaneous equations to be solved for all data points (x_n, y_n), for $n = 1, \ldots, N$. The column space view will seek to find the weights for each of the features (linear or quadratic functions of the input variables) in this choice of model.

In the python code next, the "stack" constructs the design matrix for the linear-quadratic model.

In [3]:

```
Xsq=np.square(X)
A1=np.stack((np.ones(npts),X,Xsq)).T
mpcomp = np.linalg.inv(A1.T.dot(A1)).dot(A1.T) # expression for Moore-Penrose
pseudoinverse
mpcall = np.linalg.pinv(A1) # pre-defined function call
print("Difference between Moore-Penrose pseudoinverse computed and by function
call:\n",
      mpcomp-mpcall)
what = np.linalg.pinv(A1).dot(y0)
print("Regression performed by best-fit formula: ",what)
print("Regression performed by numpy function call: ", np.polyfit(X,y0,deg=2))
res = np.atleast_2d(y0-A1.dot(what)) # residual vector
print("Length of residual: ",np.sqrt(np.square(res).sum()))
# Print the dot product of res with each column of design matrix A1;
# each column entry should be close to 0.
print("Dot product of residual with design matrix:\n",res.dot(A1))
```

```
Difference between Moore-Penrose pseudoinverse computed and by fun
ction call:
 [[ -3.33066907e-16   4.99600361e-16   3.05311332e-16   5.55111512
e-16
   -2.77555756e-17]
 [  8.88178420e-16  -2.77555756e-15  -8.88178420e-16  -1.99840144e
-15
   -1.77635684e-15]
 [  4.44089210e-16   2.22044605e-15  -8.88178420e-16   2.22044605e
-16
   -1.33226763e-15]]
Regression performed by best-fit formula:  [ 0.02762174  0.0495373
7  0.4818957 ]
Regression performed by numpy function call:  [ 0.4818957   0.0495
3737  0.02762174]
Length of residual:  0.0826871364018
Dot product of residual with design matrix:
 [[ -7.14706072e-16  -5.20417043e-16  -4.51028104e-16]]
```

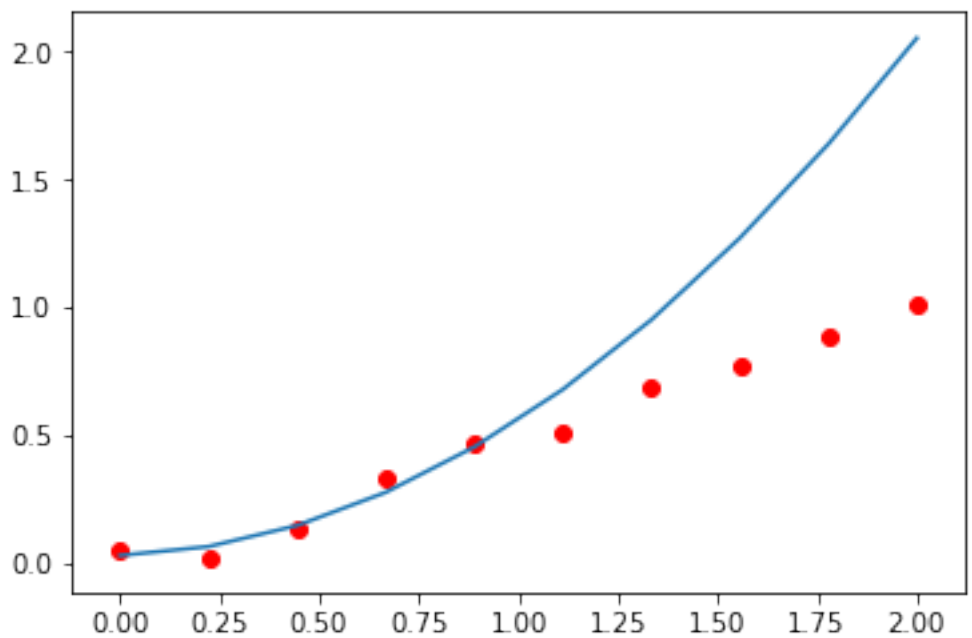
Having learnt the best weight vector, we see how it performs on unseen data from an extended range of x-values. Instead of $0 \leq x \leq 1$, the testing is performed on the range $0 \leq x \leq 2$

