Status of Grid Python Toolkit (GPT)

Christoph Lehner
(Uni Regensburg)

https://github.com/lehner/gpt

July 31st, 2023 - Lattice 2023
Grid Python Toolkit (GPT)

- A toolkit for lattice QCD and related theories as well as QIS (a parallel digital quantum computing simulator) and Machine Learning
- Python frontend, C++ backend
- Built on Grid data parallelism (MPI, OpenMP, SIMD, and SIMT)

https://github.com/lehner/gpt

Initial commit Feb. 2020, 65k lines of C++/Python, >1700 commits so far, 13 contributors
Code co-authors:

- M. Bruno
- D. Richtmann
- T. Blum
- S. Bürger
- P. Georg
- L. Jin
- D. Knüttel
- S. Meinel
- M. Schlemmer
- S. Solbrig
- T. Wettig
- T. Wurm
What’s new since lattice 2022?
What is new since 2022 (1/3):

- New Machine Learning features such as
  - Adam optimizer
  - gauge-equivariant (local) parallel-transport layers
  - gauge-equivariant (un)pooling layers
  - classical multi-grid blocking layers


see also Tilo’s talk today at 5pm in Algorithms session.
What is new since 2022 (2/3):

▶ Efficient general stencil and parallel transport:

```python
In [4]: U = g.qcd.gauge.random(grid, rng)

In [5]: code = []
   for mu in range(4):
       for nu in range(mu):
           code.append((0, -1 if code == [] else 0, 1.0, g.path().f(mu).f(nu).b(mu).b(nu)))
   plaq = g.parallel_transport_matrix(U, code, 1)

In [6]: print(g.sum(g.trace(plaq(U))).real * 2 / grid.gsites / (3 * 3 * 4))
   print(g.qcd.gauge.plaquette(U))

0.7961959248563635
0.7961959248563635
```

▶ Twisted Mass Fermions + DSDR term
▶ Quadruple precision global reduction support via Dekker tuples:

```python
grid = g.grid([4,4,4,4], g.double)

⇒

grid = g.grid([4,4,4,4], g.double_quadruple)
```
What is new since 2022 (3/3):

Performance on LUMI-G and Frontier (1 node):

15 TF/s/node possible without inter-GCD communication.
5 TF/s/node in strong scaling up to 64 nodes for $64^3 \times 256$ problem size!
Guiding principles:

- **Performance Portability**
  common Grid-based framework for current and future (exascale) architectures

- **Modularity / Composability**
  build up from modular high-performance components, several layers of composability, “composition over parametrization”
Layout and dependencies
Python script / Jupyter notebook

**gpt (Python)**
- Defines data types and objects (group structures etc.)
- Expression engine (linear algebra)
- Algorithms (Solver, Eigensystem, ...)
- File formats
- Stencils / global data transfers
- QCD, QIS, ML subsystems

**cgpt (Python library written in C++)**
- Global data transfer system (gpt creates pattern, cgpt optimizes data movement plan)
- Virtual lattices (tensors built from multiple Grid tensors)
- Optimized blocking, linear algebra, and Dirac operators
- Vectorized ranlux-like pRNG (parallel seed through 3xSHA256)
The QCD module
Example: Load QCD gauge configuration and test unitarity

```python
In [1]: import gpt as g

U = g.load("ckpoint_lat.IEEE64BIG.1100")

for mu in range(4):
    g.message("SU3 - Defect: ", g.norm2(U[mu] * g.adj(U[mu])) - g.identity(U[mu])))

GPT :  1.039211 s : SU3 - Defect:  3.345168726568745e-26
GPT :  1.094156 s : SU3 - Defect:  3.3476154954606903e-26
GPT :  1.146703 s : SU3 - Defect:  3.342180010368529e-26
GPT :  1.199097 s : SU3 - Defect:  3.3423193715873574e-26
```

