```julia
import CairoMakie as CM
```

```julia
CM.activate!(type="png")
```

Utility functions...

```julia
md"""**Utility functions...**"""
```

```
fastcos (generic function with 1 method)
```

```julia
# Make this use cos() if your angles are not very small.
function fastcos(x)
    1 - x^2/2
end
```

```
fastatan (generic function with 1 method)
```

```julia
function fastatan(x)
    x-x^3/3
end
```

Kirchhoff-Fresnel diffraction integral

```julia
md"""# Kirchhoff-Fresnel diffraction integral"""
```

Aperture functions

Shape function for apertures. You can test the different functions at the end of this section.

Shape functions are used in the Setup struct; kwargs can be given as a dictionary.

```julia
md"""## Aperture functions

Shape function for apertures. You can test the different functions at the end of this
section.

Shape functions are used in the Setup struct; kwargs can be given as a dictionary."""
```

```
circular (generic function with 1 method)
```

```julia
function circular(x, y; radius=1000e-6)::Float16
    (x^2+y^2) <= radius^2 ? 1 : 0
end
```

```
double_circle (generic function with 1 method)
```

```julia
function double_circle(x, y; radius=1000e-6, dist=1000e-6)::Float16
    d = dist/2
    rsq = radius^2
    ysq = y^2
```

```
    (((x-d)^2+ysq) <= rsq || ((x+d)^2+ysq) <= rsq) ? 1 : 0
end
```

quadratic (generic function with 1 method)

```
function quadratic(x, y; side=1000e-6)::Float16
    (-side <= x && x <= side && -side <= y && y <= side) ? 1 : 0
end
```

slit (generic function with 1 method)

```
function slit(x, y; width=800e-6, height=1000)
    (-width/2 <= x && x <= width/2 && -height <= y && y <= height) ? 1 : 0
end
```

double_slit (generic function with 1 method)

```
function double_slit(x, y; width=800e-6, off=1000e-6, height=1000)
    (-width/2-off <= x && x <= width/2-off && -height <= y && y <= height) || (-width/2+off <= x && x <= width/2+off && -height <= y && y <= height) ? 1 : 0
end
```

smallgrate (generic function with 1 method)

```
function smallgrate(x, y; width=100e-6, off=200e-6, height=1000)
    s = abs(rem(x, off+width))
    ((x < 0 && s <= width) || (x >= 0 && s >= off)) ? 1 : 0
end
```

cross (generic function with 1 method)

```
function cross(x, y; width=300e6)
    (abs(x) <= width/2 || abs(y) <= width/2) ? 0 : 1
end
```

spikes (generic function with 1 method)

```
function spikes(x, y; width=100e-6, radius=1000e-6)
    circular(x, y, radius=radius) == 1 && cross(x, y, width=width) == 1
end
```

```
#circular_fft = convert.(Float16, FileIO.load("circle.png"))
```

from_raster (generic function with 1 method)

```
function from_raster(shape, maxdim)
    maxix = size(shape, 1)
    off = convert(Int, trunc((maxix)/2))
    return function(x,y)
        i, j = convert(Int, trunc(maxix*x/maxdim)), convert(Int, trunc(maxix*y/maxdim))
        shape[min(i+off+1, maxix), min(j+off+1, maxix)]
    end
end
```

grating2d (generic function with 1 method)

```
function grating2d(x, y; width=100e-6, off=200e-6)
    s = abs(rem(x, off+width))
    a = ((x < 0 && s <= width) || (x >= 0 && s >= off))
    t = abs(rem(y, off+width))
    b = ((y < 0 && t <= width) || (y >= 0 && t >= off))
```

```
    a && b
end
```

invertshape (generic function with 1 method)

```
function invertshape(s)
    if s <= 0.1
        1
    else
        0
    end
end
```

sample_shape (generic function with 1 method)

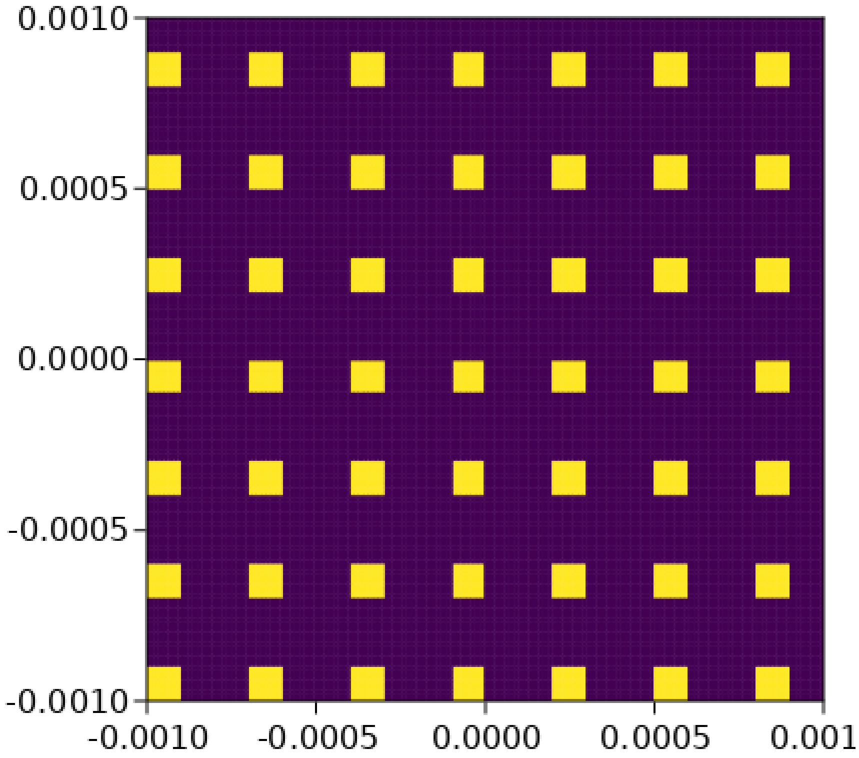
```
function sample_shape(f; dim=OUTER_DIM, res=100)::Tuple{LinRange{Float64},
Matrix{Float16}}
    field = zeros(Float16, res, res)
    xs = ys = LinRange(-dim/2, dim/2, res)
    for (i, x) = enumerate(xs)
        for (j, y) = enumerate(ys)
            field[i,j] = f(x, y)
        end
    end
    xs, field
end
```

