Spline representation of QCDNUM results

SPLINT version 2022-03-08

M. Botje*
Nikhef, Science Park 105, 1098XG Amsterdam, the Netherlands

March 3, 2022

Abstract

The SPLINT package is a QCDNUM add-on that turns results computed on the evolution grid into cubic splines in x and μ^2. Such splines are efficient representations of QCDNUM results and allow to integrate and differentiate these results.

*email m.botje@nikhef.nl

Contents

1 Introduction

The SPLINT package is an integral part of the QCDNUM distribution[1] and contains a set of routines to construct cubic spline interpolation functions.

A cubic spline is a set of piecewise third-degree polynomials defined on intervals in u:

$$S(u_i \leq u < u_{i+1}) = f_i + b_i(u - u_i) + c_i(u - u_i)^2 + d_i(u - u_i)^3. \tag{1}$$

At the node-points u_i, S is continuous up to the third derivative, which is allowed to be discontinuous. To uniquely define the spline for a given n-point interpolation, a boundary condition is usually imposed on both the derivatives $S'''(u_1)$ and $S'''(u_n)$.

Thus to construct a spline we need

1. A list of $n \geq 2$ node-points u_i in strictly ascending order;[2]
2. A list of function values $f_i = f(u_i)$, with f a smooth function of u.

Input to a spline construction are an array of node-points in u (and v in case of two-dimensional splines)[3] and a user-defined function that returns f at each node-point. We emphasise that f must be a smooth function of u (and v), without discontinuities.

To spline QCDNUM output, SPLINT provides routines that take as nodes a sub-set of the evolution grid-points. Using the full QCDNUM grid produces an interpolation function that is similar to the one already built into QCDNUM itself. Because cubic splines are very good interpolators it is then a matter of simple tuning to find, for a given function f, a considerably reduced set of nodes that optimises speed versus accuracy. This is particularly interesting when f is relatively expensive to compute, like structure functions or cross-sections. Typically you can reduce a 100×100 grid to 20×20 nodes.

Cubic splines do not only provide compact parameter-free representations of discrete data but they also make it possible to integrate and differentiate these data. The present SPLINT release provides integration routines.

2 The SPLINT package

The SPLINT package is written in FORTRAN77 but interfaces are provided so that all FORTRAN routines can be called from a C++ program.

C++

> The C++ wrappers reside in the namespace `SPLINT` and the routine names are written in lower case, as is the QCDNUM convention. We refer to the QCDNUM manual for more on C++ interfaces.

The syntax of the SPLINT calls is as follows

`call xSP_NAME ( arguments )`	`SPLINT::xsp_name ( arguments );`

[1] `https://www.nikhef.nl/~h24/qcdnum`

[2] For $n = 2$ or 3 the spline routines return a simple linear or quadratic interpolation function.

[3] For 2-dimensional splines the coefficients f, b, c and d in (1) are splined in the second coordinate v.

where x = S for subroutines and x = L, I, R or D for logical, integer, real and double-precision functions, respectively. Floating-point arguments are in double precision and input numbers must, in FORTRAN, be given in double precision format like `2.5D0` instead of `2.5`. In C++ the input format is free since the data-type is specified in the function prototype and the conversion is done automatically, if necessary.

Spline routines will—as long as space allows—dynamically create a spline-object in internal memory and return an integer pointer `ia` to that spline. Note that `ia` is an internal memory address (array index) and not a C++ pointer.

The memory size is specified by the parameter `nw0` in the file `splint.inc`; if you run out of space (error message) then you must set `nw0` as needed, and recompile SPLINT.

The call `ivers = isp_SpVers()` gives you the current SPLINT version number.

3 Create spline objects

The first call in your program must be the initialisation of the SPLINT memory.[4]

`call ssp_SpInit(nuser)`	`ssp_spinit(nuser);`

Here `nuser` is the number of words to be reserved for user storage (see below).

The next step is to create a spline object in memory, which will be put at address `iasp`.

`iasp = isp_S2Make(istepx, istepq)`	`int iasp = isp_s2make(istepx, istepq);`

The arguments `istepx` and `istepq` are the steps taken in sampling the QCDNUM x-μ^2 evolution grid. Thus with `istep` = `5` we take every 5$^{\mathrm{th}}$ grid-point as a node-point of the spline. The boundaries of the grid are always included in the set of node points.

