JOURNAL OF CHEMICAL INFORMATION AND MODELING

Article

pubs.acs.org/jcim

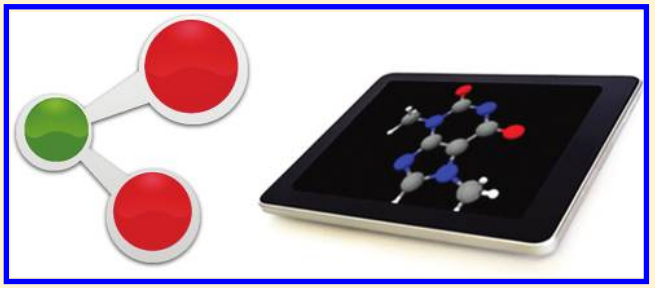
Atomdroid: A Computational Chemistry Tool for Mobile Platforms

Jonas Feldt,[†] Ricardo A. Mata,[†] and Johannes M. Dieterich*[,†]

[†]Institut für Physikalische Chemie, Universität Göttingen, Tammannstrasse 6, D-37077 Göttingen, Germany

Supporting Information

ABSTRACT: We present the implementation of a new molecular mechanics program designed for use in mobile platforms, the first specifically built for these devices. The software is designed to run on Android operating systems and is compatible with several modern tablet-PCs and smartphones available in the market. It includes molecular viewer/builder capabilities with integrated routines for geometry optimizations and Monte Carlo simulations. These functionalities allow it to work as a stand-alone tool. We discuss some particular development aspects, as well as the overall feasibility of using computational chemistry software packages in mobile platforms. Benchmark calculations show that through efficient implementation techniques even hand-held devices can be used to simulate midsized systems using force fields.



INTRODUCTION

The portability and intuitive haptic control functions of hand-held devices, such as smartphones and tablet-PCs, were until recently their only defining aspect and major advantage over other portable computing devices. However, we currently observe a rampant growth in their computing power, with dual-core systems being the current standard for tablet-PCs and first quad-core systems released by the end of 2011. It is expectable that such devices in the near future will be able to routinely handle intensive computing tasks and/or will be used as paradigms for the development of new platforms. Therefore, the availability of computational chemistry tools on these systems is of importance since any software developed for workstation or cluster usage is not directly transferable to the former.

In a first attempt to tap into the potential of these devices, we developed the Atomdroid application, a computational chemistry tool for mobile platforms. It contains both building and viewing functionalities for chemical systems and provides means of computational chemistry, namely Monte Carlo simulations and local optimizations. The program is currently limited to the use of molecular mechanics. Our discussion will focus on this type of implementation, although we also later discuss the perspective of electronic structure calculations.

Different platforms exist for portable computing devices, among them currently iOS, Android, Symbian, Windows Mobile, and WebOS. We decided to develop the application exclusively for the Android platform. Among the reasons are the freely available software development kit (SDK), the possibility to use a dialect of the Java language for development, and the high and increasing market share of Android devices.

Android devices are available from low-end to high-end segments of the market with even low-budget Android hand-helds nowadays possessing features so-far only known from expensive tablet computers and convertibles. This easy accessibility obviously broadens the target audience, allowing for the assumption that, e.g., a high and growing percentage of chemistry students has access to these devices. Therefore, these types of applications may also serve as virtual replacements for molecular model sets. We also still view the portability as a defining feature, giving researchers the possibility to develop their work on-the-go.

The number of chemistry-oriented applications has seen a steady-growth in the latest years. Most of these are very task-specific, usually geared toward student use (periodic tables, formula databases, ...). Only a selected few are geared toward visualization of 3-D molecular structures. From this very small group, only the Hyperchem app,[1] which is available for Windows Mobile and Apple devices, offers the possibility of manipulating molecular structures. All other programs work exclusively on the basis of external geometry files and/or structural libraries supplied with the application. With the Hyperchem app, it is also possible to submit computation jobs to a server, which must have a running Hyperchem server. The possibility of running local jobs on hand-held devices has remained, therefore, untapped.

CAPABILITIES

Since the Android platform is designed for mobile devices, any application developed for it should share the same characteristics. Namely, it should not rely upon any external hardware for executing any of its fundamental features. Therefore, we decided to implement besides the viewing and building functionality also computing capabilities. This includes an universal molecular mechanics engine which can be used to locally optimize given chemical systems and/or carry out Monte Carlo simulations for a glance into the conformational

Received: September 8, 2011
Published: March 11, 2012

dx.doi.org/10.1021/ci2004219 | J. Chem. Inf. Model. 2012, 52, 1072−1078

space of the system under study. All features of the Atomdroid application and their interactions are schematically summarized in Figure 1.

