- Course: YSC4103 MCS Capstone
- Date created: 2019/02/25
- Name: Linfan XIAO
- Description: Implement the Fokas method as described in "Evolution PDEs and augmented eigenfunctions. Finite interval."

Let $C = C^\infty[0,1]$ be the set of functions with derivatives of all orders on $[0,1]$.

Define $n$ linearly independent boundary forms $\{B_j : C \to \mathbb{C} \mid j \in \{1,2,\dots,n\}\}$ as follows:

$$B_j\phi = \sum_{k=0}^{n-1} \left( b_{jk}\phi^{(k)}(0) + \beta_{jk}\phi^{(k)}(1) \right), \quad j \in \{1,2,\dots,n\},$$

where $b_{jk}, \beta_{jk} \in \mathbb{R}$ are boundary coefficients. Define

$$\Phi = \{\phi \in C : B_j\phi = 0 \, \forall j \in \{1,2,\dots,n\}\}$$

to be the set of $n$ homogeneous boundary conditions

$$\begin{aligned} \left( b_{11}\phi^{(0)}(0) + \beta_{11}\phi^{(0)}(1) \right) + \cdots + \left( b_{1n}\phi^{(n-1)}(0) + \beta_{1n}\phi^{(n-1)}(1) \right) &= 0 \\ &\vdots \\ \left( b_{n1}\phi^{(0)}(0) + \beta_{n1}\phi^{(0)}(1) \right) + \cdots + \left( b_{nn}\phi^{(n-1)}(0) + \beta_{nn}\phi^{(n-1)}(1) \right) &= 0. \end{aligned}$$

Let $S : \Phi \to C$ be the linear differential operator

$$S\phi(x) = (-i)^n \phi^{(n)}(x).$$

Let $a \in \mathbb{C}$ be a constant. Consider the homogeneous initial-boundary value problem (IBVP):

$$\begin{aligned} (\partial_t + aS)q(x,t) &= 0, & (x,t) &\in (0,1) \times (0,T) \\ q(x,0) &\in \Phi, & x &\in [0,1] \\ q(\cdot,t) &\in \Phi, & t &\in [0,T] \end{aligned}$$

Let $f(x) := q(x,0)$. We reqire that if $n$ is odd then $a = \pm i$ and if $n$ is even then $\Re(a) \geq 0$. We solve the IBVP for $q(x,t)$.

# 1 Importing packages





**Contents**
7.1 check_boundaryConditions(U, fSym)
7.2 get_MPlusMinus(adjointU; symbolic, generic)
7.3 get_M(adjointU; symbolic, generic)
7.4 get_delta(adjointU; symbolic, generic)
7.5 get_Xlj(adjointU, l, j; symbolic, generic)
7.6 get_ChebyshevIntegral(l, f; symbolic, lambda, alpha)
7.6.1 get_ChebyshevTermInte(symbolic, c)
7.6.1.1 get_alpha(m, n)
7.6.2 get_ChebyshevCoeffici
7.6.3 get_ChebyshevIntegral(f; symbolic, lambda, alpha)
7.7 get_FPlusMinus(adjointU; symbolic, generic)
7.8 get_gamma(a, n, zeroList; infty, nGon)
7.8.1 get_gammaAAngles(a, n; symbolic)
7.8.2 get_gammaAAnglesSplit(n; symbolic)
7.8.3 pointOnSector(z, sectorAngles)
7.8.4 pointInSector(z, sectorAngles)
7.8.5 pointExSector(z, sectorAngles)
7.8.6 get_epsilon(zeroList, a, n)
7.8.7 get_nGonAroundZero(ze(epsilon, n)
7.8.8 get_gamma(a, n, zeroList; infty, nGon)
7.9 plot_contour(gamma; infty)
7.10 solve_IBVP(L, U, a, zeroList, f; FPlusFunc, FMinusFunc, pDerivMatrix, infty)
8 Verifying the formulas of $F_\lam
8.1 Problem 1
8.2 Problem 2

In [212]:
```
using SymPy
using PyCall
sympy = pyimport("sympy")
# using Roots
using Distributions
# using IntervalArithmetic
# using IntervalRootFinding
using ApproxFun

using Plots

using NBInclude
using QuadGK
import QuadGK.quadgk
# using HCubature
using ApproxFun
using Roots
using Gadfly
using PyPlot
# pygui(true)

using StringLiterals
```

# 2 Global variables

## 2.1 TOL

Error tolerance level.

In [213]:
```
TOL = 1e-05
```

Out[213]: 1.0e-5

## 2.2 DIGITS

The number of digits to display in symbolic expressions.

In [214]:
```
DIGITS = 3
```

Out[214]: 3

## 2.3 INFTY

A number representing infinity in numerical approximations.

```
In [215]: INFTY = 10
```

```
Out[215]: 10
```

## 2.4 t

Free symbol in the unknown function $x(t)$ in the differential equation $Lx = 0$.

```
In [216]: # t = symbols("t")
```

## 2.5 sympy_Exprs

Sample expressions of type addition, multiplication, power, and exponential in SymPy.

```
In [217]: x = symbols("x")
          sympyAddExpr = 1 + x
```

Out[217]: $x + 1$

```
In [218]: sympyMultExpr = 2*x
```

Out[218]: $2x$

```
In [219]: sympyPowerExpr = x^2
```

Out[219]: $x^2$

```
In [220]: sympyExpExpr = e^x
```

Out[220]: $e^x$

# 3 Helper functions







## 3.1 check_all(array, condition)

Checks whether all elements in an array satisfy a given condition.

```
In [221]: function check_all(array, condition)
              for x in array
                  if !condition(x)
                      return false
                  end
              end
              return true
          end
```

```
Out[221]: check_all (generic function with 1 method)
```

**Parameters**

- `array`: `Array`
  - Input array to be checked. Generic, not necessarily homogeneous.
- `condition`: `Function: Bool`
  - Boolean function to be applied to each element in the array.

**Returns**

- `check_all`: `Bool`
  - Returns `true` if all elements in `array` satisfy `condition` and `false` if any element does not satisfy the condition.

**Example**

```
In [222]: array = [0,1,2,3]
          condition = x -> x>2
          check_all(array, condition)
```

```
Out[222]: false
```

## 3.2 check_any(array, condition)

Checks whether any element in an array satisfy a given condition.

In [223]:
```
function check_any(array, condition)
    for x in array
        if condition(x)
            return true
        end
    end
    return false
end
```

Out[223]: check_any (generic function with 1 method)

**Parameters**

- array: Array
  - Input array to be checked. Generic, not necessarily homogeneous.
- condition: Function: Bool
  - Boolean function to be applied to each element in the array.

**Returns**

- check_all: Bool
  - Returns true if there exists an element in the array that satisfies a given condition and false if no element satisfies the condition.

**Example**

In [224]:
```
array = [0,1,2,3]
condition = x -> x>2
check_any(array, condition)
```

Out[224]: true

## 3.3 set_tol(x, y)

Sets an appropriate tolerance for checking whether two numbers or two arrays of numbers are approximately equal.







In [225]:
```
function set_tol(x::Union{Number, Array}, y::Union{Number, Array}; atol = TOL)
    if isa(x, Number) && isa(y, Number)
       return atol * mean([x y])
    elseif isa(x, Array) && isa(y, Array)
        if size(x) != size(y)
            throw(error("Array dimensions do not match"))
        end
        # Avoid InexactError() when taking norm()
        x = convert(Array{Complex}, x)
        y = convert(Array{Complex}, y)
        return atol * (norm(x,2) + norm(y,2))
    else
        throw(error("Invalid input"))
    end
end
```

Out[225]: set_tol (generic function with 1 method)

**Parameters**

- x, y: Number or Array of Number
  - Objects to compare.

**Returns**

- set_tol: Number
  - Returns a tolerance within which x and y are considered approximately equal (entry-wise if x, y are arrays).

**Example**

In [226]:
```
x = 14
y = 21
println("set_tol($x, $y) = $(set_tol(x,y))")

A = [4 0.6; 3 2]
B = [5 1; 10 3]
set_tol(A, B)
println("set_tol($A, $B) = $(set_tol(A,B))")
```

```
set_tol(14, 21) = 0.00017500000000000003
set_tol([4.0 0.6; 3.0 2.0], [5 1; 10 3]) = 0.00016901192145623727
```

## 3.4 evaluate(func, a)

Evaluate a univariate function or an array of them at a given value.

In [227]:
```
function evaluate(func::Union{Function, SymPy.Sym, Number}, a::Number)
    if isa(func, Function)
        funcA = func(a)
    elseif isa(func, SymPy.Sym) # SymPy.Sym must come before Number because SymPy.Sym will be recognized as
 Number
        freeSymbols = free_symbols(func)
        if length(freeSymbols) > 1
            throw(error("func should be univariate"))
        elseif length(freeSymbols) == 1
            t = free_symbols(func)[1,1]
            if isa(a, SymPy.Sym) # if x is SymPy.Sym, do not convert result to Number to preserve pretty pri
nting
                funcA = subs(func, t, a)
            else
                funcA = SymPy.N(subs(func, t, a))
            end
        else
            funcA = func
        end
    else # func is Number
        funcA = func
    end
    return funcA
end
```

Out[227]: evaluate (generic function with 1 method)

**Parameters**

- `func`: `Function`, `SymPy.Sym`, `Number`, or `Array`
  - Object to be evaluated. Note that `SymPy.Sym` is absorbed into `Number`.
- `x`: `Number`
  - Value on which `func` is to be evaluated at.
- `t`: `SymPy.Sym`
  - Free symbol in `func` if `func` is a `SymPy.Sym` object or an array of them.

**Returns**

- `evaluate`: `Number`
  - Returns the value of `func` at `x`.

**Example**







In [228]:
```
func = x -> x+1
xVal = 2
println("func($xVal) = $(evaluate(func, xVal))")

x = symbols("x")
func = x+1
println("func($xVal) = $(evaluate(func, xVal))")

a = symbols("a")
println("func($a) = $(evaluate(func, a))")

x = symbols("x")
array = [2x 1; x^3 x]
a = symbols("a")
evaluate.(array, a)
```

```
func(2) = 3
func(2) = 3
func(a) = a + 1
```

Out[228]:
$$\begin{bmatrix} 2a & 1 \\ a^3 & a \end{bmatrix}$$

## 3.5 partition(n)

Generate ordered two-integer partitions ($0$ included) of a given non-negative integer.

In [229]:
```
function partition(n::Int)
    if n < 0
        throw(error("Non-negative n required"))
    end
    output = []
    for i = 0:n
        j = n - i
        push!(output, (i,j))
    end
    return output
end
```

Out[229]: partition (generic function with 1 method)

**Parameters**

- n: Int
  - Non-negative integer to be partitioned.

**Returns**

- partition: Array of Tuple{Int, Int}
  - Returns an array of tuples, where each tuple corresponds to a two-integer parition of n. Note that a tuple is ordered, and (i, j) and (j, i) are considered distinct partitions.

**Example**

In [230]:
```
n = 5
partition(5)
```

Out[230]:
```
6-element Array{Any,1}:
 (0, 5)
 (1, 4)
 (2, 3)
 (3, 2)
 (4, 1)
 (5, 0)
```

## 3.6 get_deriv(u, k)

Constructs the symbolic expression for the $k$th derivative of a univariate function u.







In [231]:
```
function get_deriv(u::Union{SymPy.Sym, Number}, k::Int)
    if k < 0
        throw(error("Non-negative k required"))
    end
    if isa(u, SymPy.Sym)
        freeSymbols = free_symbols(u)
        if length(freeSymbols) > 1
            throw(error("u should be univariate"))
        elseif length(freeSymbols) == 1
            t = freeSymbols[1]
            y = u
            for i = 1:k
                newY = diff(y, t)
                y = newY
            end
            return y
        else
            if k > 0
                return 0
            else
                return u
            end
        end
    else
        if k > 0
            return 0
        else
            return u
        end
    end
end
```

Out[231]:
```
get_deriv (generic function with 1 method)
```

**Parameters**

- u: SymPy.Sym or Number
  - Function to be differentiated.
- k: Int
  - Degree of the desired derivative.

**Returns**

- get_deriv: SymPy.Sym
  - Returns the $k$th derivative of u.

**Example**

In [232]:
```
u = 3
k = 1
println("get_deriv($u, $k) = $(get_deriv(u, k))")

t = symbols("t")
u = t^2+2t+3
get_deriv(u, k)
```

```
get_deriv(3, 1) = 0
```

Out[232]: $2t + 2$

## 3.7 add_func(f, g)

Computes the sum of two functions using the function addition given by $(f + g)(x) := f(x) + g(x)$.

In [233]:
```
function add_func(f::Union{Number, Function}, g::Union{Number, Function})
    function h(x)
        if isa(f, Number)
            if isa(g, Number)
                return f + g
            else
                return f + g(x)
            end
        elseif isa(f, Function)
            if isa(g, Number)
                return f(x) + g
            else
                return f(x) + g(x)
            end
        end
    end
    return h
end
```

Out[233]: add_func (generic function with 1 method)

**Parameters**

- f, g: Function or Number
  - Functions to be added.

**Returns**

- add_func: Function or Number
  - Returns the sum of f and g.

**Example**







In [234]:
```
f(x) = x^3+1
g(x) = 4x
x = 2
add_func(f, g)(x) == f(x) + g(x)
```

Out[234]: true

## 3.8 mult_func(f, g)

Computes the product of two functions using the function multiplication given by $(f \cdot g)(x) := f(x) \cdot g(x)$.

In [235]:
```
function mult_func(f::Union{Number, Function}, g::Union{Number, Function})
    function h(x)
        if isa(f, Number)
            if isa(g, Number)
                return f * g
            else
                return f * g(x)
            end
        elseif isa(f, Function)
            if isa(g, Number)
                return f(x) * g
            else
                return f(x) * g(x)
            end
        end
    end
    return h
end
```

Out[235]: mult_func (generic function with 1 method)

**Parameters**

- f, g: Function or Number
  - Functions to be multiplied.

**Returns**

- add_func: Function or Number
  - Returns the product of f and g.

**Example**

In [236]:
```
f(x) = x^3+1
g(x) = 4x
x = 2
mult_func(f, g)(x) == f(x) * g(x)
```

Out[236]: true

## 3.9 sym_to_func(sym)

Converts a univariate symbolic expression to a function.

In [237]:
```
function sym_to_func(sym::Union{SymPy.Sym, Number})
    try
        freeSymbols = free_symbols(sym)
        if length(freeSymbols) > 1
            throw(error("sym should be univariate"))
        else
            function func(x)
                if length(freeSymbols) == 0
                    result = SymPy.N(sym)
                else
                    result = SymPy.N(subs(sym, freeSymbols[1], x))
                end
                return result
            end
            return func
        end
    catch
        function func(x)
            return sym
        end
        return func
    end
end
```

Out[237]: sym_to_func (generic function with 1 method)

**Parameters**

- sym: SymPy.Sym or Number
  - Symbolic expression to be converted.

**Returns**

- sym_to_func: Function
  - Function converted from sym.

**Example**







In [238]:
```
t = symbols("t")
sym = t^2+t
tVal = 2
println("sym_to_func($sym)($tVal) = $(sym_to_func(sym)(tVal))")

sym = [1 t; t^2 t^3]
sym_to_func.(sym)[2,1](2)
println("sym_to_func($sym[2,1])($tVal) = $(sym_to_func.(sym[2,1])(tVal))")

sym = 2
println("sym_to_func($sym)($tVal) = $(sym_to_func(sym)(tVal))")
```

```
sym_to_func(t^2 + t)(2) = 6
sym_to_func(SymPy.Sym[1 t; t^2 t^3][2,1])(2) = 4
sym_to_func(2)(2) = 2
```

## 3.10 prettyRound(x, digits)

Round a number at a given number of digits.

In [239]:
```
function prettyRound(x::Number; digits::Int = DIGITS)
    if isa(x, Int)
        return x
    elseif isa(x, Real)
        if isa(x, Rational) || isa(x, Irrational) # If x is rational or irrational numbers like e, pi
            return x
        elseif round(abs(x), digits) == floor(abs(x))
            return Int(round(x))
        else
            return round(x, digits)
            # return rationalize(x)
        end
    elseif isa(x, Complex)
        roundedReal = prettyRound(real(x), digits = digits)
        roundedComplex = prettyRound(imag(x), digits = digits)
        return roundedReal + im*roundedComplex
    else
        return round(x, digits)
    end
end
```

Out[239]: prettyRound (generic function with 1 method)

**Parameters**

- x: Number
  - Number to be rounded.
- digits*: Int
  - Number of decimal places to keep. Default to the global variable DIGITS.

**Returns**

- prettyRound: Number
  - Returns x rounded to the digits decimal places.

**Example**

In [240]:
```
# x = 0.0000001*im
x = 4.854572304702339 - 49.023298458074294im
prettyRound(x)
println("prettyRound($x) = $(prettyRound(x))")

x = [1/3 1/4 0]
println("prettyRound($x) = $(prettyRound.(x))")

x = [1//3 1//4 0//1]
println("prettyRound($x) = $(prettyRound.(x))")
```

```
prettyRound(4.854572304702339 - 49.023298458074294im) = 4.855 - 49.023im
prettyRound([0.333333 0.25 0.0]) = Real[0.333 0.25 0]
prettyRound(Rational{Int64}[1//3 1//4 0//1]) = Rational{Int64}[1//3 1//4 0//1]
```

## 3.11 prettyPrint(x)

Prints a symbolic scalar pretty-rounded floating numbers.







In [241]:
```
function prettyPrint(x::Union{Number, SymPy.Sym})
    expr = x
    if isa(expr, SymPy.Sym)
        prettyExpr = expr
        for a in sympy[:preorder_traversal](expr)
            if length(free_symbols(a)) == 0 && length(args(a)) == 0
                if !(a in [e, PI]) && length(intersect(args(a), [e, PI])) == 0 # keep the transcendental num
bers as symbols
                    prettyA = prettyRound.(SymPy.N(a))
                    prettyExpr = subs(prettyExpr, (a, prettyA))
                end
            end
        end
    else
        prettyExpr = prettyRound.(expr)
        prettyExpr = convert(SymPy.Sym, prettyExpr)
    end
    return prettyExpr
end
```

Out[241]: prettyPrint (generic function with 1 method)

**Parameters**

- x: Number or SymPy.Sym
  - Object to be printed.

**Returns**

- prettyPrint: SymPy.Sym
  - Returns symbolic expression of x with pretty-rounded floating numbers.

**Example**

In [242]:
```
t = symbols("t")
x = 1//3*t + e^(2t) + PI/2 + 1.00001*im + sympy[:sqrt](2) + 1//6
prettyPrint(x)
```

Out[242]: $\frac{t}{3} + e^{2t} + \frac{1}{6} + \sqrt{2} + \frac{\pi}{2} + i$

In [243]:
```
t = symbols("t")
x = [1//3*t PI; 1//6*t^2+t 1]
prettyPrint.(x)
```

Out[243]: $\begin{bmatrix} \frac{t}{3} & \pi \\ \frac{t^2}{6} + t & 1 \end{bmatrix}$

In [244]:
```
x = [0.0+1.0*im 1.0+0.0*im -1.0+0.0*im 0.0-1.0*im]
prettyPrint.(x)
```

Out[244]: $\begin{bmatrix} i & 1 & -1 & -i \end{bmatrix}$

## 3.12 is_approxLess(x, y; atol)

Checks whether $x$ is approximately less than $y$ within a tolerance. That is, whether $x \not\approx y$ and $x < y$.

In [245]:
```
function is_approxLess(x::Number, y::Number; atol = TOL)
    return !isapprox(x,y; atol = atol) && x < y
end
```

Out[245]: is_approxLess (generic function with 1 method)

**Parameters**

- x, y: Number
  - Numbers to compare.
- atol*: Number
  - Tolerance within which $x$ would be considered approximately equal to $y$. Default to the global variable TOL.

**Returns**

- is_approxLess: Bool
  - Returns true if $x \not\approx y$ within atol and $x < y$, and false otherwise.

**Example**

In [246]:
```
x = 1
y = x + TOL/2
println("is_approxLess($x,$y) = $(is_approxLess(x,y))")

y = x + TOL*2
println("is_approxLess($x,$y) = $(is_approxLess(x,y))")
```

```
is_approxLess(1,1.000005) = false
is_approxLess(1,1.00002) = true
```

## 3.13 is_approx(x, y; atol)

Checks whether $x$ is approximately equal to $y$ within a tolerance.







In [247]:
```
function is_approx(x::Number, y::Number; atol = TOL)
    return isapprox(x, y; atol = atol)
end
```

Out[247]: is_approx (generic function with 1 method)

**Parameters**

- x, y: Number
  - Numbers to compare.
- atol*: Number
  - Tolerance within which $x$ would be considered approximately equal to $y$. Default to the global variable TOL.

**Returns**

- is_approx: Bool
  - Returns true if $x \approx y$ within atol and false otherwise.

**Example**

In [248]:
```
x = 1
y = x + TOL/2
println("is_approx($x,$y) = $(is_approx(x,y))")

y = x + TOL*2
println("is_approx($x,$y) = $(is_approx(x,y))")
```

```
is_approx(1,1.000005) = true
is_approx(1,1.00002) = false
```

## 3.14 argument(z)

Finds the argument of a complex number in $[0, 2\pi)$.

In [249]:
```
function argument(z::Number)
    if angle(z) >= 0 # in [0,pi]
        return angle(z)
    else
        # angle(z) is in (-pi, 0]
        # Shift it up to (pi,2pi]
        argument = 2pi + angle(z) # This is in (pi,2pi]
        if is_approx(argument, 2pi) # Map 2pi to 0
            return 0
        else
            return argument # This is now in [0,2pi)
        end
    end
end
```

Out[249]: argument (generic function with 1 method)

**Parameters**

- z: Number
  - Complex number whose argument is to be found.

**Returns**

- argument: Number
  - Returns $\arg(z)$ in $[0, 2\pi)$.

**Example**

In [250]:
```
z = 1
println("argument($z) = $(argument(z))")

z = -1-im
println("argument($z) = $(argument(z))")
```

```
argument(1) = 0.0
argument(-1 - 1im) = 3.9269908169872414
```







## 3.15 trace_contour(a, n, infty, sampleSize)

Plots the contour sectors encircled by $\Gamma_a^+$, $\Gamma_a^-$, $\Gamma_0^+$, and $\Gamma_0^-$ defined as

$$\Gamma_a^\pm := \partial(\{\lambda \in \mathbb{C}^\pm : \Re(a\lambda^n) > 0\} \setminus \bigcup_{\substack{\sigma\in\mathbb{C};\\ \Delta(\sigma)=0}} D(\sigma, 2\epsilon)) \quad (D \text{ for disk}),$$

$$\Gamma_0^+ := \bigcup_{\substack{\sigma\in\mathbb{C}^+;\\ \Delta(\sigma)=0}} C(\sigma, \epsilon) \quad (C \text{ for circle}),$$

$$\Gamma_0^- := \bigcup_{\substack{\sigma\in\mathbb{C}^-;\\ \Delta(\sigma)=0}} C(\sigma, \epsilon)$$

by sampling points.

In [251]:
```
function trace_contour(a::Number, n::Int, sampleSize::Int; infty = INFTY)
    lambdaVec = []
    for counter = 1:sampleSize
        x = rand(Uniform(-infty,infty), 1, 1)[1]
        y = rand(Uniform(-infty,infty), 1, 1)[1]
        lambda = x + y*im
        if real(a*lambda^n)>0
            append!(lambdaVec, lambda)
        end
    end
    Gadfly.plot(x=real(lambdaVec), y=imag(lambdaVec), Guide.xlabel("Re"), Guide.ylabel("Im"), Coord.Cartesian(ymin=-infty,ymax=infty, xmin=-infty, xmax=infty, fixed = true))
end
```

Out[251]: trace_contour (generic function with 1 method)

**Parameters**

- a: Number
  - Complex number.
- n: Int
  - Integer $n$ in $\Gamma_a^{\pm}$.
- infty: Number
  - Range of plot. Default to the global variable INFTY.
- sampleSize: Int
  - Number of points to sample.

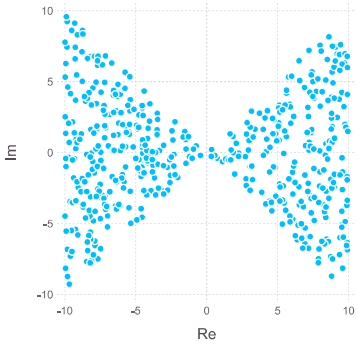
**Returns**

- trace_contour: None
  - Plots the sampled points.

**Example**

In [252]:
```
n = 2
a = 1
sampleSize = 1000
trace_contour(a, n, sampleSize)
```

Out[252]:



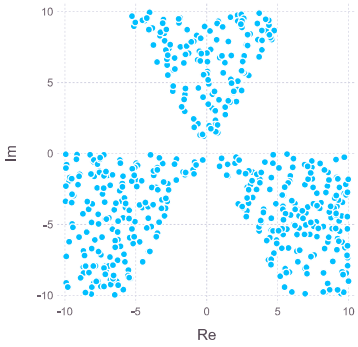






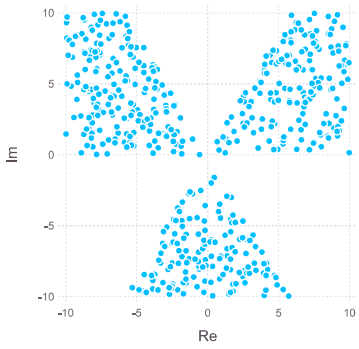

In [253]:
```
n = 3
a = im
sampleSize = 1000
trace_contour(a, n, sampleSize)
```

Out[253]:



In [254]:
```
n = 3
a = -im
sampleSize = 1000
trace_contour(a, n, sampleSize)
```

Out[254]:




## 3.16 get_distancePointLine(z, theta)

Finds the distance between a complex number and a line through the origin given by an angle in $[0, 2pi)$.







In [255]:
```
function get_distancePointLine(z::Number, theta::Number)
    if theta >= 2pi && theta < 0
        throw(error("Theta must be in [0,2pi)"))
    else
        if is_approx(argument(z), theta)
            return 0
        else
            x0, y0 = real(z), imag(z)
            if is_approx(theta, pi/2) || is_approx(theta, 3pi/2)
                return abs(x0)
            elseif is_approx(theta, 0) || is_approx(theta, 2pi)
                return abs(y0)
            else
                k = tan(theta)
                x = (y0+1/k*x0)/(k+1/k)
                y = k*x
                distance = norm(z-(x+im*y))
                return distance
            end
        end
    end
end
```

Out[255]: get_distancePointLine (generic function with 1 method)

**Parameters**

- z: Number
  - Point in the complex plane.
- theta: Number
  - Line passing the origin given by the argument of any point (besides the origin) on it.

**Returns**

- get_distancePointLine: Number
  - Returns the distance between z and the line characterized by theta.

**Example**

In [256]:
```
z = 1
theta = pi/3
println("get_distancePointLine($z, $theta) = $(get_distancePointLine(z, theta))") # sqrt(3)/2
```

```
get_distancePointLine(1, 1.0471975511965976) = 0.8660254037844386
```

# 4 Structs

## 4.1 StructDefinitionError

A struct definition error type is the class of all errors in struct definitions.

In [257]:
```
struct StructDefinitionError <: Exception
    msg::String
end
```

## 4.2 SymLinearDifferentialOperator(symPFunctions, interval, t)

A symbolic linear differential operator of order $n$ is encoded by an $1 \times (n+1)$ array of symbolic expressions with at most one free symbol, an interval $[a,b]$, and that free symbol.

