

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

© Emilio Pisanty 2014-2016. Licensed under GPL and CC-BY-SA.

## ■ 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> (2016).

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.

## ■ Implementation

---

## Supporting functions

### Initialization

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

## Version number

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.";
RBSFAtimestamp::usage = "$RBSFAtimestamp prints the timestamp of
  the current version of the RB-SFA package.";
Begin["`Private`"];
RBSFAversion := "RB-SFA v2.1.3, "<> RBSFAtimestamp;
End[];
```

Old syntax (in functional form `RBSFAversion[]`), deprecated

```
RBSFAversion::usage = "RBSFAversion[] has been deprecated in favour of $RBSFAversion.";
RBSFAversion::dprc = "RBSFAversion[] has been deprecated in favour of $RBSFAversion.";
Begin["`Private`"];
RBSFAversion[] := (Message[RBSFAversion::dprc]; $RBSFAversion);
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-timestamp"][[1]],
      Cell[
        BoxData[RowBox[
          {"Begin[\"`Private`\"];\n$RBSFAtimestamp=\"\" <> DateString[] <> "\";\nEnd[];"}]]
        , "Input", InitializationCell → True, CellTags → "version-timestamp"
      ], None, AutoScroll → False];
    NotebookSave[]
  ), PassEventsDown → True}
];
```

To reset this behaviour to normal, evaluate the cell below

```
SetOptions[EvaluationNotebook[],
  NotebookEventActions → {{"MenuCommand", "Save"} ⇒ (NotebookSave[]), PassEventsDown → True}]
```

## Timestamp

```
Begin["`Private`"];
$RBSFAtimestamp = "Fri 9 Dec 2016 00:05:03";
End[];
```

## Directory

```
$RBSFAdirectory::usage =
  "$RBSFAdirectory is the directory where the current RB-SFA package instance is located.";
```

```
Begin["`Private`"];
With[{softLinkTestString = StringSplit[StringJoin[ReadList[
  "! ls -la " <> StringReplace[$InputFileName, {" " → "\\ "}], String]], " -> "]],
  If[Length[softLinkTestString] > 1, (*Testing in case $InputFileName
    is a soft link to the actual directory.*)
    $RBSFAdirectory = StringReplace[DirectoryName[softLinkTestString[[2]], {" " → "\\ "}],
    $RBSFAdirectory = StringReplace[DirectoryName[$InputFileName], {" " → "\\ "}],
  ]];
End[];
```

## Git commit hash and message

```
$RBSFACommit::usage = "$RBSFACommit returns the git
  commit log at the location of the RB-SFA package if there is one.";
$RBSFACommit::OS = "$RBSFACommit has only been tested on Linux.";
```

```
Begin["`Private`"];
$RBSFACommit := (If[$OperatingSystem ≠ "Unix", Message[$RBSFACommit::OS]];
  StringJoin[Riffle[ReadList["!cd " <> $RBSFAdirectory <> " && git log -1", String], {"\n"}]]];
End[];
```

## Function redefinitions

### ConstantArray

This redefines ConstantArray to take the corner case of an empty dimensions list, which returns an error code (and an unevaluated ConstantArray) for Mathematica versions under 10.1.0 (cf. [mma.se/q/133078](http://mma.se/q/133078)).

```
Quiet[Check[
  ConstantArray[0, {}];,
  Unprotect[ConstantArray];
  ConstantArray[Private`x_, {}] := Private`x;
  Protect[ConstantArray];
]];
```

Similarly, this needs to be put inside an initialization code for any parallelized subkernels that may get launched later (cf. [mm.se/q/131856](http://mm.se/q/131856)).

```
Parallelize;
Parallel`Developer`$InitCode = Hold[
  Quiet[Check[
    ConstantArray[0, {}];,
    Unprotect[ConstantArray];
    ConstantArray[Private`x_, {}] := Private`x;
    Protect[ConstantArray];
  ]];
];
```

## 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  $I_p = \frac{1}{2}\kappa^2$ .";
hydrogenicDTMERegularized::usage = "hydrogenicDTMERegularized[p,κ] returns
  the dipole transition matrix element for a 1s hydrogenic state of
  ionization potential  $I_p = \frac{1}{2}\kappa^2$ , regularized to remove the denominator
  of  $1/(p^2 + \kappa^2)^3$ , where the saddle-point solutions are singular."
Begin["`Private`"];
hydrogenicDTME[p_List, κ_] :=  $\frac{8i}{\pi} \frac{\sqrt{2\kappa^5} p}{(\text{Total}[p^2] + \kappa^2)^3}$ 
hydrogenicDTME[p_?NumberQ, κ_] :=  $\frac{8i}{\pi} \frac{\sqrt{2\kappa^5} p}{(p^2 + \kappa^2)^3}$ 
hydrogenicDTMERegularized[p_List, κ_] :=  $\frac{8i}{\pi} \frac{\sqrt{2\kappa^5} p}{1}$ 
hydrogenicDTMERegularized[p_?NumberQ, κ_] :=  $\frac{8i}{\pi} \frac{\sqrt{2\kappa^5} p}{1}$ 
End[];
```

