1. **Rational and polynomial interpolation.** Approximate a function $f(x)$ in given intervals using polynomial and rational function interpolation.

   (a) $f(x) = \cos(x)$, $x \in [0, 2\pi]$.

   (b) Function with a pole in the interpolation interval, $f(x) = \tan(x)$, $x \in [0, 2]$.

   (c) Runge's function, $f(x) = 1/(1 + 25x^2)$, $x \in [-1, 1]$.

   (d) Function with an essential singularity, $f(x) = \ln(x^2/2)$, $x \in [\epsilon, 1/10]$, where $\epsilon > 0$ is a small number that is representable with single precision variable.

   Tabulate the functions at a few, N, points and use `ratint` and `polint` to do the interpolation. Write your main program so that it prompts the user for the number N, the start and end points of the interval and the output filename. Write the values of the interpolating functions into the file in, say, 200 evenly spaced abscissae that include the start and end points. Experiment with different values of N.

   The idea is that only the function f needs to be changed, so that for each item (a)–(d) the same main program will suffice.

2. **Cubic spline interpolation.** In many practical applications it is impossible to use interpolating polynomial or rational functions. If a large data set needs to be approximated by a smooth function, the cubic spline is a better method.

   (a) Approximate the function $\sin(x)$, $0 \le x < 2\pi$, using the cubic spline interpolation.

   (b) Non-smooth functions are, however, problematic to interpolate using the cubic spline. Use it to try and approximate the function

   $$f(x) = \begin{cases} x+3, & x \le -1 \\ -2x, & -1 < x \le 1 \\ x-3, & 1 < x \end{cases}, \qquad -4 < x \le 4.$$

   Use Numerical Recipes library functions `spline` and `splint` to do the interpolation.

3. **Derivatives.** Write a routine that calculates the first derivatives of a function using the (a) two point formula and (b) three point formulas. Assume, that the values of the function are tabulated at evenly spaced abscissae and that the derivatives, evaluated at the same points, are to be stored in another table. The syntax of the subroutines ought to be

```
void deriv2(double h, double y[], double dy[], int n);
void deriv3(double h, double y[], double dy[], int n);
```

   Here, `y` is a table containing the tabulated values of the function, while `n` is its length. Argument `h` is the x separation of adjacent samples. Finally, `dy` is the array, size `n`, for the routine output, that is, the derivatives.

   Run a test of the derivation routine by tabulating the values of the function $\sin(x)$. Compare the values of the two methods against each other using the same number of `n`. Study the effect of increasing the number of tabulation points.

`void polint(float xa[], float ya[], int n, float x, float *y, float *dy)`

Evaluates the polynomial interpolating given points at a point

Arguments

`float xa[]`	*input*	Array of x–coordinates
`float ya[]`	*input*	Array of y–coordinates
`int n`	*input*	Number of points
`float x`	*input*	Point where to evaluate the polynomial
`float *y`	*output*	Pointer to the value of the interpolating function
`float *dy`	*output*	Pointer to an error estimate

`void ratint(float xa[], float ya[], int n, float x, float *y, float *dy)`

Evaluates the rational function interpolating given points at a point

Arguments

`float xa[]`	*input*	Array of x–coordinates
`float ya[]`	*input*	Array of y–coordinates
`int n`	*input*	Number of points
`float x`	*input*	Point where to evaluate the rational function
`float *y`	*output*	Pointer to the value of the interpolating function
`float *dy`	*output*	Pinter to an error estimate

`void spline(float xi[], float yi[], int n, float y1p, float ynp, float y2[])`

Constructs a table of second derivatives for use with cubic spline interpolation

Arguments

`float xi[]`	*input*	Array of x–coordinates
`float yi[]`	*input*	Array of y–coordinates
`int n`	*input*	Size of arrays `xi` and `yi`
`float y1p`	*input*	Derivative of the function at start point. If $> 10^{30}$, assume natural spline (first derivative chosen so that second derivative is zero)
`float ynp`	*input*	Derivative of the function at end point. If $> 10^{30}$, assume natural spline (first derivative chosen so that second derivative is zero)
`float y2[]`	*output*	Array of the second derivatives

`void splint(float xi[], float yi[], float y2[], int n, float x, float *y)`

Evaluates the spline

Arguments

`float xi[]`	*input*	Array of x–coordinates
`float yi[]`	*input*	Array of y–coordinates
`float y2[]`	*output*	Array of the second derivatives, as constucted by `spline`–routine
`int n`	*input*	Size of arrays `xi`, `yi` and `y2`
`float x`	*input*	Point where spline is evaluated
`float *y`	*output*	Pointer to the value of the spline