show_shape (generic function with 1 method)

```
function show_shape(f; dim=OUTER_DIM, scan=OUTER_DIM/10)
    xs, field = sample_shape(f, dim=dim, res=convert(Int, div(dim,scan)))
    fig = CM.Figure(resolution=(440,400))
    CM.Axis(fig[1,1])
    CM.heatmap!(xs, xs, field)
    fig
end
```

0.3

```
begin
    # These constants are just used for testing the aperture functions.
    const OUTER_DIM = .002
    const SCREEN_DIM = 0.3
end
```




```julia
show_shape((x,y) -> grating2d(x, y), dim=OUTER_DIM, scan=OUTER_DIM/200)
```

Integral calculation

We integrate across the aperture, sampling both aperture and screen area.

Before doing that, we add some structs for configuring our calculations.

```julia
md"""## Integral calculation

We integrate across the aperture, sampling both aperture and screen area.

Before doing that, we add some structs for configuring our calculations."""
```

Source

```julia
Base.@kwdef struct Source
    x::Float64 = 0
    y::Float64 = 0
    z::Float64
end
```

```julia
struct Setup
    # (negative) position of source, left of aperture
    source::Vector{Source}
    # aperture shape function
    aperture::Function
    # aperture config
    aperture_kwargs::Dict{Symbol, Float64}
    # distance to screen
    screen_pos::Float64
    # wavelength
    lambda::Float64
```

```
end
```

default_setup = Setup([Source(0.0, -0.0, -30.0)], double_slit (generic function with 1 method), Dict()

```
# Distance of source, shape function, shape parameters, distance to screen, wavelength
default_setup = Setup([Source(z=-30, y=-.0)],
    double_slit, Dict(),
    30, 500e-9)
```

```
struct ScanParam
    aperture_size::Float64
    aperture::Float64
    screen_size::Float64
    screen::Float64
end
```

default_scan_param = ScanParam(0.005, 5.0e-5, 0.04, 0.0004)

```
default_scan_param = ScanParam(
    # Aperture scan size and scan step, in m, depends on aperture function
    0.005, .005/100,
    # Screen size and scan step in m.
    .04, .04/100)
```

Point source to screen

`calculate_integral()` assumes one or more point sources and a screen orthogonal to the (geometric) beam. It calculates the image on the screen, which is shown below.

```
md"""### Point source to screen

`calculate_integral()` assumes one or more point sources and a screen orthogonal to
the (geometric) beam. It calculates the image on the screen, which is shown below."""
```

