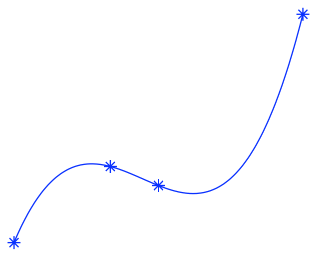
Theme 5: Useful items in the numerical toolbox: Interpolation and Quadrature

Martin Berggren

December 8, 2010

Interpolation

Common task: Need to draw a nice curve through a set of points (for instance in computer graphics)



The interpolation problem: given $n+1$ pairs of numbers (x_i, y_i), $i = 0, 1, \dots, n$, find a function f such that $f(x_i) = y_i$

Function f is the **interpolant** of the point set

A classic choice: **polynomial interpolation**

$$f(x) = a_0 + a_1 x + a_2 x^2 + \dots + a_n x^n$$

Polynomial interpolation

Theorem

Let $(x_i, y_i)_{i=0}^n$ be an arbitrary set of pair of numbers where all the x_i are distinct. Then there is a unique polynomial p of degree $\le n$ such that

$$p(x_i) = y_i \qquad i = 0, \dots, n$$

For the proof, see Theorem 5.2.1 in Eldén, Wittmeyer–Koch

Note: The number of coefficients in polynomial = the number of points to interpolate. In Matlab:

```
p = polyfit(x, y, n);
```

- Vectors `x`, `y` (length $n+1$) contain the coordinates
- `n`: polynomial order
- `p`: vector containing polynomial coefficients

Polynomial interpolation

Polynomial coefficients are easy to determined by writing the polynomial as follows:

$$\begin{aligned} p(x) = b_0 + b_1(x - x_0) + b_2(x - x_0)(x - x_1) + \dots \\ + b_n(x - x_0)(x - x_1)\cdots(x - x_{n-1}) \end{aligned}$$

Conditions $p(x_i) = y_i$, $i = 0, \dots, n$, yield **Newton's interpolation formula**:

$$\begin{aligned} y_0 &= b_0 \\ y_1 &= b_0 + b_1(x_1 - x_0) \\ y_2 &= b_0 + b_1(x_1 - x_0) + b_2(x_2 - x_0)(x_2 - x_1) \\ &\vdots \\ y_n &= b_0 + \dots \qquad \qquad \dots + b_n(x_n - x_0)\cdots(x_n - x_{n-1}) \end{aligned}$$

An **undertriangular** system of equation for coefficients $b_0, \dots, b_n$.

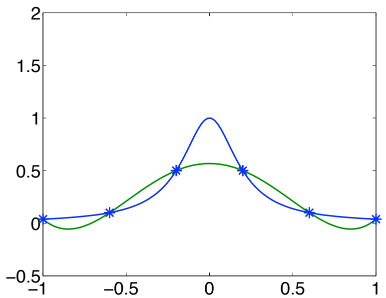
With Newton's interpolation formula, it is easy to add additional points to an already computed polynomial: just add one more row per point!

Polynomial interpolation

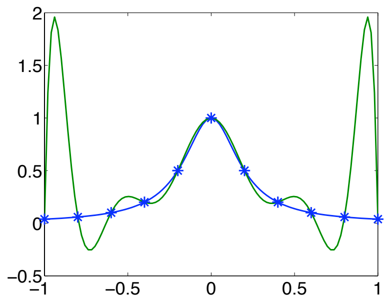
How does polynomial approximation perform?
Check: interpolate the function

$$f(x) = \frac{1}{1 + 25x^2}$$

Blue function f interpolated at $n+1$ equispaced points (marked $*$) with green polynomial p of degree n



6 points, $n = 5$



11 points, $n = 10$

Polynomial interpolation

Runge's phenomenon: Equispaced interpolation with polynomials tends to generate oscillations at the boundaries that become *worse* with increasing polynomial order

Conclusion:

- Interpolation with polynomials of high degree is often a terrible idea!
- Will often generate large oscillations **between** interpolation points

Cure 1:

- Change locations of the interpolation points by concentrating them along the boundaries
- A good choice: *Chebychev points*

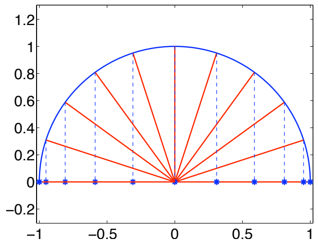
Cure 2: (the most common approach!)