Finally we have to compute the spline coefficients and store these in the spline object.

`call ssp_S2Fill(iasp, fun, rsc)`	`ssp_s2fill(iasp, fun, rsc);`

The argument `rsc` sets a $\sqrt{s}$ cut described in Section 4; just set it to zero to have no kinematic cut when filling. The function `fun` should be declared `external` in FORTRAN and be coded as a function of the evolution grid points `ix` and `iq` as follows.

```
      double precision function fun(ix, iq, first)
      implicit double precision (a-h,o-z)
      logical first
      if(first) then
        code to initialise fun, if needed
      endif
      fun = some_function_of_ix_and_iq
      return
      end
```

[4]In the C++ code examples we will omit, for clarity, the scope resolution operator `SPLINT::`. We will also omit type declarations if they are clear from the context.

The C++ interface is such that the arguments of input functions must always be passed as pointers. Thus in C++ we have,

C++
```
double fun(int *ipx, int *ipq, bool *first)
if(*first) {
  code to initialise fun, if needed
  }
int ix = *ipx;
int iq = *ipq;
return some_function_of_ix_and_iq;
```

The function may need more input than is passed through the brackets. Additional parameters can be entered via a common block in FORTRAN, via some kind of getter function in C++, or via the user-space reserved in the call to `ssp_spinit`. The routines below give read/write access to this user store, if there is one (error message if not).

```
call ssp_Uwrite(i, val)
val = dsp_Uread(i)
```

```
ssp_uwrite(i, val);
double val = dsp_uread(i);
```

The index `i` runs from `1` to `nuser`, both in FORTRAN and C++.

Here is a C++ example where we pass the pdf-set number and pdf index to `fun`.

C++
```
//-------------------------------------------------
  double fun( int *ix, int *iq, bool *first )
    if(*first) {
      static int iset = int( dsp_uread(1) );
      static int ipdf = int( dsp_uread(2) );
    }
    return QCDNUM::fvalij(iset, ipdf, *ix, *iq, 1);
//-------------------------------------------------
    ..
    ssp_spinit(2);                //reserve 2 words for user storage
    int iasp = isp_s2make(5, 5);  //create spline object
    ssp_uwrite(1, dble(iset));    //write iset into user store
    ssp_uwrite(2, dble(ipdf));    //write ipdf into user store
    ssp_s2fill(iasp, fun, 0);     //compute spline coefficients
```

Note that `iset` and `ipdf` are declared `static` in the body of `fun` (this is the equivalent of a `save` statement in FORTRAN). Note also that the last argument of `fvalij` is set to `ichk` = 1 to generate a fatal error if you run outside QCDNUM kinematic cuts, if any. It is a good idea to protect yourself from this since it would corrupt the spline.

You can set your own node-points in case the automatic sampling needs some fine-tuning or cannot be used. For instance pdfs evolved in the VFNS are discontinuous in μ^2 so that you have to spline each threshold-region separately. A spline object with user-nodes is created with:[5]

```
iasp = isp_S2User(xarr, nx, qarr, nq)
```

[5] In the following we will omit the C++ calls—they should be fairly obvious now. The C++ prototypes of all routines are listed in Appendix C.

Here `xarr` (`qarr`) are double precision input arrays filled with `nx` (`nq`) node-point candidates. The routine will discard points outside the x-μ^2 evolution grid, round the remaining nodes down to the nearest grid-point and then sort them in ascending order, discarding equal values. Thus you are allowed to enter un-sorted scattered arrays.

In this way you can spline restricted regions in x and μ^2, or edit an existing set of node-points as we will show in an example below.

You can also construct a one-dimensional spline of x or μ^2 (*e.g.* spline an α_s table).

```
iasp =  isp_SxMake(istepx)
iasp =  isp_SxUser(xarr, nx)
call    ssp_SxFill(iasp, fun, iq)
```

```
iasp =  isp_SqMake(istepq)
iasp =  isp_SqUser(qarr, nq)
call    ssp_SqFill(iasp, fun, ix)
```

In `ssp_SxFill` (`ssp_SqFill`) the last argument `iq` (`ix`) is just passed to the input function `fun(ix,iq,first)` and kept fixed when computing the spline coefficients. If this input is not relevant you may of course chose to ignore it in the body of `fun`.

In the query routines below, `ia` is the spline address, `u` the first coordinate (x or μ^2 for 1-dim and x for 2-dim) and `v` the second coordinate (`0` for 1-dim and μ^2 for 2-dim).

