1 Interpolation of unbounded derivatives of L_α functions

The purpose of this paper is to test the quality of approximation of $f(x)$ and $f'(x)$ on [0,1] using the sinc series

$$\sum_{k=-N}^{N} c_k S(k,h) \circ \phi(x_j) \tag{1}$$

but with the c_k obtained by setting up and solving the collocation system

$$\sum_{k=-N}^{N} c_k \left[S(k,h) \circ \phi(x_j)\right]' = f'(x_j) \tag{2}$$

instead of using simple interpolation of f at sinc points. The advantage, if sucessful, comes in the solution of differential equations for which derivative boundary conditions are specified even though the derivatives are known to be unbounded at certain points. For boundaries with unbounded normal derivatives, one can then use the series

$$\sum_{k=-N}^{N} c_k \left[S(k,h) \circ \phi(x_j)\right]' \tag{3}$$

rather than

$$\sum_{k=-N}^{N} c_k \left[S(k,h) \circ \phi(x_j)\right] \tag{4}$$

in a sinc-spline product for approximation of the derivative.

1.1 The Code

The code will consist of

1. a MAPLE file to generate needed C++ subroutines,
2. the main C++ program to set up and solve the system,
3. a second C++ program to plot the series and derivative.

To smoothly integrate the three and the MAPLE output, a Makefile will be used[1].

Starting with the main setup loop implementing

$$\sum_{k=-N}^{N} c_k \left[S(k,h) \circ \phi(x_j)\right]' = f'(x_j) \tag{5}$$

[1]To extract needed pieces from this noweb source, the makefile (notice the lowercase m) is used.

where the x_j are the sinc points, $j = -M..N$, we have

2a ⟨*c setup: main loop* 2a⟩≡
```
{
  //    c setup: main loop
  int k,j;
  for (j=-M;  j<=N;  j++) {
    for (k=-M;  k<=N;  k++){
      A_PUT_ENTRY(j+M, k+M, skh_pri(k, h, xj(j) ));
    }
    B_PUT_ENTRY(j+M, f_pri( xj(j) ));
  }     // for j
}
```
Uses `A_PUT_ENTRY` 2b and `B_PUT_ENTRY` 2b.
This code is used in chunk 12a.

This code segment requires some definitions and functions. First, the matrix and rhs vector, and the access macros.

2b ⟨*c setup: matrix and rhs* 2b⟩≡
```
  //    c setup: matrix and rhs
#define A_PUT_ENTRY(m_j, m_k, m_x)\
  if ((m_j < 0) || (m_j >= A->m)) {\
    cout << "row index out of range at line " << __LINE__ << " in file "\
      __FILE__ << ".  Aborting." << endl;\
        exit(1); }\
  if ((m_k < 0) || (m_k >= A->n)) {\
    cout << "column index out of range at line " << __LINE__ << " in file "\
      __FILE__ << ".  Aborting." << endl;\
        exit(1); }\
  A->me[m_j][m_k] = m_x
#define B_PUT_ENTRY(m_j, m_x)\
  if ((m_j < 0) || (m_j >= B->dim)) {\
    cout << "row index out of range at line " << __LINE__ << " in file "\
      __FILE__ << ".  Aborting." << endl;\
        exit(1); }\
  B->ve[m_j] = m_x

  MAT *A;       // define
  VEC *B;
  A = m_get(m,m);       // allocate memory
  B = v_get(m);
```
Defines:
`A_PUT_ENTRY`, used in chunk 2a.
`B_PUT_ENTRY`, used in chunk 2a.
This code is used in chunk 12a.

For these definitions, we further require some include files:

3a ⟨*c setup: include files* 3a⟩≡
```
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <math.h>

extern "C" {
#include        "matrix2.h"
#include        "matlab.h"
}
```
This definition is continued in chunk 7c.
This code is used in chunks 12a and 13b.