In [258]:
```
struct SymLinearDifferentialOperator
    # Entries in the array should be SymPy.Sym or Number. SymPy.Sym seems to be a subtype of Number, i.e., Array{Union{Number,SymPy.Sym}} returns Array{Number}. But specifying symPFunctions as Array{Number,2} gives a MethodError when the entries are Sympy.Sym objects.
    symPFunctions::Array
    interval::Tuple{Number,Number}
    t::SymPy.Sym
    SymLinearDifferentialOperator(symPFunctions::Array, interval::Tuple{Number,Number}, t::SymPy.Sym) =
    try
        symL = new(symPFunctions, interval, t)
        check_symLinearDifferentialOperator_input(symL)
        return symL
    catch err
        throw(err)
    end
end

function check_symLinearDifferentialOperator_input(symL::SymLinearDifferentialOperator)
    symPFunctions, (a,b), t = symL.symPFunctions, symL.interval, symL.t
    for symPFunc in symPFunctions
        if isa(symPFunc, SymPy.Sym)
            if size(free_symbols(symPFunc)) != (1,) && size(free_symbols(symPFunc)) != (0,)
                throw(StructDefinitionError(:"Only one free symbol is allowed in symP_k"))
            end
        elseif !isa(symPFunc, Number)
            throw(StructDefinitionError(:"symP_k should be SymPy.Sym or Number"))
        end
    end
    return true
end
```

Out[258]: check_symLinearDifferentialOperator_input (generic function with 1 method)







**Parameters**

- `symPFunctions`: `Array` of `SymPy.Sym` or `Number`
  - Array $[symP_0, symP_1, \ldots, symP_n]$ of length $n+1$, corresponding to the symbolic linear differential operator $symL$ of order $n$ given by
$$symLx = symP_0 x^{(n)} + symP_1 x^{(n-1)} + \cdots + symP_{n-1} x^{(1)} + symP_n x.$$
- `interval`: `Tuple{Number, Number}`
  - Tuple of two numbers $(a,b)$ corresponding to the real interval $[a,b]$ on which the symbolic differential operator $symL$ is defined.
- `t`: `SymPy.Sym`
  - Free symbol in each entry of `symPFunctions`.

**Returns**

- `SymLinearDifferentialOperator`
  - Returns a `SymLinearDifferentialOperator` of order $n$ with attributes `symPFunctions`, `interval`, and `t`.

**Example**

In [259]:
```
t = symbols("t")
symPFunctions = [1 t+1 t^2+t+1]
interval = (0, 1)
symL = SymLinearDifferentialOperator(symPFunctions, interval, t)
```

Out[259]: SymLinearDifferentialOperator(SymPy.Sym[1 t + 1 t^2 + t + 1], (0, 1), t)

## 4.3 LinearDifferentialOperator(pFunctions, interval, symL)

A linear differential operator $L$ of order $n$ given by
$$Lx = p_0 x^{(n)} + p_1 x^{(n-1)} + \cdots + p_{n-1} x^{(1)} + p_n x$$
is encoded by an $1 \times (n+1)$ array of univariate functions, an interval $[a,b]$, and its symbolic expression.

In [260]:
```
# symL is an attribute of L that needs to be input by the user. There are checks to make sure symL is indeed the symbolic version of L.
# Principle: Functionalities of Julia Functions >= Functionalities of SymPy. If p_k has no SymPy representation, the only consequence should be that outputs by functions that take L as arugment has no symbolic expression. E.g., we allow L.pFunctions and L.symL.pFunctions to differ.
struct LinearDifferentialOperator
    pFunctions::Array # Array of julia functions or numbers representing constant functions
    interval::Tuple{Number,Number}
    symL::SymLinearDifferentialOperator
    LinearDifferentialOperator(pFunctions::Array, interval::Tuple{Number,Number}, symL::SymLinearDifferentialOperator) =
    try
        L = new(pFunctions, interval, symL)
        check_linearDifferentialOperator_input(L)
        return L
    catch err
```





**Contents**
7.1 check_boundaryConditions(U, fSym)
7.2 get_MPlusMinus(adjointU; symbolic, generic)
7.3 get_M(adjointU; symbolic, generic)
7.4 get_delta(adjointU; symbolic, generic)
7.5 get_Xlj(adjointU, l, j; symbolic, generic)
7.6 get_ChebyshevIntegral(l, f; symbolic, lambda, alpha)
7.6.1 get_ChebyshevTermInte(symbolic, c)
7.6.1.1 get_alpha(m, n)
7.6.2 get_ChebyshevCoeffici
7.6.3 get_ChebyshevIntegral(f; symbolic, lambda, alpha)
7.7 get_FPlusMinus(adjointU; symbolic, generic)
7.8 get_gamma(a, n, zeroList; infty, nGon)
7.8.1 get_gammaAAngles(a, n; symbolic)
7.8.2 get_gammaAAnglesSplit(n; symbolic)
7.8.3 pointOnSector(z, sectorAngles)
7.8.4 pointInSector(z, sectorAngles)
7.8.5 pointExSector(z, sectorAngles)
7.8.6 get_epsilon(zeroList, a, n)
7.8.7 get_nGonAroundZero(ze(epsilon, n)
7.8.8 get_gamma(a, n, zeroList; infty, nGon)
7.9 plot_contour(gamma; infty)
7.10 solve_IBVP(L, U, a, zeroList, f; FPlusFunc, FMinusFunc, pDerivMatrix, infty)
8 Verifying the formulas of $F_\lam
8.1 Problem 1
8.2 Problem 2

```
        throw(err)
    end
end

# Assume symFunc has only one free symbol, as required by the definition of SymLinearDifferentialOperator.
# That is, assume the input symFunc comes from SymLinearDifferentialOperator.
function check_func_sym_equal(func::Union{Function,Number}, symFunc, interval::Tuple{Number,Number}, t::SymPy.Sym) # symFunc should be Union{SymPy.Sym, Number}, but somehow SymPy.Sym gets ignored
    (a,b) = interval
    # Randomly sample 1000 points from (a,b) and check if func and symFunc agree on them
    for i = 1:1000
        # Check endpoints
        if i == 1
            x = a
        elseif i == 2
            x = b
        else
            x = rand(Uniform(a,b), 1)[1,1]
        end
        funcEvalX = evaluate.(func, x)
        if isa(symFunc, SymPy.Sym)
            symFuncEvalX = SymPy.N(subs(symFunc,t,x))
            # N() converts SymPy.Sym to Number
            # https://docs.sympy.org/latest/modules/evalf.html
            # subs() works no matter symFunc is Number or SymPy.Sym
        else
            symFuncEvalX = symFunc
        end
        tol = set_tol(funcEvalX, symFuncEvalX)
        if !isapprox(real(funcEvalX), real(symFuncEvalX); atol = real(tol)) ||
            !isapprox(imag(funcEvalX), imag(symFuncEvalX); atol = imag(tol))
            println("x = $x")
            println("symFunc = $symFunc")
            println("funcEvalX = $funcEvalX")
            println("symFuncEvalX = $symFuncEvalX")
            return false
        end
    end
    return true
end

# Check whether the inputs of L are valid.
function check_linearDifferentialOperator_input(L::LinearDifferentialOperator)
    pFunctions, (a,b), symL = L.pFunctions, L.interval, L.symL
    symPFunctions, t = symL.symPFunctions, symL.t
    # domainC = Complex(a..b, 0..0) # Domain [a,b] represented in the complex plane
    p0 = pFunctions[1]
    # p0Chebyshev = Fun(p0, a..b) # Chebysev polynomial approximation of p0 on [a,b]
    if !check_all(pFunctions, pFunc -> (isa(pFunc, Function) || isa(pFunc, Number)))
        throw(StructDefinitionError(:"p_k should be Function or Number"))
    elseif length(pFunctions) != length(symPFunctions)
        throw(StructDefinitionError(:"Number of p_k and symP_k do not match"))
    elseif (a,b) != symL.interval
        throw(StructDefinitionError(:"Intervals of L and symL do not match"))
```

```
    # # Assume p_k are in C^{n-k}. Check whether p0 vanishes on [a,b].
    # # roots() in IntervalRootFinding doesn't work if p0 is sth like t*im - 2*im. Neither does find_zero()
 in Roots.
    # # ApproxFun.roots()
    # elseif (isa(p0, Function) && (!isempty(roots(p0Chebyshev)) || all(x->x>b, roots(p0Chebyshev)) || all(x
->x<b, roots(p0Chebyshev)) || p0(a) == 0 || p0(b) == 0)) || p0 == 0
    #     throw(StructDefinitionError(:"p0 vanishes on [a,b]"))
    elseif !all(i -> check_func_sym_equal(pFunctions[i], symPFunctions[i], (a,b), t), 1:length(pFunctions))
        # throw(StructDefinitionError(:"symP_k does not agree with p_k on [a,b]"))
        warn("symP_k does not agree with p_k on [a,b]") # Make this a warning instead of an error because th
e functionalities of Julia Functions may be more than those of SymPy objects; we do not want to compromise t
he functionalities of LinearDifferentialOperator because of the restrictions on SymPy.
    else
        return true
    end
end
```

```
Out[260]: check_linearDifferentialOperator_input (generic function with 1 method)
```

**Parameters**

- pFunctions: Array of Function or Number
  - Array $[p_0, p_1, \ldots, p_n]$ of length $n+1$, corresponding to the linear differential operator $L$ of order $n$ given by
$$Lx = p_0 x^{(n)} + p_1 x^{(n-1)} + \cdots + p_{n-1} x^{(1)} + p_n x.$$
- interval: Tuple{Number, Number}
  - Tuple of two numbers (a, b) corresponding to the real interval $[a, b]$ on which the differential operator $L$ is defined.
- symL: SymLinearDifferentialOperator
  - Symbolic linear differential operator corresponding to $L$.

**Returns**

- LinearDifferentialOperator
  - Returns a LinearDifferentialOperator of order $n$ with attributes pFunctions, interval, and symL.

**Example**

```
In [261]: t = symbols("t")
          symPFunctions = [1 t+1 t^2+t+1]
          interval = (0, 1)
          symL = SymLinearDifferentialOperator(symPFunctions, interval, t)
          # Direct construction
          pFunctions = [t->1 t->t+1 t->t^2+t+1]
          L = LinearDifferentialOperator(pFunctions, interval, symL)
```

```
Out[261]: LinearDifferentialOperator(Function[#174 #175 #176], (0, 1), SymLinearDifferentialOperator(SymPy.Sym[1 t +
          1 t^2 + t + 1], (0, 1), t))
```







## 4.4 VectorBoundaryForm(M, N)

A set of homogeneous boundary conditions in vector form

$$Ux = \begin{bmatrix} U_1 \\ \vdots \\ U_m \end{bmatrix} x = \begin{bmatrix} \sum_{j=1}^{n} M_{1j} x^{(j-1)}(a) + N_{1j} x^{(j-1)}(b) \\ \vdots \\ \sum_{j=1}^{n} M_{mj} x^{(j-1)}(a) + N_{mj} x^{(j-1)}(b) \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}$$

is encoded by an ordered pair of two linearly independent $m \times n$ matrices $(M, N)$ where

$$M = \begin{bmatrix} M_{11} & \cdots & M_{1n} \\ \vdots & \ddots & \vdots \\ M_{m1} & \cdots & M_{mn} \end{bmatrix}, \quad N = \begin{bmatrix} N_{11} & \cdots & N_{1n} \\ \vdots & \ddots & \vdots \\ N_{m1} & \cdots & N_{mn} \end{bmatrix}.$$

In [262]:
```
struct VectorBoundaryForm
    M::Array # Why can't I specify Array{Number,2} without having a MethodError?
    N::Array
    VectorBoundaryForm(M::Array, N::Array) =
    try
        U = new(M, N)
        check_vectorBoundaryForm_input(U)
        return U
    catch err
        throw(err)
    end
end

# Check whether the input matrices that characterize U are valid
function check_vectorBoundaryForm_input(U::VectorBoundaryForm)
    # M, N = U.M, U.N
    # Avoid Inexact() error when taking rank()
    M = convert(Array{Complex}, U.M)
    N = convert(Array{Complex}, U.N)
    if !(check_all(U.M, x -> isa(x, Number)) && check_all(U.N, x -> isa(x, Number)))
        throw(StructDefinitionError(:"Entries of M, N should be Number"))
    elseif size(U.M) != size(U.N)
        throw(StructDefinitionError(:"M, N dimensions do not match"))
    elseif size(U.M)[1] != size(U.M)[2]
        throw(StructDefinitionError(:"M, N should be square matrices"))
    elseif rank(hcat(M, N)) != size(M)[1] # rank() throws weird "InexactError()" when taking some complex ma
trices
        throw(StructDefinitionError(:"Boundary operators not linearly independent"))
    else
        return true
    end
end
```

Out[262]: check_vectorBoundaryForm_input (generic function with 1 method)

**Parameters**

- `M, N`: `Array` of `Number`
  - Two linearly independent numeric matrices of the same dimension.

**Returns**

- `VectorBoundaryForm`
  - Returns a `VectorBoundaryForm` with attributes `M` and `N`.

**Example**







In [263]:
```
M = [1 0; 2 0]
N = [0 2; 0 1]
U = VectorBoundaryForm(M, N)
```

Out[263]: VectorBoundaryForm([1 0; 2 0], [0 2; 0 1])

# 5 Construct adjoint boundary conditions

Constructs a valid adjoint boundary condition from a given (homogeneous) boundary condition based on Chapter 11 in Theory of Ordinary Differential Equations (Coddington & Levinson).

## 5.1 `get_L(symL)`

Constructs a `LinearDifferentialOperator` from a given `SymLinearDifferentialOperator`.

In [264]:
```
function get_L(symL::SymLinearDifferentialOperator)
    symPFunctions, (a,b), t = symL.symPFunctions, symL.interval, symL.t
    if check_all(symPFunctions, x->!isa(x, SymPy.Sym))
        pFunctions = symPFunctions
    else
        pFunctions = sym_to_func.(symPFunctions)
    end
    L = LinearDifferentialOperator(pFunctions, (a,b), symL)
    return L
end
```

Out[264]: get_L (generic function with 1 method)

**Parameters**

- `symL`: `SymLinearDifferentialOperator`
  - Symbolic linear differential operator to be converted.

**Returns**

- `get_L`: `LinearDifferentialOperator`
  - Returns the linear differential operator converted from `symL`.

**Example**

```
In [265]: t = symbols("t")
          symPFunctions = [1 t+1 t^2+t+1]
          interval = (0, 1)
          symL = SymLinearDifferentialOperator(symPFunctions, interval, t)
          L = get_L(symL)
```

```
Out[265]: LinearDifferentialOperator(#func#164{SymPy.Sym}[func func func], (0, 1), SymLinearDifferentialOperator(SymP
          y.Sym[1 t + 1 t^2 + t + 1], (0, 1), t))
```

## 5.2 get_URank(U)

Computes the rank of a vector boundary form $U$ by computing the equivalent $\text{rank}(M:N)$, where $M, N$ are the matrices associated with $U$ and

$$(M:N) = \begin{bmatrix} M_{11} & \cdots & M_{1n} & N_{11} & \cdots & N_{1n} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ M_{m1} & \cdots & M_{mn} & N_{m1} & \cdots & N_{mn} \end{bmatrix}.$$

```
In [266]: function get_URank(U::VectorBoundaryForm)
              # Avoid InexactError() when taking hcat() and rank()
              M = convert(Array{Complex}, U.M)
              N = convert(Array{Complex}, U.N)
              MHcatN = hcat(M, N)
              return rank(MHcatN)
          end
```

```
Out[266]: get_URank (generic function with 1 method)
```

**Parameters**

- U: VectoBoundaryForm
  - Vector boundary form whose rank is to be computed.

**Returns**

- get_URank: Number
  - Returns the rank of U.

**Example**

```
In [267]: M = [1 0; 2 0]
          N = [0 2; 0 1]
          U = VectorBoundaryForm(M, N)
          get_URank(U)
```

```
Out[267]: 2
```







## 5.3 get_Uc(U)

Given vector boundary form $U = \begin{bmatrix} U_1 \\ \vdots \\ U_m \end{bmatrix}$ of rank $m$, finds a complementary form $U_c = \begin{bmatrix} U_{m+1} \\ \vdots \\ U_{2n} \end{bmatrix}$ of rank $2n-m$ such that $\begin{bmatrix} U_1 \\ \vdots \\ U_{2n} \end{bmatrix}$ has rank $2n$.

```
In [268]: function get_Uc(U::VectorBoundaryForm)
              try
                  check_vectorBoundaryForm_input(U)
                  n = get_URank(U)
                  I = complex(eye(2*n))
                  M, N = U.M, U.N
                  MHcatN = hcat(M, N)
                  # Avoid InexactError() when taking rank()
                  mat = convert(Array{Complex}, MHcatN)
                  for i = 1:(2*n)
                      newMat = vcat(mat, I[i:i,:])
                      newMat = convert(Array{Complex}, newMat)
                      if rank(newMat) == rank(mat) + 1
                          mat = newMat
                      end
                  end
                  UcHcat = mat[(n+1):(2n),:]
                  Uc = VectorBoundaryForm(UcHcat[:,1:n], UcHcat[:,(n+1):(2n)])
                  return Uc
              catch err
                  return err
              end
          end
```

```
Out[268]: get_Uc (generic function with 1 method)
```

**Parameters**

- U: VectoBoundaryForm
  - Vector boundary form whose complementary boundary form is to be found.

**Returns**

- get_Uc: VectorBoundaryForm
  - Returns a vectory boundary form complementary to U.

**Example**

```
In [269]: M = [1 0; 2 0]
          N = [0 2; 0 1]
          U = VectorBoundaryForm(M, N)
          Uc = get_Uc(U)
```

```
Out[269]: VectorBoundaryForm(Complex[0.0+0.0im 1.0+0.0im; 0.0+0.0im 0.0+0.0im], Complex[0.0+0.0im 0.0+0.0im; 1.0+0.0im 0.0+0.0im])
```

## 5.4 get_H(U, Uc)

Given a vector boundary form $U$ and a complementary vector boundary form $U_c$, constructs

$$H = \begin{bmatrix} M & N \\ M_c & N_c \end{bmatrix},$$

where $M, N$ are the matrices associated with $U$ and $M_c, N_c$ are associated with $U_c$.

```
In [270]: function get_H(U::VectorBoundaryForm, Uc::VectorBoundaryForm)
              MHcatN = hcat(convert(Array{Complex}, U.M), convert(Array{Complex}, U.N))
              McHcatNc = hcat(convert(Array{Complex}, Uc.M), convert(Array{Complex}, Uc.N))
              H = vcat(MHcatN, McHcatNc)
              return H
          end
```

```
Out[270]: get_H (generic function with 1 method)
```

**Parameters**

- U: VectorBoundaryForm
  - Vector boundary form.
- Uc: VectorBoundaryForm
  - Vector boundary form complementary to U.

**Returns**

- get_H: Array
  - Returns the matrix $H$ defined above.

**Example**







```
In [271]: M = [1 0; 2 0]
          N = [0 2; 0 1]
          U = VectorBoundaryForm(M, N)
          Uc = get_Uc(U)
          get_H(U, Uc)
```

```
Out[271]: 4×4 Array{Complex,2}:
             1+0im      0+0im      0+0im      2+0im
             2+0im      0+0im      0+0im      1+0im
           0.0+0.0im  1.0+0.0im  0.0+0.0im  0.0+0.0im
           0.0+0.0im  0.0+0.0im  1.0+0.0im  0.0+0.0im
```

## 5.5 get_pDerivMatrix(L; symbolic)

Given a LinearDifferentialOperator L where L.pFunctions is the array

$$[p_0, p_1, \ldots, p_n],$$

constructs an $n \times n$ matrix whose $(i+1)(j+1)$-entry is a function corresponding to the $j$th derivative of $p_i$:

$$\begin{bmatrix} p_0 & \cdots & p_0^{(n-1)} \\ \vdots & \ddots & \vdots \\ p_{n-1} & \cdots & p_{n-1}^{(n-1)} \end{bmatrix}.$$

In [272]:
```
function get_pDerivMatrix(L::LinearDifferentialOperator; symbolic = false)
    if symbolic
        symL = L.symL
        symPFunctions, t = symL.symPFunctions, symL.t
        n = length(symPFunctions)-1
        symPDerivMatrix = Array{SymPy.Sym}(n,n)
        pFunctionSymbols = symPFunctions
        for i in 0:(n-1)
            for j in 0:(n-1)
                index, degree = i, j
                symP = pFunctionSymbols[index+1]
                # If symP is not a Sympy.Sym object (e.g., is a Number instead), then cannot use get_deriv()
                if !isa(symP, SymPy.Sym)
                    if degree > 0
                        symPDeriv = 0
                    else
                        symPDeriv = symP
                    end
                else
                    symPDeriv = get_deriv(symP, degree)
                end
                symPDerivMatrix[i+1,j+1] = symPDeriv
            end
        end
        return symPDerivMatrix
    else
        symPDerivMatrix = get_pDerivMatrix(L; symbolic = true)
        n = length(L.pFunctions)-1
        pDerivMatrix = sym_to_func.(symPDerivMatrix)
    end
    return pDerivMatrix
end
```

Out[272]: get_pDerivMatrix (generic function with 1 method)







**Parameters**

- `L`: `LinearDifferentialOperator`
  - Linear differential operator whose `pDerivMatrix` is to be constructed.
- `symbolic*`: `Bool`
  - Boolean indicating whether the output is symbolic.

**Returns**

- `get_pDerivMatrix`: `Array` of `Function`, `Number` or `SymPy.Sym`
  - Returns an $n \times n$ matrix whose $(i+1)(j+1)$-entry is
    - the $j$th derivative of $p_i$ (`L.pFunctions[i]`) if `symbolic = false`, or
    - the symbolic expression of the $j$th derivative of $p_i$ (`L.symL.symPFunctions[i]`) if `symbolic = true`.

**Example**

In [273]:
```
t = symbols("t")
symPFunctions = [1 t+1 t^2+t+1]
interval = (0, 1)
symL = SymLinearDifferentialOperator(symPFunctions, interval, t)
# pFunctions = [t->1 t->t+1 t->t^2+t+1]
# L = LinearDifferentialOperator(pFunctions, interval, symL)
L = get_L(symL)

tVal = 2
pDerivMatrix = get_pDerivMatrix(L; symbolic = false)
println("pDerivMatrix($tVal) = $(evaluate.(pDerivMatrix,tVal))")

symPDerivMatrix = get_pDerivMatrix(L; symbolic = true)
```

```
pDerivMatrix(2) = [1 0; 3 1]
```

Out[273]:
$$\begin{bmatrix} 1 & 0 \\ t+1 & 1 \end{bmatrix}$$

## 5.6 get_Bjk(L, j, k; symbolic, pDerivMatrix)

Given a `LinearDifferentialOperator` `L` of order $n$, for $j, k \in \{1, \ldots, n\}$, computes $B_{jk}$ defined as

$$B_{jk}(t) := \sum_{\ell=j-1}^{n-k} \binom{\ell}{j-1} p_{n-k-\ell}^{(\ell-j+1)}(t)(-1)^{\ell}.$$

In [274]:
```
function get_Bjk(L::LinearDifferentialOperator, j::Int, k::Int; symbolic = false, pDerivMatrix = get_pDerivMatrix(L; symbolic = symbolic))
    n = length(L.pFunctions)-1
    if j <= 0 || j > n || k <= 0 || k > n
        throw("j, k should be in {1, ..., n}")
    end
    sum = 0
    if symbolic
        symPDerivMatrix = get_pDerivMatrix(L; symbolic = true)
        for l = (j-1):(n-k)
            summand = binomial(l, j-1) * symPDerivMatrix[n-k-l+1, l-j+1+1] * (-1)^l
            sum += summand
        end
    else
        for l = (j-1):(n-k)
            summand = mult_func(binomial(l, j-1) * (-1)^l, pDerivMatrix[n-k-l+1, l-j+1+1])
            sum = add_func(sum, summand)
        end
    end
    return sum
end
```

Out[274]: get_Bjk (generic function with 1 method)

**Parameters**

- `L`: `LinearDifferentialOperator`
  - Linear differential operator whose `L.pFunctions` are to become the $p_{n-k-l}^{l-j+1}$ in $B_{jk}(t)$.
- `j`, `k`: `Int`
  - Integers corresponding to the $j$ and $k$ in $B_{jk}$.
- `symbolic`*: `Bool`
  - Boolean indicating whether the output is symbolic.
- `pDerivMatrix`*: `Array`
  - If `symbolic = false`, an $n \times n$ matrix whose $(i+1)(j+1)$-entry is the $j$th derivative of $p_i$ (`L.pFunctions[i]`) implemented as a `Function`, `Number`, or `SymPy.Sym`. Default to the output of `get_pDerivMatrix(L)`.

**Returns**

- `get_Bjk`: `SymPy.Sym`, `Function`, or `Number`
  - Returns $B_{jk}(t)$ defined above,
    - as `Function` if `symbolic = false`, or
    - as `SymPy.Sym` object if `symbolic = true`, where each $p_i$ is the generic expression $p_i(t)$.

**Example**







In [275]:
```
t = symbols("t")
symPFunctions = [1 t+1 t^2+t+1]
interval = (0, 1)
symL = SymLinearDifferentialOperator(symPFunctions, interval, t)

# pFunctions = [t->1 t->t+1 t->t^2+t+1]
# L = LinearDifferentialOperator(pFunctions, interval, symL)
L = get_L(symL)

# pDerivMatrix = [1 0; t->1+t t->1]
j, k = 1, 1
BjkSym = get_Bjk(L, j, k; symbolic = true)
```

Out[275]: $t+1$

In [276]:
```
tVal = 1
Bjk = get_Bjk(L, j, k; symbolic = false)
println("Bjk($tVal) = $(Bjk(tVal))")
println("BjkSym($tVal) = $(evaluate.(BjkSym, tVal))")
```

```
Bjk(1) = 2
BjkSym(1) = 2
```

## 5.7 get_B(L; symbolic, pDerivMatrix)

Given a `LinearDifferentialOperator` `L` where `L.pFunctions` is the array

$$[p_0, p_1, \ldots, p_n],$$

constructs the matrix $B(t)$ whose $ij$-entry is given by

$$B_{jk}(t) := \sum_{\ell=j-1}^{n-k} \binom{\ell}{j-1} p_{n-k-\ell}^{(\ell-j+1)}(t)(-1)^\ell.$$

In [277]:
```
function get_B(L::LinearDifferentialOperator; symbolic = false, pDerivMatrix = get_pDerivMatrix(L; symbolic = symbolic))
    n = length(L.pFunctions)-1
    B = Array{Union{Function, Number, SymPy.Sym}}(n,n)
    for j = 1:n
        for k = 1:n
            B[j,k] = get_Bjk(L, j, k; symbolic = symbolic, pDerivMatrix = pDerivMatrix)
        end
    end
    return B
end
```

Out[277]: get_B (generic function with 1 method)

**Parameters**

- `L`: `LinearDifferentialOperator`
  - Linear differential operator whose `L.pFunctions` are to become the $p_{n-k-l}^{l-j+1}$ in $B_{jk}(t)$.
- `symbolic*`: `Bool`
  - Boolean indicating whether the output is symbolic.
- `substitute*`: `Bool`
  - If `symbolic = true`, boolean indicating whether to substitute the symbolic expression of $p_i$ in `L.pFunctions` for the generic expression $p_i(t)$ created using `SymFunction("pi")(t)`. If `symbolic = false`, the value of `substitute` does not matter.
- `pDerivMatrix*`: `Array`
  - If `symbolic = false`, the non-symbolic version of `symPDerivMatrix`, i.e., an $n \times n$ matrix whose $(i+1)(j+1)$-entry is the $j$th derivative of $p_i$ (`L.pFunctions[i]`) implemented as a `Function` or `Number`.

**Returns**

- `get_B`: `Array` of `Function`, `SymPy.Sym`, or `Number`
  - Returns $B(t)$ defined above, where $B_{jk}(t)$ is
    - `Function` if `symbolic = false`, or
    - `SymPy.Sym` object if `symbolic = true`, where each $p_i$ is
      - the generic expression $p_i(t)$ if `substitute = false`, or
      - the symbolic expression of $p_i(t)$ (`L.symL.symPFunctions[i]`) if `substitute = true`.