Here: expression first parsed to a tree in Python (gpt), forwarded as abstract expression to C++ library (cgpt) for evaluation
Example: create a pion propagator on a random gauge field

```plaintext
# double-precision 8^4 grid
grid = g.grid([8,8,8,8], g.double)

# pRNG
rng = g.random("seed text")

# random gauge field
U = g.qcd.gauge.random(grid, rng)

# Mobius domain-wall fermion
fermion = g.qcd.fermion.mobius(U, mass=0.1, M5=1.8, b=1.0, c=0.0, Ls=24,
                               boundary_phases=[1,1,1,-1])

# Short-cuts
inv = g.algorithms.inverter
pc = g.qcd.fermion.preconditioner

# even-odd-preconditioned CG solver
slv_5d = inv.preconditioned(pc.eo2_ne(), inv.cg(eps = 1e-4, maxiter = 1000))

# Abstract fermion propagator using this solver
fermion_propagator = fermion.propagator(slv_5d)

# Create point source
src = g.mspincolor(U[0].grid)
g.create.point(src, [0, 0, 0, 0])

# Solve propagator on 12 spin-color components
prop = g( fermion_propagator * src )

# Pion correlator
g.message(g.slice(g.trace(prop * g.adj(prop)), 3))
```
Example: solvers are modular and can be mixed

General design principle: use modularity of python code instead of large number of parameters to configure solvers/algorithms; Python can also be used in configuration files

```python
# Create an coarse-grid deflated, even-odd preconditioned CG inverter
# (eig is a previously loaded multi-grid eigensystem)
sloppy_light_inverter = g.algorithms.inverter.preconditioned(
    g.qcd.fermion.preconditioner.eo1_ne(parity=g.odd),
    g.algorithms.inverter.sequence(
        g.algorithms.inverter.coarse_deflate(
            eig[1],
            eig[0],
            eig[2],
            block=200,
        ),
        g.algorithms.inverter.split(
            g.algorithms.inverter.cg({"eps": 1e-8, "maxiter": 200}),
            mpi_split=[1,1,1,1],
        ),
    ),
)
```
Further example: Multi-Grid solver

```python
def find_near_null_vectors(w, cgrid):
    slv = i.fgmr(eq=1e-3, maxiter=50, restartlen=25, checkres=False)(w)
    basis = g.orthonormalize(
        rng.cnorm([g.lattice(w.grid[0], w.otype[0]) for i in range(30)]))
    null = g.lattice(basis[0])
    null[:1] = 0
    for b in basis:
        slv(b, null)
        g.qcd.fermion.coarse.split_chiral(basis)
        bm = g.block.map(cgrid, basis)
        bm.orthonormalize()
        bm.check_orthogonality()
    return basis

mg_setup_3lvl = i.multi_grid_setup(
    block_size=[[2, 2, 2, 2], [2, 1, 1, 1]], projector=find_near_null_vectors
)

wrapper_solver = i.fgmr(eq=1e-1, "maxiter": 10, "restartlen": 5, "checkres": False)

smooth_solver = i.fgmr(eq=1e-14, "maxiter": 8, "restartlen": 4, "checkres": False)

coarsest_solver = i.fgmr(eq=5e-2, "maxiter": 50, "restartlen": 25, "checkres": False)

mg_3lvl_kcycle = i.sequence(
    i.coarse_grid(
        wrapper_solver.modified(
            prec=i.sequence(
                i.coarse_grid(coarsest_solver, *mg_setup_3lvl[1]), smooth_solver
            )
        ),
        *mg_setup_3lvl[0],
    ),
    smooth_solver,
)```
All algorithms implemented in Python – Example: Euler-Langevin stochastig DGL integrator

class langevin_euler:
    @g.params_convention(epsilon=0.01)
    def __init__(self, rng, params):
        self.rng = rng
        self.eps = params["epsilon"]

    def __call__(self, fields, action):
        gr = action.gradient(fields, fields)
        for d, f in zip(gr, fields):
            f @= g.group.compose(
                -d * self.eps
                + self.rng.normal_element(g.lattice(d)) * (self.eps * 2.0) ** 0.5,
                f,
            )
Implemented algorithms:

- BiCGSTAB, CG, CAGCR, FGCR, FGMRES, MR solvers
- Multi-grid, split-grid, mixed-precision, and defect-correcting solver combinations
- Coarse and fine-grid deflation
- Implicitly restarted Arnoldi and Lanczos, power iteration
- Chebyshev polynomials
- All-to-all vector generation
- SAP and even-odd preconditioners
- MSPCG (additive Schwarz)
- Gradient descent, non-linear CG, Adam optimizers
- Runge-Kutta integrators, Wilson flow
- Fourier acceleration
- Coulomb and Landau gauge fixing
- Domain-wall–overlap transformation and MADWF
- Symplectic integrators (leapfrog, OMF2, and OMF4)
- Markov: Metropolis, heatbath, Langevin, (DD-)HMC
Implemented fermion actions:

- Domain-wall fermions: Mobius and zMobius
- Twisted-mass fermions
- Wilson-clover fermions both isotropic and anisotropic (RHQ/Fermilab actions); Open boundary conditions available

Example: stout-smeared heavy-quark Mobius DWF

