



**OpenEye**  
Scientific Software

**Zap TK – Python**

*Release 2.1.2*

**OpenEye Scientific Software, Inc.**

January 11, 2012



# CONTENTS

<b>1</b>	<b>Front Matter</b>	<b>1</b>
<b>2</b>	<b>Zap Theory and Examples</b>	<b>3</b>
2.1	The What of Zap . . . . .	3
2.2	The Why of Zap . . . . .	3
2.3	The Way of Zap . . . . .	4
<b>3</b>	<b>API</b>	<b>21</b>
3.1	OEPB Classes . . . . .	21
3.2	OEPB Constants . . . . .	40
3.3	OEPB Functions . . . . .	41
<b>4</b>	<b>Release Notes</b>	<b>45</b>
4.1	ZapTK 2.1.2 . . . . .	45
4.2	ZapTK 2.1.1 . . . . .	45
4.3	ZapTK 2.1.0 . . . . .	46
4.4	ZapTK 2.0.0 . . . . .	46
4.5	Indices and tables . . . . .	46
	<b>Index</b>	<b>47</b>



# FRONT MATTER

Copyright 1997-2012 OpenEye Scientific Software, Santa Fe, New Mexico. All rights reserved.

All rights reserved. This material contains proprietary information of OpenEye Scientific Software. Use of copyright notice is precautionary only and does not imply publication or disclosure.

The information supplied in this document is believed to be true but no liability is assumed for its use or the infringement of the rights of others resulting from its use. Information in this document is subject to change without notice and does not represent a commitment on the part of OpenEye Scientific Software.

This package is sold/licensed/distributed subject to the condition that it shall not, by way of trade or otherwise, be lent, re-sold, hired out or otherwise circulated without OpenEye Scientific Software's prior consent, in any form of packaging or cover other than that in which it was produced. No part of this manual or accompanying documentation, may be reproduced, stored in a retrieval system on optical or magnetic disk, tape, CD, DVD or other medium, or transmitted in any form or by any means, electronic, mechanical, photocopying recording or otherwise for any purpose other than for the purchaser's personal use without a legal agreement or other written permission granted by OpenEye.

This product should not be used in the planning, construction, maintenance, operation or use of any nuclear facility nor the flight, navigation or communication of aircraft or ground support equipment. OpenEye Scientific Software, shall not be liable, in whole or in part, for any claims arising from such use, including death, bankruptcy or outbreak of war.

Windows is a registered trademark of Microsoft Corporation. Apple, OS X, and Macintosh are registered trademarks of Apple Computer, Inc. AIX and IBM are registered trademarks of International Business Machines Corporation. UNIX is a registered trademark of the Open Group. RedHat is a registered trademark of RedHat, Inc. Linux is a registered trademark of Linus Torvalds. SPARC is a registered trademark of SPARC International Inc.

SYBYL is a registered trademark of TRIPOS, Inc. MDL is a registered trademark and ISIS is a trademark of Accelrys, Inc. SMILES, SMARTS, and SMIRKS may be trademarks of Daylight Chemical Information Systems. Macromodel is a trademark of Schrodinger, Inc.

Python is a trademark of the Python Software Foundation. Java is a trademark or registered trademark of Sun Microsystems, Inc. in the U.S. and other countries.

Other products and software packages referenced in this document are trademarks and registered trademarks of their respective vendors or manufacturers.



# ZAP THEORY AND EXAMPLES

## 2.1 The What of Zap

“A Smooth Permittivity Function for Poisson-Boltzmann Solvation Methods,” J. Andrew Grant, Barry T. Pickup and Anthony Nicholls, *J. Comp. Chem*, Vol 22, No.6, pgs 608-640, April 2001.

ZAP is, at its heart, a Poisson-Boltzmann (PB) solver. The Poisson equation describes how electrostatic fields change in a medium of varying dielectric, such as an organic molecule in water. The Boltzmann bit adds in the effect of mobile charge, e.g. salt. PB is an effective way to simulate the effects of water in biological systems. It relies on a charge description of a molecule, the designation of low (molecular) and high (solvent) dielectric regions and a description of an ion-accessible volume and produces a grid of electrostatic potentials. From this, transfer energies between different solvents, binding energies, pKa shifts, pI's, solvent forces, electrostatic descriptors, solvent dipole moments, surface potentials and dielectric focussing can all be calculated. As electrostatics is one of the two principal components of molecular interaction (the other, of course, being shape), ZAP is OpenEye's attempt to get it right.

## 2.2 The Why of Zap

ZAP happened by surprise. It had always been at the back of my mind that Something Should Be Done About DelPhi, the program written at Columbia University in the lab of Barry Honig by Barry, Kim Sharp, Mike Gilson, Shridhara Shridharan and me. While a very useful program, it was getting harder and harder to develop, partly because it was written in FORTRAN (not that there's anything wrong with that), partly because that's just the way academic code is. We don't know how to program but we want answers and we want them now.

ZAP really came about because of Andrew Grant. Andy was in the Scheraga lab struggling, as most of us do, with the nonlinear PB equation. He left to join AstraZeneca in 1993 but we kept in contact. In '95, he and Barry Pickup at the University of Sheffield published a remarkable paper (*J. Comp. Chem.*). They reported the hard-sphere volume of a molecule could be calculated to 0.1% accuracy by using atom-centered Gaussians. The correspondence between a discrete and a smooth, continuous function was nothing short of remarkable. Since the dielectric approach of PB is essentially volumetric because molecules are low-dielectric and solvent high, it occurred to us that this was the bedrock on which to build a new PB approach. That approach became ZAP.

There were several reasons to be excited about the Gaussian based approach. A smooth-function dielectric mapped onto a grid has few of the numerical problems plaguing DelPhi - instability with respect to grid placement, sensitivity to grid spacing. Also it is an interesting alternative physical model to most PB implementations. Typically it is assumed that the dielectric changes from molecular to solvent “discretely,” or infinitely fast. Why? As there was no experimental information on the variation of dielectric it was the simplest assumption and also the simplest to implement. As is often the way with a successful scientific approach, these early decisions become ossified over time.

Another ossification is the choice of molecular surface as the dielectric threshold, as first proposed by BH (*J. Phys. Chem*). It is not a bad choice. If a water molecule is excluded from some crevice of a molecule that ought to be

reflected in a lower dielectric in the crevice: a molecular surface-delimited volume reflects this where an atomic surface does not. There was also a numerical issue. When Mike Gilson first applied DelPhi to proteins (PROTEINS: Vol 17..), the atomic approach would place grid points randomly in tiny solvent crevices. Moving the protein relative to the grid shuffled the grid representation of the protein interior leading to large, unphysical changes in energy. The molecular surface removed these crevices unless they were large enough to fit a whole water molecule, in which case a high dielectric assignment was not unreasonable.

In retrospect, however, the molecular surface is a terrible choice! It's difficult to calculate and unstable with respect to small displacements of atoms or to the choice of a radius for water. And there is always the question of what dielectric should be assigned to the volume that lies between the molecular and the atomic surfaces.

The Gaussian approach answers several of the physical objections. First and foremost, the dielectric varies smoothly from interior to exterior. Although we made no attempt to correlate this variation with empirical evidence, choosing instead to match our method to DelPhi energies, the variation of polarizability over a span of about one Angstrom is physically appealing. Secondly, we achieve much of the crevice exclusion that the molecular surface produces. Because the Gaussian functions spread out beyond the hard sphere radius of each atom, a crevice receives density contributions from all neighboring atoms, lowering its dielectric to molecular levels. Thirdly, we do not see water "pops", those large changes in dielectric from small atomic displacements (or, worse, small changes of the protein placement on the grid) that occur in active sites. The Gaussian effectively interpolates between water absence and presence.

An advantage to this approach that we did not anticipate was its correspondence to DelPhi when applied to proteins. We parameterized the dielectric variation in ZAP so that it agrees with DelPhi for small molecules ("we" meaning Christine Kitchen's heroic work for her Ph.D. thesis). Given that the difference between the molecular surface and Gaussian model will be much larger for proteins with concavities, crevices and internal water pockets, we wondered how well the two would agree. The remarkable finding was that ZAP applied to proteins looks like DelPhi with twice the polarizability: ZAP with internal dielectric set to 2.0 looks like DelPhi set to 4.0. Why is this useful? Because there is circumstantial and theoretical evidence that 4.0 is a good dielectric to use with DelPhi for proteins, but that always left a loose thread in applying the method to protein-drug interactions - small molecules should have a dielectric of 2.0, proteins 4.0, but DelPhi only allows you to choose one internal dielectric. With ZAP it appears you get the best of both worlds. Protein and drug are set to internal dielectrics of 2.0 and because the protein is a little more "hydrated" in ZAP, it acts like the dielectric is set to 4.0. Please note that while 2.0 is the historic value for the inner dielectric, the current recommended value for the inner dielectric while using ZAP is 1.0.

On the numerical side, every advantage we had hoped for in ZAP came to pass. As mentioned in the previous section, we saw faster convergence, remarkable stability with respect to grid placement and much improved asymptotic behavior with decreasing grid spacing. We also, for the first time, had gradients that could be applied in dynamics or minimization.

## 2.3 The Way of Zap

There follows a dozen or so examples of using ZAP. The list is always growing because the toolkit philosophy really does work. Most of the examples use OEInterface to provide consistent command-line argument handling.

### 2.3.1 Partial Charges and Atomic Radii

There are three essential quantities for any PB calculation: coordinates, dielectric model (including radii), and charge. Coordinates will come from the atomic coordinates of the input molecule, but partial charges and the dielectric model deserve some extra attention.

Charges and radii can be calculated at run-time using functions provided by OEChem or other OpenEye toolkits (AM1-BCC charges from QuacPac for example). The dielectric model consists of both the value of the inner and outer dielectric, as well as how the dielectric adjusts from one to the other. ZAP uses atomic radii as the basis for its dielectric algorithm. By default, we recommend using Bondi VdW radii which can be set on a molecule using the function

OEAssignBondiVdWRadii Or you can set specific radii on an atom-by-atom basis using the OEAtomBase::SetRadius function.

The current recommended value for the inner and outer dielectrics for a molecule in an aqueous solution are 1.0 and 80.0. Therefore, these are the default values for the O EZap and OE Bind implementations. The historic value for the inner dielectric is 2.0, and this value may be set using the SetInnerDielectric method. Additionally, the default value for the inner dielectric of the OEET implementation remains at 2.0 for the time being.

Accurate PB calculations also depend on quality atomic charges. In OEChem, we provide MMFF94 partial charges as the lowest level of charges that we consider usable in PB. The examples that follow will use these charges, so that they don't depend on any toolkits beside ZAP and OEChem. But for production use, we recommend AM1-BCC charges

If your input file format can store partial charges and/or radii, you can use those values instead of calculating them at run time. OEB files can store radii and partial charges, so they serve as a useful intermediate file between many OpenEye programs. Note that charges and radii will only be written to an OEB file if they are present on the molecule when written.