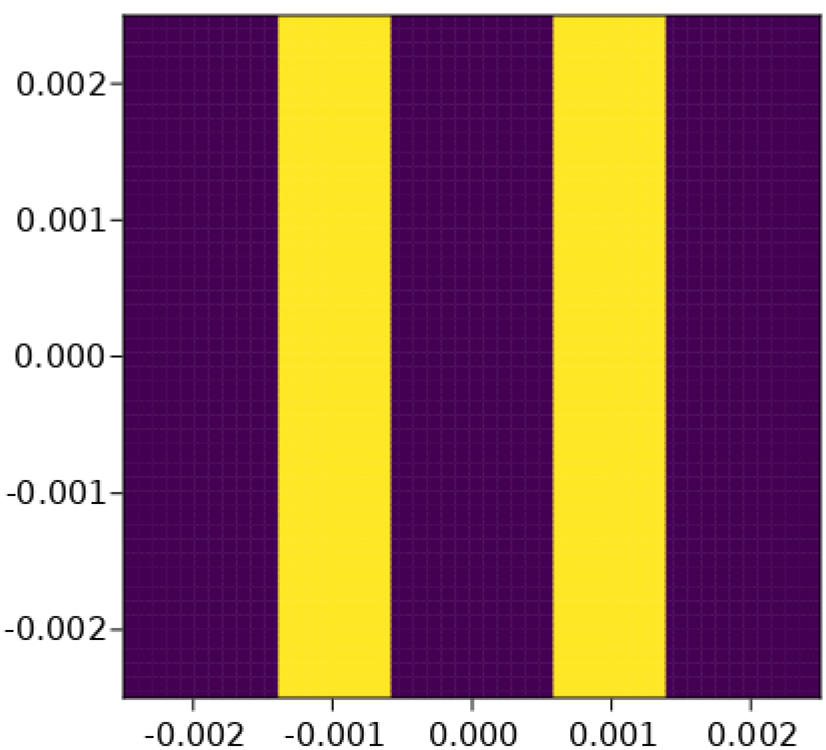
calculate_integral (generic function with 1 method)

```
function calculate_integral(setup::Setup; scan_param::ScanParam=default_scan_param)

    screendim = convert(Int, div(scan_param.screen_size, scan_param.screen))
    apdim = convert(Int, div(scan_param.aperture_size, scan_param.aperture))
    screen = zeros(Complex{Float64}, screendim, screendim)

    delta = scan_param.aperture_size / apdim

    all_aperture_coords = ((x,y)
        for x = LinRange(
                -scan_param.aperture_size/2, scan_param.aperture_size/2, apdim)
        , y = LinRange(
                -scan_param.aperture_size/2, scan_param.aperture_size/2, apdim))

    all_screen_coords = ((i, x, j, y)
        for (i, x) = enumerate(LinRange(
                -scan_param.screen_size/2, scan_param.screen_size/2, screendim))
        , (j, y) = enumerate(LinRange(
                -scan_param.screen_size/2, scan_param.screen_size/2, screendim)))
    k = 2pi/setup.lambda

    all_screen_coords = collect(all_screen_coords)
    count = 0
```

```

    @time begin for (apx, apy) = all_aperture_coords
        weight = setup.aperture(apx, apy; setup.aperture_kwargs...)
        if weight == 0
            continue
        end
        count += length(all_screen_coords)
        cdist = sqrt(apx^2+apy^2)
        dists = ((atan(sum(((s.x, s.y) .- (apx, apy)).^2)/abs(s.z)),
                    sqrt(sum(((apx,apy,0) .- (s.x, s.y, s.z)).^2)))
                for s = setup.source)
        sourceterms = [(fastcos(alpha), exp(-im * k * R)/R) for (alpha, R) = dists]

        Threads.@threads for (i, scx, j, scy) = all_screen_coords
            pointdist = sqrt((apx-scx)^2 + (apy-scy)^2)
            r = sqrt(pointdist^2 + setup.screen_pos^2)
            beta = fastatan(pointdist / abs(setup.screen_pos))
            fcb = fastcos(beta)
            term = exp(-im*k*r)/r * sum(
                    ((fcb+fca)/2 * t
                        for (fca, t) = sourceterms))

            screen[i, j] += weight * term * delta^2/(im*setup.lambda)
        end
    end
    end
    println("calculate_integral: $count iterations")
    abs.(screen).^2, LinRange(-scan_param.screen_size/2, scan_param.screen_size/2,
screendim)
end
```

Check the actual aperture used:

```
md"""**Check the actual aperture used:**"""
```



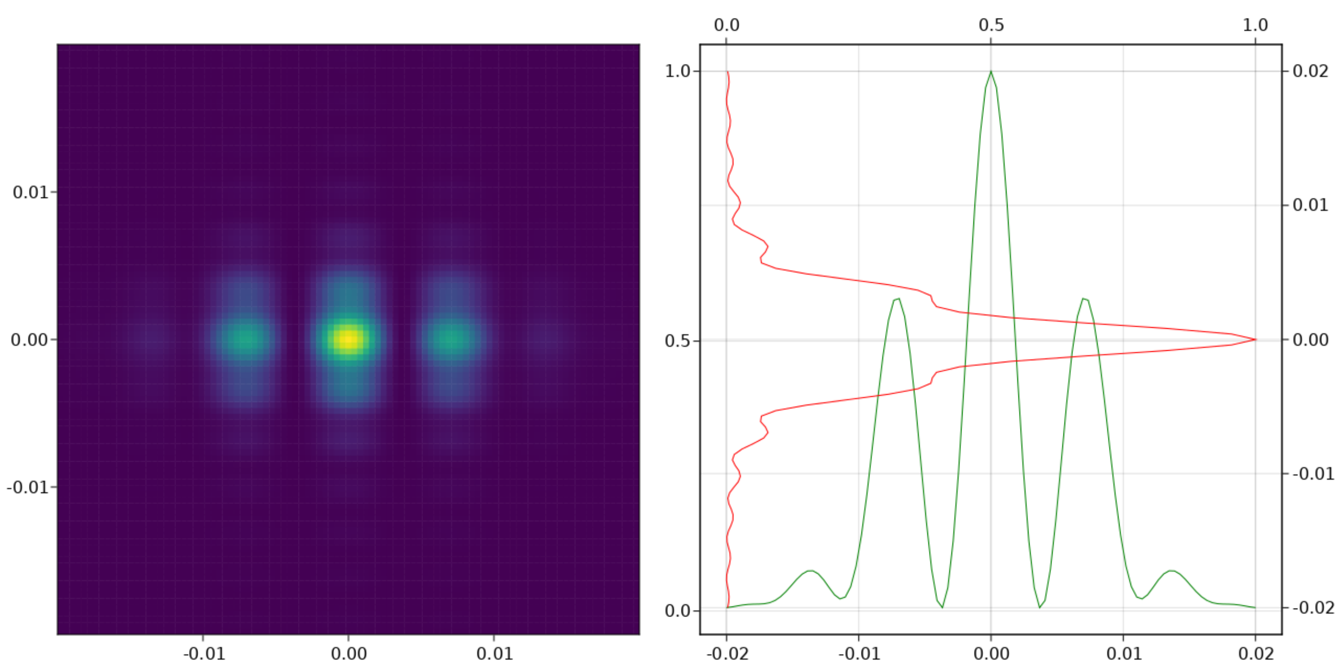

```
show_shape((x,y) -> default_setup.aperture(x, y; default_setup.aperture_kwargs...),
dim=default_scan_param.aperture_size, scan=default_scan_param.aperture)
```

