YieldCurve

A *Mathematica* Package for Fitting the Term Structure of Interest Rates with Smoothing Splines

Mark Fisher and David Zervos

The term structure of interest rates

The term structure of interest rates occupies a central position in both macroeconomics and finance. The fundamental relationship is the **discount function**, $\delta(t,\tau)$, which gives the price at time t of default-free zero-coupon bond that pays one unit at time τ. Hereafter, we assume that the current time is 0, suppress the first index, and write $\delta(\tau)$. Thus $\delta(\tau)$ is the present value of one unit to be delivered with certainty in τ periods hence. We will find it convenient to look at the **log of the discount function**, $\ell(\tau) := \log[\delta(\tau)]$.

Interest rates can be derived from bond prices. The **zero-coupon (yield) curve**, $z(\tau) := -\log[\delta(\tau)]/\tau = -\ell(\tau)/\tau$, gives the yield-to-maturity on a zero-coupon bond that matures at time τ. The **forward rate curve**, $f(\tau) := -d\log[\delta(\tau)]/d\tau$, gives the marginal return at maturity τ of extending one's investment. We can write the forward curve in terms of the zero curve by first writing $\delta(\tau) = e^{-\tau\,z(\tau)}$ and then applying the definition for forward rates: $f(\tau) = z(\tau) + \tau \,z'(\tau)$. Thus we see that the relationship between the zero and forward curves is that of average and marginal curves.

The techniques embodied in the **`YieldCurve`** package are designed to extract the term structure from a set of bonds (*i*) that are default-free and (*ii*) whose prices are determined by the present value of their stated payments. All U.S. Treasury securities probably satisfy the first criterion, but some do not reasonably satisfy the second criterion, such as callable bonds, ``flower'' bonds, and bonds ``on special'' in the repo market.

Consider a set of n bonds. Let p_i be the price of bond i, c_{ij} be its j-th payment, paid at time τ_{ij}, and m_i be the number of remaining payments. Then\footnote{^{1}}{``$^\top$'' denotes transpose. In addition, the prices, p_i, include accrued interest.}

$$ p_i = \sum_{j=1}^{m_i} c_{ij}\,\delta(\tau_{ij}) + \varepsilon_i ={c_i}^\top \widetilde{\delta}(\tau_i) + \varepsilon_i $$

where c_i is the vector of payments for bond i, τ_i is the vector of maturities of those payments, ε_i is a random error and

$$\widetilde{\delta}(\tau_i) := (\delta(\tau_{i1}), \cdots, \delta(\tau_{im_{i}}))^\top$$

is the $m_i\times 1$ column vector that results from applying δ to each element of τ_i. Thus, the present value of the payments is given by $\pi_i := {c_i}^\top \widetilde{\delta} (\tau_i)$.

Fitting the term structure

Estimating the term structure from bond prices is not, however, a trivial matter. A variety of techniques have been proposed over the past twenth-five years. McCulloch (1971, 1975) was the pioneer in this field. Essentially, McCulloch parameterized $\delta(\tau)$ as a cubic spline and estimated the spline coefficients with linear regression. Following McCulloch, Vasicek and Fong (1982), Shea (1984), Jordan (1984), Chambers, Carleton and Waldman (1984), and Coleman, Fisher, and Ibbotson (1992), among others, extended the spline-based estimation technique to explore tax-related effects on bond pricing, to consider different parameterizations of the splines, and to analyze potential sources of heteroskedasticity in the residuals. Other authors have pursued alternative estimation techniques based on parsimonious parameterizations of the discount function. For instance, Nelson and Seigel (1987) and Bliss (1993) consider a functional form with only four unknown parameters. (In contrast, for a sample of 150 securities, McCulloch would typically choose a spline with 18 parameters.)

Fisher, Nychka, and Zervos (1995) present an extension of the spline-based techniques (see their paper for the specifics.) In particular, they fit smoothing splines instead of regression splines. Smoothing splines have a penalty for excess ``roughness'' with a single parameter that controls the size of the penalty. An increase in the penalty reduces the effective number of parameters, so that a single value controls the entire parameterization of the spline. For regression splines, the number of parameters must be chosen in advance. By contrast, Fisher, Nychka, and Zervos use **generalized cross validation** to choose adaptively the roughness penalty---and hence the effective number of parameters. In other words, they let the data determine the appropriate number of parameters. In addition, they place the spline directly on the log of the discount function $\ell(\tau)$ and on the forward rate function $f(\tau)$, as well as on the discount function $\delta(\tau)$. Based on their simulations and on their estimation results (using daily data on U.S. Treasury coupon securities from December of 1987 though September 1994), they found that splining the forward rate function with a smoothing spline and choosing the effective number of parameters via generalized cross validation produced the most accurate and least biased results.

This chapter describes the *Mathematica* package **`YieldCurve`** that implements the techniques described in Fisher, Nychka, and Zervos. **`YieldCurve`** contains commands to estimate the term structure of interest rates using regression splines and smoothing splines---with or without generalized cross validation. **`YieldCurve`** also has functions to display and analyze results and produce reports. **`YieldCurve`** calls a number of other Mathematica packages---both standard packages and other included packages.

■ File names: Packages and Data

To install the packages, you should create a subdirectory, for example **ycurve**. The following files should be copied to this subdirectory: **YieldCurve.m**, **ShowTime.m**, **PageLayout.m**, **BSplineBasis.m**, **CommaDelimited.m**, **TriangularPlot3D.m**, **MakeCalendar.m**, and **daycount.m**.\footnote{^{2}}{If are using DOS or Windows, the files names are **yieldcur.m**, **showtime.m**, **pagelayo.m**, **bsplineb.m**, **commadel.m**, **triangul.m**, **makecale.m** and **daycount.m**.} The first seven files contain *Mathematica* packages. The file **daycount.m** contains information for determining the number of days until a payment is made. In addition, there are seven files with names of the form **yyyymmdd.dat** that contain data on Treasury securities, as well as **strips.dat** and **openmkt.dat**. The final file in the distribution is **iddrop.m**, which contains a list of the CUSIPs of securities with special features.

Two other files will be created in your working directory as you use the main package: **LStart.m** and **DLStart.m** If **FunctionalForm -> LogDelta** or **FunctionalForm -> DLogDelta**, **DiscountFunction** will create the file **LStart.m** or **DLStart.m** in the default directory. It is possible that bad starting values in **LStart.m** or **DLStart.m** could lead to difficultly in estimation. In any case, either file can always be deleted, causing internal defaults to be used.

MakeCalendar.m contains functions to create and combine calendar lists. You will only need to use it if you wish to extend the calendar before 1925 or after 2035 or if you wish to use bonds from other countries (which have different holidays).

Also included are the files **CRSPDailyBonds.m** and **CRSPMonthlyBonds.m**. These files contain packages that provide access to the Center for Research in Security Prices (CRSP) Daily and Monthly U.S. Government Bond Files. Each package contains (among other things) the function **MassageCRSPData**, which reads the (ASCII) data files and produces an **MD** object. Interested users should read the files for additional information.

B-spline bases

A spline is a piecewise polynomial joined at so-called knot points. Let the order of the polynomial be given by r. Thus a cubic B-spline is order 3, while a step function is order 0. At each knot point, the polynomials that meet are restricted so that one additional independent parameter is added. For example, a step function has only one parameter, and thus there are no restrictions between adjacent step functions for an 0-order spline. By contrast, a cubic polynomial has four parameters; thus for a cubic spline, the level and first two derivatives of each cubic are restricted to be identical at the knot points.

A numerically stable parameterization of a spline is provided by a B-spline basis. Let $\{s_k\}_{k = 1}^K$ denote the knot points, with $s_k < s_{k+1}$, $s_1 = 0$, and $s_K = M$, the maximum maturity of any bond in the sample.\footnote{^{3}}{In all cases, we distribute the knot points according to the distribution of the final maturities of the bonds. For example, with three knot points, we place the single interior knot point, s_2, at the median maturity.} The knot points define $K-1$ intervals over the domain of the spline, $[0,\,T]$. For the purpose of defining a B-spline basis, it is convenient to define an augmented set of knot points, $\{d_k\}_{k=1}^{K+2\,r}$, where $d_1 = \cdots = d_r = s_1$, $d_{K+r+1} = \cdots = d_{K+2\,r} = s_K$, and $d_{k+r} = s_k$ for $1 \le k \le K$. Then a B-spline basis is a vector of $\kappa = K + r - 1$ order-r B-splines defined over the domain.