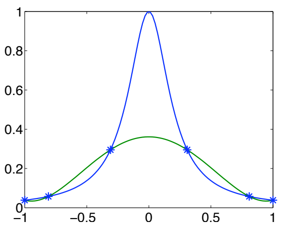
- Glue together *piecewise* polynomials of *low* degree (**splines**)

Cure 1: interpolation at Chebychev points

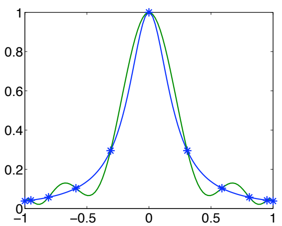
For interpolation on $[-1, 1]$ of polynomials of degree n, interpolate at the points

$$x_i = \cos\frac{i\pi}{n}, \qquad i = 0, \ldots, n$$

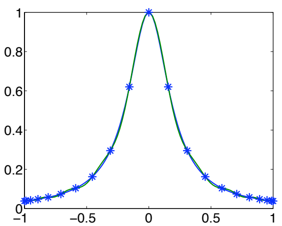




$n = 5$



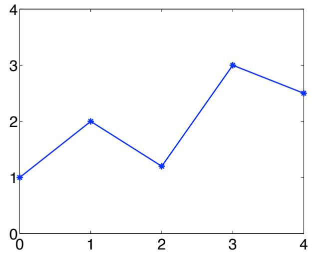
$n = 10$



$n = 20$

Cure 2: splines

- Cannot choose interpolation points in many cases! *Ex:* drawing programs
- The most common interpolation method: **splines:** piecewise polynomials of low degree. More appropriate than polynomial interpolation in most cases
- The simplest spline, **linear splines**, just continuous, piecewise-linear interpolation



Definition: A **spline** is a function that is composed by piecewise polynomials of degree k such that it is continuously differentiable $k-1$ times

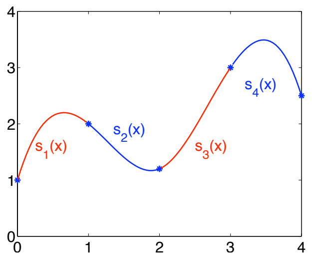
Most common, besides linear splines: **cubic** splines (e.g. CAD systems)

Cubic splines

- Assume $n+1$ pairs of numbers (x_i, y_i), $i = 0, \dots, n$
- The global function s is defined piecewise on the n intervals $[x_{i-1}, x_i]$, $i = 1, \dots, n$
- For $i = 1, \dots, n$, determine n cubic functions

  $$s_i(x) = a_0^{(i)} + a_1^{(i)} x + a_2^{(i)} x^2 + a_3^{(i)} x^3$$

  on intervals $[x_{i-1}, x_i]$



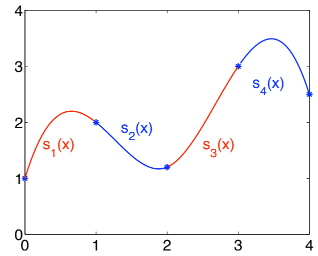
- Thus, there are $4n$ coefficients to determine
- Need $4n$ equations (conditions) to determine these coefficients

Cubic splines

- Interpolation in both ends:

  $$\begin{cases} s_i(x_{i-1}) = y_{i-1} \\ s_i(x_i) = y_i \end{cases} \qquad i = 1, \dots, n$$

  $2n$ conditions. Yields that the composite function is continuous



- Continuous derivatives and second derivatives where neighboring cubics are joined:

  $$\begin{cases} s_i'(x_i) = s_{i+1}'(x_i) \\ s_i''(x_i) = s_{i+1}''(x_i) \end{cases} \qquad i = 1, \dots, n-1$$

  $2(n-1)$ conditions
- Totally $2n + 2(n-1) = 4n - 2$ conditions. Two more conditions needed!

Cubic splines

There are several choices for the two extra conditions:

(i) "Non-a-knot" spline. Default in Matlab's `spline`. Imposes continuous **third** derivative at x_1 and x_{n-1}:

$$s_1'''(x_1) = s_2'''(x_1), \qquad s_{n-1}'''(x_{n-1}) = s_n'''(x_{n-1})$$

Note that $s_i''' = 6a_3^{(i)}$ is piecewise constant!