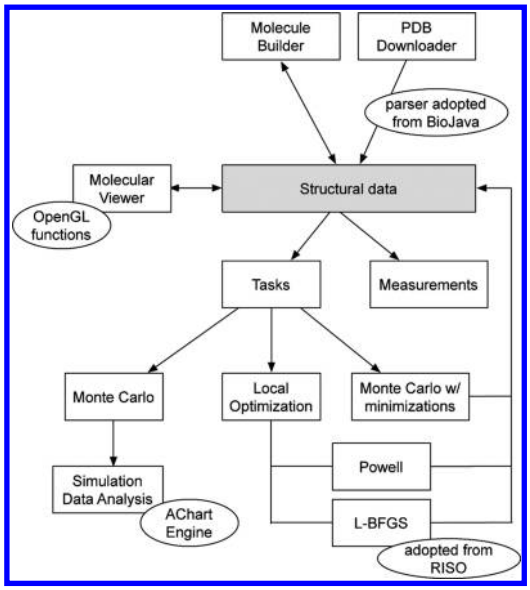


Figure 1. Schematic representation of Atomdroid capabilities and their interaction.

Our program is centered around the structural data which can be either read in from an xyz- or pdb-file or created by the user. Using OpenGL-functions, the structure is displayed and may be rotated, enlarged, or moved. Independent of these operations, in dedicated computing threads, Monte Carlo simulations, local optimizations, and Monte Carlo with Minimizations (MCM) can be carried out. This splitting of computational load into a thread dedicated for rendering and displaying and one for computations allows for a good interactivity on single-core devices and a good exploitation of the capabilities of dual-core devices. We now proceed to detail the main implemented functionalities.

Molecular Viewer/Builder. Atomdroid can be used as a standalone molecular viewer and builder. The user may either start from an external file (xyz- or pdb-format) or from scratch. The structure is displayed in 3D either in a ball-and-stick or line modus (see Figure 2). Independent of whether the structure has been loaded from a file or created by hand, the same manipulations are possible. These include atom and bond adding, removing, atom type changes and atom coordinates displacement. Structures may either be saved in the standard xyz-format or in a custom xyz-based ASCII-format that preserves connectivity information while maintaining readability. A description of this format is available in the Supporting Information. Also, the Atomdroid application makes full use of the Android system latest multitouch abilities and integrates into its haptic appearance. Zooming is performed through a pinch-and-zoom gesture, translations are realized through a two-finger move and rotations through a one-finger move.

The standard file format for biomolecular structures is the format developed by the Protein Data Bank (pdb). The data bank itself is a collection of published structures for a variety of systems, providing easy access by means of unique four-character IDs.[2] The Atomdroid program allows the user to download any of the published structures directly to the device

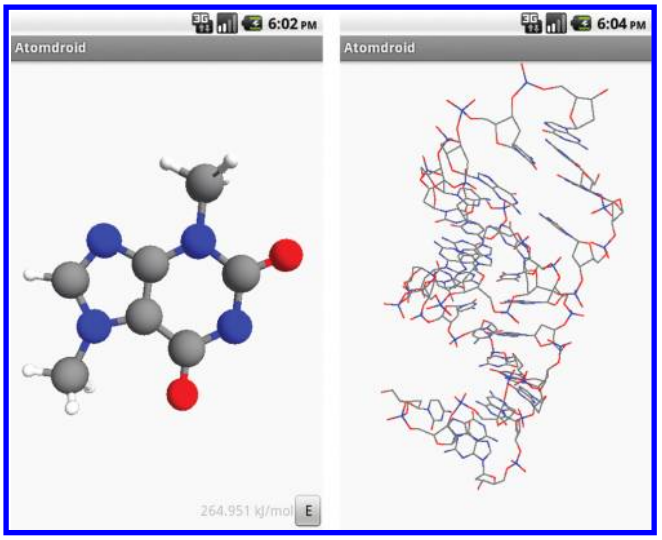


Figure 2. Left: Molecular structure read from a xyz-file displayed as balls-and-sticks. Right: Protein structure read from a pdb-file displayed in line modus.

using only the PDB-ID. The Atomdroid program builds a connection to the http server of the Protein Data Bank, downloads the compressed pdb structure, decompresses it on-the-fly, and displays it. A copy of the pdb is stored on the nonvolatile memory of the device and may be loaded again at a later point.

If a new structure is created from scratch, atoms are successively added by choosing an atom to connect to, selecting the atom type from a periodic table and specifying the bond order. Possible bond orders are no bond, single, double, triple, and resonant bond. A dragging motion allows for an approximate placement of the added atom. Hydrogen atoms can be automatically added to most organic compounds, requiring only the backbone of the molecule to be build explicitly. It is therefore possible to rapidly build rather large compounds and study relevant geometric properties directly.

Molecular Mechanics Engine. In order to minimize the amount of user input and keep the implementation as user-friendly as possible, the first implementation of Atomdroid only includes the universal forcefield UFF.[3] Two conventions are commonly used, one containing the Coulomb interaction of partial charges, the other one neglecting these contributions. Atomdroid currently ships with a version of UFF following the latter convention in agreement with e.g. the OpenBabel package[4,5] and MCCCS Towhee.[6] Therefore, the implemented energy expression simplifies to

$$E_{tot} = E_{bonds} + E_{angles} + E_{dihedrals} + E_{inversions} + E_{vdW} \quad (1)$$

For a detailed discussion on the functional form of the terms and the rationals for their design, we refer to the original paper (ref 2 and the references cited therein).

