Interpolation

The objective is to find value of the function at any point x if one has values f_i tabulated at certain points x_i.

Most straightforward (*never* dramatically *fails*) and safe method is linear interpolation: $f(x) = f_i + \frac{x-x_i}{x_{i+1}-x_i}(f_{i+1} - f_i)$. The hardest part is to find integer i such that $x_i < x < x_{i+1}$.

If one needs interpolation in non-correlated points x, the best algorithm is bisection (needs $\log_2 N$ evaluations). However, most of the time, one needs points at increasing (or decreasing) values x. In this case, it is desirable to store previous position i in table, and one can start to search from this old point. One should check first if x is stil in the interval $[x_i, x_{i+1}]$, if not, one should check weather it is in $[x_{i+1}, x_{i+2}]$, ... than one should use bisection in small interval $[x_{i+2}, x_{i+M+1}]$ for some integer M of the order of 10. If this does not work, one should use bisection on the rest of the grid. Since this occurs very seldom, bisection is not expenisive.

Here is the implementation

```
inline int mesh::find_(double x, int& ai0) const
{ // This is most often called searching routine
  // It is used for searching table in increasing order
  if (x<om[ai0+1]) return ai0; // Checks weather x is stil in [ai0:ai0+1]
  int ai1 = ai0+1;             // Makes a step
  if (ai0>=N-2) return ai0;    // Needs to check for the end of the table
  if (x<om[ai1+1]){            // Checks weather x is in [ai0+1:ai0+2]
    ai0 = ai1;
    ai1 = ai1+1;
    return ai0;
  }
  if (ai1>=N-2) return ai1; // Again checks for the end of the table
  ai0 = ai1+1;              // makes another step
  if (ai1+dN<N){            // First uses bisection is small interval between [ai1:ai1+dN]
    if (x<om[ai1+dN]) return bisection (x, ai0, ai1, ai1+dN+1);
  } else return bisection (x, ai0, ai1, N);
  if (ai1+dN<N-1) ai0 = ai1+dN;// If still not found, use bisection on the rest of the grid
  else ai0 = ai1+dN-1;
  return bisection (x, ai0, ai1, N);
}
```

The implementation of simple interpolation can be done using two large classes: `mesh` and `function`. The interaction is mantained through a small class for interpolation which contains the position in the table and interpolating coefficient.

Here is the implementaion of the class containing x values:

```
class mesh{
protected:
  int N, N0;       // size, size of the allocated memory (might be larger than size)
  double *om;      // grid points
  static const int dN = 10; // when searching ordered table, searching is done first between a0 and a0+dN...
protected:
  mesh(): N(0),N0(0),om(NULL),delta(NULL),dh(NULL){}; // constructor is made protected such that mesh can not be instantiated
  ~mesh(){};                                          // This class is used only as base class
public:
  // OPERATORS
  double& operator[](int i) {Assert(i<N,"Out of range in mesh[]"); return om[i];}
  const double& operator[](int i) const {Assert(i<N,"Out of range in mesh[]"); return om[i];}
  // ROUTINES FOR SEARCHING ORDERED TABLE
  int find(double x) const;           // searching table if previous position is not known
  int find_(double x, int& ia) const; // searching table forward from previous position
  int _find(double x, int& ia) const; // searching table backward from previous position
  int findBoth(double x, int& ia) const;  // searching table in both direction - point is usually close
  // LINEAR INTERPOLATION ROUTINES - INITIALIZATION
  tint InitInterpLeft() const {return 0;}    // Initialization of position for forward search
  tint InitInterpRight() const {return N-2;} // Initialization of position for backward search
  // LINEAR INTERPOLATION ROUTINES
  intpar Interp(const double x) const;               // Finds parameters for linear interpolation at point x
  intpar InterpLeft(const double x, tint& a) const;  // Finds parameters for linear interpolation when freqeuncy is increasing
  intpar InterpRight(const double x, tint& a) const; // Finds parameters for linear interpolation when freqeuncy is decreasing
  intpar InterpBoth(const double x, tint& a) const;  // If frequency is believed to be close to previous frequency
private:
  int bisection(double x, int& klo, int& khi, int bi) const;
};
```

There are actually four different routines for searching:

a) `find` : straighforward bisection in case there is no information about previous position in table

b) `find_` : searching in increasing order (starting from previous position)

c) `_find` : searching in decreasing order (starting from previous position)

d) `findBoth` : if new x is most probably close to previous position but not clear whether before or after.

Interpolation is done by four different functions

a) `Interp(x)`

b) `InterpLeft(x,previous_position)`

c) `InterpRight(x,previous_position)`

d) `InterpBoth(x,previous_position)`

In case of linear interpolation, all four functions above return small class `intpar`

```
class intpar{
public:
  int i;
  double p;
  intpar(int i_, double p_) : i(i_), p(p_){}
  intpar(){};
  intpar(double d) : i(0), p(d) {};
};
```

Check implementation or implement yourself

The second class is class `function` which contains function values f_i.

```
template<class T>
class function{
protected:
  T *f;
  int N0, N;
protected:
  function() : f(NULL), N0(0), N(0) {}; // constructor is made protected such that mesh can not be instantiated
  explicit function(int N_) : N0(N_), N(N_) {};// This class is used only as base class
  ~function(){};
  function(const function&){};
public:
  // OPERATOR FOR INTERPOLATION
  T operator()(const intpar& ip) const; // linear interpolation
  // OTHER OPERATORS
  T& operator[](int i) {Assert(i<N,"Out of range in function[]"); return f[i];}
  const T& operator[](int i) const {Assert(i<N,"Out of range in function[]"); return f[i];}
  function& operator+=(const function& m);
  function& operator*=(const T& m);
  function& operator=(const T& c); // used for initialization
};
```