Next, the global variables h, N and m. N, M and $alpha$[2] are read from the file `unbndd-1d-ipl-global-vars.in`, h and m are calculated. The global definition:

3b ⟨*c setup: global variable declaration* 3b⟩≡
```
//      c setup: global variable declaration
double alpha, h, pi;
int M, N, m;
```
This code is used in chunks 12a and 13b.

And the value assignment:

3c ⟨*c setup: global variable assignment* 3c⟩≡
```
//      c setup: global variable assignment

  pi = 3.14159265358979323846264338328;

  FILE *fp;
  if ((fp = fopen("unbndd-1d-ipl-global-vars.in", "r")) == NULL)
    {
      printf("unable to open input file %s \n", "unbndd-1d-ipl-global-vars.in");
      exit(1);
    }
  fscanf(fp, "%lf %d %*[^\n]", &(alpha), &(N));
  fscanf(fp, "%d %*[^\n]",  &(M));
  fclose(fp);

  h = pi/sqrt(alpha*(double)(N));
  m = M+N+1;
```
This code is used in chunks 12a and 13b.

[2]The enpoint behavior, and the grid spacing factor

For other files, the global variables will be external; we can put the declarations in a common header file, along with other needed things:

4a ⟨*unbndd-1d-ipl-stinc.h* 4a⟩≡

```
#ifndef MAIN_PROG
extern double alpha, h;
extern int N, M, m;
#endif

#include <math.h>
```

This definition is continued in chunks 7b, 8b, and 10a.
Root chunk (not used in this document).

The input file thus will look like

4b ⟨*unbndd-1d-ipl-global-vars.in* 4b⟩≡

```
1.0   10        alpha N
11              M
```

Root chunk (not used in this document).

Lastly, the functions `x(j)`, `skh_pri(k,h,x)`, `f_pri(x)` and associated ones. These will be generated by MAPLE and put into the file `unbndd-1d-ipl-setup-subs.cc`. Their prototypes go into `unbndd-1d-ipl-setup-subs.h`.

4c ⟨*unbndd-1d-ipl-maple-setup.map* 4c⟩≡

```
#
Digits := 16;
interface(errorbreak = 2);      # abort on error
read `sinc8.mas`;

⟨maple subs setup: define some needed derivatives 10b⟩

⟨maple subs setup: define functions 8c⟩

⟨maple subs setup: generate C++ code and header files 9⟩
```

Root chunk (not used in this document).

And here are the invokations used in the Makefile to obtain the generated C++ code:

5 ⟨*Makefile: get C++ setup subs from MAPLE* 5⟩≡

```
unbndd-1d-ipl-setup-subs.cc:    unbndd-1d-ipl-maple-setup.map unbndd-1d-ipl-setup-subs-1.
        (echo "gc(0);"; echo "words(0);"; \
        cat unbndd-1d-ipl-maple-setup.map ; echo "quit;") | maple > $@.mout

unbndd-1d-ipl-setup-subs.h:    unbndd-1d-ipl-maple-setup.map  unbndd-1d-ipl-setup-subs-1.
        (echo "gc(0);"; echo "words(0);"; \
        cat unbndd-1d-ipl-maple-setup.map ; echo "quit;") | maple > $@.mout
```

This code is used in chunk 12b.

To keep the size of generated code to a minimum, the above maple routines do not expand ϕ, ψ or x_j. These functions are done separately by MAPLE and are put into `unbndd-1d-ipl-setup-subs-1.cc`; their prototypes go into `unbndd-1d-ipl-setup-subs-1.h`.