plot_diffraction (generic function with 1 method)

```
function plot_diffraction()
    screen, coords = calculate_integral(default_setup, scan_param=default_scan_param);
    fig = CM.Figure(resolution=(1300, 650))
    CM.Axis(fig[1,1])
    CM.heatmap!(coords, coords, (screen))
    axh = CM.Axis(fig[1, 2])
    axv = CM.Axis(fig[1,2], xaxisposition=:top, yaxisposition=:right)
    mid = div(size(screen)[1], 2)
    CM.lines!(axh, coords, view(screen, :, mid)/maximum(view(screen, :, mid)),
        color=:green)
    CM.lines!(axv, view(screen, mid, :)/maximum(view(screen, mid, :)), coords,
        color=:red)
    fig
end
```

Show diffraction image as a heatmap with a crosssection (by default: along the middle, in x-orientation)

```
md"""**Show diffraction image as a heatmap with a crosssection (by default: along the
middle, in x-orientation)**"""
```




```
plot_diffraction()
```

Fraunhofer Integral

```
md"""## Fraunhofer Integral"""
```

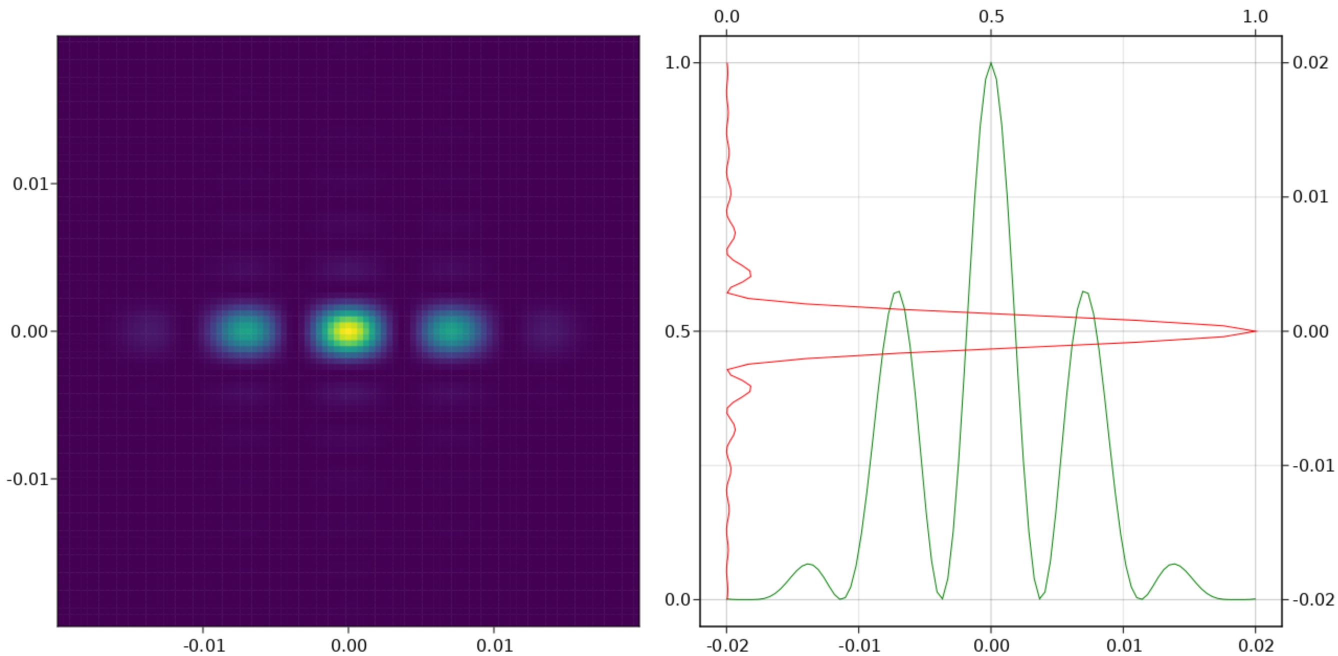
calculate_fraunhofer_integral (generic function with 1 method)

```
function calculate_fraunhofer_integral(setup::Setup;
scan_param::ScanParam=default_scan_param)

    screendim = convert(Int, div(scan_param.screen_size, scan_param.screen))
```

```
    apdim = convert(Int, div(scan_param.aperture_size, scan_param.aperture))
    screen = zeros(Complex{Float64}, screendim, screendim)
    delta = scan_param.aperture_size / apdim

    all_aperture_coords = ((x,y)
        for x = LinRange(
                -scan_param.aperture_size/2, scan_param.aperture_size/2, apdim)
        , y = LinRange(
                -scan_param.aperture_size/2, scan_param.aperture_size/2, apdim))

    all_screen_coords = ((i, x, j, y)
        for (i, x) = enumerate(LinRange(
                -scan_param.screen_size/2, scan_param.screen_size/2, screendim))
        , (j, y) = enumerate(LinRange(
                -scan_param.screen_size/2, scan_param.screen_size/2, screendim)))
    k = 2pi/setup.lambda

    all_screen_coords = collect(all_screen_coords)
    count = 0

    @time begin for (apx, apy) = all_aperture_coords
        if setup.aperture(apx, apy; setup.aperture_kwargs...) < 0.1
            continue
        end
        count += length(all_screen_coords)

        Threads.@threads for (i, scx, j, scy) = all_screen_coords
            #l, m = (sin(fastatan((scx-apx)/setup.screen_pos)),
            #       sin(fastatan((scy-apy)/setup.screen_pos)))
            l, m = scx/setup.screen_pos, scy/setup.screen_pos
            term = exp(-im*k*(l*apx + m*apy))
            screen[i, j] += term * delta^2
        end
    end
    end
    println("calculate_fraunhofer_integral: $count iterations")
    abs.(screen).^2, LinRange(-scan_param.screen_size/2, scan_param.screen_size/2,
screendim)
end
```