All parameters used in the Atomdroid application are identical to the original ones from Rappé et al.[3] The UFF energy and gradient implementation has been successfully checked for a number of systems against the OpenBabel implementation[4] for correct numerical results.

Local Optimizer. The current version of the Atomdroid application ships with two different local optimization algorithms. Due to the mobile character of the Android

dx.doi.org/10.1021/ci2004219 | J. Chem. Inf. Model. 2012, 52, 1072−1078

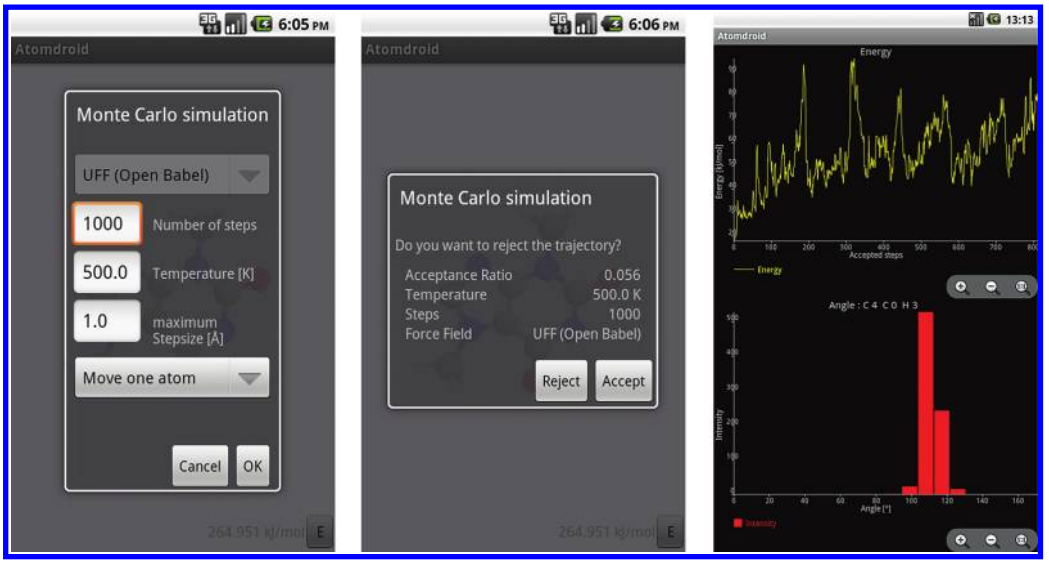


Figure 3. Left: Settings dialogue of the Metropolis Monte Carlo simulation. Middle: Statistics dialogue of the Metropolis Monte Carlo simulation. Right: Visual analysis of the Monte Carlo simulation using AChartEngine.

platform, available memory to-date is seriously restricted in most hand-helds. Therefore, the algorithms of choice need to have a low memory footprint and scaling with the system size. Since typical implementations first feature only numerical gradients, the decision was made to use two different local optimization algorithms, namely a limited-memory BFGS (L-BFGS)[7] and a gradient-free Powell optimization.[8] The L-BFGS was adapted from a GPL-licensed implementation in the RISO, a framework for distributed believe networks,[9] which is a semiautomatic translation of the original public domain version by Nocedal.[10] An analytical UFF gradient implementation is available and is used by the L-BFGS optimization routine.

For both local optimization engines, the user may control all parameters used or can (partly) select sensible default settings. This includes for the L-BFGS the number of updates stored, which, as a rule of thumb, is proportional to the efficiency of the optimization but also to the consumed memory. Therefore, it is possible to adapt to the hand-held in use and properly exploit high-end configurations.

Metropolis Monte Carlo Engine. Metropolis Monte Carlo[11] provides means to efficiently sample the phase space without requiring gradient information. This makes it a suitable computational tool for rapid investigations of relevant configurations of a system in phase-space. The implementation employs a straightforward Metropolis Monte Carlo with the user-tunable parameters temperature, maximum step length, and number of steps (Figure 3). Additionally, the user may choose to move not a single atom but all atoms in every step. While potentially causing a lower acceptance ratio, it may yield a faster sampling of the relevant minima of the conformational space. Also, the dialogue automatically makes a worst-case calculation of the space required for the trajectory file on the device, making it possible for the user to estimate the requirements for the simulation and adapt his settings accordingly.