**Example**

In [278]:
```
t = symbols("t")
symPFunctions = [1 t+1 t^2+t+1]
interval = (0, 1)
symL = SymLinearDifferentialOperator(symPFunctions, interval, t)
L = get_L(symL)

BSym = get_B(L; symbolic = true)
# Only Array{SymPy.Sym} would be automatically pretty-printed
# Enfore pretty-printing
prettyPrint.(BSym)
```

Out[278]:
$$\begin{bmatrix} t+1 & 1 \\ -1 & 0 \end{bmatrix}$$

In [279]:
```
B = get_B(L; symbolic = false)
tVal = 1
println("B($tVal) = $(evaluate.(B, tVal))")
println("BSym($tVal) = $(evaluate.(BSym, tVal))")
```

```
B(1) = [2 1; -1 0]
BSym(1) = Number[2 1; -1 0]
```







## 5.8 get_BHat(L, B)

Given a `LinearDifferentialOperator` `L` where `L.pFunctions` is the array

$$[p_0, p_1, \ldots, p_n]$$

and `L.interval` is $[a, b]$, constructs $\hat{B}$ defined as the block matrix

$$\hat{B} := \begin{bmatrix} -B(a) & 0_n \\ 0_n & B(b) \end{bmatrix}.$$

In [280]:
```
function get_BHat(L::LinearDifferentialOperator, B::Array)
#     if check_any(B, x->isa(x, SymPy.Sym))
#         throw("Entries of B should be Function or Number")
#     end
    pFunctions, (a,b) = L.pFunctions, L.interval
    n = length(pFunctions)-1
    BHat = Array{Complex}(2n,2n)
    BEvalA = evaluate.(B, a)
    BEvalB = evaluate.(B, b)
    BHat[1:n,1:n] = -BEvalA
    BHat[(n+1):(2n),(n+1):(2n)] = BEvalB
    BHat[1:n, (n+1):(2n)] = 0
    BHat[(n+1):(2n), 1:n] = 0
    return BHat
end
```

Out[280]: `get_BHat (generic function with 1 method)`

**Parameters**

- `L`: `LinearDifferentialOperator`
  - Linear differential operator whose `L.pFunctions` are to become the $p_{n-k-l}^{l-j+1}$ in $B_{jk}(t)$.
- `B`: `Array` of `Number`
  - Output of `get_B(L; symbolic = false)`.

**Returns**

- `get_BHat`: `Array` of `Number`
  - Returns $\hat{B}$ defined above.

**Example**

In [281]:
```
t = symbols("t")
symPFunctions = [1 t+1 t^2+t+1]
interval = (0, 1)
symL = SymLinearDifferentialOperator(symPFunctions, interval, t)
L = get_L(symL)

B = get_B(L; symbolic = false)
get_BHat(L, B)
```

Out[281]:
```
4×4 Array{Complex,2}:
 -1+0im  -1+0im   0+0im  0+0im
  1+0im   0+0im   0+0im  0+0im
  0+0im   0+0im   2+0im  1+0im
  0+0im   0+0im  -1+0im  0+0im
```

## 5.9 get_J(BHat, H)

Given $\hat{B}$ and $H$, constructs $J$ defined as

$$J := (\hat{B}H^{-1})^{\star}$$

where $^{\star}$ denotes conjugate transpose.

In [282]:
```
function get_J(BHat, H)
    n = size(H)[1]
    H = convert(Array{Complex}, H)
    J = (BHat * inv(H))'
    # J = convert(Array{Complex}, J)
    return J
end
```

Out[282]:
```
get_J (generic function with 1 method)
```

**Parameters**

- BHat: Array
  - Output of get_BHat().
- H: Array
  - Output of get_H().

**Returns**

- get_J: Array
  - Returns $J$ defined above.

**Example**







In [283]:
```
t = symbols("t")
symPFunctions = [1 t+1 t^2+t+1]
interval = (0, 1)
symL = SymLinearDifferentialOperator(symPFunctions, interval, t)
L = get_L(symL)

B = get_B(L; symbolic = false)
BHat = get_BHat(L, B)

M = [1 0; 2 0]
N = [0 2; 0 1]
U = VectorBoundaryForm(M, N)
Uc = get_Uc(U)
H = get_H(U, Uc)

get_J(BHat, H)
```

Out[283]:
```
4×4 Array{Complex,2}:
  0.333333-0.0im  -0.333333-0.0im   0.666667-0.0im   0.0-0.0im
 -0.666667-0.0im   0.666667-0.0im  -0.333333-0.0im   0.0-0.0im
      -1.0-0.0im        0.0-0.0im        0.0-0.0im   0.0-0.0im
       0.0-0.0im        0.0-0.0im        2.0-0.0im  -1.0-0.0im
```

## 5.10 get_adjointCand(J)

Given $J$, constructs a candidate adjoint vector boundary form $U^{+}$ from two matrices $P^{*}$, $Q^{*}$, which are the lower-left $n \times n$ submatrix of $J$, and the lower-right $n \times n$ submatrix of $J$, respectively.

In [284]:
```
function get_adjointCand(J)
    n = convert(Int, size(J)[1]/2)
    J = convert(Array{Complex}, J)
    PStar = J[(n+1):2n,1:n]
    QStar = J[(n+1):2n, (n+1):2n]
    adjointU = VectorBoundaryForm(PStar, QStar)
    return adjointU
end
```

Out[284]:
```
get_adjointCand (generic function with 1 method)
```

**Parameters**

- `J`: `Array`
  - Output of `get_J`.

**Returns**

- `get_adjoint`: `VectorBoundaryForm`
  - Returns $U^+$ defined above.

**Example**

In [285]:
```
t = symbols("t")
symPFunctions = [1 t+1 t^2+t+1]
interval = (0, 1)
symL = SymLinearDifferentialOperator(symPFunctions, interval, t)
L = get_L(symL)

B = get_B(L; symbolic = false)
BHat = get_BHat(L, B)

M = [1 0; 2 0]
N = [0 2; 0 1]
U = VectorBoundaryForm(M, N)
Uc = get_Uc(U)
H = get_H(U, Uc)

J = get_J(BHat, H)
adjoint = get_adjointCand(J)
```

Out[285]:
```
VectorBoundaryForm(Complex[-1.0-0.0im 0.0-0.0im; 0.0-0.0im 0.0-0.0im], Complex[0.0-0.0im 0.0-0.0im; 2.0-0.0im -1.0-0.0im])
```

## 5.11 get_xi(L; symbolic, xSym)

Given a `LinearDifferentialOperator` `L` of order $n$ in the differential equation $Lx = 0$, constructs $\xi(t)$, which is defined as the vector of derivatives of $x(t)$

$$\xi(t) := \begin{bmatrix} x(t) \\ x^{(1)}(t) \\ x^{(2)}(t) \\ \vdots \\ x^{(n-1)}(t) \end{bmatrix}.$$







In [286]:
```
function get_xi(L::LinearDifferentialOperator; symbolic = true, xSym= nothing)
    if symbolic
        t = L.symL.t
        n = length(L.pFunctions)-1
        symXi = Array{SymPy.Sym}(n,1)
        if isa(xSym, Void)
            throw(error("xSymrequired"))
        else
            for i = 1:n
                symXi[i] = get_deriv(xSym, i-1)
            end
            return symXi
        end
    else
        if isa(xSym, Void)
            throw(error("xSym required"))
        elseif !isa(xSym, SymPy.Sym) && !isa(xSym, Number)
            throw(error("xSym should be SymPy.Sym or Number"))
        else
            symXi = get_xi(L; symbolic = true, xSym = xSym)
            xi = sym_to_func.(symXi)
        end
    end
end
```

Out[286]:
```
get_xi (generic function with 1 method)
```

**Parameters**

- `L`: `LinearDifferentialOperator`
  - Linear differential operator in the differential equation $Lx = 0$; derivatives of $x(t)$ will be entries of $\xi(t)$.
- `symbolic`: `Bool`
  - Boolean indicating whether the output is symbolic.
- `substitute`*: `Bool`
  - If `symbolic = true`, boolean indicating whether to substitute the symbolic expression of $x(t)$ for the generic expression created using `SymFunction`.
- `xSym`*: `SymPy.Sym`
  - If `substitute = true`, symbolic expression of $x(t)$ to replace the generic expression with.

**Returns**

- `get_xi`: `Array` of `SymPy.Sym`
  - Returns an array whose $i$th entry is
    - the generic expression $\frac{d^{i-1}}{dt^{i-1}}x(t)$ if `substitute = false`, or
    - the symbolic expression of the $(i-1)$th derivative of $x(t)$ if `substitute = true`.

**Example**

```
In [287]: t = symbols("t")
          symPFunctions = [1 t+1 t^2+t+1]
          interval = (0, 1)
          symL = SymLinearDifferentialOperator(symPFunctions, interval, t)
          L = get_L(symL)
```

```
Out[287]: LinearDifferentialOperator(#func#164{SymPy.Sym}[func func func], (0, 1), SymLinearDifferentialOperator(SymP
          y.Sym[1 t + 1 t^2 + t + 1], (0, 1), t))
```

```
In [288]: xSym = t^2+2t
          symXi = get_xi(L; symbolic = true, xSym = xSym)
```

```
Out[288]: $$\begin{bmatrix} t^2 + 2t \\ 2t + 2 \end{bmatrix}$$
```

```
In [289]: xi = get_xi(L; symbolic=false, xSym = xSym)
          tVal = 1
          println("xi($tVal) = $(evaluate.(xi, tVal))")
          println("symXi($tVal) = $(evaluate.(symXi, tVal))")
```

```
          xi(1) = [3; 4]
          symXi(1) = [3; 4]
```

## 5.12 get_Ux(L, U, xSym)

Given a `LinearDifferentialOperator` `L` and a `VectorBoundaryForm` `U`, constructs the left hand side

$$Ux = M\xi(a) + N\xi(b)$$

of the homogeneous boundary condition $Ux = 0$, where

$$\begin{aligned} Ux &= \begin{bmatrix} \sum_{j=1}^{n}(M_{1j}x^{(j-1)}(a) + N_{1j}x^{(j-1)}(b)) \\ \vdots \\ \sum_{j=1}^{n}(M_{mj}x^{(j-1)}(a) + N_{mj}x^{(j-1)}(b)) \end{bmatrix} \\ &= \begin{bmatrix} M_{11} & \cdots & M_{1n} & N_{11} & \cdots & N_{1n} \\ \vdots & & \vdots & \vdots & & \vdots \\ M_{m1} & \cdots & M_{mn} & N_{m1} & \cdots & N_{mn} \end{bmatrix} \begin{bmatrix} x(a) \\ \vdots \\ x^{(n-1)}(a) \\ x(b) \\ \vdots \\ x^{(n-1)}(b) \end{bmatrix} \\ &= (M : N) \begin{bmatrix} \xi(a) \\ \xi(b) \end{bmatrix}. \end{aligned}$$







```
In [290]: function get_Ux(L::LinearDifferentialOperator, U::VectorBoundaryForm, xSym)
              (a, b) = L.interval
              xi = get_xi(L; symbolic = false, xSym = xSym)
              xiEvalA = evaluate.(xi, a)
              xiEvalB = evaluate.(xi, b)
              M, N = U.M, U.N
              Ux = M*xiEvalA + N*xiEvalB
              return Ux
          end
```

```
Out[290]: get_Ux (generic function with 1 method)
```

**Parameters**

- `L`: `LinearDifferentialOperator`
  - Linear differential operator in the differential equation $Lx = 0$; derivatives of $x(t)$ will be entries of $\xi(t)$.
- `U`: `VectorBoundaryForm`
  - Vector boundary form in the boundary condition $Ux$.
- `symX*`: `SymPy.Sym`
  - Symbolic expression of $x(t)$ whose derivatives will be entries of $\xi(t)$.

**Returns**

- `get_boundaryCondition`: `Array` of `Number`
  - Returns the $Ux$ in the homogeneous boundary condition $Ux = 0$ defined above.

**Example**

```
In [291]: t = symbols("t")
          symPFunctions = [1 t+1 t^2+t+1]
          interval = (0, 1)
          symL = SymLinearDifferentialOperator(symPFunctions, interval, t)
          L = get_L(symL)
          M = [1 0; 2 0]
          N = [0 2; 0 1]
          U = VectorBoundaryForm(M, N)

          xSym = t^2+2t
          Ux = get_Ux(L, U, xSym)
          println("U($xSym) = $Ux")
```

```
          U(t^2 + 2*t) = [8; 4]
```

## 5.13 check_adjoint(L, U, adjointU, B)

Given a boundary value problem

$$Lx = 0, \quad Ux = 0$$

with linear differential operator $L$ and vector boundary form $U$, a candidate adjoint vector boundary form $U^+$, and the matrix $B$ associated with $L$, checks whether the boundary condition

$$U^+x = 0$$

is indeed adjoint to the boundary condition

$$Ux = 0.$$

In [292]:
```
function check_adjoint(L::LinearDifferentialOperator, U::VectorBoundaryForm, adjointU::VectorBoundaryForm, B
::Array)
    (a, b) = L.interval
    M, N = U.M, U.N
    P, Q = (adjointU.M)', (adjointU.N)'
    # Avoid InexactError() when taking inv()
    BEvalA = convert(Array{Complex}, evaluate.(B, a))
    BEvalB = convert(Array{Complex}, evaluate.(B, b))
    left = M * inv(BEvalA) * P
    right = N * inv(BEvalB) * Q
#     println("left = $left")
#     println("right = $right")
    tol = set_tol(left, right)
    return all(i -> isapprox(left[i], right[i]; atol = tol), 1:length(left)) # Can't use == to deterimine eq
uality because left and right are arrays of floats
end
```

Out[292]: check_adjoint (generic function with 1 method)





**Contents**
7.1 check_boundaryConditions(U, fSym)
7.2 get_MPlusMinus(adjointU; symbolic, generic)
7.3 get_M(adjointU; symbolic, generic)
7.4 get_delta(adjointU; symbolic, generic)
7.5 get_Xlj(adjointU, l, j; symbolic, generic)
7.6 get_ChebyshevIntegral(l, f; symbolic, lambda, alpha)
7.6.1 get_ChebyshevTermInteg(symbolic, c)
7.6.1.1 get_alpha(m, n)
7.6.2 get_ChebyshevCoeffici
7.6.3 get_ChebyshevIntegral(f; symbolic, lambda, alpha)
7.7 get_FPlusMinus(adjointU; symbolic, generic)
7.8 get_gamma(a, n, zeroList; infty, nGon)
7.8.1 get_gammaAAngles(a, n; symbolic)
7.8.2 get_gammaAAnglesSplit(n; symbolic)
7.8.3 pointOnSector(z, sectorAngles)
7.8.4 pointInSector(z, sectorAngles)
7.8.5 pointExSector(z, sectorAngles)
7.8.6 get_epsilon(zeroList, a, n)
7.8.7 get_nGonAroundZero(ze(epsilon, n)
7.8.8 get_gamma(a, n, zeroList; infty, nGon)
7.9 plot_contour(gamma; infty)
7.10 solve_IBVP(L, U, a, zeroList, f; FPlusFunc, FMinusFunc, pDerivMatrix, infty)
8 Verifying the formulas of $F_\lam
8.1 Problem 1
8.2 Problem 2

**Parameters**

- L: LinearDifferentialOperator
  - Linear differential operator in the differential equation $Lx = 0$.
- U: VectorBoundaryForm
  - Vector boundary form in the boundary condition $Ux = 0$.
- adjointU: VectorBoundaryForm
  - Vector boundary form in the candidate adjoint boundary condition $U^+x = 0$.
- B: Array of Number
  - Output of get_B(L).

**Returns**

- check_adjoint: Bool
  - Returns
    - true if adjointU is indeed adjoint to U, or
    - false otherwise.

**Example**

In [293]:
```
t = symbols("t")
symPFunctions = [1 t+1 t^2+t+1]
interval = (0, 1)
symL = SymLinearDifferentialOperator(symPFunctions, interval, t)
L = get_L(symL)
M = [1 0; 2 0]
N = [0 2; 0 1]
# M = [3.9 5.4; 1+2*im 2]
# N = [4.7 8.1 + im; 0.5*im 10]
U = VectorBoundaryForm(M, N)
Uc = get_Uc(U)

# Non-symbolic
B = get_B(L)
BHat = get_BHat(L, B)
H = get_H(U, Uc)
J = get_J(BHat, H)
adjointU = get_adjointCand(J)

check_adjoint(L, U, adjointU, B)
```

Out[293]: true

## 5.14 get_adjointU(L, U, pDerivMatrix)

Given a boundary value problem

$$Lx = p_0 x^{(n)} + p_1 x^{(n-1)} + \cdots + p_{n-1} x^{(1)} + p_n x = 0, \quad Ux = 0$$

with linear differential operator $L$ and vector boundary form $U$, an $n \times n$ matrix of derivatives

$$\begin{bmatrix} p_0 & \cdots & p_0^{(n-1)} \\ \vdots & \ddots & \vdots \\ p_{n-1} & \cdots & p_{n-1}^{(n-1)} \end{bmatrix},$$

construct $U^+$ such that the boundary condition $U^+ = 0$ is adjoint to the original boundary condition $Ux = 0$.

In [294]:
```
function get_adjointU(L::LinearDifferentialOperator, U::VectorBoundaryForm, pDerivMatrix=get_pDerivMatrix(L))
    B = get_B(L; pDerivMatrix = pDerivMatrix)
    BHat = get_BHat(L, B)
    Uc = get_Uc(U)
    H = get_H(U, Uc)
    J = get_J(BHat, H)
    adjointU = get_adjointCand(J)
    if check_adjoint(L, U, adjointU, B)
        return adjointU
    else
        throw(error("Adjoint found not valid"))
    end
end
```

Out[294]: get_adjointU (generic function with 2 methods)

**Parameters**

- L: LinearDifferentialOperator
  - Linear differential operator in the differential equation $Lx = 0$.
- U: VectorBoundaryForm
  - Vectory boundary form in the boundary condition $Ux = 0$.
- pDerivMatrix: Array of Function, Number, or SymPy.#
  - An $n \times n$ matrix defined above, which can be
    - output of get_pDerivMatrix (SymPy.#), or
    - user input.

**Returns**

- get_adjointU: VectorBoundaryForm
  - Returns a valid vector boundary form $U^+$ such that the boundary condition $U^+x = 0$ is adjoint to $Ux = 0$.

**Example**







In [295]:
```
t = symbols("t")
symPFunctions = [1 t+1 t^2+t+1]
interval = (0, 1)
symL = SymLinearDifferentialOperator(symPFunctions, interval, t)
# pFunctions = [t->1 t->t+1 t->t^2+t+1]
# L = LinearDifferentialOperator(pFunctions, interval, symL)
L = get_L(symL)
M = [1 0; 2 0]
N = [0 2; 0 1]
U = VectorBoundaryForm(M, N)

adjointU = get_adjointU(L, U)
```

Out[295]: VectorBoundaryForm(Complex[-1.0-0.0im 0.0-0.0im; 0.0-0.0im 0.0-0.0im], Complex[0.0-0.0im 0.0-0.0im; 2.0-0.0im -1.0-0.0im])

# 6 Approximate roots of exponential polynomial

Helps user to find the roots of an exponential polynomial $\Delta(\lambda)$ where $\lambda \in \mathbb{C}$ by visualizing the roots as the intersections of the level curves $\Re(\Delta) = 0$ and $\Im(\Delta) = 0$.

## 6.1 separate_real_imaginary(delta)

Separates real and imaginary parts of the symbolic expression of $\Delta(\lambda)$, an exponential polynomial in one variable.

### 6.1.1 separate_real_imaginary_exp(expr)

Helper function that deals with the case where the toplevel operation is exponentiation.

In [296]:
```
# although the function body is the same as "power" and "others", this case is isolated because negative exponents, e.g., factor_list(e^(-im*x)), give PolynomialError('a polynomial expected, got exp(-I*x)',), while factor_list(cos(x)) runs normally
function separate_real_imaginary_exp(expr::SymPy.Sym)
    result = real(expr) + im*imag(expr)
    return result
end
```

Out[296]: separate_real_imaginary_exp (generic function with 1 method)

**Parameters**

- expr: SymPy.Sym
  - Symbolic expression in $x + iy$ whose toplevel operation is exponentiation, i.e., `SymPy.func(expr) = SymPy.func(sympyExpExpr)`.

**Returns**

- separate_real_imaginary_exp: SymPy.Sym
  - Returns a symbolic expression whose real and imaginary parts are separated.

**Example**

In [297]:
```
x = symbols("x", real = true)
y = symbols("y", real = true)
expr = e^(x+im*y)
println("func($expr) = $(SymPy.func(expr))")
separate_real_imaginary_exp(expr)
```

```
func(exp(x + I*y)) = exp
```

Out[297]: $ie^x \sin(y) + e^x \cos(y)$

### 6.1.2 separate_real_imaginary_power(expr)

Helper function that deals with the case where the toplevel operation is power.

In [298]:
```
# we won't be dealing with cases like x^(x^x)
function separate_real_imaginary_power(expr::SymPy.Sym)
    result = real(expr) + im*imag(expr)
    return result
end
```

Out[298]: separate_real_imaginary_power (generic function with 1 method)

**Parameters**

- expr: SymPy.Sym
  - Symbolic expression in $x + iy$ whose toplevel operation is power, i.e., `SymPy.func(expr) = SymPy.func(sympyPowerExpr)`.

**Returns**

- separate_real_imaginary_power: SymPy.Sym
  - Returns a symbolic expression whose real and imaginary parts are separated.

**Example**







In [299]:
```
x = symbols("x", real = true)
y = symbols("y", real = true)
expr = (x+im*y)^2
println("func($expr) = $(SymPy.func(expr))")
separate_real_imaginary_power(expr)
```

```
func((x + I*y)^2) = <class 'sympy.core.power.Pow'>
```

Out[299]: $x^2 + 2ixy - y^2$

### 6.1.3 separate_real_imaginary_mult(expr)

Helper function that deals with the case where the toplevel operation is multiplication.

In [300]:
```
function separate_real_imaginary_mult(expr::SymPy.Sym)
    terms = args(expr)
    result = 1
    # if the expanded expression contains toplevel multiplication, the individual terms must all be exponentials or powers
    for term in terms
        # println("term = $term")
        # if term is exponential
        if SymPy.func(term) == SymPy.func(sympyExpExpr)
            termSeparated = separate_real_imaginary_exp(term)
        # if term is power (not sure if this case and the case below overlaps)
        elseif SymPy.func(term) == SymPy.func(sympyPowerExpr)
            termSeparated = separate_real_imaginary_power(term)
            # else, further split each product term into indivdual factors (this case also includes the case where term is a number, which would go into the "constant" below)
        else
            termSeparated = term # term is a number
#             (constant, factors) = factor_list(term)
#             termSeparated = constant
#             # separate each factor into real and imaginary parts and collect the product of separated factors
#             for (factor, power) in factors
#                 factor = factor^power
#                 termSeparated = termSeparated * (real(factor) + im*imag(factor))
#             end
        end
        # println("termSeparated = $termSeparated")
        # collect the product of separated term, i.e., product of separated factors
        result = result * termSeparated
    end
    result = real(result) + im*imag(result)
    return result
end
```

Out[300]: separate_real_imaginary_mult (generic function with 1 method)

**Parameters**

- expr: SymPy.Sym
  - Symbolic expression in $x + iy$ whose toplevel operation is multiplication, i.e., SymPy.func(expr) = SymPy.func(sympyMultExpr).

**Returns**

- separate_real_imaginary_mult: SymPy.Sym
  - Returns a symbolic expression whose real and imaginary parts are separated.

**Example**

In [301]:
```
x = symbols("x", real = true)
y = symbols("y", real = true)
expr = (x+im*y)*2x
println("func($expr) = $(SymPy.func(expr))")
separate_real_imaginary_mult(expr)
```

```
func(2*x*(x + I*y)) = <class 'sympy.core.mul.Mul'>
```

Out[301]: $2x^2 + 2ixy$

### 6.1.4 separate_real_imaginary_add(expr)

Helper function that deals with the case where the toplevel operation is addition.







In [302]:
```
function separate_real_imaginary_add(expr::SymPy.Sym)
    x = symbols("x")
    # if the expanded expression contains toplevel addition, the individual terms must all be products or sy
mbols
    terms = args(expr)
    result = 0
    # termSeparated = 0 # to avoid undefined error if there is no else (case incomplete)
    for term in terms
        # println("term = $term")
        # if term is a symbol
        if SymPy.func(term) == SymPy.func(x)
            termSeparated = term
        # if term is exponential
        elseif SymPy.func(term) == SymPy.func(sympyExpExpr)
            termSeparated = separate_real_imaginary_exp(term)
        # if term is a power
        elseif SymPy.func(term) == SymPy.func(sympyPowerExpr)
            termSeparated = separate_real_imaginary_power(term)
        # if term is a product
        elseif SymPy.func(term) == SymPy.func(sympyMultExpr)
            termSeparated = separate_real_imaginary_mult(term)
        # if term is a number
        else
            termSeparated = term
        end
        # println("termSeparated = $termSeparated")
        result = result + termSeparated
    end
    result = real(result) + im*imag(result)
    return result
end
```

Out[302]: separate_real_imaginary_add (generic function with 1 method)

**Parameters**

- expr: SymPy.Sym
  - Symbolic expression in $x + iy$ whose toplevel operation is addition, i.e., SymPy.func(expr) = SymPy.func(sympAddExpr).

**Returns**

- separate_real_imaginary_add: SymPy.Sym
  - Returns a symbolic expression whose real and imaginary parts are separated.

**Example**

In [303]:
```
x = symbols("x", real = true)
y = symbols("y", real = true)
expr = (x+im*y) + 2x
println("func($expr) = $(SymPy.func(expr))")
separate_real_imaginary_add(expr)
```

```
func(3*x + I*y) = <class 'sympy.core.add.Add'>
```

Out[303]: $3x + iy$

### 6.1.5 separate_real_imaginary_power_mult_add(expr)

Helper function that deals with the cases where the toplevel operation is power, addition, or multiplication.

In [304]:
```
function separate_real_imaginary_power_mult_add(expr::SymPy.Sym)
    if SymPy.func(expr) == SymPy.func(sympyPowerExpr)
        result = separate_real_imaginary_power(expr)
    elseif SymPy.func(expr) == SymPy.func(sympyMultExpr)
        result = separate_real_imaginary_mult(expr)
        else #if SymPy.func(expr) == SymPy.func(sympyAddExpr)
        result = separate_real_imaginary_add(expr)
#     else
#         result = expr
    end
    return result
end
```

Out[304]: separate_real_imaginary_power_mult_add (generic function with 1 method)

**Parameters**

- expr: SymPy.Sym
  - Symbolic expression in $x + iy$ whose toplevel operation is power, multiplication, or addition, i.e., SymPy.func(expr) = SymPy.func(sympPowerExpr), SymPy.func(sympMultExpr), or SymPy.func(sympAddExpr).

**Returns**

- separate_real_imaginary_power_mult_add: SymPy.Sym
  - Returns a symbolic expression whose real and imaginary parts are separated.

**Example**







In [305]:
```
x = symbols("x", real = true)
y = symbols("y", real = true)
expr = (x+im*y) + 2x
println("func($expr) = $(SymPy.func(expr))")
separate_real_imaginary_power_mult_add(expr)
```

```
func(3*x + I*y) = <class 'sympy.core.add.Add'>
```

Out[305]: $3x + iy$

### 6.1.6 separate_real_imaginary_others(expr)

Helper function that deals with the case where the toplevel operation is not exponentiation, power, multiplication or addition. In this case, expr must be a single term, e.g., x or cos(2x+1), which is a function wrapping around an expression. So we use the other helper functions to separate the expression wrapped in the function and feed it back to the function.

In [306]:
```
function separate_real_imaginary_others(expr::SymPy.Sym)
    # if the expanded expression is neither of the above, it must be a single term, e.g., x or cos(2x+1), wh
ich is a function wrapping around an expression; in this case, use the helper function to clean up the expre
ssion and feed it back to the function
    term = args(expr)[1]
    termCleaned = separate_real_imaginary_power_mult_add(term)
    result = subs(expr,args(expr)[1],termCleaned)
    result = real(result) + im*imag(result)
    return result
end
```

Out[306]: separate_real_imaginary_others (generic function with 1 method)

**Parameters**

- expr: SymPy.Sym
  - Symbolic expression in $x + iy$ whose toplevel operation is not power, multiplication, or addition, i.e., SymPy.func(expr) != SymPy.func(sympPowerExpr), SymPy.func(sympMultExpr), or SymPy.func(sympAddExpr).