(ii) "Natural spline". Impose **zero curvature** at the end points:

$$s_1''(x_0) = 0, \qquad s_n''(x_n) = 0$$

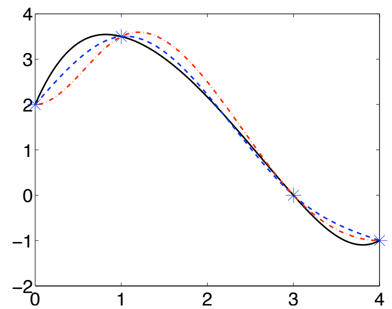
(iii) Impose **given slopes** g_L, g_R at the end points:

$$s_1'(x_0) = g_L, \qquad s_n'(x_n) = g_R$$

Option in Matlab's `spline`.

Cubic splines

——: not-a-knot
- - - -: natural
-·-·-: prescribed slopes $g_L = g_R = 0$



Note that splines are "global": local changes (for instance at the boundary) can affect the function everywhere!

Quadrature

Quadrature, also called *numerical integration*, concerns numerical computation of the definite integral

$$I(f) = \int_a^b f(x)\,dx = \sum_{i=1}^{n} w_i\, f(x_i) + \underbrace{R_n}_{\text{rest term (error)}}$$

(That is, we are **not** using primitive functions!)
In Matlab: use `quad` or `quadl`:

```
I = quad(func, a, b);
```

`func` is a **function handle**.

Quadrature

Example: We wish to compute the definite integral

$$I = \int_a^b e^{-x^2}\,dx$$

There is no primitive function to this integrand! Numerical integration is necessary! Typical procedure:

- If the integrand is a simple, one-line formula, easiest to use "anonymous functions" (Matlab terminology):

  ```
  func = @(x)  exp(-x.*x);
  I = quad(func, a, b);
  ```

  Then, variable `I` will contain a numerical estimate of $\int_a^b e^{-x^2}\,dx$
- **Note:** Functions used in `quad` **must** be written to accept vector arguments and to return corresponding vector result!
- For more complicated functions, it is more convenient to use a function m-file instead

Quadrature

- Implement a Matlab function `func` in the file `func.m`

  ```
  function f = func(x)
  f = exp(-x.*x);
  end
  ```

  Again note that function used in `quad` **must** be written to accept vector arguments and to return corresponding vector result!
- When integrating using the function `func`, write

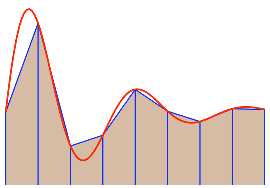
  ```
  I = quad(@func, a, b);
  ```
- Also possible to use an intermediate variable:

  ```
  integrand = @func;
  I = quad(integrand, a, b);
  ```


Quadrature

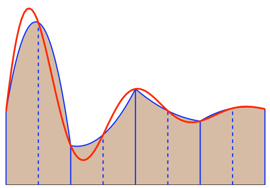
How is quadrature done? Two examples:

Example 1:



- Divide interval $[a, b]$ into a number (here 8) of intervals
- Interpolate f with continuous piecewise linears
- Sum up the areas of all right trapezoids (*paralleltrapetser*)
- Called the **trapezoidal rule** (*trapetsformeln*)

Example 2:



- Divide $[a, b]$ into a number (here 4) of double intervals
- Interpolate f with continuous piecewise quadratics
- Compute and sum up the integrals of the interpolated function
- Called **Simpson's** rule

Quadrature

- Examples can be generalized to higher-order: called **Newton–Cotes rules** when using piecewise polynomial interpolation with equispaced interpolation points on each piece
- Software for quadrature (e.g. `quad` and `quadl`) accept an input **tolerance**: the integral is computed within the given error tolerance by adjusting the step length
- Often most efficient to use **adaptive** methods: the step length is varied so that smaller steps are used where f changes rapidly. *Ex:* Matlab's `quad` uses adaptive Simpson
- Question: How can the method compute the error without knowledge of the exact solution?