```python
# load configuration
U = g.load(config)
grid = U[0].grid

# smeared gauge link
U_stout = U
for n in range(3):
    U_stout = g.qcd.gauge.smear.stout(U_stout, rho=0.1)

fermion_exact = g.qcd.fermion.mobius(U_stout,
    "mass": 0.6,
    "M5": 1.0,
    "b": 1.5,
    "c": 0.5,
    "Ls": 12,
    "boundary_phases": [1.0, 1.0, 1.0, -1.0],
)}
```
Performance
Results available for GPU and CPU architectures. In the following, focus on Frontier/LUMI-G (AMD MI250X), Juwels/Leonardo booster (NVIDIA A100) and Fugaku/QPace4 (A64FX).

<table>
<thead>
<tr>
<th>Rank</th>
<th>System</th>
<th>Cores</th>
<th>Rmax (PFlop/s)</th>
<th>Rpeak (PFlop/s)</th>
<th>Power (kW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Frontier - HPE Cray EX35a, AMD Optimized 3rd Generation EPYC 64C 20Hz, AMD Instinct MI250X, Slingshot-11, HPE</td>
<td>8,699,904</td>
<td>1,194.00</td>
<td>1,679.82</td>
<td>22,703</td>
</tr>
<tr>
<td>2</td>
<td>Supercalculator Fugaku - Supercalculator Fugaku, A64FX 4C 2.40Hz, Tofu interconnect 0, Fujitsu</td>
<td>7,630,848</td>
<td>442.01</td>
<td>537.21</td>
<td>29,099</td>
</tr>
<tr>
<td>3</td>
<td>LUMI - HPE Cray EX35a, AMD Optimized 3rd Generation EPYC 64C 20Hz, AMD Instinct MI250X, Slingshot-11, HPE</td>
<td>2,220,288</td>
<td>309.10</td>
<td>428.70</td>
<td>6,016</td>
</tr>
<tr>
<td>4</td>
<td>Leonardo - BuiSequana XH2000, Xeon Platinum 8358 32C 2.60Hz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR10 Infiniband, Atos</td>
<td>1,824,768</td>
<td>238.70</td>
<td>304.47</td>
<td>7,404</td>
</tr>
<tr>
<td>5</td>
<td>Summit - IBM Power System AC32, IBM POWER9 22C 3.007GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM</td>
<td>2,414,592</td>
<td>148.60</td>
<td>200.79</td>
<td>10,096</td>
</tr>
</tbody>
</table>

Juwels Booster (node has $4 \times$ A100-40GB): Single-node domain-wall fermion $\not\psi$ operator

Compare to HBM bandwidth of 1,555 GB/s per GPU
QPace4 (node has one A64FX): Single-node domain-wall fermion \( \mathcal{D} \) operator

---

**Initialized GPT**

Copyright (C) 2020 Christoph Lehner

---

GPT : 0.265714 s :

- DWF Dslash Benchmark with
  - fdimensions : [24, 24, 24, 24]
  - precision : single
  - Ls : 8

GPT : 20.218240 s : 1000 applications of Dhop

- Time to complete : 3.67 s
- Total performance : 954.90 GFlops/s
- Effective memory bandwidth : 677.11 GB/s

GPT : 20.218842 s :

- DWF Dslash Benchmark with
  - fdimensions : [24, 24, 24, 24]
  - precision : double
  - Ls : 8

GPT : 45.245379 s : 1000 applications of Dhop

- Time to complete : 7.36 s
- Total performance : 475.80 GFlops/s
- Effective memory bandwidth : 674.77 GB/s

---

**Finalized GPT**

---

Compare to HBM bandwidth of 1,000 GB/s per A64FX
Juwels Booster (node has $4 \times$ A100-40GB): Single-node site-local matrix products

Compare to HBM bandwidth of 1,555 GB/s per GPU
Juwels Booster (node has $4 \times A100-40GB$): Inner product (reduction)

GPT : 28.406798 s : 100 rank_inner_product
: Object type : ot_vector_singlet(12)
: Block : 4 x 4
: Data resides in : accelerator
: Performed on : accelerator
: Time to complete : 0.13 s
: Effective memory bandwidth : 4827.16 GB/s
:
: rip: timing: unprofiled = 0.000000e+00 s (= 0.00 %)
: rip: timing: rip: view = 9.706920e-04 s (= 0.70 %)
: rip: timing: rip: loop = 1.369079e-01 s (= 99.30 %)