**Returns**

- separate_real_imaginary_others: SymPy.Sym
  - Returns a symbolic expression whose real and imaginary parts are separated.

**Example**

In [307]:
```
x = symbols("x", real = true)
y = symbols("y", real = true)
expr = cos(x+im*y)
println("func($expr) = $(SymPy.func(expr))")
separate_real_imaginary_others(expr)
```

```
func(cos(x + I*y)) = cos
```

Out[307]: $-i\sin(x)\sinh(y)+\cos(x)\cosh(y)$

### 6.1.7 separate_real_imaginary(delta)

The main function that separates the real and imaginary parts of $\Delta$, an exponential polynomial in one variable.







In [308]:
```
function separate_real_imaginary(delta::SymPy.Sym)
    x = symbols("x", real = true)
    y = symbols("y", real = true)

    freeSymbols = free_symbols(delta)
    # check if delta has one and only one free symbol (e.g., global variable lambda)
    if length(freeSymbols) == 1
        lambda = freeSymbols[1]
        # substitute lambda with x+iy
        expr = subs(delta, lambda, x+im*y)
        # expand the new expression
        expr = expand(expr)

        if SymPy.func(expr) == SymPy.func(sympyPowerExpr)
#             println(expr)
#             println("power!")
            result = separate_real_imaginary_power(expr)
#             println("result = $result")
        elseif SymPy.func(expr) == SymPy.func(sympyAddExpr)
#             println(expr)
#             println("addition!")
            result = separate_real_imaginary_add(expr)
#             println("result = $result")
        elseif SymPy.func(expr) == SymPy.func(sympyMultExpr)
#             println(expr)
#             println("multiplication!")
            result = separate_real_imaginary_mult(expr)
#             println("result = $result")
        else
#             println(expr)
#             println("single term!")
            result = separate_real_imaginary_others(expr)
#             println("result = $result")
        end
        result = expand(result)
        return real(result) + im*imag(result)

    else
        throw("Delta has more than one variable")
    end
end
```

Out[308]: separate_real_imaginary (generic function with 1 method)

$$\Delta(\lambda)=\lambda+1=x+iy+1.$$

In [309]:
```
lambda = symbols("lambda")
delta = lambda + 1
separate_real_imaginary(delta)
```

Out[309]: $x+iy+1$

$$\Delta(\lambda) = e^{\lambda} = e^{x+iy} = e^x e^{iy} = e^x \cos(y) + ie^x \sin(y)).$$

In [310]:
```
delta = e^(lambda)
separate_real_imaginary(delta)
```

Out[310]: $ie^x \sin(y) + e^x \cos(y)$

$$\Delta(\lambda) = \lambda^2 = (x+iy)^2 = x^2 - y^2 + i2xy.$$

In [311]:
```
delta = lambda^2
separate_real_imaginary(delta)
```

Out[311]: $x^2 + 2ixy - y^2$

$$\begin{aligned}\Delta(\lambda) &= \cos(\lambda) \\ &= \frac{1}{2}e^{i\lambda} + \frac{1}{2}e^{-i\lambda} \\ &= \frac{1}{2}e^{i(x+iy)} + \frac{1}{2}e^{-i(x+iy)} \\ &= \frac{1}{2}(e^{-y}e^{ix} + e^{y}e^{-ix}) \\ &= \frac{1}{2}e^{-y}(\cos(x) + i\sin(x)) + \frac{1}{2}e^{y}(\cos(x) - i\sin(x)) \\ &= \cos(x)\cosh(y) - i\sin(x)\sinh(y)\end{aligned}$$

In [312]:
```
delta = cos(lambda)
separate_real_imaginary(delta)
```

Out[312]: $-i \sin(x) \sinh(y) + \cos(x) \cosh(y)$

$$\begin{aligned}\Delta(\lambda) &= \cos(\lambda)e^{\lambda} \\ &= \cos(x+iy)e^{x+iy} \\ &= \cos(x+iy)e^x e^{iy} \\ &= [\cos(x)\cosh(y) - i\sin(x)\sinh(y)]\, e^x(\cos(y) + i\sin(y)) \\ &= e^x\cos(x)\cosh(y)\cos(y) + e^x\sin(x)\sinh(y)\sin(y) + i(e^x\cos(x)\cosh(y)\sin(y) - e^x\sin(x)\sinh(y)\cos(y)).\end{aligned}$$

In [313]:
```
delta = cos(lambda)*e^(lambda)
separate_real_imaginary(delta)
```

Out[313]: $i\left(-e^x \sin(x) \cos(y) \sinh(y) + e^x \sin(y) \cos(x) \cosh(y)\right) + e^x \sin(x) \sin(y) \sinh(y) + e^x \cos(x) \cos(y) \cosh(y)$

$$\Delta(\lambda) = (\lambda^3 + \lambda + 2)e^{\lambda}$$







In [314]:
```
delta = (lambda^3+lambda+2)*e^(lambda)
separatedDelta = separate_real_imaginary(delta)
prettyPrint(separatedDelta)
# Verified with WolframAlpha
```

Out[314]:
$$x^3e^x\cos(y) - 3x^2ye^x\sin(y) - 3xy^2e^x\cos(y) + xe^x\cos(y) + y^3e^x\sin(y) - ye^x\sin(y)$$
$$+ i\left(x^3e^x\sin(y) + 3x^2ye^x\cos(y) - 3xy^2e^x\sin(y) + xe^x\sin(y) - y^3e^x\cos(y) + ye^x\cos(y) + 2e^x\sin(y)\right)$$
$$+ 2e^x\cos(y)$$

## 6.2 plot_levelCurves(bivariateDelta; realFunc, imagFunc, xRange, yRange, step, width, height)

Plots the level curves $\Re(\Delta(x+iy)) = 0$ and $\Im(\Delta(x+iy)) = 0$ in the $\Re - \Im$ plane.

In [315]:
```
function plot_levelCurves(bivariateDelta::SymPy.Sym; realFunc = real(bivariateDelta), imagFunc = imag(bivariateDelta), xRange = (-INFTY, INFTY), yRange = (-INFTY, INFTY), step = INFTY/1000, width = 1500, height = 1000)
    freeSymbols = free_symbols(bivariateDelta)
    x = symbols("x", real = true)
    y = symbols("y", real = true)

    xGridStep = (xRange[2] - xRange[1])/50
    yGridStep = (yRange[2] - yRange[1])/50

    if freeSymbols == [x, y]
        Plots.contour(xRange[1]:step:xRange[2], yRange[1]:step:yRange[2], realFunc, levels=[0], size = (width, height), tickfontsize = 20, seriescolor=:reds, transpose = false, linewidth = 4, linealpha = 1, xticks = xRange[1]:xGridStep:xRange[2], yticks = yRange[1]:yGridStep:yRange[2], grid = true, gridalpha = 0.5)
        Plots.contour!(xRange[1]:step:xRange[2], yRange[1]:step:yRange[2], imagFunc, levels=[0], size = (width, height), tickfontsize = 20, seriescolor=:blues, transpose = false, linewidth = 4, linealpha = 1, xticks = xRange[1]:xGridStep:xRange[2], yticks = yRange[1]:yGridStep:yRange[2], grid = true, gridalpha = 0.5)
    else
        Plots.contour(xRange[1]:step:xRange[2], yRange[1]:step:yRange[2], realFunc, levels=[0], size = (width, height), tickfontsize = 20, seriescolor=:reds, transpose = true, linewidth = 4, linealpha = 1, xticks = xRange[1]:xGridStep:xRange[2], yticks = yRange[1]:yGridStep:yRange[2], grid = true, gridalpha = 0.5)
        Plots.contour!(xRange[1]:step:xRange[2], yRange[1]:step:yRange[2], imagFunc, levels=[0], size = (width, height), tickfontsize = 20, seriescolor=:blues, transpose = true, linewidth = 4, linealpha = 1, xticks = xRange[1]:xGridStep:xRange[2], yticks = yRange[1]:yGridStep:yRange[2], grid = true, gridalpha = 0.5)
    end
end
```

Out[315]: plot_levelCurves (generic function with 1 method)

**Parameters**

- `bivariateDelta`: `SymPy.Sym`
  - Symbolic expression of $\Delta(\lambda)$ with $\lambda$ replaced by $x + iy$.
- `realFunc*`, `imagFunc*`: `SymPy.Sym` or `Function`
  - Real and imaginary parts of $\Delta(\lambda)$. Default to `real(separatedDelta)` and `imag(separatedDelta)`. If input manually, they need to be functions in two variables $x$, $y$ where $\lambda = x + iy$.
- `xRange*`, `yRange*`: `Tuple{Number, Number}`
  - Ranges of $\Re(\Delta(\lambda))$ and $\Im(\Delta(\lambda))$ in the plot. Default to `(-INFTY, INFTY)`.
- `step*`: `Number`
  - The distance between two points that are plotted as projected onto the x and y axes, i.e., the step in the sequence of points in x and y to be plotted.
- `width*`, `height*`: `Number`
  - Width and height of the plot.

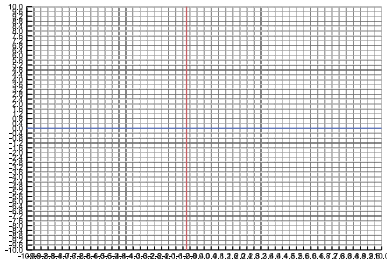
**Returns**

- `plot_levelCurves`: `None`
  - Plots the contour plots without returning them.

**Example**

In [316]:
```
lambda = symbols("lambda")
x = symbols("x", real = true)
y = symbols("y", real = true)
delta = lambda + 1
bivariateDelta = subs(delta, lambda, x+im*y)
# plot_levelCurves(separatedDelta) # somehow causes method error, probably because real(separatedDelta) is a function of x only and imag(separatedDelta) is a function of y only. In this case, we need to input the real Func and imagFunc manually
plot_levelCurves(bivariateDelta; realFunc = (x, y) -> x + 1, imagFunc = (x, y) -> y)
```

Out[316]:



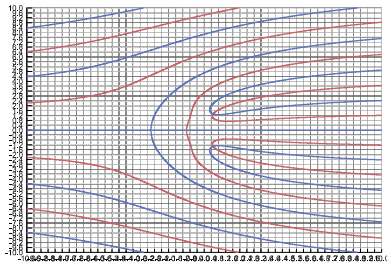






In [317]:
```
delta = (lambda^3+lambda+2)*e^(lambda)
bivariateDelta = subs(delta, lambda, x+im*y)
plot_levelCurves(bivariateDelta)
```

Out[317]:



# 7 The Fokas Transform pair

Implement the transform pair (2.15a), (2.15b) on page 10 of "Evolution PDEs and augmented eigenfunctions. Finite interval," given by

$$F_\lambda : f(x) \mapsto F(\lambda): \quad F_\lambda(f) = \begin{cases} F_\lambda^+(f), & \text{if } \lambda \in \Gamma_0^+ \cup \Gamma_a^+ \\ F_\lambda^-(f), & \text{if } \lambda \in \Gamma_0^- \cup \Gamma_a^- \end{cases}$$

$$f_x : F(\lambda) \mapsto f(x): \quad f_x(F) = \int_\Gamma e^{i\lambda x} F(\lambda)\, d\lambda, \quad x \in [0,1].$$

## 7.1 `check_boundaryConditions(L, U, fSym)`

Checks whether $f$ satisfies the boundary conditions, i.e., whether $f \in \Phi$.

In [318]:
```
function check_boundaryConditions(L::LinearDifferentialOperator, U::VectorBoundaryForm, fSym::Union{SymPy.Sym, Number})
    # Checks whether f satisfies the homogeneous boundary conditions
    Ux = get_Ux(L, U, fSym)
    return check_all(Ux, x->is_approx(x, 0))
end
```

Out[318]: `check_boundaryConditions (generic function with 1 method)`

**Parameters**

- L: LinearDifferentialOperator
  - Linear differential operator in the IBVP.
- U: VectorBoundaryForm
  - Vector boundary form in the IBVP.
- fSym: SymPy.Sym or Number
  - Symbolic expression of the initial condition in the IBVP.

**Returns**

- check_boundaryConditions: Bool
  - Returns true if $f$ satisfies the homogeneous boundary conditions within a tolerance (TOL) and false otherwise.

**Example**

In [319]:
```
t = symbols("t")
symPFunctions = [-1 0 0]
interval = (0,1)
symL = SymLinearDifferentialOperator(symPFunctions, interval, t)
L = get_L(symL)
M = [1 0; 0 1]
N = [-1 0; 0 -1]
U = VectorBoundaryForm(M, N)
x = symbols("x")
fSym = sin(2*pi*x)
check_boundaryConditions(L, U, fSym)
```

Out[319]: true

## 7.2 get_MPlusMinus(adjointU; symbolic, generic)

Let $\alpha = e^{2\pi i/n}$; given adjoint vector boundary form associated with two matrices $b^\star$, $\beta^\star$, computes matrices $M^+(\lambda)$, $M^-(\lambda)$ given by

$$M^+_{kj}(\lambda) = \sum_{r=0}^{n-1}(-i\alpha^{k-1}\lambda)^r b^\star_{jr}$$

$$M^-_{kj}(\lambda) = \sum_{r=0}^{n-1}(-i\alpha^{k-1}\lambda)^r \beta^\star_{jr}$$

as functions of $\lambda$ (for fixed $b^\star$, $\beta^\star$) or their symbolic expressions.





**Contents**
7.1 check_boundaryConditions(U, fSym)
7.2 get_MPlusMinus(adjointU; symbolic, generic)
7.3 get_M(adjointU; symbolic, generic)
7.4 get_delta(adjointU; symbolic, generic)
7.5 get_Xlj(adjointU, l, j; symbolic, generic)
7.6 get_ChebyshevIntegral(l, f; symbolic, lambda, alpha)
7.6.1 get_ChebyshevTermInte(symbolic, c)
7.6.1.1 get_alpha(m, n)
7.6.2 get_ChebyshevCoeffici
7.6.3 get_ChebyshevIntegral(f; symbolic, lambda, alpha)
7.7 get_FPlusMinus(adjointU; symbolic, generic)
7.8 get_gamma(a, n, zeroList; infty, nGon)
7.8.1 get_gammaAAngles(a, n; symbolic)
7.8.2 get_gammaAAnglesSplit(n; symbolic)
7.8.3 pointOnSector(z, sectorAngles)
7.8.4 pointInSector(z, sectorAngles)
7.8.5 pointExSector(z, sectorAngles)
7.8.6 get_epsilon(zeroList, a, n)
7.8.7 get_nGonAroundZero(ze(epsilon, n)
7.8.8 get_gamma(a, n, zeroList; infty, nGon)
7.9 plot_contour(gamma; infty)
7.10 solve_IBVP(L, U, a, zeroList, f; FPlusFunc, FMinusFunc, pDerivMatrix, infty)
8 Verifying the formulas of $F_\lam
8.1 Problem 1
8.2 Problem 2

In [320]:
```
function get_MPlusMinus(adjointU::VectorBoundaryForm; symbolic = false, generic = false)
    # these are numeric matrices
    bStar, betaStar = adjointU.M, adjointU.N
    n = size(bStar)[1]
    if symbolic
        # return MPlus and MMinus as symbolic expressions with (the global variable) lambda as free variable
        lambda = symbols("lambda")
```

```
        if generic
            alpha = symbols("alpha")
        else
            alpha = e^(2*PI*im/n)
        end
        MPlusMat = Array{SymPy.Sym}(n,n)
        for k = 1:n
            for j = 1:n
                sumPlus = 0
                for r = 0:(n-1)
                    summandPlus = (-im*alpha^(k-1)*lambda)^r * bStar[j,r+1]
                    sumPlus += summandPlus
                end
                sumPlus = simplify(sumPlus)
                MPlusMat[k,j] = sumPlus
            end
        end
        MPlusSym = MPlusMat
        MMinusMat = Array{SymPy.Sym}(n,n)
        for k = 1:n
            for j = 1:n
                sumMinus = 0
                for r = 0:(n-1)
                    summandMinus = (-im*alpha^(k-1)*lambda)^r * betaStar[j,r+1]
                    sumMinus += summandMinus
                end
                sumMinus = simplify(sumMinus)
                MMinusMat[k,j] = sumMinus
            end
        end
        MMinusSym = MMinusMat
        return (MPlusSym, MMinusSym)
    else
        alpha = e^(2*pi*im/n)
        # if not symbolic, return MPlus and MMinus as functions of lambda
        function MPlus(lambda::Number)
            MPlusMat = Array{Number}(n,n)
            for k = 1:n
                for j = 1:n
                    sumPlus = 0
                    for r = 0:(n-1)
                        summandPlus = (-im*alpha^(k-1)*lambda)^r * bStar[j,r+1]
                        sumPlus += summandPlus
                    end
                    MPlusMat[k,j] = sumPlus
                end
            end
            return MPlusMat
        end
        function MMinus(lambda::Number)
            MMinusMat = Array{Number}(n,n)
            for k = 1:n
                for j = 1:n
                    sumMinus = 0
```







```
                    for r = 0:(n-1)
                        summandMinus = (-im*alpha^(k-1)*lambda)^r * betaStar[j,r+1]
                        sumMinus += summandMinus
                    end
                    MMinusMat[k,j] = sumMinus
                end
            end
            return MMinusMat
        end
    end
    return (MPlus, MMinus)
end
```

```
Out[320]: get_MPlusMinus (generic function with 1 method)
```

**Parameters**

- adjointU: VectorBoundaryForm
  - Output of get_adjointU().
- symbolic*: Bool
  - Boolean indicating whether the output is symbolic.
- generic*: Bool
  - If symbolic = true, boolean indicating whether to keep $\alpha = e^{2\pi i/n}$ as a symbol.

**Returns**

- get_MPlusMinus: Function or SymPy.Sym
  - Returns $M^+(\lambda), M^-(\lambda)$ as functions if symbolic = false and as symbolic expressions if symbolic = true.

**Example**

```
In [321]: t = symbols("t")
          symPFunctions = [1 t+1 t^2+t+1]
          interval = (0, 1)
          symL = SymLinearDifferentialOperator(symPFunctions, interval, t)
          L = get_L(symL)
          M = [1 0; 2 0]
          N = [0 2; 0 1]
          U = VectorBoundaryForm(M, N)
          adjointU = get_adjointU(L, U)
```

```
Out[321]: VectorBoundaryForm(Complex[-1.0-0.0im 0.0-0.0im; 0.0-0.0im 0.0-0.0im], Complex[0.0-0.0im 0.0-0.0im; 2.0-0.0
          im -1.0-0.0im])
```

```
In [322]: (MPlusSym, MMinusSym) = get_MPlusMinus(adjointU; symbolic = true, generic = false)
          prettyPrint.(MMinusSym)
```

Out[322]:
$$\begin{bmatrix} 0 & i\lambda + 2 \\ 0 & -i\lambda + 2 \end{bmatrix}$$

In [323]:
```
(MPlus, MMinus) = get_MPlusMinus(adjointU; symbolic = false)
lambda = 1+im
println("MPlus($lambda) = $(MPlus(lambda))")
println("MMinus($lambda) = $(MMinus(lambda))")
println("MPlusSym($lambda) = $(prettyPrint.(evaluate.(MPlusSym, lambda)))")
println("MMinusSym($lambda) = $(prettyPrint.(evaluate.(MMinusSym, lambda)))")
```

```
MPlus(1 + 1im) = Number[-1.0-0.0im 0.0+0.0im; -1.0+0.0im 0.0+0.0im]
MMinus(1 + 1im) = Number[0.0+0.0im 1.0+1.0im; 0.0+0.0im 3.0-1.0im]
MPlusSym(1 + 1im) = SymPy.Sym[-1 0; -1 0]
MMinusSym(1 + 1im) = SymPy.Sym[0 1 + I; 0 3 - I]
```

## 7.3 get_M(adjointU; symbolic, generic)

Computes $M$ given by

$$M_{kj}(\lambda) = M^{+}_{kj}(\lambda) + M^{-}_{kj}(\lambda)e^{-i\alpha^{k-1}\lambda}$$

as a function of $\lambda$ or its symbolic expression.







In [324]:
```
function get_M(adjointU::VectorBoundaryForm; symbolic = false, generic = false)
    bStar, betaStar = adjointU.M, adjointU.N
    n = size(bStar)[1]
    if symbolic
        # return M as a symbolic expression with lambda as free variable
        lambda = symbols("lambda")
        if generic
            alpha = symbols("alpha")
        else
            alpha = e^(2*PI*im/n)
        end
        (MPlusSym, MMinusSym) = get_MPlusMinus(adjointU; symbolic = true, generic = generic)
        MLambdaSym = Array{SymPy.Sym}(n,n)
        for k = 1:n
            for j = 1:n
                MLambdaSym[k,j] = simplify(MPlusSym[k,j] + MMinusSym[k,j] * e^(-im*alpha^(k-1)*lambda))
            end
        end
        MSym = simplify.(MLambdaSym)
        return MSym
    else
        alpha = e^(2*pi*im/n)
        function M(lambda::Number)
            (MPlus, MMinus) = get_MPlusMinus(adjointU)
            MPlusLambda, MMinusLambda = MPlus(lambda), MMinus(lambda)
            MLambda = Array{Number}(n,n)
            for k = 1:n
                for j = 1:n
                    MLambda[k,j] = MPlusLambda[k,j] + MMinusLambda[k,j] * e^(-im*alpha^(k-1)*lambda)
                end
            end
            return MLambda
        end
        return M
    end
end
```

Out[324]: get_M (generic function with 1 method)

**Parameters**

- adjointU: VectorBoundaryForm
  - Output of get_adjointU().
- symbolic*: Bool
  - Boolean indicating whether the output is symbolic.
- generic*: Bool
  - If symbolic = true, boolean indicating whether to keep $\alpha = e^{2\pi i/n}$ as a symbol.

**Returns**

- get_M: Function or SymPy.Sym
  - Returns $M(\lambda)$ as function if symbolic = false and as symbolic expression if symbolic = true.

**Example**

In [325]:
```
t = symbols("t")
symPFunctions = [1 t+1 t^2+t+1]
interval = (0, 1)
symL = SymLinearDifferentialOperator(symPFunctions, interval, t)
L = get_L(symL)

M = [1 0; 2 0]
N = [0 2; 0 1]
U = VectorBoundaryForm(M, N)

adjointU = get_adjointU(L, U)
```

Out[325]:
```
VectorBoundaryForm(Complex[-1.0-0.0im 0.0-0.0im; 0.0-0.0im 0.0-0.0im], Complex[0.0-0.0im 0.0-0.0im; 2.0-0.0im -1.0-0.0im])
```

In [326]:
```
MSym = get_M(adjointU; symbolic = true, generic = false)
prettyPrint.(MSym)
```

Out[326]:
$$\begin{bmatrix} -1 & (i\lambda+2)\,e^{-i\lambda} \\ -1 & (-i\lambda+2)\,e^{i\lambda} \end{bmatrix}$$

In [327]:
```
M = get_M(adjointU; symbolic = false)
lambda = 1+im
println("M($lambda) = $(M(lambda))")
println("MSym($lambda) = $(evaluate.(sym_to_func.(MSym), lambda))")
```

```
M(1 + 1im) = Number[-1.0+0.0im 3.75605-0.818661im; -1.0+0.0im 0.905858+0.729914im]
MSym(1 + 1im) = Number[-1.0 3.75605-0.818661im; -1.0 0.905858+0.729914im]
```







## 7.4 get_delta(adjointU; symbolic, generic)

Computes $\Delta := \det(M)$ as a function of $\lambda$ (for fixed adjointU) or its symbolic expression.

In [328]:
```
function get_delta(adjointU::VectorBoundaryForm; symbolic = false, generic = false)
    if symbolic
        MSym = get_M(adjointU; symbolic = true, generic = generic)
        deltaSym = simplify(SymPy.det(MSym))
        return deltaSym
    else
       function delta(lambda::Number)
            M = get_M(adjointU; symbolic = false)
            MLambda = convert(Array{Complex}, M(lambda))
            return det(MLambda)
        end
        return delta
    end
end
```

Out[328]:
```
get_delta (generic function with 1 method)
```

**Parameters**

- adjointU: VectorBoundaryForm
  - Output of get_adjointU().
- symbolic*: Bool
  - Boolean indicating whether the output is symbolic.
- generic*: Bool
  - If symbolic = true, boolean indicating whether to keep $\alpha = e^{2\pi i/n}$ as a symbol.

**Returns**

- get_delta: Function or SymPy.Sym
  - Returns $\Delta(\lambda)$ as function if symbolic = false and as symbolic expression if symbolic = true.

**Example**

```
In [329]:
t = symbols("t")
symPFunctions = [1 t+1 t^2+t+1]
interval = (0, 1)
symL = SymLinearDifferentialOperator(symPFunctions, interval, t)
L = get_L(symL)

M = [1 0; 2 0]
N = [0 2; 0 1]
U = VectorBoundaryForm(M, N)

adjointU = get_adjointU(L, U)
```

```
Out[329]: VectorBoundaryForm(Complex[-1.0-0.0im 0.0-0.0im; 0.0-0.0im 0.0-0.0im], Complex[0.0-0.0im 0.0-0.0im; 2.0-0.0im -1.0-0.0im])
```

```
In [330]:
deltaSym = simplify(get_delta(adjointU; symbolic = true, generic = false))
prettyPrint(deltaSym)
```

Out[330]: $\left(i\lambda + (i\lambda - 2)\, e^{2i\lambda} + 2\right) e^{-i\lambda}$

```
In [331]:
delta = get_delta(adjointU; symbolic = false)
lambda = 1+im
println("delta($lambda) = $(delta(lambda))")
println("deltaSym($lambda) = $(evaluate(deltaSym, lambda))")
```

```
delta(1 + 1im) = 2.8501910204023764 - 1.548574863875881im
deltaSym(1 + 1im) = 2.8501910204023764 - 1.548574863875881im
```

## 7.5 get_Xlj(adjointU, l, j; symbolic, generic)

Gets $X_{lj}$, which is the $(n-1)\times(n-1)$ submatrix of $M(\lambda)$ whose 11-entry is the $(l+1)(j+1)$-entry of $M(\lambda)$, as a function of $\lambda$ (for fixed `adjointU`, $M$, $l$, $j$), or its symbolic expression.







```
In [332]:
function get_Xlj(adjointU::VectorBoundaryForm, l::Number, j::Number; symbolic = false, generic = false)
    bStar, betaStar = adjointU.M, adjointU.N
    n = size(bStar)[1]
    if symbolic
        MSym = get_M(adjointU; symbolic = true, generic = generic)
        MBlockSym = [MSym MSym; MSym MSym]
        XljSym = MBlockSym[(l+1):(l+1+n-2), (j+1):(j+1+n-2)]
        return XljSym
    else
        M = get_M(adjointU; symbolic = false)
        function Xlj(lambda::Number)
            MLambda = M(lambda)
            MLambdaBlock = [MLambda MLambda; MLambda MLambda]
            XljLambda = MLambdaBlock[(l+1):(l+1+n-2), (j+1):(j+1+n-2)]
            return XljLambda
        end
        return Xlj
    end
end
```

```
Out[332]: get_Xlj (generic function with 1 method)
```

**Parameters**

- `adjointU`: `VectorBoundaryForm`
  - Output of `get_adjointU()`.
- `l`, `j`: `Int`
  - Indices specifying the submatrix of $M$ that is $X_{lj}$.
- `symbolic*`: `Bool`
  - Boolean indicating whether the output is symbolic.
- `generic*`: `Bool`
  - If `symbolic = true`, boolean indicating whether to keep $\alpha = e^{2\pi i/n}$ as a symbol.

**Returns**

- `get_Xlj`: `Array` of `Function` or `SymPy.Sym`
  - Returns $X_{lj}(\lambda)$ as a matrix of functions if `symbolic = false` and as symbolic expressions if `symbolic = true`.

