RB-SFA: High Harmonic Generation in the Strong Field Approximation via *Mathematica*

Introduction

Readme

RB-SFA is a compact and flexible *Mathematica* package for calculating High Harmonic Generation emission within the Strong Field Approximation. It combines *Mathematica*'s analytical integration capabilities with its numerical calculation capacities to offer a fast and user-friendly plug-and-play solver for calculating harmonic spectra and other properties. In addition, it can calculate first-order nondipole corrections to the SFA results to evaluate the effect of the driving laser's magnetic field on harmonic spectra.

The name RB-SFA comes from its first application (as Rotating Bicircular High Harmonic Generation in the Strong field Approximation) but the code is general so RB-SFA just stands for itself now. This first application was used to calculate the polarization properties of the harmonics produced by multi-colour circularly polarized fields, as reported in the paper

Spin conservation in high-order-harmonic generation using bicircular fields. E. Pisanty, S. Sukiasyan and M. Ivanov. *Phys. Rev. A* **90**, 043829 (2014), arXiv:1404.6242.

This code is dual-licensed under the GPL and CC-BY-SA licenses. If you use this code or its results in an academic publication, please cite the paper above or the GitHub repository where the latest version will always be available. An example citation is

E. Pisanty. RB-SFA: High Harmonic Generation in the Strong Field Approximation via *Mathematica*. https://github.-com/episanty/RB-SFA (2014).

This software consists of this notebook, which contains the code and its documentation, a corresponding auto-generated package file. This notebook also contains a Usage and Examples section which explains how to use the code and documents the calculations used in the original publication.

Specifications

This code calculates the harmonic dipole as per the original Lewenstein *et al.* paper,

M. Lewenstein, Ph. Balcou, M.Yu. Ivanov, A. L'Huillier and P.B. Corkum. Theory of high-harmonic generation by low-frequency laser fields. *Phys. Rev. A* **49** no. 3, 2117 (1994).

The time-dependent dipole is given by

$$\boldsymbol{D}(t) = i \int_0^t dt' \int d^3\boldsymbol{p}\, \boldsymbol{d}(\boldsymbol{\pi}(\boldsymbol{p}, t)) \times \boldsymbol{F}(t') \cdot \boldsymbol{d}(\boldsymbol{\pi}(\boldsymbol{p}, t')) \times \exp[-i\, S(\boldsymbol{p}, t, t')] + c.c.$$

Here $\boldsymbol{F}(t)$is the time-dependent applied electric field, with vector potential $\boldsymbol{A}(t)$; the electron charge is taken to be −1. The integration time t' can be interpreted as the ionization time and the integration limit t represents the recollision time. (Upon applying the saddle-point approximation to the temporal integrals these notions apply exactly to the

corresponding quantum orbits.) The ionization and recollision events are governed by the corresponding dipole matrix elements, which are taken for a hydrogenic 1s-type orbital with characteristic momentum κ and ionization potential $I_p = \frac{1}{2}\kappa^2$:

$$\boldsymbol{d}(\boldsymbol{p}) = \left\langle \boldsymbol{p} \middle| \hat{\boldsymbol{d}} \middle| g \right\rangle = \frac{8i}{\pi} \frac{\sqrt{2\kappa^5}\,\boldsymbol{p}}{(\boldsymbol{p}^2+\kappa^2)^3}.$$

(Taken from B. Podolsky & L. Pauling. *Phys. Rev.* **34** no. 1, 109 (1929).) The displaced momentum $\boldsymbol{\pi}(\boldsymbol{p}, t) = \boldsymbol{p} + \boldsymbol{A}(t)$ is used to aid in the generalization to nondipole cases.

The action

$$S(\boldsymbol{p}, t, t') = \int_{t'}^{t} \left(\frac{1}{2}\kappa^2 + \frac{1}{2}\boldsymbol{\pi}(\boldsymbol{p}, \tau)^2 \right) d\tau$$

is the governing factor of the integral. It is highly oscillatory and must be dealt with carefully. The momentum integral is approximated using the saddle point method; this is valid and straightforward since there is always one unique saddle point,

$$\boldsymbol{p}_s(t, t') = -\frac{\int_{t'}^{t} \boldsymbol{A}(t'')\,dt''}{t-t'},$$

as long as the action's dependence on the momentum is quadratic. After the momentum has been integrated via the saddle point method, the time-dependent dipole is given by

$$\boldsymbol{D}(t) = i \int_0^t dt' \left(\frac{2\pi}{\epsilon + i(t-t')} \right)^{3/2} \boldsymbol{d}(\boldsymbol{\pi}(\boldsymbol{p}_s(t, t'), t)) \times \boldsymbol{F}(t') \cdot \boldsymbol{d}(\boldsymbol{\pi}(\boldsymbol{p}_s(t, t'), t')) \times \exp[-i\, S(\boldsymbol{p}_s(t, t'), t, t')] + c.c.$$

A regularization factor ϵ has been introduced to prevent divergence of the prefactor as t' approaches t, representing the fact that at $t = t'$ the $\boldsymbol{p}$ integral no longer has a quadratic exponent.

The integral is performed with respect to the excursion time $\tau = t - t'$ for simplicity. To reduce integration time, a gating function gate[$\omega\,\tau$] has been introduced, and the integration time has been cut short to a controllable length. This is physically reasonable as the wavepacket spreading (which scales as $\tau^{-3/2}$) makes the contributions after the gate negligible.
More physically, supressing the contributions from large excursion times represents the fact that the extra-long trajectories they represent produce harmonics which are much harder to phase match (as their harmonic phase is very sensitively dependent on many factors) which means that they are rarely present in experimental spectra unless a concerted effort is performed to observe them.