plot_fraunhofer_diffraction (generic function with 1 method)

```
function plot_fraunhofer_diffraction()
    screen, coords = calculate_fraunhofer_integral(default_setup,
        scan_param=default_scan_param);
    fig = CM.Figure(resolution=(1300, 650))
    CM.Axis(fig[1,1])
    CM.heatmap!(coords, coords, (screen))
    axh = CM.Axis(fig[1, 2])
    axv = CM.Axis(fig[1,2], xaxisposition=:top, yaxisposition=:right)
    mid = div(size(screen)[1], 2)
    CM.lines!(axh, coords, view(screen, :, mid)/maximum(view(screen, :, mid)),
        color=:green)
    CM.lines!(axv, view(screen, mid, :)/maximum(view(screen, mid, :)), coords,
        color=:red)
    fig
end
```




```
plot_fraunhofer_diffraction()
```

Diffraction pattern on plane

Here we calculate the intensities along a plane containing the (geometric) beam as it travels towards the screen.

The plane is configured in the `Plane` struct, which determines the rotation angle of the plane around the geometric beam axis, as well as the width and length of the plane.

Below we link these values with the aperture and screen settings used above, so that we always see the pattern on the way to the screen.

```
md"""### Diffraction pattern on plane

Here we calculate the intensities along a plane containing the (geometric) beam as it travels towards the screen.

The plane is configured in the `Plane` struct, which determines the rotation angle of the plane around the geometric beam axis, as well as the width and length of the plane.

Below we link these values with the aperture and screen settings used above, so that we always see the pattern on the way to the screen.
"""
```

```
struct Plane
    # 0 degrees = x plane
    angle::Float64
    width::Float64
    wscan::Float64
    length::Float64
    lscan::Float64
end
```

```
calculate_plane_integral (generic function with 1 method)
```

```
function calculate_plane_integral(setup::Setup, plane::Plane;
        scan_param::ScanParam=default_scan_param)

    screendim = (round(Int, div(plane.width, plane.wscan)),
        round(Int, div(plane.length, plane.lscan)))
    apdim = convert(Int, div(scan_param.aperture_size, scan_param.aperture))
    screen = zeros(Complex{Float64}, screendim)

    delta = scan_param.aperture_size / apdim
    k = 2pi / setup.lambda

    all_aperture_coords = ((x,y)
        for x = LinRange(
                -scan_param.aperture_size/2, scan_param.aperture_size/2, apdim)
        , y = LinRange(
                -scan_param.aperture_size/2, scan_param.aperture_size/2, apdim))

    # For stepping, it is important to have a very slight angle at the least.
    ang = plane.angle != 0 ? plane.angle : 0.1
    plane_n, plane_l_n = screendim
    cosine = cos(ang/180*pi)
    sine = sin(ang/180*pi)

    all_trans_coords = enumerate(zip(
            LinRange(-cosine*plane.width/2, cosine*plane.width/2, plane_n),
            LinRange(-sine*plane.width/2, sine*plane.width/2, plane_n)))

    all_plane_coords = ((i, x, y, j, z)
        for (i, (x, y)) = all_trans_coords
            for (j, z) = enumerate(LinRange(0, plane.length, plane_l_n))
                if i < plane_n && j < plane_l_n)
    all_plane_coords = collect(all_plane_coords)

    count = 0

    @time begin for (apx, apy) = all_aperture_coords
        if setup.aperture(apx, apy; setup.aperture_kwargs...) < 0.1
            continue
        end
        count += length(all_plane_coords)
        cdist = sqrt(apx^2+apy^2)
        dists = [(fastatan(sum(((s.x, s.y) .- (apx, apy)).^2)/abs(s.z)),
                    sqrt(sum(((apx,apy,0) .- (s.x, s.y, s.z)).^2)))
                for s = setup.source]
        sourceterms = [(alpha, exp(-im * k * R)/R) for (alpha, R) = dists]

        Threads.@threads for (i, x, y, j, z) = all_plane_coords
            pointdist = sqrt((apx-x)^2 + (apy-y)^2)
            r = sqrt(pointdist^2 + z^2)
            beta = atan(pointdist / abs(setup.screen_pos))
            term = exp(-im*k*r)/r * sum(
                    ((fastcos(beta)+fastcos(alpha))/2 * t
                        for (alpha, t) = sourceterms))
            # cos terms are only sometimes required
            K = 1/(im*setup.lambda)

            screen[i, j] += K * term * delta^2
        end
    end
    end

    println("calculate_plane_integral: $count iterations")
    (abs.(screen).^2,
        LinRange(-plane.width/2, plane.width/2, plane_n),
        LinRange(0, plane.length, plane_l_n))
```