**Example**

In [333]:
```
t = symbols("t")
symPFunctions = [1 t+1 t^2+t+1]
interval = (0, 1)
symL = SymLinearDifferentialOperator(symPFunctions, interval, t)
L = get_L(symL)

M = [1 0; 2 0]
N = [0 2; 0 1]
U = VectorBoundaryForm(M, N)

adjointU = get_adjointU(L, U)
```

Out[333]:
```
VectorBoundaryForm(Complex[-1.0-0.0im 0.0-0.0im; 0.0-0.0im 0.0-0.0im], Complex[0.0-0.0im 0.0-0.0im; 2.0-0.0im -1.0-0.0im])
```

In [334]:
```
l, j = 1, 1
XljSym = prettyPrint.(get_Xlj(adjointU, l, j; symbolic = true))
```

Out[334]: $\left[ (-i\lambda + 2)\, e^{i\lambda} \right]$

In [335]:
```
Xlj = get_Xlj(adjointU, l, j; symbolic = false)
lambda = 1+1im
println("Xlj($lambda) = $(Xlj(lambda))")
println("XljSym($lambda) = $(evaluate.(XljSym, lambda))")
```

```
Xlj(1 + 1im) = Number[0.905858+0.729914im]
XljSym(1 + 1im) = Complex{Float64}[0.905858+0.729914im]
```

## 7.6 get_ChebyshevIntegral(l, f; symbolic, lambda, alpha)

Computes

$$\int_0^1 e^{-i\alpha^{l-1}\lambda x} f(x)\, dx$$

by approximating $f(x)$ using Chebyshev polynomials and obtaining an explicit expression for the integral.







### 7.6.1 get_ChebyshevTermIntegral(n; symbolic, c)

Computes $\tilde{T}_n(c)$ given by

$$\tilde{T}_n(c) = \int_0^\pi e^{-ic\cos\theta}\cos(n\theta)\sin\theta\, d\theta = \begin{cases} 2 & \text{if } n = 0 \\ 0 & \text{if } n = 1 \\ \frac{(-1)^{n+1}-1}{n^2-1} & \text{if } n \geq 2 \end{cases}, \qquad \text{if } c = 0$$

$$= \sum_{m=1}^{n+1} \alpha(m,n) \left[ \frac{e^{i\lambda}}{(i\lambda)^m} + (-1)^{m+n} \frac{e^{-i\lambda}}{(i\lambda)^m} \right] \quad \text{if } c \neq 0,$$

where

$$\alpha(m,n) = \begin{cases} (-1)^n & \text{if } m = 1 \\ (-1)^{n+1} n^2 & \text{if } m = 2 \\ (-1)^{n+m-1} 2^{m-2} n \sum_{k=1}^{n-m+2} \binom{m+k-3}{k-1} \prod_{j=k}^{m+k-3} (n-j) & \text{else} \end{cases}$$

or its symbolic expression.

#### 7.6.1.1 get_alpha(m, n)

Computes $\alpha(m,n)$.

In [336]:
```
function get_alpha(m::Int, n::Int)
    if m == 1
        result = (-1)^n
    elseif m == 2
        result = (-1)^(n+1)*n^2
    else
        sum = 0
        for k = 1:(n-m+2)
            product = 1
            for j = k:(m+k-3)
                product *= (n-j)
            end
            sum += binomial(m+k-3, k-1) * product
        end
        result = (-1)^(n+m-1)*2^(m-2)*n*sum
    end
    return result
end
```

Out[336]:
```
get_alpha (generic function with 1 method)
```

**Parameters**

- m, n: Int
  - Indices in $\alpha(m,n)$.

**Returns**

- get_alpha: Number
  - Returns the value of $\alpha(m,n)$.

**Example**

In [337]:
```
n = 3
println("alpha(1,$n) = $(get_alpha(1, n))")
println("alpha(2,$n) = $(get_alpha(2, n))")
println("alpha(3,$n) = $(get_alpha(3, n))")
```

```
alpha(1,3) = -1
alpha(2,3) = 9
alpha(3,3) = -24
```







In [338]:
```
function get_ChebyshevTermIntegral(n::Int; symbolic = true, c = symbols("c"))
    if symbolic
        if c == 0
            if n == 0
                expr = 2
            elseif n == 1
                expr = 0
            else
                expr = ((-1)^(n+1)-1)/(n^2-1)
            end
        else
            expr = 0
            for m = 1:(n+1)
                summand = get_alpha(m, n) * (e^(im*c)/(im*c)^m  + (-1)^(m+n)*e^(-im*c)/(im*c)^m)
                expr += summand
            end
        end
        expr = simplify(expr)
        return expr
    else
        function TTilde(c)
            if c == 0
                if n == 0
                    result = 2
                elseif n == 1
                    result = 0
                else
                    result = ((-1)^(n+1)-1)/(n^2-1)
                end
            else
                result = 0
                for i = 1:(n+1)
                    summand = get_alpha(i, n) * (e^(im*c)/(im*c)^i  + (-1)^(i+n)*e^(-im*c)/(im*c)^i)
                    result += summand
                end
            end
            return result
        end
        return TTilde
    end
end
```

Out[338]: get_ChebyshevTermIntegral (generic function with 1 method)

**Parameters**

- n: Int
  - $n$ in $\tilde{T}_n$.
- symbolic*: Bool
  - Boolean indicating whether the output is symbolic.
- c*: SymPy.Sym
  - If symbolic = true, argument of $\tilde{T}_n$. Default to symbols("c").

**Returns**

- get_ChebyshevTermintegral: Function or SymPy.Sym
  - Returns $\tilde{T}_n(c)$ as function of $c$ if symbolic = false and as symbolic expression if symbolic = true.

**Example**

```
In [339]: # alpha = e^(2pi*im/1)
          # lambda = symbols("lambda")
          # c = alpha^(l-1)*lambda/2
          chebTermIntSym = get_ChebyshevTermIntegral(2; symbolic = true)
          prettyPrint(chebTermIntSym)
```

Out[339]:
$$\frac{\left(-ic^2e^{2ic} + ic^2 + 4ce^{2ic} + 4c + 4ie^{2ic} - 4i\right)e^{-ic}}{c^3}$$

```
In [340]: chebTermInt = get_ChebyshevTermIntegral(2; symbolic = false)
          alpha = e^(2pi*im/1)
          lambda = 1+im
          l = 2
          c = alpha^(l-1)*lambda/2
          println("chebTermInt($(prettyPrint(c))) = $(chebTermInt(c))")
          println("chebTermIntSym($(prettyPrint(c))) = $(evaluate(chebTermIntSym, c))")
```

```
chebTermInt(0.5 + 0.5*I) = -0.6684521617501114 - 0.033305777098087574im
chebTermIntSym(0.5 + 0.5*I) = -0.6684521617501128 - 0.033305777098085666im
```







### 7.6.2 get_ChebyshevCoefficients(f)

Obtains the coefficients in the Chebyshev approximation of $f$ on $[0,1]$.

Note that the coefficients are obtained by shifting the interval $[0,1]$ to $[-1,1]$ and then back. That is, define $g:[0,1]\to[-1,1]$ by $g(x)=2x-1$. For $t\in[-1,1]$, define $q(t)=f\circ g^{-1}(t)=f(\frac{t+1}{2})=:f(x)$ for $x\in[0,1]$. Then the returned coefficients are $\{b_0,\dots,b_N\}$ in

$$f(x)=f\circ g^{-1}(g(x))=q(g(x))=q(t)=\sum_{n=0}^{N}b_nT_n(t)=\sum_{n=0}^{N}b_nT_n(g(x))=\sum_{n=0}^{N}b_nT_n(2x-1)$$

instead of $\{a_0,\dots,a_N\}$ in

$$f(x)=\sum_{n=0}^{N}a_nT_n(x).$$

```
In [341]: function get_ChebyshevCoefficients(f::Union{Function,Number})
              fCheb = ApproxFun.Fun(f, 0..1) # Approximate f on [0,1] using chebyshev polynomials
              chebCoefficients = ApproxFun.coefficients(fCheb) # get coefficients of the Chebyshev polynomial
              return chebCoefficients
          end
```

```
Out[341]: get_ChebyshevCoefficients (generic function with 1 method)
```

**Parameters**

- f: Function or Number
  - Function whose Chebyshev coefficients are to be returned.

**Returns**

- get_ChebyshevCoefficients: Array of Number
  - Returns $\{b_0,\dots,b_N\}$.

**Example**

```
In [342]: f(x) = x^2+1
          fChebCoeffs = get_ChebyshevCoefficients(f)
```

```
Out[342]: 3-element Array{Float64,1}:
           1.375
           0.5
           0.125
```

### 7.6.3 get_ChebyshevIntegral(l, f; symbolic, lambda, alpha)

Computes

$$\int_0^1 e^{-i\alpha^{l-1}\lambda x} f(x)\,dx$$

by computing

$$\frac{1}{2e^{ic}}\int_{-1}^{1} e^{-ict} q(t)\,dt = \int_0^1 e^{-i\alpha^{l-1}\lambda x} f(x)\,dx$$

where $c = \frac{\alpha^{l-1}\lambda}{2}$, $q(t) := f \circ g^{-1}(t)$, and

$$\begin{aligned}\int_{-1}^{1} e^{-ict} q(t)\,dt &= \int_{-1}^{1} e^{-ict} \sum_{n=0}^{N} b_n T_n(t)\,dt \\ &= \sum_{n=0}^{N} b_n \int_{-1}^{1} e^{-ict} T_n(t)\,dt \\ &= \sum_{n=0}^{N} b_n \int_{-1}^{1} e^{-ict} \cos(n \cdot \cos^{-1}(t))\,dt \\ &= \sum_{n=0}^{N} b_n \tilde{T}_n(c).\end{aligned}$$





**Contents**
7.1 check_boundaryConditions(U, fSym)
7.2 get_MPlusMinus(adjointU; symbolic, generic)
7.3 get_M(adjointU; symbolic, generic)
7.4 get_delta(adjointU; symbolic, generic)
7.5 get_Xlj(adjointU, l, j; symbolic, generic)
7.6 get_ChebyshevIntegral(l, f; symbolic, lambda, alpha)
7.6.1 get_ChebyshevTermInteg(symbolic, c)
7.6.1.1 get_alpha(m, n)
7.6.2 get_ChebyshevCoeffici
7.6.3 get_ChebyshevIntegral(f; symbolic, lambda, alpha)
7.7 get_FPlusMinus(adjointU; symbolic, generic)
7.8 get_gamma(a, n, zeroList; infty, nGon)
7.8.1 get_gammaAAngles(a, n; symbolic)
7.8.2 get_gammaAAnglesSplit(n; symbolic)
7.8.3 pointOnSector(z, sectorAngles)
7.8.4 pointInSector(z, sectorAngles)
7.8.5 pointExSector(z, sectorAngles)
7.8.6 get_epsilon(zeroList, a, n)
7.8.7 get_nGonAroundZero(ze(epsilon, n)
7.8.8 get_gamma(a, n, zeroList; infty, nGon)
7.9 plot_contour(gamma; infty)
7.10 solve_IBVP(L, U, a, zeroList, f; FPlusFunc, FMinusFunc, pDerivMatrix, infty)
8 Verifying the formulas of $F_\lam
8.1 Problem 1
8.2 Problem 2

In [343]:
```
function get_ChebyshevIntegral(l::Number, f::Union{Function, Number}; symbolic = false, lambda = nothing, al
pha = nothing)
    fChebCoeffs = get_ChebyshevCoefficients(f)
    # Replace coefficients too close to 0 by 0
    # fChebCoeffs = [if is_approx(x, 0) 0 else x end for x in fChebCoeffs]
    if symbolic
        lambda = symbols("lambda")
        c = alpha^(l-1)*lambda/2
        integralSym = 0
        for m = 1:length(fChebCoeffs)
            fChebCoeff = fChebCoeffs[m]
            if is_approx(fChebCoeff, 0)
                continue
            else
            integralSym += fChebCoeffs[m] * get_ChebyshevTermIntegral(m-1; symbolic = true, c = c)
            end
        end
        integralSym = integralSym/(2*e^(im*c))
        integralSym = simplify(integralSym)
        return integralSym
    else
        if isa(lambda, Void) || isa(alpha, Void)
            throw("lambda, alpha required")
        else
            c = alpha^(l-1)*lambda/2
            integral = 0
            for m = 1:length(fChebCoeffs)
                fChebCoeff = fChebCoeffs[m]
                if is_approx(fChebCoeff, 0)
                    continue
                else
                    integral += fChebCoeff * get_ChebyshevTermIntegral(m-1; symbolic = false)(c)
                end
            end
            integral = integral/(2*e^(im*c))
            return integral
        end
    end
end
```

Out[343]: get_ChebyshevIntegral (generic function with 1 method)

**Parameters**

- `l`: `Int`
  - $l$ in the integral.
- `f`: `Function` or `Number`
  - $f(x)$ in the integral.
- `symbolic*` : `Bool`
  - Boolean indicating whether the output is symbolic.
- `lambda*`: `SymPy.Sym` or `Number`
  - $\lambda$ in the integral. Symbolic if `symbolic = true` and numeric if `symbolic = false`. Default to `symbols("lambda")`.
- `alpha*`: `SymPy.Sym` or `Number`
  - $\alpha$ in the integral. Default to `symbols("alpha")`.

**Returns**

- `get_ChebyshevIntegral` : `Function` or `SymPy.Sym`
  - Returns the integral as a symbolic expression if `symbolic = true` and as a function of $\lambda$ if `symbolic = false`.

**Example**

In [344]:
```
alpha = e^(2pi*im/1)
l = 2
# f(x) = x^2+1
f(x) = sin(2*pi*x)
```

Out[344]: f (generic function with 1 method)

In [345]:
```
chebIntSym = simplify(get_ChebyshevIntegral(l, f; symbolic = true, alpha = alpha))
prettyPrint(chebIntSym)
```

Out[345]:
$$-\frac{0.5\left(12.564\lambda^8 e^{i\lambda} - 12.564\lambda^8 - 0.219i\lambda^7 e^{i\lambda} - 0.219i\lambda^7 + 506.239\lambda^6 e^{i\lambda} - 506.239\lambda^6 + 318.277i\lambda^5 e^{i\lambda} + 318.277i\lambda^5 + 12372.486\lambda^4 e^{i\lambda} - 12372.486\lambda^4 - 120132.078i\lambda^3 e^{i\lambda} - 120132.078i\lambda^3 + 2222127.374\lambda^2 e^{i\lambda} - 2222127.374\lambda^2 + 11891179.312i\lambda e^{i\lambda} + 11891179.312i\lambda - 23782358.623e^{i\lambda} + 23782358.623\right)e^{-i\lambda}}{\lambda^{10}}$$







In [346]:
```
lambda = 1+im
chebInt = get_ChebyshevIntegral(l, f; symbolic = false, lambda = lambda, alpha = alpha)

# Directly compute the integral
g(x) = e^(-im*alpha^(l-1)*lambda*x)
directInt = quadgk(mult_func(g,f), 0, 1)[1]

println("chebIntegral($lambda) = $chebInt")
println("chebIntegralSym($lambda) = $(evaluate.(chebIntSym, lambda))")
println("directIntegral = $directInt")
```

```
chebIntegral(1 + 1im) = -0.09279945603348573 + 0.3593425680615589im
chebIntegralSym(1 + 1im) = -0.09279945602468914 + 0.3593425680538266im
directIntegral = -0.09279946736487799 + 0.3593426246245006im
```

## 7.7 get_FPlusMinus(adjointU; symbolic, generic)

Gets $F_\lambda^+, F_\lambda^-$ given by

$$F_\lambda^+(f) = \frac{1}{2\pi\Delta(\lambda)}\sum_{l=1}^{n}\sum_{j=1}(-1)^{(n-1)(l+j)}\det X^{lj}(\lambda)M_{1j}^+(\lambda)\int_0^1 e^{-i\alpha^{l-1}\lambda x}f(x)\,dx,$$

$$F_\lambda^-(f) = \frac{-e^{-i\lambda}}{2\pi\Delta(\lambda)}\sum_{l=1}^{n}\sum_{j=1}(-1)^{(n-1)(l+j)}\det X^{lj}(\lambda)M_{1j}^-(\lambda)\int_0^1 e^{-i\alpha^{l-1}\lambda x}f(x)\,dx.$$

as functions of $f$ or their symbolic expressions.

In [347]:
```
function get_FPlusMinus(adjointU::VectorBoundaryForm; symbolic = false, generic = false)
    bStar, betaStar = adjointU.M, adjointU.N
    n = size(bStar)[1]
    if symbolic
        lambda = symbols("lambda")
        (MPlusSym, MMinusSym) = get_MPlusMinus(adjointU; symbolic = true, generic = generic)
        deltaSym = get_delta(adjointU; symbolic = true, generic = generic)
        if generic
            alpha = symbols("alpha")
            c = symbols("c")
            FT = SymFunction("FT[f]")(c)
            sumPlusSymGeneric = 0
            sumMinusSymGeneric = 0
            for l = 1:n
                summandPlusSymGeneric = 0
                summandMinusSymGeneric = 0
                for j = 1:n
                    XljSym = get_Xlj(adjointU, l, j; symbolic = true, generic = true)
                    integralSymGeneric = subs(FT, c, alpha^(l-1)*lambda)
                    summandPlusSymGeneric += (-1)^((n-1)*(l+j)) * SymPy.det(XljSym) * MPlusSym[1,j] * integr
alSymGeneric
                    summandMinusSymGeneric += (-1)^((n-1)*(l+j)) * SymPy.det(XljSym) * MMinusSym[1,j] * inte
gralSymGeneric
                end
```





**Contents**
7.1 check_boundaryConditions(U, fSym)
7.2 get_MPlusMinus(adjointU; symbolic, generic)
7.3 get_M(adjointU; symbolic, generic)
7.4 get_delta(adjointU; symbolic, generic)
7.5 get_Xlj(adjointU, l, j; symbolic, generic)
7.6 get_ChebyshevIntegral(l, f; symbolic, lambda, alpha)
7.6.1 get_ChebyshevTermInte(symbolic, c)
7.6.1.1 get_alpha(m, n)
7.6.2 get_ChebyshevCoeffici
7.6.3 get_ChebyshevIntegral(f; symbolic, lambda, alpha)
7.7 get_FPlusMinus(adjointU; symbolic, generic)
7.8 get_gamma(a, n, zeroList; infty, nGon)
7.8.1 get_gammaAAngles(a, n; symbolic)
7.8.2 get_gammaAAnglesSplit(n; symbolic)
7.8.3 pointOnSector(z, sectorAngles)
7.8.4 pointInSector(z, sectorAngles)
7.8.5 pointExSector(z, sectorAngles)
7.8.6 get_epsilon(zeroList, a, n)
7.8.7 get_nGonAroundZero(ze(epsilon, n)
7.8.8 get_gamma(a, n, zeroList; infty, nGon)
7.9 plot_contour(gamma; infty)
7.10 solve_IBVP(L, U, a, zeroList, f; FPlusFunc, FMinusFunc, pDerivMatrix, infty)
8 Verifying the formulas of $F_\lam
8.1 Problem 1
8.2 Problem 2

```
                sumPlusSymGeneric += summandPlusSymGeneric
                sumMinusSymGeneric += summandMinusSymGeneric
            end
            FPlusSymGeneric = simplify(1/(2*PI*deltaSym)*sumPlusSymGeneric)
            FMinusSymGeneric = simplify((-e^(-im*lambda))/(2*PI*deltaSym)*sumMinusSymGeneric)
            return (FPlusSymGeneric, FMinusSymGeneric)
        else
            alpha = e^(2*PI*im/n)
            function FPlusSym(f::Union{Function, Number})
                sumPlusSym = 0
                for l = 1:n
                    summandPlusSym = 0
                    for j = 1:n
                        XljSym = get_Xlj(adjointU, l, j; symbolic = true)
                        integralSym = get_ChebyshevIntegral(l, f; symbolic = true, lambda = lambda, alpha =
alpha)
                        summandPlusSym += (-1)^((n-1)*(l+j)) * SymPy.det(XljSym) * MPlusSym[1,j] * integralS
ym
                    end
                    sumPlusSym += summandPlusSym
                end
                return simplify(1/(2*PI*deltaSym)*sumPlusSym)
            end
            function FMinusSym(f::Union{Function, Number})
                sumMinusSym = 0
                for l = 1:n
                    summandMinusSym = 0
                    for j = 1:n
                        XljSym = get_Xlj(adjointU, l, j; symbolic = true)
                        c = alpha^(l-1)*lambda/2
                        integralSym = get_ChebyshevIntegral(l, f; symbolic = true, lambda = lambda, alpha =
alpha)
                        summandMinusSym += (-1)^((n-1)*(l+j)) * SymPy.det(XljSym) * MMinusSym[1,j] * integra
lSym
                    end
                    sumMinusSym += summandMinusSym
                end
                return simplify((-e^(-im*lambda))/(2*PI*deltaSym)*sumMinusSym)
            end
            return (FPlusSym, FMinusSym)
            end
    else
        alpha = e^(2pi*im/n)
        (MPlus, MMinus) = get_MPlusMinus(adjointU; symbolic = false)
        function FPlus(lambda::Number, f::Union{Function, Number})
            MPlusLambda, MMinusLambda = MPlus(lambda), MMinus(lambda)
            M = get_M(adjointU; symbolic = false)
            MLambda = convert(Array{Complex}, M(lambda))
            deltaLambda = det(MLambda) # or deltaLambda = (get_delta(adjointU))(lambda)
            sumPlus = 0
            for l = 1:n
                summandPlus = 0
                for j = 1:n
                    Xlj = get_Xlj(adjointU, l, j; symbolic = false)
```

```
                    XljLambda = convert(Array{Complex}, Xlj(lambda))
                    integral = get_ChebyshevIntegral(l, f; symbolic = false, lambda = lambda, alpha = alpha)
                    summandPlus += (-1)^((n-1)*(l+j)) * det(XljLambda) * MPlusLambda[1,j] * integral
                end
                sumPlus += summandPlus
            end
            return 1/(2pi*deltaLambda)*sumPlus
        end
        function FMinus(lambda::Number, f::Union{Function, Number})
            MPlusLambda, MMinusLambda = MPlus(lambda), MMinus(lambda)
            M = get_M(adjointU; symbolic = false)
            MLambda = convert(Array{Complex}, M(lambda))
            deltaLambda = det(MLambda) # or deltaLambda = (get_delta(adjointU))(lambda)
            sumMinus = 0
            for l = 1:n
                summandMinus = 0
                for j = 1:n
                    Xlj = get_Xlj(adjointU, l, j)
                    XljLambda = convert(Array{Complex}, Xlj(lambda))
                    integral = get_ChebyshevIntegral(l, f; symbolic = false, lambda = lambda, alpha = alpha)
                    summandMinus += (-1)^((n-1)*(l+j)) * det(XljLambda) * MMinusLambda[1,j] * integral
                end
                sumMinus += summandMinus
            end
            return (-e^(-im*lambda))/(2pi*deltaLambda)*sumMinus
        end
        return (FPlus, FMinus)
    end
end
```

Out[347]: get_FPlusMinus (generic function with 1 method)

**Parameters**

- adjointU: VectorBoundaryForm
  - Adjoint vector boundary form associated with the matrices $b^*$ and $\beta^*$ in the definition of $M(\lambda)$.
- symbolic*: Bool
  - Boolean indicating whether the output is symbolic.
- generic*: Bool
  - If symbolic = true, boolean indicating whether $f$ and $\alpha = e^{2\pi i/n}$ are kept as generic symbols.

**Returns**

- get_FPlusMinusLambda: Tuple of Function or SymPy.Sym
  - Returns $F_\lambda^+, F_\lambda^-$ as symbolic expressions of $\lambda$ if symbolic = true (where $\mathrm{FT}[f]$ indicates the Fourier transform integral of $f$, $\int_0^1 e^{-i\alpha^{l-1}\lambda x} f(x)\,dx$) and as functions in $\lambda$ and $f$ if symbolic = false.

**Example**







In [348]:
```
t = symbols("t")
symPFunctions = [1 t+1 t^2+t+1]
interval = (0, 1)
symL = SymLinearDifferentialOperator(symPFunctions, interval, t)
L = get_L(symL)

M = [1 0; 2 0]
N = [0 2; 0 1]
U = VectorBoundaryForm(M, N)

adjointU = get_adjointU(L, U)
```

Out[348]: VectorBoundaryForm(Complex[-1.0-0.0im 0.0-0.0im; 0.0-0.0im 0.0-0.0im], Complex[0.0-0.0im 0.0-0.0im; 2.0-0.0im -1.0-0.0im])

In [349]:
```
# generic f
(FPlusSymGeneric, FMinusSymGeneric) = get_FPlusMinus(adjointU; symbolic = true, generic = true)
prettyPrint(FPlusSymGeneric)
```

Out[349]:
$$\frac{0.5\left(\left(i\lambda+2\right)\mathrm{FT}[\mathrm{f}]\left(\alpha\lambda\right)e^{i\alpha\lambda}-\left(i\alpha\lambda+2\right)\mathrm{FT}[\mathrm{f}]\left(\lambda\right)e^{i\lambda}\right)}{\pi\left(\left(i\lambda+2\right)e^{i\alpha\lambda}-\left(i\alpha\lambda+2\right)e^{i\lambda}\right)}$$

In [350]:
```
f(x) = x^2-1

# Keep lambda as a free symbol
(FPlusSym, FMinusSym) = get_FPlusMinus(adjointU; symbolic = true)
prettyPrint(FPlusSym(f))
```

Out[350]:
$$\frac{-0.625i\lambda^3e^{\frac{3i\lambda}{2}}\sin\left(\frac{\lambda}{2}\right)-0.625i\lambda^3e^{\frac{i\lambda}{2}}\sin\left(\frac{\lambda}{2}\right)-0.188\lambda^3e^{2i\lambda}+0.188\lambda^3+1.25\lambda^2e^{\frac{3i\lambda}{2}}\sin\left(\frac{\lambda}{2}\right)-1.25\lambda^2e^{\frac{i\lambda}{2}}\sin\left(\frac{\lambda}{2}\right)}{\pi\lambda^3\left(i\lambda e^{2i\lambda}+i\lambda-2e^{2i\lambda}+2\right)}$$
$$\frac{-0.375i\lambda^2e^{2i\lambda}+0.75i\lambda^2e^{i\lambda}-0.375i\lambda^2-\lambda e^{2i\lambda}+\lambda-2ie^{2i\lambda}+4ie^{i\lambda}-2i}{\pi\lambda^3\left(i\lambda e^{2i\lambda}+i\lambda-2e^{2i\lambda}+2\right)}$$

In [351]:
```
lambda = 1+im
(FPlus, FMinus) = get_FPlusMinus(adjointU; symbolic = false)
println("FPlus($lambda, f) = $(FPlus(lambda, f))")
println("FMinus($lambda, f) = $(FMinus(lambda, f))")
println("FPlusSym($lambda, f) = $(evaluate.(FPlusSym(f), lambda))")
println("FMinusSym($lambda, f) = $(evaluate.(FMinusSym(f), lambda))")
```

```
FPlus(1 + 1im, f) = -0.030615744795935648 - 0.011678556508991562im
FMinus(1 + 1im, f) = 0.1091262126064784 - 0.07682555657657669im
FPlusSym(1 + 1im, f) = -0.030615744795935648 - 0.011678556508991567im
FMinusSym(1 + 1im, f) = 0.10912621260647842 - 0.07682555657657673im
```

## 7.8 get_gamma(a, n, zeroList; infty, nGon)

Computes the contour $\Gamma$ given by

$$\Gamma := \Gamma_0 \cup \Gamma_a,$$

where

$$\Gamma_a^{\pm} := \partial(\{\lambda \in \mathbb{C}^{\pm} : \Re(a\lambda^n) > 0\} \setminus \bigcup_{\substack{\sigma\in\mathbb{C};\\ \Delta(\sigma)=0}} D(\sigma, 2\epsilon)) \quad (D \text{ for disk}),$$

$$\Gamma_a := \Gamma_a^+ \cup \Gamma_a^-,$$

$$\Gamma_0^+ := \bigcup_{\substack{\sigma\in\overline{\mathbb{C}^+};\\ \Delta(\sigma)=0}} C(\sigma, \epsilon) \quad (C \text{ for circle}),$$

$$\Gamma_0^- := \bigcup_{\substack{\sigma\in\mathbb{C}^-;\\ \Delta(\sigma)=0}} C(\sigma, \epsilon),$$

$$\Gamma_0 := \Gamma_0^+ \cup \Gamma_0^-.$$

### 7.8.1 get_gammaAAngles(a, n; symbolic)

Finds the angles (in $[-2\pi, 2\pi)$) representing the lines through the origin that mark the beginning and end of the sectors encircled by $\Gamma_a$, boundary of the domain $\{\lambda \in \mathbb{C} : \Re(a\lambda^n) > 0\}$.