First-order nondipole calculations

This code can also be used to calculate the first-order effect of nondipole terms, which become relevant at high intensities or long wavelengths, when the electron's speed from the laser-driven oscillations (which scales as F/ω) becomes comparable to the speed of light (equal to $c = 1/\alpha \approx 137$ in atomic units). (More specifically, the nondipole effects on the harmonic generation become relevant when the magnetic pushing per cycle, which offsets the returning wavepacket by $F^2/c\omega^3$, becomes comparable with the wavepacket spread upon recollision.)

This code implements a generalization of the theory presented in various forms in the papers N.J. Kylstra et al. *J. Phys B: At. Mol. Opt. Phys.* **34** no. 3, L55 (2001); N..J. Kylstra et al. *Laser Phys.* **12** no. 2, 409 (2002); M.W. Walser et al. *Phys. Rev. Lett.* **85** no. 24, 5082 (2000); V. Averbukh et al. *Phys Rev. A* **65,** 063402 (2002); and C.C. Chirilă et al. *Phys. Rev. A* **66,** 063411 (2002).

The nondipole-HHG code calculates the harmonic dipole of exactly the same form as the standard SFA,

$$\boldsymbol{D}(t) = i \int_0^t dt' \int d^3\boldsymbol{p}\, \boldsymbol{d}(\boldsymbol{\pi}(\boldsymbol{p}, t, t')) \times \boldsymbol{F}(t') \cdot \boldsymbol{d}(\boldsymbol{\pi}(\boldsymbol{p}, t', t')) \times \exp[-i\, S(\boldsymbol{p}, t, t')] + c.c.$$

with $S(\boldsymbol{p}, t, t') = \int_{t'}^t dt'' \left(\frac{1}{2}\kappa^2 + \frac{1}{2}\boldsymbol{\pi}(\boldsymbol{p}, t'', t')^2\right)$ as before, but with a nondipole term in the displaced momentum, which now reads

$$\boldsymbol{\pi}(\boldsymbol{p}, t, t') = \boldsymbol{p} + \boldsymbol{A}(t) - \int_{t'}^t \nabla \boldsymbol{A}(\tau) \cdot (\boldsymbol{p} + \boldsymbol{A}(\tau))\, d\tau,$$

or in component notation

$$\pi_j(\boldsymbol{p}, t, t') = p_j + A_j(t) - \int_{t'}^t \partial_j A_k(\tau)\, (p_k + A_k(\tau))\, d\tau.$$

Implementation: supporting functions

Initialization

```
BeginPackage["RBSFA`"];
```

Version number and timestamp

The command RBSFAversion prints the version of the RB-SFA package currently loaded and its timestamp

```
RBSFAversion::usage = "RBSFAversion[] prints the
    current version of the RB-SFA package in use and its timestamp.";
Begin["`Private`"];
RBSFAversion[] = "RB-SFA v2.0.5, Wed 30 Mar 2016 15:58:24";
End[];
```

The timestamp is updated every time the notebook is saved via an appropriate notebook option, which is set by the code below.

```
SetOptions[
  EvaluationNotebook[],
  NotebookEventActions → {{"MenuCommand", "Save"} :> (
      NotebookWrite[
       Cells[CellTags → "version-definition"][[1]],
       Cell[BoxData[RowBox[{
            "RBSFAversion::usage=\"RBSFAversion[] prints the current version of
               the RB-SFA package in use and its timestamp.\";
               Begin[\"`Private`\"];
               RBSFAversion[]=\"" <> First[StringSplit[
               Select[
                 Flatten[
                  NotebookRead[Cells[CellTags → "version-definition"][[1]]]
                    /. {Cell → List, BoxData → List, RowBox → List}
                 ],
                 Quiet[StringContainsQ[#, "RB-SFA v"]] &
                ][[1]], {"\"", ","}]] <> ", " <> DateString[] <> "\";
               End[];"
           }]], "Input", InitializationCell → True, CellTags → "version-definition"]
       , None
      ];
      NotebookSave[]
     ), PassEventsDown → True}
 ];
```

Dipole transition matrix elements

Default DTME is for a hydrogenic 1s state.

```
hydrogenicDTME::usage =
  "hydrogenicDTME[p,κ] returns the dipole transition matrix element for
    a 1s hydrogenic state of ionization potential Ip=1/2 κ^2.";
hydrogenicDTMERegularized::usage = "hydrogenicDTMERegularized[p,κ] returns
   the dipole transition matrix element for a 1s hydrogenic state of
   ionization potential Ip=1/2 κ^2, regularized to remove the denominator
   of 1/(p^2+κ^2)^3, where the saddle-point solutions are singular."
Begin["`Private`"];
hydrogenicDTME[p_List, κ_] := 8 I/π Sqrt[2 κ^5] p/(Total[p^2] + κ^2)^3
hydrogenicDTME[p_?NumberQ, κ_] := 8 I/π Sqrt[2 κ^5] p/(p^2 + κ^2)^3
hydrogenicDTMERegularized[p_List, κ_] := 8 I/π Sqrt[2 κ^5] p/1
hydrogenicDTMERegularized[p_?NumberQ, κ_] := 8 I/π Sqrt[2 κ^5] p/1
End[];
```

```
hydrogenicDTMERegularized[p,κ] returns the dipole transition matrix element for
  a 1s hydrogenic state of ionization potential Ip=1/2 κ^2, regularized to remove
  the denominator of 1/(p^2+κ^2)^3, where the saddle-point solutions are singular.
```