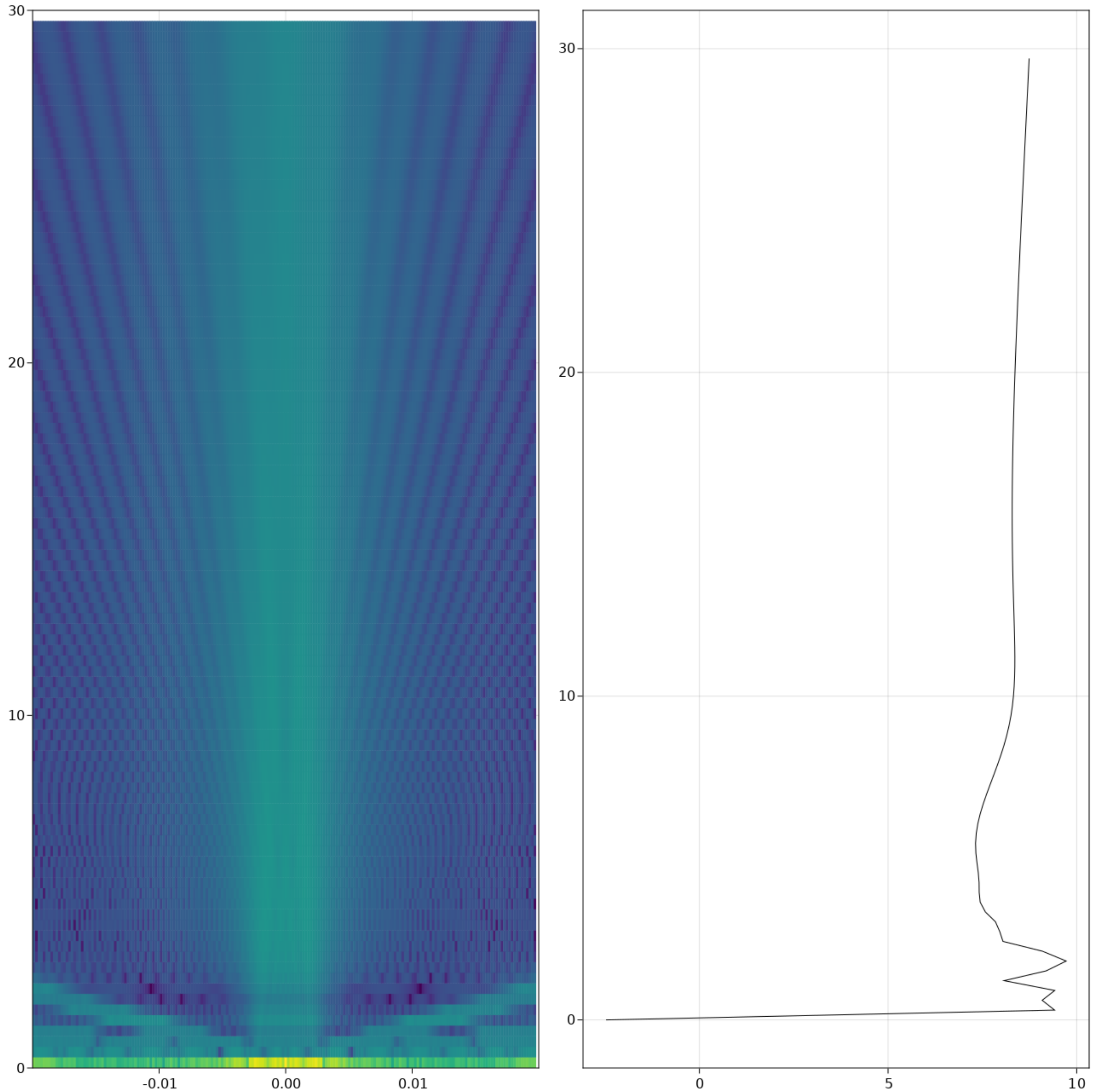
```
end
```

```
Plane(90.0, 0.04, 0.0002, 30.0, 0.3)
```

```
begin
    #plane = Plane(0, 0.025, 0.025/200, 20, 20/100)
    plane = Plane(90, default_scan_param.screen_size, default_scan_param.screen/2,
        default_setup.screen_pos, default_setup.screen_pos/100)
end
```

```
plot_diffraction_on_plane (generic function with 1 method)
```

```
function plot_diffraction_on_plane()
    screen, wcoords, lcoords = calculate_plane_integral(default_setup, plane);
    fig = CM.Figure(resolution=(1300, 1300))
    CM.Axis(fig[1,1])
    CM.heatmap!(wcoords, lcoords, log.(screen))
    CM.Axis(fig[1, 2])
    mid = div(size(screen)[1], 2)
    CM.lines!(-log.(screen[mid, :]), lcoords)
    fig
end
```




```
plot_diffraction_on_plane()
```

Babinet's principle

[Babinet's Principle](https://en.wikipedia.org/wiki/Babinet%27s_principle)

Show diffraction for inverted aperture.

Consider that the used aperture is square and usually of very limited length: Major artifacts will stem from that fact.