hydrogenicDTMERegularized[p,κ] returns the dipole transition matrix element for a 1s hydrogenic state of ionization potential  $I_p = \frac{1}{2}\kappa^2$ , regularized to remove the denominator of  $1/(p^2 + \kappa^2)^3$ , 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[- $\frac{\text{Total}[p^2]}{2 \kappa^2}$ ]
gaussianDTME[p_?NumberQ, κ_] := -i (4 π)3/4 κ-7/2 p Exp[- $\frac{p^2}{2 \kappa^2}$ ]
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[ $\frac{\omega t}{4 nRamp}$ ]2, 0 ≤ t <  $\frac{2 \pi}{\omega} nRamp$ }, {1,  $\frac{2 \pi}{\omega} nRamp \leq t < \frac{2 \pi}{\omega} (num - nRamp)$ },
        {Sin[ $\frac{\omega (\frac{2 \pi}{\omega} num - t)}{4 nRamp}$ ]2,  $\frac{2 \pi}{\omega} (num - nRamp) \leq t < \frac{2 \pi}{\omega} num$ }, {0,  $\frac{2 \pi}{\omega} num \leq t$ }}]]
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[ $\frac{\omega t}{2 num}$ ]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, IntegrationPointsPerCycle → Automatic};

```

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[{
      { $\frac{1}{\text{num}}$  Range[0., Round[ $\frac{\text{npp num} + 1}{2.}$ ] - 1 + OptionValue[LengthCorrection]],
        OptionValue[TargetLength] === Automatic},
      { $\frac{\text{Round}[\frac{\text{npp num} + 1}{2.}]}{\text{num}}$  Range[0, OptionValue[TargetLength] - 1],
        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  $\frac{2\pi}{\omega}$ ,  $\frac{\text{num}}{\text{num} \times \text{npp} + \text{OptionValue[PointNumberCorrection]}} \frac{2\pi}{\omega}$ }
]
End[];
```

```
tInit = 0;
tFinal =  $\frac{2\pi}{\omega}$  num;
δt =  $\frac{\text{tFinal} - \text{tInit}}{\text{num} \times \text{npp} + \text{OptionValue[PointNumberCorrection]}}$ ; (*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  $\omega$ , TotalCycles and PointsPerCycle,
- a polarization parameter  $\epsilon$ , which gives an unpolarized spectrum when given False, or polarizes along an ellipticity vector  $\epsilon$  (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  $\omega$ ,  $\omega$ Power, 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[#]<sup>2</sup> & are reasonable choices).

If no option is passed to  $\omega$ Power 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 = "ComplexPart 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, ComplexPart, 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 =  $\frac{2 \pi / \omega}{npp}$ 
  },
  polarizationVector =  $\frac{\text{OptionValue[Polarization]}}{\text{Norm[OptionValue[Polarization] ]}}$ ;

  differentiatedList = OptionValue[ComplexPart][Piecewise[{
    {dipoleList, OptionValue[DifferentiationOrder] == 0},
    { $\frac{1}{2 \delta t}$  (Most[Most[dipoleList]] - Rest[Rest[dipoleList]]),
      OptionValue[DifferentiationOrder] == 1},
    { $\frac{1}{\delta t^2}$  (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[
  ( $\frac{\omega}{\text{num}} k$ )2 OptionValue[ωPower], {k, 1, Round[ $\frac{\text{Length}[\text{differentiatedList}]}{2}$ ]}
] × 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[ $\frac{\text{Length}[\text{differentiatedList}]}{2}$ ]]]]
  ]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]
]
, 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, +1},
      Sequence@@FilterRules[{options}, Options[getSpectrum]]],
    PlotStyle → Red, Sequence@@FilterRules[{options}, Options[spectrumPlotter]]],
  spectrumPlotter[
    getSpectrum[dipoleList, Polarization → {1, -1},
      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[ $\frac{2 \pi nGate - \omega \tau}{4 nGateRamp}$ ]2, 2 π (nGate - nGateRamp) < ωτ ≤ 2 π nGate}, {0, nGate < ωτ}}]
LinearRampGate[nGateRamp_][ωτ_, nGate_] := Piecewise[{{1, ωτ ≤ 2 π (nGate - nGateRamp)},
  {- $\frac{\omega \tau - 2 \pi (nGate + nGateRamp)}{2 \pi nGateRamp}$ , 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.

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"])]
getIonizationPotential[{Target_, Charge_: 0}] := getIonizationPotential[Target, Charge]
End[];
```

## makeDipoleList: main numerical integrator

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  $\partial_i A_j[t]$ .";
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  $\omega t/\omega$  by  $g[\omega t, n]$ . The default is Gate→SineSquaredGate[1/2].";
nGate::usage = "nGate is an option for makeDipoleList which specifies
the total number of cycles in the integration gate.";
IonizationPotential::usage = "IonizationPotential is an option for makeDipoleList
which specifies the ionization potential  $I_p$  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 specifies a function f to use as the dipole transition
matrix element, or a pair of functions {fion, frec} to be used separately for the
ionization and recombination dipoels, to be used in the form  $f[p, \kappa] = f[p, \sqrt{2 I_p}]$ .";
eCorrection::usage = "eCorrection is an option for makeDipoleList which specifies
the regularization correction  $\epsilon$ , i.e. as used in the factor  $\frac{1}{(t - tt + i\epsilon)^{3/2}}$ .";
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]}.";
Simplifier::usage = "Simplifier is an option for makeDipoleList which specifies a
function to use to simplify the intermediate and final analytical results.";
CheckNumericFields::usage = "CheckNumericFields is an option for makeDipoleList which
specifies whether to check for numeric values of A[t] and GA[t] for numeric t.";
QuadraticActionTerms::usage = "QuadraticActionTerms is an option for makeDipoleList
which specifies whether to use quadratic terms in  $\nabla A^2$  in the action.";

Protect[VectorPotential, VectorPotentialGradient, FieldParameters, Preintegrals,
ReportingFunction, Gate, nGate, IonizationPotential, Target, eCorrection,
PointNumberCorrection, DipoleTransitionMatrixElement, IntegrationPointsPerCycle,
RunInParallel, Simplifier, CheckNumericFields, QuadraticActionTerms];

Begin["`Private`"];
Options[makeDipoleList] = standardOptions~Join~{
VectorPotential → Automatic, FieldParameters → {}, VectorPotentialGradient → None,
Preintegrals → "Analytic", ReportingFunction → Identity,
Gate → SineSquaredGate[1 / 2], nGate → 3 / 2, eCorrection → 0.1,
IonizationPotential → 0.5,

```

```

    Target → Automatic, DipoleTransitionMatrixElement → hydrogenicDTME,
    PointNumberCorrection → 0, Verbose → 0, CheckNumericFields → True,
    RunInParallel → Automatic,
    Simplifier → Identity, QuadraticActionTerms → True
  };
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::carrfreq = "Non-numeric option CarrierFrequency `1`.";

makeDipoleList[OptionsPattern[]] := Block[
{
  num = OptionValue[TotalCycles], npp = OptionValue[PointsPerCycle],  $\omega$ ,
  dipoleRec, dipoleIon,  $\kappa$ ,
  A, F, GA, pi, ps, S,
  gate, tGate, setPreintegral,
  tInit, tFinal,  $\delta t$ ,  $\delta t_{int}$ ,  $\epsilon$  = OptionValue[ $\epsilon$ Correction],
  AInt, A2Int, GAInt, GAdotAInt, AdotGAInt, GAIntInt,
  PScorrectionInt, constCorrectionInt, GAIntdotGAIntInt, QuadMatrix, q,
  simplifier, prefactor, 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]
];