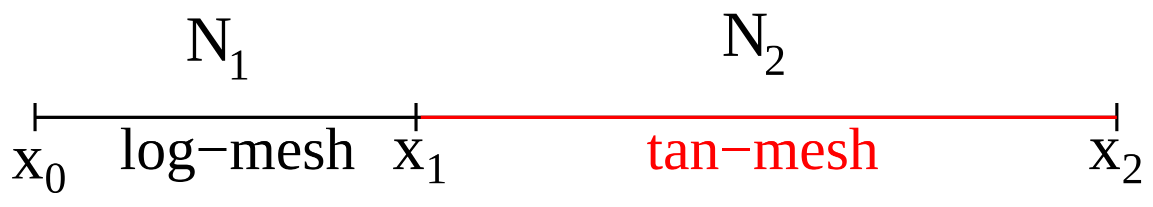
Here is an example of the usage of above classes:

```
  mesh1D om, x;
  function1D<double> f(om.size());
  .......
  tint position = om.InitInterpLeft();   // Initialization of interpolation with increasing argument
  for (int i=0; i<x.size(); i++){        // x and om grid points do not coinside
    intpar p = om.InterpLeft(x[i],position); // calculates location in ordered table and interpolating coefficient
    cout<<x[i]<<" "<<f(p)<<endl;         // operator()(intpar p) actually interpolates function f
    cout<<x[i]<<" "<<f(om.InterpLeft(x[i],position))<<endl; // Equivalent to the above two lines
  }
```

Like in case of integration, it is crucial to keep function on grid points, on which the function is resolved. Typical non-equidistant grids are

- logarithmic grids - are natural for highly peaked functions
- tan grid - is natural for lorentzian-like functions (is linear at small frequency and grows fast at high frequency).

Sometimes it is desirable to combine various grids to resolve functions which contain a lot of structure. We will show here how to combine logarithmic grid at low frequency and tan-grid at high frequency.



Input parameters of the grid are:

Start of the log-mesh x_0

Start of the tan-mesh x_1

End of the mesh x_2

Total number of points N

Number of log-points N_1

$$\begin{aligned} \omega_i &= \exp\left(\log x_0 + \tfrac{i}{N_1-1}\log(\tfrac{x_1}{x_0})\right) & i \in [0, N_1-1] \quad & \omega \in [x_0, x_1] \\ \omega_i &= w\tan(a + b\tfrac{i+1}{N_2}) & i \in [0, N_2-1] \quad & \omega \in (x_1, x_2] \end{aligned} \tag{1}$$

The grid should be continuous with continuous derivative. We have

$$x_1 = w\tan a \tag{2}$$

$$x_2 = w\tan(a+b) \tag{3}$$

$$d\omega \equiv \frac{d\omega_i}{di} = \frac{\log(x_1/x_0)}{N_1-1}x_1 = \frac{w\ b}{N_2\cos^2 a} \tag{4}$$

This system of equations can be solved by solving the following transcendental equations

$$u - \arctan\left(\frac{\tan u}{\widetilde{x}}\right) = d\widetilde{\omega}\widetilde{x}\frac{\tan u}{\widetilde{x}^2 + \tan^2 u} \tag{5}$$

Here $u = a + b$, $\widetilde{x} = x_2/x_1$, $d\widetilde{\omega} = \log(x_1/x_0)N_2/(N_1-1)$.

This particular form of equation is most convenient for numerics, because it has either no solutions or one solution in the interval $u \in [0, \pi/2]$.

The condition for solution is that the left-hand side part of equation should increase faster than right-hand side, i.e., $u - u/\widetilde{x} > d\widetilde{\omega}\widetilde{x}u/\widetilde{x}^2$. This leads to the following condtion

$$\frac{N_1 - 1}{N_2} > \frac{\log\left(\frac{x_1}{x_0}\right)}{\frac{x_2}{x_1} - 1} \equiv \eta \tag{6}$$

equivalent to

$$N_1 > \frac{1 + \eta N}{1 + \eta} \tag{7}$$

Instead of taking N_1 and N_2 as input parameters, one can choose total number of points N and the coefficient α such that

$$N_1 = (1 + \alpha)\left[\frac{1 + \eta N}{1 + \eta} + \frac{1}{2}\right] \tag{8}$$