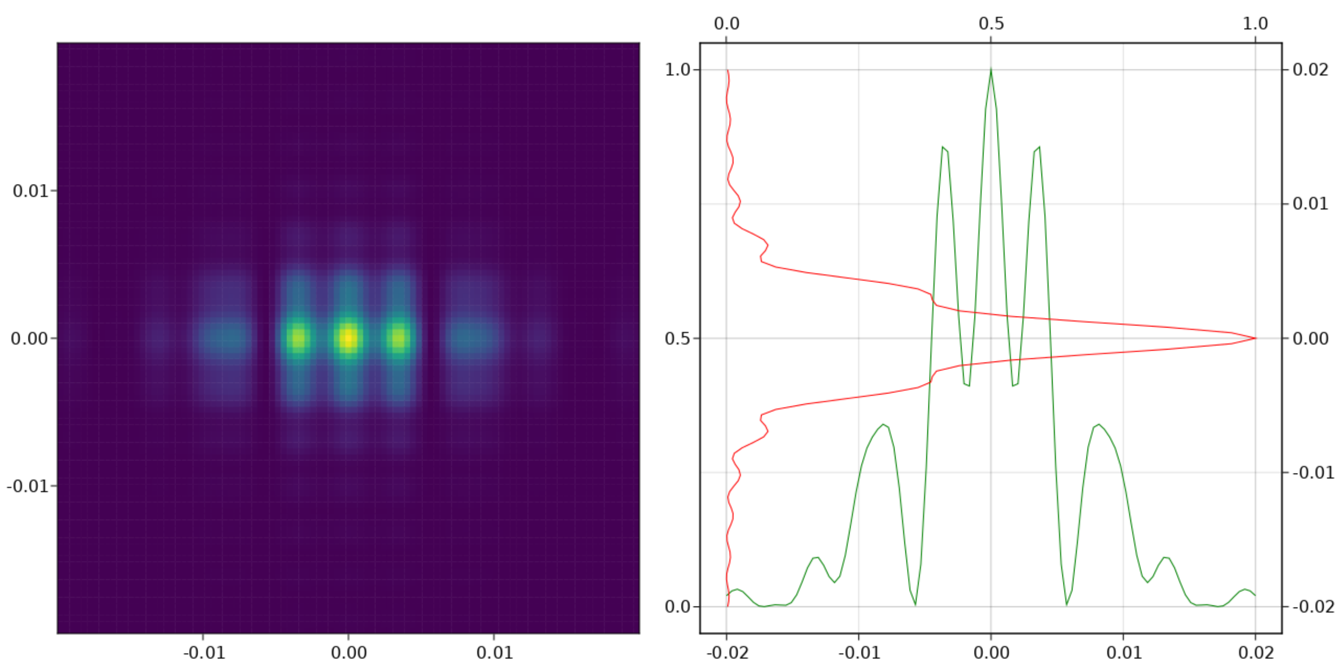
```
md"""### Babinet's principle

[Babinet's Principle](https://en.wikipedia.org/wiki/Babinet%27s_principle)

Show diffraction for inverted aperture.
```

```
Consider that the used aperture is square and usually of very limited length: Major
artifacts will stem from that fact.
"""
```

```
using Setfield
```

plot_inverse_diffraction (generic function with 1 method)

```
function plot_inverse_diffraction()
    setup = default_setup
    aperture = setup.aperture
    setup = @set setup.aperture = (x, y; kwargs...) -> (invertshape(aperture(x, y;
kwargs...)))
    screen, coords = calculate_integral(setup);
    fig = CM.Figure(resolution=(1300, 650))
    CM.Axis(fig[1,1])
    CM.heatmap!(coords, coords, (screen))
    axh = CM.Axis(fig[1, 2])
    axv = CM.Axis(fig[1,2], xaxisposition=:top, yaxisposition=:right)
    mid = div(size(screen)[1], 2)
    CM.lines!(axh, coords, view(screen, :, mid)/maximum(view(screen, :, mid)),
        color=:green)
    CM.lines!(axv, view(screen, mid, :)/maximum(view(screen, mid, :)), coords,
        color=:red)
    fig
end
```




```
plot_inverse_diffraction()
```

Plane Wave diffraction

In the limit of far distance of the point source to the screen, we get plane waves.

By directly assuming plane waves, we can calculate faster. D different setup and scan configurations are used here, too. However, they are linked to the configuration above by default to allow a 1:1 comparison of point source images to plane wave images.

```
md"""### Plane Wave diffraction

In the limit of far distance of the point source to the screen, we get plane waves.

By directly assuming plane waves, we can calculate faster. D different setup and scan
configurations are used here, too. However, they are linked to the configuration above
by default to allow a 1:1 comparison of point source images to plane wave images.
"""
```

calculate_integral_planewave (generic function with 1 method)

```
function calculate_integral_planewave(setup::Setup;
        scan_param::ScanParam=plane_scan_param)

    screendim = convert(Int, div(scan_param.screen_size, scan_param.screen))
    apdim = convert(Int, div(scan_param.aperture_size, scan_param.aperture))
    screen = zeros(Complex{Float64}, screendim, screendim)

    delta = scan_param.aperture_size / apdim

    all_aperture_coords = ((x,y)
        for x = LinRange(
                -scan_param.aperture_size/2, scan_param.aperture_size/2, apdim)
        for y = LinRange(
                -scan_param.aperture_size/2, scan_param.aperture_size/2, apdim))

    all_screen_coords = ((i, x, j, y)
        for (i, x) = enumerate(LinRange(
                -scan_param.screen_size/2, scan_param.screen_size/2, screendim))
        for (j, y) = enumerate(LinRange(
                -scan_param.screen_size/2, scan_param.screen_size/2, screendim)))
    all_screen_coords = collect(all_screen_coords)
    k = 2pi/setup.lambda

    count = 0
    @time begin for (apx, apy) = all_aperture_coords
        if setup.aperture(apx, apy; setup.aperture_kwargs...) < 0.1
            continue
        end
        count += length(all_screen_coords)
        alpha = 0
        Threads.@threads for (i, scx, j, scy) = all_screen_coords
            pointdist = sqrt((apx-scx)^2 + (apy-scy)^2)
            r = sqrt(pointdist^2 + setup.screen_pos^2)
            beta = atan(pointdist / abs(setup.screen_pos))
            term = exp(-im*(k*r))/(r)
            K = 1/(im*setup.lambda) * (fastcos(alpha)+fastcos(beta))/2

            screen[i, j] += K * term * delta^2
        end
    end
    end
    println("calculate_integral_planewave: $count iterations")
    (abs.(screen).^2,
        LinRange(-scan_param.screen_size/2, scan_param.screen_size/2, screendim))
end
```