Atomic charges can come in a PDB file, along with atomic radii, in the form of the extra fields added after the coordinates. This style originated with DelPhi. Note that to get OEChem to read charges and radii from these fields in a PDB file, you need to set a specific flavor on the input oemolistream before reading any molecules from the stream.

```
if ifs.GetFormat() == OEFormat_PDB:
    ifs.SetFlavor(OEFormat_PDB, OEIFlavor_PDB_Default | OEIFlavor_PDB_DELPHI)
```

## 2.3.2 Grids

ZAP uses regular cubic lattices, or grids, to solve the PB equation. The examples here illustrate how to retrieve such from the calculation and to manipulate and store the information held by each. Typically, grids are not used per se but information is extracted, such as the potential at particular points in space, as is illustrated in the next section. However, some programs, such as VIDA, can read grids and display their properties. Also, having direct access to grids allows for manipulations of the potential map that may not have been anticipated.

This first example shows the simplest method of generating a grid of potentials using ZAP. We save the grid in OpenEye GRD (\*.grd) format which is a compact, binary format that can be visualized in VIDA. Alternatively, the grid can be written into GRASP format (\*.phi), which means that the grid stored will be 65 cubed irregardless of the size of the grid used in the calculation. A 65<sup>3</sup> grid is obtained by interpolation to a grid with that many points that fits over the largest dimension of the grid calculated.

### Listing 1: Calculating a potential grid

```
#!/usr/bin/env python
#####
# Copyright (C) 2006, 2007, 2008, 2009 OpenEye Scientific Software, Inc.
#####
### zap_grid1.py
#####

import os, sys
from openeye.oechem import *
from openeye.oegrid import *
from openeye.oezap import *

def main(argv = [__name__]):
    if len(argv) != 2:
```

```

    OEThrow.Usage("%s <molfile>" % argv[0])

ifs = oemolistream()
if not ifs.open(argv[1]):
    OEThrow.Fatal("Unable to open %s for reading" % argv[1])
mol = OEGraphMol()
OEReadMolecule(ifs, mol)
OEAssignBondiVdWRadii(mol)
OEMMFFAtomTypes(mol)
OEMMFF94PartialCharges(mol)

epsin = 1.0
zap = OEZap()
zap.SetInnerDielectric(epsin)
zap.SetMolecule(mol)

grid = OEScalarGrid()
if zap.CalcPotentialGrid(grid):
    OEWriteGrid("zap.grd", grid)

if __name__ == "__main__":
    sys.exit(main(sys.argv))

```

The next example is an elaboration of the previous simple version where we add in control of the parameters of the calculation. Options are provided to set the internal and external (solute and solvent) dielectric constants, the distance between the molecule and the edges of the grid (boundary spacing or buffer) and the grid spacing. A smaller grid spacing implies a more dense and accurate grid, but it does come with a larger memory footprint.

## Listing 2: Calculating potential grid with optional parameters

```

#!/usr/bin/env python
#####
# Copyright (C) 2006, 2008, 2009 OpenEye Scientific Software, Inc.
#####
### zap_grid2.py
#####

import os, sys
from openeye.oechem import *
from openeye.oegrid import *
from openeye.oezap import *

def main(argv = [__name__]):
    itf = OEInterface()
    if not SetupInterface(argv, itf):
        return 1

    zap = OEZap()
    zap.SetInnerDielectric(itf.GetFloat("-epsin"))
    zap.SetOuterDielectric(itf.GetFloat("-epsout"))
    zap.SetGridSpacing(itf.GetFloat("-grid_spacing"))
    zap.SetBoundarySpacing(itf.GetFloat("-buffer"))

    mol = OEGraphMol()

```

```

ifs = oemolistream()
if not ifs.open(itf.GetString("-in")):
    OETHrow.Fatal("Unable to open %s for reading" % itf.GetString("-in"))
OEReadMolecule(ifs,mol)
OEAssignBondiVdWRadii(mol)
OEMMFFAtomTypes(mol)
OEMMFF94PartialCharges(mol)

zap.SetMolecule(mol)

grid = OEScalarGrid()
if zap.CalcPotentialGrid(grid):
    if itf.GetBool("-mask"):
        OEMaskGridByMolecule(grid, mol)
    OEWriteGrid(itf.GetString("-out"), grid)
return 0

InterfaceData = """
#zap_grid2 interface definition

!PARAMETER -in
  !TYPE string
  !BRIEF Input molecule file
  !REQUIRED true
!END

!PARAMETER -out
  !TYPE string
  !BRIEF Output grid file
  !REQUIRED true
!END

!PARAMETER -epsin
  !TYPE float
  !BRIEF Inner dielectric
  !DEFAULT 1.0
  !LEGAL_RANGE 0.0 100.0
!END

!PARAMETER -epsout
  !TYPE float
  !BRIEF Outer dielectric
  !DEFAULT 80.0
  !LEGAL_RANGE 0.0 100.0
!END

!PARAMETER -grid_spacing
  !TYPE float
  !DEFAULT 0.5
  !BRIEF Spacing between grid points (Angstroms)
  !LEGAL_RANGE 0.1 2.0
!END

!PARAMETER -buffer
  !TYPE float
  !DEFAULT 2.0
  !BRIEF Extra buffer outside extents of molecule.
  !LEGAL_RANGE 0.1 10.0

```

```
!END

!PARAMETER -mask
  !TYPE bool
  !DEFAULT false
  !BRIEF Mask potential grid by the molecule
!END
"""

def SetupInterface(argv, itf):
    OEConfigure(itf, InterfaceData)
    if OECheckHelp(itf, argv):
        return False
    if not OEParseCommandLine(itf, argv):
        return False
    if not OEIsReadable(OEGetFileType(OEGetFileExtension(itf.GetString("-in")))):
        OEThrow.Warning("%s is not a readable input file" % itf.GetString("-in"))
        return False
    if not OEIsWritableGrid(OEGetGridFileType(OEGetFileExtension(itf.GetString("-out")))):
        OEThrow.Warning("%s is not a writable grid file" % itf.GetString("-out"));
        return False
    return True

if __name__ == "__main__":
    sys.exit(main(sys.argv))
```

Here we show how to calculate a difference-map, that is to say the potential difference between a standard, two dielectric, calculation and a single dielectric calculation (approximating pure Coulombic potentials). These difference potentials represent the electrostatic response of the solvent to the charges within the solute molecule. If we mask away everything outside the molecule, we can see the contributions from the charges inside the molecule.

### Listing 3: Calculating potential grid with optional parameters

```
#!/usr/bin/env python
#####
# Copyright (C) 2006, 2007, 2008, 2009 OpenEye Scientific Software, Inc.
#####
### zap_grid3.py
#####
import os, sys
from openeye.ochem import *
from openeye.oegrid import *
from openeye.oezap import *

def main(argv = [__name__]):
    if len(argv) != 2:
        OEThrow.Usage("%s <molfile>" % argv[0])

    ifs = oemolistream()
    if not ifs.open(argv[1]):
        OEThrow.Fatal("Unable to open %s for reading" % argv[1])
    mol = OEGraphMol()

    OEReadMolecule(ifs, mol)
    OEAssignBondiVdWRadii(mol)
```

```

OEMMFFAtomTypes (mol)
OEMMFF94PartialCharges (mol)

zap = OEZap()
zap.SetInnerDielectric(1.0)
zap.SetMolecule(mol)

# calculate standard 2-dielectric grid
grid1 = OEScalarGrid()
zap.CalcPotentialGrid(grid1)

# calculate grid with single dielectric
grid2 = OEScalarGrid()
zap.SetOuterDielectric(zap.GetInnerDielectric())
zap.CalcPotentialGrid(grid2)

# take the difference
OESubtractScalarGrid(grid1, grid2)

# mask out everything outside the molecule
OEMaskGridByMolecule(grid1, mol, OEGridMaskType_GaussianMinus)

OEWriteGrid("zap_diff.grd", grid1)

if __name__ == "__main__":
    sys.exit(main(sys.argv))

```

### 2.3.3 Atom Potentials

The potentials at any charge, as calculated by PB, can be decomposed into three forms

1. Induced solvent potential: the potential produced by the polarization of the solvent.
2. Inter-charge Coulombic energy: the potential that would be produced if the solvent had the same dielectric as the solute molecule from all other charges.
3. Self potential: the potential produced by a charge at itself. Of course, as mentioned above, this is infinite analytically, but PB will produce an “approximation” to this infinity because the grid spacing is finite.

This example program shows how to calculate each of these quantities. In addition to command line flags to control dielectric, grid spacing, etc., there are three flags that affect the type of potentials calculated:

- `-calc_type solvent_only`
- `-calc_type remove_self`
- `-calc_type coulombic`

With none of these flags, the code executes a single PB calculation, interpolates the potentials from the grid produced, reports the sum of these potentials multiplied by the charge on the respective atoms and outputs these potentials and charges in a table. This potential corresponds to (a)+(b)+(b) above.

Using the `-calc_type solvent_only` option executes two PB calculations, the standard calculation plus a second calculation where the external dielectric has been set to the solute dielectric. The atom potentials are then formed from the difference of these two calculations such that the remaining potential is that produced by the solvent alone. The sum of this set of atomic potentials multiplied by atomic charges all multiplied by 0.5 is the electrostatic component of transferring from a media of the same dielectric as the solute to water. These potentials correspond to (a) above.

The `-calc_type remove_self` flag for the example program below executes an internal algorithm in the ZAP toolkit that extracts from each atom potential that potential produced by that atom, if it is charged. This it removes the artifactual energy from the grid, i.e. energy that is not actually physical but a manifestation of the finite resolution of the grids used. The remaining potential is then that from the solvent and that from all the other charges, that is (a)+(b) above.

The `-calc_type coulombic` flag actually prevents any PB calculation. It merely uses Coulomb's Law to calculate the potential (b), the inner-atomic Coulombic potential.