6 ⟨*unbndd-1d-ipl-maple-setup-1.map* 6⟩≡

```
Digits := 16;
interface(errorbreak = 2);      # abort on error
read ‘sinc8.mas‘;

phi := x -> ln((x)/(1-x));      # from [0,1] to R
z = phi(u):
solve(", u):
psi := unapply(", z);           # and the inverse

phi_p := x-> diff(phi(x),x):
psi_p := x-> diff(psi(x),x):

psi(j*h);
xj := unapply(", j);

cff:= proc(expr);               # factor, convert to float (no factor)
   convert((expr), float);
end:
if (macroc_loaded <> true) then;
   read ‘macroc‘;
fi;
optimized := true;
precision := double;
autodeclare := double;

OUTPUT_FILE := ‘unbndd-1d-ipl-setup-subs-1.cc‘;
PROTOTYPE_FILE := ‘unbndd-1d-ipl-setup-subs-1.h‘;
WHICH_UNKNOWN := ‘ ‘;

prog := [
      [textC, ‘#include "unbndd-1d-ipl-stinc.h"‘]
      ];
start_fun(prog);
prot := [textC, ‘// prototypes from unbndd-1d-ipl-maple-setup-1.map‘];
start_proto(prot);

function_list := [
      [‘xj‘, ‘j‘, ‘[[int, j]]‘],
```

```
        [`phi`, `x`, `[[double, x]]`],
        [`psi`, `x`, `[[double, x]]`],
        [`psi_p`, `x`, `[[double, x]]`],
        [`phi_p`, `x`, `[[double, x]]`]
        ];
  gen_func(function_list);
  gen_proto(function_list);
  interface(errorbreak = 1);
```

Root chunk (not used in this document).

To actually get the C++ code out of this MAPLE listing, the following Makefile entry is used:

7a ⟨*Makefile: get C++ setup subs 1 from MAPLE* 7a⟩≡

```
unbndd-1d-ipl-setup-subs-1.cc:    unbndd-1d-ipl-maple-setup-1.map
        (echo "gc(0);"; echo "words(0);"; \
        cat $? ; echo "quit;") | maple > $@.mout
unbndd-1d-ipl-setup-subs-1.h:    unbndd-1d-ipl-maple-setup-1.map
        (echo "gc(0);"; echo "words(0);"; \
        cat $? ; echo "quit;") | maple > $@.mout
```

This code is used in chunk 12b.

To ensure inclusion of the prototypes generated above:

7b ⟨*unbndd-1d-ipl-stinc.h* 4a⟩+≡

```
#include "unbndd-1d-ipl-setup-subs-1.h"
```

This will also be needed in the main C code. Thus, add it:

7c ⟨*c setup: include files* 3a⟩+≡

```
#include "unbndd-1d-ipl-stinc.h"
```

The extra MAPLE commands keep messages from corrupting the generated code. The only functions left to make are the sinc functions and their derivatives. Because this requires some care for small argument values[3], these routines are done manually:

8a ⟨*c setup: sinc functions/derivatives* 8a⟩≡

```
double sinc(double x)
{
  if (fabs(x) < 0.004)
    return 1.0-PI*PI*x*x/6+pow(PI,4.0)*pow(x,4.0)/120;
  else
    return sin(PI*x)/(PI*x);
}

double sinc_p(double x)          // first derivative
{
  if (fabs(x) < 0.004)
    return -PI*PI*x/3+pow(PI,4.0)*x*x*x/30-pow(PI,6.0)*pow(x,5.0)/840;
  else
    return cos(PI*x)/x-sin(PI*x)/PI/(x*x);
}
```

This code is used in chunks 12a and 13b.

To make these definitions available globally, include the prototypes in

8b ⟨*unbndd-1d-ipl-stinc.h* 4a⟩+≡

```
double sinc(double x);
double sinc_p(double x);
```

In the following, we define all functions needed in the main system loop.

8c ⟨*maple subs setup: define functions* 8c⟩≡

```
skh := unapply(sinc((phi(x)-k*h)/h), (k,h,x));
skh_pri := unapply(diff(skh(k,h,x), x), (k,h,x));

f := unapply(x**(1/2)*(1-x), x);
f_pri := unapply(diff(f(x),x), x);
```

This code is used in chunk 4c.

[3]For x near zero, sinc(x) is best evaluated using a taylor series.

Next, we generate the C++ code and headers; the listing is followed by the appropriate Makefile entry to invoke maple.