Function	Description	
`isp_SplineType(ia)`	Type of spline `ia`	(1)
`ssp_SpLims(ia,nu,u1,u2,nv,v1,v2,n)`	Get node limits	(2)
`ssp_Unodes(ia,array,n,nu)`	Copy u-nodes to a local array	(3)
`ssp_Vnodes(ia,array,n,nv)`	Copy v-nodes to a local array	
`ssp_Nprint(ia)`	Print list of nodes and grid indices	(4)
`dsp_RsCut(ia)`	Get $\sqrt{s}$ cut	
`dsp_RsMax(ia,rsc)`	Get $\sqrt{s}$ cut limit (Section 4)	

(1) Spline types are: `-1` = `x`, `0` = not a spline, `+1` = `q` and `2` = 2-dim spline.

(2) In `n` is given the number of *active* nodes below the kinematic cut, if any.

(3) In `Unodes` (`Vnodes`), `n` is the dimension of `array` as declared in the calling routine and `nu` (`nv`) is the number of node-points copied (`nv` = `0` for a 1-dim spline).

(4) Also printed are, as node-indices, for each x-node the upper kinematic limit in μ^2, and for each μ^2-node the lower kinematic limit in x.

The `U`- and `Vnodes` routines are useful to fine-tune the automatic sampling, if desired:

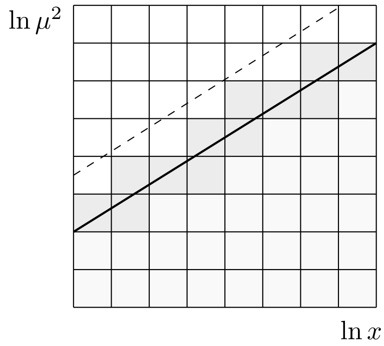
```
dimension xnodes(n)
..
ia_old    = isp_SxMake(istepx)           !spline with auto-sampled x-nodes
call ssp_Unodes(ia_old, xnodes, n, nx)   !get current list of x-nodes
xnodes(n) = 0.15D0                       !edit xnodes array (e.g. add a node)
ia_new    = isp_SxUser(xnodes, n)        !spline with modified set of nodes
```

You can add new node-points anywhere in `xnodes` because `SxUser` will sort them in ascending order, discarding double entries and anything outside the evolution grid. If you want `SxUser` to delete node-points, just set these to zero. Please enter the full array dimension `n` and not `nx`, otherwise the routine may not see your modifications.

4 Kinematic limit

Up to now we have assumed that the function to spline is defined on the entire QCDNUM evolution grid. This is of course true for pdfs and structure functions but not for DIS cross-sections which are un-defined above the kinematic limit $\mu^2 > xs$, with $\sqrt{s}$ the centre-of-mass energy of the collisions (about 300 GeV at HERA). A kinematic cut can be entered by setting the `rsc` argument of `ssp_S2Fill` to a non-zero value of $\sqrt{s}$.

The problem with a kinematic cut is that there are, around this cut, node-bins where the input function is not everywhere defined which implies that input at some corners of the bin is missing (dark-shaded bins in the figure). But to guarantee that the spline is well defined below the cut, input data must be available at all the corners of the dark-shaded bins.



The SPLINT package offers two ways to achieve this.

1. Set the `rsc` parameter to $-\sqrt{s}$ (at present not available). The dark-shaded bins will not be included in the determination of the spline coefficients and the spline will, in these bins, be estimated from an *extrapolation*. Spline extrapolation (see Section 5) can be unreliable so that it is better to use the option below, if possible.
2. Set the `rsc` parameter to $+\sqrt{s}$. The dark-shaded bins are included in the definition of the spline and it is the responsibility of the user to provide, in the input function, an extrapolation beyond the kinematic limit. Note that this extrapolation does not extend beyond the dark-shaded bins crossed by the cut.

A call to `dsp_RsCut(ia)` returns `rsc` (full line in the figure), and `dsp_RsMax(ia,rsc)` the limit `rsmax` (dashed line). Both routines return `0` if there is no cut. If the filling function reads another spline that has different nodes, you should set the cut on the source spline not lower than the `rsmax` of the target spline; this guarantees that the target spline does not suffer from missing input.

