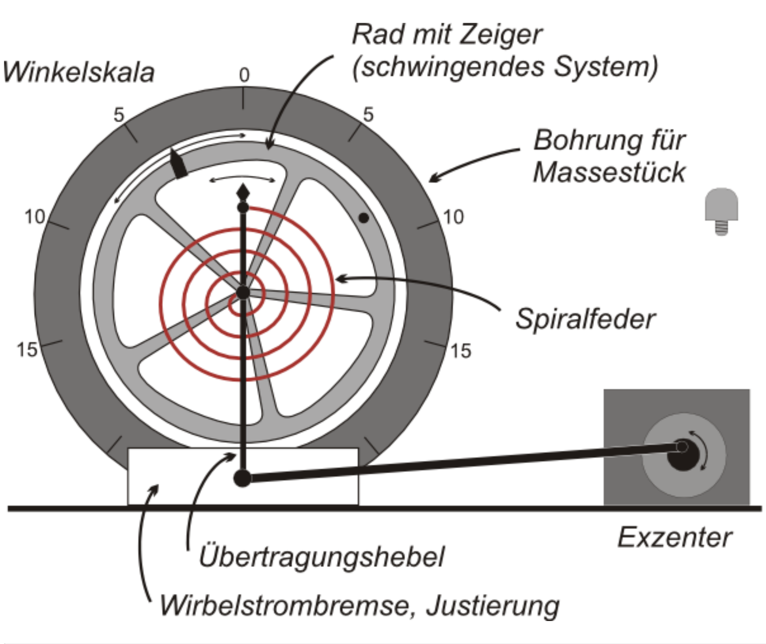
Computerphysik Programmiertutorial 8b

Prof. Dr. Matteo Rizzi und Dr. Markus Schmitt - Institut für Theoretische Physik, Universität zu Köln

Github: https://github.com/markusschmitt/compphys2022

Inhalt dieses Notebooks: Virtuelles Pohl'sches Rad: Einbinden von Code aus externen Dateien, Diskrete Fouriertransformation, Fehlerfortpflanzung, least-squares Fits

Ein virtuelles Experiment



In [1]:
```
using Plots, LaTeXStrings

include("pohl_experiment.jl")

using .PohlExperiment
```

In [2]:
```
?run_experiment
```

```
search: run_experiment
```

Out[2]:
```
run_experiment(t_max::Real, phi_init::Real; damping::Int64=0, driving::Bool=false, Omega_F::Real=0.0)
```

Diese Funktion simuliert ein Experiment mit einem Pohl'schen Rad, das durch folgende DGL beschrieben wird:

$\varphi'' + \gamma\varphi' + \Omega_0\varphi = Fsin(\Omega_F t)$

Die Dämpfung kann auf drei Stufen eingestellt werden. Der äußere Antrieb kann ein- oder ausgeschaltet werden und die Frequenz des Antriebs kann frei gewählt werden.

Die jeweiligen Parameter γ, Ω_0 und A werden zufällig gewählt.

Als Ergebnis des Experiments gibt die Funktion die zeitabhängige Beobachtung der Amplitude φ zurück, die leicht fehlerbehaftet ist.

Arguments

- `t_max::Real` : Laufzeit des Experiments. Start bei $t = 0$, Ende bei $t = t_{max}$.
- `phi_init::Real` : Anfangswert für die Auslenkung φ.
- `damping::Int64` : Einstellung der Dämpfung. `0` : keine Dämpfung, `1` : schwache Dämfpung, `2` : starke Dämpfung.
- `driving::Bool` : Antrieb ein/aus.
- `Omega_F::Real` : Antriebsfrequenz Ω_F.