Note that we choose the interval $[-2\pi, 2\pi)$ to ensure that "z is in a sector" if and only if "`angle(z)` is greater than or equal to the start of the sector and smaller than or equal to the end of the sector.





**Contents**
7.1 check_boundaryConditions(U, fSym)
7.2 get_MPlusMinus(adjointU; symbolic, generic)
7.3 get_M(adjointU; symbolic, generic)
7.4 get_delta(adjointU; symbolic, generic)
7.5 get_Xlj(adjointU, l, j; symbolic, generic)
7.6 get_ChebyshevIntegral(l, f; symbolic, lambda, alpha)
7.6.1 get_ChebyshevTermInte(symbolic, c)
7.6.1.1 get_alpha(m, n)
7.6.2 get_ChebyshevCoeffici
7.6.3 get_ChebyshevIntegral(f; symbolic, lambda, alpha)
7.7 get_FPlusMinus(adjointU; symbolic, generic)
7.8 get_gamma(a, n, zeroList; infty, nGon)
7.8.1 get_gammaAAngles(a, n; symbolic)
7.8.2 get_gammaAAnglesSplit(n; symbolic)
7.8.3 pointOnSector(z, sectorAngles)
7.8.4 pointInSector(z, sectorAngles)
7.8.5 pointExSector(z, sectorAngles)
7.8.6 get_epsilon(zeroList, a, n)
7.8.7 get_nGonAroundZero(ze(epsilon, n)
7.8.8 get_gamma(a, n, zeroList; infty, nGon)
7.9 plot_contour(gamma; infty)
7.10 solve_IBVP(L, U, a, zeroList, f; FPlusFunc, FMinusFunc, pDerivMatrix, infty)
8 Verifying the formulas of $F_\lam
8.1 Problem 1
8.2 Problem 2

In [352]:
```
function get_gammaAAngles(a::Number, n::Int; symbolic = false)
    # thetaA = argument(a)
    thetaA = angle(a)
    if symbolic
        thetaStartList = Array{SymPy.Sym}(n) # List of angles that characterize where domain sectors start
        thetaEndList = Array{SymPy.Sym}(n) # List of angles that characterize where domain sectors end
        k = symbols("k")
        counter = 0
        while (2pi*counter + pi/2 - thetaA)/n < 2pi
        # Substituting counter for k
        # while SymPy.N(subs((2PI*k + PI/2 - thetaA)/n, k, counter)) < 2pi
            thetaStart = (2*PI*counter - PI/2 - rationalize(thetaA/pi)*PI)/n
            thetaEnd = (2*PI*counter + PI/2 - rationalize(thetaA/pi)*PI)/n
            counter += 1
            thetaStartList[counter] = thetaStart
            thetaEndList[counter] = thetaEnd
        end
    else
        thetaStartList = Array{Number}(n)
        thetaEndList = Array{Number}(n)
        k = 0
        while (2pi*k + pi/2 - thetaA)/n < 2pi
            thetaStart = (2pi*k - pi/2 - thetaA)/n
            thetaEnd = (2pi*k + pi/2 - thetaA)/n
            k += 1
            thetaStartList[k] = thetaStart
            thetaEndList[k] = thetaEnd
        end
    end
    return (thetaStartList, thetaEndList)
end
```

Out[352]: get_gammaAAngles (generic function with 1 method)

**Parameters**

- a: Number
  - $a$ in $\Gamma_a$, given along with $S$, $f$, and $B$ as parameters of get_input().
- n: Int
  - $n$ in the definition of $\Gamma_a^{\pm}$.
- symbolic*: Bool
  - Boolean indicating whether the output is symbolic.

**Returns**

- get_gammaAAngles: Tuple{Array, Array}
  - Returns a tuple containing an array of angles in $[-2\pi, 2\pi)$ that characterize the starts of the sectors and an array of angles that characterize the ends of the sectors.

**Example**

In [353]:
```
n = 2
a = 1
gammaAAnglesSym = get_gammaAAngles(a, n; symbolic = true)
println("n=$n, a=$a, gammaAAnglesSym = $gammaAAnglesSym")

gammaAAngles = get_gammaAAngles(a, n; symbolic = false)
println("n=$n, a=$a, gammaAAngles = $gammaAAngles")

n = 3
a = im
gammaAAnglesSym = get_gammaAAngles(a, n; symbolic = true)
println("n=$n, a=$a, gammaAAnglesSym = $gammaAAnglesSym")

n = 3
a = -im
gammaAAnglesSym = get_gammaAAngles(a, n; symbolic = true)
println("n=$n, a=$a, gammaAAnglesSym = $gammaAAnglesSym")
```

```
n=2, a=1, gammaAAnglesSym = (SymPy.Sym[-pi/4, 3*pi/4], SymPy.Sym[pi/4, 5*pi/4])
n=2, a=1, gammaAAngles = (Number[-0.785398, 2.35619], Number[0.785398, 3.92699])
n=3, a=im, gammaAAnglesSym = (SymPy.Sym[-pi/3, pi/3, pi], SymPy.Sym[0, 2*pi/3, 4*pi/3])
n=3, a=0 - 1im, gammaAAnglesSym = (SymPy.Sym[0, 2*pi/3, 4*pi/3], SymPy.Sym[pi/3, pi, 5*pi/3])
```

### 7.8.2 get_gammaAAnglesSplit(a, n; symbolic)

On top of the sectors characterized by the array of start angles and the array of end angles obtained via get_gammaAAngles(), splits the sectors that contain the real line.







In [354]:
```
function get_gammaAAnglesSplit(a::Number, n::Int; symbolic = false)
    (thetaStartList, thetaEndList) = get_gammaAAngles(a, n; symbolic = symbolic)
    # Split sectors that contain the positive half of the real line (angle = 0)
    zeroIndex = find(i -> ((is_approxLess(thetaStartList[i], 0) && is_approxLess(0, thetaEndList[i]))), 1:length(thetaStartList))
    if !isempty(zeroIndex)
        index = zeroIndex[1]
        # Insert 0 after thetaStart
        splice!(thetaStartList, (index+1):index, 0)
        # Insert 0 before thetaEnd
        splice!(thetaEndList, index:(index-1), 0)
    end
    # Split sectors that contain the negative half of the real line (angle = pi)
    piIndex = find(i -> ((is_approxLess(thetaStartList[i], pi) && is_approxLess(pi, thetaEndList[i]))), 1:length(thetaStartList))
    if !isempty(piIndex)
        index = piIndex[1]
        if symbolic
            # Insert pi after thetaStart
            splice!(thetaStartList, (index+1):index, pi)
            # Insert pi before thetaEnd
            splice!(thetaEndList, index:(index-1), pi)
        else
            # Use pi*1 instead of pi to avoid "<= not defined for Irrational{:pi}" error in get_gamma()
            splice!(thetaStartList, (index+1):index, pi*1)
            splice!(thetaEndList, index:(index-1), pi*1)
        end
    end
    return (thetaStartList, thetaEndList)
end
```

Out[354]: get_gammaAAnglesSplit (generic function with 1 method)

**Parameters**

- a: Number
  - $a$ in $\Gamma_a$, given along with $S$, $f$, and $B$ as parameters of get_input().
- n: Int
  - $n$ in the definition of $\Gamma_a^{\pm}$.
- symbolic*: Bool
  - Boolean indicating whether the output is symbolic.

**Returns**

- get_gammaAAnglesSplit: Tuple{Array, Array}
  - Returns a tuple containing an array of angles in $[-2\pi, 2\pi)$ that characterize the starts of the sectors and an array of angles that characterize the ends of the sectors. Compared to the output of get_gammaAAngles(), the arrays of angles reflect the fact that sectors containing the real line are splitted into upper half and lower half.

**Example**

```
In [355]:
n = 2
a = 1
gammaAAnglesSplitSym = get_gammaAAnglesSplit(a, n; symbolic = true)
println("n=$n, a=$a, gammaAAnglesSplitSym = $gammaAAnglesSplitSym")
gammaAAnglesSplit = get_gammaAAnglesSplit(a, n; symbolic = false)
println("n=$n, a=$a, gammaAAnglesSplit = $gammaAAnglesSplit")

n = 3
a = im
gammaAAnglesSplitSym = get_gammaAAnglesSplit(a, n; symbolic = true)
println("n=$n, a=$a, gammaAAnglesSplitSym = $gammaAAnglesSplitSym")
gammaAAnglesSplit = get_gammaAAnglesSplit(a, n; symbolic = false)
println("n=$n, a=$a, gammaAAnglesSplit = $gammaAAnglesSplit")
```

```
n=2, a=1, gammaAAnglesSplitSym = (SymPy.Sym[-pi/4, 0, 3*pi/4, pi], SymPy.Sym[0, pi/4, pi, 5*pi/4])
n=2, a=1, gammaAAnglesSplit = (Number[-0.785398, 0, 2.35619, 3.14159], Number[0, 0.785398, 3.14159, 3.92699])
n=3, a=im, gammaAAnglesSplitSym = (SymPy.Sym[-pi/3, pi/3, pi], SymPy.Sym[0, 2*pi/3, 4*pi/3])
n=3, a=im, gammaAAnglesSplit = (Number[-1.0472, 1.0472, 3.14159], Number[0.0, 2.0944, 4.18879])
```

### 7.8.3 pointOnSector(z, sectorAngles)

Determines whether a point $z$ is on the boundary of a sector characterized by a start angle and an end angle.

```
In [356]:
function pointOnSector(z::Number, sectorAngles::Tuple{Number, Number})
    (startAngle, endAngle) = sectorAngles
    return is_approx(argument(z), startAngle) || is_approx(argument(z), endAngle) || is_approx(angle(z), startAngle) || is_approx(angle(z), endAngle)
end
```

```
Out[356]: pointOnSector (generic function with 1 method)
```

**Parameters**

- z: Number
  - Point in the complex plane.
- sectorAngles: Tuple{Number, Number}
  - Pair of start and end angles characterizing a sector.

**Returns**

- pointOnSector: Bool
  - Returns true if z is on the boundary of the sector given by sectorAngles and false otherwise.

**Example**







```
In [357]:
z = 1+im
sectorAngles = (-pi/4, pi/4)
pointOnSector(z, sectorAngles)
```

```
Out[357]: true
```

### 7.8.4 pointInSector(z, sectorAngles)

Determines whether a point $z$ is in the interior of a sector characterized by a start angle and an end angle.

```
In [358]:
function pointInSector(z::Number, sectorAngles::Tuple{Number, Number})
    (startAngle, endAngle) = sectorAngles
    # First check if z is on the sector boundary
    if pointOnSector(z, sectorAngles)
        return false
    else
        # angle(z) would work if it's in the sector with positive real parts and both positive and negative imaginary parts; argument(z) would work if it's in the sector with negative real parts and both positive and negative imaginary parts
        return (angle(z) > startAngle && angle(z) < endAngle) || (argument(z) > startAngle && argument(z) < endAngle) # no need to use is_approxLess because the case of approximatedly equal is already checked in pointOnSector
    end
end
```

```
Out[358]: pointInSector (generic function with 1 method)
```

**Parameters**

- z: Number
  - Point in the complex plane.
- sectorAngles: Tuple{Number, Number}
  - Pair of start and end angles characterizing a sector.

**Returns**

- pointInSector: Bool
  - Returns true if z is interior to the sector given by sectorAngles and false otherwise.

**Example**

In [359]:
```
z = 1+im
sectorAngles = (-pi/4, pi/4)
println("pointInSector($z, $sectorAngles) = $(pointInSector(z, sectorAngles))")

z = 1
println("pointInSector($z, $sectorAngles) = $(pointInSector(z, sectorAngles))")

z = -1
println("pointInSector($z, $sectorAngles) = $(pointInSector(z, sectorAngles))")
```

```
pointInSector(1 + 1im, (-0.7853981633974483, 0.7853981633974483)) = false
pointInSector(1, (-0.7853981633974483, 0.7853981633974483)) = true
pointInSector(-1, (-0.7853981633974483, 0.7853981633974483)) = false
```

### 7.8.5 pointExSector(z, sectorAngles)

Determines whether a point $z$ is in the exterior of a sector characterized by a start angle and an end angle.

In [360]:
```
function pointExSector(z::Number, sectorAngles::Tuple{Number, Number})
    return !pointOnSector(z, sectorAngles) && !pointInSector(z, sectorAngles)
end
```

Out[360]: pointExSector (generic function with 1 method)

**Parameters**

- z: Number
  - Point in the complex plane.
- sectorAngles: Tuple{Number, Number}
  - Pair of start and end angles characterizing a sector.

**Returns**

- pointExSector: Bool
  - Returns true if z is exterior to the sector given by sectorAngles and false otherwise.

**Example**







In [361]:
```
z = 1+im
sectorAngles = (-pi/4, pi/4)
println("pointExSector($z, $sectorAngles) = $(pointExSector(z, sectorAngles))")

z = 1
println("pointExSector($z, $sectorAngles) = $(pointExSector(z, sectorAngles))")

z = -1
println("pointExSector($z, $sectorAngles) = $(pointExSector(z, sectorAngles))")
```

```
pointExSector(1 + 1im, (-0.7853981633974483, 0.7853981633974483)) = false
pointExSector(1, (-0.7853981633974483, 0.7853981633974483)) = false
pointExSector(-1, (-0.7853981633974483, 0.7853981633974483)) = true
```

### 7.8.6 get_epsilon(zeroList, a, n)

Given a list of zeroes of $\Delta(\lambda)$, using the arrays of angles that characterize the starts and ends of sectors, computes an appropriate value for the radius $\epsilon$ of the circle to be drawn around each zero of $\Delta(\lambda)$ in the contours $\Gamma_0^{\pm}$. This is the minimum of pairwise distances between zeroes that are not interior to any sector (since interior zeroes would not matter in any way) and the distances from any of these zeroes to any line that mark the boundary of some sector.

In [362]:
```
function get_epsilon(zeroList::Array, a::Number, n::Int)
    (thetaStartList, thetaEndList) = get_gammaAAnglesSplit(a, n; symbolic = false)
    thetaStartEndList = collect(Iterators.flatten([thetaStartList, thetaEndList]))
    truncZeroList = []
    for zero in zeroList
        # If zero is interior to any sector, discard it
        if any(i -> pointInSector(zero, (thetaStartList[i], thetaEndList[i])), 1:n)
        else # If not, append it to truncZeroList
            append!(truncZeroList, zero)
        end
    end
    # List of distance between each zero and each line marking the boundary of some sector
    pointLineDistances = [get_distancePointLine(z, theta) for z in zeroList for theta in thetaStartEndList]
    if length(truncZeroList)>1
        # List of distance between every two zeroes
        pairwiseDistances = [norm(z1-z2) for z1 in zeroList for z2 in truncZeroList]
    else
        pairwiseDistances = []
    end
    distances = collect(Iterators.flatten([pairwiseDistances, pointLineDistances]))
    # Distances of nearly 0 could be instances where the zero is actually on some sector boundary
    distances = filter(x -> !is_approx(x, 0), distances)
    epsilon = minimum(distances)/4
    return epsilon
end
```

Out[362]: get_epsilon (generic function with 1 method)

**Parameters**

- zeroList: Array of Number
  - List of zeroes of $\Delta(\lambda)$. Found by human input with the aid of `plot_levelCurves()`.
- a: Number
  - $a$ in $\Gamma_a$, given along with $S$, $f$, and $B$ as parameters of `get_input()`.
- n: Int
  - $n$ in the definition of $\Gamma_{\vec{a}}^{\pm}$.

**Returns**

- get_epsilon: Number
  - Returns one-fourth the minimum of the distances between any two exterior zeroes of $\Delta(\lambda)$ and the distances between any exterior zero to any line marking the boundary of some sector.

**Example**

In [363]:
```
n = 2
a = 1
zeroList = [1+sqrt(3)*im, 2+2*sqrt(3)*im, 0+0*im, 0+5*im, 0-5*im]
get_epsilon(zeroList, a, n)
```

Out[363]: 0.1294095225512604

### 7.8.7 get_nGonAroundZero(zero, epsilon, n)

Given a zero of $\Delta(\lambda)$, returns an array of $n$ complex numbers representing the vertices of an $n$-gon around the zero in the complex plane; each vertex is of distance $\epsilon$ from the zero.

In [364]:
```
function get_nGonAroundZero(zero::Number, epsilon::Number, n::Int)
    z = zero
    theta = argument(zero)
    deltaAngle = 2pi/n
    vertices = []
    for i = 1:n
        newAngle = pi-deltaAngle*(i-1)
        vertex = z + epsilon*e^(im*(theta+newAngle))
        append!(vertices, vertex)
    end
    # vertices = vcat(vertices, vertices[1])
    return vertices
end
```

Out[364]: get_nGonAroundZero (generic function with 1 method)







**Parameters**

- zero: Number
  - A root of $\Delta(\lambda)$.
- epsilon: Number
  - Output of `get_epsilon()`. Distance between each vertex of the $n$-gon from `zero`.
- n: Int
  - $n$ in $n$-gon.

**Returns**

- get_nGonAroundZero: Array of Number
  - Returns an array of $n$ points that are the vertices of the $n$-gon in the complex plane.

**Example**

In [365]:
```
zero = im
epsilon = 1
n = 4
prettyPrint.(get_nGonAroundZero(zero, epsilon, n))
```

Out[365]:
$$\begin{bmatrix} 0 \\ -1+i \\ 2i \\ 1+i \end{bmatrix}$$

### 7.8.8 get_gamma(a, n, zeroList; infty, nGon)

Finds the contours $\Gamma_a^+$, $\Gamma_a^-$, $\Gamma_0^+$, $\Gamma_0^-$ as arrays of points ordered in the order they will be integrated over.

In [366]:
```
function get_gamma(a::Number, n::Int, zeroList::Array; infty = INFTY, nGon = 8)
    (thetaStartList, thetaEndList) = get_gammaAAnglesSplit(a, n; symbolic = false)
    nSplit = length(thetaStartList)
    gammaAPlus, gammaAMinus, gamma0Plus, gamma0Minus = [], [], [], []
    epsilon = get_epsilon(zeroList, a, n)
    for i in 1:nSplit
        thetaStart = thetaStartList[i]
        thetaEnd = thetaEndList[i]
        # Initialize the boundary of each sector with the ending boundary, the origin, and the starting boun
dary (start and end boundaries refer to the order in which the boundaries are passed if tracked counterclock
wise)
        initialPath = [infty*e^(im*thetaEnd), 0+0*im, infty*e^(im*thetaStart)]
        initialPath = convert(Array{Complex{Float64}}, initialPath)
        if thetaStart >= 0 && thetaStart <= pi && thetaEnd >= 0 && thetaEnd <= pi # if in the upper half pla
ne, push the boundary path to gamma_a+
            push!(gammaAPlus, initialPath) # list of lists
        else # if in the lower half plane, push the boundary path to gamma_a-
```







```
            push!(gammaAMinus, initialPath)
        end
    end
    # Sort the zeroList by norm, so that possible zero at the origin comes last. We need to leave the origin
in the initial path unchanged until we have finished dealing with all non-origin zeros because we use the or
igin in the initial path as a reference point to decide where to insert the deformed path
    zeroList = sort(zeroList, lt=(x,y)->!isless(norm(x), norm(y)))
    for zero in zeroList
        # println(zero)
        # If zero is not at the origin
        if !is_approx(zero, 0+0*im)
            # Draw an n-gon around it
            vertices = get_nGonAroundZero(zero, epsilon, nGon)
            # If zero is on the boundary of some sector
            if any(i -> pointOnSector(zero, (thetaStartList[i], thetaEndList[i])), 1:nSplit)
                # Find which sector(s) zero is on
                indices = find(i -> pointOnSector(zero, (thetaStartList[i], thetaEndList[i])), 1:nSplit)
                # If zero is on the boundary of one sector
                if length(indices) == 1
                    # if vertices[2] is interior to any sector, include vertices on the other half of the n-
gon in the contour approximation
                    z0 = vertices[2]
                    if any(i -> pointInSector(z0, (thetaStartList[i], thetaEndList[i])), 1:nSplit)
                        # Find which sector vertices[2] is in
                        index = find(i -> pointInSector(z0, (thetaStartList[i], thetaEndList[i])), 1:nSplit)
[1]
                    else # if vertices[2] is exterior, include vertices on this half of the n-gon in the con
tour approximation
                        # Find which sector vertices[length(vertices)] is in
                        z1 = vertices[length(vertices)]
                        index = find(i -> pointInSector(z1, (thetaStartList[i], thetaEndList[i])), 1:nSplit)
[1]
                    end
                    thetaStart = thetaStartList[index]
                    thetaEnd = thetaEndList[index]
                    # Find all vertices exterior to or on the boundary of this sector, which would form the
 nGonPath around the zero
                    nGonPath = vertices[find(vertex -> !pointInSector(vertex, (thetaStart, thetaEnd)), verti
ces)]
                    # If this sector is in the upper half plane, deform gamma_a+
                    if thetaStart >= 0 && thetaStart <= pi && thetaEnd >= 0 && thetaEnd <= pi
                        gammaAPlusIndex = find(path -> (is_approx(argument(zero), argument(path[1])) || is_a
pprox(argument(zero), argument(path[length(path)]))), gammaAPlus)[1]
                        deformedPath = copy(gammaAPlus[gammaAPlusIndex])
                        if any(i -> is_approx(argument(zero), thetaStartList[i]) || is_approx(angle(zero), t
hetaStartList[i]), 1:nSplit) # if zero is on the starting boundary, insert the n-gon path after 0+0*im
                            splice!(deformedPath, length(deformedPath):(length(deformedPath)-1), nGonPath)
                        else # if zero is on the ending boundary, insert the n-gon path before 0+0*im
                            splice!(deformedPath, 2:1, nGonPath)
                        end
                        deformedPath = convert(Array{Complex{Float64}}, deformedPath)
                        gammaAPlus[gammaAPlusIndex] = deformedPath
                    else # if sector is in the lower half plane, deform gamma_a-
                        # # Find all vertices interior to or on the boundary of this sector, which would for
```

```
m the nGonPath around the zero
                        # nGonPath = vertices[find(vertex -> !pointExSector(vertex, (thetaStart, thetaEnd)), vertices)]
                        gammaAMinusIndex = find(path -> (is_approx(argument(zero), argument(path[1])) || is_approx(argument(zero), argument(path[length(path)]))), gammaAMinus)[1]
                        deformedPath = copy(gammaAMinus[gammaAMinusIndex])
                        if any(i -> is_approx(argument(zero), thetaStartList[i]) || is_approx(angle(zero), thetaStartList[i]), 1:nSplit) # if zero is on the starting boundary, insert the n-gon path after 0+0*im
                            splice!(deformedPath, length(deformedPath):(length(deformedPath)-1), nGonPath)
                        else # if zero is on the ending boundary, insert the n-gon path before 0+0*im
                            splice!(deformedPath, 2:1, nGonPath)
                        end
                        deformedPath = convert(Array{Complex{Float64}}, deformedPath)
                        gammaAMinus[gammaAMinusIndex] = deformedPath
                    end
                else # If zero is on the boundary of two sectors, then it must be on the real line, and we need to deform two sectors
                    # Find out which vertices are in the lower half plane
                    nGonPath = vertices[find(vertex -> !pointInSector(vertex, (0, pi)), vertices)]
                    for index in indices
                        thetaStart = thetaStartList[index]
                        thetaEnd = thetaEndList[index]
                        # If this is the sector in the upper half plane, deform gamma_a+
                        if thetaStart >= 0 && thetaStart <= pi && thetaEnd >= 0 && thetaEnd <= pi
                            gammaAPlusIndex = find(path -> (is_approx(argument(zero), argument(path[1])) || is_approx(argument(zero), argument(path[length(path)]))), gammaAPlus)[1]
                            deformedPath = copy(gammaAPlus[gammaAPlusIndex])
                            if is_approx(argument(zero), argument(deformedPath[length(deformedPath)])) # if zero is on the starting boundary, insert the n-gon path after 0+0*im
                                splice!(deformedPath, length(deformedPath):(length(deformedPath)-1), nGonPath)
                            else # if zero is on the ending boundary, insert the n-gon path before 0+0*im
                                splice!(deformedPath, 2:1, nGonPath)
                            end
                            deformedPath = convert(Array{Complex{Float64}}, deformedPath)
                            gammaAPlus[gammaAPlusIndex] = deformedPath
                        else # If this is the sector in the lower half plane, deform gamma_a-
                            gammaAMinusIndex = find(path -> (is_approx(argument(zero), argument(path[1])) || is_approx(argument(zero), argument(path[length(path)]))), gammaAMinus)[1]
                            deformedPath = copy(gammaAMinus[gammaAMinusIndex])
                            if is_approx(argument(zero), argument(deformedPath[length(deformedPath)])) # if zero is on the starting boundary, insert the n-gon path after 0+0*im
                                splice!(deformedPath, length(deformedPath):(length(deformedPath)-1), nGonPath)
                            else # if zero is on the ending boundary, insert the n-gon path before 0+0*im
                                splice!(deformedPath, 2:1, nGonPath)
                            end
                            deformedPath = convert(Array{Complex{Float64}}, deformedPath)
                            gammaAMinus[gammaAMinusIndex] = deformedPath
                        end
                    end
                end
                # Sort each sector's path in the order in which they are integrated over
                gammaAs = [gammaAPlus, gammaAMinus]
```







```
                for j = 1:length(gammaAs)
                    gammaA = gammaAs[j]
                    for k = 1:length(gammaA)
                        inOutPath = gammaA[k]
                        originIndex = find(x->x==0+0*im, inOutPath)[1]
                        inwardPath = inOutPath[1:(originIndex-1)]
                        outwardPath = inOutPath[(originIndex+1):length(inOutPath)]
                        # Sort the inward path and outward path
                        if length(inwardPath) > 0
                            inwardPath = sort(inwardPath, lt=(x,y)->!isless(norm(x), norm(y)))
                        end
                        if length(outwardPath) > 0
                            outwardPath = sort(outwardPath, lt=(x,y)->isless(norm(x), norm(y)))
                        end
                        inOutPath = vcat(inwardPath, 0+0*im, outwardPath)
                        inOutPath = convert(Array{Complex{Float64}}, inOutPath)
                        gammaA[k] = inOutPath
                    end
                    gammaAs[j] = gammaA
                end
                gammaAPlus, gammaAMinus = gammaAs[1], gammaAs[2]
            # If zero is interior to any sector (after splitting by real line), ignore it
            # If zero is exterior to the sectors, avoid it
            elseif all(i -> pointExSector(zero, (thetaStartList[i], thetaEndList[i])), 1:nSplit)
                nGonPath = vcat(vertices, vertices[1]) # counterclockwise
                nGonPath = convert(Array{Complex{Float64}}, nGonPath)
                # If zero is in the upper half plane, add the n-gon path to gamma_0+
                if argument(zero) >= 0 && argument(zero) <= pi
                    push!(gamma0Plus, nGonPath)
                else # If zero is in the lower half plane, add the n-gon path to gamma_0-
                    push!(gamma0Minus, nGonPath)
                end
            end
        else # If zero is at the origin, we deform all sectors and draw an n-gon around the origin
            # deform each sector in gamma_a+
            for i = 1:length(gammaAPlus)
                deformedPath = gammaAPlus[i]
                # find the index of the zero at origin in the sector boundary path
                index = find(j -> is_approx(deformedPath[j], 0+0*im), 1:length(deformedPath))
                # If the origin is not in the path, then it has already been bypassed
                if isempty(index)
                else # If not, find its index
                    index = index[1]
                end
                # create a path around zero (origin); the origin will not be the first or the last point in any sector boundary because it was initialized to be in the middle, and only insertions are performed. Moreover, the boundary path has already been sorted into the order in which they will be integrated over, so squarePath defined below has deformedPath[index-1], deformedPath[index+1] in the correct order.
                squarePath = [2*epsilon*e^(im*argument(deformedPath[index-1])), 2*epsilon*e^(im*argument(deformedPath[index+1]))]
                # replace the zero with the deformed path
                deleteat!(deformedPath, index) # delete the origin
                splice!(deformedPath, index:(index-1), squarePath) # insert squarePath into where the origin was at
```

```
                deformedPath = convert(Array{Complex{Float64}}, deformedPath)
                gammaAPlus[i] = deformedPath
            end
            # deform each sector in gamma_a-
            for i = 1:length(gammaAMinus)
                deformedPath = gammaAMinus[i]
                index = find(j -> is_approx(deformedPath[j], 0+0*im), 1:length(deformedPath))
                if isempty(index)
                else
                    index = index[1]
                end
                squarePath = [2*epsilon*e^(im*argument(deformedPath[index-1])), 2*epsilon*e^(im*argument(def
ormedPath[index+1]))]
                deleteat!(deformedPath, index)
                splice!(deformedPath, index:(index-1), squarePath)
                deformedPath = convert(Array{Complex{Float64}}, deformedPath)
                gammaAMinus[i] = deformedPath
            end
            # Draw an n-gon around the origin and add to gamma_0+
            vertices = get_nGonAroundZero(zero, epsilon, nGon)
            nGonPath = vcat(vertices, vertices[1])
            nGonPath = convert(Array{Complex{Float64}}, nGonPath)
            push!(gamma0Plus, nGonPath)
        end
    end
    return (gammaAPlus, gammaAMinus, gamma0Plus, gamma0Minus)
end
```

Out[366]: get_gamma (generic function with 1 method)

**Parameters**

- a: Number
  - $a$ in $\Gamma_a$, given along with $S$, $f$, and $B$ as parameters of get_input().
- n: Int
  - $n$ in the definition of $\Gamma_a^{\pm}$.
- zeroList: Array of Number
  - List of zeroes of $\Delta(\lambda)$. Found by human input with the aid of plot_levelCurves().
- infty*: Number
  - Range of search when finding points in $\Gamma$. Default to the global variables INFTY.
- nGon*: Int
  - $n$ in $n$-gon.