It is important to realise that `rsc` is a filling cut that may very well have been set above the actual value of $\sqrt{s}$, to `rsmax` for instance. For this reason the 2-dimensional integration routine in the next section has its own $\sqrt{s}$ input argument which must, of course, not be set above `rsc` (error message).

5 Spline function and integrals

For one- or two-dimensional splines with address `ia` the spline functions are

`val = dsp_FunS1(ia, u, ichk)`	`val = dsp_FunS2(ia, x, q, ichk)`

where `u` stands for x or μ^2, and `q` stands for μ^2, not μ. The argument `ichk` defines what happens when you venture outside the range of the spline, or access an empty node-bin above a $\sqrt{s}$ cut: `ichk` = `1` error message; `0` return a value of zero; `-1` extrapolate the spline. By default, a spline will extrapolate as a cubic polynomial but you can re-set this for a spline with address `ia` by calling one or both of the routines:[6]

`ssp_ExtrapU(ia, n)`	`ssp_ExtrapV(ia, n)`

which makes the extrapolation constant (`n` = `0`), linear (`1`), quadratic (`2`) or cubic (`3`).

Integrals can be computed with:

`val = dsp_IntS1(ia, u1, u2)`	`val = dsp_IntS2(ia, x1, x2, q1, q2, rs, np)`

The argument `rs` = $\sqrt{s}$ in the 2-dimensional integration routine imposes a kinematic limit $\mu^2 \leq xs$ (set `rs` = `0` to have no cut). If the integration domain is crossed by the limit, part of the integration is done with n-point Gauss quadrature (see Appendix A.3). You can fix n to `np` = `2`, `3`, or `4`; for `np` > `4` the adaptive n-point routine `dmb_dgauss` is called.[7] This MBUTIL routine is slower than the fixed-point routines but accurate to at least 10^{-7}. Integration over a domain that is fully beyond the limit is set to zero. If there is a filling cut `rsc` on the spline, the routine insists that `rs` ≤ `rsc` and also `rs` ≠ `0`.

6 Fast structure function input

Fast routines are provided to fill splines with structure functions. These routines exploit the capability of the ZMSTF package to create lists of structure functions which is much faster than computing them one-by-one in a loop, as is the case when you fill the spline with `ssp_SxFill`, `ssp_SqFill` or `ssp_S2Fill`. These fast filling routines are

Subroutine	Description
`ssp_SxF123(ia, iset, def, istf, iq)`	Fill x-spline with F_L, F_2, xF_3 or F'_L
`ssp_SqF123(ia, iset, def, istf, ix)`	Fill μ^2-spline
`ssp_S2F123(ia, iset, def, istf, rs)`	Fill 2-dim spline

Here `iset` is the QCDNUM pdf-set index, `def` an array of (anti-)quark coefficients and `istf` the structure function index `1` = F_L, `2` = F_2, `3` = xF_3 and `4` = F'_L. The coefficient array defines a linear combination of quarks and anti-quarks and must be declared `def(-6:6)` in FORTRAN or `def[13]` in C++.[8]

Note that in a full cross-section calculation the electroweak charges are Q^2-dependent and that this cannot be accommodated in `def`. Here one must create structure function splines separately for the sum of up-type and down-type flavours and multiply these afterwards by the appropriate electroweak charges for up and down, respectively.

[6]Calling `extrapv` on a one-dimensional spline is allowed but has no effect.

[7]For `np` < `2` the bins crossed by the cut are fully included in the integral, disregarding the cut.

[8]The indexing of `def` is given by

	$\bar{t}$	$\bar{b}$	$\bar{c}$	$\bar{s}$	$\bar{u}$	$\bar{d}$	g	d	u	s	c	b	t
C++	0	1	2	3	4	5	6	7	8	9	10	11	12
FORTRAN	−6	−5	−4	−3	−2	−1	0	1	2	3	4	5	6

To investigate the timing we filled a 20×10-node spline with NNLO F_2 using the `S2Fill` and `S2F123` routines. The table shows a large CPU gain for the fast filling routine.

Fill routine	ZMSTF routine	t [ms]
`ssp_S2Fill`	`zmstfun`	110
`ssp_S2Fill`	`zmstfij`	33
`ssp_S2F123`		1

Because spline interpolation is fast it often pays to invest about 1–2 ms CPU to create a spline and obtain the structure function at any x and Q^2 from interpolation instead of directly from ZMSTF; see the ZMSTF write-up for a more detailed timing study.