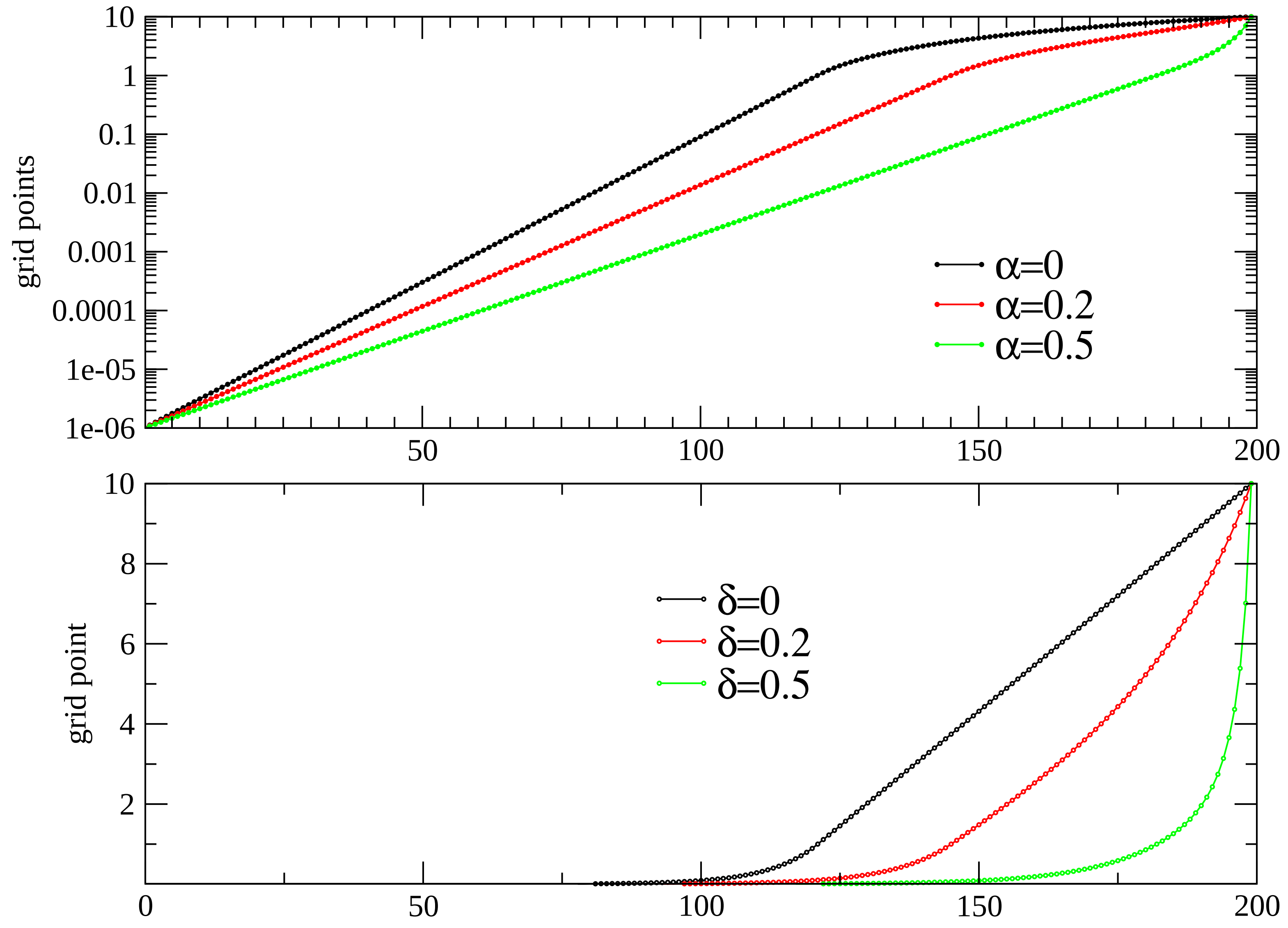


Figure 1: Here is figure for the combined mesh with parameters $x_0 = 10^{-6}$, $x_1 = 1$, $x_2 = 10$, $N = 200$ and various α's.

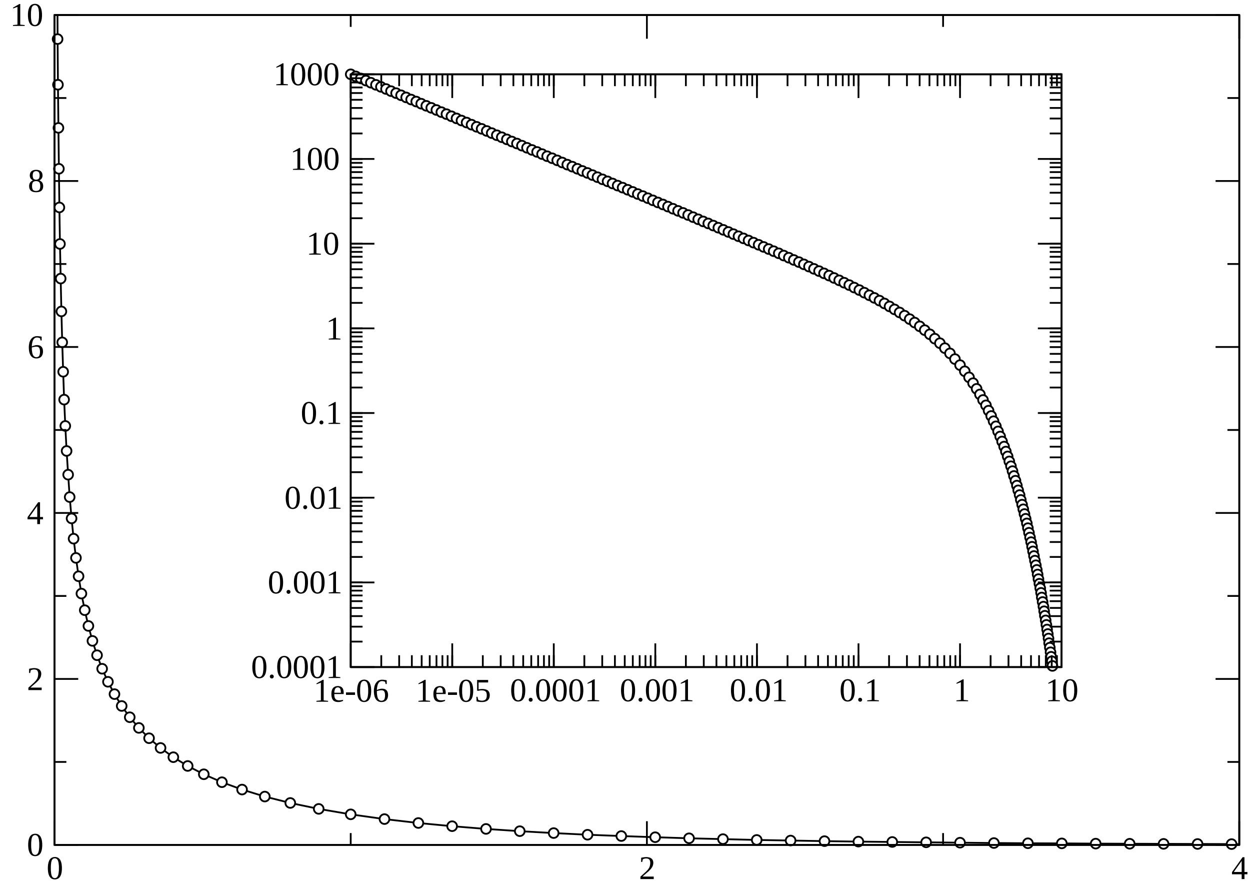


Figure 2: Here we plotted function $e^{-x}/\sqrt{x}$ on the above mentioned grid. Its integral, using trapezoid rule, is 1.772884 while the exact answer is $\sqrt{\pi}$ which is just $4\,10^{-4}$ off. However, this agreement is accidental since the error should be around 10^{-3} or larger. (Cutoff at small x is balanced by trapezoid approximation).