## Listing 4: Calculating atom potentials

```
#!/usr/bin/env python
#####
# Copyright (C) 2006, 2007, 2008, 2009 OpenEye Scientific Software, Inc.
#####

import os, sys
from openeye.oechem import *
from openeye.oezap import *

def Output(mol, apot, showAtomTable):
    print "Title: %s"%mol.GetTitle()
    if showAtomTable:
        OEThrow.Info("Atom potentials");
        OEThrow.Info("Index  Elem      Charge      Potential");

    energy=0.0
    for atom in mol.GetAtoms():
        energy += atom.GetPartialCharge()*apot[atom.GetIdx()]
        if showAtomTable:
            print "%3d      %2s      %6.3f      %8.3f"%(atom.GetIdx(),
                OEGetAtomicSymbol(atom.GetAtomicNum()),
                atom.GetPartialCharge(),
                apot[atom.GetIdx()])

    print "Sum of {Potential * Charge over all atoms * 0.5} in kT = %f\n" % (0.5*energy)

def CalcAtomPotentials(itf):
    mol = OEGraphMol()

    ifs = oemolistream()
    if not ifs.open(itf.GetString("-in")):
        OEThrow.Fatal("Unable to open %s for reading" % itf.GetString("-in"))

    OEReadMolecule(ifs,mol)
    OEAssignBondiVdWRadii(mol)

    if not itf.GetBool("-file_charges"):
        OEMMFFAtomTypes(mol)
        OEMMFF94PartialCharges(mol)

    zap = OEZap()
    zap.SetMolecule(mol)
    zap.SetInnerDielectric(itf.GetFloat("-epsin"))
    zap.SetBoundarySpacing(itf.GetFloat("-boundary"))
    zap.SetGridSpacing(itf.GetFloat("-grid_spacing"))
```

```

showAtomTable = itf.GetBool("-atomtable")
calcType = itf.GetString("-calc_type")
if calcType=="default":
    apot = OEFloatArray(mol.GetMaxAtomIdx())
    zap.CalcAtomPotentials(apot)
    Output(mol, apot, showAtomTable)

elif calcType == "solvent_only":
    apot = OEFloatArray(mol.GetMaxAtomIdx())
    zap.CalcAtomPotentials(apot)

    apot2 = OEFloatArray(mol.GetMaxAtomIdx())
    zap.SetOuterDielectric(zap.GetInnerDielectric())
    zap.CalcAtomPotentials(apot2)

    # find the differences
    for atom in mol.GetAtoms():
        idx=atom.GetIdx()
        apot[idx] -= apot2[idx]

    Output(mol, apot, showAtomTable)

elif calcType == "remove_self":
    apot = OEFloatArray(mol.GetMaxAtomIdx())
    zap.CalcAtomPotentials(apot, True)
    Output(mol, apot, showAtomTable)

elif calcType == "coulombic":
    epsin = itf.GetFloat("-epsin")
    x = OECoulombicSelfEnergy(mol, epsin)
    print "Coulombic Assembly Energy"
    print " = Sum of {Potential * Charge over all atoms * 0.5} in kT = %f"%x
    apot = OEFloatArray(mol.GetMaxAtomIdx())
    OECoulombicAtomPotentials(mol, epsin, apot)
    Output(mol, apot, showAtomTable)

return 0

def SetupInterface(itf, InterfaceData):
    OEConfigure(itf, InterfaceData)
    if OECheckHelp(itf, sys.argv):
        return False
    if not OEParseCommandLine(itf, sys.argv):
        return False
    return True

def main(InterfaceData):
    itf=OEInterface()
    if not SetupInterface(itf, InterfaceData):
        return 1

    return CalcAtomPotentials(itf)

InterfaceData=""
#zap_atompot interface definition

!PARAMETER -in
!TYPE string

```

```
!BRIEF Input molecule file.
!REQUIRED true
!KEYLESS 1
!END

!PARAMETER -file_charges
!TYPE bool
!DEFAULT false
!BRIEF Use partial charges from input file rather than calculating with MMFF.
!END

!PARAMETER -calc_type
!TYPE string
!DEFAULT default
!LEGAL_VALUE default
!LEGAL_VALUE solvent_only
!LEGAL_VALUE remove_self
!LEGAL_VALUE coulombic
!LEGAL_VALUE breakdown
!BRIEF Choose type of atom potentials to calculate
!END

!PARAMETER -atomtable
!TYPE bool
!DEFAULT false
!BRIEF Output a table of atom potentials
!END

!PARAMETER -epsin
!TYPE float
!BRIEF Inner dielectric
!DEFAULT 1.0
!LEGAL_RANGE 0.0 100.0
!END

!PARAMETER -grid_spacing
!TYPE float
!DEFAULT 0.5
!BRIEF Spacing between grid points (Angstroms)
!LEGAL_RANGE 0.1 2.0
!END

!PARAMETER -boundary
!ALIAS -buffer
!TYPE float
!DEFAULT 2.0
!BRIEF Extra buffer outside extents of molecule.
!LEGAL_RANGE 0.1 10.0
!END
"""

if __name__ == "__main__":
    sys.exit(main(InterfaceData))
```

### 2.3.4 Solvation Energies: PBSA

A principle use of PB is to calculate the energy stored in the exterior dielectric, for example the partial alignment of water dipoles. This corresponds to the difference between a calculation with uniform dielectric and one with a different external dielectric. In the examples for atom potentials this would be the sum of the solvent potentials at each atom, multiplied by the charges on that atom all multiplied by 0.5.

However, we know water also has an energy component due to hydrophobicity, which is typically estimated as a constant (approximately 10-25 calories per square Angstrom) multiplied by the accessible area of the molecule. This value is an open scientific question. We recommend using 10 for vacuum-water transfer energies and 25 for protein calculations, but ultimately your own scientific judgement should be used. These examples include an area calculation. Taken together these numbers are an approximation of the transfer energy from a low dielectric medium (alkane solvent, protein interior) into water.

The following examples use the OEArea object to calculate the accessible surface area of the molecule. The area is calculated using the same grid based Gaussians that ZAP uses. Therefore, it will not give the same results as triangle summation methods for calculating surface area, which are used in OESpicoli.

#### Listing 5: Calculating solvation energies

```
#!/usr/bin/env python
#####
# Copyright (C) 2006, 2007, 2008, 2009 OpenEye Scientific Software, Inc.
#####
import os, sys
from openeye.ochem import *
from openeye.oezap import *

KCalsPerKT = 0.59
KCalsPerSqAngstrom = 0.025

def PrintHeader():
    print "Title          Solv(kcal)   Area(Ang^2)   Total(kcal)   Int.Coul(kcal)\n"

def PrintLine(title, solv, area, coul):
    total = KCalsPerKT*solv + KCalsPerSqAngstrom*area;
    print "%-12s  %6.2f  %6.2f  %6.2f  %6.2f"%(title,
        KCalsPerKT*solv, area, total, KCalsPerKT*coul)

def main(argv = [__name__]):
    if len(argv) != 2:
        OThrow.Usage("%s <molfile>" % argv[0])

    epsin = 1.0

    zap=OEZap()
    zap.SetInnerDielectric(epsin)
    zap.SetGridSpacing(0.5)

    area = OEArea()

    PrintHeader()
    mol = OEGraphMol()
    ifs = oemolistream()
    if not ifs.open(argv[1]):
        OThrow.Fatal("Unable to open %s for reading" % argv[1])
```

```
while OEReadMolecule(ifs, mol):
    OEAssignBondiVdWRadii(mol)
    OEMMFFAtomTypes(mol)
    OEMMFF94PartialCharges(mol)
    zap.SetMolecule(mol)
    solv = zap.CalcSolvationEnergy()
    aval = area.GetArea(mol)
    coul = OECoulombicSelfEnergy(mol, epsin)
    PrintLine(mol.GetTitle(), solv, aval, coul)

return 0

if __name__ == "__main__":
    sys.exit(main(sys.argv))
```

This next solvation example calculates the transfer energy from vacuum to water.

## Listing 6: Calculating vacuum-water transfer energies

```
#!/usr/bin/env python
#####
# Copyright (C) 2006, 2007, 2008, 2009 OpenEye Scientific Software, Inc.
#####
import os, sys
from openeye.oechem import *
from openeye.oezap import *

KCalsPerKT = 0.59
KCalsPerSqAngstrom = 0.010

def main(argv = [__name__]):

    if len(argv) != 2:
        OETHrow.Usage("%s <molfile>" % argv[0])

    epsin = 1.0

    zap=OEZap()
    zap.SetInnerDielectric(epsin)
    zap.SetGridSpacing(0.5)

    area = OEArea()

    mol = OEGraphMol()
    ifs = oemolistream()
    if not ifs.open(argv[1]):
        OETHrow.Fatal("Unable to open %s for reading" % argv[1])
    OETHrow.Info("%-20s %6s\n" % ("Title", "Vacuum->Water(kcal)"));
    while OEReadMolecule(ifs, mol):
        OEAssignBondiVdWRadii(mol)
        OEMMFFAtomTypes(mol)
        OEMMFF94PartialCharges(mol)
        zap.SetMolecule(mol)
        solv = zap.CalcSolvationEnergy()
        aval = area.GetArea(mol)
```

```

    OEThrow.Info("%-20s %6.2f" % (mol.GetTitle(),
                               KCalsPerKT*solv+KCalsPerSqAngstrom*aval));

    return 0

if __name__ == "__main__":
    sys.exit(main(sys.argv))

```

### 2.3.5 Binding Energies

Binding energies and other binding related properties can be calculated using the `OEBind` class. The `OEBind` class serves two primary purposes: 1) to have an class in place for users to easily calculate binding related data, and 2) to show how binding related data may be calculated. `OEBind` is not particularly well-suited for extremely efficient calculations required by simulations. This is because some of the data available from `OEBind` may be considered uninteresting for the project and there is no need to spend time calculating it, or because `OEBind` calculates the unbound protein and ligand properties everytime, which is unnecessary if the same protein and ligand appear in multiple calculations. The API section for `OEBind` is detailed with regard to implementation, so the user can see how to create their own class or set of function calls if needed.

There are two classes that store the results of a binding calculation, `OEBindResults` and `OESimpleBindResults`. `OEBindResults` is a superset of `OESimpleBindResults`. For an in-depth understanding of what is calculated, the electrostatics portion of the algorithm for the `SimpleBind` function is shown in the API section.

The following is a sample program for using the `OEBind` class that is provided in the distribution.

#### Listing 7: Bind example

```

#!/usr/bin/env python
#####
# Copyright (C) 2006, 2007, 2008, 2009 OpenEye Scientific Software, Inc.
#####
### bind.py
#####

import os, sys
from openeye.ochem import *
from openeye.oezap import *

def main(argv = [__name__]):
    if len(argv) != 3:
        OEThrow.Usage("%s <protein> <ligands>" % argv[0])

    protein = OEMol()

    ifs = oemolistream()
    if not ifs.open(argv[1]):
        OEThrow.Fatal("Unable to open %s for reading" % argv[1])
    OEReadMolecule(ifs, protein)

    OEAssignBondiVdWRadii(protein)
    OEMMFFAtomTypes(protein)
    OEMMFF94PartialCharges(protein)
    print >>sys.stderr, "protein: " + protein.GetTitle()

```

```
    epsin = 1.0;
    bind = OEBind()
    bind.GetZap().SetInnerDielectric(epsin);
    bind.SetProtein(protein)
    results = OEBindResults()

    if not ifs.open(argv[2]):
        OETHrow.Fatal("Unable to open %s for reading" % argv[2])
    ifs.SetConfTest(OEIsomericConfTest())

    ligand = OEMol()
    while OEReadMolecule(ifs, ligand):
        OEAssignBondiVdWRadii(ligand)
        OEMMFFAtomTypes(ligand)
        OEMMFF94PartialCharges(ligand)
        print >>sys.stderr, "ligand: %s has %d conformers" % \
            (ligand.GetTitle(), ligand.NumConfs())

        for conf in ligand.GetConfs():
            bind.Bind(conf, results)
            print >>sys.stderr, " conf# %d be = %f" % \
                (conf.GetIdx(), results.GetBindingEnergy())

    return 0

if __name__ == "__main__":
    sys.exit(main(sys.argv))
```

### 2.3.6 Focussing

Focussing is a way to achieve the desired precision around a target (such as a ligand) while maintaining reasonable time and memory limits for the calculation. The target for focussing is set using the `OEZap.SetFocusTarget` method.

For a typical ZAP electrostatics calculation, a consistent grid spacing is used for the entire system, where the default value is 0.5 Angstroms but it may be set to a custom value using `OEZap.SetGridSpacing`. This method is entirely appropriate for certain calculations, such as solvation energy calculations for small molecules. For other types of calculations, such as a binding energy calculation for a protein and ligand, a consistent grid spacing may cause the calculation to be either prohibitively expensive or insufficiently precise around the binding area.