 $\omega$  = OptionValue[CarrierFrequency];
If[! NumberQ[ $\omega$ ] && TrueQ[OptionValue[CheckNumericFields]],
  Message[makeDipoleList::carrfreq,  $\omega$ ];
Abort[]];

```

```

tInit = 0;
tFinal =  $\frac{2\pi}{\omega}$  num;
(*looping timestep*)
 $\delta t = \frac{tFinal - tInit}{num \times npp + OptionValue[PointNumberCorrection]}$ ;
(*integration timestep*)
 $\delta tint = If[OptionValue[IntegrationPointsPerCycle] === Automatic, \delta t, (tFinal - tInit) /$ 
   $(num \times OptionValue[IntegrationPointsPerCycle] + OptionValue[PointNumberCorrection])]$ ;
tGate = OptionValue[nGate]  $\frac{2\pi}{\omega}$ ;
(*Check potential and potential gradient for correctness.*)
(*To do: change logic conditions to constructions on VectorQ[#,NumberQ]& and MatrixQ.*)
If[TrueQ[OptionValue[CheckNumericFields]],
  With[{ $\omega tRandom = RandomReal[\{\omega tInit, \omega tFinal\}]$ },
    If[! And @@ (NumberQ /@ A[ $\omega tRandom / \omega$ ]),
      Message[makeDipoleList::pot, OptionValue[VectorPotential],  $\omega tRandom$ , A[ $\omega tRandom$ ]];
      Abort[]];
    If[! And @@ (NumberQ /@ Flatten[GA[ $\omega tRandom / \omega$ ]]), Message[makeDipoleList::gradpot,
      OptionValue[VectorPotentialGradient],  $\omega tRandom$ , GA[ $\omega tRandom$ ]];
      Abort[]];
  ]];

gate[ $\omega \tau$ _] := OptionValue[Gate][ $\omega \tau$ , OptionValue[nGate]];
With[{ $\omega tRandom = RandomReal[\{\omega tInit, \omega tFinal\}]$ },
  If[! TrueQ[NumberQ[gate[ $\omega tRandom$ ]]],
    Message[makeDipoleList::gate,
      OptionValue[Gate],  $\omega tRandom$ , OptionValue[nGate], gate[ $\omega tRandom$ ]];
    Abort[]];
];

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

```



```

, True,
dipoleIon[{p1_, p2_, p3_}][1 ;; dim],  $\kappa\kappa$ _] =
  OptionValue[DipoleTransitionMatrixElement][{p1, p2, p3}[1 ;; dim],  $\kappa\kappa$ ];
dipoleRec[{p1_, p2_, p3_}[1 ;; dim],  $\kappa\kappa$ _] = Assuming[{p1, p2, p3,  $\kappa\kappa$ }  $\in$  Reals], Simplify[
  Conjugate[OptionValue[DipoleTransitionMatrixElement][{p1, p2, p3}[1 ;; dim],  $\kappa\kappa$ ]
]];
];
];

simplifier = OptionValue[Simplifier];
q = Boole[TrueQ[OptionValue[QuadraticActionTerms]]];

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_] =
      simplifier[(# /. { $\tau \rightarrow t$ }) - (# /. { $\tau \rightarrow tt$ })] & [Integrate[preintegrand[ $\tau$ , tt],  $\tau$ ]];

    , OptionValue[Preintegrals] == "Numeric",
    Which[
      TrueQ[Not[parametric]],
      Block[{innerVariable},
        integralVariable[t_, tt_] = (innerVariable[t] - innerVariable[tt] /. First[
          NDSolve[{innerVariable'[ $\tau$ ] == preintegrand[ $\tau$ ,
            innerVariable[tInit] == ConstantArray[0, dimensions]],
            innerVariable, { $\tau$ , tInit, tFinal}, MaxStepSize  $\rightarrow$  0.25 /  $\omega$ ]
        ])
      ];
      , True,
      Block[{matrixpreintegrand, innerVariable,  $\tau$ pre},
        matrixpreintegrand[indices_, t_?NumericQ, tt_?NumericQ] :=
          preintegrand[t, tt][[## & @@ indices]];
        integralVariable[t_, tt_] = Array[(
          innerVariable[##][t - tt, tt] /. First@NDSolve[{
            D[innerVariable[##][ $\tau$ pre, tt],  $\tau$ pre] == Piecewise[
              {{matrixpreintegrand[##, tt +  $\tau$ pre, tt], tt +  $\tau$ pre  $\leq$  tFinal}}, 0],
            innerVariable[##][0, tt] == 0
          }, innerVariable[##]
          , { $\tau$ pre, 0, tFinal - tInit}, {tt, tInit, tFinal}
          , MaxStepSize  $\rightarrow$  0.25 /  $\omega$ 
        ]
      ) &, 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] &
  A2Int          A[#1].A[#1] &
  GAInt          GA[#1] &
  GAdotAInt      GA[#1].A[#1] &
  AdotGAInt      A[#1].GA[#1] &
  GAIntInt       GAInt[#1, #2] &
  PScorrectionInt GAdotAInt[#1, #2] + A[#1].GAInt[#1, #2] - q GAInt[#1, #2]^T.GAdotAInt[#1, #2]
  GAIntdotGAIntInt q GAInt[#1, #2]^T.GAInt[#1, #2] &
  constCorrectionInt (A[#1] -  $\frac{q}{2}$  GAdotAInt[#1, #2]).GAdotAInt[#1, #2] &

```

```

  )
, {1}];

```

```

(*{ $\int_{t_0}^t A(\tau) d\tau$ ,  $\int_{t_0}^t A(\tau)^2 d\tau$ ,  $\int_{t_0}^t \nabla A(\tau) d\tau$ ,  $\int_{t_0}^t \nabla A(\tau) \cdot A(\tau) d\tau$ ,  $\int_{t_0}^t A(\tau) \cdot \nabla A(\tau) d\tau$ ,  $\int_t^t \int_{t'}^t \nabla A(\tau') d\tau' d\tau$ ,
 $\int_t^t \int_{t'}^t \partial_j A_k(\tau') A_k(\tau') d\tau' + A_k(\tau) \int_t^t \partial_k A_j(\tau') d\tau' - \int_t^t \partial_i A_j(\tau') d\tau' \int_t^t \partial_i A_k(\tau') A_k(\tau') d\tau' d\tau$ ,
 $\int_t^t \int_{t_0}^t \partial_i A_j(\tau') A_j(\tau') d\tau' \int_{t_0}^t \partial_i A_k(\tau') A_k(\tau') d\tau' d\tau$ ,
 $\int_t^t (A_k(\tau) - \frac{1}{2} \int_t^t \partial_k A_i(\tau') A_i(\tau') d\tau') \cdot \int_t^t \partial_k A_j(\tau') A_j(\tau') d\tau' d\tau$ };*)

```