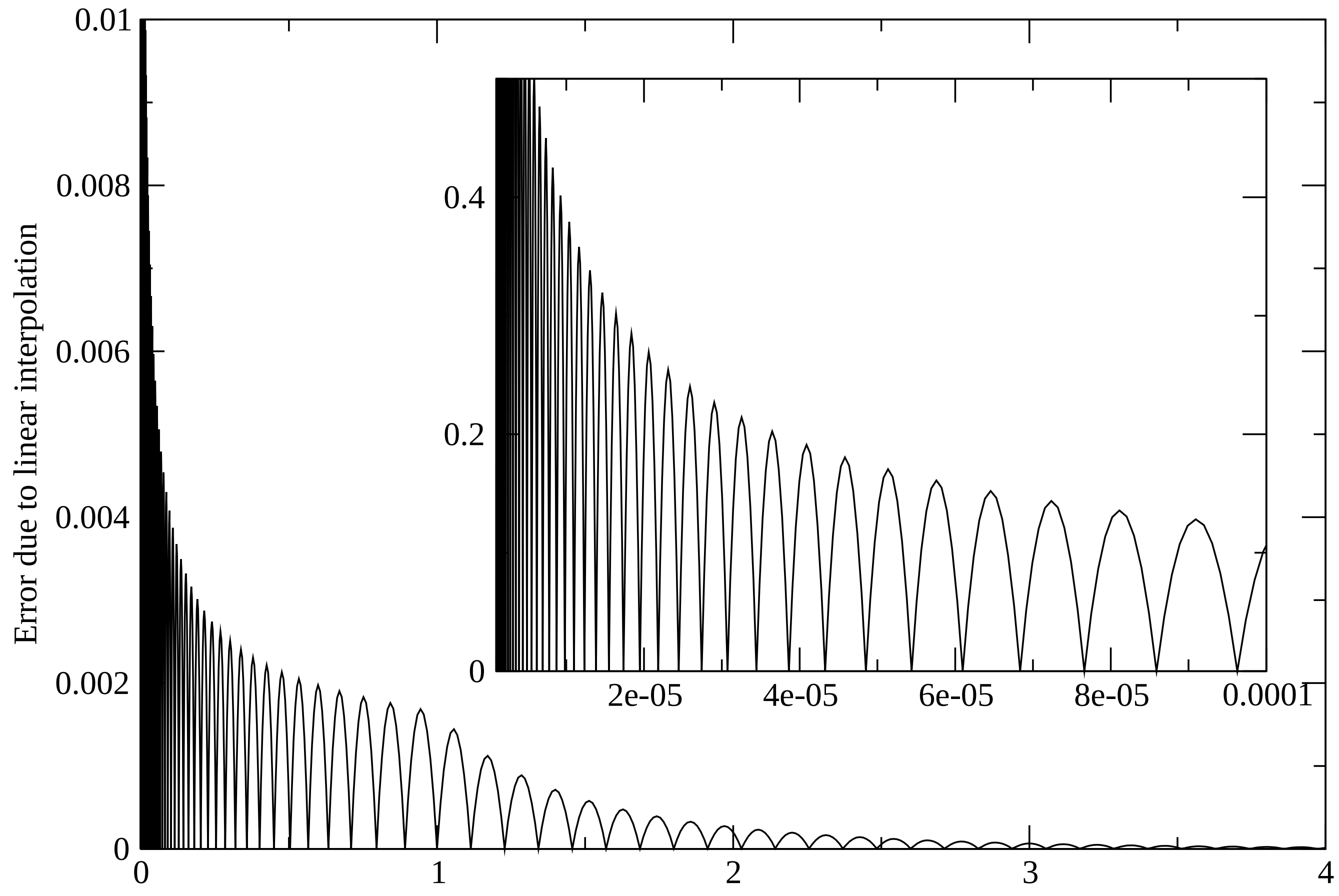


Figure 3: The linear interpolation in case of concave function is always larger than the original function. Here we see the difference between the linear interpolated function and the exact function (this is errror due to linear interpolation) which is always positive. The integral of this difference is $2.4\,10^{-3}$ which is a better estimate of the error due to trapezoid integration (== error of linear interpolation).

Splines

When linear interpolation is not good enough and if we believe that function is smooth enough (smooth on the mesh it is specified), we might want to use higher order interpolation. It turns out that the best of high-order interpolations is spline : pice-vice cubic interpolation.

The idea is to require continuous first derivative and continuous second derivative. The second derivative changes linearly between points x_i to x_{i+1}.

If we denote the normalized difference between points

$$p = \frac{x - x_j}{x_{j+1} - x_j}$$

the desired interpolation in the interval $[x_j - x_{j+1}]$ might be written in the form

$$y = p\, y_{j+1} + (1-p)\, y_j + \frac{1}{6}[(1-p)^3 - (1-p)](x_{j+1} - x_j)^2 y_j'' + \frac{1}{6}[p^3 - p](x_{j+1} - x_j)^2 y_{j+1}'' \tag{9}$$

If we take the first derivative, and take into account that $dp/dx = 1/(x_{j+1} - x_j)$ we get

$$\frac{dy}{dx} = \frac{y_{j+1} - y_j}{x_{j+1} - x_j} + \frac{1}{6}[1 - 3(1-p)^2](x_{j+1} - x_j) y_j'' - \frac{1}{6}[1 - 3p^2](x_{j+1} - x_j) y_{j+1}'' \tag{10}$$

Second derivative than becomes

$$\frac{d^2y}{dx^2} = py''_{j+1} + (1-p)y''_j \tag{11}$$

and is therefore pice-vice linear (second derivative is continuous by construction and is linearly interpolated between the points).