In [4]:

```
npred=10
Xpred = np.linspace(0,2,npred)
A2pred = np.stack((np.ones(npred),Xpred,np.square(Xpred))).T
np.random.seed(0)
y0new=f(Xpred)+0.03*np.random.normal(0,1,npred)
y2pred = A2pred.dot(what)
plt.plot(Xpred,y2pred)
plt.scatter(Xpred,y0new, color='r')
```

Out[4]:

```
<matplotlib.collections.PathCollection at 0x11568aeb8>
```



It is clear from the graph above that while the model fits the data in the same range as the training set (the range that was seen by the best-fit procedure, $0 \leq x \leq 1$) the predictions and real data diverge when $1 \leq x \leq 2$.

- **The model fails to generalise.**

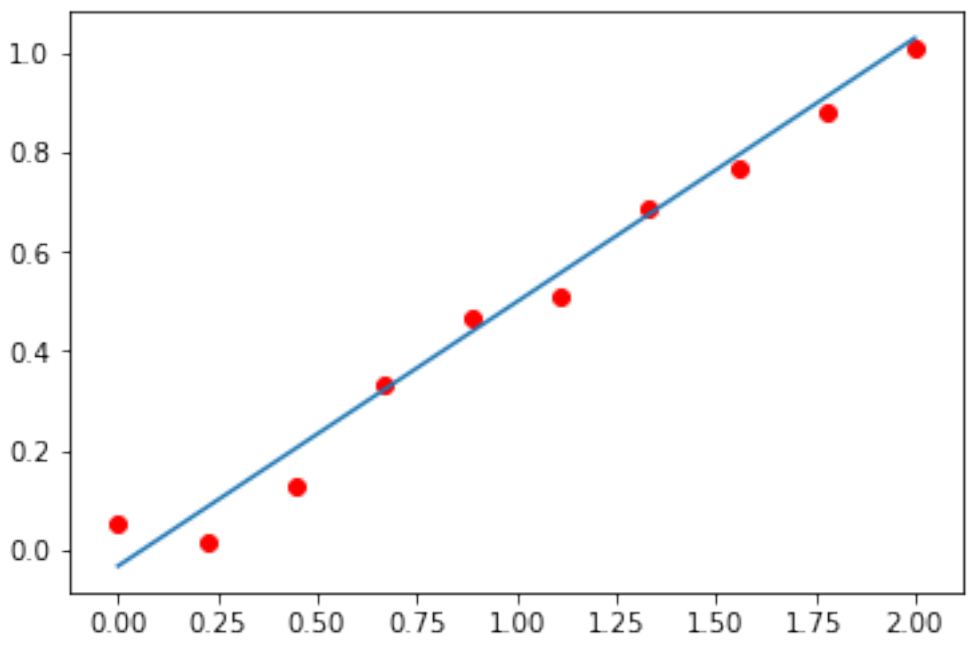
In [5]:

```
Alin=np.stack((np.ones(npts),X)).T
w0hat = np.linalg.pinv(Alin).dot(y0)
res0 = np.atleast_2d(y0-Alin.dot(w0hat)) # residual vector
print("Length of residual: ",np.sqrt(np.square(res0).sum()))
Alinpred = np.stack((np.ones(npred),Xpred)).T
y0pred = Alinpred.dot(w0hat)
plt.plot(Xpred,y0pred)
plt.scatter(Xpred,y0new, color='r')
```

```
Length of residual:  0.139774402086
```

Out[5]:

```
<matplotlib.collections.PathCollection at 0x1159ec5f8>
```



When we try to fit a linear model to the (x_i, y_i) for $0 \leq x_i \leq 1$ and plot it for $0 \leq x \leq 2$, we find a better generalisation performance. The simpler model appears to do better at generalising, even though the length of the residual vector is larger than that for the quadratic case.