```
(*Displaced momentum*)
```

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

```
(*Quadratic coefficient in nondipole action*)
```

```
QuadMatrix[t_, tt_] :=  $\frac{\text{GAIntInt}[t, tt] + \text{GAIntInt}[t, tt]^T}{2} - \frac{1}{2} \text{GAIntdotGAIntInt}[t, tt];$ 
```

```
(*Stationary momentum and action*)
```

```
ps[t_, tt_] := ps[t, tt] =
-  $\frac{1}{t - tt - i\epsilon}$  Inverse[IdentityMatrix[Length[A[tInit]]]] -  $\frac{1}{t - tt - i\epsilon}$  2 QuadMatrix[t, tt].
```

```

    (AInt[t, tt] - PScorrectionInt[t, tt]);
S[t_, tt_] := simplifier[
   $\frac{1}{2} (\text{Total}[\text{ps}[t, \text{tt}]^2] + \kappa^2) (t - \text{tt}) + \text{ps}[t, \text{tt}] \cdot \text{AInt}[t, \text{tt}] + \frac{1}{2} \text{A2Int}[t, \text{tt}] - ($ 
    ps[t, tt].QuadMatrix[t, tt].ps[t, tt] +
    ps[t, tt].PScorrectionInt[t, tt] + constCorrectionInt[t, tt]
  )
];

prefactor[t_,  $\tau$ _] :=  $\frac{2\pi}{\epsilon + i\tau}^{3/2}$  dipoleRec[pi[ps[t, t -  $\tau$ ], t, t -  $\tau$ ],  $\kappa$ ] ×
  dipoleIon[pi[ps[t, t -  $\tau$ ], t -  $\tau$ , t -  $\tau$ ],  $\kappa$ ].F[t -  $\tau$ ];
integrand[t_,  $\tau$ _] := prefactor[t,  $\tau$ ] Exp[-i S[t, t -  $\tau$ ]] gate[ $\omega \tau$ ];

(*Debugging constructs. Verbose→
  1 prints information about the internal functions. Verbose→2 returns all the relevant
  internal functions and stops. Verbose→3 for quantum-orbit constructs.*)
Which[
  OptionValue[Verbose] == 1, Information /@ {A, GA, ps, pi, S, AInt, A2Int, GAInt, GAdotAInt,
    AdotGAInt, GAIntInt, PScorrectionInt, constCorrectionInt, GAIntdotGAIntInt},
  OptionValue[Verbose] == 2, Return[With[{t = Symbol["t"], tt = Symbol["tt"],  $\tau$  = Symbol[" $\tau$ "],
    p = {Symbol["p1"], Symbol["p2"], Symbol["p3"]}]][1 ;; Length[A[ $\omega$  tInit]]],
    {A[t], GA[t], ps[t, tt], pi[p, t, tt], S[t, tt], AInt[t, tt], A2Int[t, tt],
      GAInt[t, tt], GAdotAInt[t, tt], AdotGAInt[t, tt], GAIntInt[t, tt],
      PScorrectionInt[t, tt], constCorrectionInt[t, tt],
      GAIntdotGAIntInt[t, tt], QuadMatrix[t, tt], integrand[t,  $\tau$ ]}]],
  OptionValue[Verbose] == 3,
  Return[{
    Function[Evaluate[prefactor[#1, #1 - #2]]], Function[Evaluate[S[#1, #2]]]
  }]
];

(*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 = Table[
  OptionValue[ReportingFunction][
     $\delta t$ int Sum[ (
```

```

        integrand[t,  $\tau$ ]
    ),
    { $\tau$ , 0, If[OptionValue[Preintegrals] == "Analytic", tGate, Min[t - tInit, tGate]],  $\delta t_{int}$ }]
]
, {t, tInit, tFinal,  $\delta t$ }
];
dipoleList

]
End[];

```

## Quantum orbit functions suite

### GetSaddlePoints

**GetSaddlePoints::usage =**

"GetSaddlePoints[ $\Omega$ , S, {tmin, tmax}, { $\tau$ min,  $\tau$ max}] finds a list of solutions {t,  $\tau$ } of the HHG temporal saddle-point equations at harmonic energy  $\Omega$  for action S, in the range {tmin, tmax} of recombination time and { $\tau$ min,  $\tau$ max} of excursion time, where both ranges should be the lower-left and upper-right corners of rectangles in the complex plane.

GetSaddlePoints[ $\Omega$ Range, S, {tmin, tmax}, { $\tau$ min,  $\tau$ max}] finds solutions of the HHG temporal saddle-point equations for a range of harmonic energies  $\Omega$ Range, and returns an Association with each harmonic energy  $\Omega$  indexing a list of saddle-point solution pairs {t,  $\tau$ }.

GetSaddlePoints[ $\Omega$ spec, S, {{tmin<sub>1</sub>, tmax<sub>1</sub>}, { $\tau$ min<sub>1</sub>,  $\tau$ max<sub>1</sub>}}, {{tmin<sub>2</sub>, tmax<sub>2</sub>}, { $\tau$ min<sub>2</sub>,  $\tau$ max<sub>2</sub>}}, ...] uses multiple time domains and combines the solutions.

GetSaddlePoints[ $\Omega$ spec, S, {{urange, vrange}, ...}, IndependentVariables→{u, v}] uses the explicit independent variables u and v to solve the equations and over the given ranges, where u and v can be any of "RecombinationTime", "IonizationTime" and "ExcursionTime", or their shorthands "t", "tt" and " $\tau$ " resp.;

SortingFunction::usage = "SortingFunction is an option of GetSaddlePoints which sets a function f, to be used as f[t,  $\tau$ , S,  $\Omega$ ], to be used to sort the solutions, or a list of such functions.";

SelectionFunction::usage = "SelectionFunction is an option of GetSaddlePoints that sets a function f, to be used as f[t,  $\tau$ , S,  $\Omega$ ], such that roots are only kept if f returns True.";

IndependentVariables::usage = "IndependentVariables is an option for GetSaddlePoints that specifies the two independent variables, out of "RecombinationTime", "IonizationTime" and "ExcursionTime" (or their shorthands "t", "tt" and " $\tau$ ", respectively), to be used in solving the saddle-point equations, and which range over the given regions.";