Here is an example.

```
SetDirectory["/tr/data1/m1mef00/math/ycurve"]; (* your
           directory here *)
Needs["YieldCurve`"]; (* loads BSplineBasis as well *)
Off[ShowTime]; (* turn off timings *)

knots = Range[0, 5];
basisknots = PadKnots[knots, 3]
```

```
{0, 0, 0, 0, 1, 2, 3, 4, 5, 5, 5, 5}
```

A B-spline is defined by the following recursion, where $1\le k \le \kappa $:\footnote{^{4}}{For a more detailed discussion of b-spline bases and their properties, see de Boor (1978).}

```
(* i-th spline, r-th order, evaluated at tau *)
phi[i_, r_, tau_] :=
  ((tau - k[i]) phi[i, r - 1, tau])/(k[i + r] - k[i]) +
    ((k[i + r + 1] - tau) phi[i + 1, r - 1, tau])/
        (k[i + r + 1] - k[i + 1])

(* terminal condition *)
phi[i_, 0, tau_] := d[i]

(* i-th spline, j-th region, r-th order,
    evaluated at tau *)
phi[i_, j_, r_, tau_] :=
    phi[i, r, tau] /. {d[j] -> 1, d[_] -> 0}
```

We can see that the fourth cubic B-spline is a function of five knot points beginning with **k[4]**:

```
Cases[phi[4, 3, tau], k[_], Infinity] //Union
```

```
{k[4], k[5], k[6], k[7], k[8]}
```

The functional form for the fourth cubic spline over the sixth region (between **k[6]** and **k[7]**) is given by

```
phi[4, 6, 3, tau]
```

$$\frac{(\text{tau} - k[4])\,(-\text{tau} + k[7])^2}{(-k[4] + k[7])\,(-k[5] + k[7])\,(-k[6] + k[7])} + \left((-\text{tau} + k[8])\left(\frac{(\text{tau} - k[5])\,(-\text{tau} + k[7])}{(-k[5] + k[7])\,(-k[6] + k[7])} + \frac{(\text{tau} - k[6])\,(-\text{tau} + k[8])}{(-k[6] + k[7])\,(-k[6] + k[8])}\right)\right) / (-k[5] + k[8])$$

The function **MakeBSpline** will produce a **BSplineFunction**:

```
?MakeBSpline
```

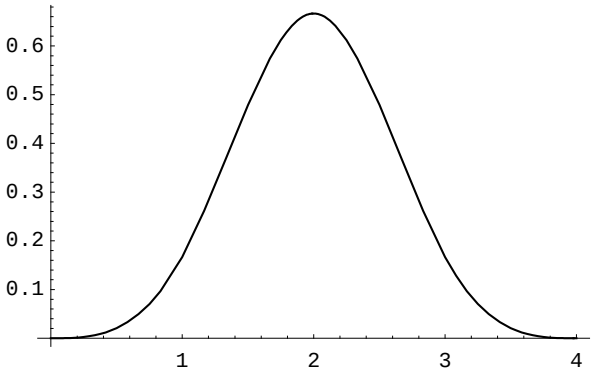
```
MakeBSpline[knots, d:0] takes a list of knot points and
   returns an InterpolatingFunction that represents a
   B-spline of order Length[knots] - 2. Valid orders are 0
   (step function), 1 (linear), 2 (quadratic), and 3
   (cubic), or higher. (Higher order splines use less
   efficient routines.) If the optional second argument is
   given as 1, then the derivative of the spline is
   returned. If the optional third argument is given as -1,
   then the integral of the spline is returned.
   Differentiation is supported to any order; integration is
   supported to order -2.
```

We can construct a cubic BSplineFunction for the forth B-spline over the sixth region:

```
bs463 = MakeBSpline[Take[basisknots, {4, 8}]]
```

```
BSplineFunction[{0, 4}, <>]
```

```
Plot[bs463[tau], {tau, 0, 4}];
```



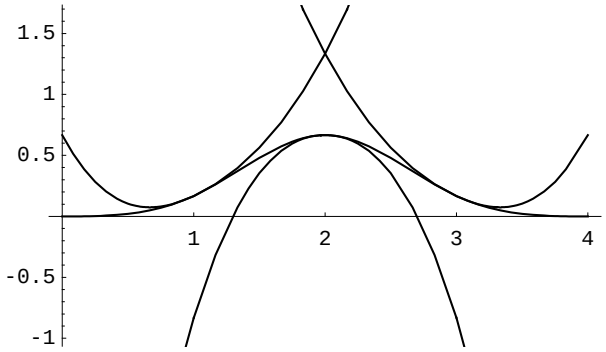
The internal structure of a **`BSplineFunction`** can be seen to be an **`InterpolatingFunction`** wrapped in a **`Which`**:

```
InputForm[bs463]
```

```
BSplineFunction[{0, 4}, Which[#1 < 0, 0, #1 <= 4,
    InterpolatingFunction[{0, 4},
      {{0, 0, {0, 0}, {0}},
       {1, 0, {1/6, 1/2, 1/3, 1/6}, {0, 0, 1}},
       {2, 1, {2/3, 0, -1/2, -1/2}, {0, 0, 1}},
       {3, 2, {1/6, -1/2, 0, 1/2}, {0, 0, 1}},
       {4, 3, {0, 0, 1/6, -1/6}, {0, 0, 1}}}][#1], True, 0]\
     & ]
```

The **`InterpolatingFunction`** contains information about the polynomials that make up the B-spline. We can see the polynomials with **`PlotIFPolynomials`**:

```
PlotIFPolynomials[bs463];
```



A **B-spline basis** is a row vector:

$$\phi^r(\tau) := (\phi_1^r(\tau),\cdots,\phi_\kappa^r(\tau)). $$

For example, we can make a cubic B-spline basis and evaluate it at $\tau = 3/2$:

```
phi3 = MakeBSplineBasis[knots, 3]
Through[phi3[3/2]]
```

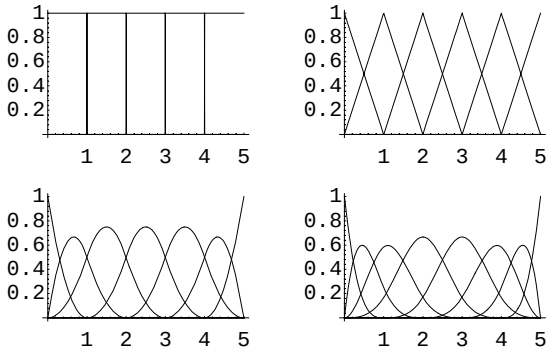
```
{BSplineFunction[{0, 1}, <>], BSplineFunction[{0, 2}, <>],
  BSplineFunction[{0, 3}, <>], BSplineFunction[{0, 4}, <>],
  BSplineFunction[{1, 5}, <>], BSplineFunction[{2, 5}, <>],
  BSplineFunction[{3, 5}, <>], BSplineFunction[{4, 5}, <>]}
```

$$\{0, \frac{1}{32}, \frac{15}{32}, \frac{23}{48}, \frac{1}{48}, 0, 0, 0\}$$

Over any interval between adjacent knot points, s_k and s_{k+1}, there are $r+1$ non-zero B-splines, with adjacent intervals sharing r. This gives $\phi^r(\tau)$ a quasi-orthogonal structure from which it gets its numerical stability.

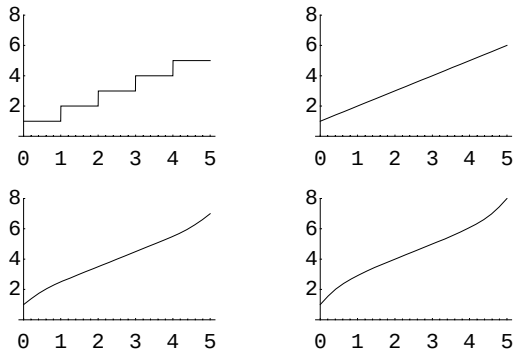
Here we plot $\phi^r(\tau)$ for r from 0 to 3 and τ from 0 to 5:

```
bases = Table[MakeBSplineBasis[knots, i], {i, 0, 3}];
Show[GraphicsArray[Partition[
    PlotBSplineBasis[#,
        DisplayFunction -> Identity]& /@ bases, 2],
        DisplayFunction -> $DisplayFunction]];
```



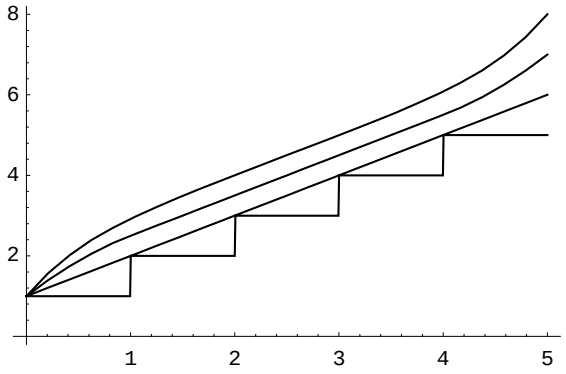
Any r-order spline can be constructed from linear combinations of the B-splines, $\phi^r(\tau)\,\beta$, where $\beta:=(\beta_1,\cdots,\beta_\kappa)^\top$ is a vector of coefficients. Let's plot $\phi^r(\tau)\,\beta$ for the given bases, where the βs are given by **`Range[r + 5]`**:

```
betas = Table[Range[r + 5], {r, 0, 3}];
splines = MapThread[Through[#1[tau]] . #2 &,
    {bases, betas}];
Show[GraphicsArray[Partition[
    Plot[#, {tau, 0, 5}, PlotRange -> {0, 8},
        Ticks -> {Automatic, {2,4,6,8}},
        DisplayFunction -> Identity]& /@ splines, 2]],
        DisplayFunction -> $DisplayFunction];
```



As an alternative, we can construct the linear combination directly with **`MakeFinalSpline`**:

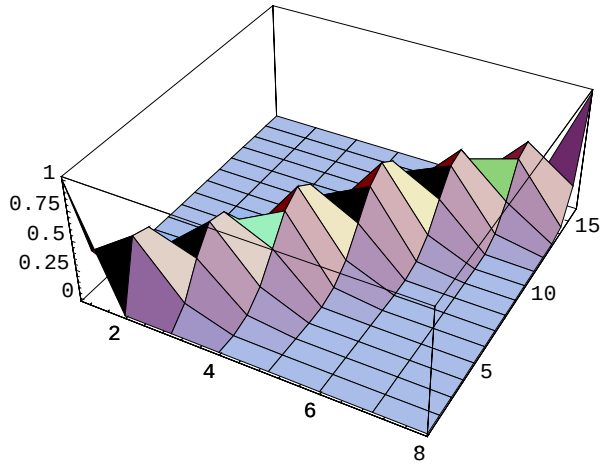
```
fsplines = MakeFinalSpline[knots, #]& /@ betas;
Plot[Evaluate[Through[fsplines[tau]]], {tau, 0, 5}];
```



As is stands, $\phi^r(\tau)$ is a vector-valued function of a scalar argument, τ. In what follows, it will prove useful to have notation for a B-spline basis as a function of vector-valued argument, τ_i. To that end, define $\widetilde\phi_k^r(\tau_i) := (\phi_k^r(\tau_{i1}),\cdots, \phi_k^r(\tau_{im_i}))^\top$, an $m_i \times 1$ column vector, and $\widetilde\phi^r(\tau_i) := (\widetilde\phi_1^r(\tau_i),\cdots, \widetilde\phi_\kappa^r(\tau_i)),$ an $m_i\times \kappa$ matrix.