: rip: timing: total = 1.379585e-01 s (= 100.00 %)
:

Compare to HBM bandwidth of 1,555 GB/s per GPU
## Performance summary

<table>
<thead>
<tr>
<th>Machine</th>
<th>Operation</th>
<th>Performance</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frontier</td>
<td>$\emptyset$</td>
<td>9 TF/s</td>
<td>6.2 TB/s</td>
</tr>
<tr>
<td>Booster</td>
<td>$\emptyset$</td>
<td>12 TF/s</td>
<td>7.8 TB/s</td>
</tr>
<tr>
<td>Booster</td>
<td>ColorMatrix $\times$</td>
<td></td>
<td>5.2 TB/s</td>
</tr>
<tr>
<td>Booster</td>
<td>SpinColorMatrix $\times$</td>
<td></td>
<td>5.1 TB/s</td>
</tr>
<tr>
<td>Booster</td>
<td>SpinColorVector $\langle \cdot , \cdot \rangle$</td>
<td></td>
<td>4.8 TB/s</td>
</tr>
<tr>
<td>QPace4</td>
<td>$\emptyset$</td>
<td>0.95 TF/s</td>
<td>0.68 TB/s</td>
</tr>
<tr>
<td>SuperMUC-NG</td>
<td>$\emptyset$</td>
<td>0.72 TF/s</td>
<td>0.51 TB/s</td>
</tr>
</tbody>
</table>

Single-node SP performance of Wilson $\emptyset$ and linear algebra on Juwels Booster (4xA100, HBM BW 1.6 TB/s per A100), Qpace4 (A64FX, HBM BW of 1 TB/s per node), and the SuperMUC-NG (Skylake 8174). The $\emptyset$ performance is inherited from Grid, the linear algebra performance is based on cgpt.
Example applications
RBC ensemble generation

(the generating GPT scripts are linked below; around 200 lines of Python script each)

<table>
<thead>
<tr>
<th>ID</th>
<th>$a^{-1}$/GeV</th>
<th>$N_f$</th>
<th>$L^3 \times T \times L_s$</th>
<th>$b + c$</th>
<th>$m_{res} \times 10^4$</th>
<th>$m_\pi$/MeV</th>
<th>$m_K$/MeV</th>
<th>$m_{D_s}$/GeV</th>
<th>$m_\pi L$</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>48l</td>
<td>1.73</td>
<td>2+1</td>
<td>$48^3 \times 96 \times 24$</td>
<td>2</td>
<td>6.1</td>
<td>139</td>
<td>499</td>
<td>–</td>
<td>3.87</td>
<td>CPS</td>
</tr>
<tr>
<td>64l</td>
<td>2.35</td>
<td>2+1</td>
<td>$64^3 \times 128 \times 12$</td>
<td>2</td>
<td>3.1</td>
<td>139</td>
<td>507</td>
<td>–</td>
<td>3.77</td>
<td>CPS</td>
</tr>
<tr>
<td>96l</td>
<td>2.69</td>
<td>2+1</td>
<td>$96^3 \times 192 \times 12$</td>
<td>2</td>
<td>2.3</td>
<td>132</td>
<td>486</td>
<td>–</td>
<td>4.70</td>
<td>CPS</td>
</tr>
<tr>
<td>1</td>
<td>1.73</td>
<td>2+1</td>
<td>$32^3 \times 64 \times 24$</td>
<td>2</td>
<td>6.3</td>
<td>208</td>
<td>513</td>
<td>–</td>
<td>3.85</td>
<td>GPT script</td>
</tr>
<tr>
<td>2</td>
<td>1.73</td>
<td>2+1</td>
<td>$24^3 \times 48 \times 32$</td>
<td>2</td>
<td>4.6</td>
<td>284</td>
<td>534</td>
<td>–</td>
<td>3.96</td>
<td>GPT script</td>
</tr>
<tr>
<td>3</td>
<td>1.73</td>
<td>2+1</td>
<td>$32^3 \times 64 \times 24$</td>
<td>2</td>
<td>6.5</td>
<td>210</td>
<td>597</td>
<td>–</td>
<td>3.88</td>
<td>GPT script</td>
</tr>
<tr>
<td>4</td>
<td>1.74</td>
<td>2+1</td>
<td>$24^3 \times 48 \times 24$</td>
<td>2</td>
<td>6.2</td>
<td>279</td>
<td>534</td>
<td>–</td>
<td>3.84</td>
<td>GPT script</td>
</tr>
<tr>
<td>5</td>
<td>1.75</td>
<td>2+1+1</td>
<td>$24^3 \times 48 \times 24$</td>
<td>2</td>
<td>6.7</td>
<td>280</td>
<td>536</td>
<td>1.99</td>
<td>3.84</td>
<td>GPT script</td>
</tr>
<tr>
<td>6</td>
<td>1.75</td>
<td>2+1+1</td>
<td>$24^3 \times 48 \times 24$</td>
<td>2</td>
<td>6.7</td>
<td>280</td>
<td>536</td>
<td>1.5</td>
<td>3.84</td>
<td>GPT script</td>
</tr>
<tr>
<td>7</td>
<td>1.76</td>
<td>2+1+1</td>