```

FiniteDifference::usage =
  "FiniteDifference is a value for the option Jacobian of FindRoot, FindComplexRoots,
  GetSaddlePoints, and related functions, which specifies that the Jacobian at
  each step should be evaluated using numerical finite difference procedures.";

GetSaddlePoints::error = "Errors encountered for harmonic energy  $\Omega = 1$ .";

Begin["`Private`"];
Options[GetSaddlePoints] =
  Join[{SortingFunction → (#2 &), SelectionFunction → (True &), IndependentVariables →
    {"RecombinationTime", "ExcursionTime"}}, Options[FindComplexRoots]];
Protect[SortingFunction, SelectionFunction, IndependentVariables, FiniteDifference];

GetSaddlePoints[ $\Omega$ spec_, S_, {tmin_, tmax_}, { $\tau$ min_,  $\tau$ max_}, options : OptionsPattern[]] :=
  GetSaddlePoints[ $\Omega$ spec, S, {{tmin, tmax}, { $\tau$ min,  $\tau$ max}}, options]

GetSaddlePoints[ $\Omega$ _, S_, timeRanges_, options : OptionsPattern[]] :=
  Block[{equations, roots, t = Symbol["t"], tt = Symbol["tt"],
     $\tau$  = Symbol[" $\tau$ "], indVars, depVar, depVarRule, tolerances},
    indVars = OptionValue[IndependentVariables] /.
      {"RecombinationTime" → "t", "ExcursionTime" → " $\tau$ ", "IonizationTime" → "tt"};
    depVar = First[DeleteCases[{"t", " $\tau$ ", "tt"}, Alternatives @@ indVars]];
    depVarRule = depVar /. {"tt" → {tt → t -  $\tau$ }, "t" → {t → tt +  $\tau$ }, " $\tau$ " → { $\tau$  → t - tt}};
    equations = {D[S[t, tt], t] ==  $\Omega$ , D[S[t, tt], tt] == 0} /. depVarRule;
    tolerances = Which[
      ListQ[OptionValue[Tolerance]], OptionValue[Tolerance],
      True, ConstantArray[
        Which[
          NumberQ[OptionValue[Tolerance]], OptionValue[Tolerance],
          True, 10^If[NumberQ[OptionValue[WorkingPrecision]],
            2 - OptionValue[WorkingPrecision], 2 - $MachinePrecision]
        ]
      , 2]];

SortBy[
  DeleteDuplicates[
    Flatten[Table[
      Select[
        Check[
          roots = ({t,  $\tau$ } /. depVarRule) /. (FindComplexRoots[
            equations
            , Evaluate[Sequence[{Symbol[indVars[[1]], range[[1, 1]],
              range[[1, 2]], {Symbol[indVars[[2]], range[[2, 1]], range[[2, 2]]}]]]
            , Evaluate[Sequence @@ FilterRules[{options}, Options[FindComplexRoots]]]
            , SeedGenerator → RandomSobolComplexes
            , Seeds → 50
          ] /. {} → (({t,  $\tau$ } /. depVarRule) → {})) (*to deal with empty results*)
        ]
      ]
    ]
  ]

```

```

    , Message[GetSaddlePoints::error,  $\Omega$ ]; roots
  ]
  ,
  Function[timePair, OptionValue[SelectionFunction][timePair[[1]], timePair[[2]], S,  $\Omega$ ]]
]
, {range, timeRanges}], 1]
, Function[{timePair1, timePair2},
  And@@Thread[Abs[timePair1 - timePair2] < tolerances] ]
]
, If[
  ListQ[OptionValue[SortingFunction]],
  Table[Function[timePair, f[timePair[[1]], timePair[[2]], S,  $\Omega$ ]],
    {f, OptionValue[SortingFunction]}],
  Function[timePair, OptionValue[SortingFunction][timePair[[1]], timePair[[2]], S,  $\Omega$ ]]
]
]
]
]
GetSaddlePoints[ $\Omega$ Range_List, S_, timeRanges_, options : OptionsPattern[]] :=
  Association[ParallelTable[
     $\Omega$  → GetSaddlePoints[ $\Omega$ , S, timeRanges, options]
    , { $\Omega$ , Sort[ $\Omega$ Range]}]]
End[];

```

## GetSaddlesFromSeeds

**GetSaddlesFromSeeds::usage =**  
 "GetSaddlesFromSeeds[{ $\{t_1, \tau_1\}, \{t_2, \tau_2\}, \dots\}$ ,  $\Omega$ , S] finds a list of solutions  $\{t, \tau\}$  of the HHG temporal saddle-point equations at harmonic energy  $\Omega$  for action S, using the given  $\{t_i, \tau_i\}$  as seeds for the process.

**GetSaddlesFromSeeds**[ $\langle \Omega_1 \rightarrow \{\{t_{11}, \tau_{11}\}, \{t_{12}, \tau_{12}\}, \dots\}, \Omega_2 \rightarrow \{\{t_{21}, \tau_{21}\}, \{t_{22}, \tau_{22}\}, \dots\}, \dots \rangle$ ,  $\Omega$ , S] finds solutions of the HHG temporal saddle-point equations, using the seeds list from the  $\Omega_i$  that's closest to  $\Omega$ , or as specified by the value of KeyChooserFunction.

**GetSaddlesFromSeeds**[seeds,  $\{\Omega_1, \Omega_2, \dots\}$ , S] iterates over the given set of harmonic energies."

**SeedsChooserFunction::usage =**  
 "SeedsChooserFunction is an option for GetSaddlesFromSeeds that specifies a function f (set by default to Nearest) that, when used as f[ $\{\Omega_1, \Omega_2, \dots\}, \Omega$ ], should return the indices  $\{\Omega_i, \Omega_j, \dots\}$  corresponding to the seed sets  $\{\{t_{i1}, \tau_{i1}\}, \dots\}, \{\{t_{j1}, \tau_{j1}\}, \dots\}$  to be used to solve the HHG saddle-point equations."

**RecalculateRoots::usage =** "RecalculateRoots is an option for GetSaddlesFromSeeds that specifies whether to re-solve the saddle-point equations if the given harmonic energy  $\Omega$  is among the set of keys of the given seeds association. The default is False, which is appropriate for S being the same action used to find the seeds, in which case setting RecalculateRoots→True will produce multiple FindRoot