- This observation, that a **simple model gives a better generalisation performance than a more complex one**, is common in machine learning.

Using basis functions other than polynomials

You should work through the examples below to get acquainted with the idea that linear methods can be used with complicated non-linear functions. The goal here is to try and use the basis functions 'chf(x,n)' to fit the functions 'chirp' and 'somefun'. We wish to find w_i so that samples $(x_n, y_n = F(x_n))$ are reproduced using

$$y_n = \sum_i w_i \times \mathrm{chf}(x, i).$$

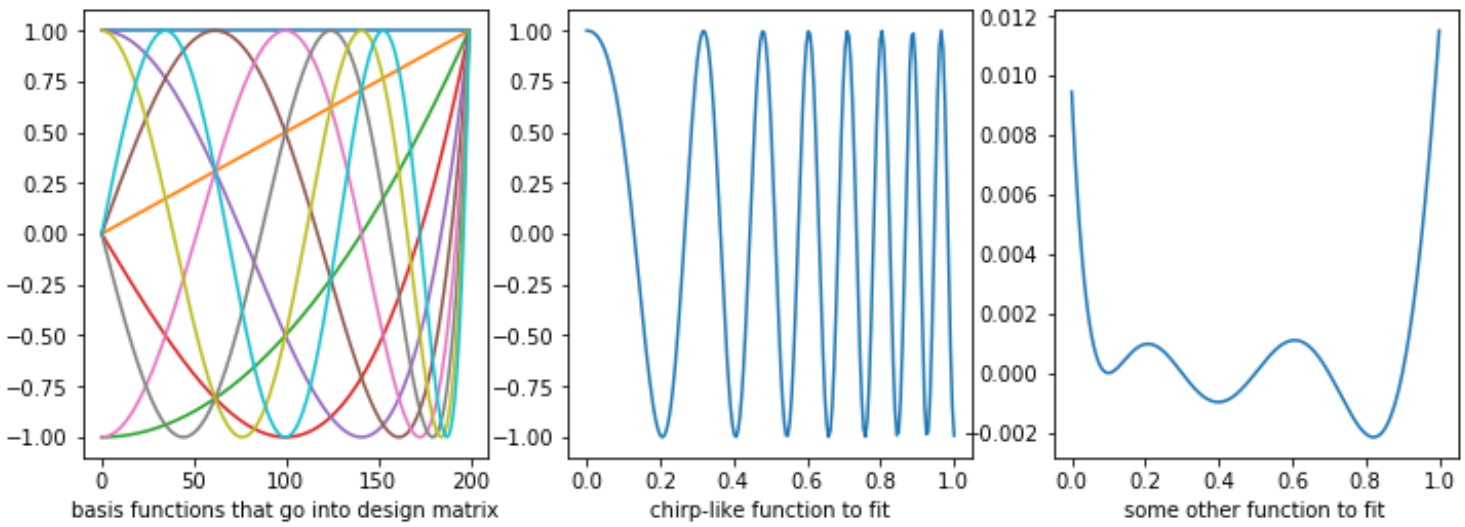
The design matrix is constructed by stacking the functions 'chf(x,n)' using 'vstack'.

In [15]:

```
def chf(x, n): # These are the basis functions to be used
    return np.cos(n*np.arccos(x))
def chirp(w1, w2, x, M): # function to draw datapoints from
    return np.cos(w1*x + (1/M)*(w1-w2)*x**2)
def somefun(x): # yet another -- you should try out your own
    return 10*np.exp(-2*x**2)*(x - 0.9)*(x - 0.7)*(x - 0.5)*(x - 0.3)*(x - 0.1
)**2

X1=np.linspace(0,1,200)
fig, ax = plt.subplots(ncols=3,nrows=1,figsize=(12,4))
for n in range(10):
    ax[0].plot(chf(X1,n))
ax[1].plot(X1,chirp(7.,6.,X1,.025))
ax[2].plot(X1,somefun(X1))
ax[0].set_xlabel('basis functions that go into design matrix')
ax[1].set_xlabel('chirp function to fit')
ax[2].set_xlabel('function somefun to fit')
```