9 ⟨*maple subs setup: generate C++ code and header files* 9⟩≡

```
#       Preliminaries:
#
OUTPUT_FILE := `unbndd-1d-ipl-setup-subs.cc`;
PROTOTYPE_FILE := `unbndd-1d-ipl-setup-subs.h`;
WHICH_UNKNOWN := ` `;
#
cff:= proc(expr);               # factor, convert to float (no factor)
   convert((expr), float);
end:
if (macroc_loaded <> true) then;
   read `macroc`;
fi;
optimized := true;
precision := double;
autodeclare := double;

#       Subroutines:
#       ~~~~~~~~~~~~
#
#       startup:
prog := [
      [textC, `#include "unbndd-1d-ipl-stinc.h"`]
      ];
start_fun(prog);
prot := [textC, `// prototypes from unbndd-1d-ipl-maple-setup.map`];
start_proto(prot);

function_list := [
      [`f`, `x`, `[[double, x]]`],
      [`f_pri`, `x`, `[[double, x]]`],
      [`skh`, `k,h,x`, `[[int, k], [double, h], [double, x]]`],
      [`skh_pri`, `k,h,x`, `[[int, k], [double, h], [double, x]]`]
      ];
gen_func(function_list);
gen_proto(function_list);
interface(errorbreak = 1);
```

This code is used in chunk 4c.

Since the prototypes generated above will be needed by most files, add them to

10a ⟨*unbndd-1d-ipl-stinc.h* 4a⟩+≡

```
#include "unbndd-1d-ipl-setup-subs.h"
```

10b ⟨*maple subs setup: define some needed derivatives* 10b⟩≡

```
# Defs for derivatives (psi_p(), etc.)
`diff/psi` := proc(a,x);
   psi_p(a)*diff(a,x);
end:
`diff/phi` := proc(a,x);
   phi_p(a)*diff(a,x);
end:
`diff/psi_p` := proc(a,x);
   psi_pp(a)*diff(a,x);
end:
`diff/phi_p` := proc(a,x);
   phi_pp(a)*diff(a,x);
end:
```

This code is used in chunk 4c.

With the interpolation/collocation matrix thus set up, it only remains to solve the system and save the answer vector obtained. Thus,

10c ⟨*c setup: solve system* 10c⟩≡

```
  PERM  *pivot;
  VEC *u_vec;
  pivot = px_get(m);
  LUfactor(A, pivot);
  u_vec = LUsolve(A, pivot, B, VNULL);
```

This code is used in chunk 12a.

The vector obtained will be saved in the file unbndd-1d-ipl-answer.mat:

11a ⟨*c setup: save answer vector* 11a⟩≡
```
{
  FILE *fp;
  if ( (fp=fopen("unbndd-1d-ipl-answer.mat","w")) == (FILE *) NULL ) {
    printf("Cannot open save file \"%s\"\n", "unbndd-1d-ipl-answer.mat");
    exit(1);
  }

  v_save(fp, u_vec, "u_vec");

  fclose(fp);

}
```
This code is used in chunk 12a.

It may be useful to have the matrix and right hand side available in matlab; for this, we use

11b ⟨*c setup: save matrix and rhs* 11b⟩≡
```
{
  FILE *fp;
  if ( (fp=fopen("unbndd-1d-ipl-A_and_b.mat","w")) == (FILE *) NULL ) {
    printf("Cannot open save file \"%s\"\n", "unbndd-1d-ipl-A_and_B.mat");
    exit(1);
  }

  v_save(fp, B, "b");
  m_save(fp, A, "A");

  fclose(fp);
}
```
This code is used in chunk 12a.

We thus have the following main program:

12a ⟨*unbndd-1d-ipl-setup.cc* 12a⟩≡
```
#define MAIN_PROG
⟨c setup: include files 3a⟩
⟨c setup: global variable declaration 3b⟩
⟨c setup: sinc functions/derivatives 8a⟩
int main()  {
⟨c setup: global variable assignment 3c⟩
⟨c setup: matrix and rhs 2b⟩
⟨c setup: main loop 2a⟩
⟨c setup: save matrix and rhs 11b⟩
⟨c setup: solve system 10c⟩
⟨c setup: save answer vector 11a⟩
}
```
Root chunk (not used in this document).