```

errors. If using a different action than used to find the seeds, set to True.";

GetSaddlesFromSeeds::error = "Errors encountered for harmonic energy  $\Omega = \omega_1$ .";
GetSaddlesFromSeeds::norecalc =
  "Skipping re-calculation of roots at harmonic energy  $\omega_1$  since
  it is already in the key set of the given seeds association. To
  run the calculation for this case set RecalculateRoots to True.";

Begin["`Private`"];
Options[GetSaddlesFromSeeds] =
  Join[{RecalculateRoots → False, SeedsChooserFunction → Nearest}, Options[GetSaddlePoints]];
Protect[SeedsChooserFunction, RecalculateRoots];

GetSaddlesFromSeeds[seedsSpec_,  $\Omega$ Range_List, S_, options : OptionsPattern[]] :=
  Association[ParallelTable[
     $\Omega \rightarrow$  GetSaddlesFromSeeds[seedsSpec,  $\Omega$ , S, options]
    , { $\Omega$ , Sort[ $\Omega$ Range]}]]

GetSaddlesFromSeeds[seedsAssociation_Association,  $\Omega_?$ , S_, options : OptionsPattern[]] :=
  With[{keys = OptionValue[SeedsChooserFunction][Keys[seedsAssociation],  $\Omega$ ]},
    If[MemberQ[keys,  $\Omega$ ] && TrueQ[! OptionValue[RecalculateRoots]],
      Message[GetSaddlesFromSeeds::norecalc,  $\Omega$ ];
      Return[seedsAssociation[ $\Omega$ ]];
    GetSaddlesFromSeeds[Flatten[Values[seedsAssociation[Key /@ keys]], 1],  $\Omega$ , S, options]
  ]

GetSaddlesFromSeeds[seedsList_List,  $\Omega_?$ NumberQ, S_, options : OptionsPattern[]] := Block[
  {equations, roots, t = Symbol["t"], tt = Symbol["tt"],
     $\tau$  = Symbol[" $\tau$ "], indVars, depVar, depVarRule, fullSeedVars, tolerances},
  indVars = OptionValue[IndependentVariables] /.
    {"RecombinationTime" → "t", "ExcursionTime" → " $\tau$ ", "IonizationTime" → "tt"};
  depVar = First[DeleteCases[{"t", " $\tau$ ", "tt"}, Alternatives @@ indVars]];
  depVarRule = depVar /. {"tt" → {tt → t -  $\tau$ }, "t" → {t → tt +  $\tau$ }, " $\tau$ " → { $\tau$  → t - tt}};
  fullSeedVars[seed_] := <|"t" → seed[[1]], " $\tau$ " → seed[[2]], "tt" → seed[[1]] - seed[[2]]|>;
  equations = {D[S[t, tt], t] ==  $\Omega$ , D[S[t, tt], tt] == 0} /. depVarRule;
  tolerances = Which[
    ListQ[OptionValue[Tolerance]], OptionValue[Tolerance],
    True, ConstantArray[
      Which[
        NumberQ[OptionValue[Tolerance]], OptionValue[Tolerance],
        True, 10^If[NumberQ[OptionValue[WorkingPrecision]],
          2 - OptionValue[WorkingPrecision], 2 - $MachinePrecision]
      ]
    , 2]];

```

```

SortBy[
DeleteDuplicates[
Select[
Table[
Check[
roots = ({t,  $\tau$ } /. depVarRule) /. (
FindRoot[
equations
, {Symbol[#], fullSeedVars[seed][[#]]} & /@ indVars
, Evaluate[Sequence @@ FilterRules[{options}, Options[FindRoot]]]
]
/. {{}  $\rightarrow$  (({t,  $\tau$ } /. depVarRule)  $\rightarrow$  {}))
, Message[GetSaddlesFromSeeds::error,  $\Omega$ ]; roots
]
, {seed, seedsList}]
, Function[timesPair, OptionValue[SelectionFunction][timesPair[[1], timesPair[[2]], S,  $\Omega$ ]]
]
, Function[{timesPair1, timesPair2},
And @@ Thread[Abs[timesPair1 - timesPair2] < tolerances] ]
]
, If[
ListQ[OptionValue[SortingFunction]],
Table[Function[timesPair, f[timesPair[[1], timesPair[[2]], S,  $\Omega$ ]],
{f, OptionValue[SortingFunction]}],
Function[timesPair, OptionValue[SortingFunction][timesPair[[1], timesPair[[2]], S,  $\Omega$ ]]
]
]
]
End[];

```

## ClassifyQuantumOrbits



```

ClassifyQuantumOrbits::usage =
  "ClassifyQuantumOrbits[saddlePoints,f] sorts an indexed set of saddle
    points of the form  $\langle |\Omega_1 \rightarrow \{\{t_{11}, \tau_{11}\}, \{t_{12}, \tau_{12}\}, \dots\} \dots \rangle$  using a function f,
    which should turn  $f[t, \tau, \Omega]$  into an appropriate label, and returns an
    association of the form  $\langle |\text{label}_1 \rightarrow \langle |\Omega_1 \rightarrow \langle |1 \rightarrow \{t, \tau\}, 2 \rightarrow \{t, \tau\}, \dots \rangle, \dots \rangle, \dots \rangle$ .

ClassifyQuantumOrbits[saddlePoints,f,sortFunction] uses the function sortFunction to sort
  the sets of saddle points  $\{\{t_{11}, \tau_{11}\}, \{t_{12}, \tau_{12}\}, \dots\}$  for each label and harmonic energy.

ClassifyQuantumOrbits[saddlePoints,f,sortFunction,DiscardedLabels $\rightarrow\{\text{label}_1, \text{label}_2, \dots\}$ ]
  specifies a list of labels to discard from the final output.";
DiscardedLabels::usage = "DiscardedLabels is an option for ClassifyQuantumOrbits
  which specifies a list of labels to discard from the final output.";

Begin["`Private`"];

Options[ClassifyQuantumOrbits] = {DiscardedLabels  $\rightarrow \{\}$ };
Protect[DiscardedLabels];

ClassifyQuantumOrbits[saddlePointList_,
  classifierFunction_, sortingFunction_: Sort, OptionsPattern[]] := Map[
  Composition[
    Association,
    MapIndexed[#2[[1]]  $\rightarrow$  #1 &],
    sortingFunction
  ],
  Delete[DeleteMissing[
    Query[Transpose][
      MapIndexed[
        GroupBy[classifierFunction@@# &][
          Flatten/@Transpose[{#1, ConstantArray[#2[[{1}], 1], Length[#1]]}] &
          , saddlePointList][All, All, All, {1, 2}]
        ]
      , 2], List /@ OptionValue[DiscardedLabels]]
    , {2}]

End[];

```

## ReperiodSaddles

```

ClearAll[ReperiodSaddles]
ReperiodSaddles::usage =
  "ReperiodSaddles[{{t1,τ1},{t2,τ2},...},f] readjusts the assigned cycle of
  the saddle points {ti,τi}, returning the list {{t1+f[t1,τ1],τ1},...}."