We do not have second derivatives, however, we need to require that first derivatives are continuous. We thus need to equate $\frac{dy}{dx}(p=0)$ from the interval $[x_j - x_{j+1}]$ to be equal to $\frac{dy}{dx}(p=1)$ from the interval $[x_{j-1} - x_j]$. We thus have a system of linear symmetric tridiagonal equations

$$\frac{(x_j - x_{j-1})}{6} y''_{j-1} + \frac{(x_{j+1} - x_{j-1})}{3} y''_j + \frac{(x_{j+1} - x_j)}{6} y''_{j+1} \tag{12}$$

$$= \frac{y_{j+1} - y_j}{x_{j+1} - x_j} - \frac{y_j - y_{j-1}}{x_j - x_{j-1}} \tag{13}$$

There are only $N-2$ equations because j is allowed to run between 1 and $N-2$. We need 2 additional conditions. There are two possibilities

- The second derivative is set to zero at both ends (”natural splines”)
- The first derivatives are given at both ends (They have to be known or estimated).

In the second case, assuming the first derivative at x_0 is $\dot{f}_0$ and at x_{N-1} is $\dot{f}_N$ we have

$$\frac{x_1 - x_0}{3} y_0'' + \frac{x_1 - x_0}{6} y_1'' = \frac{y_1 - y_0}{x_1 - x_0} - \dot{f}_0 \tag{14}$$

$$\frac{x_{N-1} - x_{N-2}}{6} y_{N-2}'' + \frac{x_{N-1} - x_{N-2}}{3} y_{N-1}'' = \dot{f}_N - \frac{y_{N-1} - y_{N-2}}{x_{N-1} - x_{N-2}} \tag{15}$$

If we take a vector of unknowns to be,

$$X = \left(y_0'', y_1'', y_2'', \cdots, y_{N-3}'', y_{N-2}'', y_{N-1}''\right) \tag{16}$$

we need to solve the following equation

$$AX = B \tag{17}$$

where tridiagonal matrix to be inverted takes the form

$$A = \left(\begin{array}{c|cccccc|c} \frac{x_1-x_0}{3} & \frac{x_1-x_0}{6} & 0 & 0 & 0 & 0 & \cdots \\ \hline \frac{x_1-x_0}{6} & \frac{x_2-x_0}{3} & \frac{x_2-x_1}{6} & 0 & 0 & 0 & \cdots \\ 0 & \frac{x_2-x_1}{6} & \frac{x_3-x_1}{3} & \frac{x_3-x_2}{6} & 0 & 0 & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \frac{x_{N-2}-x_{N-3}}{6} & \frac{x_{N-1}-x_{N-3}}{3} & \frac{x_{N-1}-x_{N-2}}{6} \\ \hline \cdots & \cdots & \cdots & \cdots & 0 & \frac{x_{N-1}-x_{N-2}}{6} & \frac{x_{N-1}-x_{N-2}}{3} \end{array}\right) \tag{18}$$

Using short notation $\dot{y}_0 = \frac{y_1-y_0}{x_1-x_0}$, the right-hand side of the tridiagonal system is

$$B = \left(\dot{y}_0 - \dot{f}_0, \dot{y}_1 - \dot{y}_0, \dot{y}_2 - \dot{y}_1, \cdots, \dot{y}_{N-3} - \dot{y}_{N-4}, \dot{y}_{N-2} - \dot{y}_{N-3}, \dot{f}_N - \dot{y}_{N-2},\right) \tag{19}$$

In case of ”natural splines”, we take $y''_0 = 0$ and $y''_{N-1} = 0$ therefore the first and the last equation should be omitted (the first and the last row and column of the matrix).

The alorithm for splinning is

- First solve the above system of tridiagonal linear equations to obtain y'' (ones and for all)
- Use the equation

$$y = (1-p)\, y_j + p\, y_{j+1} + \frac{1}{6}[(1-p)^3 - (1-p)](x_{j+1}-x_j)^2 y''_j + \frac{1}{6}[p^3 - p](x_{j+1}-x_j)^2 y''_{j+1} \tag{20}$$

  to calculate value of the function at any point x. To search the order table, use the above algorithm for linear interpolation.

Once we splined a function, we can use splines to evaluate integral or any other transformation pice-vice analytically.

For the integral we have

$$\int_{x_j}^{x_{j+1}} y(x)dx = \frac{1}{2}(y_{j+1} + y_j)(x_{j+1} - x_j) - \frac{1}{24}(y''_{j+1} + y''_j)(x_{j+1} - x_j)^3 \tag{21}$$

One can get very precise Fourier transformation using splines

$$\int_{x_j}^{x_{j+1}} e^{i\omega x} y(x) dx = \Delta x e^{i\omega x_j} \left[y_j \frac{1 + iu - e^{iu}}{u^2} + y_{j+1} \frac{(1-iu)e^{iu} - 1}{u^2} \right.$$
$$\left. + \frac{1}{6}(\Delta x)^2 \left(y''_j \frac{e^{iu}(6+u^2) + 2(u^2 - 3iu - 3)}{u^4} + y''_{j+1} \frac{6 + u^2 + 2e^{iu}(u^2 + 3iu - 3)}{u^4} \right) \right]$$

where $\Delta x = x_{j+1} - x_j$ and $u = \omega \Delta x$ For small ω or small Δx, the above equation is not stable to evaluate. One needs to use Taylot expansion, which gives

$$\int_{x_j}^{x_{j+1}} e^{i\omega x} y(x) dx = \Delta x e^{i\omega x_j} \left[y_j \frac{1}{2}(1 + \frac{i}{3}u) + y_{j+1} \frac{1}{2}(1 + \frac{2i}{3}u) \right.$$
$$\left. - \frac{1}{24}(\Delta x)^2 \left(y''_j (1 + \frac{7i}{15}u) + y''_{j+1}(1 + \frac{8i}{15}u) \right) \right]$$