To properly run the MAPLE code and compile the subsequent C++ code, a Makefile is put in order:

12b ⟨*Makefile* 12b⟩≡
```
cc = gcc

CFLAGS = -g -I/root/meschach
LIB = /root/meschach/meschach.a -lm

.SUFFIXES:      .map .cc .h

⟨Makefile: get C++ setup subs from MAPLE 5⟩

⟨Makefile: get C++ setup subs 1 from MAPLE 7a⟩

#       main program:
D1 = unbndd-1d-ipl-setup.o unbndd-1d-ipl-setup-subs-1.o \
        unbndd-1d-ipl-setup-subs.o
unbndd-1d-ipl-setup.exe:       $(D1)
        $(CXX) -o $@ $(LFLAGS) $(D1) $(LIB)

.cc.o:
        $(CXX) -c $*.cc $(CFLAGS)
```

This definition is continued in chunk 14.
Root chunk (not used in this document).

The above will set up and solve the system for equation (2). It now remains to use the answer vector obtained to calculate the interpolant for f and f' at arbitrary points in the domain. This will be most flexible if done in two steps: (a) making a list of plot points in MATLAB (or a clone thereof) and saving it in a file; (b) running a C++ program to read the list, calculate all functions, and save results in MATLAB-readable format for plotting etc.

A simple OCTAVE code to save the plot points, run the plot program and read the results is

13a ⟨*Unbndd1dIplPlotInput.m* 13a⟩≡

```
%       save plot points as column vector
x = [0.01:0.01:0.99]';
save -mat-binary unbndd-1d-ipl-plot-data.mat x

%       do the plotting
system("unbndd-1d-ipl-plot.exe")

%       read output
load -force unbndd-1d-ipl-plot-results.mat
%       release 1.1.1 of octave has braindamage reading matlab files
%       in row-major order, so adjust results:
results = reshape(results, size(results, 2), size(results, 1))';
```

Uses `points` 14c.
Root chunk (not used in this document).

The C++ code to do the actual work will have a structure very similar to the setup program:

13b ⟨*unbndd-1d-ipl-plot.cc* 13b⟩≡

```
#define MAIN_PROG
⟨c setup: include files 3a⟩
⟨c setup: global variable declaration 3b⟩
⟨c setup: sinc functions/derivatives 8a⟩
int main()  {
⟨c setup: global variable assignment 3c⟩
⟨c plot: read plot points 14c⟩
⟨c plot: main loop 15⟩
⟨c plot: save results 16b⟩
}
```

Root chunk (not used in this document).

Thus, only three more code segments are required. But first, the Makefile entry:

14a ⟨*Makefile* 12b⟩+≡
```
#
#       plotting program:
DPLOT = unbndd-1d-ipl-plot.o unbndd-1d-ipl-setup-subs-1.o \
        unbndd-1d-ipl-setup-subs.o
unbndd-1d-ipl-plot.exe: $(DPLOT)
        $(CXX) -o $@ $(LFLAGS) $(DPLOT) $(LIB)
#
```

Also, the names are somewhat long, so add an "all" target:

14b ⟨*Makefile* 12b⟩+≡
```
#
all:    unbndd-1d-ipl-plot.exe unbndd-1d-ipl-setup.exe
```

Now the remaining code chunks.

14c ⟨*c plot: read plot points* 14c⟩≡
```

MAT *points;
char *name;

{
  FILE *fp;
  if ( (fp=fopen("unbndd-1d-ipl-plot-data.mat","r")) == (FILE *) NULL ) {
    printf("Cannot open input file \"%s\"\n", "unbndd-1d-ipl-plot-data.mat");
    exit(1);
  }

  points = m_load(fp, &name);

  if (points->n != 1) {
    cout << "Error: unbndd-1d-ipl-plot-data.mat does not contain "\
      "a column vector.  Aborting." << endl;
    exit(1);
  }

  fclose(fp);
}
```

Defines:
  name, used in chunk 15.
  points, used in chunks 13a and 16a.
This code is used in chunk 13b.