Out[15]:

```
<matplotlib.text.Text at 0x117b8b198>
```

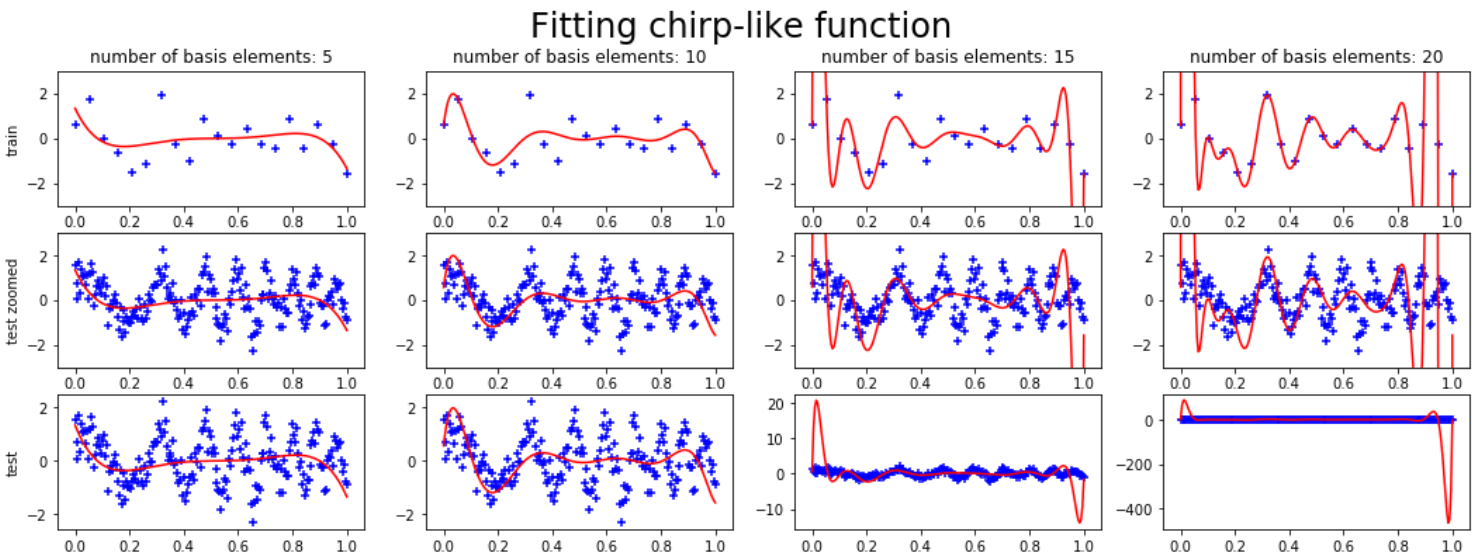


In [22]:

```
num_basis = [5, 10, 15, 20]
noise = 0.5
fig, ax = plt.subplots(nrows=3,ncols=len(num_basis),figsize=(18,6))
ntrain=20
Xtrain=np.linspace(0,1,ntrain)
def F(x):
    return chirp(7.,6.,x,.025)
ytrain=np.atleast_2d(F(Xtrain)+noise*np.random.normal(0,1,ntrain)).T


ntest=200
Xtest=np.linspace(0,1,ntest)
ytest=F(Xtest)+noise*np.random.normal(0,1,ntest)


for i, nb in enumerate(num_basis):
    ax[0][i].scatter(Xtrain,ytrain,marker='+',color='b')
    Amat = [np.ones(ntrain)]
    Amat_test = [np.ones(ntest)]
    for n in range(nb):
        Amat.append(chf(Xtrain,n+1))
    Amat = np.vstack(Amat).T
    what = np.linalg.pinv(Amat).dot(ytrain)
    y1train=Amat.dot(what)
    for n in range(nb):
        Amat_test.append(chf(Xtest,n+1))
    Amat_test = np.vstack(Amat_test).T
    ypred=Amat_test.dot(what)
    ax[1][i].scatter(Xtest,ytest,color='b',marker='+')
    ax[0][i].plot(Xtest,ypred,color='r')
    ax[1][i].plot(Xtest,ypred,color='r')
    ax[0][i].set_ylim([-3,3])
    ax[1][i].set_ylim([-3,3])
    ax[2][i].scatter(Xtest,ytest,color='b',marker='+')
    ax[2][i].plot(Xtest,ypred,color='r')
    title = "number of basis elements: "+str(num_basis[i])
    ax[0][i].set_title(title)
    title = ""
ax[0][0].set_ylabel('train')
ax[1][0].set_ylabel('test zoomed')
ax[2][0].set_ylabel('test')
fig.suptitle(r'Fitting chirp',fontsize=24)
```