Another possibility is for a gaussian orbital

```
gaussianDTME::usage = "gaussianDTME[p,κ] returns the dipole transition
    matrix element for a gaussian state of characteristic size 1/κ.";
Begin["`Private`"];
gaussianDTME[p_List, κ_] := -I (4 π)^(3/4) κ^(-7/2) p Exp[-Total[p^2]/(2 κ^2)]
gaussianDTME[p_?NumberQ, κ_] := -I (4 π)^(3/4) κ^(-7/2) p Exp[-p^2/(2 κ^2)]
End[];
```

Various field envelopes

flatTopEnvelope

```
flatTopEnvelope::usage =
  "flatTopEnvelope[ω,num,nRamp] returns a Function object representing
    a flat-top envelope at carrier frequency ω lasting a total
    of num cycles and with linear ramps nRamp cycles long.";
Begin["`Private`"];
flatTopEnvelope[ω_, num_, nRamp_] := Function[t,
  Piecewise[{{0, t < 0}, {Sin[ω t / (4 nRamp)]^2, 0 ≤ t < 2 π / ω nRamp},
    {1, 2 π / ω nRamp ≤ t < 2 π / ω (num - nRamp)},
    {Sin[ω (2 π / ω num - t) / (4 nRamp)]^2, 2 π / ω (num - nRamp) ≤ t < 2 π / ω num},
    {0, 2 π / ω num ≤ t}}]]
End[];
```

cosPowerFlatTop

```
cosPowerFlatTop::usage =
  "cosPowerFlatTop[ω,num,power] returns a Function object representing
    a smooth flat-top envelope of the form 1-Cos(ω t/2 num)^power";
Begin["`Private`"];
cosPowerFlatTop[ω_, num_, power_] := Function[t, 1 - Cos[ω t / (2 num)]^power]
End[];
```

Field duration standard options

The standard options for the duration of the pulse and the resolution are

```
PointsPerCycle::usage =
  "PointsPerCycle is a sampling option which specifies the number of sampling
    points per cycle to be used in integrations.";
TotalCycles::usage = "TotalCycles is a sampling option which specifies
    the total number of periods to be integrated over.";
CarrierFrequency::usage = "CarrierFrequency is a sampling option
    which specifies the carrier frequency to be used.";
Protect[PointsPerCycle, TotalCycles, CarrierFrequency];

standardOptions = {PointsPerCycle → 90, TotalCycles → 1, CarrierFrequency → 0.057};
```

PointsPerCycle dictates how many sampling points are used per laser cycle (at frequency CarrierFrequency, of the infrared laser), and it should be at least twice the highest harmonic of interest. The total duration is TotalCycles cycles. CarrierFrequency is the frequency of the fundamental laser, in atomic units.

harmonicOrderAxis

harmonicOrderAxis produces a list that can be used as a harmonic order axis for the given pulse parameters.

The length can be fine-tuned (to match exactly a spectrum, for instance, and get a matrix of the correct shape) using the correction option, or a TargetLength can be directly specified.

```
harmonicOrderAxis::usage =
  "harmonicOrderAxis[opt→value] returns a list of frequencies which can be
    used as a frequency axis for Fourier transforms, scaled in units of
    harmonic order, for the provided field duration and sampling options.";
TargetLength::usage = "TargetLength is an option for harmonicOrderAxis
    which specifies the total length required of the resulting list.";
LengthCorrection::usage = "LengthCorrection is an option for harmonicOrderAxis
    which allows for manual correction of the length of the resulting list.";
Protect[LengthCorrection, TargetLength];
Begin["`Private`"];
Options[harmonicOrderAxis] =
  standardOptions ~ Join ~ {TargetLength → Automatic, LengthCorrection → 1};
harmonicOrderAxis::target =
  "Invalid TargetLength option `1`. This must be a positive integer or Automatic.";
harmonicOrderAxis[OptionsPattern[]] :=
 Module[{num = OptionValue[TotalCycles], npp = OptionValue[PointsPerCycle]},
  Piecewise[{
    {1/num Range[0., Round[(npp num + 1)/2.] - 1 + OptionValue[LengthCorrection]],
     OptionValue[TargetLength] === Automatic},
    {Round[(npp num+1)/2.]/num Range[0, OptionValue[TargetLength] - 1]/OptionValue[TargetLength],
     IntegerQ[OptionValue[TargetLength]] && OptionValue[TargetLength] ≥ 0}
   },
   Message[harmonicOrderAxis::target, OptionValue["TargetLength"]]; Abort[]
  ]
 ]
End[];
```

frequencyAxis

frequencyAxis produces a list that can be used as a harmonic order axis for the given pulse parameters. Identical to harmonicOrderAxis but produces a frequency axis (in atomic units) instead.

```
frequencyAxis::usage =
  "frequencyAxis[opt→value] returns a list of frequencies which can be used
    as a frequency axis for Fourier transforms, in atomic units of
    frequency, for the provided field duration and sampling options.";
Begin["`Private`"];
Options[frequencyAxis] = Options[harmonicOrderAxis];
frequencyAxis[options : OptionsPattern[]] :=
 OptionValue[CarrierFrequency] harmonicOrderAxis[options]
End[];
```