Focussing alleviates this problem by using a fine grid for the target volume, and coarser grids away from the target. The grid spacing setting for ZAP is applied to the target volume, and the grid spacing is doubled for each addition coarse grid surrounding the target. A quadratic interpolation is used for the grid intersections. The implementation of the bind uses the `SetFocusTarget()` method to set the ligand as the target for focussing.

The following example program computes the binding energy of a protein and ligand twice, once without focussing and once with the ligand as the focussing target. The values of the binding energy and the time it took to calculate them are displayed. For a focussed atom potential calculation, the full electrostatics for the protein and complex would each be computed with multiple electrostatics calculations. The first calculation would create the potential grid for the target volume with a grid spacing of 0.5 (assuming the default spacing is being used). An additional calculation with a grid spacing of 1.0 would be done on a larger volume volume, and then additional calculations with grid spacings of 2.0, 4.0, and so on would be done until the grid is large enough to contain the entire volume of the system.

## Listing 8: Focussing example

```
#!/usr/bin/env python
#####
# Copyright (C) 2006, 2007, 2008, 2009 OpenEye Scientific Software, Inc.
#####
import os, sys
from openeye.oechem import *
from openeye.oezap import *

def PrintHeader(protTitle, ligTitle):
    OEThrow.Info("\nBinding Energy and Wall Clock Time for %s and %s" %
                (protTitle, ligTitle))
    OEThrow.Info("%15s %15s %10s" % ("Focussed?", "Energy(kT)", "Time(s)"))

def PrintInfo(focussed, energy, time):
    OEThrow.Info("%15s %15.3f %10.1f" % (focussed, energy, time))

def CalcBindingEnergy(zap, protein, ligand, cmplx):
    stopwatch = OESTopwatch()
    stopwatch.Start()

    ppot = OEFloatArray(protein.GetMaxAtomIdx())
    zap.SetMolecule(protein)
    zap.CalcAtomPotentials(ppot)
    proteinEnergy = 0.0
    for atom in protein.GetAtoms():
        proteinEnergy+=ppot[atom.GetIdx()]*atom.GetPartialCharge()
    proteinEnergy *= 0.5

    lpot = OEFloatArray(ligand.GetMaxAtomIdx())
    zap.SetMolecule(ligand)
    zap.CalcAtomPotentials(lpot)
    ligandEnergy = 0.0
    for atom in ligand.GetAtoms():
        ligandEnergy+=lpot[atom.GetIdx()]*atom.GetPartialCharge()
    ligandEnergy *= 0.5

    cpot = OEFloatArray(cmplx.GetMaxAtomIdx())
    zap.SetMolecule(cmplx)
    zap.CalcAtomPotentials(cpot)
    cmplxEnergy = 0.0
    for atom in cmplx.GetAtoms():
        cmplxEnergy+=cpot[atom.GetIdx()]*atom.GetPartialCharge()
    cmplxEnergy *= 0.5

    energy = cmplxEnergy - ligandEnergy - proteinEnergy
    time = stopwatch.Elapsed()

    if (zap.IsFocusTargetSet()):
        focussed = "Yes"
    else:
        focussed = "No"

    PrintInfo(focussed, energy, time);

def main(argv = [__name__]):
```

```
if len(argv) != 3:
    OEThrow.Usage("%s <protein> <ligand>" % argv[0])

ifs = oemolistream()
if not ifs.open(argv[1]):
    OEThrow.Fatal("Unable to open %s for reading" % argv[1])
protein = OEGraphMol()
OEReadMolecule(ifs, protein)

if not ifs.open(argv[2]):
    OEThrow.Fatal("Unable to open %s for reading" % argv[2])
ligand = OEGraphMol()
OEReadMolecule(ifs, ligand)

OEAssignBondiVdWRadii(protein)
OEMMFFAtomTypes(protein)
OEMMFF94PartialCharges(protein)

OEAssignBondiVdWRadii(ligand)
OEMMFFAtomTypes(ligand)
OEMMFF94PartialCharges(ligand)

cplx = OEGraphMol(protein)
OEAddMols(cplx, ligand)

epsin = 1.0
spacing = 0.5
zap = OEZap()
zap.SetInnerDielectric(epsin)
zap.SetGridSpacing(spacing)

PrintHeader(protein.GetTitle(), ligand.GetTitle())

CalcBindingEnergy(zap, protein, ligand, cplx)
zap.SetFocusTarget(ligand)
CalcBindingEnergy(zap, protein, ligand, cplx)

return 0

if __name__ == "__main__":
    sys.exit(main(sys.argv))
```

### 2.3.7 Gradients/Forces

The solvent forces acting on the atoms in a molecule may be calculated using the `OEZap.CalcForces` method. The components of the forces are set to a float array of length  $3N$ , where  $N$  is the number of atoms. The components of the gradient are equal in magnitude to the force, but have the opposite sign.

#### Listing 9: Calculating solvation forces

```
#!/usr/bin/env python
#####
# Copyright (C) 2006, 2007, 2008, 2009 OpenEye Scientific Software, Inc.
#####
```

```

import os, sys
from openeye.oechem import *
from openeye.oezap import *

def PrintHeader(title):
    OEThrow.Info("\nForce Components for %s, in kT/Angstrom" % title)
    OEThrow.Info("%6s %8s %8s %8s %8s" % ("Index", "Element", "-dE/dx",
        "-dE/dy", "-dE/dz"))

def PrintForces(mol, forces):
    for atom in mol.GetAtoms():
        OEThrow.Info("%6d %8s %8.2f %8.2f %8.2f" % (atom.GetIdx(),
            OEGetAtomicSymbol(atom.GetAtomicNum()), forces[atom.GetIdx()],
            forces[atom.GetIdx()+1], forces[atom.GetIdx()+2]))

def main(argv = [__name__]):
    if len(argv) != 2:
        OEThrow.Usage("%s <molfile>" % argv[0])

    epsin = 1.0
    zap = OEZap()
    zap.SetInnerDielectric(epsin);

    mol = OEGraphMol()
    ifs = oemolistream()
    if not ifs.open(argv[1]):
        OEThrow.Fatal("Unable to open %s for reading" % argv[1])
    while OEReadMolecule(ifs, mol):
        PrintHeader(mol.GetTitle())
        forces = OEFloatArray(mol.GetMaxAtomIdx()*3)
        OEAssignBondiVdWRadii(mol)
        OEMMFFAtomTypes(mol)
        OEMMFF94PartialCharges(mol)
        zap.SetMolecule(mol)
        zap.CalcForces(forces)
        PrintForces(mol, forces)

    return 0

if __name__ == "__main__":
    sys.exit(main(sys.argv))

```

### 2.3.8 Electrostatic Similarity

Electrostatic similarity may be calculated using the `OEEET` class. The following example program shows how to obtain the electrostatic Tanimoto between a reference molecule and a trial molecule. The electrostatic Tanimoto is affected by the partial charges of the molecules as well as the three-dimensional structure, including the tautomer state, spacial orientation, and conformation. In the example program shown below, MMFF charges are assigned to the molecules, but it is assumed that they have already been spatially aligned.

#### Listing 10: Calculating electrostatic tanimoto

```

#!/usr/bin/env python
#####

```

```
# Copyright (C) 2006, 2007, 2008, 2009 OpenEye Scientific Software, Inc.
#####
### calc_et.py
#####

import os, sys
from openeye.oechem import *
from openeye.oezap import *

def main(argv = [__name__]):
    if len(argv) != 3:
        OEThrow.Usage("calc_et.py <reffile> <fitfile>")

    refmol=OEGraphMol()

    ifs = oemolistream()
    if not ifs.open(argv[1]):
        OEThrow.Fatal("Unable to open %s for reading" % argv[1])
    OEReadMolecule(ifs,refmol)
    OEAssignBondiVdWRadii(refmol)
    OEMMFFAtomTypes(refmol)
    OEMMFF94PartialCharges(refmol)

    et = OEET()
    et.SetRefMol(refmol)

    OEThrow.Info("dielectric: %.4f" % et.GetDielectric())
    OEThrow.Info("inner mask: %.4f" % et.GetInnerMask())
    OEThrow.Info("outer mask: %.4f" % et.GetOuterMask())
    OEThrow.Info("salt conc : %.4f" % et.GetSaltConcentration())
    OEThrow.Info("join      : %d" % et.GetJoin())

    if not ifs.open(argv[2]):
        OEThrow.Fatal("Unable to open %s for reading" % argv[2])
    fitmol = OEGraphMol()
    while OEReadMolecule(ifs, fitmol):
        OEAssignBondiVdWRadii(fitmol)
        OEMMFFAtomTypes(fitmol)
        OEMMFF94PartialCharges(fitmol)
        OEThrow.Info("Title: %s, ET %.2f" %
                    (fitmol.GetTitle(), et.Tanimoto(fitmol)))
    return 0

if __name__ == "__main__":
    sys.exit(main(sys.argv))
```

## 3.1 OEPB Classes

### 3.1.1 OEArea

`class OEArea`

This class represents *OEArea*. *OEArea* is a simple object that calculates surface area using the same grid-based Gaussians that Zap uses. This class is mostly used for calculating the area term in solvation calculations or the buried area term in bind calculations.

#### Constructors

```
OEArea()  
OEArea(const OEArea &)
```

Default and copy constructors.

#### operator=

```
OEArea &operator=(const OEArea &)
```

Assignment operator.

#### GetArea

```
float GetArea(const OEChem::OEMolBase &mol)  
bool GetArea(const OEChem::OEMolBase &mol, float *atomArea)
```

Calculate the surface area of the passed-in molecule. The first version calculates the surface area of the entire molecule. The second version takes an array sized by `OEMolBase::GetMaxAtomIdx` to return the contribution of each atom to the total surface area.

Note that whether or not hydrogens are included in these calculations is controlled by `OEArea.SetUseHydrogens` which is set to `false` by default.

To calculate the area of a molecule, `mol`:

```
OEArea area;  
float a = area.GetArea(mol);
```

To retrieve the atom contributions:

```
// declare an array of the correct size  
float * atomArea = new float[mol.GetMaxAtomIdx()];  
  
OEArea area;  
if (area.GetArea(mol, atomArea))  
{  
  unsigned int idx = 0;  
  for (OEIter<OEAtomBase> atom=mol.GetAtoms();atom;++atom)  
  {  
    idx = atom.GetIdx();  
    OEThrow.Info("atom %d %6.3f", idx, atomArea[idx]);  
  }  
}  
  
// clean up array  
delete [] atomArea;
```

### GetMethod

```
unsigned int GetMethod() const
```

Returns an `int` indicating the method that is used to model the area of the molecule. The two possible return values are `OEAreaMethod_Gaussian` and `OEAreaMethod_Discrete`.

### GetUseHydrogens

```
bool GetUseHydrogens() const
```

Returns a `bool` where `true` means explicit hydrogens are turned on and `false` means they are turned off for the area calculation.