7 Handle spline objects

Spline objects are stored—one after another[9]—in a (hidden) double precision array of size `nw0` as is specified in the file `splint.inc`. If you run out of space (error message) then SPLINT must be re-compiled with an increased value of `nw0`.

Object sizes (in words) are accessed by a call to

```
nw = isp_SpSize(ia)
```

which returns the total memory size when `ia` = `0`, the used space when `ia` = `1`, the size of a spline object when `ia` is its address, or zero when `ia` has some other value.

You cannot delete a single spline but you can clear the memory from address `ia` onwards:

```
call ssp_Erase(ia)
```

Here `ia` must be a valid spline address—error message otherwise—except that you can enter `ia` = `0`, instead of the first spline address, to erase all spline objects in memory.

To write a spline `ia` to disk, or to read it back at address `ia`, use

```
call ssp_SpDumnp(ia, 'filename')        ia = isp_SpRead('filename')
```

In FORTRAN, `filename` must be a `character*n` variable or be given as a literal string embedded in single quotes. In C++ it must be a string variable or a literal string embedded in double quotes. Note that a file can contain only *one* spline object. Note also that spline creation and filling are coupled to QCDNUM, but the spline object itself is not. Thus a dumped spline can always be read-back whatever the QCDNUM settings but cannot be re-filled (error message) because its nodes might not line-up anymore with the QCDNUM grid. Finally note that a SPLINT file may become obsolete (error message) after an upgrade of SPLINT or of the memory manager WSTORE.

The routines enable you to dump pdfs from a QCDNUM run with very high grid densities—which is expensive—and then read these back as a high-accuracy reference to tune the QCDNUM grids for optimal performance.

You can also do some kind of garbage collection by saving your favourite splines to disk,

[9] To see the memory layout you can call `ssp_Mprint()` to print a dump on the standard output.

clean the memory with `ssp_Erase(0)`, and then read the splines back-in.

Before you write a spline to disk it may be useful to store some extra information like a particle code, QCDNUM evolution parameters, *etc.* To read/write into a spline `ia` use

`call ssp_SpSetVal(ia, i, val)`	`val = dsp_SpGetVal(ia, i)`

The index `i` runs from `1` to `100` (error if out of range), depending on the value of `nusr0` in `splint.inc`. The call `dsp_SpGetVal(0,0)` returns, as a double precision number, the current value of `nusr0`. Do not confuse these routines with `ssp_Uwrite/read` which access a storage space in memory that is common to all splines.

Here is an example that (re-)initialises the storage of spline `ia` (usually not necessary) and then stores the current QCDNUM evolution parameters. Note the indexing of `pars` in the C++ code.

```
dimension pars(13)
n = int( dsp_SpGetVal(0,0) )
do i = 1,n
  call ssp_SpSetVal(ia, i, 0.D0)
enddo
..
call cpypar(pars, 13, 0)
do i = 1,13
  call ssp_SpSetVal(ia, i, pars(i))
enddo
```

```
double pars[13];
int n = int( dsp_spgetval(0,0) );
for( int i=1; i<=n; i++ ) {
  ssp_spsetval(ia, i, 0);
  }
..
QCDNUM::cpypar( pars, 13, 0);
for( i=1; i<=13; i++ )  {
  ssp_spsetval(ia, i, pars[i-1]);
  }
```

A Spline integration

In SPLINT a function F of x and μ^2 is approximated as a cubic spline in the logarithmic variables $y = -\ln x$ and $t = \ln \mu^2$,

$$F(x, \mu^2) = S(-\ln x, \ln \mu^2) = S(y, t).$$

This gives for the integral

$$\int_{x_1}^{x_2} \int_{\mu_1^2}^{\mu_2^2} F(x, \mu^2)\, \mathrm{d}x \mathrm{d}\mu^2 = \int_{y_1}^{y_2} \int_{t_1}^{t_2} e^{-y} e^{t}\, S(y, t)\, \mathrm{d}y \mathrm{d}t$$

where $y_{1,2} = -\ln x_{2,1}$ and $t_{1,2} = \ln \mu^2_{1,2}$. Note that the upper (lower) limit of x becomes the lower (upper) limit of y. When the lower integration limit does not coincide with a node-point the spline is, in some cases, locally re-parameterised with respect to (y_1, t_1).