All accepted steps are written to the nonvolatile memory of the device in the format of a Cartesian trajectory file. A dialogue is then displayed containing the most important statistics of the simulation (Figure 3). In the case of an unsuccessful simulation (e.g., acceptance ratio too low), the trajectory can be rejected, directly freeing space on the device. The user may then choose specific degrees of freedom (interatomic distances, angles, or dihedrals). Binned information will then be displayed as well as the energy progress using AChartEngine (Figure 3).[12]

Monte Carlo with Minimizations. As yet another possibility to explore the conformational space of the system in study, a Monte Carlo with minimizations implementation is available for global optimization.[13] Every Cartesian-based Monte Carlo move is followed by a local minimization before the Metropolis criterion is evaluated. This algorithm allows for a relatively rapid exploration of relevant minima for small and medium systems on the mobile device at the cost of additional local optimization steps requiring multiple energy and gradient evaluations. This also makes it potentially prohibitive for the study of bigger systems since the necessary runtime may become too high to keep the app in the foreground context. Therefore, we provide this engine as an option targeted at exploring small and medium sized systems.

IMPLEMENTATION DETAILS

General Considerations. The Android platform is a complete operating system targeted at mobile devices. It is based on a Linux kernel and userland extended with a virtual machine, the DalvikVM, and a set of libraries. Android applications are written in a Java-dialect based on Java 1.5 and are executed in the DalvikVM, using the Android libraries for e.g. graphical operations. Exceptions arise only for applications containing native code, which are typically written in C/C++. Therefore, different operating system versions typically contain also different versions of the Linux kernel and the DalvikVM and can be identified through their *Application Programming Interface* (API) level.

The Android system itself is rapidly progressing and being developed by Google Inc., with nine different API levels being currently available and in use.[14] At the same time, the liberal licensing terms cause a version fragmentation on different hand-helds from different manufacturers. Therefore, the most important decision to be made was the minimum API level required to run Atomdroid. In general, a lower API level allows for a broader availability of the application including low-budget devices. Higher API levels provide a higher performance

due to low-level optimizations in the Android system as well as more rapid development processes due to optimized APIs.

We decided to target level 8 (equals to Android 2.2) as the minimum requirement for Atomdroid. From this release on, the DalvikVM contains a *just-in-time* compiler (JIT), known from e.g. Java and Python. Especially for compute intensive tasks, JITs increase the performance (up to several orders of magnitude) by compiling aggressively optimized native code at runtime.

The decision for a minimum API level must be followed by an implementation allowing for a smooth user experience. Albeit, as pointed out above, the hand-held performance increasing and low-level optimizations in the Android system taking place, certain rules must be obeyed for a reasonable application performance. Memory is in comparison to classical computing systems very tight with the maximum heap size being in the order of 16 to 32 megabytes (device dependent) per application. Also, the memory management of the DalvikVM in version 2.2 cannot yet compete with garbage collectors of recent desktop or server JVMs. Therefore, it is of utmost importance to avoid unnecessary memory allocation/freeing and instead heavily rely upon reusing and caching memory (*allocate once, use often* paradigm) even for small arrays of primitives (e.g., doubles).

The Atomdroid application is only available on the Android platform. Therefore, only the standard graphical APIs of the Android system need to be used. While Android provides own APIs for dialogues and other 2D patterns, it relies on OpenGL ES[15] native operations for 3D purposes. As the OpenGL ES is the implementation for embedded devices, it contains significantly less primitives than the desktop version, therefore requiring own implementations of needed functionality like the drawing of 3D spheres and cylinders. The resulting code is therefore a mixture of OpenGL ES and Android calls. It has been abstracted as much as possible to allow for rapid replacements if e.g. the APIs of either Android or OpenGL ES should be extended or changed in the future.

The use of the UFF expression is conditioned by the input form used, as the topology needs to be provided by the user. The program is able to recognize, based on sensible defaults, whether two atoms are connected. However, the bond order, if other than single, needs to be inputted as an extra. Once the bonding matrix is defined, the UFF atom types are automatically assigned. In exotic cases, the user may change the UFF-type manually as needed. The goal is to provide a straightforward interface which can be used by a nonversed user, while at the same time offering the necessary flexibility to take full advantage of the implemented force field. Afterward it is possible to do one-click energy evaluations by simply pressing an on-screen button.

SPECIFIC PORTING ISSUES

When discussing specific porting issues associated to the development of the Atomdroid application, two independent efforts must be distinguished. The more basic question of the two is the object-oriented implementation of numerical methods. Although the development of scientific program codes to date is still dominated by legacy languages, insight can be gained from both dedicated in-depth analysis of this topic[16] as well as previous work from one of us.[17] Our conclusion is that the actual algorithmic implementation can be left relatively untouched when moving to object-oriented languages. It should be noted that this assertion does not hold true when moving to even more modern programming languages containing functional properties like, e.g., Scala. The interaction in between methods in terms of objects and design patterns[18] should be the greatest concern. The program architecture should reflect natural choices for objects (e.g., a geometry object) without introducing too much syntactic overhead (e.g., an object for a single coordinate) and, whenever possible, be agnostic toward the actual implementation to allow for a plug-and-play replacement and extension of, e.g., force field implementations or local optimization algorithms by means of design patterns to secure a good maintainability. The most important design patterns employed therefore include adaptors for interfacing with external libraries and decorators for the easy extension of force fields. These efforts should be the generic goal of any modern scientfic program code and are independent of the Android platform or the Atomdroid application.