<td>$24^3 \times 48 \times 24$</td>
<td>2</td>
<td>7.9</td>
<td>284</td>
<td>540</td>
<td>1.39</td>
<td>3.88</td>
<td>GPT script</td>
</tr>
<tr>
<td>8</td>
<td>2.37</td>
<td>2+1+1</td>
<td>$32^3 \times 64 \times 12$</td>
<td>2</td>
<td>3.0</td>
<td>280</td>
<td>536</td>
<td>1.99</td>
<td>3.88</td>
<td>GPT script</td>
</tr>
<tr>
<td>9</td>
<td>2.37</td>
<td>2+1</td>
<td>$32^3 \times 64 \times 12$</td>
<td>2</td>
<td>3.0</td>
<td>281</td>
<td>535</td>
<td>–</td>
<td>3.80</td>
<td>GPT script</td>
</tr>
<tr>
<td>A</td>
<td>1.76</td>
<td>2+1</td>
<td>$24^3 \times 48 \times 8$</td>
<td>2</td>
<td>41.5</td>
<td>303</td>
<td>548</td>
<td>–</td>
<td>4.15</td>
<td>GPT script</td>
</tr>
<tr>
<td>B</td>
<td>1.73</td>
<td>2+1</td>
<td>$32^3 \times 64 \times 24$</td>
<td>2</td>
<td>6.1</td>
<td>139</td>
<td>499</td>
<td>–</td>
<td>2.58</td>
<td>GPT script</td>
</tr>
<tr>
<td>C</td>
<td>1.73</td>
<td>2+1</td>
<td>$64^3 \times 128 \times 24$</td>
<td>2</td>
<td>6.1</td>
<td>139</td>
<td>499</td>
<td>–</td>
<td>5.16</td>
<td>GPT script</td>
</tr>
<tr>
<td>D</td>
<td>1.74</td>
<td>2+1</td>
<td>$32^3 \times 64 \times 24$</td>
<td>2</td>
<td>6.2</td>
<td>279</td>
<td>534</td>
<td>–</td>
<td>5.12</td>
<td>GPT script</td>
</tr>
<tr>
<td>E</td>
<td>3.50</td>
<td>2+1</td>
<td>$48^3 \times 192 \times 12$, openBC</td>
<td>2</td>
<td>1.4</td>
<td>280</td>
<td>535</td>
<td>–</td>
<td>3.87</td>
<td>GPT script</td>
</tr>
</tbody>
</table>
Further examples

▶ QED corrections to g-2 HVP and tau decays of RBC/UKQCD
▶ Ensemble parameters and g-2 HVP (Tuesday talk C.L.)
▶ g-2 HLbL project of RBC/UKQCD (combined with QLattice)
▶ Scattering studies in scalar field theory (Bruno et al.)
▶ Testing stochastic locality with CP($n$) models (Bruno, Morandi)

Also applied by BNL and Bielefeld groups for ongoing projects.
Teaching
LGT lecture based on interactive GPT notebooks

(link to lecture)

Chapters:

- Chapter 1: path integral formulation of quantum mechanics (Jupyter Notebook, PDF)
- Chapter 2: Markov Chain Monte Carlo (MCMC) methods (Jupyter Notebook, PDF, last update 09.11. 10:14)
- Chapter 3: statistics of continous variables (Jupyter Notebook, PDF, last update 07.11. 21:00)
- Chapter 4: scalar field theory on a lattice (Jupyter Notebook, PDF, last update 02.12. 10:45)
- Chapter 5: symmetries of fundamental field theories (Jupyter Notebook, PDF, last update 21.11. 18:30)
- Chapter 6: gauge theories on a lattice (Jupyter Notebook, PDF, last update 28.11. 16:30)
- Chapter 7: static quark potential and spectrum of pure QCD (Jupyter Notebook, PDF, last update 03.12. 10:11)
- Chapter 8: strong coupling expansion (Jupyter Notebook, PDF, last update 05.12. 21:11)
- Chapter 9: continuum limit and phase transitions (Jupyter Notebook, PDF, last update 13.12. 10:14)
- Chapter 10: fermions on a lattice (Jupyter Notebook, PDF, last update 19.12. 18:04)
- Chapter 11: symmetries of the Wilson action (Jupyter Notebook, PDF, last update 10.01. 09:35)
- Chapter 12: chiral symmetry on a lattice (Jupyter Notebook, PDF, last update 24.01. 10:00)
- Chapter 13: Hybrid Monte Carlo (HMC) (Jupyter Notebook, PDF, last update 28.01. 09:46)
- Chapter 14: Hadron spectroscopy (Jupyter Notebook, PDF, last update 04.02. 09:56)
The machine learning module
Example: train simple feed-forward network