In the following sections we will first describe spline re-parameterisation and then present the integration in one and two dimensions.

A.1 Re-parameterisation

In the bin $u_i \leq u < u_{i+1}$ a one-dimensional spline of $u = y$ or t can be written as

$$S(u) = \sum_{n=0}^{3} A_n\, (u - u_i)^n.$$

Here the A_n stand for the coefficients (f, b, c, d) of (1) in Section 1.

The k^{th} differential quotient is given by

$$\frac{\mathrm{d}^k S(u)}{\mathrm{d}u^k} \equiv D_k(u) = \sum_{n=k}^{3} A_n\, \frac{n!}{(n-k)!}\, (u - u_i)^{n-k} \quad \text{with} \quad 0 \leq k \leq 3.$$

We now re-parameterise the spline with respect to a reference point u_i' inside the bin. The spline and its derivatives are invariant under such a transformation so that the coefficients A' of the new parameterisation must satisfy the equations, for $0 \leq k \leq 3$,

$$\sum_{n=k}^{3} A_n'\, \frac{n!}{(n-k)!}\, (u - u_i')^{n-k} = D_k(u).$$

Setting $u = u_i'$ in these equations we find

$$A_n' = \frac{1}{n!}\, D_n(u_i'), \qquad 0 \leq n \leq 3.$$

For a 2-dimensional spline in the bin $y_i \leq y < y_{i+1},\ t_j \leq t < t_{j+1}$ we have

$$S(y, t) = \sum_{n=0}^{3} \sum_{m=0}^{3} A_{nm}\, (y - y_i)^n (t - t_j)^m$$

and

$$\frac{\mathrm{d}^{k+l} S(y,t)}{\mathrm{d}y^k \mathrm{d}t^l} \equiv D_{kl}(y,t) = \sum_{n=k}^{3} \sum_{m=l}^{3} A_{nm} \, \frac{n!}{(n-k)!} \, \frac{m!}{(m-l)!} \, (y-y_i)^{n-k} (t-t_j)^{m-l}.$$

Transforming to a reference point (y'_i, t'_j) within the bin we get for the new coefficients

$$A'_{nm} = \frac{1}{n!\, m!} D_{nm}(y'_i, t'_j), \qquad 0 \le n \le 3, \quad 0 \le m \le 3.$$

A.2 Integration in one dimension

In the bin $y_i \le y < y_{i+1}$ the integral of a one-dimensional spline $S(y)$ is given by

$$I_i(y) \equiv \int_{y_i}^{y} e^{-u} S(u) \, \mathrm{d}u.$$

Transforming to a local y-coordinate $\xi = y - y_i$ this can be written as

$$I_i(y) = e^{-y_i} \int_0^{\xi} e^{-u} S(u) \, \mathrm{d}u = e^{-y_i} \sum_{n=0}^{3} A_n^i \int_0^{\xi} u^n e^{-u} \, \mathrm{d}u = e^{-y_i} \sum_{n=0}^{3} A_n^i \, E^-(\xi, n),$$

with $E^-(x,n) = \int_0^x z^n e^{-z} \, \mathrm{d}z$.[10]
Likewise integration of a one-dimensional spline $S(t)$ over a bin in t is given by

$$J_i(t) \equiv \int_{t_i}^{t} e^{v} S(v) \, \mathrm{d}v = e^{t_i} \sum_{n=0}^{3} A_n^i \, E^+(\eta, n).$$

Here η is the local coordinate $t - t_i$ and $E^+(x,n) = \int_0^x z^n e^{+z} \, \mathrm{d}z$.
Repeated partial integration gives simple recursion relations for the $E^{\pm}(x,n)$:

$$\begin{array}{lcl|lcl} E^-(x,0) & = & 1 - e^{-x} & E^+(x,0) & = & e^x - 1 \\ E^-(x,n) & = & nE^-(x,n-1) - x^n e^{-x} & E^+(x,n) & = & x^n e^x - nE^+(x,n-1) \end{array}$$

Let y_a and y_b be integration limits of y, lying inside the node-bins with indices $i = a$ and $i = b$, respectively. The integral of $F(x)$ over the interval $x_{a,b} = \exp(-y_{b,a})$ can then be computed from

$$\int_{x_a}^{x_b} F(z) \, \mathrm{d}z = \sum_{j=a}^{b-1} I_j(y_{j+1}) + I_b(y_b) - I_a(y_a),$$

with a similar expression for integrals over μ^2.