The Android specific part of porting issues is associated with the embedded character of the platform and has been generically discussed in the previous section. Since actual code examples from every implemented routine would be rather tedious and repetitive, we limit the discussion to a couple of examples that we consider suitable for a general impression. The overhead and performance decrease associated with memory allocation and freeing in the DalvikVM has been discussed in detail in the previous section. This reflects in the actual implementation 2-fold. One can distinguish two general cases, small helper methods and larger algorithms.

A typical small helper method is, e.g., the computation of an angle in between three atoms as needed for the bond angle term in force field expressions. Such methods are likely inlined by the JIT as they feature only a few bytecode instructions and are marked static, which helps the JIT to decide for inlining. A sideffect of the static keyword is that any local field of the containing object used for temporary storage must also be static, giving rise to potential race conditions when methods get parallelized.

The code listing in Figure 4 shows our implementation of the angle in between three atoms. To avoid overhead, the two

```
static double angle(double[] p1, double[] p2, double[] p3, double[] t1,
   double[] t2) {
       for (int i = 0; i < 3; i++) {
           t1[i] = p1[i] - p2[i];
           t2[i] = p3[i] - p2[i];
       }
       return acos(scalarProduct(t1, t2)/(value(t1)*value(t2)));
}
```

Figure 4. Code listing for the angle implementation. p1, p2, and p3 are the three Cartesian coordinate sets, and t1 and t2 are temporary scratch arrays used for the directional vectors.

arrays necessary for the directional vectors (t1 and t2) are arguments of the method call. Using individual arrays for the Cartesian coordinates of the atoms (p1, p2, and p3) does not introduce an overhead since Dalvik stores array fields (as used for all Cartesian coordinates) as arrays of arrays. This optimization is also used for other similar small methods as for example the dihedral angle computations and would also be carried out in standard Java.

An example for a larger algorithm obviously is the UFF force field as such. Here the array fields storing information on the interatomic distances, bonds, natural bond lengths, and 1−3 connections are reused in between different calls for energy and gradient. Obviously, it must be ensured that no atoms are

added or removed in between these calls or the caches are poisoned. Additionally, the minimal number of small scratch arrays for the angle and dihedral computations are reused. Since storing them in the form of an array field would involve two checks for array bounds and the JIT might not be capable of optimizing them away under all conditions, said temporary arrays are stored individually. This optimization is Android specific, as the overhead of allocating and freeing such array fields is neglectable in comparison to the time spent in the actual computations.

One can conclude that the coding techniques required for an efficient implementation in the Android platform do not differ much from those employed in legacy codes (especially Fortran). Just as in their historic counterpart, the price to pay for efficiency is readability and maintainability.

Benchmark Calculations. Some scepticism about the possibility of performing calculations in hand-held devices is legitimate. Therefore, we see it as reasonable to actually discuss their performance. Contrary to most works discussing molecular mechanics, we will not be looking into large scale simulations and how they can be performed in a matter of days. These timeframes would also be unacceptable for any mobile application which is typically used only for minutes, not for hours. We mainly wish to show that one is able to carry out simple calculations without running out of battery. We demonstrate the possibilities of current mobile device hardware through a series of energy evaluations of systems of increasing size. The systems of choice span a region of 6 to 201 atoms and are depicted in Figure 5.

Due to inherent characteristics of languages running in virtual machines, short microbenchmarks are not an adequate tool for benchmarking. This is mainly due to the way a JIT compiler works. At runtime, it will detect compute intensive spots in the code and, once identified, compile aggressively optimized machine code. Detection works normally based upon the relative time spent in a certain function and the number of times the function is executed. Therefore, it is of utmost importance to allow the JIT to *warm up* when benchmarking to draw a realistic picture.

To ensure this, we present here averaged timings. These were obtained from four series of 2000 energy or combined energy and gradient evaluations per system, with the actual time obtained through the standard *System.nanoTime()* call. The influence of the JIT compiler should therefore be correctly contained in the data. Standard deviations of sets are calculated based on the averaged results.

All Dalvik results depicted in Table 1 and Table 2 were obtained on a Point of View Mobii tablet running Android 2.2 on an NVIDIA Tegra2 250 dual-core with Atomdroid version 1.2.0. The difference between the timings denoted as GC optimized and the standard ones lies in the memory allocation. While the standard timings were obtained with local memory allocation typical for desktop or server applications, the GC optimized ones allocated scratch arrays only once as part of the earlier mentioned *allocate once, reuse often* paradigm to reduce the stress on the garbage collector. The difference between the two schemes becomes obvious when considering the calculation of the distance matrix of all atoms in the system. This matrix needs to be recalculated for every step in a local optimization or Monte Carlo simulation. In a desktop or server application, one would typically allocate a new matrix in memory in every step and rely on the GC to either reuse the previous memory space or find a better aligned one somewhere else. The performance of the GC in the Android system makes it necessary to reuse the matrix explicitly. The same paradigm holds true for even smaller tasks, like the mentioned calculation of an angle which requires, despite the three sets of Cartesian coordinates, two temporary arrays with three elements each. Even for these small amounts of memory, explicit reuse needs to take place to ensure a reasonable performance.