Out[22]:

<matplotlib.text.Text at 0x119777b70>

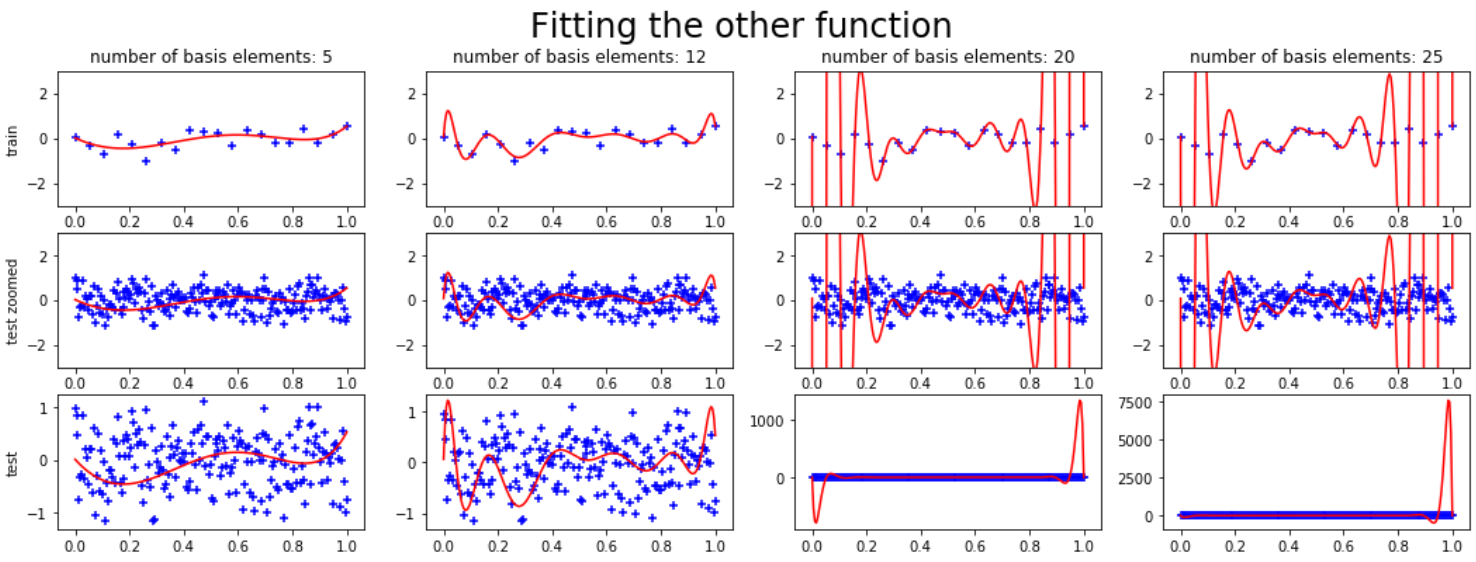


In [23]:

```
# Exactly the same as the previous cell, but for a different target function

num_basis = [5, 12, 20, 25]
noise = 0.5
fig, ax = plt.subplots(nrows=3,ncols=len(num_basis),figsize=(18,6))
ntrain=20
Xtrain=np.linspace(0,1,ntrain)
def F(x):
#    return chirp(7.,6.,x,.025)
    return somefun(x)
ytrain=np.atleast_2d(F(Xtrain)+noise*np.random.normal(0,1,ntrain)).T


ntest=200
Xtest=np.linspace(0,1,ntest)
ytest=F(Xtest)+noise*np.random.normal(0,1,ntest)


for i, nb in enumerate(num_basis):
    ax[0][i].scatter(Xtrain,ytrain,marker='+',color='b')
    Amat = [np.ones(ntrain)]
    Amat_test = [np.ones(ntest)]
    for n in range(nb):
        Amat.append(chf(Xtrain,n+1))
    Amat = np.vstack(Amat).T
    what = np.linalg.pinv(Amat).dot(ytrain)
    y1train=Amat.dot(what)
    for n in range(nb):
        Amat_test.append(chf(Xtest,n+1))
    Amat_test = np.vstack(Amat_test).T
    ypred=Amat_test.dot(what)
    ax[1][i].scatter(Xtest,ytest,color='b',marker='+')
    ax[0][i].plot(Xtest,ypred,color='r')
    ax[1][i].plot(Xtest,ypred,color='r')
    ax[0][i].set_ylim([-3,3])
    ax[1][i].set_ylim([-3,3])
    ax[2][i].scatter(Xtest,ytest,color='b',marker='+')
    ax[2][i].plot(Xtest,ypred,color='r')
    title = "number of basis elements: "+str(num_basis[i])
    ax[0][i].set_title(title)
    title = ""
ax[0][0].set_ylabel('train')
ax[1][0].set_ylabel('test zoomed')
ax[2][0].set_ylabel('test')
fig.suptitle(r'Fitting somefun',fontsize=24)
```

Out[23]:

<matplotlib.text.Text at 0x11a2c30b8>



Singular Value Decomposition of the design matrix