[10] In fact $E^-(x,n) = \gamma(n+1, x)$ with γ the incomplete gamma function.

A.3 Integration in two dimensions

In two dimensions the spline factorises into piecewise cubic polynomials of y and t. In the node-bin (i, j) this can be written as

$$S_{ij}(y,t) = \sum_{n=0}^{3}\sum_{m=0}^{3} A^{ij}_{nm}\, \xi^n\, \eta^m,$$

where $\xi = y - y_i$ and $\eta = t - t_i$ are the local bin-coordinates. In analogy with the one-dimensional case we find for the 2-dimensional integral within the bin

$$I_{ij}(y,t) = \int_{y_i}^{y}\int_{t_j}^{t} e^{-u} e^{v}\, S_{ij}(u,v)\,\mathrm{d}u\mathrm{d}v = e^{-y_i}\, e^{t_j} \sum_{n=0}^{3}\sum_{m=0}^{3} A^{ij}_{nm}\, E^{-}(\xi,n)\, E^{+}(\eta,m).$$

In the left picture of Figure 1 we show the integration $I_{ij}(y,t)$ which always starts at

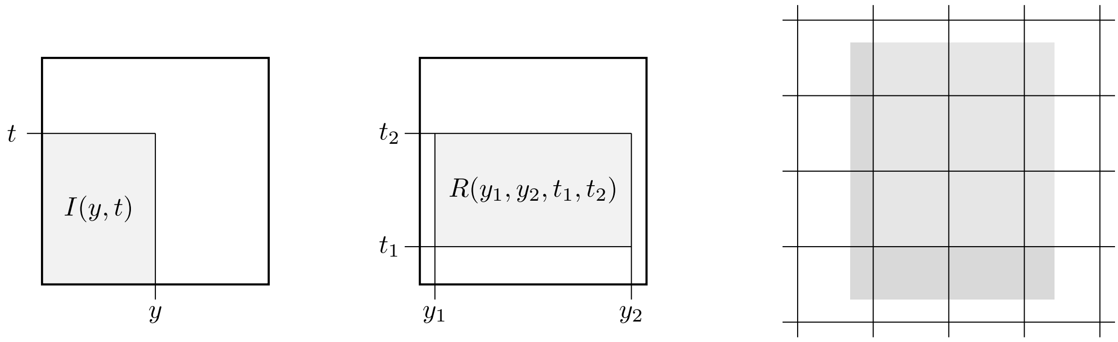


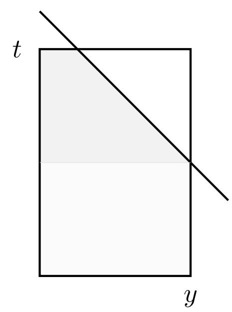
Figure 1 – Two-dimensional integration inside a node-bin and over a collection of node-bins

the lower left-hand corner of a node-bin. To integrate over a rectangle inside the bin (middle picture of Figure 1) we first transform the spline to the reference point (y_1, t_1), as is described in Appendix A.1, and compute

$$R_{ij}(y_1,y_2,t_1,t_2) = I'_{ij}(y_2 - y_1, t_2 - t_1),$$

where the prime indicates a prior spline coefficient transformation.[11]

The function R_{ij} is of course slower than I_{ij} but it only needs to be called in the dark-shaded area of the right picture of Figure 1. In the light-shaded area it is sufficient to call I_{ij}.



It may happen that a kinematic limit runs across an integration domain. If this is the case the light-shaded rectangle in the figure on the right is integrated as is described above. In the dark-shaded area the integral over y up to the limit is written as a function of t which is then integrated using 2, 3, 4 or n-point (your choice) Gaussian quadrature.