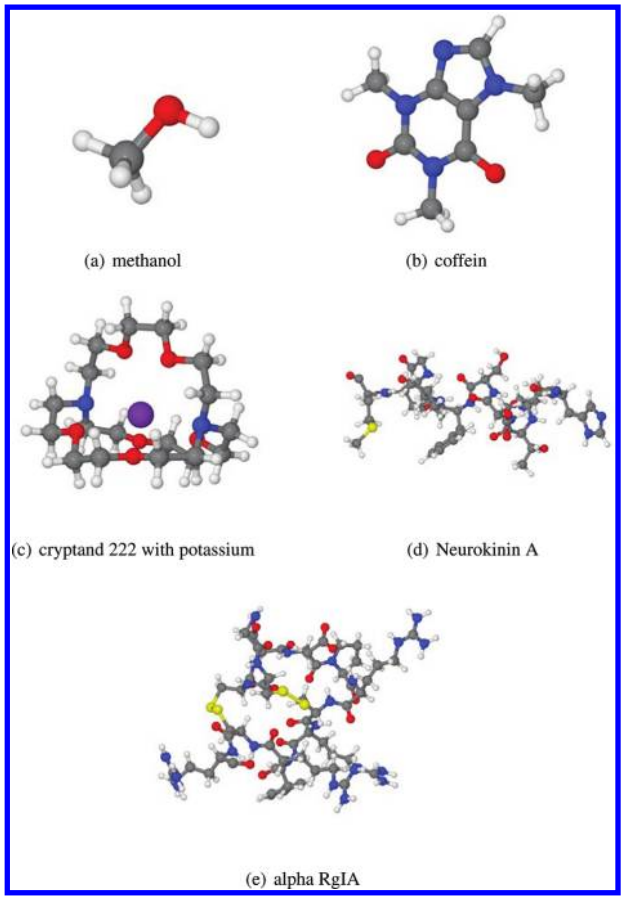


Figure 5. Benchmark systems used. All structural representation have been obtained using Jmol[19] and PovRay.[20]

The higher timing introduced by more GC activity are obvious and especially drastic in the larger benchmark systems D and E. As pointed out before, the GC shipped with Android 2.2 is suboptimal and has seen a significant upgrade from the already available Android 3.0 on. When doing local optimizations or Monte Carlo simulations, the Atomdroid program employs the later memory allocation scheme. The scaling is subquadratic in the tested regime. Since no cutoffs are employed, this reduction is due to some constant time overhead like memory allocation being present independent of the system size.

For comparison, Table 1 contains timings of a ported version of our GC-optimized UFF implementation on the standard JVM, namely the current openJDK7. The hardware chosen for this benchmark was a contemporary Samsung NC10 netbook (Intel Atom N455 1.6 GHz) running openSUSE 11.4. This system comes closest to a tablet in prize point, form factor and battery life. It should be noted though that these results can only serve as mere hints since they contain multiple differences. Despite the different CPU architectures (ARM 32 bit vs x86 64 bit), different virtual machines (Dalvik vs JVM) are used. Only the latter makes a huge difference as Dalvik is stack-based while

dx.doi.org/10.1021/ci2004219 | J. Chem. Inf. Model. 2012, 52, 1072−1078

Table 1. Benchmark Timings for the Systems Depicted in Figure 5 Comparing a Standard and a *Garbage Collector* (GC)-Optimized Implementation of the Force Field on Dalvik against a Standard JVM for Energy Evaluations

system	no. of atoms	Dalvik [ms]	Dalvik, GC opt. [ms]	JVM, GC opt. [ms]
A	6	0.264 ± 0.006	0.172 ± 0.022	0.151 ± 0.200
B	24	2.484 ± 0.016	1.578 ± 0.007	0.608 ± 0.274
C	63	16.172 ± 0.013	12.297 ± 0.003	3.228 ± 0.771
D	155	132.231 ± 0.155	106.982 ± 0.057	26.478 ± 0.114
E	201	263.325 ± 0.539	227.872 ± 0.307	54.368 ± 0.834