The implementation is surprisingly simple in the above scheme of classes. Class `mesh` does not need to be changed, neither the small class `intpar`. We can derive a class `spline1D` from class `function` adding some new members

```
template <class T>
class spline1D : public function<T>{
  T* f2;       // second derivatives
  double *dxi; // x_{j+1}-x_j
public:
  // CONSTRUCTORS AND DESTRUCTORS
  spline1D() : f2(NULL), dxi(NULL) {};// default constructor exists allocating arrays
  // constructor
  spline1D(const mesh1D& om, const function<T>& fu, // the function being splined
   const T& df0=std::numeric_limits<double>::max(), // the derivatives at both ends
   const T& dfN=std::numeric_limits<double>::max());
  ~spline1D();                 // destructor
  spline1D(const spline1D& m); //copy constructor
  // INITIALIZATION ROUTINES
  void resize(int N_);
  // OPERATORS
  T operator()(const intpar& ip) const; // spline interpolation
  spline1D& operator=(const spline1D& m); // copy operator
  // ADVANCED FUNCTIONS
  T integrate();
  dcomplex Fourier(double om, const mesh1D& xi);
};
```

The system of equations is solved by calling LAPACK routine `dptsv_` in the following way

```
template <class T>
inline spline1D<T>::spline1D(const mesh1D& om, const function<T>& fu, const T& df0, const T& dfN)  : f2(NULL), dxi(NULL)
{
  if (om.size()!=fu.size()) cerr<<"Sizes of om and f are different in spline setup"<<endl;
  resize(om.size()); // Calling constructor to initialize memory
  std::copy(fu.f,fu.f+N,f);
  function1D<T> diag(om.size()), offdiag(om.size()-1); // matrix is stored as diagonal values + offdiagonal values
  // Below, matrix and rhs is setup
  diag[0] = (om[1]-om[0])/3.;
  double dfu0 = (fu[1]-fu[0])/(om[1]-om[0]);
  f2[0] = dfu0-df0;
  for (int i=1; i<om.size()-1; i++){
    diag[i] = (om[i+1]-om[i-1])/3.;
    double dfu1 = (fu[i+1]-fu[i])/(om[i+1]-om[i]);
    f2[i] = dfu1-dfu0;
    dfu0 = dfu1;
  }
  diag[N-1] = (om[N-1]-om[N-2])/3.;
  f2[N-1] = dfN - (fu[N-1]-fu[N-2])/(om[N-1]-om[N-2]);
  for (int i=0; i<om.size()-1; i++) offdiag[i] = (om[i+1]-om[i])/6.;
  // The system of symmetric tridiagonal equations is solved by lapack
  int one=1, info=0;
  if (df0==std::numeric_limits<double>::max() || dfN==std::numeric_limits<double>::max()){
    int size = N-2;// natural splines
    dptsv_(&size, &one, diag.MemPt()+1, offdiag.MemPt()+1, f2+1, &N, &info);
    f2[0]=0; f2[N-1]=0;
  } else  dptsv_(&N, &one, diag.MemPt(), offdiag.MemPt(), f2, &N, &info);

  if (info!=0) cerr<<"dptsv return an error "<<info<<endl;
  // Setup of other necessary information for doing splines.
  for (int i=0; i<om.size()-1; i++) dxi[i] = (om[i+1]-om[i]);
}
```

The interpolation is trivial

```
template <class T>
T spline1D<T>::operator()(const intpar& ip) const
{
  int i= ip.i; double p = ip.p, q=1-ip.p;
  return q*f[i] + p*f[i+1] + dxi[i]*dxi[i]*(q*(q*q-1)*f2[i] + p*(p*p-1)*f2[i+1])/6.;
}
```

The integration is simple

```
template <class T>
inline T spline1D<T>::integrate()
{
  T sum=0;
  for (int i=0; i<N-1; i++) sum += 0.5*dxi[i]*(f[i+1]+f[i]-(f2[i+1]+f2[i])*dxi[i]*dxi[i]/12.);
  return sum;
}
```

and the Fourier transformation slightly more involved

```
template <class T>
inline dcomplex spline1D<T>::Fourier(double om, const mesh1D& xi)
{
  dcomplex ii(0,1);
  dcomplex sum=0;
  dcomplex val;
  for (int i=0; i<N-1; i++) {
    double u = om*dxi[i], u2=u*u, u4=u2*u2;
    if (fabs(u)<1e-4){// Taylor expansion for small u
      val = 0.5*(1.+ii*(u/3.))*f[i]+0.5*(1.+ii*(2*u/3.))*f[i+1];
      val -= dxi[i]*dxi[i]/24.*(f2[i]*(1.+ii*7.*u/15.)+f2[i+1]*(1.+ii*8.*u/15.));
    }else{
      dcomplex exp(cos(u),sin(u));
      val  = (f[i]*(1.+ii*u-exp) + f[i+1]*((1.-ii*u)*exp-1.))/u2;
      val += dxi[i]*dxi[i]/(6.*u4)*(f2[i]*(exp*(6+u2)+2.*(u2-3.*ii*u-3.))+f2[i+1]*(6+u2+2.*exp*(u2+3.*ii*u-3.)));
    }
    sum += dxi[i]*dcomplex(cos(om*xi[i]),sin(om*xi[i]))*val;
  }
  return sum;
}
```