**Returns**

- get_gamma: Tuple{Array, Array, Array, Array}
  - Returns the contours $\Gamma_a^+, \Gamma_a^-, \Gamma_0^+$, and $\Gamma_0^-$ as arrays of numbers in the complex plane. Note that each of these contour is an array of arrays, reflecting the fact that it is a union of individual paths.

**Example**







In [367]:
```
n = 2
a = 1
zeroList = [3+3*sqrt(3)*im, 2+2*sqrt(3)*im, 0+0*im, 0+5*im, 0-5*im]
(gammaAPlus, gammaAMinus, gamma0Plus, gamma0Minus) = get_gamma(a, n, zeroList)

println("gammaAPlus = $gammaAPlus")
println("gammaAMinus = $gammaAMinus")
println("gamma0Plus = $gamma0Plus")
println("gamma0Minus = $gamma0Minus")
```

```
gammaAPlus = Any[Complex{Float64}[7.07107+7.07107im, 0.366025+0.366025im, 0.517638+0.0im, 10.0+0.0im], Comp
lex{Float64}[-10.0+1.22465e-15im, -0.517638+6.33924e-17im, -0.366025+0.366025im, -7.07107+7.07107im]]
gammaAMinus = Any[Complex{Float64}[10.0+0.0im, 0.517638+0.0im, 0.366025-0.366025im, 7.07107-7.07107im], Com
plex{Float64}[-7.07107-7.07107im, -0.366025-0.366025im, -0.517638+6.33924e-17im, -10.0+1.22465e-15im]]
gamma0Plus = Any[Complex{Float64}[2.87059+4.97201im, 2.75+5.12917im, 2.77586+5.32556im, 2.93301+5.44615im,
3.12941+5.4203im, 3.25+5.26314im, 3.22414+5.06674im, 3.06699+4.94615im, 2.87059+4.97201im], Complex{Float6
4}[-4.75443e-17+4.74118im, -0.183013+4.81699im, -0.258819+5.0im, -0.183013+5.18301im, 1.58481e-17+5.25882i
m, 0.183013+5.18301im, 0.258819+5.0im, 0.183013+4.81699im, -4.75443e-17+4.74118im], Complex{Float64}[1.8705
9+3.23996im, 1.75+3.39711im, 1.77586+3.59351im, 1.93301+3.7141im, 2.12941+3.68825im, 2.25+3.53109im, 2.2241
4+3.33469im, 2.06699+3.2141im, 1.87059+3.23996im], Complex{Float64}[-0.258819+3.16962e-17im, -0.183013+0.18
3013im, 1.58481e-17+0.258819im, 0.183013+0.183013im, 0.258819+0.0im, 0.183013-0.183013im, 1.58481e-17-0.258
819im, -0.183013-0.183013im, -0.258819+3.16962e-17im]]
gamma0Minus = Any[Complex{Float64}[7.92405e-17-4.74118im, 0.183013-4.81699im, 0.258819-5.0im, 0.183013-5.18
301im, -4.75443e-17-5.25882im, -0.183013-5.18301im, -0.258819-5.0im, -0.183013-4.81699im, 7.92405e-17-4.741
18im]]
```

## 7.9 plot_contour(gamma; infty)

Visualize the contours $\Gamma_a^+, \Gamma_a^-, \Gamma_0^+, \Gamma_0^-$.

In [368]:
```
function plot_contour(gamma::Array; infty = INFTY)
    sectorPathList = Array{Any}(length(gamma),1)
    for i = 1:length(gamma)
        # For each sector path in the gamma contour, plot the points in the path and connect them in the ord
er in which they appear in the path
        sectorPath = gamma[i]
        # labels = map(string, collect(1:1:length(sectorPath)))
        sectorPathList[i] = layer(x = real(sectorPath), y = imag(sectorPath), Geom.line(preserve_order=true
))
    end
    coord = Coord.cartesian(xmin=-infty, xmax=infty, ymin=-infty, ymax=infty, fixed=true)
    Gadfly.plot(Guide.xlabel("Re"), Guide.ylabel("Im"), coord, sectorPathList...)
end
```

Out[368]: plot_contour (generic function with 1 method)

**Parameters**

- gamma: Array
  - Contour obtained from get_gamma().
- infty*: Number
  - Range of search when finding points in $\Gamma$. Default to the global variable INFTY.

**Returns**

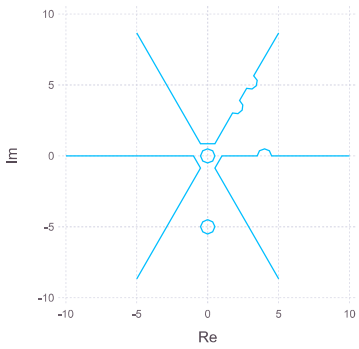
- plot_contour: None
  - Plots the contour $\Gamma$ in the complex plane.

**Example**

In [369]:
```
n = 3
a = im
zeroList = [3+3*sqrt(3)*im, 2+2*sqrt(3)*im, 0+0*im, 0+5*im, 0-5*im, 4]
(gammaAPlus, gammaAMinus, gamma0Plus, gamma0Minus) = get_gamma(a, n, zeroList)
gamma = collect(Iterators.flatten([gammaAPlus, gammaAMinus, gamma0Plus, gamma0Minus]))

plot_contour(gamma)
```

Out[369]:





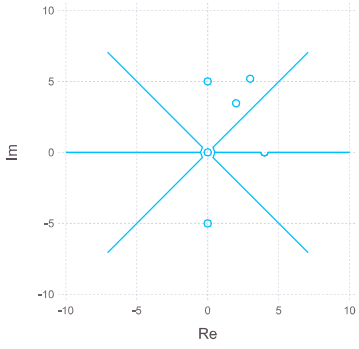





In [370]:
```
n = 2
a = 1
(gammaAPlus, gammaAMinus, gamma0Plus, gamma0Minus) = get_gamma(a, n, zeroList)
gamma = collect(Iterators.flatten([gammaAPlus, gammaAMinus, gamma0Plus, gamma0Minus]))

plot_contour(gamma)
```

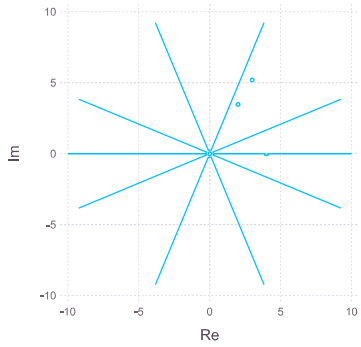
Out[370]:



In [371]:
```
n = 4
a = 1
(gammaAPlus, gammaAMinus, gamma0Plus, gamma0Minus) = get_gamma(a, n, zeroList)
gamma = collect(Iterators.flatten([gammaAPlus, gammaAMinus, gamma0Plus, gamma0Minus]))

plot_contour(gamma)
```

Out[371]:




## 7.10 solve_IBVP(L, U, a, zeroList, f; FPlusFunc, FMinusFunc, pDerivMatrix, infty)

Computes the solution

$$\begin{aligned} q(x,t) &= f_x\left(e^{-a\lambda^n t}F_\lambda(f)\right) \\ &= \int_{\Gamma_0^+} e^{i\lambda x}e^{-a\lambda^n t}F_\lambda^+(f)\,d\lambda + \int_{\Gamma_a^+} e^{i\lambda x}e^{-a\lambda^n t}F_\lambda^+(f)\,d\lambda + \int_{\Gamma_0^-} e^{i\lambda x}e^{-a\lambda^n t}F_\lambda^-(f)\,d\lambda + \int_{\Gamma_a^-} e^{i\lambda x}e^{-a\lambda^n t}F_\lambda^-(f)\,d\lambda \\ &= \int_{\Gamma_0^+} + \int_{\Gamma_a^+} e^{ix\lambda - at\lambda^n}F_\lambda^+(f)\,d\lambda + \int_{\Gamma_0^-} + \int_{\Gamma_a^-} e^{ix\lambda - at\lambda^n}F_\lambda^-(f)\,d\lambda. \end{aligned}$$





**Contents**
7.1 check_boundaryConditions(U, fSym)
7.2 get_MPlusMinus(adjointU; symbolic, generic)
7.3 get_M(adjointU; symbolic, generic)
7.4 get_delta(adjointU; symbolic, generic)
7.5 get_Xlj(adjointU, l, j; symbolic, generic)
7.6 get_ChebyshevIntegral(l, f; symbolic, lambda, alpha)
7.6.1 get_ChebyshevTermInte
symbolic, c)
7.6.1.1 get_alpha(m, n)
7.6.2 get_ChebyshevCoeffici
7.6.3 get_ChebyshevIntegral
f; symbolic, lambda, alpha)
7.7 get_FPlusMinus(adjointU; symbolic, generic)
7.8 get_gamma(a, n, zeroList; infty, nGon)
7.8.1 get_gammaAAngles(a, n; symbolic)
7.8.2 get_gammaAAnglesSplit
n; symbolic)
7.8.3 pointOnSector(z, sectorAngles)
7.8.4 pointInSector(z, sectorAngles)
7.8.5 pointExSector(z, sectorAngles)
7.8.6 get_epsilon(zeroList, a, n)
7.8.7 get_nGonAroundZero(ze
epsilon, n)
7.8.8 get_gamma(a, n, zeroList; infty, nGon)
7.9 plot_contour(gamma; infty)
7.10 solve_IBVP(L, U, a, zeroList, f; FPlusFunc, FMinusFunc, pDerivMatrix, infty)
8 Verifying the formulas of $F_\lam
8.1 Problem 1
8.2 Problem 2

In [372]:
```
function solve_IBVP(L::LinearDifferentialOperator, U::VectorBoundaryForm, a::Number, zeroList::Array, f::Fun
ction; FPlusFunc = lambda->get_FPlusMinus(adjointU; symbolic = false)[1](lambda, f), FMinusFunc = lambda->ge
```

```
t_FPlusMinus(adjointU; symbolic = false)[2](lambda, f), pDerivMatrix = get_pDerivMatrix(L), infty = INFTY)
    n = length(L.pFunctions)-1
    adjointU = get_adjointU(L, U, pDerivMatrix)
    (gammaAPlus, gammaAMinus, gamma0Plus, gamma0Minus) = get_gamma(a, n, zeroList)
    function q(x,t)
        integrandPlus(lambda) = e^(im*lambda*x)*e^(-a*lambda^n*t) * FPlusFunc(lambda)
        integrandMinus(lambda) = e^(im*lambda*x)*e^(-a*lambda^n*t) * FMinusFunc(lambda)
        tic()
        println("integrandPlus = $(integrandPlus(1+im))")
        println("integrandMinus = $(integrandMinus(1+im))")
        toc()
        # Integrate over individual paths in the Gamma contours
        println("gamma0Plus = $gamma0Plus")
        tic()
        integralGamma0Plus = 0
        for path in gamma0Plus
            println("path = $path")
            if length(path) == 0
                path = [im,im]
            end
            tic()
            integralGamma0Plus += quadgk(integrandPlus, path...)[1]
            toc()
        end
        toc()
        println("int_0_+ = $integralGamma0Plus")

        println("gammaAPlus = $gammaAPlus")
        tic()
        integralGammaAPlus = 0
        for path in gammaAPlus
            println("path = $path")
            if length(path) == 0
                path = [im,im]
            end
            tic()
            integralGammaAPlus += quadgk(integrandPlus, path...)[1]
            toc()
        end
        toc()
        println("int_a_+ = $integralGammaAPlus")

        println("gamma0Minus = $gamma0Minus")
        tic()
        integralGamma0Minus = 0
        for path in gamma0Minus
            println("path = $path")
            if length(path) == 0
                path = [-im,-im]
            end
            tic()
            integralGamma0Minus += quadgk(integrandMinus, path...)[1]
            toc()
        end
```







```
        toc()
        println("int_0_- = $integralGamma0Minus")

        println("gammaAMinus = $gammaAMinus")
        tic()
        integralGammaAMinus = 0
        for path in gammaAMinus
            println("path = $path")
            if length(path) == 0
                path = [-im,-im]
            end
            tic()
            integralGammaAMinus += quadgk(integrandMinus, path...)[1]
            toc()
        end
        toc()
        println("int_a_- = $integralGammaAMinus")
        return (integralGamma0Plus + integralGammaAPlus + integralGamma0Minus + integralGammaAMinus)
    end
    return q
end
```

```
Out[372]: solve_IBVP (generic function with 1 method)
```

**Parameters**

- `L`: `LinearDifferentialOperator`
  - Linear differential operator in the differential equation of the IBVP.
- `U`: `VectorBoundaryForm`
  - Vector boundary form encoding the boundary conditions of the IBVP.
- `a`: `Number`
  - Coefficient of `L` in the differential equation.
- `zeroList`: `Array`
  - List of (approximate) zeroes of $\Delta(\lambda)$.
- `f`: `Function`
  - Initial condition of the IBVP. It also satisfies the boundary conditions.
- `FPlusFunc`, `FMinusFunc`: `Function`
  - $F^+, F^-$. Output of `get_FPlusMinus()`. Default to `lambda-> get_FPlusMinusLambda2(adjointU; symbolic = false)[1](lambda, f)`. For better performance, the user may usee the symbolic expressions of $F^+, F^-$ to directly define `FPlusFunc`, `FMinusFunc` as functions.
- `pDerivMatrix*`: `Array`
  - Matrix whose $(i+1)(j+1)$-entry is the $j$th derivative of $p_i$ (`L.pFunctions[i]`) implemented as a `Function`, `Number`, or `SymPy.Sym`. Default to the output of `get_pDerivMatrix(L)`.
- `infty*`: `Number`
  - Range of search when finding points in $\Gamma$. Default to the global variable `INFTY`.

**Returns**

- `solve_IBVP`: `Function`
  - Returns a function `q(x,t)` that solves the IBVP.

**Example**

Recall that the IBVP is posed as follows.

Define $n$ linearly independent boundary forms $\{B_j : C \to \mathbb{C} \mid j \in \{1,2,\dots,n\}\}$

$$B_j\phi = \sum_{k=0}^{n-1} \left( b_{jk}\phi^{(k)}(0) + \beta_{jk}\phi^{(k)}(1) \right), \quad j \in \{1,2,\dots,n\}.$$

Define

$$\Phi = \{\phi \in C : B_j\phi = 0 \; \forall j \in \{1,2,\dots,n\}\}.$$

Let $S : \Phi \to C$ be the linear differential operator

$$S\phi(x) = (-i)^n \phi^{(n)}(x).$$

Let $a \in \mathbb{C}$ be a constant.

Consider the IBVP

$$\begin{aligned} (\partial_t + aS)q(x,t) &= 0, \quad (x,t) \in (0,1) \times (0,T) \\ q(x,0) &= f(x) \in \Phi, \qquad x \in [0,1] \\ q(\cdot,t) &\in \Phi, \qquad t \in [0,T], \end{aligned}$$







**Case 1.1**

Suppose $n = 2$. Then

$$S\phi(x) = (-i)^2\phi^{(2)}(x) = -\phi^{(2)}.$$

Suppose $a = e^{i\theta}$ for $\theta \in [-\frac{\pi}{2}, \frac{\pi}{2}]$.

For $\beta_0, \beta_1 \in \hat{\mathbb{C}}$ ($\mathbb{C}$ including $0$ and $\infty$), consider the following boundary conditions $\Phi$:

$$\begin{aligned} \varphi(0) + \beta_0\varphi(1) &= 0 \\ \varphi'(0) + \beta_1\varphi'(1) &= 0. \end{aligned}$$

We note that in complete form,

- $S$ is given by

  $$S\phi = p_0\phi^{(2)} + p_1\phi^{(1)} + p_2\phi^{(0)}$$

  where $p_0 = -1, p_1 = p_2 = 0$.
- $\Phi$ is given by

  $$\begin{aligned} 1 \cdot \varphi(0) + \beta_0 \cdot \varphi(1) + 0 \cdot \varphi^{(1)}(0) + 0 \cdot \varphi^{(1)}(1) &= 0 \\ 0 \cdot \varphi(0) + 0 \cdot \varphi(1) + 1 \cdot \varphi^{(1)}(0) + \beta_1 \cdot \varphi^{(1)}(1) &= 0. \end{aligned}$$

  Thus,

  $$b = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \beta = \begin{bmatrix} \beta_0 & 0 \\ 0 & \beta_1 \end{bmatrix}.$$

Thus, $f(x) \in \Phi$ needs to satisfy

$$Uf = \begin{bmatrix} 1 & 0 & \beta_0 & 0 \\ 0 & 1 & 0 & \beta_1 \end{bmatrix} \begin{bmatrix} f(0) \\ f'(0) \\ f(1) \\ f'(1) \end{bmatrix} = 0.$$

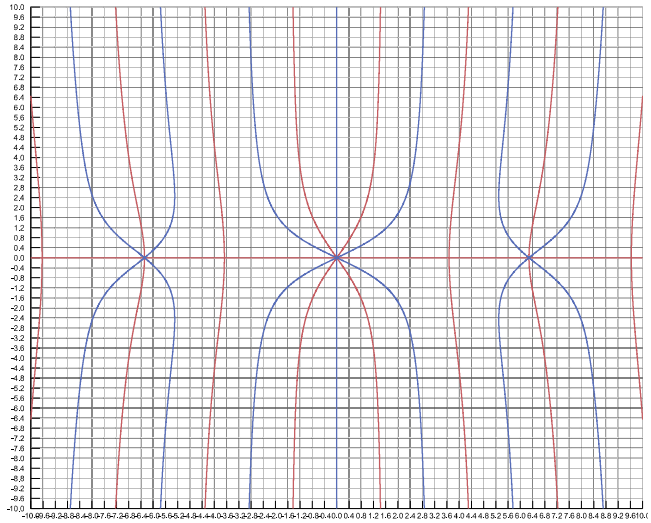
In [373]:
```
n = 2
beta0 = -1
beta1 = -1
theta = 0
a = e^(im*theta)

t = symbols("t")
symPFunctions = [-1 0 0]
interval = (0,1)
symL = SymLinearDifferentialOperator(symPFunctions, interval, t)
L = get_L(symL)
b = [1 0; 0 1]
beta = [beta0 0; 0 beta1]
U = VectorBoundaryForm(b, beta)
f(x) = sin(x*2*pi)
x = symbols("x")
fSym = sin(x*2*PI)
check_boundaryConditions(L, U, fSym)
```

Out[373]: true

In [374]:
```
adjointU = get_adjointU(L, U)
delta = get_delta(adjointU; symbolic = true)
lambda = free_symbols(delta)[1]
x = symbols("x", real = true)
y = symbols("y", real = true)
bivariateDelta = subs(delta, lambda, x+im*y)
plot_levelCurves(bivariateDelta; width = 2500, height = 2000)
```

Out[374]:



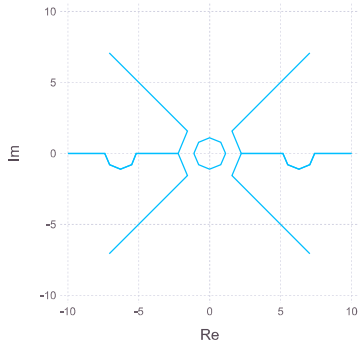






In [375]:
```
zeroList = [0, 2*pi, -2*pi]
(gammaAPlus, gammaAMinus, gamma0Plus, gamma0Minus) = get_gamma(a, n, zeroList)
gamma = collect(Iterators.flatten([gammaAPlus, gammaAMinus, gamma0Plus, gamma0Minus]))
plot_contour(gamma)
```

Out[375]:




In [376]:
```
(FPlusSym, FMinusSym) = get_FPlusMinus(adjointU; symbolic = true)
FPlusSymF = prettyPrint(simplify(FPlusSym(f)))
```

Out[376]:

$$\frac{2\left(-e^{i\lambda}+1\right)\left(0.785\lambda^8e^{i\lambda}-0.785\lambda^8-0.014i\lambda^7e^{i\lambda}-0.014i\lambda^7+31.64\lambda^6e^{i\lambda}-31.64\lambda^6+19.892i\lambda^5e^{i\lambda}+19.892i\lambda^5+773.28\lambda^4e^{i\lambda}-773.28\lambda^4-7508.255i\lambda^3e^{i\lambda}-7508.255i\lambda^3+138882.961\lambda^2e^{i\lambda}-138882.961\lambda^2+743198.707i\lambda e^{i\lambda}+743198.707i\lambda-1486397.414e^{i\lambda}+1486397.414\right)e^{-i\lambda}}{\pi\lambda^{10}\left(\cos\left(\lambda\right)-1\right)}$$

In [377]:
```
function FPlusSymFunc(lambda)
    result = (2*(-e^(im*lambda)+1) * (0.785*lambda^8*e^(im*lambda) - 0.785*lambda^8 - 0.014*im*lambda^7*e^(im*lambda) - 0.014*im*lambda^7 + 31.64*lambda^6*e^(im*lambda) - 31.64*lambda^6 + 19.892*im*lambda^5*e^(im*lambda) + 19.892*im*lambda^5 + 773.28*lambda^4*e^(im*lambda) - 773.28*lambda^4 - 7508.255*im*lambda^3*e^(im*lambda) - 7508.255*im*lambda^3 + 138882.961*lambda^2*e^(im*lambda) - 138882.961*lambda^2 + 743198.707*im*lambda*e^(im*lambda) + 743198.707*im*lambda - 1486397.414*e^(im*lambda) + 1486397.414)*e^(-im*lambda))/(pi*lambda^10*(cos(lambda)-1))
    return result
end
```

Out[377]: FPlusSymFunc (generic function with 1 method)

**Contents**



In [378]:
```
FMinusSymF = prettyPrint(FMinusSym(f))
```

Out[378]:

$$\frac{2\left(-e^{i\lambda}+1\right)\left(-0.785\lambda^8e^{i\lambda}+0.785\lambda^8+0.014i\lambda^7e^{i\lambda}+0.014i\lambda^7-31.64\lambda^6e^{i\lambda}+31.64\lambda^6-19.892i\lambda^5e^{i\lambda}-19.892i\lambda^5-773.28\lambda^4e^{i\lambda}+773.28\lambda^4+7508.255i\lambda^3e^{i\lambda}+7508.255i\lambda^3-138882.961\lambda^2e^{i\lambda}+138882.961\lambda^2-743198.707i\lambda e^{i\lambda}-743198.707i\lambda+1486397.414e^{i\lambda}-1486397.414\right)e^{-2i\lambda}}{\pi\lambda^{10}\left(-\cos\left(\lambda\right)+1\right)}$$

In [379]:
```
function FMinusSymFunc(lambda)
    result = (2*(-e^(im*lambda)+1) * (-0.785*lambda^8*e^(im*lambda) + 0.785*lambda^8 + 0.014*im*lambda^7*e^(im*lambda) + 0.014*im*lambda^7 - 31.64*lambda^6*e^(im*lambda) + 31.64*lambda^6 - 19.892*im*lambda^5*e^(im*lambda) - 19.892*im*lambda^5 - 773.28*lambda^4*e^(im*lambda) + 773.28*lambda^4 + 7508.255*im*lambda^3*e^(im*lambda) + 7508.255*im*lambda^3 - 138882.961*lambda^2*e^(im*lambda) + 138882.961*lambda^2 - 743198.707*im*lambda*e^(im*lambda) - 743198.707*im*lambda + 1486397.414*e^(im*lambda) - 1486397.414) * e^(-2*im*lambda))/(pi*lambda^10*(-cos(lambda)+1))
    return result
end
```

Out[379]: FMinusSymFunc (generic function with 1 method)

In [380]:
```
q = solve_IBVP(L, U, a, zeroList, f; FPlusFunc = FPlusSymFunc, FMinusFunc = FMinusSymFunc)
```

Out[380]: q (generic function with 1 method)