The function **`SplineMatrix`** returns the matrix $\widetilde\phi^r(\tau_i)$. The quasi-orthogonal structure is evident in a plot:

```
taui = Table[i, {i, 0, 5, 1/3}];
phi3mat = SplineMatrix[knots, taui, 3];
ListPlot3D[phi3mat];
```



Splining functional forms

Consider splining each of the three functional forms: let $\delta_s(\tau) := \phi^r(\tau)\,\beta$, $\ell_s(\tau) := \phi^r(\tau)\,\beta$, and $f_s(\tau) := \phi^r(\tau)\,\beta$. In the latter two cases, we can transform the splined functional form to produce a discount function that can be used to represent present values

$$\delta_s^\ell(\tau) := \exp(-\ell_s(\tau)) = \exp(\phi^r(\tau)\, \beta)$$

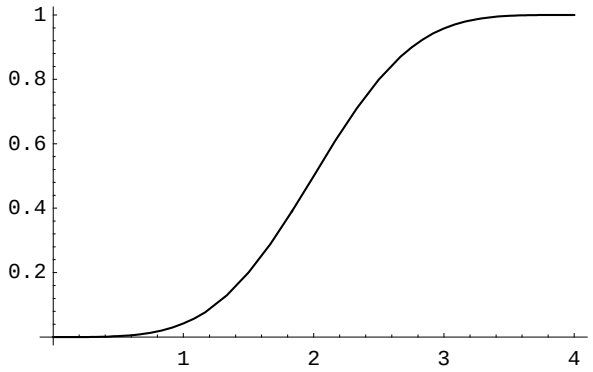
and

$$\delta_s^f(\tau) = \exp(-\int_0^\tau f_s(u)\,du) = \exp(-\int_0^\tau \phi^r(u)\,\beta \, du) = \exp(-\psi(\tau)\, \beta),$$

where $\psi(\tau) := \int_0^\tau \phi^r(u)\,\beta \, du$ is the integral of a B-spline basis.
We replace $\widetilde\delta(\tau_i)$ in the present value expression with the spline representation; where we had $\pi_i = c_i^\top\widetilde\delta(\tau_i)$, we now have (*i*) $\pi_i^\delta(\beta) = c_i^\top\widetilde\phi^r(\tau_i)\,\beta$, (*ii*) $\pi_i^\ell(\beta) = c_i^\top \exp(-\widetilde\phi^r(\tau_i)\,\beta)$, and (*iii*) $\pi_i^f(\beta) = c_i^\top \exp(-\widetilde \psi^r(\tau_i)\,\beta)$.

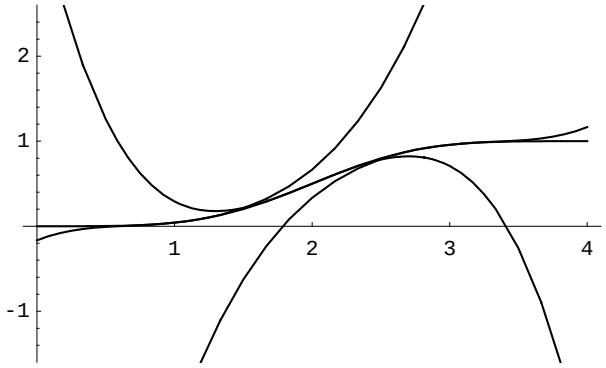
In order to spline the forward-rate curve, we need the integrals B-splines, whic all of spline-making functions can produce. For example,

```
bs463i = MakeBSpline[Take[basisknots, {4, 8}], -1];
Plot[bs463i[x], {x, 0, 4}];
```



Mathematica automatically differentiates an **`InterpolatingFunction`** but will not integrate it. The integral of an **`InterpolatingFunction`** is instead calculated by **`IntegrateIF`**, which returns another **`InterpolatingFunction`**, which we can visually decompose:

```
PlotIFPolynomials[bs463i];
```



`SplineMatrix` can return $\widetilde \psi^r(\tau_i)$. The last argument indicates the order of differentiation (in this case integration):

```
phi3int = SplineMatrix[knots, {1, 3/2, 4.2}, 3, -1]

{{1/4, 7/16, 13/48, 1/24, 0, 0, 0, 0},
  {1/4, 127/256, 141/256, 77/384, 1/384, 0, 0, 0},
  {1/4, 1/2, 3/4, 1, 0.982933, 0.587867, 0.1288, 0.0004}}
```

We see that the integral of a B-spline basis does not retain the quasi-orthogonality of the B-Spline basis.

Regression Splines

Since neither $\pi_i^\ell(\beta)$ nor $\pi_i^f(\beta)$ is linear in β, we must use some sort of nonlinear least squares to fit the spline when we choose either of these functional forms.

Let P be an $n\times 1$ vector of bond prices, p_i, and ${\mit \Pi}(\beta)$ be the corresponding vector of present values of the bonds, $\pi_i^j(\beta)$. Then for a regression spline, we choose the β that minimizes the sum of squared residuals:

$$\min_\beta\, [(P-{\mit \Pi}(\beta))^\top(P-{\mit \Pi}(\beta))].$$

Using a standard technique (see Chow (1983)), we linearize ${\mit \Pi}(\beta)$ around an initial guess β^0,

$${\mit \Pi}(\beta) \approx {\mit \Pi}(\beta^0) + \left.(\beta - \beta^0){\partial{\mit \Pi} (\beta) \over \partial \beta^\top}\right|_{\beta = \beta^0},$$

and define $X(\beta^0) := \left. {\partial {\mit \Pi}(\beta)/\partial\beta^\top} \right|_{\beta=\beta^0}$ and $Y(\beta^0) := P - {\mit \Pi}(\beta^0) + \beta^0\, X(\beta^0)$. Rearranging the minimization problem using these definitions yields

$$ \min_{\beta} \left[\Big(Y(\beta^0) - X(\beta^0) \beta \Big)^\top \Big(Y(\beta^0) - X(\beta^0) \beta \Big)\right].$$

The minimizer for the previous expression is

$$\beta^1 = \Big(X(\beta^0)^\top X(\beta^0)\Big)^{-1} X (\beta^0)^\top Y(\beta^0),$$

where β^1 is an updated β^0. We can use β^1 as the initial guess for the next iteration, obtaining β^2. We iterate until convergence. The solution is the fixed-point $\beta^* = (X(\beta^*)^\top X(\beta^*))^{-1} X(\beta^*)^\top Y(\beta^*)$.

By contrast, splining the discount function is easier, since $\pi_i^\delta(\beta)$ is linear in β. The relationship between bond prices and present values can be written $p_i = x_i\,\beta + \varepsilon_i$, where the ``independent variables'' are the bond payments evaluated according to the B-spline basis; *i.e.* $x_i := c_i^\top\widetilde \phi^r(\tau_i)$. Now let X be the matrix of corresponding to the x_i. We can use ordinary least squares (OLS) to determine $\beta^* = (X^\top X)^{-1} X^\top P$. This produces a regression spline of the discount function of the sort McCulloch used.

Smoothing Splines

A smoothing spline is a regression spline with a penalty for roughness. With a smoothing spline, one can use a large number of knot points but penalize excess variability in the estimated discount function, which has the effect of reducing the effective number of parameters since the penalty forces implicit relationships between the parameters of the spline. The penalty we describe here applies only to a cubic spline. Let $h(\tau)$ be the function being splined. The penalty is defined as

$$\lambda \int_0^T h"(\tau)^2 d\tau,$$

a constant times the integral of the squared second derivative of the function being splined.

The penalty can be written in terms of the B-spline basis as follows:

$$\lambda \int_0^T \left({\partial^2 \phi^3(\tau)\, \beta \over \partial \tau^2}\right)^2 \, d\tau = \lambda\, \beta^\top\left(\int_0^T \phi^3"(\tau)^\top\phi^3"(\tau) d\tau\right)\beta =\lambda \,\beta^\top H\beta .$$

H is a $\kappa\times\kappa$ matrix that is band diagonal by the structure of a B-spline basis. Since any β that makes $h(\tau)$ linear in τ is not penalized, H has two zero eigenvalues. Also note that H is completely determined by the knot points. For example, we can construct H for the knots given above:\footnote{^{5}}{We thank Jon Faust and Ludger Hentschel for the code that constructs the penalty matrix.}

```
H = PenaltyMatrix[knots, 3];
NullSpace[H]
```

```
{{-14, -13, -11, -8, -5, -2, 0, 1},
  {15, 14, 12, 9, 6, 3, 1, 0}}
```

We see the the null space of H is indeed two-dimensional.