timeAxis

timeAxis produces a list that can be used as a time axis for the given pulse parameters.

```
Quit
```

```
timeAxis::usage =
  "timeAxis[opt→value] returns a list of times which can be used as a time axis ";
TimeScale::usage = "TimeScale is an option for timeAxis which specifies the units
    the list should use: AtomicUnits by default, or LaserPeriods if required.";
AtomicUnits::usage = "AtomicUnits is a value for the option TimeScale of timeAxis
    which specifies that the times should be in atomic units of time.";
LaserPeriods::usage = "LaserPeriods is a value for the option TimeScale of timeAxis
    which specifies that the times should be in multiples of the carrier laser period.";
Protect[TimeScale, AtomicUnits, LaserPeriods];
Begin["`Private`"];
Options[timeAxis] =
  standardOptions~Join~{TimeScale → AtomicUnits, PointNumberCorrection → 0};
timeAxis::scale =
  "Invalid TimeScale option `1`. Available values are AtomicUnits and LaserPeriods";
timeAxis[OptionsPattern[]] := Block[{T = 2 π / ω, ω = OptionValue[CarrierFrequency],
   num = OptionValue[TotalCycles], npp = OptionValue[PointsPerCycle]},
  Piecewise[{
     {1, OptionValue[TimeScale] === AtomicUnits},
     {1/T, OptionValue[TimeScale] === LaserPeriods}
    },
    Message[timeAxis::scale, OptionValue[TimeScale]]; Abort[]
   ] × Table[t
    , {t, 0, num 2 π/ω, num/(num × npp + OptionValue[PointNumberCorrection]) 2 π/ω}
   ]
 ]
End[];


tInit = 0;
tFinal = 2 π/ω num;
δt = (tFinal - tInit)/(num × npp + OptionValue[PointNumberCorrection]);(*integration and looping timestep*)
```

getSpectrum

getSpectrum takes a time-dependent dipole list and returns its Fourier transform in absolute-value-squared. It takes as options

- pulse parameters ω, TotalCycles and PointsPerCycle,
- a polarization parameter ϵ, which gives an unpolarized spectrum when given False, or polarizes along an ellipticity vector ϵ (this is meant primarily to select right- and left-circularly polarized spectra using $\epsilon = \{1, i\}$ and $\epsilon = \{1, -i\}$ respectively),
- a DifferentiationOrder, which can return the dipole value (default, = 0), velocity (= 1), or acceleration (= 2),
- a power of ω, ωPower, with which to multiply the spectrum before returning it (which should be equivalent to DifferentiationOrder except for pathological cases), and
- a ComplexPart function to apply immediately after differentiation (default is the identity function, but Re, Im, or Abs[#]^2 & are reasonable choices).

If no option is passed to ωPower and DifferentiationOrder, the pulse parameters do not really affect the output,

except by a global factor of TotalCycles.

```
getSpectrum::usage = "getSpectrum[DipoleList] returns the power spectrum of DipoleList.";
Polarization::usage =
  "Polarization is an option for getSpectrum which specifies a polarization
    vector along which to polarize the dipole list. The default,
    Polarization→False, specifies an unpolarized spectrum.";
ComplexPart::usage = "part is an option for getSpectrum which specifies
    a function (like Re, Im, or by default #&) which should be
    applied to the dipole list before the spectrum is taken.";
ωPower::usage = "ωPower is an option for getSpectrum which specifies
    a power of frequency which should multiply the spectrum.";
DifferentiationOrder::usage = "DifferentiationOrder is an option for
    getSpectrum which specifies the order to which the dipole
    list should be differentiated before the spectrum is taken.";
Protect[Polarization, part, ωPower, DifferentiationOrder];

Begin["`Private`"];
Options[getSpectrum] = {Polarization → False, ComplexPart → (# &),
    ωPower → 0, DifferentiationOrder → 0} ~Join~ standardOptions;

getSpectrum::diffOrd = "Invalid differentiation order `1`.";
getSpectrum::ωPow = "Invalid ω power `1`.";

getSpectrum[dipoleList_, OptionsPattern[]] := Block[
  {polarizationVector, differentiatedList, depth, dimensions,
   num = OptionValue[TotalCycles],
   npp = OptionValue[PointsPerCycle], ω = OptionValue[CarrierFrequency], δt = 2 π / ω / npp
  },
  polarizationVector = OptionValue[Polarization] / Norm[OptionValue[Polarization]];

  differentiatedList = OptionValue[ComplexPart][Piecewise[{
      {dipoleList, OptionValue[DifferentiationOrder] == 0},
      {1/(2 δt) (Most[Most[dipoleList]] - Rest[Rest[dipoleList]]),
       OptionValue[DifferentiationOrder] == 1},
      {1/δt^2 (Most[Most[dipoleList]] - 2 Most[Rest[dipoleList]] + Rest[Rest[dipoleList]]),
       OptionValue[DifferentiationOrder] == 2}}],
     Message[getSpectrum::diffOrd, OptionValue[DifferentiationOrder]];
     Abort[]
    ];

  If[NumberQ[OptionValue[ωPower]], Null;, Message[getSpectrum::ωPow, OptionValue[ωPower]];
   Abort[] ];

  num Table[
```

```
    (ω/num k)^(2 OptionValue[ωPower]), {k, 1, Round[Length[differentiatedList]/2]}
  ] × If[
   OptionValue[Polarization] === False, (*unpolarized spectrum*)
   (*funky depth thing so this can take lists of numbers and lists of vectors,
   of arbitrary length. Makes for easier benchmarking.*)
   depth = Length[Dimensions[dipoleList]];
   dimensions = If[Length[#] > 1, #[[2]], 1(*#[[1]]*)] &[Dimensions[dipoleList]];
   Sum[Abs[
      Fourier[
        If[depth > 1, Re[differentiatedList[[All, i]]], Re[differentiatedList[[All]]]]
        , FourierParameters → {-1, 1}
       ][[1 ;; Round[Length[differentiatedList]/2]]]
     ]^2, {i, 1, dimensions}]]
   , (*polarized spectrum*)
   Abs[
     Transpose[Table[
         Fourier[
          Re[differentiatedList[[All, i]]]
          , FourierParameters → {-1, 1}
         ]
         , {i, 1, 2}]][[1 ;; Round[Length[differentiatedList]/2]]].polarizationVector
    ]^2
  ]
 ]
End[];
```

spectrumPlotter

spectrumPlotter takes a spectrum and a list of options and returns a plot of the spectrum. The available options are