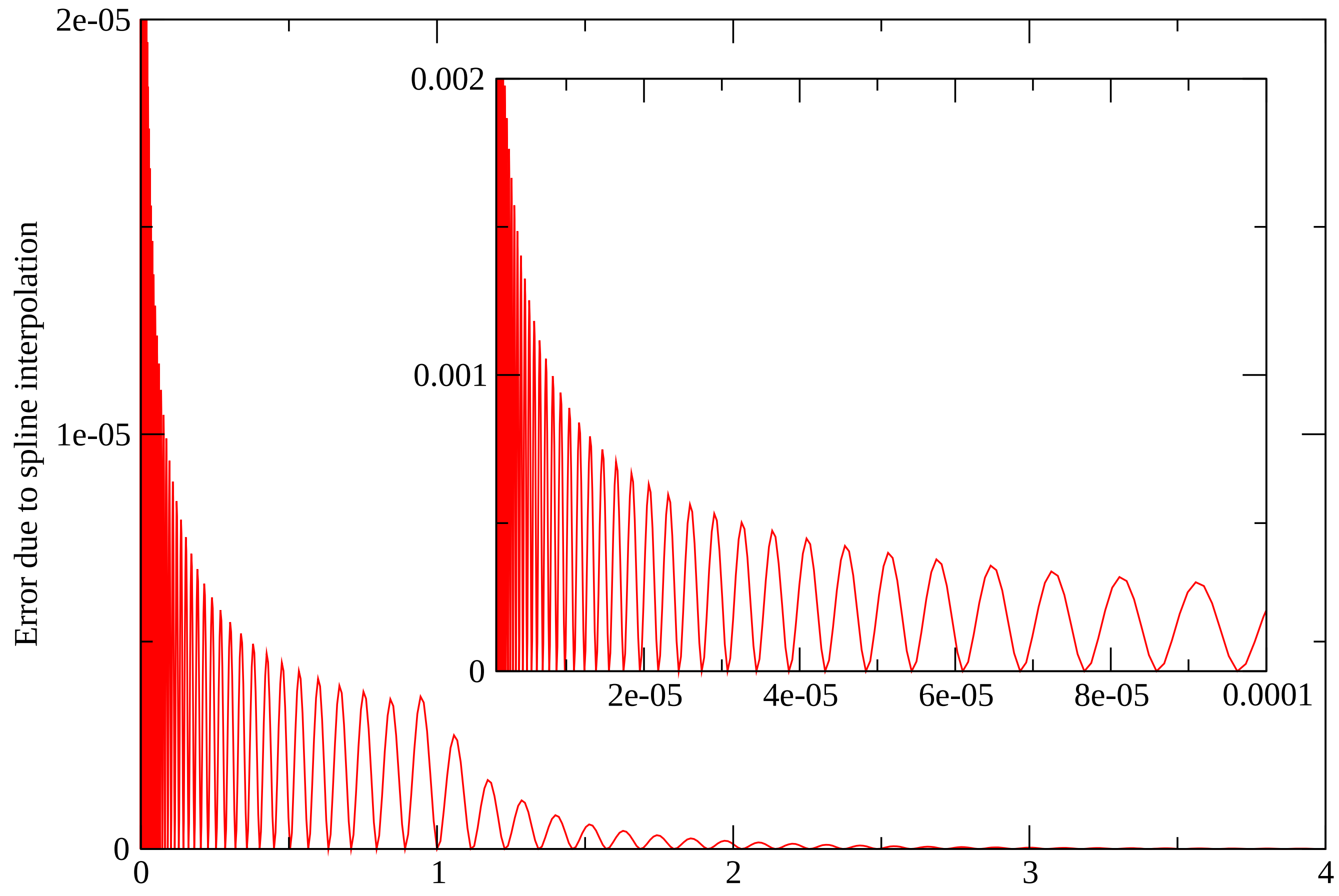


Figure 4: The spline interpolation of the same function $e^{-x}/\sqrt{x}$ gives 2-3 orders of magnitude smaller error. The integral of spline from $[10^{-6} - 10]$ is 1.7704363 while the exact integral ($\sqrt{\pi}\mathrm{erf}(\sqrt{x})$) is 1.7704401, the error is thus $4\ 10^{-6}$ which is roughly the integral of the above function ($5\ 10^{-6}$ is a good estimate for the error).