For running the main loop, we need the previously computed answer vector.

15 ⟨*c plot: main loop* 15⟩≡

```
//      define and read the answer vector:
MAT *u_vec;
{
  FILE *fp;
  if ( (fp=fopen("unbndd-1d-ipl-answer.mat","r")) == (FILE *) NULL ) {
    printf("Cannot open input file \"%s\"\n", "unbndd-1d-ipl-answer.mat");
    exit(1);
  }

  u_vec = m_load(fp, &name);

  if (u_vec->n != 1) {
    cout << "Error: unbndd-1d-ipl-answer.mat does not contain "\
      "a column vector.  Aborting." << endl;
    exit(1);
  }
  if (u_vec->m != m) {
    cout << "Error: size of answer vector in unbndd-1d-ipl-answer.mat" \
      " is not compatible with global variable m.  Aborting." << endl;
    exit(1);
  }

  fclose(fp);
}
```

Uses `name` 14c.
This definition is continued in chunk 16a.
This code is used in chunk 13b.

Next, create a matrix with 5 columns to hold, in order, x, f, skh, f', skh_pri.

16a ⟨*c plot: main loop* 15⟩+≡
```
//
//      do plotting, put results in matrix:
MAT *results;
results = m_get(points->m, 5);
{

int i,j, k;
for (i=0;  i<points->m;  i++) {         // cover plot points
  double df = 0;
  double dfp = 0;
  for (j=-M;  j<=N;  j++) {
    df  += skh(j, h, points->me[i][0] ) * u_vec->me[j+M][0];
    dfp += skh_pri(j, h, points->me[i][0] ) * u_vec->me[j+M][0] ;
  }     // for j
  results->me[i][0] = points->me[i][0];
  results->me[i][1] = f(points->me[i][0]);
  results->me[i][2] = df;
  results->me[i][3] = f_pri(points->me[i][0]);
  results->me[i][4] = dfp;
}       // for i
}
```
Uses `points` 14c.

16b ⟨*c plot: save results* 16b⟩≡
```
{
  FILE *fp;
  if ( (fp=fopen("unbndd-1d-ipl-plot-results.mat","w")) == (FILE *) NULL ) {
    printf("Cannot open save file \"%s\"\n", "unbndd-1d-ipl-plot-results.mat");
    exit(1);
  }

  m_save(fp, results, "results");

  fclose(fp);
}
```
This code is used in chunk 13b.

2 Some test runs and results

For $f(x) = \sqrt{x}(1-x), \alpha = 1.0$, we have the following results.

- First derivative matricies of odd order are singular, and the solution obtained via direct elimination is useless – this would reflect in the 2D problem as well. Of course, with $N = M$, the matrix is always of odd order, so using $M = N + 1$ (or a similar adjustment) may help the 2D case also.
- The least squares solution obtained via the SVD for odd order matrices has very oscillatory results near the endpoints for the derivative, and a slight oscillation and an offset for f itself.
- Using even-order matricies, here with $M = N + 1 = 11$, there is no visual difference between the interpolants and the functions for $x = 0.01 : 0.01 : 0.99$. Numerically, the relative error for f's interpolant is less than 1%, except for $x = 0.99$, where it is 12%. For f', the error is less than 5%, uniformly.

For $f(x) = \sqrt{x}(1-x)^4, \alpha = 1.0, M = N + 1 = 11$, we get very good visual results, although the relative error deteriorates severely near the fourth order zero. The derivative approximation has a small jump near $x = 1$ as well.
With $\alpha = 0.5$, the endpoint results improve somewhat, but the interior is not nearly as good – as expected.

For $f(x) = x^{1/8}(1-x)^2, \alpha = 0.5, M = N + 1 = 11$, there is an offset in the approximation of f, and oscillations at the endpoints for f'.
Using $\alpha = 1.0$, the results are much worse; worse yet for $\alpha = 2.0$.

3 Indicies

3.1 Code Chunks

3.2 Identifiers