The minimization problem can be stated as follows for a given λ:

$$ \min_{\beta(\lambda)} \left[\Big(P - {\mit \Pi} (\beta(\lambda)) \Big)^\top \Big(P - {\mit \Pi}(\beta (\lambda)) \Big) + \lambda\, \beta(\lambda)^\top H \beta (\lambda) \right].$$

In general, the minimizer is found by nonlinear least squares as described in the previous section, iterating on

$$\beta^{i+1}(\lambda) = \Big(X(\beta^i(\lambda))^\top X(\beta^i(\lambda)) + \lambda H \Big)^{-1} X (\beta^i (\lambda))^\top Y (\beta^i (\lambda))$$

until convergence:

$$\beta^*(\lambda) = \Big(X(\beta^*(\lambda))^\top X(\beta^*(\lambda)) + \lambda H \Big)^{-1} X(\beta^*(\lambda))^\top Y(\beta^*(\lambda)).$$

(Note that when splining $\delta(\tau)$, $\beta^*(\lambda) = (X^\top X + \lambda H)^{-1} X^\top P.$)

Formally, the previous expression is ridge regression estimator. By employing a roughness penalty, we can over-parameterize the spline, making $X(\beta^*(\lambda))^\top X(\beta^*(\lambda))$ nearly singular, and use the penalty to reduce the effective number of parameters. The penalty thus ``solves'' the multicollinearity problem. The advantage of this technique is that the shape of the spline is controlled by a single parameter, λ.

One common measure of the effective number of parameters is the trace of $A(\lambda)$, denoted ${\rm tr}(A(\lambda))$, where

$$A(\lambda) := X(\beta^*(\lambda))\Big(X(\beta^*(\lambda))^\top X(\beta^*(\lambda)) + \lambda H \Big)^{-1} X(\beta^*(\lambda))^\top.$$

Note that $A(\lambda)\,Y(\beta^*(\lambda))$ is the vector of fitted Y values which in the linear case is the vetcor of fitted prices. The extreme cases are ${\rm tr}(A(0)) = \kappa$ (with no penalty the number of effective parameters equals the number of B-splines), and ${\rm tr}(A(\infty)) = 2$ (with an infinite penalty the number of effective parameters equals 2).

Generalized cross validation

In this section, we present a technique for choosing the appropriate value for λ. We choose the value of λ that minimizes the ``generalized cross validation'' (GCV) value,

$$\gamma(\lambda) := {\Big((I - A(\lambda))Y (\beta^*(\lambda))\Big)^\top \Big((I - A (\lambda))) Y(\beta^*(\lambda)) \Big) \over \Big(n - \theta\,{\rm tr}(A(\lambda))\Big)^2}.$$

The numerator of the previous expression is the residual sum of squares. When $\theta = 1$, the denominator is the squared effective degrees of freedom (the difference between the number of observations and the effective number of parameters). The parameter θ is called the cost. It controls the trade-off between goodness-of-fit and parsimony. In plain-vanilla GCV, $\theta=1$. However, θ can be increased to reduce the signal extracted, thereby stiffening the spline. (In the code, θ is called the **`TuningParameter`**, which is an option to **`DiscountFunction`**.)

When the discount function is splined directly, $A(\lambda) = X(X^\top X + \lambda H)^{-1} X^\top$ and there is a simplified expression for $\gamma(\lambda)$ that can be minimized directly. In general, however, a new $X(\beta^*(\lambda))$ matrix must be formed for each value of λ. Thus for each value of λ we test, we must solve for $\beta^*(\lambda)$ and then calculate $\gamma(\lambda)$. The overall solution is given by

$$\beta^*(\lambda^*) = \Big (X (\beta^* (\lambda^*))^\top X (\beta^* (\lambda^*)) + \lambda H \Big)^{-1} X (\beta^*(\lambda^*))^\top Y(\beta^*(\lambda^*)),$$

where λ^* minimizes $\gamma(\lambda)$.

Implementing the Estimators

For smoothing splines, we need starting values for λ. However, it is not easy to know in advance what a good starting value is. At extreme values of λ (both large and small) the GCV function becomes flat (see below) and optimizers can get stuck at non-minimums. Experimenting with differnt values is important to find out where good starting values are for a given data set. We have found that starting near 10^{10} usually (but not always) converges to the true global minimu. In addition, one may need to ``tune'' the **FindMinimum** routine. For example, one may need to boost the accuracy goal; *e.g.,* **AccuracyGoal -> 10**.

When splining either $\ell(\tau)$ or $f(\tau)$, we need starting values for β as well. The fixed point problem will not converge with bad starting values. Fortunately, good starting values for β are easy to calculate. One of the properties of B-splines is that $\sum_{k=1}^\kappa \phi(\tau) = 1$. As a consequence, the coefficients, β, track the value of the function, $\phi(\tau)\,\beta$. Thus any reasonable estimate of the function to be splined can be used to form starting values. For example, suppose a crude estimate of the function to be splined is $\widehat h(\tau)$. For a cubic spline, let the starting value for β_k be $\beta_{k0} = {1 \over 3} \sum_{i=k}^{k+2} \widehat h(d_i)$. With these starting values, the fixed point problem converges rapidly. The internal default for the yield curve is a flat 5 percent curve. If **DLStart.m** or **LStart.m** is used, the yield curve stored in there is used.

Usage Overview

A typical session will include reading data, calculating when the coupon payments occur, estimating the yield curve. A session may include calculating yields and yield errors and producing graphs and reports.

We start by reading some data:

```
md = ReadData["19880719.dat"]
```

```
-MD[{1988, 7, 19}]-
```

ReadData reads the data in **19880719.dat** and creates an **MD** object, which in this case is named **md** (for future reference). The **MD** object contains information about the bonds found in the input file. **ReadData** requires the data be in a particular format, which we can see as follows:

```
First @ ReadList["19880719.dat", String]
```

```
7/19/1988,912794PY4,8/4/1988,0.5,0,99.7321
```

The order of fields is quote date, I.D. number, maturity date, original term to maturity (in years), coupon rate, and price per \$100 of face value (not including accrued interest for coupon bonds).

Let's see what options **ReadData** takes:

```
Options[ReadData]
```

```
{Description -> {}}
```

One can add a description (a string or list of strings) to help keep track of different **MD** objects.

A list of information the object contains can be had by entering **MDSelectors**. The selectors can be used to extract information the bonds included:

```
MDSelectors
```

```
{CouponRate, Description, IDNumber, MaturityDate,
  NumberOfSecurities, Price, QuoteDate, SettlementDate,
  Term}
```

Here are two examples:

```
NumberOfSecurities[md]
CouponRate[md] //Short
```

```
224
```

```
{0, 0, 0, 0, 0, 0, 0, 0, <<212>>, 8.75, 8, 8.25, 8.875}
```

The next step is to construct all the payments (c_i) and the number of days to each payments (τ_i) for each of the bonds in the sample. **ConstructPayments** does this. First, let's see what the options are:

```
Options[ConstructPayments]
```

```
{FullPrice -> False, PaymentsPerYear -> 2}
```

These are the appropriate options for our data: U.S Treasury coupon securities pay coupons every six months, and our coupon-bond price data does not include accrued interest (which therefore must be added by **ConstructPayments**). Neither of these options has any effect on bills or STRIPs.

```
cp = ConstructPayments[md]
```

```
-CP[{1988, 7, 19}]-
```

ConstructPayments calculates the number of days to each coupon payment for each bond in the sample, using information in **daycount.m**. It takes an **MD** object as an argument and returns a **CP** object that contains a reference to the **MD** object used to create it. Thus, **CouponRate[cp]** will return the same list of coupon rates for the bonds in the sample. **CPSelectors** is a list of information stored in **CP** objects.

```
CPSelectors
```

```
{AccruedDays, AccruedInterest, DaysInPeriod,
  DaysToPayments, FullPriceQ, LastPayment, NextPayment,
  Payments, PaymentsPerYear, QuoteDate, RemainingPayments}
```

```
DaysToPayments[cp] //Shallow
Payments[cp] //Shallow
CouponRate[cp] //Shallow
```

```
{{15}, {43}, {71}, {99}, {128}, {155}, {1}, {8}, {22},
  {29}, <<214>>}
```

```
{{100}, {100}, {100}, {100}, {100}, {100}, {100}, {100},
  {100}, {100}, <<214>>}
```

```
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, <<214>>}
```

We see that each of the first 10 securities is a Treasury bill with one remaining payment of 100.

We are now ready to fit the term structure. **DiscountFunction** is the main estimation function. It takes a **CP** object as an argument and returns a **DF** object. The default functional form is **DLogDelta**, which indicates that the forward rates should be splined. The default **LambdaValue** is **Automatic**, which indicates the generalized cross validation should be used to determine the value of λ, the weight on the penalty. The **DF** object contains the results of the estimation, including for example the estimated spline (**FinalSpline[df]**) and the difference between the actual and fitted bond prices (**PriceError[df]**).

First look at the options available for **DiscountFunction**:

```
Options[DiscountFunction]
```

```
{BetaConvergenceTest -> (Max[Abs[(#2 - #1)/#2]] < 0.0001 & ),

  Bounds -> {0., 1. 10^30}, DropByTerm -> 0,
  FunctionalForm -> DLogDelta, IDNumberDropList -> {},
  InitialCurve -> Automatic, Knots -> Automatic,
  LambdaValue -> Automatic, Lambda0 -> Automatic,
  MaximumMaturity -> 30, ObservationWeights -> Automatic,
  OvernightForwardRate -> Automatic, Restriction -> True,
  ShowLambdaProgress -> True, SplineOrder -> 3,
  TuningParameter -> 2}
```

In addition to the options listed, one can pass options for **FindMinimum** (such as **AccuracyGoal -> 10**).