**Contents**



In [381]:
```
tic()
q(1/2, 1/2)
toc()
# quadgk() is slow around zeroes of Delta (poles)
```

```
integrandPlus = 0.013769402222841216 - 0.006604001599243898im
integrandMinus = 0.005117339626383156 - 0.041194772103712281m
elapsed time: 0.000742296 seconds
gamma0Plus = Any[Complex{Float64}[-1.11072+1.36024e-16im, -0.785398+0.785398im, 6.8012e-17+1.11072im, 0.785398+0.785398im, 1.11072+0.0im, 0.785398-0.785398im, 6.8012e-17-1.11072im, -0.785398-0.785398im, -1.11072+1.36024e-16im]]
path = Complex{Float64}[-1.11072+1.36024e-16im, -0.785398+0.785398im, 6.8012e-17+1.11072im, 0.785398+0.785398im, 1.11072+0.0im, 0.785398-0.785398im, 6.8012e-17-1.11072im, -0.785398-0.785398im, -1.11072+1.36024e-16im]
elapsed time: 10.156287285 seconds
elapsed time: 10.157382224 seconds
int_0_+ = 0.00037599136340613694 + 4.4745495122309166e-13im
gammaAPlus = Any[Complex{Float64}[7.07107+7.07107im, 1.5708+1.5708im, 2.22144+5.84189e-17im, 5.17246+1.36024e-16im, 5.49779-0.785398im, 6.28319-1.11072im, 7.06858-0.785398im, 7.39391+0.0im, 10.0+0.0im], Complex{Float64}[-10.0+1.22465e-15im, -7.39391+1.36024e-16im, -7.06858-0.785398im, -6.28319-1.11072im, -5.49779-0.785398im, -5.17246-2.72048e-16im, -2.22144+2.72048e-16im, -1.5708+1.5708im, -7.07107+7.07107im]]
path = Complex{Float64}[7.07107+7.07107im, 1.5708+1.5708im, 2.22144+5.84189e-17im, 5.17246+1.36024e-16im, 5.49779-0.785398im, 6.28319-1.11072im, 7.06858-0.785398im, 7.39391+0.0im, 10.0+0.0im]
elapsed time: 0.000598361 seconds
path = Complex{Float64}[-10.0+1.22465e-15im, -7.39391+1.36024e-16im, -7.06858-0.785398im, -6.28319-1.11072im, -5.49779-0.785398im, -5.17246-2.72048e-16im, -2.22144+2.72048e-16im, -1.5708+1.5708im, -7.07107+7.07107im]
elapsed time: 0.000564433 seconds
elapsed time: 0.002126648 seconds
int_a_+ = -2.3835726671437045e-5 - 2.6686552273558206e-15im
gamma0Minus = Any[]
elapsed time: 3.599e-6 seconds
int_0_- = 0
gammaAMinus = Any[Complex{Float64}[10.0+0.0im, 7.39391+0.0im, 7.06858-0.785398im, 6.28319-1.11072im, 5.49779-0.785398im, 5.17246+1.36024e-16im, 2.22144+5.84189e-17im, 1.5708-1.5708im, 7.07107-7.07107im], Complex{Float64}[-7.07107-7.07107im, -1.5708-1.5708im, -2.22144+2.72048e-16im, -5.17246-2.72048e-16im, -5.49779-0.785398im, -6.28319-1.11072im, -7.06858-0.785398im, -7.39391+1.36024e-16im, -10.0+1.22465e-15im]]
path = Complex{Float64}[10.0+0.0im, 7.39391+0.0im, 7.06858-0.785398im, 6.28319-1.11072im, 5.49779-0.785398im, 5.17246+1.36024e-16im, 2.22144+5.84189e-17im, 1.5708-1.5708im, 7.07107-7.07107im]
elapsed time: 0.210465143 seconds
path = Complex{Float64}[-7.07107-7.07107im, -1.5708-1.5708im, -2.22144+2.72048e-16im, -5.17246-2.72048e-16im, -5.49779-0.785398im, -6.28319-1.11072im, -7.06858-0.785398im, -7.39391+1.36024e-16im, -10.0+1.22465e-15im]
elapsed time: 0.000716594 seconds
elapsed time: 0.213329975 seconds
int_a_- = 2.3835726645822938e-5 - 5.652220363841845e-15im
elapsed time: 10.598060468 seconds
```

Out[381]: 10.598060468

In [382]:
```
t = 0.1
# Gadfly.plot(x -> real(q(x,t)), 0, 1)
# Slow if FPlusSymFunc, FMinusSymFunc contain many high powers of lambda (long compilation time of integrand Plus, integrandMinus at each x)
# https://github.com/JuliaLang/julia/issues/19158
```

Out[382]: 0.1







**Case 1.2**

Now, for $b_0, b_1 \in \hat{\mathbb{C}}$, consider the following boundary conditions $\Phi$:

$$\varphi(0) + b_0\varphi'(0) = 0$$
$$\varphi(1) + b_1\varphi'(1) = 0.$$

We note that in complete form,

- $\Phi$ is given by

$$1 \cdot \varphi(0) + 0 \cdot \varphi(1) + b_0 \cdot \varphi^{(1)}(0) + 0 \cdot \varphi^{(1)}(1) = 0$$
$$0 \cdot \varphi(0) + 1 \cdot \varphi(1) + 0 \cdot \varphi^{(1)}(0) + b_1 \cdot \varphi^{(1)}(1) = 0.$$

  Thus,

$$b = \begin{bmatrix} 1 & b_0 \\ 0 & 0 \end{bmatrix}, \quad \beta = \begin{bmatrix} 0 & 0 \\ 1 & b_1 \end{bmatrix}.$$

  Thus, $f(x) \in \Phi$ needs to satisfy

$$Uf = \begin{bmatrix} 1 & b_0 & 0 & 1 \\ 0 & 0 & 1 & b_1 \end{bmatrix} \begin{bmatrix} f(0) \\ f'(0) \\ f(1) \\ f'(1) \end{bmatrix} = 0.$$

**Case 2.1**

Suppose $n = 3$. Then

$$S\phi(x) = (-i)^3\phi^{(3)}(x) = i\phi^{(3)}.$$

Suppose $a = \pm i$.

For $\beta_0, \beta_1, \beta_2 \in \hat{\mathbb{C}}$ ($\mathbb{C}$ including $0$ and $\infty$), consider the following boundary conditions $\Phi$:

$$\phi(0) + \beta_0\phi(1) = 0$$
$$\phi^{(1)}(0) + \beta_1\phi^{(1)}(1) = 0$$
$$\phi^{(2)}(0) + \beta_2\phi^{(2)}(1) = 0.$$

We note that in complete form,

- $S$ is given by

$$S\phi = p_0\phi^{(3)} + p_1\phi^{(2)} + p_2\phi^{(1)} + p_3\phi$$

  where $p_0 = i, p_1 = p_2 = p_3 = 0$.
- $\Phi$ is given by

$$1 \cdot \phi(0) + \beta_0 \cdot \phi(1) + 0 \cdot \phi^{(1)}(0) + 0 \cdot \phi^{(1)}(1) + 0 \cdot \phi^{(2)}(0) + 0 \cdot \phi^{(2)}(1) = 0$$
$$0 \cdot \phi(0) + 0 \cdot \phi(1) + 1 \cdot \phi^{(1)}(0) + \beta_1 \cdot \phi^{(1)}(1) + 0 \cdot \phi^{(2)}(0) + 0 \cdot \phi^{(2)}(1) = 0$$
$$0 \cdot \phi(0) + 0 \cdot \phi(1) + 0 \cdot \phi^{(1)}(0) + 0 \cdot \phi^{(1)}(1) + 1 \cdot \phi^{(2)}(0) + \beta_2 \cdot \phi^{(2)}(1) = 0.$$

  Thus,

$$b = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad \beta = \begin{bmatrix} \beta_0 & 0 & 0 \\ 0 & \beta_1 & 0 \\ 0 & 0 & \beta_2 \end{bmatrix}.$$

  Thus, $f(x) \in \Phi$ needs to satisfy

$$Uf = \begin{bmatrix} 1 & 0 & 0 & \beta_0 & 0 & 0 \\ 0 & 1 & 0 & 0 & \beta_1 & 0 \\ 0 & 0 & 1 & 0 & 0 & \beta_2 \end{bmatrix} \begin{bmatrix} f(0) \\ f'(0) \\ f''(0) \\ f(1) \\ f'(1) \\ f''(1) \end{bmatrix} = 0.$$

**Case 2.2**

Now, for $b_0 \in \hat{\mathbb{C}}$, consider the following boundary conditions $\Phi$:

$$\begin{aligned} \varphi(0) &= 0 \\ \varphi(1) &= 1 \\ \varphi^{(1)}(0) + b_0\varphi^{(1)}(1) &= 0. \end{aligned}$$

We note that in complete form,

- $\Phi$ is given by

$$\begin{aligned} 1\cdot\varphi(0) + 0\cdot\varphi(1) + 0\cdot\varphi^{(1)}(0) + 0\cdot\varphi^{(1)}(1) + 0\cdot\varphi^{(2)}(0) + 0\cdot\varphi^{(2)}(1) &= 0 \\ 0\cdot\varphi(0) + 1\cdot\varphi(1) + 0\cdot\varphi^{(1)}(0) + 0\cdot\varphi^{(1)}(1) + 0\cdot\varphi^{(2)}(0) + 0\cdot\varphi^{(2)}(1) &= 0 \\ 0\cdot\varphi(0) + 0\cdot\varphi(1) + 1\cdot\varphi^{(1)}(0) + b_0\cdot\varphi^{(1)}(1) + 0\cdot\varphi^{(2)}(0) + 0\cdot\varphi^{(2)}(1) &= 0 \end{aligned}$$

  Thus,

$$b = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, \quad \beta = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & b_0 & 0 \end{bmatrix}.$$

  Thus, $f(x) \in \Phi$ needs to satisfy

$$Uf = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & b_0 & 0 \end{bmatrix} \begin{bmatrix} f(0) \\ f'(0) \\ f''(0) \\ f(1) \\ f'(1) \\ f''(1) \end{bmatrix} = 0.$$





**Contents**
7.1 check_boundaryConditions( U, fSym)
7.2 get_MPlusMinus(adjointU; symbolic, generic)
7.3 get_M(adjointU; symbolic, generic)
7.4 get_delta(adjointU; symbolic, generic)
7.5 get_Xlj(adjointU, l, j; symbolic, generic)
7.6 get_ChebyshevIntegral(l, f; symbolic, lambda, alpha)
7.6.1 get_ChebyshevTermInte( symbolic, c)
7.6.1.1 get_alpha(m, n)
7.6.2 get_ChebyshevCoeffici
7.6.3 get_ChebyshevIntegral f; symbolic, lambda, alpha)
7.7 get_FPlusMinus(adjointU; symbolic, generic)
7.8 get_gamma(a, n, zeroList; infty, nGon)
7.8.1 get_gammaAAngles(a, n; symbolic)
7.8.2 get_gammaAAnglesSplit n; symbolic)
7.8.3 pointOnSector(z, sectorAngles)
7.8.4 pointInSector(z, sectorAngles)
7.8.5 pointExSector(z, sectorAngles)
7.8.6 get_epsilon(zeroList, a, n)
7.8.7 get_nGonAroundZero(ze epsilon, n)
7.8.8 get_gamma(a, n, zeroList; infty, nGon)
7.9 plot_contour(gamma; infty)
7.10 solve_IBVP(L, U, a, zeroList, f; FPlusFunc, FMinusFunc, pDerivMatrix, infty)
8 Verifying the formulas of $F_\lam
8.1 Problem 1
8.2 Problem 2

# 8 Verifying the formulas of $F_\lambda^+$, $F_\lambda^-$

## 8.1 Problem 1

$$\begin{aligned} q_t(x,t) + q_{xxx}(x,t) &= 0, & (x,t) &\in (0,1)\times(0,T) \\ q(x,0) &= f(x), & x &\in [0,1] \\ q(0,t) &= 0, & t &\in [0,T] \\ q(1,t) &= 0 & t &\in [0,T] \\ q_x(1,t) &= \frac{1}{2}q_x(0,t) & t &\in [0,T]. \end{aligned}$$

$$\begin{aligned} F_\lambda^+(f) = \frac{1}{2\pi\Delta(\lambda)}\Big[&\mathrm{FT}[f](\lambda)(e^{i\lambda} + 2\alpha e^{-i\alpha\lambda} + 2\alpha^2 e^{-i\alpha^2\lambda}) + \mathrm{FT}[f](\alpha\lambda)(\alpha e^{i\alpha\lambda} - 2\alpha e^{-i\lambda}) \\ &+\mathrm{FT}[f](\alpha^2\lambda)(\alpha^2 e^{i\alpha\lambda} - 2\alpha^2 e^{-i\lambda})\Big] \\ F_\lambda^-(f) = \frac{e^{-i\lambda}}{2\pi\Delta(\lambda)}\Big[&-\mathrm{FT}[f](\lambda)(2 + \alpha^2 e^{-\alpha\lambda} + \alpha e^{-\alpha^2\lambda}) - \alpha\mathrm{FT}[f](\alpha\lambda)(2 - e^{-i\alpha^2\lambda}) \\ &-\alpha^2\mathrm{FT}[f](\alpha^2\lambda)(2 - e^{-i\alpha\lambda})\Big], \end{aligned}$$

where

$$\Delta(\lambda) = e^{i\lambda} + \alpha e^{i\alpha\lambda} + \alpha^2 e^{i\alpha^2\lambda} + 2(e^{-i\lambda} + \alpha e^{-i\alpha\lambda} + \alpha^2 e^{-i\alpha^2\lambda}).$$

In [383]:
```
n = 3

t = symbols("t")
symPFunctions = [1 0 0 0]
interval = (0, 1)
symL = SymLinearDifferentialOperator(symPFunctions, interval, t)
L = get_L(symL)
M = [1 0 0; 0 0 0; 0 1 0]
N = [0 0 0; 1 0 0; 0 -2 0]
U = VectorBoundaryForm(M, N)
```

Out[383]: VectorBoundaryForm([1 0 0; 0 0 0; 0 1 0], [0 0 0; 1 0 0; 0 -2 0])

In [384]:
```
c = symbols("c")
FT = SymFunction("FT[f]")(c)
FTc = symbols("FT[f](c)")
alpha = symbols("alpha")
lambda = symbols("lambda")
```

Out[384]: $\lambda$

In [385]:
```
deltaFormula = e^(im*lambda) + alpha*e^(im*alpha*lambda) + (alpha)^2*e^(im*(alpha)^2*lambda) + 2*(e^(-im*lambda) + alpha*e^(-im*alpha*lambda) + (alpha)^2*e^(-im*(alpha)^2*lambda))
```

Out[385]: $\alpha^2 e^{i\alpha^2\lambda} + 2\alpha^2 e^{-i\alpha^2\lambda} + \alpha e^{i\alpha\lambda} + 2\alpha e^{-i\alpha\lambda} + e^{i\lambda} + 2e^{-i\lambda}$

In [386]:
```
FPlusFormula = 1/(2*PI*deltaFormula) * (FT(lambda)*(e^(im*lambda) + 2*alpha*e^(-im*alpha*lambda) + 2*alpha^2*e^(-im*alpha^2*lambda)) + FT(alpha*lambda)*(alpha*e^(im*alpha*lambda) - 2*alpha*e^(-im*lambda)) + FT(alpha^2*lambda)*(alpha^2*e^(im*alpha^2*lambda) - 2*alpha^2*e^(-im*lambda)))
```

Out[386]: $$\frac{\left(\alpha e^{i\alpha\lambda} - 2\alpha e^{-i\lambda}\right)\mathrm{FT}[\mathrm{f}]\left(\alpha\lambda\right) + \left(\alpha^2 e^{i\alpha^2\lambda} - 2\alpha^2 e^{-i\lambda}\right)\mathrm{FT}[\mathrm{f}]\left(\alpha^2\lambda\right) + \left(2\alpha^2 e^{-i\alpha^2\lambda} + 2\alpha e^{-i\alpha\lambda} + e^{i\lambda}\right)\mathrm{FT}[\mathrm{f}]\left(\lambda\right)}{2\pi\left(\alpha^2 e^{i\alpha^2\lambda} + 2\alpha^2 e^{-i\alpha^2\lambda} + \alpha e^{i\alpha\lambda} + 2\alpha e^{-i\alpha\lambda} + e^{i\lambda} + 2e^{-i\lambda}\right)}$$

In [387]:
```
FMinusFormula = (e^(-im*lambda))/(2*PI*deltaFormula) * (-FT(lambda)*(2+alpha^2*e^(-im*alpha*lambda)+alpha*e^(-im*alpha^2*lambda)) - alpha*FT(alpha*lambda)*(2-e^(-im*alpha^2*lambda)) - alpha^2*FT(alpha^2*lambda)*(2-e^(-im*alpha*lambda)))
```

Out[387]: $$\frac{\left(-\alpha^2\left(2 - e^{-i\alpha\lambda}\right)\mathrm{FT}[\mathrm{f}]\left(\alpha^2\lambda\right) - \alpha\left(2 - e^{-i\alpha^2\lambda}\right)\mathrm{FT}[\mathrm{f}]\left(\alpha\lambda\right) - \left(\alpha^2 e^{-i\alpha\lambda} + \alpha e^{-i\alpha^2\lambda} + 2\right)\mathrm{FT}[\mathrm{f}]\left(\lambda\right)\right)e^{-i\lambda}}{2\pi\left(\alpha^2 e^{i\alpha^2\lambda} + 2\alpha^2 e^{-i\alpha^2\lambda} + \alpha e^{i\alpha\lambda} + 2\alpha e^{-i\alpha\lambda} + e^{i\lambda} + 2e^{-i\lambda}\right)}$$

In [388]:
```
adjointU = get_adjointU(L, U)
(FPlusSymGeneric, FMinusSymGeneric) = get_FPlusMinus(adjointU; symbolic = true, generic = true)
prettyPrint(simplify(FPlusSymGeneric))
```

Out[388]: $$\frac{\begin{array}{l}0.25\alpha\,\mathrm{FT}[\mathrm{f}]\left(\lambda\right)e^{i\lambda} - 0.5\alpha\,\mathrm{FT}[\mathrm{f}]\left(\lambda\right)e^{i\lambda(\alpha^2+1)} - 0.25\alpha\,\mathrm{FT}[\mathrm{f}]\left(\alpha\lambda\right)e^{i\alpha\lambda} + 0.5\alpha\,\mathrm{FT}[\mathrm{f}]\left(\alpha\lambda\right)e^{i\alpha\lambda(\alpha+1)} + 0.5\,\mathrm{FT}[\mathrm{f}]\\ \left(\lambda\right)e^{i\lambda(\alpha+1)} - 0.5\,\mathrm{FT}[\mathrm{f}]\left(\lambda\right)e^{i\lambda(\alpha^2+1)} - 0.25\,\mathrm{FT}[\mathrm{f}]\left(\alpha\lambda\right)e^{i\alpha\lambda} + 0.5\,\mathrm{FT}[\mathrm{f}]\left(\alpha\lambda\right)e^{i\alpha\lambda(\alpha+1)} + 0.25\,\mathrm{FT}[\mathrm{f}]\left(\alpha^2\lambda\right)e^{i\alpha^2\lambda}\\ - 0.5\,\mathrm{FT}[\mathrm{f}]\left(\alpha^2\lambda\right)e^{i\alpha\lambda(\alpha+1)}\end{array}}{\pi\left(0.5\alpha e^{i\lambda} - 0.5\alpha e^{i\alpha\lambda} - \alpha e^{i\lambda(\alpha^2+1)} + \alpha e^{i\alpha\lambda(\alpha+1)} - 0.5e^{i\alpha\lambda} + 0.5e^{i\alpha^2\lambda} + e^{i\lambda(\alpha+1)} - e^{i\lambda(\alpha^2+1)}\right)}$$

In [389]:
```
prettyPrint(simplify(FMinusSymGeneric))
```

Out[389]: $$\frac{\begin{array}{l}0.25\alpha\,\mathrm{FT}[\mathrm{f}]\left(\lambda\right)e^{i\alpha\lambda} - 0.5\alpha\,\mathrm{FT}[\mathrm{f}]\left(\lambda\right)e^{i\alpha\lambda(\alpha+1)} - 0.25\alpha\,\mathrm{FT}[\mathrm{f}]\left(\alpha\lambda\right)e^{i\alpha\lambda} + 0.5\alpha\,\mathrm{FT}[\mathrm{f}]\left(\alpha\lambda\right)e^{i\alpha\lambda(\alpha+1)} + 0.25\,\mathrm{FT}[\mathrm{f}]\\ \left(\lambda\right)e^{i\alpha\lambda} - 0.25\,\mathrm{FT}[\mathrm{f}]\left(\lambda\right)e^{i\alpha^2\lambda} - 0.25\,\mathrm{FT}[\mathrm{f}]\left(\alpha\lambda\right)e^{i\alpha\lambda} + 0.5\,\mathrm{FT}[\mathrm{f}]\left(\alpha\lambda\right)e^{i\alpha\lambda(\alpha+1)} + 0.25\,\mathrm{FT}[\mathrm{f}]\left(\alpha^2\lambda\right)e^{i\alpha^2\lambda}\\ - 0.5\,\mathrm{FT}[\mathrm{f}]\left(\alpha^2\lambda\right)e^{i\alpha\lambda(\alpha+1)}\end{array}}{\pi\left(0.5\alpha e^{i\lambda} - 0.5\alpha e^{i\alpha\lambda} - \alpha e^{i\lambda(\alpha^2+1)} + \alpha e^{i\alpha\lambda(\alpha+1)} - 0.5e^{i\alpha\lambda} + 0.5e^{i\alpha^2\lambda} + e^{i\lambda(\alpha+1)} - e^{i\lambda(\alpha^2+1)}\right)}$$







In [390]:
```
function test_formula(formula1, formula2)
    results = []
    for i = 1:100
#         println("i = $i")
        realLambda = rand(Uniform(-50.0,50.0), 1, 1)[1]
        imagLambda = rand(Uniform(-50.0,50.0), 1, 1)[1]
        lambdaVal = realLambda + im*imagLambda
#         println("Lambda = $lambdaVal")
#         println("------------------------------------------------------------------")

        # coefficient of FT[f](lambda)
#         println(FT(lambda))
        coeff1 = subs(formula1, (FT(lambda), 1), (FT(alpha*lambda), 0), (FT(alpha^2*lambda), 0))
        coeff2 = subs(formula2, (FT(lambda), 1), (FT(alpha*lambda), 0), (FT(alpha^2*lambda), 0))
        value1 = SymPy.N(subs(coeff1, (alpha, e^(2*pi*im/n)), (lambda, lambdaVal)))
        value2 = SymPy.N(subs(coeff2, (alpha, e^(2*pi*im/n)), (lambda, lambdaVal)))
#         println("Formula1 = $value1")
#         println("Formula2 = $value2")
#         println("------------------------------------------------------------------")
        result = is_approx(value1-value2, 0)
        append!(results, result)

        # coefficient of FT[f](alpha*Lambda)
#         println(FT(alpha*lambda))
        coeff1 = subs(formula1, (FT(lambda), 0), (FT(alpha*lambda), 1), (FT(alpha^2*lambda), 0))
        coeff2 = subs(formula2, (FT(lambda), 0), (FT(alpha*lambda), 1), (FT(alpha^2*lambda), 0))
        value1 = SymPy.N(subs(coeff1, (alpha, e^(2*pi*im/n)), (lambda, lambdaVal)))
        value2 = SymPy.N(subs(coeff2, (alpha, e^(2*pi*im/n)), (lambda, lambdaVal)))
#         println("Formula1 = $value1")
#         println("Formula2 = $value2")
#         println("------------------------------------------------------------------")
        result = is_approx(value1-value2, 0)
        append!(results, result)

        # coefficient of FT[f](alpha^2*Lambda)
#         println(FT(alpha^2*lambda))
        coeff1 = subs(formula1, (FT(lambda), 0), (FT(alpha*lambda), 0), (FT(alpha^2*lambda), 1))
        coeff2 = subs(formula2, (FT(lambda), 0), (FT(alpha*lambda), 0), (FT(alpha^2*lambda), 1))
        value1 = SymPy.N(subs(coeff1, (alpha, e^(2*pi*im/n)), (lambda, lambdaVal)))
        value2 = SymPy.N(subs(coeff2, (alpha, e^(2*pi*im/n)), (lambda, lambdaVal)))
#         println("Formula1 = $value1")
#         println("Formula2 = $value2")
#         println("------------------------------------------------------------------")
        result = is_approx(value1-value2, 0)
        append!(results, result)
#         println("==================================================================")
    end
    return all(results)
end
```

Out[390]: test_formula (generic function with 1 method)

```
In [391]: test_formula(FPlusFormula, FPlusSymGeneric)
```

```
Out[391]: true
```

```
In [392]: test_formula(FMinusFormula, FMinusSymGeneric)
```

```
Out[392]: true
```

## 8.2 Problem 2

$$\begin{aligned} q_t(x,t) + q_{xxx}(x,t) &= 0, & (x,t) &\in (0,1)\times(0,T) \\ q(x,0) &= f(x), & x &\in [0,1] \\ q(0,t) &= 0, & t &\in [0,T] \\ q(1,t) &= 0 & t &\in [0,T] \\ q_x(1,t) &= 0 & t &\in [0,T]. \end{aligned}$$

$$F_\lambda^+(f) = \frac{1}{2\pi\Delta(\lambda)}\left[\mathrm{FT}[f](\lambda)(\alpha e^{-\alpha\lambda} + \alpha^2 e^{-i\alpha^2\lambda}) - (\alpha\mathrm{FT}[f](\alpha\lambda) + \alpha^2\mathrm{FT}[f](\alpha^2\lambda))e^{-i\lambda}\right]$$

$$F_\lambda^-(f) = \frac{e^{-i\lambda}}{2\pi\Delta(\lambda)}\left[-\mathrm{FT}[f](\lambda) - \alpha\mathrm{FT}[f](\alpha\lambda) - \alpha^2\mathrm{FT}[f](\alpha^2\lambda)\right],$$

where

$$\Delta(\lambda) = e^{-i\lambda} + \alpha e^{-i\alpha\lambda} + \alpha^2 e^{-i\alpha^2\lambda}.$$

```
In [393]: n = 3

          t = symbols("t")
          symPFunctions = [1 0 0 0]
          interval = (0, 1)
          symL = SymLinearDifferentialOperator(symPFunctions, interval, t)
          L = get_L(symL)
          M = [1 0 0; 0 0 0; 0 0 0]
          N = [0 0 0; 1 0 0; 0 1 0]
          U = VectorBoundaryForm(M, N)
```

```
Out[393]: VectorBoundaryForm([1 0 0; 0 0 0; 0 0 0], [0 0 0; 1 0 0; 0 1 0])
```

```
In [394]: c = symbols("c")
          FT = SymFunction("FT[f]")(c)
          FTc = symbols("FT[f](c)")
          alpha = symbols("alpha")
          lambda = symbols("lambda")
```

```
Out[394]: λ
```







```
In [395]: deltaFormula = e^(-im*lambda) + alpha*e^(-im*alpha*lambda) + alpha^2*e^(-im*alpha^2*lambda)
```

```
Out[395]: α²e^(-iα²λ) + αe^(-iαλ) + e^(-iλ)
```

```
In [396]: FPlusFormula = 1/(2*PI*deltaFormula) * (FT(lambda)*(alpha*e^(-im*alpha*lambda) + alpha^2*e^(-im*alpha^2*lambda)) - (alpha*FT(alpha*lambda) + alpha^2*FT(alpha^2*lambda))*e^(-im*lambda))
```

```
Out[396]: (-(α² FT[f](α²λ) + α FT[f](αλ)) e^(-iλ) + (α²e^(-iα²λ) + αe^(-iαλ)) FT[f](λ)) / (2π(α²e^(-iα²λ) + αe^(-iαλ) + e^(-iλ)))
```

```
In [397]: FMinusFormula = (e^(-im*lambda))/(2*PI*deltaFormula) * (-FT(lambda) - alpha*FT(alpha*lambda) - alpha^2*FT(alpha^2*lambda))
```

```
Out[397]: ((-α² FT[f](α²λ) - α FT[f](αλ) - FT[f](λ)) e^(-iλ)) / (2π(α²e^(-iα²λ) + αe^(-iαλ) + e^(-iλ)))
```

```
In [398]: adjointU = get_adjointU(L, U)
          (FPlusSymGeneric, FMinusSymGeneric) = get_FPlusMinus(adjointU; symbolic = true, generic = true)
          prettyPrint(simplify(FPlusSymGeneric))
```

```
Out[398]: (0.5(α FT[f](λ)e^(iλ(α²+1)) - α FT[f](αλ)e^(iαλ(α+1)) - FT[f](λ)e^(iλ(α+1)) + FT[f](λ)e^(iλ(α²+1)) - FT[f](αλ)e^(iαλ(α+1))
          + FT[f](α²λ)e^(iαλ(α+1))))
          / (π(αe^(iλ(α²+1)) - αe^(iαλ(α+1)) - e^(iλ(α+1)) + e^(iλ(α²+1))))
```

```
In [399]: prettyPrint(simplify(FMinusSymGeneric))
```

```
Out[399]: (0.5(α FT[f](λ) - α FT[f](αλ) - FT[f](αλ) + FT[f](α²λ)) e^(iαλ(α+1)))
          / (π(αe^(iλ(α²+1)) - αe^(iαλ(α+1)) - e^(iλ(α+1)) + e^(iλ(α²+1))))
```

```
In [400]: test_formula(FPlusFormula, FPlusSymGeneric)
```

```
Out[400]: true
```

```
In [401]: test_formula(FMinusFormula, FMinusSymGeneric)
```

```
Out[401]: true
```