Returns

Ein `DataFrame` mit zwei Spalten: eine für die Zeit t und eine für die Auslenkung $\varphi(t)$

Ein virtuelles Experiment ausführen

In [3]:
```
t_max = 90
phi_0 = pi/2

data = run_experiment(t_max, phi_0)
```

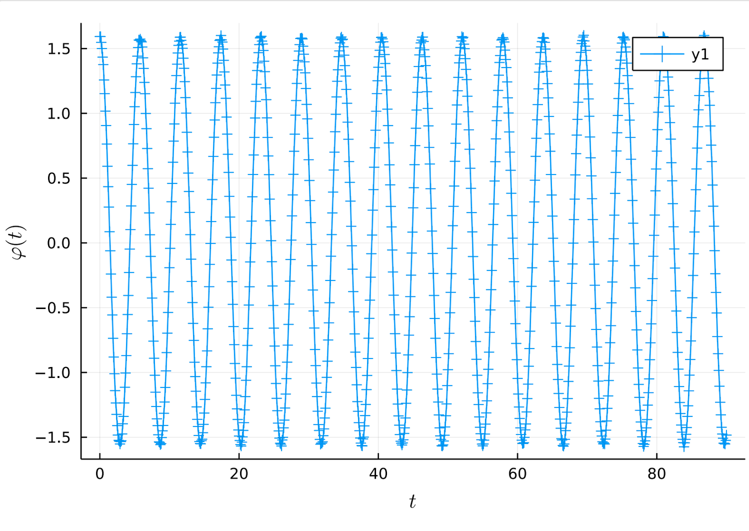
Out[3]: 901 rows × 2 columns

	t	phi
	Float64	Float64
1	0.0	1.59307
2	0.1	1.59288
3	0.2	1.54762
4	0.3	1.50021
5	0.4	1.43372
6	0.5	1.37642
7	0.6	1.2574
8	0.7	1.15198
9	0.8	1.01728
10	0.9	0.905786
11	1.0	0.762952
12	1.1	0.591743
13	1.2	0.434204
14	1.3	0.276598
15	1.4	0.0862293
16	1.5	-0.0608106
17	1.6	-0.240393
18	1.7	-0.41932
19	1.8	-0.556164
20	1.9	-0.736349
21	2.0	-0.880878
22	2.1	-1.00987
23	2.2	-1.14011
24	2.3	-1.2354
25	2.4	-1.32362
26	2.5	-1.41034
27	2.6	-1.48314
28	2.7	-1.52411
29	2.8	-1.53142
30	2.9	-1.5519
⋮	⋮	⋮

Daten inspizieren:

In [4]:
```
plot(data.t,data.phi,marker=:+)

xlabel!(L"t")
ylabel!(L"$\varphi(t)$")
```

Out[4]:



Eigenfrequenz bestimmen

Fouriertransformation des Schwingungssignals:

Die diskrete Fouriertransformation eines Signals $a = (a_1, \dots, a_N)$ ergibt die Fourierkoeffizienten

$$\hat{a}_k = \sum_{j=1}^{N} e^{-2\pi i \frac{jk}{N}} a_j$$

Der Index k ist also der Frequenz $\omega_k = \frac{2\pi k}{N}$ zugeordnet. Für ein reelles Signal gilt $\hat{a}_k = (\hat{a}_{N-k})^*$.

Diskrete Fouriertransformation ist im Julia Paket `FFTW` implementiert (siehe hier). Fouriertransformation eines reellen Signals wird mit der Funktion `rfft` ausgeführt:

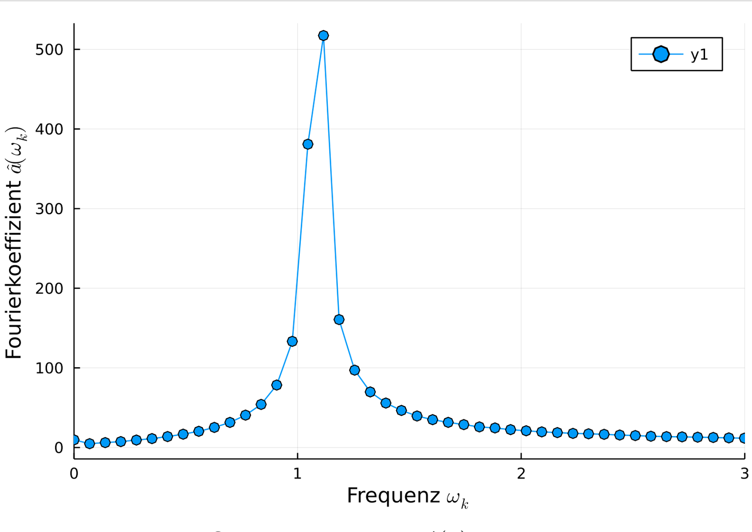
In [5]:
```
using FFTW

phi_omega = rfft(data.phi)
```

Out[5]:
```
451-element Vector{ComplexF64}:
   9.597386008850554 + 0.0im
  -4.475572182400781 + 1.8033911767273176im
  -4.663395844954867 + 4.0293088204513055im
  -4.734971192381479 + 5.710651050681589im
  -4.918014992483439 + 7.960009016341804im
  -4.858135548991763 + 10.066039093669076im
  -5.450914331020927 + 12.606429841851483im
  -5.754818739332955 + 15.723548799937522im
  -6.681111689850186 + 19.22233091299221im
  -7.498904778960503 + 24.079871113541856im
  -8.765090255810433 + 30.293265209750338im
 -10.669482733425088 + 39.227099112929835im
 -13.343620466013924 + 52.4497356381539255im
                     ⋮
   1.522378325930143 - 0.2823778332953697im
  1.4137451346296397 - 0.06730577298239182im
   1.635161659175551 + 0.13367796358951428im
  1.6626874901574311 - 0.29886263855529044im
   1.723057848250952 - 0.07611799428437305im
  1.5401997528180456 + 0.06806140501924035im
    1.48321757690724 + 0.17741601180411382im
  1.3714574518978373 + 0.1198646774110651im
  1.4862181390034659 - 0.06358346558926853im
  1.2548714878451217 + 0.1549754501401333im
  1.4345405620789529 - 0.06399901223773558im
  1.4649202431566641 + 0.21105317163272597im
```

Spektrum plotten:

In [6]:
```
# Frequenzen berechnen
dt = data.t[2]-data.t[1]
k_values = collect(1:length(phi_omega)).-1
omega_k = k_values*2*pi/length(data.phi)/dt

plot(omega_k, abs.(phi_omega), marker=:o)

xlabel!(L"Frequenz $\omega_k$")
ylabel!(L"Fourierkoeffizient $\hat a(\omega_k)$");
xlims!(0,3)
```

Out[6]:



Bestimme den Parameter Ω_0 als den Wert, bei dem $A(\omega)$ maximal ist.

In [7]:
```
maxIdx = argmax(abs.(phi_omega))
println("Omega_0 = ", omega_k[maxIdx])
```

```
Omega_0 = 1.1157709757477623
```

Dieses Ergebnis ist fehlerbehaftet. Das können wir in Julia berücksichtigen und das Paket `Measurements` verwenden (siehe hier). `Measurements` führt einen Datentyp für fehlerbehaftete Zahlen ein und ermöglicht damit sehr einfach die Fehlerfortpflanzung zu berechnen.

Da wir unser Ergebnis für Ω_0 durch Ablesen des Maximums erhalten haben, ist es sinnvoll als Fehler die Auflösung auf der Frequenz-Achse anzugeben. Dafür nutzen wir `measurement` aus dem `Measurements` Paket:

In [8]:
```
using Measurements

Omega_0 = measurement(omega_k[maxIdx], omega_k[maxIdx+1]-omega_k[maxIdx])
```

Out[8]: 1.116 ± 0.07

Beispiel zur Fehlerfortpflanzung:

Typischerweise kann der Hersteller des Pohl'schen Rades das zugehörige Trägheitsmoment Θ angeben. In unserem Modul `PohlExperiment` haben wir eine solche Angabe mit eingebaut, inklusive einer Fehlerangabe:

In [9]:
```
PohlExperiment.Theta
```

Out[9]: 13.0 ± 0.5

Mit dieser Angabe und unserem Ergebnis für Ω_0 können wir die "Federkonstante" k des Rades ausrechnen:

$$k = \Omega_0^2 \Theta$$

Wenn wir mit `measurement` s rechnen, wird die zugehörige Fehlerfortpflanzung automatisch mitberechnet:

In [10]:
```
println("k = ", Omega_0^2 * PohlExperiment.Theta)
```

```
k = 16.2 ± 2.1
```

Analyse inklusive Dämpfung und Antrieb

Die maximale Schwingungsamplitude des Pohl'sche Rades mit Dämpfung und Antrieb ist im Anschluss an die Einschwingphase gegeben durch

$$\varphi_{max}(\Omega_F) = \frac{F}{\sqrt{(\Omega_F^2 - \Omega_0^2)^2 + \gamma^2 \Omega_0^2}}$$

Wir werden jetzt eine Reihe virtueller Experimente mit verschiedenen Ω_F durchführen um $\varphi_{max}(\Omega_F)$ zu messen. Anschließend werden wir eine Funktion an die Daten fitten um die Parameter γ und F zu bestimmen.

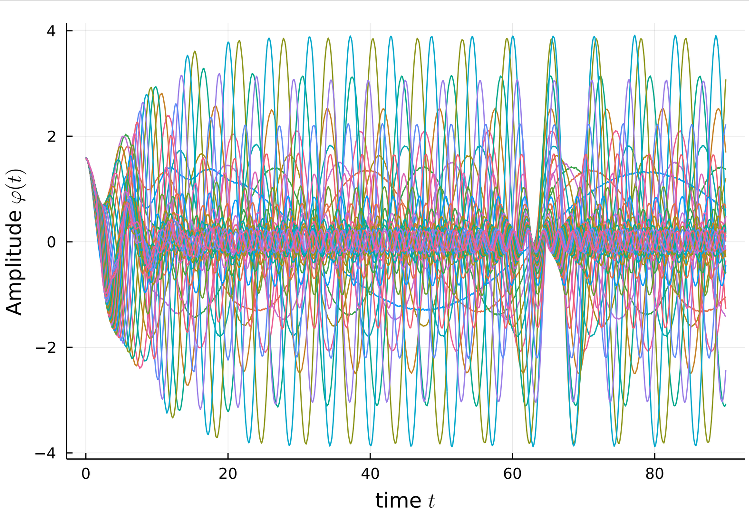
Wir beginnen also mit der Datenaufnahme:

In [11]:
```
Omega_F_list = collect(0.1:0.1:pi)

dataList = [run_experiment(t_max, phi_0; damping=1, driving=true, Omega_F=w) for w in Omega_F_list];
```

Daten visualisieren:

In [12]:
```
plot()
for d in dataList
    plot!(d.t, d.phi, label="")
end

xlabel!(L"time $t$")
ylabel!(L"Amplitude $\varphi(t)$")
```

Out[12]:



Nun wollen wir für jede Antriebsfrequenz die konstante Schwingungsamplitude nach der Einschwingphase bestimmen. Dafür definieren wir zunächst eine Funktion, die aus einem Array von Werten die lokalen Maxima heraussucht:

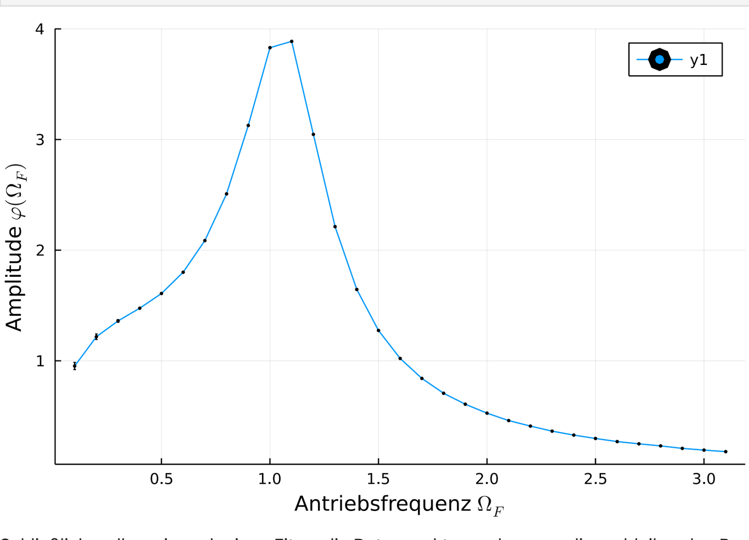
In [13]:
```
function get_absolute_extrema(a)

    a_abs = abs.(a) # Wir wollen den Betrag betrachten

    # In dieser Liste werden wir die Extremwerte sammeln
    extrema = []

    # Schleife über Array-Einträge
    for i in 2:length(a_abs)-1

        # Testen ob der aktuelle eintrag ein Maximum ist
        if a_abs[i]-a_abs[i-1] > 0 && a_abs[i+1] - a_abs[i] < 0
            # Maxima der Liste hinzufügen
            push!(extrema, a_abs[i])
        end
    end

    return extrema

end
```

Out[13]:
```
get_absolute_extrema (generic function with 1 method)
```

Nun können wir die Extrema unserer "gemessenen" Daten suchen und so ein Ergebnis für $\varphi(\Omega_F)$ bekommen:

In [14]:
```
using Statistics

amplitude = []
amplitude_error = []

I0 = Int(50 / dt) # Start Index: ignoriere Einschwingzeit

for d in dataList

    A = get_absolute_extrema(d.phi[I0:end])

    push!(amplitude, mean(A))                        # Mittelwert der Amplitude
    push!(amplitude_error, std(A)/sqrt(length(A)))   # Standardfehler des Mittelwerts

end

# Plot
plot(Omega_F_list, amplitude, yerror=amplitude_error, marker=:o, markersize=1)
xlabel!(L"Antriebsfrequenz $\Omega_F$")
ylabel!(L"Amplitude $\varphi(\Omega_F)$")
```

Out[14]:



Schließlich wollen wir noch einen Fit an die Datenpunkte machen, um die verbleibenden Parameter zu extrahieren. Dafür verwenden wir das Paket `LsqFit` (siehe hier).

In [15]:
```
using LsqFit
```

Damit können wir leicht "least squares" Fits durchführen. Zunächst müssen wir die Modellfunktion (den Ansatz) definieren:

In [16]:
```
Omega_0_val = Omega_0.val # Wert unseres Ergebnisses für Omega_0

function model(omega, p)

    global Omega_0_val

    gamma = p[1]
    F0 = p[2]

    return @. F0 / sqrt((omega^2 - Omega_0_val^2)^2 + gamma^2 * omega^2)
end
```

Out[16]:
```
model (generic function with 1 method)
```

Dann können wir den Fit durchführen:

In [17]:
```
initial_p = [0.5, 0.7]

fit = curve_fit(model, Omega_F_list, amplitude, initial_p)
```