### SetMethod

```
bool SetMethod(const unsigned int method)
```

Takes an `int` with the value of `OEAreaMethod_Gaussian` or `OEAreaMethod_Discrete`

### SetUseHydrogens

```
bool SetUseHydrogens(bool state)
```

This method takes a `bool` where `true` turns on explicit hydrogens for calculating area and `false` turn them off.

### 3.1.2 OEBind

```
class OEBind
```

This class represents *OEBind*. *OEBind* is used to calculate protein-ligand binding properties. The protein may either be passed in as an argument during construction or set with `OEBind.SetProtein`. The ligand is passed in as an argument to `OEBind.Bind` as well as an `OEBindResults` instance.

#### Constructors

```
OEBind()
OEBind(const OEBind &rhs)
OEBind(const OEChem::OEMolBase &protein)
```

Default and copy constructors. Besides the default and copy constructors, *OEBind* includes a constructor that takes the protein to be used in binding calculations. Constructing an *OEBind* object with the protein is identical to using the default constructor and then calling `OEBind.SetProtein`.

#### operator=

```
OEBind &operator=(const OEBind &rhs)
```

Assignment operator.

#### Bind

```
bool Bind(const OEChem::OEMolBase &ligand, OEBindResults &results)
OESystem::OEIterBase<OEBindResults> *Bind(const OEChem::OEMCMolBase &ligand)
```

Calculate full binding results, including Coulombic terms. For a single molecule, pass in the molecule and the `OEBindResults` class will be filled in. The following is a simple example of how *Bind* is used.

```
OEBind bind;
bind.SetProtein(protein);

OEBindResults results;
bind.Bind(ligand, results);

results.Print(OEThrow);
```

*Bind* runs through the steps shown in the `OEBind.SimpleBind` section, and also takes these addition steps.

```
float OriginalOuterDielectric = _zap.GetOuterDielectric();
_zap.SetOuterDielectric(_zap.GetInnerDielectric());

float *uniform_prot_pot = new float[patoms];
_zap.SetMolecule(_protein);
_zap.CalcAtomPotentials(uniform_prot_pot);

float *uniform_lig_pot = new float[latoms];
_zap.SetMolecule(ligand);
_zap.CalcAtomPotentials(uniform_lig_pot);
```

```
float *uniform_cmplx_pot = new float[catoms];
_zap.SetMolecule(cmplx);
_zap.CalcAtomPotentials(uniform_cmplx_pot);

resultsImpl->complexZapEnergy = 0.5f*(ecp+ecl);

// interaction terms
// protein - free
float cp = 0.0f;
for (OEIter<OEAtomBase> atom=_protein.GetAtoms();atom;++atom)
    cp+=uniform_prot_pot[atom->GetIdx()]*(float)atom->GetPartialCharge();
resultsImpl->unboundProteinCoulombEnergy = 0.5f*cp;

// ligand - free
float cl = 0.0f;
for (OEIter<OEAtomBase> atom=ligand.GetAtoms();atom;++atom)
    cl+=uniform_lig_pot[atom->GetIdx()]*(float)atom->GetPartialCharge();
resultsImpl->unboundLigandCoulombEnergy = 0.5f*cl;

// protein - bound
float ccp = 0.0f;
for (OEIter<OEAtomBase> atom=cmplx.GetAtoms(!OEIsLigandAtom());atom;++atom)
    ccp+=uniform_cmplx_pot[atom->GetIdx()]*(float)atom->GetPartialCharge();
resultsImpl->boundProteinCoulombEnergy = 0.5f*ccp;

// ligand - bound
float ccl = 0.0f;
for (OEIter<OEAtomBase> atom=cmplx.GetAtoms(OEIsLigandAtom());atom;++atom)
    ccl+=uniform_cmplx_pot[atom->GetIdx()]*(float)atom->GetPartialCharge();
resultsImpl->boundLigandCoulombEnergy = 0.5f*ccl;

resultsImpl->complexCoulombEnergy = 0.5f*(ccp+ccl);

resultsImpl->c_bind=OECalculateCoulombicBinding(_protein, ligand,
                                             _zap.GetInnerDielectric());

_zap.SetOuterDielectric(OriginalOuterDielectric);
```

## GetZap

```
OEZap &GetZap()
const OEZap &GetZap() const
```

Get a reference to the internal `OEZap` object contained in an `OEBind` instance. With the non-const version, you can adjust `OEZap` parameters that affect the `OEBind` calculation. For example, to use an `OEBind` instance with a specific grid spacing:

```
OEBind bind;
bind.GetZap().SetGridSpacing(0.6);
```

Note that some properties of the `OEZap` object are not controllable via this mechanism.

## SetProtein

```
bool SetProtein(const OEChem::OEMolBase &protein)
```

Set the protein molecule to be used by the `OEBind` object. Note that the protein should have radii and partial charges pre-calculated before passing to `OEBind.SetProtein`.

## SetZap

```
bool SetZap(const OEZap &zap)
```

Update the internal `OEZap` object. Note that some properties of the `OEZap` object are not controllable via this mechanism.

## SimpleBind

```
bool SimpleBind(const OEChem::OEMolBase &ligand, OESimpleBindResults &results)
OESystem::OEIterBase<OESimpleBindResults> *
    SimpleBind(const OEChem::OEMCMolBase &ligand)
```

Calculate simple binding results, without the Coulombic terms. See the discussion in *Binding Energies* about the differences between `OEBind.Bind` and `OEBind.SimpleBind`. See `OESimpleBindResults` to see all the values available.

The following is how *SimpleBind* is calculated. Note that the *\_protein* has already been set and the *ligand* is passed in as an argument.

```
OEMol cmplx(_protein);
OEAddMols(cmplx, ligand);

_zap.SetFocusTarget(ligand);

float *prot_pot = new float[patoms];
_zap.SetMolecule(_protein);
_zap.CalcAtomPotentials(prot_pot);

float *lig_pot = new float[latoms];
_zap.SetMolecule(ligand);
_zap.CalcAtomPotentials(lig_pot);

float *cmplx_pot = new float[catoms];
_zap.SetMolecule(cmplx);
_zap.CalcAtomPotentials(cmplx_pot);

// protein - free
float ep=0.0f;
for (OEIter<OEAtomBase> atom=_protein.GetAtoms();atom;++atom)
    ep+=prot_pot[atom->GetIdx()]* (float) atom->GetPartialCharge();
resultsImpl->unboundProteinZapEnergy = 0.5f*ep;

// ligand - free
float el=0.0f;
for (OEIter<OEAtomBase> atom=ligand.GetAtoms();atom;++atom)
    el+=lig_pot[atom->GetIdx()]* (float) atom->GetPartialCharge();
```

```
resultsImpl->unboundLigandZapEnergy = 0.5f*el;

// protein - bound
float ecp = 0.0f;
for (OEIter<OEAtomBase> atom=cplx.GetAtoms(!OEIsLigandAtom()); atom;++atom)
    ecp+=cplx_pot[atom->GetIdx()]* (float) atom->GetPartialCharge();
resultsImpl->boundProteinZapEnergy = 0.5f*ecp;

// ligand - bound
float ecl = 0.0f;
for (OEIter<OEAtomBase> atom=cplx.GetAtoms(OEIsLigandAtom()); atom;++atom)
    ecl+=cplx_pot[atom->GetIdx()]* (float) atom->GetPartialCharge();
resultsImpl->boundLigandZapEnergy = 0.5f*ecl;
```

Additionally, both the *SimpleBind* and *OEBind.Bind* methods calculate area contributions using the following algorithm. The constant *AREA\_TO\_ENERGY* is set to 0.0423 kT per square Angstrom.

```
// area analysis
OEArea area;
x=area.GetArea(ligand);
results->unboundLigandArea = x;
results->unboundLigandAreaEnergy = x*AREA_TO_ENERGY;

x=area.GetArea(protein);
results->unboundProteinArea = x;
results->unboundProteinAreaEnergy = x*AREA_TO_ENERGY;

x=area.GetArea(cplx);
results->complexArea = x;
results->complexAreaEnergy = x*AREA_TO_ENERGY;

float *atom_area = new float [cplx.GetMaxAtomIdx()];
area.GetArea(cplx, atom_area);

float bpa=0.0f;
for (OEIter<OEAtomBase> atom=cplx.GetAtoms(!OEIsLigandAtom()); atom;++atom)
    bpa += atom_area[atom->GetIdx()];
results->boundProteinArea = bpa;
results->boundProteinAreaEnergy = bpa*AREA_TO_ENERGY;

float bla=0.0f;
for (OEIter<OEAtomBase> atom=cplx.GetAtoms(OEIsLigandAtom()); atom;++atom)
    bla += atom_area[atom->GetIdx()];
results->boundLigandArea = bla;
results->boundLigandAreaEnergy = bla*AREA_TO_ENERGY;
```

### 3.1.3 OEBindResults

```
class OEBindResults : public OESimpleBindResults
```

This class represents *OEBindResults*.

The following methods are publicly inherited from *OESimpleBindResults*:

operator= GetBindingEnergy GetBoundLigandArea GetBoundLigandAreaEnergy GetBoundLigandZapEnergy GetBoundProteinArea GetBoundProteinAreaEnergy GetBoundProteinZapEnergy	GetBuriedArea GetBuriedAreaEnergy GetComplexArea GetComplexAreaEnergy GetComplexZapEnergy GetConf GetUnboundLigandArea GetUnboundLigandAreaEnergy	GetUnboundLigandZapEnergy GetUnboundProteinArea GetUnboundProteinAreaEnergy GetUnboundProteinZapEnergy GetZapEnergy Print
--	--	--

## Constructors

```
OEBindResults()
OEBindResults(const OEBindResults &)
```

Default and copy constructors.

## operator=

```
OEBindResults &operator=(const OEBindResults &)
```

Assignment operator.

## GetAnalyticCoulombicBindingEnergy

```
float GetAnalyticCoulombicBindingEnergy() const
```

This returns the value of the analytic binding energy for the ligand and protein that have been set to the OEBind object. This is equivalent to calling `OECalculateCoulombicBinding(protein, ligand, dielectric)` where the same ligand and protein and inner dielectric are the arguments.

## GetBoundLigandCoulombEnergy

```
float GetBoundLigandCoulombEnergy() const
```

Returns the grid-based coulomb energy for the ligand in the bound state, including self-interaction. Specially, it returns the `boundLigandCoulombEnergy` variable shown in the `OEBind.Bind` API section.

## GetBoundProteinCoulombEnergy

```
float GetBoundProteinCoulombEnergy() const
```

Returns the grid-based coulomb energy for the protein in the bound state, including self-interaction. Specially, it returns the `boundProteinCoulombEnergy` variable shown in the `OEBind.Bind` API section.

## GetComplexCoulombEnergy

```
float GetComplexCoulombEnergy() const
```

Returns the grid-based coulomb energy for the complex, including self-interaction. Specially, it returns the `complexCoulombEnergy` variable shown in the `OEBind.Bind` API section. This is equivalent to the sum of the results returned by `OEBindResults.GetBoundLigandCoulombEnergy` and `OEBindResults.GetBoundProteinCoulombEnergy`.