It's always a good idea to read the usage statement.

```
?DiscountFunction
```

```
DiscountFunction[cp] estimates a discount function for a
   given day.  Run MassageData and ConstructPayments first
   to create the CP object. The default options are Knots ->
   (number of securities)/3, DropByTerm -> 0 (drop the 0
   most recently issued securities of each term),
   MaximumMaturity -> 30. The default FunctionalForm is
   DLogDelta, which uses a nonlinear routine to estimate the
   forward rate function; other functional forms are
   LogDelta and Delta. If DLogDelta or LogDelta is used,
   DiscountFunction takes options to set the starting values
   (Lambda0, Bounds, and options to tune FindMinimum) and
   looks for a file named "LStart.m" or "DLStart.m" to find
   starting values for beta0 and lambda0.  If the
   appropriate file cannot be opened, internal default
   values are used.
```

```
df1 = DiscountFunction[cp, FunctionalForm -> Delta,
    LambdaValue -> 10^10]
```

```
-DF[{1988, 7, 19}]-
```

Setting **`LambdaValue -> 10^10`** fixes the value of λ without minimizing the GCV function. The **`DFSelectors`** can be used to extract information from the **`DF`** object:

```
DFSelectors
```

```
{AverageAbsoluteError, BetaHat, BetaHatCovarianceMatrix,
  Delta, DropByTerm, EffectiveParameters, FinalSpline,
  FixedLambdaQ, ForwardCurve, GCVMinimizedQ, GCVValue,
  KeepList, Knots, Lambda, MaximumMaturity,
  NumberOfObservations, ObservationWeights, PredictedPrice,
  PriceError, QuoteDate, ResidualVariance, RestrictionQ,
  SemiAnnualForwardCurve, SemiAnnualZeroCurve,
  Significance, SplineOrder, TuningParameter, ZeroCurve}
```

For example, we can find the average absolute pricing error:

```
AverageAbsoluteError[df1]
```

```
0.82538
```

This is an error of 82.5 basis points per \$100 of face value. This is not very good, and we will see why in a moment.

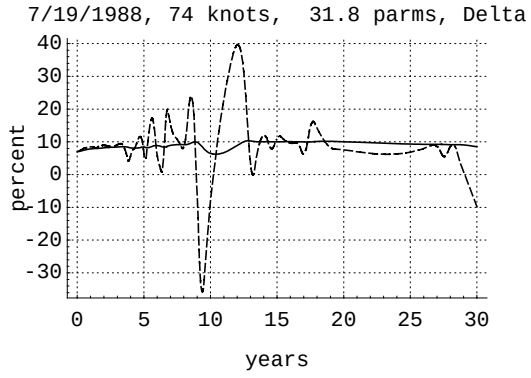
There are other functions that take DF objects as arguments as well. For example,

```
DFSummaryStatistics[df1]
```

```
no. of obs.........   224
longest payment....    29.82 years
no. dropped........     0
functional form....  Delta
restriction........  True
minimize GCV.......  False
tuning parameter...     2
no. of knots.......    74
no. of parameters..    31.8
lambda.............     1.00e10
residual variance..     7.53003
avg. abs. error....     0.8254
```

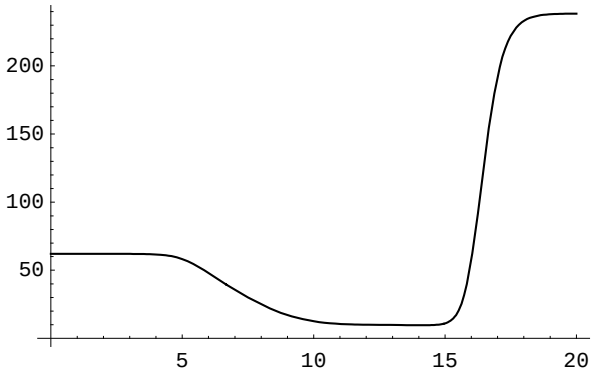
We see that the number of knot points (chosen automatically) is 74, but with the the penalty we used, $\lambda = 10^{10}$, the effective number of parameters is only 31.8. Here are the forward rate (dashing) and zero-coupon rate curves (If you have a color screen, use can use the option **`Color -> True`** in ShowGraph):\footnote{^{6}}{Note that both ShowGraph and DFSummaryStatistics can be combined with DFSummaryPage, which is especially useful for sending reports to a printer.}

```
ShowGraph[df1, PlotStyle -> {Dashing[{.02, .01}],{}}];
```



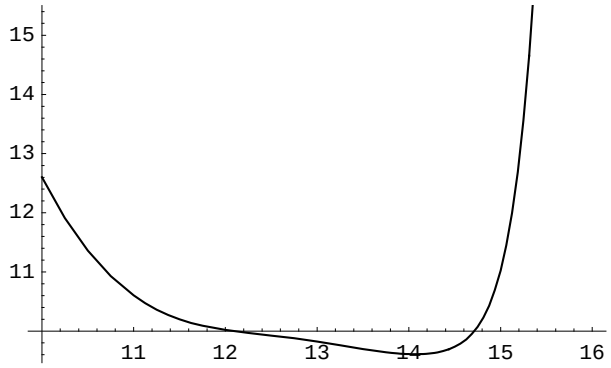
This looks bad---the term structure is over-parameterized. Let's examine the generalized cross validation function:

```
Plot[GCVFunction[10^x], {x, 0, 20}];
```



Notice the fuction is flat at both ends. An adaptive optimizer (such as FindMinimum) can get stuck in these flat regions. It looks like the minimum is between 10^{10} and 10^{16}.

```
Plot[GCVFunction[10^x], {x, 10, 16}];
```



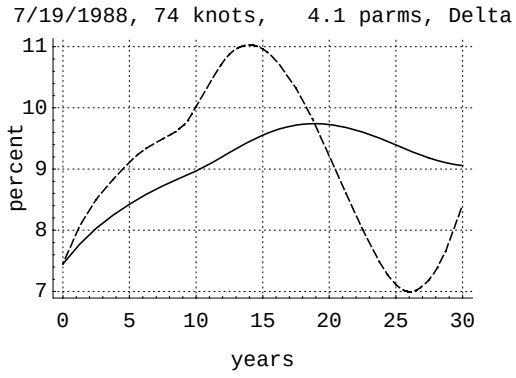
The minimum appears to be close to 10^{14}. Let's use that as a starting value to applay GCV. Note that a pair of starting values is required.

```
df2 = DiscountFunction[cp, FunctionalForm -> Delta,
    Lambda0 -> {10^14, 1.1 10^14}];
{Lambda[df2], EffectiveParameters[df2],
    AverageAbsoluteError[df2]}
```

```
{1.16709 10^14, 4.14526, 0.919242}
```

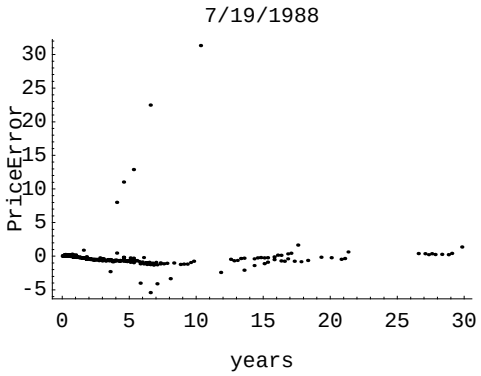
We see that λ^* is close to our guess, and that the number of parameters has fallen to 4.1 while the average absolute error has risen a bit to 92 basis points. Let's look at the yield curves:

```
ShowGraph[df2, PlotStyle -> {Dashing[{.02, .01}],{}}];
```



These curves are much more well-behaved. But recall that the average pricing error is large (for Treasury bonds). Let's look at the pricing errors:

```
MaturityPlot[PriceError[df2]];
```

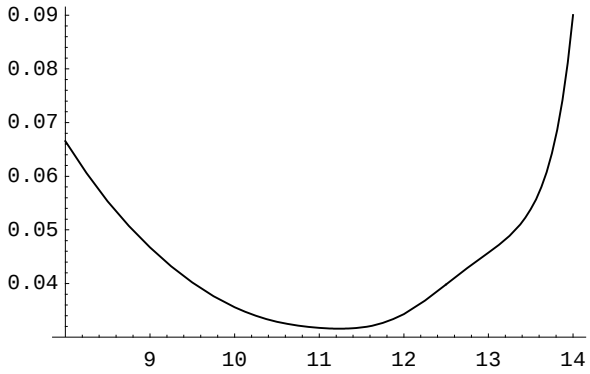


There are a few enormous outliers. What is wrong with these securities? The answer is that we have included some callable bonds and some flower bonds in the sample. The prices of these bonds reflect features that are not captured in the present value function directly. Including them in the estimation will affect the results in two ways. First, their inclusion will affect the value of $\beta^* (\lambda^*)$, the estimated spline coefficients, conditional on the value of λ. But perhaps more importantly, their inclusion will affect the value of λ^* chosen via GCV. This is because minimizing the GCV function is a signal-extraction technique: When there is more noise in the data, a larger penalty will be chosen for a smoother curve at the expense of the sum-of-squares fit.