Out[17]:
```
LsqFit.LsqFitResult{Vector{Float64}, Vector{Float64}, Matrix{Float64}, Vector{Any}}([0.3696218404650772, 1.5832438902742634], [0.3284056848472795, 0.09403949
55421362, 0.003848279113709996, -0.029717798291272413, -0.04397042627631964, -0.06515086789333013, -0.10276019879105025, -0.15759069936131365, -0.23582455412
038827, -0.2594938263050093  …  0.019346946198531334, 0.01871046598483589, 0.015320086644111497, 0.012974355738505194, 0.012794360739316946, 0.00860086656666
8586, 0.005998209241467128, 0.009960775994493742, 0.009179764610397656, 0.008325368454960513], [-0.0031029831869349217 0.8093902937796108; -0.013305050626905
172 0.828356017335419; … ; -0.010954911102735386 0.12764984764704113; -0.0093435669184002 0.1184389700888159], true, Any[])
```

Ergebnis: Die Variable `fit` hat nun ein Feld `param` , das das Ergebnis für die gefitteten Parameter enthält. Außerdem gibt die Funktion `estimate_covar` die Kovarianzmatrix aus, woraus sich die Fehlerabschätzung ergibt. Damit können wir das Ergebnis wieder als `measurement` s behandeln:

In [18]:
```
function get_error(fit, i)
    return sqrt(estimate_covar(fit)[i,i])
end

errors = [get_error(fit, i) for i in 1:2]

gamma_fit = measurement(fit.param[1], errors[1])
F_fit = measurement(fit.param[2], errors[2])

println("gamma = ", gamma_fit)
println("F = ", F_fit)
```

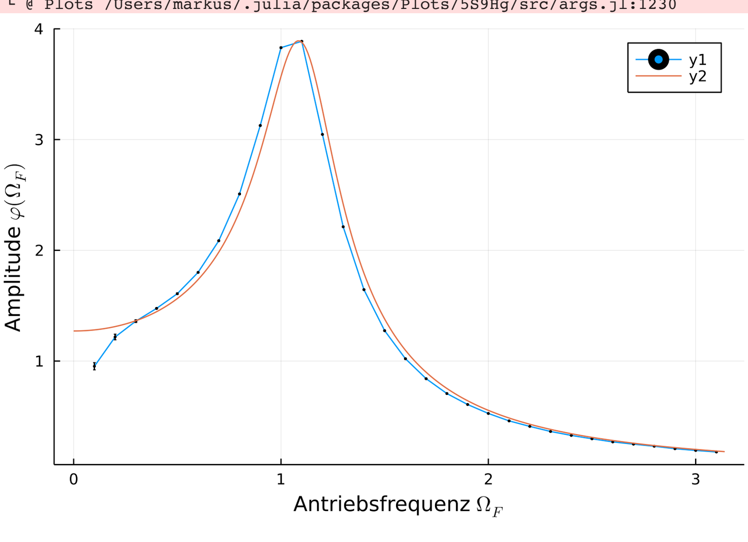
```
gamma = 0.37 ± 0.014
F = 1.583 ± 0.036
```

Plot:

In [19]:
```
plot(Omega_F_list, amplitude, yerror=amplitude_error, marker=".", markersize=1)
plot!(collect(0:0.01:pi), model(collect(0:0.01:pi), [gamma_fit.val, F_fit.val]))
xlabel!(L"Antriebsfrequenz $\Omega_F$")
ylabel!(L"Amplitude $\varphi(\Omega_F)$")
```

```
┌ Warning: Skipped marker arg ..
└ @ Plots /Users/markus/.julia/packages/Plots/5S9Hg/src/args.jl:1230
```

Out[19]:



Ergebnisse

In [20]:
```
println("Exact:")
println("Omega_0 = ", PohlExperiment.Omega_0)
println("gamma = ", PohlExperiment.gamma[2])
println("F = ", PohlExperiment.F)
println("")
println("Unsere Auswertung:")
println("Omega_0 = ", Omega_0)
println("gamma = ", gamma_fit)
println("F = ", F_fit)
```

```
Exact:
Omega_0 = 1.1785089118864922
gamma = 0.35565412315612527
F = 1.5242794505793964

Unsere Auswertung:
Omega_0 = 1.116 ± 0.07
gamma = 0.37 ± 0.014
F = 1.583 ± 0.036
```