### GetCoulombEnergy

`float` `GetCoulombEnergy()` `const`

Returns the difference in Coulomb energy between the complex and the unbound ligand and protein. This is equivalent to `OEBindResults.GetComplexCoulombEnergy - OEBindResults.GetUnboundProteinCoulombEnergy - OEBindResults.GetUnboundLigandCoulombEnergy`.

### GetDesolvationEnergy

`float` `GetDesolvationEnergy()` `const`

This is equivalent to `OESimpleBindResults.GetZapEnergy - OEBindResults.GetCoulombEnergy`.

### GetLigandDesolvationEnergy

`float` `GetLigandDesolvationEnergy()` `const`

This is equivalent to `OESimpleBindResults.GetBoundLigandZapEnergy - OEBindResults.GetBoundLigandCoulombEnergy - OESimpleBindResults.GetUnboundLigandZapEnergy + OEBindResults.GetUnboundLigandCoulombEnergy`.

### GetProteinDesolvationEnergy

`float` `GetProteinDesolvationEnergy()` `const`

This is equivalent to `OESimpleBindResults.GetBoundProteinZapEnergy - OEBindResults.GetBoundProteinCoulombEnergy - OESimpleBindResults.GetUnboundProteinZapEnergy + OEBindResults.GetUnboundProteinCoulombEnergy`.

### GetUnboundLigandCoulombEnergy

`float` `GetUnboundLigandCoulombEnergy()` `const`

Returns the grid-based coulomb energy for the ligand in the unbound state, including self-interaction. Specially, it returns the `unboundLigandCoulombEnergy` variable shown in the `OEBind.Bind` API section.

### GetUnboundProteinCoulombEnergy

`float` `GetUnboundProteinCoulombEnergy()` `const`

Returns the grid-based coulomb energy for the protein in the unbound state, including self-interaction. Specially, it returns the `unboundProteinCoulombEnergy` variable shown in the `OEBind.Bind` API section.

## Print

```
void Print(OEPlatform::oeostream &ofs) const
void Print(OESystem::OSErrorHandler &log) const
```

Prints out all of the available data to the oeostream of the OSErrHandle passed in.

## 3.1.4 OEET

```
class OEET
```

This class represents *OEET*, which is used to calculate electrostatic similarity.

### Constructors

```
OEET(const OEET &rhs)
OEET(float dielectric=80.0f)
```

Default and copy constructors.

### operator=

```
OEET &operator=(const OEET &rhs)
```

Assignment operator

### GetDielectric

```
float GetDielectric() const
```

Returns the setting for the outer dielectric constant.

### GetGridBuffer

```
float GetGridBuffer() const
```

Returns the setting for the amount of space between the molecule and the edge of the grid.

### GetGridSpacing

```
float GetGridSpacing() const
```

Returns the setting for the grid spacing.

### GetInnerMask

`float` GetInnerMask() `const`

Returns the value for the inner mask.

### GetJoin

`bool` GetJoin() `const`

Returns the setting for whether or not to join the molecules for masking.

### GetOuterMask

`float` GetOuterMask() `const`

Returns the value for the outer mask.

### GetSaltConcentration

`float` GetSaltConcentration() `const`

Returns the setting `for` the salt concentration

### SetDielectric

`void` SetDielectric(`float` d)

Sets the outer dielectric constant

### SetGridBuffer

`void` SetGridBuffer(`float` f)

Sets the grid buffer, or boundary spacing, which is the amount of distance between the molecule and the edge of the grid. The default value is 6.0.

### SetGridSpacing

`void` SetGridSpacing(`float` f)

Sets the grid spacing. The default value is 0.75.

## SetInnerMask

```
void SetInnerMask(float f)
```

Sets the inner mask for electrostatic comparison. The default value is set to 0.05. The inner mask is used to mask out the inner part of a molecule. The inner part of the molecule is masked out so that the important regions surrounding the molecule dominate the electrostatic comparison.

## SetJoin

```
void SetJoin(bool j)
```

Turns molecular joining on(`true`) or off(`false`). The default value is `true`, which uses the combine reference molecule and trial molecule when applying the inner and outer mask.

## SetOuterMask

```
void SetOuterMask(float f)
```

Sets the outer mask for electrostatic comparison. The default value is set to 0.0005. The outer mask is used to mask out regions far from the molecule.

## SetRefMol

```
bool SetRefMol(const OEChem::OEMolBase &mol)
```

Sets the reference molecule.

## SetSaltConcentration

```
void SetSaltConcentration(float conc)
```

Sets the salt concentration. The default value is 0.04.

## Tanimoto

```
float Tanimoto(const OEChem::OEMolBase &mol)
```

Returns the electrostatic Tanimoto between the reference molecule that has been set and the trial molecule that is passed in as an argument.

## 3.1.5 OESimpleBindResults

```
class OESimpleBindResults
```

This class represents *OESimpleBindResults*. The Zap calculations used for the SimpleBindResults are performed with the 2 separate dielectric constants, the internal for the molecule and the external for the solvent.

The following classes derive from this class:

- `OEBindResults`

### Constructors

```
OESimpleBindResults()  
OESimpleBindResults(const OESimpleBindResults &)
```

Default and copy constructors.

### operator=

```
OESimpleBindResults &operator=(const OESimpleBindResults &)
```

Assignment operator

### GetBindingEnergy

```
float GetBindingEnergy() const
```

Returns the binding energy of the ligand and protein. Equivalent to `OESimpleBindResults.GetBuriedAreaEnergy + OESimpleBindResults.GetZapEnergy`.

### GetBoundLigandArea

```
float GetBoundLigandArea() const
```

Returns the `boundLigandArea` variable shown in the `OEBind.SimpleBind` API section. The bound ligand area is the area of the ligand exposed to the solvent while in the bound state.

### GetBoundLigandAreaEnergy

```
float GetBoundLigandAreaEnergy() const
```

Returns the `boundLigandAreaEnergy` variable shown in the `OEBind.SimpleBind` API section. This is the energy associated with the bound ligand area.

### GetBoundLigandZapEnergy

```
float GetBoundLigandZapEnergy() const
```

Returns the `boundLigandZapEnergy` variable shown in the `OEBind.SimpleBind` API section. This is the ligand grid energy obtained from a Zap calculation of the complex.

### GetBoundProteinArea

```
float GetBoundProteinArea() const
```

Returns the `boundProteinArea` variable shown in the `OEBind.SimpleBind` API section. This is the area of the protein exposed to the solvent while in the bound state.

### GetBoundProteinAreaEnergy

```
float GetBoundProteinAreaEnergy() const
```

Returns the `boundProteinAreaEnergy` variable shown in the `OEBind.SimpleBind` API section. This is the energy associated with the bound protein area.

### GetBoundProteinZapEnergy

```
float GetBoundProteinZapEnergy() const
```

Returns the `boundProteinZapEnergy` variable shown in the `OEBind.SimpleBind` API section. This is the protein grid energy obtained from a Zap calculation of the complex.

### GetBuriedArea

```
float GetBuriedArea() const
```

Returns the amount of area that has become buried as a result of binding. Equivalent to `OESimpleBindResults.GetUnboundProteinArea + OESimpleBindResults.GetUnboundLigandArea - OESimpleBindResults.GetComplexArea`.

### GetBuriedAreaEnergy

```
float GetBuriedAreaEnergy() const
```

Returns the energy penalty associated with the buried area of the complex. Equivalent to `OEComplexAreaEnergy - OESimpleBindResults.GetUnboundProteinAreaEnergy - OESimpleBindResults.GetUnboundLigandAreaEnergy`.

### GetComplexArea

```
float GetComplexArea() const
```

Returns the `complexArea` variable shown in the `OEBind.SimpleBind` API section. This is the solvent accessible surface area of the complex.

### GetComplexAreaEnergy

**float** GetComplexAreaEnergy() **const**

Returns the `complexAreaEnergy` variable shown in the `OEBind.SimpleBind` API section. This is the energy of the solvent accessible surface area of the complex.

### GetComplexZapEnergy

**float** GetComplexZapEnergy() **const**

Returns the grid energy of the complex. This is equivalent to `OESimpleBindResults.GetBoundProteinZapEnergy + OESimpleBindResults.GetBoundLigandZapEnergy`.

### GetConf

`OEChem::OEConfBase *GetConf()`  
**const** `OEChem::OEConfBase *GetConf()` **const**

Returns a pointer to the active conformer.

### GetUnboundLigandArea

**float** GetUnboundLigandArea() **const**

Returns the `unboundLigandArea` variable shown in the `OEBind.SimpleBind` API section. This is the solvent accessible surface area of the unbound ligand.

### GetUnboundLigandAreaEnergy

**float** GetUnboundLigandAreaEnergy() **const**

Returns the `unboundLigandAreaEnergy` variable shown in the `OEBind.SimpleBind` API section. This is the energy of the solvent accessible surface area of the unbound ligand.

### GetUnboundLigandZapEnergy

**float** GetUnboundLigandZapEnergy() **const**

Returns the `unboundProteinZapEnergy` variable shown in the `OEBind.SimpleBind` API section. This is the grid energy of the unbound protein obtained from a Zap calculation.

### GetUnboundProteinArea

**float** GetUnboundProteinArea() **const**

Returns the `unboundProteinArea` variable shown in the `OEBind.SimpleBind` API section. This is the solvent accessible surface area of the unbound protein.

### GetUnboundProteinAreaEnergy

```
float GetUnboundProteinAreaEnergy() const
```

Returns the `unboundProteinAreaEnergy` variable shown in the `OEBind.SimpleBind` API section. This is the energy of the solvent accessible surface area of the unbound protein.

### GetUnboundProteinZapEnergy

```
float GetUnboundProteinZapEnergy() const
```

Returns the `unboundProteinZapEnergy` variable shown in the `OEBind.SimpleBind` API section. This is the grid energy of the unbound protein obtained from a Zap calculation.

### GetZapEnergy

```
float GetZapEnergy() const
```

Returns the difference in grid energy between the complex and the unbound ligand and protein. This is equivalent to `OESimpleBindResults.GetComplexZapEnergy - OESimpleBindResults.GetUnboundProteinZapEnergy - OESimpleBindResults.GetUnboundLigandZapEnergy`.

### Print

```
void Print(OEPlatform::oeostream &ofs) const
void Print(OESystem::OSErrorHandler &log) const
```

Prints out all of the available data to the oestream of the `OSErrorHandle` passed in.

## 3.1.6 OEZap

```
class OEZap
```

This class represents *OEZap*.

### Constructors

```
OEZap()
OEZap(const OEZap &rhs)
```

Default and copy constructors.

### operator=

```
OEZap &operator=(const OEZap &rhs)
```

Assignment operator.

### CalcAtomPotentials

```
bool CalcAtomPotentials(float *pot, bool no_grid=false)
```

Calculates the atom potentials and fills the `float` array with their values. The `float` array should be of length `N`, where `N` is the number of atoms.

### CalcForces

```
bool CalcForces(float *forces)
```

Calculates the forces on the atoms and fills the `float` array with the values of their x, y, and z components. The `float` array should be of length `3N`, where `N` is the number of atoms.

### CalcPotentialGrid

```
bool CalcPotentialGrid(OESystem::OEScalarGrid &grid)
```

Calculates the potential grid and will the `OEScalarGrid` with the values.