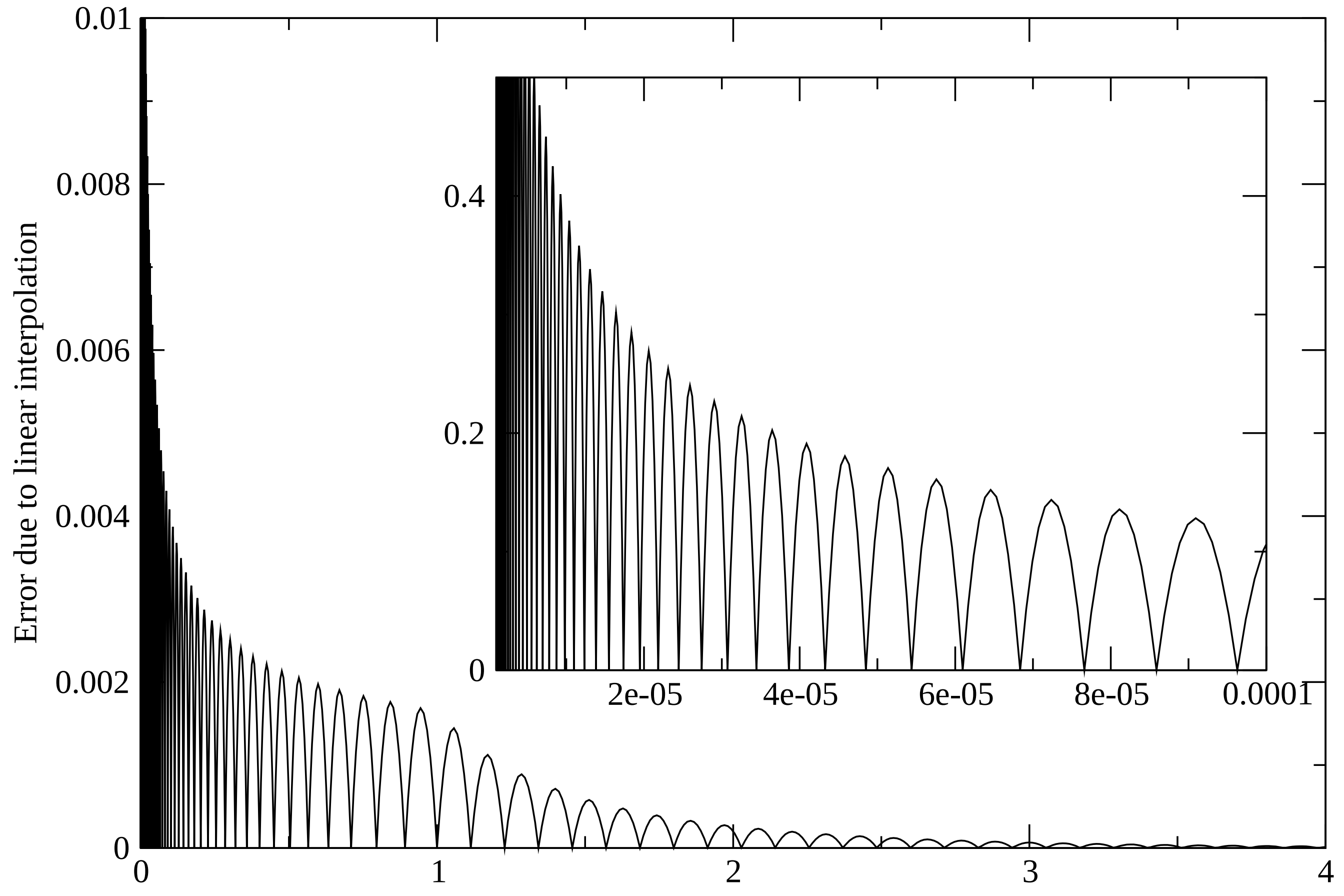


Figure 5: The linear interpolation repeated. Just for easier comparison with spline interpolation.

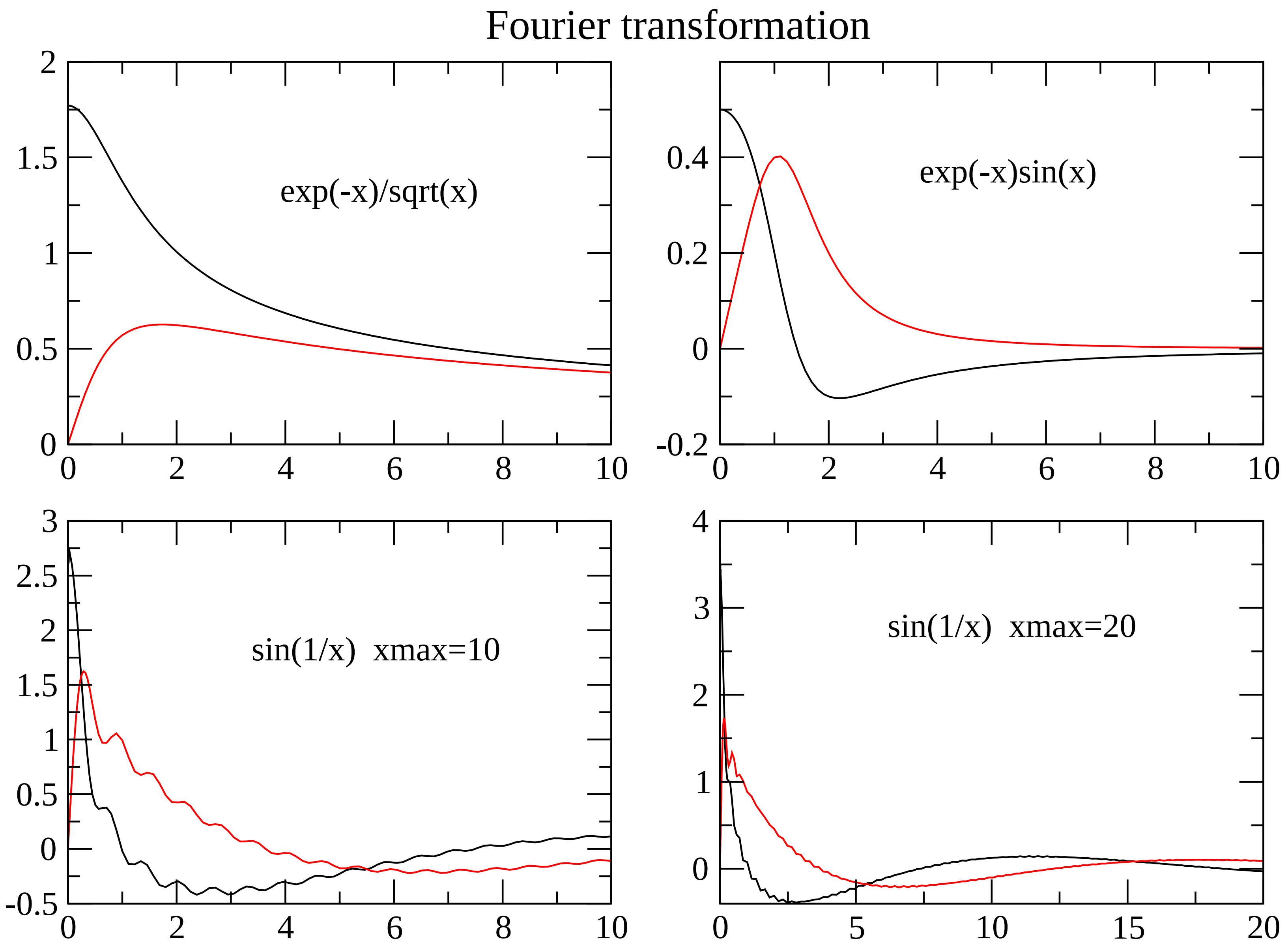


Figure 6: Fourier transformation using splines of couple of ”not-so well behaving” functions.