We will explore how the SVD can be used as a tool to explore how the features contribute to the efficacy of capturing the functional dependence between (input, output) data in the training and test sets.

First, the exploration of how many singular vectors is needed to reproduce the matrix itself. You could try out the "reconstruction_error" function with different values of n, $1 \leq n \leq 20$.

In [9]:

```
[u,s,v]=np.linalg.svd(Amat)
print(Amat.shape,u.shape, s.shape, v.shape)
```

(20, 26) (20, 20) (20,) (26, 26)

In [10]:

```
def smat_reduced(shape, sdiag, n):
    # creates matrix with n elements of sdiag along diagonal
    smat = np.zeros(shape, dtype=complex)
    drop = len(sdiag)-n
    if (drop > 0):
        sdiag=np.concatenate((sdiag[:-drop],np.zeros(drop)))
    smat[:len(sdiag),:len(sdiag)]=np.diag(sdiag)
    return smat


def svd_to_mat(u, s, v, n):
    # computes a rank n matrix using (u, s, v) coming from svd(some matrix)
    smat = smat_reduced((len(u),len(v)),s,n)
    return np.dot(u,np.dot(smat,v))


def svd_reconstruction_error(mat, n):
    [u, s, v] = np.linalg.svd(mat)
    # the error upon creating rank n matrix from (u, s, v) computed here
    return np.square(mat-svd_to_mat(u,s,v,n)).sum()
```

Using the code given, we can calculate the reconstruction error upon retaining all 20 of the singular values is 10^{-28}.

In [11]:

```
print(svd_reconstruction_error(Amat,20))
```

```
(8.21014907755e-28+0j)
```

Exercise

- Construct a design matrix using a polynomial basis on some (input, output) data
- Perform a reduced rank decomposition of the design matrix for different ranks, noting the reconstruction error
- Explore for yourself how the nature of the fit changes as the rank is altered, both for training and test sets
- Repeat the same using the basis functions chf(x, n) that I introduced above.