ReperiodSaddles[<|Ω1→{{t11,τ11},...},Ω2→...|>,f]
  reperiods saddle-point pairs in a harmonic-energy-indexed association.

ReperiodSaddles[<|label1→<|Ω1→{{t11,τ11},...},...|>,...|>,f]
  reperiods saddle-point pairs of a classified set of saddle points.";

Begin["`Private`"];

ReperiodSaddles[pair_ /; Depth[pair] == 2, f_] := {pair[[1]] + f[pair[[1]], pair[[2]]], pair[[2]]}
ReperiodSaddles[association_, f_] := Apply[f, association, {Depth[association] - 2}]

End[];

```

## HessianRoot

```

HessianRoot::usage = "HessianRoot[S,t,τ] calculates the Hessian root  $\sqrt{\frac{(2\pi)^2}{\hbar^2 \text{Det}[\partial_{\{t,\tau\}}^2 S]}}$ .";

Begin["`Private`"];

HessianRoot[S_, t_, τ_] :=  $\sqrt{\frac{2\pi}{\hbar \text{Derivative}[0, 2][S][t, t - \tau]}}$ 
 $\sqrt{((2\pi \text{Derivative}[0, 2][S][t, t - \tau]) / (\hbar (\text{Derivative}[2, 0][S][t, t - \tau] \text{Derivative}[0, 2][S][t, t - \tau] - \text{Derivative}[1, 1][S][t, t - \tau]^2)))}$ 

End[];

```

## FindStokesTransitions

```

FindStokesTransitions::usage =
  "FindStokesTransitions[S,<|Ω1→<|1→{t11,τ11},2→{t12,τ12}|>,Ω2→<|1→{t21,τ21},2→{t22,τ22}|>,...
  |>] finds the set {{ΩS},{ΩAS},n} of the Stokes and anti-Stokes transition
  energies for the given set of saddle points, where Re(S) changes sign after the
  ΩS and Im(S) changes sign after the ΩAS, and n is the index of the member of
  the pair that should be chosen after the transition (taken as the member with
  a positive imaginary part of the action at the largest Ωi in the given keys).

FindStokesTransitions[S,<|label1→<|Ω1→...|>|>] finds the Stokes transitions for the given set
  of saddle-point curve pairs, and returns them labeled with the labeli.";

FindStokesTransitions::saddleno = "FindStokesTransitions called with `1`

```

```

of `2` saddle-point sets of length different from 2, with set
length structure `3`. Excluding those sets from the calculation.";
FindStokesTransitions::multipleS = "FindStokesTransitions found multiple
Stokes transitions; using `1` to return a single transition.";
FindStokesTransitions::multipleAS = "FindStokesTransitions found multiple
anti-Stokes transitions; using `1` to return a single transition.";
ChooserFunction::usage = "ChooserFunction is an option for FindStokesTransitions
that specifies which transition to take if there are multiple transitions
in the given dataset. The default is Last and gives the one with higher
energy; to get the full set of transitions found use Full or Identity.";
ReperiodingFunction::usage = "ReperiodingFunction is an option for FindStokesTransitions,
SPAdipole and UAdipole which specifies a function f[t,τ] of recombination
time t and excursion time τ that will be used to re-period the pairs
{t,τ} into the form {t+f[t,τ],τ}. The default is Function[0], but
if pairs are split it can be useful to set ReperiodingFunction to
Function[{t,τ},Floor[-Re[t-τ], $\frac{2\pi}{\omega}$ ]] for ω the carrier frequency. In general,
however, it is preferable to do this in a single go using ReperiodSaddles.";

Begin["`Private`"];

Protect[ReperiodingFunction, ChooserFunction];
Options[FindStokesTransitions] =
{ReperiodingFunction → Function[{t, τ}, 0], ChooserFunction → Automatic};

FindStokesTransitions[S_,
deeperAssociation_ /; Depth[deeperAssociation] == 5, options : OptionsPattern[]] := Map[
FindStokesTransitions[S, #, options] &,
deeperAssociation
]

FindStokesTransitions[S_, saddlesAssociation_, options : OptionsPattern[]] :=
Block[{reducedSaddlesAssociation, actionList, signsList, s, processor},
reducedSaddlesAssociation = KeySort[Select[saddlesAssociation, Length[#] == 2 &]];
If[Length[saddlesAssociation] - Length[reducedSaddlesAssociation] > 0,
Message[FindStokesTransitions::saddleno,
Length[saddlesAssociation] - Length[reducedSaddlesAssociation],
Length[saddlesAssociation], First /@ Tally /@ Split[Values[Length /@ saddlesAssociation]]
]
];
actionList = ReIm[
Map[(#reduces each  $\Omega \rightarrow \langle 1 \rightarrow S_1, 2 \rightarrow S_2 \rangle$  to  $\Omega \rightarrow (S_1 - S_2) *$ )
Apply[Subtract],
MapIndexed[(#reduces each  $\Omega \rightarrow \langle 1 \rightarrow \{t_1, \tau_1\}, 2 \rightarrow \{t_2, \tau_2\} \rangle$  to  $\Omega \rightarrow \langle 1 \rightarrow S_1, 2 \rightarrow S_2 \rangle *$ )
With[
{t = #1[[1]] + OptionValue[ReperiodingFunction][#1[[1]], #2[[2]]], τ = #1[[2]], Ω = #2[[1, 1]]},
S[t, t - τ] - Ω t
] &

```

```

        , reducedSaddlesAssociation, {2}
    ]]
];
signsList = Sign[Times[
    Rest[actionList],
    AssociationThread[Rest[Keys[actionList]], Most[Values[actionList]]]
]];
processor = OptionValue[ChooserFunction] /. {Automatic → Last, Full → Identity};
If[Length[Keys[Select[signsList, #[[1]] < 0 &]]] > 1,
    Message[FindStokesTransitions::multipleS, processor]];
If[Length[Keys[Select[signsList, #[[2]] < 0 &]]] > 1,
    Message[FindStokesTransitions::multipleAS, processor]];
{
    processor[Keys[Select[signsList, #[[1]] < 0 &]] /. {{}} → {Missing["No transition"]}],
    processor[Keys[Select[signsList, #[[2]] < 0 &]] /. {{}} → {Missing["No transition"]}],
    Sign[Last[actionList][[2]] /. {1 → 2, -1 → 1}
}
]

End[];

```

SPAdipole

```

SPAdipole::usage =
  "SPAdipole[S,prefactor, $\Omega$ ,{t, $\tau$ }] returns the saddle-point approximation amplitude
  corresponding to action  $S[t,t-\tau]-\Omega t$  and the given prefactor.