Table 2. Benchmark Timings for the Systems Depicted in Figure 5 Comparing a Standard and a GC-Optimized Implementation of the Force Field on Dalvik against a Standard JVM for Combined Energy and Gradient Evaluations

system	no. of atoms	Dalvik [ms]	Dalvik, GC opt. [ms]	JVM, GC opt. [ms]
A	6	0.368 ± 0.013	0.240 ± 0.004	0.197 ± 0.231
B	24	3.416 ± 0.021	2.739 ± 0.013	0.870 ± 0.394
C	63	21.573 ± 0.032	16.470 ± 0.029	3.636 ± 0.494
D	155	150.661 ± 0.051	124.552 ± 0.080	29.395 ± 0.486
E	201	286.656 ± 0.266	243.517 ± 0.157	58.398 ± 0.429

the standard JVMs are register-based[21] and may easily cause speedups of up to a factor 3 with the JVM, as seen in a recent benchmark study by Oracle[22] comparing Java SE embedded to Android 2.2 on a similar Tegra-based device. On the other hand it should be noted that the implementation was optimized for Android devices with a focus on limited memory and interactive usage, causing certain design decisions. For example to reduce memory consumption of intermediate results, simple floating point operations may be carried out redundantly. Therefore, this implementation is currently unsuitable to be benchmarked against normal implementations. As memory increases in embedded devices, implementations will eventually converge though. In general, the netbook executes the UFF energy and combined energy/gradient calculations four times faster than the tablet, neglecting artificats occurring in the small regime. One can conclude that the raw performance of current Android hardware is in the same ballpark as netbooks and that possible enhancements in the virtual machine have the potential to close this gap even further.

In general, the results allow for the conclusion that systems up to around 200 atoms can be studied on present-day hardware without serious negative impacts to the interactivity. We expect that further advancements in the hardware and even more in the design of the DalvikVM will allow for even larger systems to be computed on mobile devices. This may well include semiempirical methods requiring a linear algebra library for, e.g., matrix multiplications and diagonalizations. Existing libraries are focused on desktop applications (e.g., Jama,[23] netlib,[24] and UJMP[25]). Although they may be easily ported to the Android platform, the current GC of the Android platform is, as discussed above, prohibitive since it requires another memory allocation model for reasonable performance. If the GC in newer versions proves to work similar to a desktop Java, it will be trivial to make use of any of the before mentioned linear algebra libraries. The processor speed is already satisfactory for these tasks, although one should consider that typical numerical operations are carried out in 64bit accuracy and currently the ARM-based CPUs found in most hand-held devices are using 32bit word sizes, slowing these operations significantly down.

CONCLUSION

We have presented a stand-alone program for use in molecular mechanics calculations, completed with molecule viewer/builder functionality freely available for Android platforms. This is the first molecular mechanics software specifically designed for hand-held devices. The current implementation makes use of an universal forcefield, delivering a straightforward easy-to-use software package. Benchmark timings are given for a small set of test systems. Results show that basic modeling tasks can be performed with reasonable timings for systems under 200 atoms.

The Android market automatically creates anonymous statistics of the program usage for the developer when submitting an application. This provides, among other things, a fairly fine-grained view on which devices run the application. It distinguishes between downloads and active users, where active users are the users that keep the application installed on their device. As of December 2011 (within six months), the application has about 2800 active users. 28.8% of the devices run Android 2.2, 51.3% run 2.3.3 or 2.3. The choice of using 2.2 as the minimum standard proves to be reasonable and the focus on optimizing the implementation for the DalvikVM of this particular release is worthwhile. Also, we can see from the statistics that although most users run the Atomdroid on smartphones, tablets take a significant and increasing share (approximately 20%). The application proves useful in very different environments (screen sizes and resolutions, hardware resources).

As pointed out previously, the design of this application followed the rational that the broad availability of more powerful mobile devices changes the way people use computers and programs. This in turn will also ultimately require that tools, among them scientific ones which are currently used on workstations, will be made available to the mobile world. The exact integration and adoption is obviously user dependent and one can only imagine how these will develop. Nevertheless, some uses can be readily identified and were the reason behind this work.

Students and lecturers already use Atomdroid to have three-dimensional representations of relevant compounds and apply this in the context of chemical education. One is not restricted to simply viewing the structure. The molecular mechanics engine allows a rough estimation of properties like bond

lengths, relative conformational energies and nonbonded interactions. Unfavorable conformations can be readily corrected with a local optimization on the device.

Researchers in the field should also significantly profit from the application. Tablets and smartphones are quickly replacing laptop PCs as the devices of choice when traveling. With Atomdroid, a pdb or xyz attachment to a mail can be readily opened, the structure visualized or even changed. One can have direct access to the Protein Data Bank or quickly assemble a molecule, explore different conformations, and perhaps even later use the structure obtained for larger simulations. The haptic functionalities are also a very desireable feature. Visualization of 3D molecular structures using multitouch commands is much more intuitive than with a mouse and/or keyboard. This also applies in the workplace. Additional features targeted especially at this audience are planned in development, including support for more file formats and structural databases.

■ ASSOCIATED CONTENT

Supporting Information

Description of the Atomdroid file format. This material is available free of charge via the Internet at http://pubs.acs.org.

■ AUTHOR INFORMATION

Corresponding Author

*E-mail: jdieter@gwdg.de.

Notes

The authors declare no competing financial interest.