[11] For bins not crossed by a cut SPLINT uses a faster algorithm:

$$R_{ij}(y_1,y_2,t_1,t_2) = I_{ij}(y_2,t_2) - I_{ij}(y_1,t_2) - I_{ij}(y_2,t_1) + I_{ij}(y_1,t_1).$$

B List of FORTRAN routines

Function	Description
	Initialisation
`isp_SpVers()`	Get SPLINT version number
`ssp_SpInit(nuser)`	Initialise SPLINT memory
	Spline in two dimensions
`isp_S2Make(istepx, istepq)`	Create spline object (auto nodes)
`isp_S2User(xarr, nx, qarr, nq)`	Create spline object (user nodes)
`ssp_S2Fill(ia, fun, rsc)`	Create spline coefficients
`ssp_S2F123(ia, iset, def, istf, rsc)`	Enter structure function
`dsp_FunS2(ia, x, q, ichk)`	2-dim spline function
`dsp_IntS2(ia, x1, x2, q1, q2, rs, np)`	2-dim spline integration
	Spline in one dimension
`isp_Sx|qMake(istep)`	Create spline object (auto)
`isp_Sx|qUser(array, n )`	Create spline object (user)
`ssp_SxFill(ia, fun, iq)`	Create x-spline coefficients
`ssp_SxF123(ia, iset, def, istf, iq)`	Enter structure function
`ssp_SqFill(ia, fun, ix)`	Create μ^2-spline coefficients
`ssp_SqF123(ia, iset, def, istf, ix)`	Enter structure function
`dsp_FunS1(ia, u, ichk)`	1-dim spline function
`dsp_IntS1(ia, u1, u2)`	1-dim spline integration
	Extrapolation
`ssp_ExtrapU(ia, n)`	Set degree u-extrapolation
`ssp_ExtrapV(ia, n)`	Set degree v-extrapolation
	User store
`ssp_Uwrite(i, val)`	Write user store
`dsp_Uread(i)`	Read user store
	Spline info
`isp_SplineType(ia)`	Type of spline `ia`
`ssp_SpLims(ia, nu, u1, u2, nv, v1, v2, n)`	Get node limits
`ssp_Unodes(ia, array, n, nu)`	Copy u-nodes
`ssp_Vnodes(ia, array, n, nv)`	Copy v-nodes
`ssp_Nprint(ia)`	Print list of nodes
`dsp_RsCut(ia)`	Get `rs` cut
`dsp_RsMax(ia, rsc)`	Get `rs` limit (Section 4)
	Spline object handling
`isp_SpSize(ia)`	Get object size
`ssp_Erase(ia)`	Clear memory
`ssp_SpDump(ia,'filename')`	Write spline to disk
`isp_SpRead('filename')`	Read spline from disk
`ssp_SpSetVal(ia, i, val)`	Write info into a spline
`dsp_SpGetVal(ia, i)`	Read info from a spline

C List of C++ prototypes

	Initialisation
int	isp_spvers()
void	ssp_spinit(int nuser)
	Spline in two dimensions
int	isp_s2make(int istepx, int istepq)
int	isp_s2user(double *xarr, int nx, double *qarr, int nq)
void	ssp_s2fill(int ia, double (*fun)(int*,int*,bool*), double rsc)
void	ssp_s2f123(int ia, int iset, double *def, int istf, double rsc)
double	dsp_funs2(int ia, double x, double q, int ichk)
double	dsp_ints2(int ia, double x1, double x2,
	double q1, double q2, double rs, int np)
	Spline in one dimension
int	isp_sx|qmake(int istep)
int	isp_sx|quser(double* array, int n)
void	ssp_sxfill(int ia, double (*fun)(int*,int*,bool*), int iq)
void	ssp_sxf123(int ia, int iset, double *def, int istf, int iq)
void	ssp_sqfill(int ia, double (*fun)(int*,int*,bool*), int ix)
void	ssp_sqf123(int ia, int iset, double *def, int istf, int ix)
double	dsp_funs1(int ia, double u, int ichk)
double	dsp_ints1(int ia, double u1, double u2)
	Extrapolation
void	ssp_extrapu(int ia, int n)
void	ssp_extrapv(int ia, int n)
	User store
void	ssp_uwrite(int i, double val)
double	dsp_uread(int i)
	Spline info
int	isp_splinetype(int ia)
void	ssp_splims(int ia, int &nu, double &u1, double &u2,
	int &nv, double &v1, double &v2, int &n)
void	ssp_unodes(int ia, double *array, int n, int &nu)
void	ssp_vnodes(int ia, double *array, int n, int &nv)
void	ssp_nprint(int ia)
double	dsp_rscut(int ia)
double	dsp_rsmax(int ia, double rsc)
	Spline object handling
int	isp_spsize(int ia)
void	ssp_erase(int ia)
void	ssp_spdump(int ia, string filename)
int	isp_spread(string filename)
void	ssp_spsetval(int ia, int i, double val)
double	dsp_spgetval(int ia, int i)