### CalcSolvationEnergy

```
float CalcSolvationEnergy()
```

Returns the solvation energy of the molecule that has been set. This result does not include an area term.

### ClearFocusTarget

```
void ClearFocusTarget()
```

Clears the target setting for focussing.

### GetBoundarySpacing

```
float GetBoundarySpacing() const
```

Returns the boundary spacing setting, which is the amount of distance between the molecule and the edge of the grid. The default setting is 4.0.

### GetDielectricModel

```
unsigned int GetDielectricModel() const
```

Returns an `int` representing the dielectric model being used. The two available models are `OEZapDielectricModel_Gaussian` and `OEZapDielectricModel_Molecular`.

### GetError

```
float GetError() const
```

Returns the error setting. See `OEZap.SetError`.

### GetFocusTarget

```
const OEChem::OEMolBase *GetFocusTarget() const
```

Returns a pointer to the molecule that has been set as the focus target.

### GetGridSpacing

```
float GetGridSpacing() const
```

Returns the setting for the grid spacing. The default value is 0.5.

### GetInnerDielectric

```
float GetInnerDielectric() const
```

Returns the setting for the internal dielectric constant. The default value is 1.0.

### GetIterations

```
int GetIterations() const
```

Returns the maximum number of iterations allowed for the zap calculation. The default value is 10.

### GetMolecule

```
const OEChem::OEMolBase *GetMolecule() const
```

Returns a pointer to the molecule that has been set to Zap.

### GetOuterDielectric

`float` GetOuterDielectric() `const`

Returns the setting for the outer dielectric constant. The default value is 80.0.

### GetProbeRadius

`float` GetProbeRadius() `const`

Returns the setting for the probe radius. The default value is 1.4.

### GetVerbose

`bool` GetVerbose() `const`

Returns the setting for very low-level verbosity. The default value is false.

### GetSaltConcentration

`float` GetSaltConcentration() `const`

Returns the setting for the salt concentration. The default value is 0.0.

### IsFocusTargetSet

`bool` IsFocusTargetSet()

Returns a `bool` for whether the focus target is currently set. See the section of Focussing for more information.

### SetBoundarySpacing

`bool` SetBoundarySpacing(`float` spacing)

Sets the boundary spacing, which is the amount of distance between the molecule and the edge of the grid. The default setting is 4.0.

### SetDielectricModel

`bool` SetDielectricModel(`unsigned int` model)

Sets the dielectric model. The two available models are `OEZapDielectricModel_Gaussian` and `OEZapDielectricModel_Molecular`. The default is the Gaussian model.

### SetError

```
bool SetError(float zaperr)
```

Rather than setting the grid spacing, the acceptable error may be set instead using this method. The default value is set to 0.0, which means that the grid spacing is set explicitly.

### SetFocusTarget

```
bool SetFocusTarget(const OEChem::OEMolBase &mol)
```

Sets the focus target. See the section of *Focussing* for more information.

### SetGridSpacing

```
bool SetGridSpacing(float spacing)
```

Sets the grid spacing. The default value is 0.5.

### SetInnerDielectric

```
bool SetInnerDielectric(float epsin)
```

Sets the internal dielectric constant. The default value is 1.0.

### SetIterations

```
bool SetIterations(int iters)
```

Sets the maximum number of iterations allowed for the zap calculation. The default value is 10

### SetMolecule

```
bool SetMolecule(const OEChem::OEMolBase &mol)
```

Sets the molecule to Zap. Partial charges, 3-D coordinates, and radii need to be set to the molecule before the molecule is set to Zap.

### SetOuterDielectric

```
bool SetOuterDielectric(float epsout)
```

Sets the external dielectric constant. The default value is 80.0.

### SetProbeRadius

`bool SetProbeRadius(float radius)`

Sets the probe radius, the default value is 1.4.

### SetSaltConcentration

`bool SetSaltConcentration(float conc)`

Sets the salt concentration. The default value is 0.0.

### SetVerbose

`bool SetVerbose(bool verbose)`

Sets very low-level verbosity. The default value is false.

## 3.2 OEPB Constants

### 3.2.1 OEAreaMethod

This namespace contains constants.

#### Undefined

This value represents an unset OEAreaMethod.

#### Gaussian

This method uses Gaussians to compute area and is faster and less sensitive to rotation and translation on the grid.

#### Discrete

This method uses a hard sphere model to compute area.

#### Default

The default is set to Gaussian.

### 3.2.2 OEZapDielectricModel

This namespace contains constants.

## Gaussian

This model uses Gaussians to represent atoms and is faster and less sensitive to rotation and translation on the grid.

## Molecular

This model uses hard spheres to represent atoms.

## Default

The default model is Gaussian.

## 3.3 OEPB Functions

### 3.3.1 OECalculateCoulombicBinding

```
float OECalculateCoulombicBinding(const OEChem::OEMolBase &protein,
                                  const OEChem::OEMolBase &ligand,
                                  float dielectric)
```

Returns the analytic binding energy for the ligand and protein in the specified dielectric using Coulomb's Law.

### 3.3.2 OECoulombicAtomPotentials

```
bool OECoulombicAtomPotentials(const OEChem::OEMolBase &mol, float D,
                                float *apot)
```

Calculates the Coulombic atom potentials with the dielectric constant D and fills the float array `apot` with the values. `apot` should be of length N, where N is the number of atoms.

### 3.3.3 OECoulombicSelfEnergy

```
float OECoulombicSelfEnergy(const OEChem::OEMolBase &mol, float D)
```

Returns the integral energy of a molecule from its partial charges in universal dielectric D using Coulomb's Law.

### 3.3.4 OEMakeETGrid

```
bool OEMakeETGrid(OESystem::OEScalarGrid &grid, const OEChem::OEMolBase &mol,
                  const OEET &et)
```

Calculates the electrostatic Tanimoto grid for the `mol` passed in and the reference molecule that needs to be already set to the `OEET` instance.

### 3.3.5 O EZapBind

```
bool O EZapBind(const OEChem::OEMolBase &protein,  
               const OEChem::OEMolBase &ligand, OEBindResults &results)  
bool O EZapBind(const OEChem::OEMolBase &protein,  
               const OEChem::OEMolBase &ligand, OESimpleBindResults &results)
```

Convenience functions for obtaining `OEBindResults` or `OESimpleBindResults` of a ligand and protein.

### 3.3.6 O EZapGetArch

```
const char *O EZapGetArch()
```

Returns the architecture on which the release was built

### 3.3.7 O EZapGetLicensee

```
bool O EZapGetLicensee(std::string &licensee)
```

Fills the parameter string `&licensee` with the licensee of the license file being used

### 3.3.8 O EZapGetPlatform

```
const char *O EZapGetPlatform()
```

Returns the platform on which the release was built

### 3.3.9 O EZapGetRelease

```
const char *O EZapGetRelease()
```

Returns the versions number as a `const char*` in major.minor.bugfix form

### 3.3.10 O EZapGetSite

```
bool O EZapGetSite(std::string &site)
```

Fills the parameter string `&site` with the site on the license file being used

### 3.3.11 O EZapGetVersion

```
unsigned int O EZapGetVersion()
```

Returns the build date of the release

### 3.3.12 OZapIsLicensed

```
bool OZapIsLicensed(const char *feature=0, unsigned int *expdate=0)
```

Returns a `bool` indicating if a valid license for ZapTK has been set



# RELEASE NOTES

## 4.1 ZapTK 2.1.2

### 4.1.1 New features

- Added new methods `OEZap.GetVerbose` and `OEZap.SetVerbose` to enable Zap to print out and abundance of low-level information.

### 4.1.2 Bug fixes

- Fixed memory leak that occurred with multiple calls to `OEArea.GetArea`
- `OEMakeETGrid` now properly wrapped in Python.

## 4.2 ZapTK 2.1.1

### 4.2.1 Bug fixes

- `OEZap.GetMolecule` and `OEZap.GetFocusTarget` now return a pointer instead of a reference. This is a breaking change and any code that calls these methods will have to be updated. These methods will return a null pointer if an acceptable molecule has not been set for them.
- A warning has been added for molecules passed into ZapTK that do not return 3 when `GetDimension` is called. The dimension must be 3 for all molecules passed into ZapTK. The dimension is set automatically when using `OEReadMolecule` but a warning will be thrown and the molecule will not be accepted by ZapTK if the molecule has been made from scratch in the toolkits and `SetDimension(3)` has not been called on it.

### 4.2.2 Minor bug fixes

- Fixed a memory bug related to extremely large files on Windows.

## 4.3 ZapTK 2.1.0

### 4.3.1 New features

- Internal defaults have been changed for the `OEZap` object. The internal dielectric now has a default of 1.0, the boundary spacing now has a default of 4.0, and the default number of iterations has been set to 10. The change from 2.0 to 1.0 for the internal dielectric led to dramatically improved results when we performed tests with the Rizzo set, with the RMS error for vacuum-water transfer energy decreasing by more than half.
- Large manual update with complete examples and API documentation.
- `OEZap.IsFocusTargetSet` has been added, which returns a `bool` which indicates whether the focus target has been set.
- New examples for focussing and forces have been added
- All examples have been ported across all three languages (C++, Java, Python)
- Examples assign MMFF94 charges by default

### 4.3.2 Bug fixes

- Internal and external dielectrics are now properly reset each time the `OEBind` instance is used.
- Internal error checking has been added so that no internal calculations will be attempted if `OEZap.SetMolecule` fails. (`OEZap.SetMolecule` most often fails because all charges are equal to zero.) Calls to methods that would attempt to run internal calculations (e.g. `OEZap.CalcSolvationEnergy`) will return zero or false.
- Internal summations in `OEBind` use doubles now instead of floats for appropriate precision
- Charges are properly assigned to both the `refmol` and `fitmol` in the `calc_et.cpp` example.
- Examples perform error checking when opening files

## 4.4 ZapTK 2.0.0

### 4.4.1 New features

- This is the first official release of the C++ version of ZapTK. Note that this is a significant change in the API as we migrate from the C API to the new C++ API built on top of OEChem.