plot_diffraction_planewave (generic function with 1 method)

```
function plot_diffraction_planewave()
    screen, coords = calculate_integral_planewave(plane_setup);
    fig = CM.Figure(resolution=(1300, 650))
    CM.Axis(fig[1,1])
    CM.heatmap!(coords, coords, (screen))
```

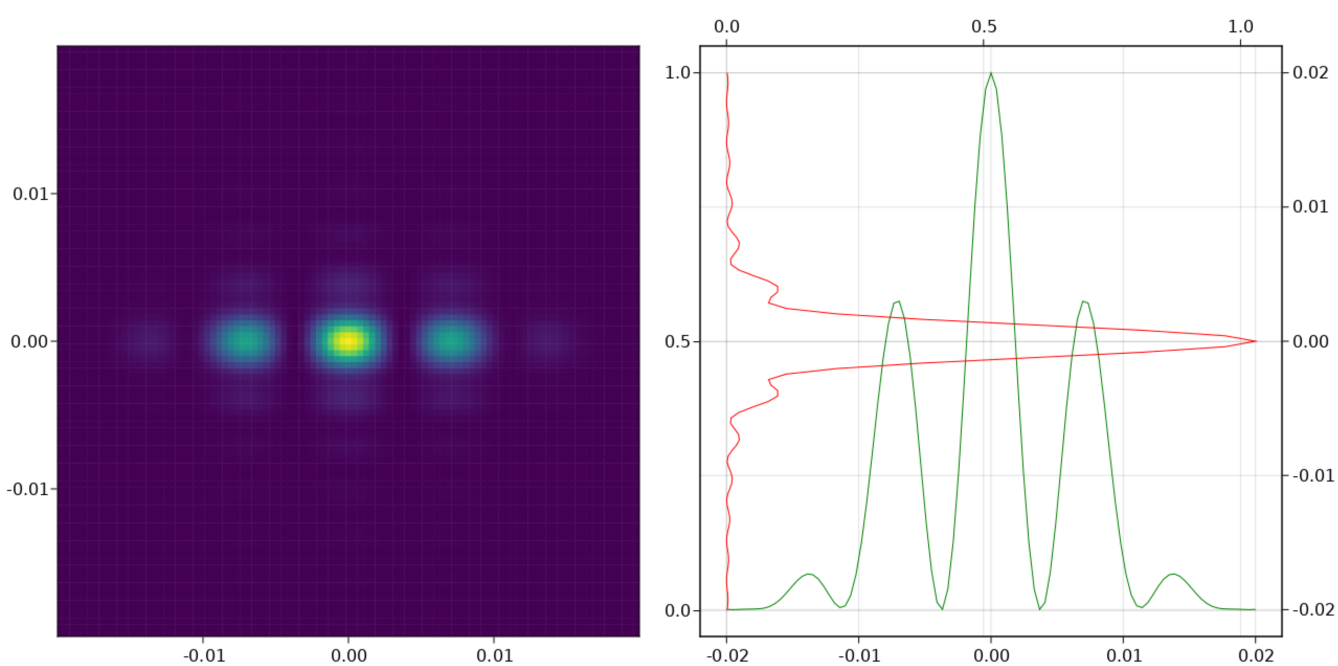
```
    axh = CM.Axis(fig[1, 2])
    axv = CM.Axis(fig[1,2], xaxisposition=:top, yaxisposition=:right)
    mid = div(size(screen)[1], 2)
    CM.lines!(axh, coords, view(screen, :, mid)/maximum(view(screen, :, mid)),
        color=:green)
    CM.lines!(axv, view(screen, mid, :)/maximum(view(screen, :, mid)), coords,
        color=:red)
    fig
end
```

```
Setup([Source(0.0, -0.0, -30.0)], double_slit (generic function with 1 method), Dict()
```

```
begin
    # Distance of source, shape function, shape parameters,
    # distance to screen, wavelength
    plane_setup = Setup([Source(z=-80)], double_slit, Dict(), 20, 500e-9)
    plane_setup = default_setup
end
```

```
ScanParam(0.005, 5.0e-5, 0.04, 0.0004)
```

```
begin
    plane_scan_param = ScanParam(
    # Aperture scan size and scan step, in m, depends on aperture function
    0.005, .005/100,
    # Screen size and scan step in m.
    .025, .025/100)

    # Override: use identical plane as above
    plane_scan_param = default_scan_param
end
```




```
plot_diffraction_planewave()
```