· a FrequencyAxis option, which will give the harmonic order as a horizontal axis by default, and an arbitrary scale with any other option,

· all the options of harmonicOrderAxis, which will be passed to the call that makes the horizontal axis, and

· all the options of ListLinePlot, which will be used to format the plot.

```
spectrumPlotter::usage = "spectrumPlotter[spectrum] plots
    the given spectrum with an appropriate axis in a log10 scale.";
FrequencyAxis::usage = "FrequencyAxis is an option for spectrumPlotter
    which specifies the axis to use.";
Protect[FrequencyAxis];
Begin["`Private`"];
Options[spectrumPlotter] = Join[{FrequencyAxis → "HarmonicOrder"},
   Options[harmonicOrderAxis], Options[ListLinePlot]];
spectrumPlotter[spectrum_, options : OptionsPattern[]] := ListPlot[
  {Which[
     OptionValue[FrequencyAxis] === "HarmonicOrder",
     harmonicOrderAxis["TargetLength" → Length[spectrum], Sequence @@
       FilterRules[{options} ~Join~ Options[spectrumPlotter], Options[harmonicOrderAxis]]],
     OptionValue[FrequencyAxis] === "Frequency",
     frequencyAxis["TargetLength" → Length[spectrum], Sequence @@
       FilterRules[{options} ~Join~ Options[spectrumPlotter], Options[harmonicOrderAxis]]],
     True, Range[Length[spectrum]]
    ],
    Log[10, spectrum]
   }ᵀ
  , Sequence @@ FilterRules[{options}, Options[ListLinePlot]]
  , Joined → True
  , PlotRange → Full
  , PlotStyle → Thick
  , Frame → True
  , Axes → False
  , ImageSize → 800
 ]
End[];
```

biColorSpectrum

biColorSpectrum takes a time-dependent dipole list and produces overlaid plots of the right- and left-circular components of the spectrum, in red and blue respectively. It takes all the options of getSpectrum and spectrumPlotter, which are passed directly to the corresponding calls, as well as the options of Show, which can be used to modify the plot appearance.

```
Quit
```

```
biColorSpectrum::usage =
  "biColorSpectrum[DipoleList] produces a two-colour spectrum of DipoleList,
    separating the two circular polarizations.";
Begin["`Private`"];
Options[biColorSpectrum] = Join[{PlotRange → All}, Options[Show],
   Options[spectrumPlotter], DeleteCases[Options[getSpectrum], Polarization → False]];
biColorSpectrum[dipoleList_, options : OptionsPattern[]] := Show[{
   spectrumPlotter[
    getSpectrum[dipoleList, Polarization → {1, +i},
     Sequence @@ FilterRules[{options}, Options[getSpectrum]]],
    PlotStyle → Red, Sequence @@ FilterRules[{options}, Options[spectrumPlotter]]],
   spectrumPlotter[
    getSpectrum[dipoleList, Polarization → {1, -i},
     Sequence @@ FilterRules[{options}, Options[getSpectrum]]],
    PlotStyle → Blue, Sequence @@ FilterRules[{options}, Options[spectrumPlotter]]]
  }
  , PlotRange → OptionValue[PlotRange]
  , Sequence @@ FilterRules[{options}, Options[Show]]
 ]
End[];
```

Various gate functions

Gate functions are used to suppress the contributions of extra-long trajectories with long excursion times, partly to reflect the effect of phase matching but mostly to keep integration times reasonable. They are provided to the main numerical integrator makeDipoleList via its Gate option.

```
SineSquaredGate::usage =
  "SineSquaredGate[nGateRamp] specifies an integration gate with a sine-squared
    ramp, such that SineSquaredGate[nGateRamp][ωt,nGate]
    has nGate flat periods and nGateRamp ramp periods.";
LinearRampGate::usage = "LinearRampGate[nGateRamp] specifies an integration
    gate with a linear ramp, such that SineSquaredGate[nGateRamp][ωt,nGate]
    has nGate flat periods and nGateRamp ramp periods.";
Begin["`Private`"];
SineSquaredGate[nGateRamp_][ωτ_, nGate_] := Piecewise[{{1, ωτ ≤ 2 π (nGate - nGateRamp)},
   {Sin[(2 π nGate - ωτ)/(4 nGateRamp)]^2, 2 π (nGate - nGateRamp) < ωτ ≤ 2 π nGate}, {0, nGate < ωτ}}]
LinearRampGate[nGateRamp_][ωτ_, nGate_] := Piecewise[{{1, ωτ ≤ 2 π (nGate - nGateRamp)},
   {-(ωτ - 2 π (nGate + nGateRamp))/(2 π nGateRamp), 2 π (nGate - nGateRamp) < ωτ ≤ 2 π nGate}, {0, nGate < ωτ}}]
End[];
```

getIonizationPotential

```
getIonizationPotential::usage =
  "getIonizationPotential[Target] returns the ionization potential
    of an atomic target, e.g. \"Hydrogen\", in atomic units.

getIonizationPotential[Target,q] returns the ionization
    potential of the q-th ion of the specified Target, in atomic units.";
Begin["`Private`"];
getIonizationPotential[Target_, Charge_: 0] :=
 UnitConvert[ElementData[Target, "IonizationEnergies"][[Charge + 1]] /
   (Quantity[1, "AvogadroConstant"] Quantity[1, "Hartrees"])]
End[];
```

