Abstract

This document is intended to serve as a user guide for the Numerical Steepest Descent (NSD) MATLAB package created by Daan Huybrechs and Andrew Gibbs at KU Leuven.

Contents

1 Setup

Our NSD code is available to download from https://github.com/AndrewGibbs/NSDpackage. Once downloaded, open MATLAB, navigate to `NSDpackage-master` and run `StandardPaths.m` to add all of the necessary subfolders to the MATLAB path.

2 Overview

This package is designed to solve integrals of the form

$$I_\omega[f,g] = \int_a^b f(x)\mathrm{e}^{\mathrm{i}\omega g(x)}\,\mathrm{d}x, \tag{2.1}$$

where

- a and b can be real numbers. Under certain restrictions, a and b may denote infinite points in the complex plane, see §3.2. For a (or b) finite, we represent this by the variable `a` (or `b`). For a (or b) infinite, we represent this by the angle of the infinite point by the variable `a` (or `b`).
- Frequency parameter $\omega > 0$, represented by the variable `freq`.
- g complex analytic in some neighbourhood of $[a,b]$.
- f is non-oscillatory, has a finite number of weak singularities inside of $[a,b]$, otherwise analytic inside a neighbourhood of $[a,b]$

Requiring only the derivatives of g, our package will return a quadrature rule

$$I_\omega[f,g] \approx Q_\omega[f,g] = \sum_i^N w_i f(z_i) \tag{2.2}$$

where the number of weights and nodes N does not depend on ω, however N will typically increase with the number of stationary points close to the region of integration. The only

requirement for our quadrature rule to work is a cell array containing vectorised anonymous functions corresponding to increasing derivatives of g. Hence,

```
G{j+1}= g_j ;,
```
where `g_j` represents $g^{(j)}(z)$.

For example, if $g(x) = x^2$, then we would have

```
G = { @(x) x.^2, @(x) 2*x, @(x) 2, @(x) 0} ;
```

Each derivative of g should return an upright vector of values, if the input is also an upright vector of values. Without this, the code will most likely error. The minimum number of derivatives required depends on the order of the stationary points, which in general is not known. Typically four or five derivatives should be more than enough, the user is notified if more are required. To obtain the weights and nodes of the quadrature rule (2.2), one can type into MATLAB:

```
[z,w] = PathFinder( a, b, freq, N, G);
```

This can then be evaluated by typing:

```
Q=w.'*f(z);
```

3 Additional options and features

A range of optional arguments can be input to `PathFinder`. Some of these can improve performance, whilst others broaden the type of problems solvable by our routine.

3.1 Polynomial phase

If g is an order m polynomial,

$$g(x) = c_1 x^m + c_2 x^{m-1} + \ldots + c_m x + c_{m+1}, \tag{3.1}$$

then the derivatives in `G`, the stationary points and their orders can all be determined from the coefficients of g. We can define a vector `C` in $\mathbb{C}^{m+1}$ such that the ith entry is equal to c_i of (3.1), and the routine will compute derivates as required.

```
[z,w] = PathFinder( a, b, freq, N, [],'gpolycoeffs',C,...);
```

3.2 Infinite and semi-infinite contour integrals

If a and/or b are/is an infinite point in the complex plane, then the optional argument of `'ainf'`, and/or `'binf'` flags this. The inputs `a` and `b` should be the angle of the infinite point at which the contour ends.

In the case of infinite endpoint(s), it is also necessary to provide a radius `R`> 0, such that outside of this region the steepest descent paths can be assumed to be straight lines. With a polynomial, this is simply where the leading order term will become dominant. This argument should be preceeded by `'settledRad'`.

In summary, for a fully infinite contour we would type:

```
[z,w] = PathFinder( a, b, freq, N, G,...,'ainf','binf','settledRad',R,...);
```

where the dots `...` denote any other optional arguments discussed in this section.

3.3 Weakly singular f

If f has a weak singularity inside of $[a, b]$, this is handled via *Generalised Gaussian Quadrature* [2]. The user must create an instance of the singularity class, containing all of the necessary information about the singularity, which has the inputs `singInfo=Singularity1D(position, blowUpType)`. Here `position` denotes the point in $\mathbb{R}$ where the singularity occurs, and `blowUpType` is a string describing the type of singularity, e.g. `blowUpType='log'`. To use this function, the extra argument should be added like so:

```
[z,w] = PathFinder( a, b, freq, N, G,...,'fSingularities',S,...);
```

where the dots `...` denote any other optional arguments discussed in this section.

3.4 Improving performance by providing stationary points and/or a search radius

By default, the routine will search for them, inside of a rectangle $\mathcal{R}$, for some search radius R such that

$$\mathcal{R} := \{z \in \mathbb{C} | \operatorname{dist}(z, [a, b]) < R\}.$$

The default option is $R \sim 1/\omega$, justified as stationary points which are far from $[a, b]$ will have little contribution to the integral. Alternatively the user can specify a search radius via the option `'rectRad'`,

```
[z,w] = PathFinder( a, b, freq, N, G,...,'rectRad',R,...);
```

where the dots `...` denote any other optional arguments discussed in this section.