```python
In [ ]:
import gpt as g
grid = g.grid([4, 4, 4], g.double)
rng = g.random("test")

# network and training data
n = g.ml.network.feed_forward([g.ml.layer.nearest_neighbor(grid)] * 2)
training_input = [rng.uniform_real(g.complex(grid)) for i in range(2)]
training_output = [rng.uniform_real(g.complex(grid)) for i in range(2)]

# cost functional
c = n.cost(training_input, training_output)

# train network
W = n.random_weights(rng)
gd = g.algorithms.optimize.gradient_descent
gd(maxiter=4000, eps=1e-4, step=0.2)(c)(W, W)
```
The quantum computing module
Example: create and measure a 5-qubit bell state

```python
import gpt as g
from gpt.qis.gate import *

rng = g.random("qis_test")

# initial state with 5 qubits, stored in double-precision
st = g.qis.backends.dynamic.state(rng, 5, precision=g.double)

g.message("Initial state:
", st)

# prepare Bell-type state
st = (H(0) | CNOT(0,1) | CNOT(0,2) | CNOT(0,3) | CNOT(0,4)) * st

g.message("Bell-type state:
", st)

# measure
st = M() * st

g.message("After single measurement:
", st)
g.message("Classically measured bits:
", st.classical_bit)
```

GPT: 197.943668 s: Initial state:
    : + (1+0j) |00000>
GPT: 197.949198 s: Bell-type state:
    : + (0.7071067811865475+0j) |00000>
    : + (0.7071067811865475+0j) |11111>
GPT: 197.951478 s: After single measurement:
    : + (1+0j) |11111>
GPT: 197.952545 s: Classically measured bits:
    : [1, 1, 1, 1, 1]
How to use GPT?

https://github.com/lehner/gpt

Quick Start

The fastest way to try GPT is to install Docker, start a Jupyter notebook server with the latest GPT version by running

```bash
docker run --rm -p 8888:8888 gptdev/notebook
```

and then open the shown link http://127.0.0.1:8888/?token=<token> in a browser. You should see the tutorials folder pre-installed.

The docker images are automatically generated for each version that passes the CI interface.

CI currently has test coverage of 96%, running on each pushed commit.
Thank you