Implementation: main integrator function

The main integration function is makeDipoleList, and its basic syntax is of the form makeDipoleList[VectorPotential→A]. Here the vector potential A must be a function object, such that for numeric t the construct A[t] returns a list of numbers after the appropriate field parameters have been introduced: thus the criterion is that, for a call of the form makeDipoleList[VectorPotential→A, FieldParameters→pars], a call of the form A[t]//.pars returns a list of numbers for numeric t. To see the available options use Options[makeDipoleList], and to get information on each option use the ?VectorPotential construct.

```
makeDipoleList::usage = "makeDipoleList[VectorPotential→A]
    calculates the dipole response to the vector potential A.";

VectorPotential::usage =
  "VectorPotential is an option for makeDipole list which specifies the field's vector
    potential. Usage should be VectorPotential→A, where A[t]//.pars must yield a
    list of numbers for numeric t and parameters indicated by FieldParameters→pars.";
VectorPotentialGradient::usage = "VectorPotentialGradient is an option for makeDipole
    list which specifies the gradient of the field's vector potential. Usage should be
    VectorPotentialGradient→GA, where GA[t]//.pars must yield a square matrix of the
    same dimension as the vector potential for numeric t and parameters indicated by
    FieldParameters→pars. The indices must be such that GA[t][[i,j]] returns ∂ᵢAⱼ[t].";
FieldParameters::usage = "FieldParameters is an option for makeDipole list which ";
Preintegrals::usage =
  "Preintegrals is an option for makeDipole list which specifies whether the
    preintegrals of the vector potential should be \"Analytic\" or \"Numeric\".";
ReportingFunction::usage = "ReportingFunction is an option for makeDipole
    list which specifies a function used to report the results, either
    internally (by the default, Identity) or to an external file.";
Gate::usage = "Gate is an option for makeDipole list which specifies the
    integration gate to use. Usage as Gate→g, nGate→n will gate the integral
    at time ωt/ω by g[ωt,n]. The default is Gate→SineSquaredGate[1/2].";
nGate::usage = "nGate is an option for makeDipole list which specifies
    the total number of cycles in the integration gate.";
IonizationPotential::usage = "IonizationPotential is an option for makeDipoleList
    which specifies the ionization potential Iₚ of the target.";
Target::usage = "Target is an option for makeDipoleList which specifies
    chemical species producing the HHG emission, pulling the ionization
```

```
    potential from the Wolfram ElementData curated data set.";
DipoleTransitionMatrixElement::usage = "DipoleTransitionMatrixElement is an option
    for makeDipoleList which secifies a function f to use as the dipole transition
    matrix element, or a pair of functions {f_ion,f_rec} to be used separately for the
    ionization and recombination dipoels, to be used in the form f[p,κ]=f[p,√(2 I_p )].";
ϵCorrection::usage = "ϵCorrection is an option for makeDipoleList which specifies
    the regularization correction ϵ, i.e. as used in the factor 1/(t - tt + ⅈ ϵ)^(3/2).";
PointNumberCorrection::usage = "PointNumberCorrection is an option for
    makeDipoleList and timeAxis which specifies an extra number of points
    to be integrated over, which is useful to prevent Indeterminate errors
    when a Piecewise envelope is being differentiated at the boundaries.";
IntegrationPointsPerCycle::usage = "IntegrationPointsPerCycle is an option for
    makeDipoleList which controls the number of points per cycle to use for the
    integration. Set to Automatic, to follow PointsPerCycle, or to an integer.";
RunInParallel::usage = "RunInParallel is an option for makeDipoleList which
    controls whether each RB-SFA instance is parallelized. It accepts False
    as the (Automatic) option, True, to parallelize each instance, or a pair
    of functions {TableCommand, SumCommand} to use for the iteration and
    summing, which could be e.g. {Inactive[ParallelTable], Inactive[Sum]}.";


Protect[VectorPotential, VectorPotentialGradient,
  FieldParameters, Preintegrals, ReportingFunction, Gate, nGate,
  IonizationPotential, Target, ϵCorrection, PointNumberCorrection,
  DipoleTransitionMatrixElement, IntegrationPointsPerCycle, RunInParallel];



Begin["`Private`"];
Options[makeDipoleList] = standardOptions~Join~{
    VectorPotential → Automatic, FieldParameters → {}, VectorPotentialGradient → None,
    Preintegrals → "Analytic", ReportingFunction → Identity,
    Gate → SineSquaredGate[1 / 2], nGate → 3 / 2, ϵCorrection → 0.1,