The solution to this problem is to remove the securities with special features from the estimation. We can do that with **`IDNumberDropList`**, an option to **`DiscountFunction`**. The file **`iddrop.m`** contains a list of CUSIPs of the securities with special features.

```
iddrop = << iddrop.m;
df3 = DiscountFunction[cp, FunctionalForm -> Delta,
    LambdaValue -> 10^10, IDNumberDropList -> iddrop];
```

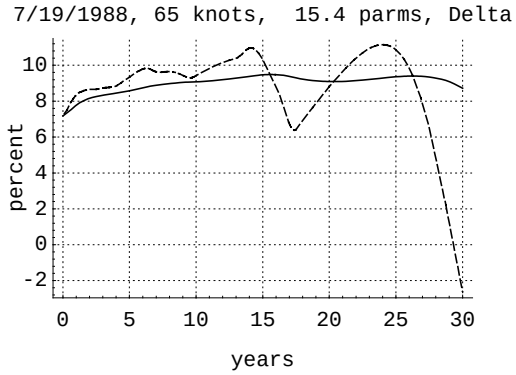
```
Plot[GCVFunction[10^x],{x, 8, 14}];
```



Now the minimum appears to be between 10^{11} and 10^{12}.

```
df4 = DiscountFunction[cp, FunctionalForm -> Delta,
    Lambda0 -> {10^11.5, 1.1 10^11.5},
    IDNumberDropList -> iddrop];
DFSummaryStatistics[df4]
ShowGraph[df4, PlotStyle -> {Dashing[{.02, .01}],{}}];
```

```
no. of obs.........   196
longest payment....    29.82 years
no. dropped........     0
functional form....  Delta
restriction........  True
minimize GCV.......  True
tuning parameter...     2
no. of knots.......    65
no. of parameters..    15.4
lambda.............     1.74e11
residual variance..     0.02436
avg. abs. error....     0.0990
```

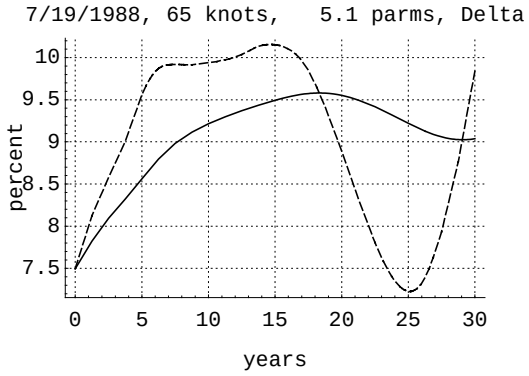


The number of effective parameters is 15.4 and the average absolute error is down to about 10 basis points. This is quite an improvement from the previous fit.

It turns out that there is another class of securities that tends to have another additional feature that is not captured in the present value of the the stated payments. The one or two most-recently-issued securities of a given original term-to-maturity are often ``on special'' in the repo market. As such, an additional stream of payments can be obtained in the financing market by the holder of the security. We can screen out the most recently issued securities by term with another option to **`DiscountFunction`**:

```
df5 = DiscountFunction[cp, FunctionalForm -> Delta,
    DropByTerm -> 2,
    Lambda0 -> {10^11.5, 1.1 10^11.5},
    IDNumberDropList -> iddrop];
DFSummaryStatistics[df5]
ShowGraph[df5, PlotStyle -> {Dashing[{.02, .01}],{}}];
```

```
no. of obs.........   197
longest payment....    29.82 years
no. dropped........     2
functional form....  Delta
restriction........  True
minimize GCV.......  True
tuning parameter...     2
no. of knots.......    65
no. of parameters..     5.1
lambda.............     3.34e13
residual variance..     1.36962
avg. abs. error....     0.4149
```



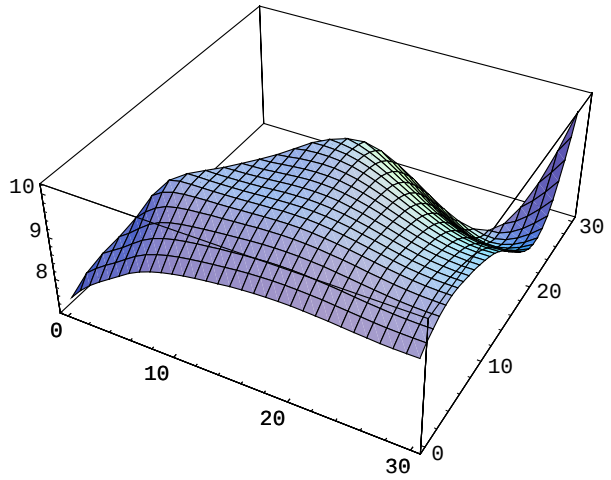
The pricing errors are a bit smaller. And although the number of knots chosen was 7 fewer, the number of effective parameters dropped by less than 4.

Let's use this discount function for an illustrative digression (even though it could probably be improved on by using a differenct functional form). Consider a coupon bond with a face value of \$1 that pays its coupon as a continuous stream--an instantaneous coupon bond. Let this bond mature at time τ. The (instantaneous) swap rate is the coupon rate that makes the value of this bond equal \$1 today. The (instantaneous) forward swap rate is the coupon rate that makes the forward value of the bond equal the forward value of \$1 at time t'. Here is the formula for the forward swap rate:

$$s(t',\tau) = {\delta(t') - \delta(\tau) \over \int_{t'}^\tau \delta(u)\,du}$$

We can plot the instantaneous forward swap surface as follows. First extract $\delta(\tau)$ from the **DF** object (otherwise it the plotting routine will run much slower) and then run **PlotSwapSurface**:

```
delta = Delta[df5];
PlotSwapSurface[delta];
```



The spot swap curve (also known as the par coupon yield curve) is on the nearest face, while the forward rate curve runs along the diagonal. **PlotSwapSurface** calls **TriangularPlot3D**, a package that is included in the distribution along with **YieldCurve**.

Now let's calculate the actual and fitted yields. First consider how to compute the compouning. The default uses the Securities Industry Association Standard Securities Calculation Methods.

```
Options[YieldCalc]
?Compounding
```

```
{Compounding -> SIACompounding}

Compounding is an option for YieldCalc that specifies how to
   handle the compounding of yields. The Default is
   Compounding -> SIACompounding, which specifies the use of
   the Securities Industry Association formulas. Currently
   this handles zeros incorrectly. The other valid setting
   is Compounding -> ContinuousCompounding, which specifies
   the use of continuous compounding for bills and zeros.
   Compounding is also a YCSelector. It returns the setting
   used for the constructions of the YC object.
```

SIACompounding is useful for comparing with yields quoted in the newpaper. On the other hand, to compare the yields on bills or strips to the continuously compounded **ZeroCurve**, it is useful to use **Compounding -> ContinuousCompounding**. (As an alternative, one can use **SemiAnnualZeroCurve** instead.)

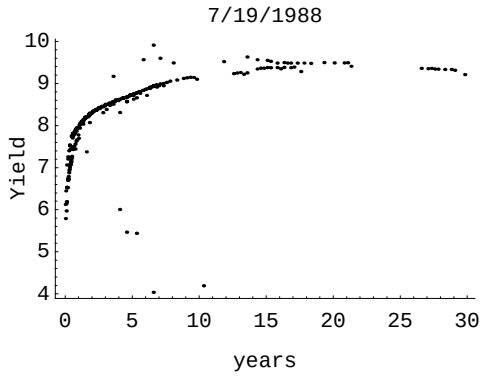
Now let's calculate the actual and fitted yields.

```
yc = YieldCalc[df5]
```

```
-YC[{1988, 7, 19}]-
```

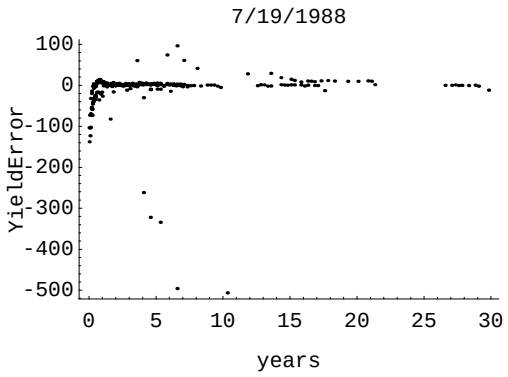
YieldCalc calculates yield to maturity for each of the bonds in the sample (**Yield[yc]**) and the difference between that yield and the predicted yield (**YieldError[yc]**), among other things. **YieldCalc** can be run on a **CP** object, in which case it will not attempt to calculate predicted yields.

```
MaturityPlot[Yield[yc]];
```



Although yields are ploted in percent, yield errors are ploted in basis points:

```
MaturityPlot[YieldError[yc]];
```



As we see, the outliers compress the image of the rest of the data. If one is not interested in pricing the securities with special features, one can omit them from the data file or create another **MD** object that excludes them (see below).

So far we have restricted ourselves to splining the discount function directly, largely for speed considerations. With a large number of securities, the nonlinear routines run quite slowly. So to speed things up, we will spline only Treasury bills.