The user *can* provide stationary points if they are known. This will significantly decrease the time taken to construct the quadrature rule, as searching for stationary points is often the most computationally expensive part of the method. This can be achieved simply by

```
[z,w] = PathFinder( a, b, freq, N, G,...,'stationary points', C, 'order', D,,...);
```

where `C` is a vector of stationary points, and `D` is a vector of their orders (see e.g. [1] for definitions of these terms).

3.5 Visualisation

Add the optional argument `'visuals on'` to produce a plot describing the process of the NSD package. The concentric rectangles used to find the roots are visible, alongside the nodes in the complex plane. Chosen paths are plotted in pink, and discarded paths are plotted in blue. For example, the following code

```
G =  @(x) x.^3, @(x) 3*x.^2, @(x) 6*x, @(x) 6, @(x) 0;
PathFinder( -1, 1, 10, 15, G, 'visuals on');
```

produced Figure 1.

4 Examples

Here we provide a series of example MATLAB code for some typical oscillatory integrals.

4.1 Simple phase

The integral

$$\int_{-1}^{1} \sin(x) \mathrm{e}^{100\mathrm{i}x^2} \, \mathrm{d}x$$

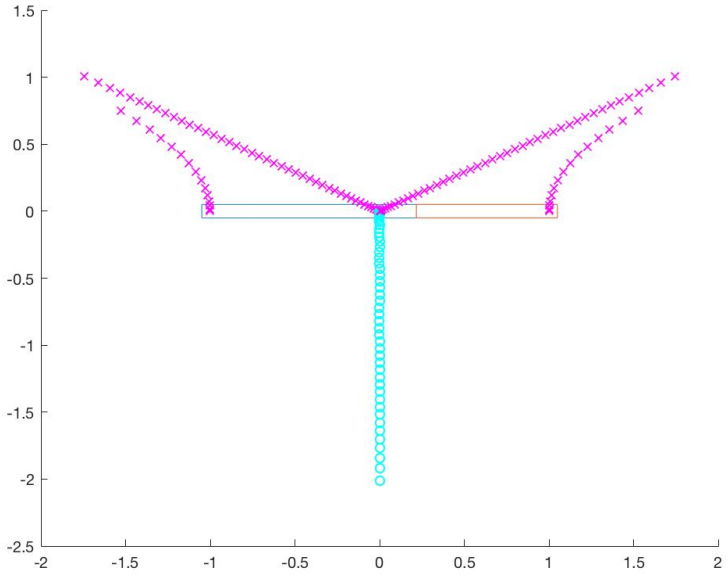

Figure 1

can be approximated using only a few lines with:

```
G={@(x) x.^2, @(x) 2*x, @(x) 2, @(x) 0};
[z,w] = PathFinder( -1,1,100,10,G);
Q = w.'*cos(z)
```

Alternatively, as the phase is a polynomial and we know the only coefficient, we may replace the first two lines with `[z,w] = PathFinder( -1,1,100,10,[],'gpolycoeffs',[1 0 0]);`. As explained in §3.1, this bypasses the need to search for stationary points. The fourth input to `PathFinder` ensures that there will be (roughly speaking) 10 quadrature points at each endpoint, and 2×10 on each path from the stationary point at zero.

4.2 Singular amplitude function

The integral

$$\int_0^1 \log(x) \mathrm{e}^{12345\mathrm{i}x} \, \mathrm{d}x$$

can be approximated in a few lines:

```
S=Singularity1D(0, 'log');
G = {@(x) x, @(x) 1, @(x) 0 };
[ z, w ] = PathFinder( 0,1,12345,10,G, 'fSingularities',S);
Q = w.'*log(z)
```

4.3 Contour integral

The following non-oscillatory integral over an infinite contour

$$\int_{\mathrm{e}^{\mathrm{i}9\pi/10}}^{\mathrm{e}^{\mathrm{i}\pi/10}} \mathrm{e}^{\mathrm{i}(2x^2/5 - x^4/2 - x^2)} \, \mathrm{d}x$$

can be approximated using five lines:

```
polyCoeffs=[2/5 -1/2 0 -1 0 0];
```

```
R = monomialSettleRadius(polyCoeffs);
a=exp( 1i*pi/10);
b=exp( 1i*pi*9/10);
[z,w] = PathFinder( a,b,1,10,[], 'ainf', 'binf', 'settleRad', R, 'gpolycoeffs', polyCoeffs);
Q = sum(w)
```

References

[1] Alfredo Deaño Gamallo, Daan Huybrechs, and Arieh Iserles. *Computing Highly Oscillatory Integrals*, volume 155. SIAM, 2018.

[2] D. Huybrechs and R. Cools. On generalized Gaussian Quadrautre rules for singular and nearly singular integrals. *SIAM J. Numer. Anal.*, 47:719–739, 2009.