Quadrature rules

The Trapezoidal Rule:

$$\int_{x_k}^{x_{k+1}} f(x)\,dx \approx (x_{k+1} - x_k)\frac{f(x_k) + f(x_{k+1})}{2}$$
$$= h\frac{f(x_k) + f(x_{k+1})}{2}$$

The Simpson Rule:

$$\int_{x_k}^{x_{k+2}} f(x)\,dx \approx (x_{k+2} - x_k)\frac{f(x_k) + 4f(x_{k+1}) + f(x_{k+2})}{6}$$
$$= 2h\frac{f(x_k) + 4f(x_{k+1}) + f(x_{k+2})}{6}$$

Note: double interval with $h = x_{k+2} - x_{k+1} = x_{k+1} - x_k$

Composite (*sammansatta*) rules

When using equidistant partitioning, we may sum up as below.

The Composite Trapezoidal Rule:

$$\int_a^b f(x)\,dx \approx \frac{h}{2}\sum_{k=0}^{n-1}[f(x_k) + f(x_{k+1})]$$
$$= \frac{h}{2}[f(x_0) + 2f(x_1) + \cdots + 2f(x_{n-1}) + f(x_n)] = I_T^{(h)}(a,b)$$

The Composite Simpson Rule: (n odd, i.e. even number of intervals)

$$\int_a^b f(x)\,dx \approx \frac{h}{3}\sum_{k=0,2,4,\ldots}^{n-1}[f(x_k) + 4f(x_{k+1}) + f(x_{k+2})]$$
$$= \frac{h}{3}[f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3)\cdots + 2f(x_{n-2}) + 4f(x_{n-1}) + f(x_n)]$$
$$= I_S^{(h)}(a,b)$$

Accuracy

The quadrature error is a **discretization error**

Theorem

For twice continuously differentiable f hold

$$\int_a^b f(x)\,dx = I_T^{(h)}(a,b) - \frac{h^2}{12}(b-a)f''(\xi)$$

for some $\xi \in [a,b]$.
For four time continuously differentiable f hold

$$\int_a^b f(x)\,dx = I_S^{(h)}(a,b) - \frac{h^4}{180}(b-a)f''''(\xi)$$

for some $\xi \in [a,b]$.

- Thus, the error is $O(h^2)$ and $O(h^4)$ for the trapezoidal and Simpson rule, respectively
- The Simpson rule requires a more regular f!

Error estimators

The error formulas can be used to estimate the error *without knowledge of the exact integral!*

$$\int_a^b f(x)\,dx = I_T^{(h)}(a,b) - \frac{h^2}{12}(b-a)f''(\xi_1) \qquad \xi_1 \in [a,b] \qquad (1)$$

$$\int_a^b f(x)\,dx = I_T^{(2h)}(a,b) - \frac{4h^2}{12}(b-a)f''(\xi_2) \qquad \xi_2 \in [a,b] \qquad (2)$$

Assume $f''(\xi_1) \approx f''(\xi_2)$ and set $E_T^{(h)} = -\frac{h^2}{12}(b-a)f''(\xi_1)$. Subtracting expression (2) from expression (2) yields

$$E_T^h = \frac{I_T^{(h)}(a,b) - I_T^{(2h)}(a,b)}{3} \qquad (\textit{“tredjedelsregeln”})$$

Thus, by performing **two** computations, with steps $2h$ and h, we can estimate the error when using step h

Error estimators

An analogous analysis yields for the Simpson rule:

$$E_S^h = \frac{I_S^{(h)}(a,b) - I_S^{(2h)}(a,b)}{15} \qquad (\textit{“femtondelsregeln”})$$

In general, for a integration rule with error term $O(h^p)$:

$$E_M^h = \frac{I_M^{(h)}(a,b) - I_M^{(2h)}(a,b)}{2^p - 1}$$

These estimates can be used in an adaptive process to locally refine in places where needed

Adaptive Simpson: a recursive algorithm

Want

$$\left| \int_a^b f(x)\,dx - I_{AS}(a,b) \right| \le \epsilon$$

0. Set actual interval to $[a,b]$
1. Compute I_S^h and I_S^{2h} on actual interval
2. Estimate the error using the $1/15$ rule
3. If the error $< \epsilon \frac{\textit{interval length}}{b-a}$
   - Accept $I_S^{(h)}$
   - Take next interval, otherwise done

   Otherwise
   - Reject $I_S^{(h)}$
   - Cut the interval in two halves
   - For each subinterval, continue at 1. with $h \leftarrow h/2$