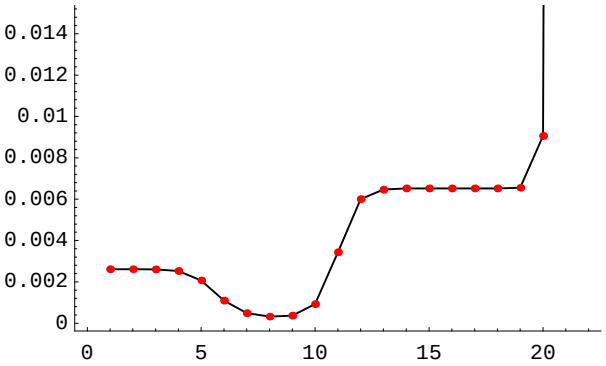
```
pos = Flatten @ Position[CouponRate[md], 0|0.];
mdbills = MDSubset[md, pos];
cpbills = ConstructPayments[mdbills];
```

For comparison purposes, we will spline the discount function first with no penalty:

```
dfbills1 = DiscountFunction[cpbills,
    FunctionalForm -> Delta, MaximumMaturity -> 1,
    LambdaValue -> 0]
```

```
-DF[{1988, 7, 19}]-
```

```
PlotGCVFunction[];
```




```
dfbills2 = DiscountFunction[cpbills,
    FunctionalForm -> Delta, MaximumMaturity -> 1,
    Lambda0 -> {10^8, 10^9}];
```

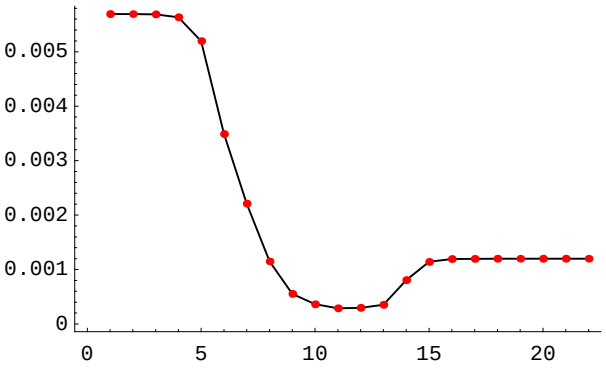
```
-DF[{1988, 7, 19}]-
```

```
dfbills3 = DiscountFunction[cpbills, DropNumber -> 0,
    FunctionalForm -> DLogDelta, MaximumMaturity -> 1,
    LambdaValue -> 0];
```

```
lambdastar = 0
```

```
-DF[{1988, 7, 19}]-
```

```
PlotGCVFunction[];
```




```
dfbills4 = DiscountFunction[cpbills,
    FunctionalForm -> DLogDelta, MaximumMaturity -> 1,
    Lambda0 -> {10^11, 10^12}];
```

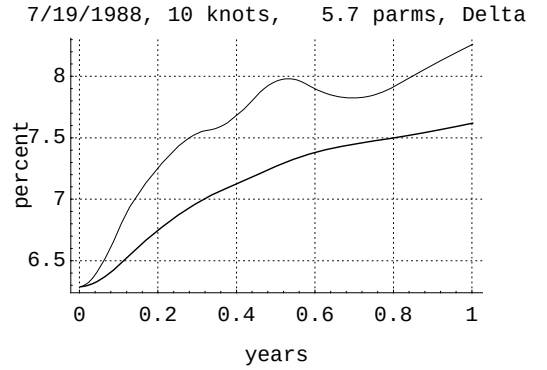
```
lambda = 1.*10^11
lambda = 8.*10^10
lambda = 1.12360679775*10^11
lambda = 1.11014526840039*10^11
lambda = 1.200000000000047*10^11
lambda = 1.076393202250034*10^11
lambda = 1.152786404500073*10^11
                        11
lambdastar = 1.15279 10
```

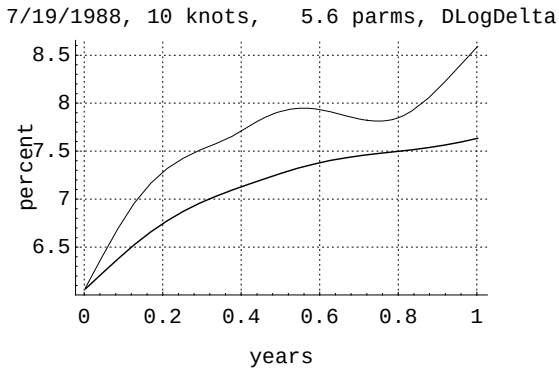
```
-DF[{1988, 7, 19}]-
```

Note that when the functional form is either **LogDelta** or **DLogDelta**, the option **ShowLambdaProgress** controls whether the adaptive seach routine reports its progess or not. For long calculations involving many securities and many knot points, it is useful to see how λ evolved and where it ended if the the calculation was ultimately aborted by the user or if it bombed on its own.

Now let's compare the graphs of the two estimation techniques:

```
ShowGraph[dfbills2];
ShowGraph[dfbills4];
```





Very similar looking results in this case. In other cases, the results may differ substantially.

Two ways to use the program

■ Interactive

The program was designed to facilitate an interactive examination of how well different splining methods fit the term structure. To that end, we have built in flexibility as to the number and placement of knot points, the functional form to be splined, and the size of the penalty and technique of choosing it. The way in which we have stored the output at each stage (in self-contained, insulated objects) allows the user to have multiple estimates of the term structure available at once, making comparisons easy. For example, one can run **DiscountFunction** with different settings on the same data as we have done above. All of this is easily accomplished by giving different objects different names.

■ Estimation loop

Some users may wish to run the same settings on many different days of data and save some of the results to a file. Here are some suggestions as to how to do that. In a separate subdirectory (for convenience) create your data files, one per day. Name them in such a way that they are in calendar order (for convenience). Use **SetDirectory** to change the directory. Use **FileNames** to get a list of the file names. Write a function that (*i*) estimates the term structure according to the settings you have chosen and (*ii*) extracts the information you want and appends it to a file. Map that function onto the list of filenames.

Here is an example of such a function. It takes two arguments, an input file name (where to find the data) and an output file name (where to append the results).

```
EstimationLoop[in_String, out_String] :=
  Module[{md, cp, df, qd, zc, fc, zeros, forwards},
  (* estimate the term structure *)
  md = ReadData[in]; (* or use a version of MassageData *)
  cp = ConstructPayments[md];
  df = DiscountFunction[cp]; (* use your settings *)
  (* extract the zero and forward curves *)
  zc = ZeroCurve[df];
  fc = ForwardCurve[df];
  (* read off zero and forward rates every 6 months *)
  zeros = Table[{m, zc[m Years]}, {m, .5, 30, .5}];
  forwards = Table[{m, fc[m Years]}, {m, 0, 30, .5}];
  (* get the instantaneous zero rate from the forward curve *)
  PrependTo[zeros, First @ forwards];
  (* prepare data for file *)
  qd = DateToString @ QuoteDate[md];
  zeros = Join[{qd, "z"}, #]& /@ zeros;
  forwards = Join[{qd, "f"}, #]& /@ forwards;
  (* append data to output file *)
  AppendDelimited[out, #]& /@ {zeros, forwards}]
```

Here is the usage. **EsitmationLoop[#, "output.dat"]& /@ fn**, where **fn** is the list of input data file names. A typical row in **output.dat** will look like:

```
"9/29/1993","z",7.5,5.21428
```

This file can be read back into *Mathematica* using

```
data = ToExpression @ ReadDelimited["output.dat"];
```

The dates can be converted to **{yyyy, mm, dd}** format with **StringToDate**. The file can also be searched using **FindList**. (**StringToDate**, **DateToString**, **AppendDelimited** and **WriteDelimited** are defined the package **CommaDelimited**, which is distributed with **YieldCurve**.)

The structure of MD, CP, and DF objects

In this section we describe the structure of **MD**, **CP**, and **DF** objects. These objects are merely ``wrappers'' that contain (and hide) the data and results. This section will serve two purposes. First, it will help those who wish to write their own version of **ReadData** (or its older cousin, **MassageData**), and second it describes the overall structure of the **YieldCurve** package.

■ MD objects

The structure of the flat file can be seen as follows. We select a few rows from the file:

```
pos = {1, 30, 60, -1};
ReadList["19880719.dat", String][[ pos ]]
```

```
{7/19/1988,912794PY4,8/4/1988,0.5,0,99.7321,
  7/19/1988,912794SC9,6/8/1989,0.5,0,93.5669,
  7/19/1988,912810CM8,2/15/2005,25,11.75,118.719,
  7/19/1988,912827WK4,7/15/1995,7,8.875,99.625}
```

The fields in the file are quote date, identification number, maturity date, original term to maturity (in years), coupon rate, and price per hundred dollars of face value (in decimal form). The price does not include accrued interest for coupon bonds. Fields are separated by commas with no intervening spaces. The identification number and the term are both required, but neither need be meaningful nor unique. (In fact, in the data above, the term for all bills is set to 0.5, which is not true since some bills have an original term to maturity of 1 year.) The identification number is useful for **IDNumber-DropList** and the term is useful for **DropByTerm**, both options of **DiscountFunction**.

We can easily create a mini-**MD** object of these five securities to see what it looks like internally:

```
?MDSubset
```

```
MDSubset[md, poslist] creates an MD object that contains a
   subset of bonds from another MD object according to the
   list of positions given.
```

```
mdsub = MDSubset[md, pos];
InputForm[mdsub]
```

```
MD[{1988, 7, 19}, {{1988, 8, 4}, {1989, 6, 8},
   {2005, 2, 15}, {1995, 7, 15}}, {0.5, 0.5, 25, 7},
  {0, 0, 11.75, 8.875}, {99.7321, 93.5669, 118.719,
   99.625}, {"912794PY4", "912794SC9", "912810CM8", "91282\
    7WK4"}, {}]
```

The **MD** object contains seven items, each of which is accessible via an **MDSelector**: **QuoteDate**, **MaturityDate**, **Term**, **CouponRate**, **Price**, **IDNumber**, and **Description**. An additional **MDSelector**, **SettlementDate** is not contained in the **MD** object; instead, it is calculated on demand (see below).