■ ACKNOWLEDGMENTS

Financial support from the German Excellence Initiative, through the Free Floater Research Group program of the University of Göttingen, is gratefully acknowledged.

■ REFERENCES

(1) *iHyperchem Release 1.0*; Hypercube Inc.: Gainesville, Florida, USA, 2011.

(2) Berman, H.; Westbrook, J.; Feng, Z.; Gilliland, G.; Bhat, T.; Weissig, H.; Shindyalov, I.; Bourne, P. The Protein Data Bank. *Nucleic Acids Res.* **2000**, *28*, 235−242.

(3) Casewit, A. K. R. C. J.; Colwell, K. S.; Goddard, W. A.; Skiff, W. M. UFF, a full periodic table force field for molecular mechanics and molecular dynamics simulations. *J. Am. Chem. Soc.* **1992**, *114*, 10024−10035.

(4) Guha, R.; Howard, M. T.; Hutchison, G. R.; Murray-Rust, P.; Rzepa, H.; Steinbeck, C.; Wegner, J. K.; Willighagen, E. The Blue Obelisk: Interoperability in Chemical Informatics. *J. Chem. Inf. Model.* **2006**, *46*, 991−998.

(5) *The Open Babel Package*, version 2.0.1; http://openbabel.sourceforge.net (accessed December 07, 2011).

(6) MCCCS Towhee. towhee.sourceforge.net (accessed December 07, 2011).

(7) Matthies, H.; Strang, G. The solution of nonlinear finite element equations. *Int. J. Num. Meth. Eng.* **1979**, *14*, 1613−1626.

(8) Powell, M. J. D. An Efficient Method for Finding the Minimum of a Function of Several Variables Without Calculating Derivatives. *Comput. J.* **1964**, *7*, 155−162.

(9) Dodier, R. RISO: an implementation of distributed belief networks in Java. http://riso.sourceforge.net (accessed December 07, 2011).

(10) Nocedal, J. Updating quasi-Newton matrices with limited storage. *Math. Comput.* **1980**, *35*, 773−782.

(11) Metropolis, N.; Ulam, S. The Monte Carlo Method. *J. Am. Stat. Assoc.* **1949**, *44*, 335−341.

(12) AChartEngine: a charting software library for Android applications. www.achartengine.org (accessed December 07, 2011).

(13) Li, Z.; Scheraga, H. A. Monte Carlo-minimization approach to the multiple-minima problem in protein folding. *Proc. Natl. Acad. Sci. U.S.A.* **1987**, *84*, 6611−6615.

(14) Android Platform Versions. 2011. http://developer.android.com/resources/dashboard/platform-versions.html (accessed December 07, 2011).

(15) Munshi, A.; Leech, J. OpenGL ES Common Profile Specification Version 2.0.25. 2010. http://www.khronos.org/opengles (accessed December 07, 2011).

(16) Besset, D. H. *Object-Oriented Implementation of Numerical Methods. An Introduction with Java and Smalltalk*; Morgan Kaufmann: San Francisco, CA, 2000.

(17) Dieterich, J. M.; Hartke, B. OGOLEM: Global cluster structure optimisation for arbitrary mixtures of flexible molecules. A multi-scaling, object-oriented approach. *Mol. Phys.* **2010**, *108*, 279−291.

(18) Gamma, E.; Helm, R.; Johnson, R. E. *Design Patterns. Elements of Reusable Object-Oriented Software.*; Addison-Wesley Longman: New York, 1994.

(19) Jmol: an open-source Java viewer for chemical structures in 3D. http://www.jmol.org/ (accessed December 07, 2011).

(20) POV-Ray - The Persistence of Vision Raytracer. www.povray.org (accessed December 07, 2011).

(21) Shi, Y.; Gregg, D.; Beatty, A.; Ertl, M. A. Virtual Machine Showdown: Stack Versus Registers in Proceedings of the VEE'05. ACM Press: New York, 2005. VEE'05 is the Virtual Execution Environments (VEE'05) conference. See http://research.ihost.com/vee/vee05/cfp.html (accessed December 07, 2011).

(22) Oracle, Java SE Embedded Performance Versus Android 2.2. 2010. http://blogs.oracle.com/javaseembedded/entry/how_does_android_22s_performance_stack_up_against_java_se_embedded (accessed December 07, 2011).

(23) Hicklin, J.; Moler, C.; Webb, P.; Boisvert, R. F.; Miller, B.; Pozo, R.; Remington, K. JAMA: A Java Matrix Package. http://math.nist.gov/javanumerics/jama/ (accessed December 07, 2011).

(24) Doolin, D.; Dongarra, J.; Seymour, K. JLAPACK - Compiling LAPACK Fortran to Java. *Scientific Programming* **1999**, *7*, 111−138.

(25) Arndt, H.; Bundschus, M.; Nägele, A. Towards a Next-Generation Matrix Library for Java. 2009. http://www.ujmp.org (accessed December 07, 2011).