    IonizationPotential → 0.5,
    Target → Automatic, DipoleTransitionMatrixElement → hydrogenicDTME,
    PointNumberCorrection → 0, Verbose → 0,
    RunInParallel → Automatic,
    IntegrationPointsPerCycle → Automatic
   };
makeDipoleList::gate =
  "The integration gate g provided as Gate→`1` is incorrect. Its usage as
    g[`2`,`3`] returns `4` and should return a number.";
makeDipoleList::pot = "The vector potential A provided as VectorPotential→`1`
    is incorrect or is missing FieldParameters. Its usage as
    A[`2`] returns `3` and should return a list of numbers.";
makeDipoleList::gradpot = "The vector potential GA provided as
    VectorPotentialGradient→`1` is incorrect or is missing FieldParameters.
    Its usage as GA[`2`] returns `3` and should return a square matrix
    of numbers. Alternatively, use VectorPotentialGradient→None.";
makeDipoleList::preint = "Wrong Preintegrals option `1`. Valid
```

```
    options are \"Analytic\" and \"Numeric\".";
makeDipoleList::runpar = "Wrong RunInParallel option `1`.";



makeDipoleList[OptionsPattern[]] := Block[
  {
   num = OptionValue[TotalCycles],
   npp = OptionValue[PointsPerCycle], ω = OptionValue[CarrierFrequency],
   dipoleRec, dipoleIon, κ,
   A, F, GA, pi, ps, S,
   gate, tGate, setPreintegral,
   tInit, tFinal, δt, δtint, ϵ = OptionValue[ϵCorrection],
   AInt, A2Int, GAInt, GAdotAInt, AdotGAInt, GAIntInt, bigPScorrectionInt, AdotGAdotAInt,
   integrand, dipoleList,
   TableCommand, SumCommand
  },

  A[t_] = OptionValue[VectorPotential][t] //. OptionValue[FieldParameters];
  F[t_] = -D[A[t], t];
  GA[t_] = If[
    TrueQ[OptionValue[VectorPotentialGradient] == None],
    Table[0, {Length[A[tInit]]}, {Length[A[tInit]]}],
    OptionValue[VectorPotentialGradient][t] //. OptionValue[FieldParameters]
   ];

  tInit = 0;
  tFinal = 2 π / ω num;
  (*looping timestep*)
  δt = (tFinal - tInit) / (num × npp + OptionValue[PointNumberCorrection]);
  (*integration timestep*)
  δtint = If[OptionValue[IntegrationPointsPerCycle] === Automatic, δt, (tFinal - tInit) /
     (num × OptionValue[IntegrationPointsPerCycle] + OptionValue[PointNumberCorrection])];

  tGate = OptionValue[nGate] 2 π / ω;
  (*Check potential and potential gradient for correctness.*)
  With[{ωtRandom = RandomReal[{ω tInit, ω tFinal}]},
   If[! And @@ (NumberQ /@ A[ωtRandom / ω]),
    Message[makeDipoleList::pot, OptionValue[VectorPotential], ωtRandom, A[ωtRandom]];
    Abort[]];
   If[! And @@ (NumberQ /@ Flatten[GA[ωtRandom / ω]]), Message[makeDipoleList::gradpot,
     OptionValue[VectorPotentialGradient], ωtRandom, GA[ωtRandom]];
    Abort[]];
  ];

  gate[ωτ_] := OptionValue[Gate][ωτ, OptionValue[nGate]];
  With[{ωtRandom = RandomReal[{ω tInit, ω tFinal}]},
```

```
 If[! TrueQ[NumberQ[gate[ωtRandom]]],
  Message[makeDipoleList::gate,
   OptionValue[Gate], ωtRandom, OptionValue[nGate], gate[ωtRandom]];
  Abort[]]
];

(*Target setup*)
Which[
 OptionValue[Target] === Automatic, κ = √(2 OptionValue[IonizationPotential]),
 True, κ = √(2 getIonizationPotential[OptionValue[Target]])
];
With[{dim = Length[A[RandomReal[{ω tInit, ω tFinal}]]]},
 (*Explicit conjugation of the
  recombination matrix element to keep the integrand analytic.*)
 Which[
   Head[OptionValue[DipoleTransitionMatrixElement]] === List,
   dipoleIon[{p1_, p2_, p3_}[[1 ;; dim]], κκ_] =
    First[OptionValue[DipoleTransitionMatrixElement]][{p1, p2, p3}, κκ];
   dipoleRec[{p1_, p2_, p3_}[[1 ;; dim]], κκ_] = Assuming[{{p1, p2, p3, κκ} ∈ Reals}, Simplify[
      Conjugate[Last[OptionValue[DipoleTransitionMatrixElement]][{p1, p2, p3}, κκ]]
     ]];
   , True,
   dipoleIon[{p1_, p2_, p3_}[[1 ;; dim]], κκ_] =
    OptionValue[DipoleTransitionMatrixElement][{p1, p2, p3}, κκ];
   dipoleRec[{p1_, p2_, p3_}[[1 ;; dim]], κκ_] = Assuming[{{p1, p2, p3, κκ} ∈ Reals}, Simplify[
      Conjugate[OptionValue[DipoleTransitionMatrixElement][{p1, p2, p3}, κκ]]
     ]];
  ];
];


setPreintegral[integralVariable_, preintegrand_,
  dimensions_, integrateWithoutGradient_, parametric_] := Which[
  OptionValue[VectorPotentialGradient] =!= None || TrueQ[integrateWithoutGradient],
  (*Vector potential gradient specified,
  or integral variable does not depend on it, so integrate*)
  Which[
    OptionValue[Preintegrals] == "Analytic",
    integralVariable[t_, tt_] =
      ((# /. {τ → t}) - (# /. {τ → tt})) &[Integrate[preintegrand[τ, tt], τ]];

    , OptionValue[Preintegrals] == "Numeric",
    Which[
      TrueQ[Not[parametric]],
      Block[{innerVariable},
        integralVariable[t_, tt_] = (innerVariable[t] - innerVariable[tt] /. First[
            NDSolve[{innerVariable'[τ] == preintegrand[τ],