## 4.5 Indices and tables

- *Index*
- *Search Page*

# INDEX

## B

Bind  
    OEPB::OEBind, 23  
bind.py  
    Example Code, 15

## C

calc\_et.py  
    Example Code, 19  
CalcAtomPotentials  
    OEPB::OEZap, 36  
CalcForces  
    OEPB::OEZap, 36  
CalcPotentialGrid  
    OEPB::OEZap, 36  
CalcSolvationEnergy  
    OEPB::OEZap, 36  
ClearFocusTarget  
    OEPB::OEZap, 36  
Constructors  
    OEPB::OEArea, 21  
    OEPB::OEBind, 23  
    OEPB::OEBindResults, 27  
    OEPB::OEET, 29  
    OEPB::OESimpleBindResults, 32  
    OEPB::OEZap, 35

## E

Example Code  
    bind.py, 15  
    calc\_et.py, 19  
    transfer.py, 14  
    zap\_atompot.py, 10  
    zap\_focussing.py, 16  
    zap\_forces.py, 18  
    zap\_grid1.py, 5  
    zap\_grid2.py, 6  
    zap\_grid3.py, 8  
    zap\_solv1.py, 13

## G

GetAnalyticCoulombicBindingEnergy  
    OEPB::OEBindResults, 27  
GetArea  
    OEPB::OEArea, 21  
GetBindingEnergy  
    OEPB::OESimpleBindResults, 32  
GetBoundarySpacing  
    OEPB::OEZap, 36  
GetBoundLigandArea  
    OEPB::OESimpleBindResults, 32  
GetBoundLigandAreaEnergy  
    OEPB::OESimpleBindResults, 32  
GetBoundLigandCoulombEnergy  
    OEPB::OEBindResults, 27  
GetBoundLigandZapEnergy  
    OEPB::OESimpleBindResults, 32  
GetBoundProteinArea  
    OEPB::OESimpleBindResults, 33  
GetBoundProteinAreaEnergy  
    OEPB::OESimpleBindResults, 33  
GetBoundProteinCoulombEnergy  
    OEPB::OEBindResults, 27  
GetBoundProteinZapEnergy  
    OEPB::OESimpleBindResults, 33  
GetBuriedArea  
    OEPB::OESimpleBindResults, 33  
GetBuriedAreaEnergy  
    OEPB::OESimpleBindResults, 33  
GetComplexArea  
    OEPB::OESimpleBindResults, 33  
GetComplexAreaEnergy  
    OEPB::OESimpleBindResults, 34  
GetComplexCoulombEnergy  
    OEPB::OEBindResults, 27  
GetComplexZapEnergy  
    OEPB::OESimpleBindResults, 34  
GetConf  
    OEPB::OESimpleBindResults, 34  
GetCoulombEnergy  
    OEPB::OEBindResults, 28

- GetDesolvationEnergy
    - OEPPB::OEBindResults, 28
  - GetDielectric
    - OEPPB::OEET, 29
  - GetDielectricModel
    - OEPPB::OEZap, 37
  - GetError
    - OEPPB::OEZap, 37
  - GetFocusTarget
    - OEPPB::OEZap, 37
  - GetGridBuffer
    - OEPPB::OEET, 29
  - GetGridSpacing
    - OEPPB::OEET, 29
    - OEPPB::OEZap, 37
  - GetInnerDielectric
    - OEPPB::OEZap, 37
  - GetInnerMask
    - OEPPB::OEET, 30
  - GetIterations
    - OEPPB::OEZap, 37
  - GetJoin
    - OEPPB::OEET, 30
  - GetLigandDesolvationEnergy
    - OEPPB::OEBindResults, 28
  - GetMethod
    - OEPPB::OEArea, 22
  - GetMolecule
    - OEPPB::OEZap, 37
  - GetOuterDielectric
    - OEPPB::OEZap, 38
  - GetOuterMask
    - OEPPB::OEET, 30
  - GetProbeRadius
    - OEPPB::OEZap, 38
  - GetProteinDesolvationEnergy
    - OEPPB::OEBindResults, 28
  - GetSaltConcentration
    - OEPPB::OEET, 30
    - OEPPB::OEZap, 38
  - GetUnboundLigandArea
    - OEPPB::OESimpleBindResults, 34
  - GetUnboundLigandAreaEnergy
    - OEPPB::OESimpleBindResults, 34
  - GetUnboundLigandCoulombEnergy
    - OEPPB::OEBindResults, 28
  - GetUnboundLigandZapEnergy
    - OEPPB::OESimpleBindResults, 34
  - GetUnboundProteinArea
    - OEPPB::OESimpleBindResults, 34
  - GetUnboundProteinAreaEnergy
    - OEPPB::OESimpleBindResults, 35
  - GetUnboundProteinCoulombEnergy
    - OEPPB::OEBindResults, 28
  - GetUnboundProteinZapEnergy
    - OEPPB::OESimpleBindResults, 35
  - GetUseHydrogens
    - OEPPB::OEArea, 22
  - GetVerbose
    - OEPPB::OEZap, 38
  - GetZap
    - OEPPB::OEBind, 24
  - GetZapEnergy
    - OEPPB::OESimpleBindResults, 35
- I**
- IsFocusTargetSet
    - OEPPB::OEZap, 38
- O**
- OEPPB::OEArea, 21
    - Constructors, 21
    - GetArea, 21
    - GetMethod, 22
    - GetUseHydrogens, 22
    - operator=, 21
    - SetMethod, 22
    - SetUseHydrogens, 22
  - OEPPB::OEAreaMethod, 40
    - OEPPB::OEAreaMethod::Default, 40
    - OEPPB::OEAreaMethod::Discrete, 40
    - OEPPB::OEAreaMethod::Gaussian, 40
    - OEPPB::OEAreaMethod::Undefined, 40
  - OEPPB::OEBind, 23
    - Bind, 23
    - Constructors, 23
    - GetZap, 24
    - operator=, 23
    - SetProtein, 25
    - SetZap, 25
    - SimpleBind, 25
  - OEPPB::OEBindResults, 26
    - Constructors, 27
    - GetAnalyticCoulombicBindingEnergy, 27
    - GetBoundLigandCoulombEnergy, 27
    - GetBoundProteinCoulombEnergy, 27
    - GetComplexCoulombEnergy, 27
    - GetCoulombEnergy, 28
    - GetDesolvationEnergy, 28
    - GetLigandDesolvationEnergy, 28
    - GetProteinDesolvationEnergy, 28
    - GetUnboundLigandCoulombEnergy, 28
    - GetUnboundProteinCoulombEnergy, 28
    - operator=, 27
    - Print, 29
  - OEPPB::OECalculateCoulombicBinding, 41
  - OEPPB::OECoulombicAtomPotentials, 41
  - OEPPB::OECoulombicSelfEnergy, 41

- OEPB::OEET, 29
    - Constructors, 29
    - GetDielectric, 29
    - GetGridBuffer, 29
    - GetGridSpacing, 29
    - GetInnerMask, 30
    - GetJoin, 30
    - GetOuterMask, 30
    - GetSaltConcentration, 30
    - operator=, 29
    - SetDielectric, 30
    - SetGridBuffer, 30
    - SetGridSpacing, 30
    - SetInnerMask, 31
    - SetJoin, 31
    - SetOuterMask, 31
    - SetRefMol, 31
    - SetSaltConcentration, 31
    - Tanimoto, 31
  - OEPB::OEMakeETGrid, 41
  - OEPB::OESimpleBindResults, 31
    - Constructors, 32
    - GetBindingEnergy, 32
    - GetBoundLigandArea, 32
    - GetBoundLigandAreaEnergy, 32
    - GetBoundLigandZapEnergy, 32
    - GetBoundProteinArea, 33
    - GetBoundProteinAreaEnergy, 33
    - GetBoundProteinZapEnergy, 33
    - GetBuriedArea, 33
    - GetBuriedAreaEnergy, 33
    - GetComplexArea, 33
    - GetComplexAreaEnergy, 34
    - GetComplexZapEnergy, 34
    - GetConf, 34
    - GetUnboundLigandArea, 34
    - GetUnboundLigandAreaEnergy, 34
    - GetUnboundLigandZapEnergy, 34
    - GetUnboundProteinArea, 34
    - GetUnboundProteinAreaEnergy, 35
    - GetUnboundProteinZapEnergy, 35
    - GetZapEnergy, 35
    - operator=, 32
    - Print, 35
  - OEPB::OEZap, 35
    - CalcAtomPotentials, 36
    - CalcForces, 36
    - CalcPotentialGrid, 36
    - CalcSolvationEnergy, 36
    - ClearFocusTarget, 36
    - Constructors, 35
    - GetBoundarySpacing, 36
    - GetDielectricModel, 37
    - GetError, 37
    - GetFocusTarget, 37
    - GetGridSpacing, 37
    - GetInnerDielectric, 37
    - GetIterations, 37
    - GetMolecule, 37
    - GetOuterDielectric, 38
    - GetProbeRadius, 38
    - GetSaltConcentration, 38
    - GetVerbose, 38
    - IsFocusTargetSet, 38
    - operator=, 36
    - SetBoundarySpacing, 38
    - SetDielectricModel, 38
    - SetError, 39
    - SetFocusTarget, 39
    - SetGridSpacing, 39
    - SetInnerDielectric, 39
    - SetIterations, 39
    - SetMolecule, 39
    - SetOuterDielectric, 39
    - SetProbeRadius, 40
    - SetSaltConcentration, 40
    - SetVerbose, 40
  - OEPB::OEZapBind, 42
  - OEPB::OEZapDielectricModel, 40
    - OEPB::OEZapDielectricModel::Default, 41
    - OEPB::OEZapDielectricModel::Gaussian, 41
    - OEPB::OEZapDielectricModel::Molecular, 41
  - OEPB::OEZapGetArch, 42
  - OEPB::OEZapGetLicensee, 42
  - OEPB::OEZapGetPlatform, 42
  - OEPB::OEZapGetRelease, 42
  - OEPB::OEZapGetSite, 42
  - OEPB::OEZapGetVersion, 42
  - OEPB::OEZapIsLicensed, 43
  - operator=
    - OEPB::OEArea, 21
    - OEPB::OEBind, 23
    - OEPB::OEBindResults, 27
    - OEPB::OEET, 29
    - OEPB::OESimpleBindResults, 32
    - OEPB::OEZap, 36
- ## P
- Print
    - OEPB::OEBindResults, 29
    - OEPB::OESimpleBindResults, 35
- ## S
- SetBoundarySpacing
    - OEPB::OEZap, 38
  - SetDielectric
    - OEPB::OEET, 30
  - SetDielectricModel

OEPB::OEZap, 38  
SetError  
    OEPB::OEZap, 39  
SetFocusTarget  
    OEPB::OEZap, 39  
SetGridBuffer  
    OEPB::OEET, 30  
SetGridSpacing  
    OEPB::OEET, 30  
    OEPB::OEZap, 39  
SetInnerDielectric  
    OEPB::OEZap, 39  
SetInnerMask  
    OEPB::OEET, 31  
SetIterations  
    OEPB::OEZap, 39  
SetJoin  
    OEPB::OEET, 31  
SetMethod  
    OEPB::OEArea, 22  
SetMolecule  
    OEPB::OEZap, 39  
SetOuterDielectric  
    OEPB::OEZap, 39  
SetOuterMask  
    OEPB::OEET, 31  
SetProbeRadius  
    OEPB::OEZap, 40  
SetProtein  
    OEPB::OEBind, 25  
SetRefMol  
    OEPB::OEET, 31  
SetSaltConcentration  
    OEPB::OEET, 31  
    OEPB::OEZap, 40  
SetUseHydrogens  
    OEPB::OEArea, 22  
SetVerbose  
    OEPB::OEZap, 40  
SetZap  
    OEPB::OEBind, 25  
SimpleBind  
    OEPB::OEBind, 25

**T**

Tanimoto  
    OEPB::OEET, 31  
transfer.py  
    Example Code, 14

**Z**

zap\_atompot.py  
    Example Code, 10  
zap\_focussing.py  
    Example Code, 16  
zap\_forces.py  
    Example Code, 18  
zap\_grid1.py  
    Example Code, 5  
zap\_grid2.py  
    Example Code, 6  
zap\_grid3.py  
    Example Code, 8  
zap\_solv1.py  
    Example Code, 13