The user may write a different **ReadData** function to read data in a different form. What is required for the program to work properly is to have the new **ReadData** production an **MD** object with the correct internal structure. Here is the relevant **Mathematica** code the sets up the **MDSelectors** and formats the **MD** objects:

```
MD /: QuoteDate[md_MD]      := md[[1]]
MD /: MaturityDate[md_MD]   := md[[2]]
MD /: Term[md_MD]           := md[[3]]
MD /: CouponRate[md_MD]     := md[[4]]
MD /: Price[md_MD]          := md[[5]]
MD /: IDNumber[md_MD]          := md[[6]]
MD /: Description[md_MD]    := md[[7]]
MD /: NumberOfSecurities[md_MD] := Length[Price[md]]
MD /: SettlementDate[md_MD] := FirstBusinessDay @
     DaysPlus[QuoteDate[md], BusinessDaysToSettlement]
MD /: WhenceMD[md_MD]       := md

Format[md_MD] := StringForm["-MD[`1`]-", QuoteDate[md]]

MDSelectors = Sort @ {QuoteDate, MaturityDate, Term,
    CouponRate, Price, IDNumber, Description,
    SettlementDate}
```

Here is an earlier version of **ReadData**, **MassageData**, that uses symbolic switches to identify the structure of the data file.

MassageData[OpenMarket] reads the file **openmkt.dat**. That file has a different structure. In particular, security price quotes are given; bills are quoted as banker's discounts and coupon securities are quoted in 32nds.

```
First @ ReadList["openmkt.dat",String]
```

```
{1993,10,27} 912810DP0 {2015,2,15} 11.250 10957 159.13
```

```
mdopen = MassageData[OpenMarket]
```

```
-MD[{1993, 10, 27}]-
```

MassageData[Strips] looks for the flat file **strips.dat** in the working directory and assumes it has the following structure:

```
Take[ReadList["strips.dat", String], 3]
```

```
{9/30/1994,11/15/1994,99.46875,
  9/30/1994,11/15/1994,99.46875,
  9/30/1994,2/15/1995,98.09375}
```

Note the following aspects of structure of the data file. There are only three fields per record: quote date, maturity date, and price as a decimal. The missing information, which is not necessary to run **DiscountFunction**, is simply ``made up'' by **MassageData**.
In this case, the terms-to-maturity are arbitrarily set to **-1** and the IDNumbers are set to **"N/A"**. The **Description** is set to **{"Strips"}**. The term of **-1** is used by **ConstructPayments** to distinguish strips from bills for the purposes of the Security Industry Associations standard securities yield calculations, which treat bills and strips separately.\footnote{^{7}}{Bills and strips are treated the same when yields are calculated continuously.}

If one wanted to distinguish between principal and coupon STRIPs, one could add a fourth column to **strips.dat**: an **IDNumber** that took on two values, say **"P"** for principal and **"C"** for coupon. One would write a new version of **MassageData** (or **ReadData**) to put this **IDNumber** in the proper place in the **MD** object. Then one could use the option **IDNumberDropList -> "P"** to specify that only principal should be used in the estimation.

Note that STRIPs data can be read correctly with **ReadData**, and STRIPs data can be mixed with other security types as well.

■ CP objects

Here is the **InputForm** of the **CP** object created by **ConstructPayments** from **mdsub**:

```
cpsub = ConstructPayments[mdsub];
InputForm[cpsub]
```

```
CP[{1988, 7, 19}, {{100}, {100},
   {5.875, 5.875, 5.875, 5.875, 5.875, 5.875, 5.875,
    5.875, 5.875, 5.875, 5.875, 5.875, 5.875, 5.875,
    5.875, 5.875, 5.875, 5.875, 5.875, 5.875, 5.875,
    5.875, 5.875, 5.875, 5.875, 5.875, 5.875, 5.875,
    5.875, 5.875, 5.875, 5.875, 5.875, 105.875},
   {4.4375, 4.4375, 4.4375, 4.4375, 4.4375, 4.4375,
    4.4375, 4.4375, 4.4375, 4.4375, 4.4375, 4.4375,
    4.4375, 104.4375}}, {{15}, {323},
   {26, 210, 391, 575, 756, 940, 1121, 1308, 1489, 1672,
    1853, 2036, 2217, 2401, 2582, 2766, 2948, 3135, 3313,
    3499, 3680, 3863, 4044, 4227, 4409, 4593, 4774, 4958,
    5139, 5326, 5504, 5690, 5871, 6054},
   {181, 362, 545, 726, 909, 1090, 1274, 1456, 1640, 1821,
    2008, 2186, 2372, 2553}}, {0, 0, 155, 5}, {2, 2, 2, 2},
  False, Hold[mdsub]]
```

There are seven items contained in the object, all of which can be extracted (*i.e.*, selected) by the **CPSelectors**: **QuoteDate**, **Payments**, **DaysToPayments**, **AccruedDays**, **PaymentsPerYear**, **FullPrice**, and **WhenceMD**. The other **CPSelectors** calculate what they return on demand.

■ DF objects

Here is the **InputForm** of the **DF** object created by **DiscountFunction** from the **CP** object given above:

```
dfsub = DiscountFunction[cpsub, MaximumMaturity -> 16.6,
    Knots -> 0, FunctionalForm -> Delta, LambdaValue -> 0]
```

```
-DF[{1988, 7, 19}]-
```

```
InputForm[dfsub]
```

```
DF[{1988, 7, 19}, InterpolatingFunction[{0, 6064},
   {{0, 0, {1, 0}, {0}},
    {6064, 0, {0.2117929536117513,
      -0.00001535208454085584, 1.890324679160249*10^-8,
      1.195289841813016*10^-12}, {0, 0, 6064}}}], 0, 3,
  Delta, {0.03298646745656697, -0.001870960109926045,
   -(2.131225215862287*10^-6), 0.00003979571182810559},
  {1, 2, 3, 4}, 0.001091609115236887, 0.00872483862588425,
  {0.996796948422626, -0.04007415733109442,
   -0.00004561217393878103, 0.000851702378932987},
  {0, 6064}, {{0, 0, 0, 0},
   {0, 0.005933603944256982, -0.01007417955412868,
    0.001370051508054565},
   {0, -0.01007417955412868, 0.01787216815776105,
    -0.002587323349510649},
   {0, 0.001370051508054565, -0.002587323349510649,
    0.0004489482759882174}},
  {1, 0.5944046122549978, 0.2428246338303346,
   0.2117929536117513}, True, 0, True, 2, Automatic, 3,
  0.001091609116883774, Hold[cpsub]]
```

There are nineteen items contained in the object, all of which can be extracted (*i.e.* selected) by the **DFSelectors**: **QuoteDate**, **FinalSpline**, **Lambda**, **EffectiveParameters**, **FunctionalForm**, **PriceError**, **KeepList**, **ResidualVariance**, **AverageAbsoluteError**, **Significance**, **Knots**, **BetaHatCovarianceMatrix**, **BetaHat**, **Restriction**, **DropByTerm**, **FixedLambda**, **TuningParameter**, **ObservationWeights**, **Spline-Order**, **GCVValue**, and **WhenceCP**. The remaining **DFSelectors** calculate what they return on demand.

References

Adams, K. J., and D. R. Van Deventer. ``Fitting Yield Curves and Forward Rate Curves with Maximum Smoothness." *Journal of Fixed Income*, June (1994): 52--62.

Bliss, R. R., Jr. ``Testing Term Structure Estimation Methods." Working Paper, Indiana University (1993).

Chambers, D., W. Carleton and D. Waldman. ``A New Approach to Estimation of Term Structure of Interest Rates." *Journal of Financial and Quantitative Studies*, 19 No. 3 (1984): 233--252.

Chow, G. C. *Econometrics* McGraw-Hill, New York (1983).

Coleman, T. S., L. Fisher and R. Ibbotson. ``Estimating the Term Structure of Interest From Data That Include the Prices of Coupon Bonds." *The Journal of Fixed Income*, September (1992): 85--116.

de Boor, C., *A Practical Guide to Splines*, Springer-Verlag (1978).

Fisher, M., D. Nychka, and D. Zervos. ``Fitting the Term Structure of Interest Rates with Smoothing Splines," Finance and Economics Discussion Series, 95-1, Federal Reserve Board (1995).

Gilles, C. ``Forward Rates and Expected Future Short Rates." Working paper, Federal Reserve Board (1994).

Jordan, J.V. ``Tax Effects in Term Structure Estimation." *Journal of Finance*, 39 No. 2 (1984): 393--406.

McCulloch, J. H. ``Measuring the Term Structure of Interest Rates.' '*Journal of Business*, 44 (1971): 19--31.

McCulloch, J. H. ``The Tax-Adjusted Yield Curve." *Journal of Finance*, 30 (1975): 811--830.

Nelson, C. and A. Siegel. ``Parsimonious Modeling of Yield Curves." *Journal of Business*, 60 No. 4 (1987): 473--489.

Shea, G. ``Pitfalls in Smoothing Interest Rate Term Structure Data: Equilibrium Models and Spline Approximations." *Journal of Financial and Quantitative Studies*, 19 No. 3 (1984): 253--269.

Shea, G. ``Interest Rate Term Structure Estimation with Exponential Splines: A Note." *Journal of Finance*, 40 No. 1 (1985): 319--325.

Vasicek, O. and G. Fong. ``Term Structure Estimation Using Exponential Splines." *Journal of Finance*, 38 (1982): 339--348.

Wahba, G. *Spline Models for Observational Data* SIAM: Philadelphia (1990).