              innerVariable[tInit] == ConstantArray[0, dimensions]},
             innerVariable, {τ, tInit, tFinal}, MaxStepSize → 0.25 / ω]
```

```
            ])
        ];
       , True,
       Block[{matrixpreintegrand, innerVariable, τpre},
        matrixpreintegrand[indices_, t_?NumericQ, tt_?NumericQ] :=
         preintegrand[t, tt][[## & @@ indices]];
        integralVariable[t_, tt_] = Array[(
            innerVariable[##][t - tt, tt] /. First@NDSolve[{
                D[innerVariable[##][τpre, tt], τpre] == Piecewise[
                  {{matrixpreintegrand[{##}, tt + τpre, tt], tt + τpre ≤ tFinal}}, 0],
                innerVariable[##][0, tt] == 0
               }, innerVariable[##]
               , {τpre, 0, tFinal - tInit}, {tt, tInit, tFinal}
               , MaxStepSize → 0.25 / ω
              ]
           ) &, dimensions];
       ]
      ];
    ];
   , OptionValue[VectorPotentialGradient] === None,
   (*Vector potential gradient has not been specified,
   and integral variable depends on it, so return appropriate zero matrix*)
   integralVariable[t_] = ConstantArray[0, dimensions];
   integralVariable[t_, tt_] = ConstantArray[0, dimensions];
  ];
Apply[setPreintegral,

  (
  AInt                A[#1] &                                  {Length[A[tInit]]}
  A2Int               A[#1].A[#1] &                            {}
  GAInt               GA[#1] &                                 {Length[A[tInit]], Length[A[tI
  GAdotAInt           GA[#1].A[#1] &                           {Length[A[tInit]]}
  AdotGAInt           A[#1].GA[#1] &                           {Length[A[tInit]]}
  GAIntInt            GAInt[#1, #2] &                          {Length[A[tInit]], Length[A[tI
  AdotGAdotAInt       A[#1].GAdotAInt[#1, #2] &                {}
  bigPScorrectionInt  GAdotAInt[#1, #2] + A[#1].GAInt[#1, #2] & {Length[A[tInit]]}

  ), {1}];

(*{∫_t0^t A(τ)dτ, ∫_t0^t A(τ)^2 dτ, ∫_t0^t ∇A(τ)dτ, ∫_t0^t ∇A(τ)·A(τ)dτ, ∫_t0^t A(τ)·∇A(τ)dτ, ∫_t'^t ∫_t'^τ ∇A(τ')dτ'dτ,
  ∫_t'^t A_k(τ)·∫_t'^τ ∂_k A_j(τ') A_j(τ')dτ'dτ, ∫_t'^t ∫_t'^τ (A_k(τ')∂_j A_k(τ') + A_k(τ)∂_k A_j(τ'))dτ'dτ};*)


(*Displaced momentum*)
pi[p_, t_, tt_] := p + A[t] - GAInt[t, tt].p - GAdotAInt[t, tt];
```

```
(*Stationary momentum and action*)
ps[t_, tt_] := ps[t, tt] = -1/(t - tt - i ϵ) Inverse[IdentityMatrix[Length[A[tInit]]] - 1/(t - tt - i ϵ)
        (GAIntInt[t, tt] + GAIntInt[t, tt]^T)].(AInt[t, tt] - bigPScorrectionInt[t, tt]);


S[t_, tt_] := 1/2 (Total[ps[t, tt]^2] + κ^2) (t - tt) + ps[t, tt].AInt[t, tt] + 1/2 A2Int[t, tt] - (
    ps[t, tt].GAIntInt[t, tt].ps[t, tt] +
     ps[t, tt].bigPScorrectionInt[t, tt] + AdotGAdotAInt[t, tt]
  );

integrand[t_, τ_] := i (2 π/(ϵ + i τ))^(3/2) dipoleRec[pi[ps[t, t - τ], t, t - τ], κ] ×
  dipoleIon[pi[ps[t, t - τ], t - τ, t - τ], κ].F[t - τ] Exp[-i S[t, t - τ]] gate[ω τ];

(*Debugging constructs. Verbose→
 1 prints information about the internal functions. Verbose→
  2 returns all the relevant internal functions and stops.*)
Which[
 OptionValue[Verbose] == 1, Information /@ {A, GA, ps, pi, S, AInt, A2Int,
   GAInt, GAdotAInt, AdotGAInt, GAIntInt, bigPScorrectionInt, AdotGAdotAInt},
 OptionValue[Verbose] == 2, Return[With[{t = Global`t,
    tt = Global`tt, p = Global`t, τ = Global`τ},
   {A[t], GA[t], ps[t, tt], pi[p, t, tt], S[t, tt], AInt[t], AInt[t, tt], A2Int[t],
    A2Int[t, tt], GAInt[t], GAInt[t, tt], GAdotAInt[t], GAdotAInt[t, tt], AdotGAInt[t],
    AdotGAInt[t, tt], GAIntInt[t], GAIntInt[t, tt], bigPScorrectionInt[t],
    bigPScorrectionInt[t, tt], AdotGAdotAInt[t], AdotGAdotAInt[t, tt], integrand[t, τ]}]]
];

(*Single-run parallelization*)
Which[
 OptionValue[RunInParallel] === Automatic ||
  OptionValue[RunInParallel] === False, TableCommand = Table;
 SumCommand = Sum;,
 OptionValue[RunInParallel] === True, TableCommand = ParallelTable;
 SumCommand = Sum;,
 True, TableCommand = OptionValue[RunInParallel][[1]];
 SumCommand = OptionValue[RunInParallel][[2]];
];

(*Numerical integration loop*)
dipoleList = TableCommand[
  OptionValue[ReportingFunction][
   δtint SumCommand[(
      integrand[t, τ]
     ),
     {τ, 0, If[OptionValue[Preintegrals] == "Analytic", tGate, Min[t - tInit, tGate]], δtint}]
  ]
  , {t, tInit, tFinal, δt}
 ];
```

```
  dipoleList

 ]
End[];
```

Package closure

End of package

```
EndPackage[]
```

Add to distributed contexts.

```
$DistributedContexts::overwrite =
  "Warning: overwriting previous value of $DistributedContexts.
    Reinstate your old definition, and include the RBSFA context
    to ensure proper parallelization of RB-SFA calculations.";
If[
 ValueQ[$DistributedContexts],
 Message[$DistributedContexts::overwrite]
]
$DistributedContexts := {$Context, "RBSFA`"}
```