SPAdipole[S,prefactor, $\Omega$ ,<|1→{t1, $\tau$ 1},2→{t2, $\tau$ 2},...|>] returns the total harmonic-dipole
  contribution in the saddle-point approximation from the specified saddle points.

SPAdipole[S,prefactor, $\Omega$ ,<|1→{t1, $\tau$ 1},2→{t2, $\tau$ 2}|>,transition] uses the given Stokes
  transition set to drop the relevant saddle after the anti-Stokes transition.";
SPAdipole::wrongno = "SPAdipole called with a Stokes transition but
  with an input association of length `1` at harmonic
  energy  $\Omega=\text{'2'}$ . Reverting to unstructured evaluation.";
SPAdipole::invldtrns = "SPAdipole called with invalid Stokes transition
  set `1`. Reverting to unstructured evaluation.";

Begin["`Private`"];

Options[SPAdipole] = {ReperiodingFunction → Function[{t,  $\tau$ ], 0}};

SPAdipole[S_, prefactor_,  $\Omega$ _, {t_,  $\tau$ _}, options : OptionsPattern[]] :=
  Block[{tr = t + OptionValue[ReperiodingFunction][t,  $\tau$ ]},
    HessianRoot[S, tr,  $\tau$ ] prefactor[tr, tr -  $\tau$ ] Exp[- $\frac{i}{2} S[tr, tr - \tau] + \frac{i}{2} \Omega tr$ ]
  ]
SPAdipole[S_, prefactor_,  $\Omega$ _, times_Association, options : OptionsPattern[]] := Block[{},
  Total[SPAdipole[S, prefactor,  $\Omega$ , #, options] & /@ times]
]

SPAdipole[S_, prefactor_,  $\Omega$ _, times_Association,
  transition_, options : OptionsPattern[]] := Block[{},
  If[! NumberQ[transition[[2]]], Message[SPAdipole::invldtrns, transition];
  Return[SPAdipole[S, prefactor,  $\Omega$ , times]]];
  If[Length[times] ≠ 2, Message[SPAdipole::wrongno, Length[times],  $\Omega$ ];
  Return[SPAdipole[S, prefactor,  $\Omega$ , times, options]]];
  If[ $\Omega$  < transition[[2]],
    SPAdipole[S, prefactor,  $\Omega$ , times, options],
    SPAdipole[S, prefactor,  $\Omega$ , KeySelect[times, # == transition[[3]] &], options]
  ]
]

End[];

```

## UAdipole

```

UAdipole::usage =
  "UAdipole[S,prefactor,Ω,<|1→{t1,τ1},2→{t2,τ2},...|>,transition] returns the total
    harmonic-dipole contribution in the uniform approximation from the specified
    saddle points, taking the given Stokes transition set as a reference.";
UAdipole::saddleno = "UAdipole called with `1` time pairs at Ω=`2`.
  Reverting to the saddle-point approximation for this set.";
UAdipole::invldtrns = "UAdipole called with invalid Stokes transition set
  `1`. Reverting to the saddle-point approximation for this set.";

Begin["`Private`"];

Options[UAdipole] = {ReperiodingFunction → Function[{t, τ}, 0]};

UAdipole[S_, prefactor_, Ω_, times_, transition_, options:OptionsPattern[]] := (
  If[Length[times] ≠ 2, Message[UAdipole::saddleno, Length[times], Ω];
    Return[SPAdipole[S, prefactor, Ω, times] ]];
  If[! NumberQ[transition[[2]]], Message[UAdipole::invldtrns, transition];
    Return[SPAdipole[S, prefactor, Ω, times] ]];
  Block[
    {A1, A2, S1, S2, Ss, Sm, z,
      t1 = times[1][[1]] + OptionValue[ReperiodingFunction][times[1][[1]], times[1][[2]]],
      τ1 = times[1][[2]],
      t2 = times[2][[1]] + OptionValue[ReperiodingFunction][times[2][[1]], times[2][[2]]],
      τ2 = times[2][[2]]},
    A1 = HessianRoot[S, t1, τ1] prefactor[t1, t1 - τ1];
    S1 = S[t1, t1 - τ1] - Ω t1;
    A2 = HessianRoot[S, t2, τ2] prefactor[t2, t2 - τ2];
    S2 = S[t2, t2 - τ2] - Ω t2;
    Ss =  $\frac{S1 + S2}{2}$ ; Sm =  $\frac{S1 - S2}{2}$ ;
    If[Ω < transition[[2]], z =  $\left(-\frac{3}{2} Sm\right)^{2/3}$ ,
      z =  $\left(-\frac{3}{2} Sm\right)^{2/3} \text{Exp}\left[\pm \left(\text{transition}[[3]] /. \{2 \rightarrow -1, 1 \rightarrow 1\}\right) \frac{2\pi}{3}\right]$ ;
       $\sqrt{6\pi Sm} \text{Exp}\left[-\pm Ss + \pm \frac{\pi}{4}\right] \left(\frac{A1 - \pm A2}{2} \frac{\text{AiryAi}[-z]}{\sqrt{z}} + \pm \frac{A1 + \pm A2}{2} \frac{\text{AiryAi}'[-z]}{z}\right)$ 
    ]
  )

End[];

```

---

## Package closure

### End of package

```
EndPackage[];
```

### Add to distributed contexts.

```
DistributeDefinitions["RBSFA`"];
```

### Call dependency

```
Needs["RootFinder`", FileNameJoin[{$RBSFAdirectory, "RootFinder.m"}]]  
(*Called *after* the EndPackage[] call,  
so that the contexts RBSFA` and RootFinder` will both be subcontexts to Global`.*)
```